BERKELEY ▪ FALL 2025

CS182

Deep Neural Networks

An Illustrated Course Reader

Courtesy of Jason Trinh

Derivations • Worked examples • Architecture diagrams

Optimization • Sequence models • Generative learning

Based on the Fall 2025 course taught by

Anant Sahai and Gireeja Ranade

Independently written study synthesis. Original course sources are credited throughout.

Editable LaTeX edition

Courtesy of Jason Trinh

A note from Jason

I put this reader together to bring the CS182 lectures, notes, discussions, and explanations into one place. My aim was to make the course easier to study: to connect the intuition to the mathematics, show the steps behind the derivations, and use diagrams to make the ideas easier to see.

The lecture chapters follow the course, while the linked supplements provide more detailed explanations, worked examples, and additional visualizations. I hope this reader is useful both for learning the material for the first time and for returning to ideas that take a few passes to understand.

Jason Trinh

Acknowledgments

The course material and teaching credited throughout this reader come from Anant Sahai and Gireeja Ranade, together with the credited course contributors. This is an independently assembled study reader, not an official course publication.

Special thanks to Tung Tung Tung Sahur and Tralelero Tralala

Contents

I Lectures in course order 1

0 Introduction and the deep learning viewpoint 2

1 Function approximation, learning paradigms, and optimization 10

2 Optimization: implicit regularization, SGD, and momentum 24

3 Optimization: Adam, locally linear perspective, features 42

4 Optimizers 51

5 Optimizers: insights from induced matrix norms 62

6 muP: maximal update parameterization 73

7 Muon and Newton–Schulz 83

8 Convolutional networks: foundations 95

9 Pooling/Downsampling, Data Augmentation, Normalization Layers 107

10 Normalization Layers, Dropout, Residual Connections 120

11 ResNets, ConvNeXt, U-Nets 134

12 Graph Neural Networks 150

13 GNN Pooling and Intro to RNNs 164

14 Recurrent Neural Networks and Self-Supervision 175

15 Self-Supervision and State-Space Models 188

16 State-Space Models (S4 and Mamba) 202

17 Attention Mechanisms 214

18 Attention and Transformers 225

19 Advanced Transformer Architectures 238

20 In-Context Learning, Prompting, and Fine-Tuning 256

21 Fine-Tuning and Meta-Learning 266

22 Meta-Learning and Generative Models Intro 275

23 Generative Models and Post-Training 290

24 Post-Training and Test-Time Compute 301

26 RL Post-Training and Diffusion Models 313

27 Diffusion Models 325

II Detailed supplements and worked explanations 341

A Learning functions and differentiating programs 342

B Least squares, spectral dynamics, and regularization 352

C Stochastic gradients, momentum, Adam, and weight decay 368

D Optimizers as choices of geometry 373

E Initialization, maximal updates, and Muon 377

F Convolution, locality, and weight sharing 382

G Normalization, residual paths, and spatial reconstruction 388

H Graph neural networks and differentiable pooling 395

I Recurrent networks and self-supervised prediction 404

J State-space models: recurrence, convolution, and selective memory **412**

K Attention as differentiable retrieval **416**

L Transformers, position, and efficient inference **421**

M Adapting a pretrained model **429**

N Learning to adapt and remembering earlier tasks **436**

O Probability, information, and gradients through randomness **440**

P Variational autoencoders and latent-variable models **451**

Q Decoding and computation at inference time **457**

R Reward optimization, RLHF, and preference learning **463**

S Diffusion: learning to reverse a known corruption process **469**

T Discussion companion: thirteen sets of practice **476**

U Reference, corrections, and a study route **490**

V Source register **500**

Part I

Lectures in course order

Lecture 0

Introduction and the deep learning viewpoint

Course sources: 2025-08-28 transcript.

Today's Agenda

1. Introductions
2. Administrative Details
3. Introduction to Deep Learning

Course Philosophy

A core tenet of this class, inspired by the late Professor Patrick Winston of MIT, is that *"All great ideas are simple. Good ideas are complicated."* Our goal is to distill complex topics into their simplest, most elegant forms. To appreciate this simplicity, a strong foundation in powerful mathematical tools is essential. Just as the Singular Value Decomposition (SVD) simplifies complex matrices by projecting them into the right coordinates, we will use the language of mathematics to find clarity and beauty in deep learning.

0.1 Course Administration (Fall 2025)

These announcements describe the original Fall 2025 lecture context.

Instructors

- **Prof. Gireeja Ranade:** Course instructor at UC Berkeley.
- **Prof. Anant Sahai:** Professor at UC Berkeley, has taught this course multiple times.
- **Office Hours:** Right after class in 400 Cory Hall.

Course Staff

- **GSI:** Sultan Daniels (3rd year PhD student)
- **uGSIs:** Patrick (4th year EECS & Applied Math), Alex (3rd year EECS & Math)

Resources

- **Website:** `eecs182.org`. All official policies are documented here.
- **Ed Discussion:** The primary mode of communication. All posts are required reading. It is the fastest way to get a response from faculty and staff. For private matters, use private posts on Ed.
- **Textbook (Recommended):** *Understanding Deep Learning* by Simon Prince. This is a good, recent textbook, but the course will cover material not in the book, including recent research papers. It is a reference, not a requirement.

Enrollment

The class may expand, but this is not under the instructors' control. To prioritize engaged and committed students, we have the following policy:

- A **Homework 0** will be due next week.
- Students enrolled in the class who **do not submit Homework 0 will be dropped**.
- Students on the waitlist **must complete and submit Homework 0** to be considered for a spot in the class.

Sections and Getting Help

- **Discussion Sections:** Wednesdays at 3-4 PM and 4-5 PM. Sections are **required** and will cover important material. They start Wednesday, September 3rd. You may attend either section.
- **Getting Help:**
    1. Instructor Office Hours (after class)
    2. GSI Office Hours (TBD)
    3. Discussion Sections
    4. Ed Discussion Forum
    5. **Each other:** Form study groups! This is a fast-paced course, and working together is critical for success. Set up regular meeting times to work on homework and hold each other accountable.

Prerequisites

This course assumes **mastery** of the following subjects. This background material will not be re-taught or supported in office hours.

- **Machine Learning:** At the level of CS 189 or Stat 154.
- **Linear Algebra & Optimization:** At the level of EECS 127/227A. Optimization is the workhorse of modern deep learning.
- **Probability:** At the level of EECS 126, Data 140, or Stat 134. Probability provides the language for generalization and the intuition behind many algorithms.

This is not a "cookbook" deep learning class. We aim for a deep, fundamental understanding of the material.

Grading and Policies

There will be **no curve** in this class. Grading bins are published on the course website.

Grade Breakdown

- **Final Exam: 60%**
- **Project: 25%**
- **Participation: 15%**
- **Homework: 0%** (Nominally)

Gating Effect: You must pass the final exam to pass the class, and you must pass the project to pass the class. This is to ensure individual understanding and prevent free-riding on group projects.

Homework Policies

- **Submission:** Join GradeScope. Homeworks involve both written proofs/derivations and coding components. Submit a single PDF containing scans of your written work and a PDF of your code notebook.
- **Deadline:** Homeworks are due weekly on Friday nights. However, the GradeScope deadline is set to the end of the semester to provide automatic extensions if needed. You are responsible for keeping up.
- **Purpose:** Homework is the best way to learn the material. While worth 0%, we may look at a random subset of your homeworks if your final grade is near a boundary. Using

an LLM to do your homework is like driving around the block instead of running; you miss the entire point, which is the workout (learning).

Project

- **Groups:** Groups of four are required.
- **Topics:** We will provide a few open-ended project options that build on published papers and allow you to explore near the research frontier. Graduate students from outside EECS may be permitted to propose projects aligned with their own research.
- **Compute:** We are working to secure some cloud compute resources, but they will be limited. You may need to use your own compute for ambitious projects.
- **Peer Review:** There will be a mandatory individual peer review component. Your project grade will be a combination of your group's technical submission and your individual performance as a peer reviewer. Failure to complete peer reviews will result in a zero for the project.

Participation and Citizenship (15%)

Everyone starts with a failing grade (0%) in participation. To earn credit, you must contribute to the class community.

- **Ordinary Participation:** Achieved by:
  1. Attending and actively participating in weekly discussion sections.
  2. Being actively helpful on the Ed Discussion forum.
- **Special Contributions:** You are expected to complete **at least four** of these throughout the semester. These are tasks that engage your learning and improve the class for everyone. Examples include:
  1. **LLM for Homework (max 1):** Use an LLM (e.g., Gemini) to solve a homework problem. Document the entire interaction trace and provide an annotated report critiquing the LLM's process and solution.
  2. **LLM for Code (capped):** Similar to the above, but for a coding problem using tools like Co-pilot, Cursor, etc.
  3. **Code Cleanup (capped):** Take a homework coding problem and refactor it to reflect best practices in Python and machine learning coding style.
  4. **Problem Modernization:** Update an older homework problem to reflect recent advances and concepts taught in this semester's class.

*The Nature of Deep Learning Deep learning as a field is young. Our understanding is much better than it was a decade ago, but we are still in the infancy of this science. You will often encounter concepts that are not fully understood, which can be uncomfortable.

What is Deep Learning?

A neural network is fundamentally a **simulated, typically analog circuit**.

- It processes information in a massively parallel fashion using continuous values (represented by floating point numbers).
- Its very many parameters are not handcrafted but are set using an **optimization algorithm** (like gradient descent) driven by data.
- It is "deep" because these circuits have many layers; data must pass through many circuit elements to get from input to output.

The goal is for this learned circuit to approximately capture relevant patterns in the training data, so it can be used to solve problems involving new, unseen data.

The "Drunk Man" Principle

A recurring theme in this course can be explained by a joke: *A person sees a drunk man searching for his keys under a streetlamp. "Did you lose your keys here?" the person asks. "No," the drunk man replies, "I lost them over there in the dark." "Then why are you looking here?" "Because this is where the light is."*

Many times in deep learning, we want to solve problem X, but we don't know how. Instead, we solve problem Y, because Y is something we **do** understand and have tools for. It's better to do something with some chance of success than nothing with a guaranteed chance of failure.

The Pile of Linear Algebra

As immortalized by the XKCD comic, much of machine learning involves "stirring a big pile of linear algebra." We lean on linear algebra for several key reasons:

1. **We understand it.** It provides a solid foundation with connections to geometry and intuition.
2. **Computers are good at it.** Linear algebra operations map naturally to parallel hardware like GPUs.
3. It provides a set of powerful building blocks for more complex systems.

Our approach will be to ground the "soup" of deep learning in the elegant and beautiful fundamentals of mathematics you have already learned.

Deep Learning as Modern Alchemy

For most of human history, progress was made empirically, without full theoretical understanding. People built cathedrals before Newton formalized $F = ma$. Alchemists, in their quest to turn lead into gold, developed the experimental apparatus and foundational knowledge that led to modern chemistry.

Deep learning today is in a similar phase. We can build amazing things, but our theoretical understanding often lags behind our empirical results. It's okay to not understand everything fully. This is an opportunity to experience science in a state of formation, much like our ancestors did.

A Brief History of Neural Networks

- **1950s-60s (First Golden Age):** The Perceptron and early data-driven methods like LMS (Least Mean Squares).
- **1969 ("AI Winter"):** The book *Perceptrons* by Minsky and Papert highlighted the limitations of single-layer networks, contributing to a decline in funding and interest.
- **1980s (Second Golden Age):** The popularization of backpropagation for training multi-layer networks led to a resurgence of interest.
- **1990s (Second Winter):** Neural networks were again superseded by methods with stronger mathematical guarantees and better performance at the time (e.g., Support Vector Machines).
- **2000s-2010s (The Rise of GPUs):** The demand for video games drove massive improvements in parallel computing hardware (GPUs). Researchers began using GPUs to train much larger neural networks much faster.
- **2012-Present (Modern Golden Age):** The AlexNet paper demonstrated a massive leap in performance on image classification, decisively beating all previous methods and kicking off the modern deep learning revolution.

Our course will focus primarily on the post-2012 era.

Function Approximation with Neural Networks

Deeper explanation: Constructing a prescribed piecewise-linear function (p. 342)

Let's build a neural network from first principles. Our goal is to approximate a function $f : \mathbb{R} \to \mathbb{R}$. We are given training data as pairs (x_i, y_i), where $y_i \approx f(x_i)$.

Piecewise Approximations

A simple way to approximate a function is with a piecewise function.

- **Piecewise Constant:** Like a Riemann sum, this approximates the function with horizontal steps.
- **Piecewise Linear:** Like the trapezoid rule for integration, this approximates the function with connected line segments. For the same number of "pieces" or "kinks," this is generally a better approximation.

Our goal is to create a neural network that represents a trainable piecewise linear function.

Parameterizing a Piecewise Linear Function

To specify a piecewise linear function, we need to know:

1. The locations of the "kinks" or "elbows" on the x-axis.
2. The slopes of the linear segments between the kinks.
3. An overall vertical offset or intercept (the height at one point).

How can we represent this in a way that is friendly to computation, particularly parallel, linear-algebraic operations? A good approach is to represent the function as a weighted sum of simpler basis functions:

$$N_\theta(x) = b + \sum_{i=1}^{d} w_i g_i(x)$$

This structure is "computer-friendly" because each $g_i(x)$ can be computed in parallel.

The ReLU Basis Function

A powerful choice for the basis function $g_i(x)$ is a function with a single "elbow". By adding many of these functions together with different weights, locations, and slopes, we can construct any arbitrary piecewise linear function.

This elbow function is the **Rectified Linear Unit (ReLU)**, also known as an activation function.

$$\text{ReLU}(z) = \max(0, z)$$

To control the location and slope of the elbow, we make the argument to the ReLU an affine function of the input x:

$$g_i(x) = \text{ReLU}(w_i' x + b_i')$$

Here, w_i' controls the slope after the kink, and the location of the kink is at $x = -b_i'/w_i'$.

A One-Hidden-Layer Neural Network

Putting it all together, we can represent our piecewise linear approximator as:

$$N_\theta(x) = b^2 + \sum_{i=1}^{d} w_i^2 \cdot \text{ReLU}(w_i^1 x + b_i^1)$$

This is the mathematical form of a **one-hidden-layer multilayer perceptron (MLP)** for a scalar input and scalar output. The parameters θ consist of all the weights and biases $\{W^1, \vec{b}^1, W^2, b^2\}$.

In vectorized form, we can write this as:

$$N_\theta(x) = b^2 + W^2 \cdot \text{ReLU}(W^1 x + \vec{b}^1)$$

where W^1 is a matrix (here, a column vector), $\vec{b}^1$ is a vector, W^2 is a matrix (here, a row vector), and b^2 is a scalar. The ReLU function is applied element-wise to the vector argument.

This structure can be visualized as a network diagram:

- **Input Layer:** A single node for the input x.
- **First Hidden Layer:** A set of d nodes. For each node i, the computation $h_i = \text{ReLU}(w_i^1 x + b_i^1)$ is performed. The values h_i are called **activations**.
- **Output Layer:** The activations from the hidden layer are combined in a weighted sum, $b^2 + \sum_i w_i^2 h_i$, to produce the final output.

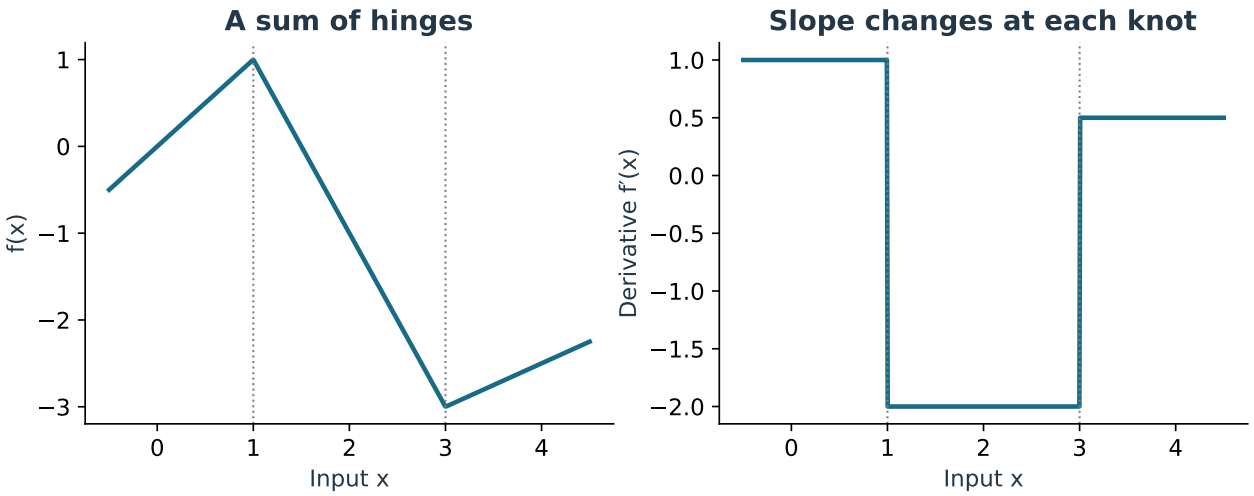


Figure 1: A nonlinear curve assembled from a linear term and two ReLU hinges. The derivative makes each slope change visible.

Lecture 1

Function approximation, learning paradigms, and optimization

Course sources: lecture_01_sahai; lecture_01_ranade; 2025-09-02 transcript.

1.1 Recap: Function Approximation with Neural Networks

Deeper explanation: Constructing a prescribed piecewise-linear function (p. 342)

The Problem: Learning a Function from Samples

We begin by considering the problem of function approximation. Imagine there is a true, unknown function $F : \mathbb{R} \to \mathbb{R}$. We do not have access to F directly. Instead, we are given a set of training samples, which are pairs of inputs and outputs (x_i, y_i), where $y_i = F(x_i)$. The goal of machine learning, in this context, is to use these samples to learn an approximation of the true function F.

This can be visualized by plotting the true function (a smooth curve) and then marking the training points (x_i, y_i) that lie on this curve. Our task is to find a function that passes through or near these points, thereby approximating the original unknown function.

Piecewise Linear Approximation

A simple yet powerful way to approximate a continuous function is with a piecewise linear function. Such a function is defined by a series of connected line segments.

A piecewise linear function can be specified by three components:

1. The locations of the "elbows" or kinks.
2. The slopes of the linear segments.
3. An overall vertical bias or offset.

To represent this in a way that is computationally friendly (i.e., "linear-algebra friendly"), we can construct it as a sum of simpler functions. We can use a basis function $g_\theta(x)$ that has a single kink. A canonical choice for this is the Rectified Linear Unit (ReLU) function, which is an affine function followed by a $\max(0, \cdot)$ operation.

Specifically, we can define our basis function as:

$$g_{\theta_i}(x) = \max(0, w_i' x + b_i')$$

By summing up multiple such functions, each with its own parameters (w_i', b_i') and a corresponding weight w_i, and adding an overall bias b, we can create a piecewise linear function with as many kinks as we need. The overall approximation $f(x)$ takes the form:

$$f(x) = b + \sum_{i=1}^{d} w_i g_{\theta_i}(x) = b + \sum_{i=1}^{d} w_i \max(0, w_i' x + b_i')$$

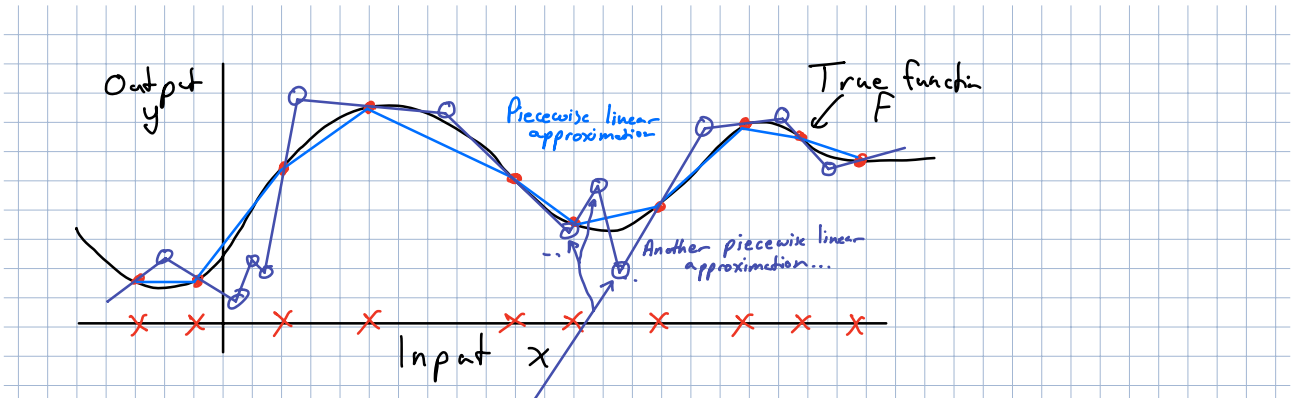


Figure 1.1: Competing piecewise-linear fits. Different functions can agree on the observations. The breakpoints specify where a ReLU-based fit changes slope; interpolation alone does not determine behavior between or beyond the samples. Original annotated source excerpt; Anant Sahai, faculty handout 01 sahai, p. 2.

Neural Network Representation

The piecewise linear approximation model can be directly translated into the language and structure of a simple neural network.

The ReLU Activation Function

The non-linear function $\text{ReLU}(z) = \max(0, z)$ is a cornerstone of modern deep learning. It is known as the **Rectified Linear Unit** and is a type of **activation function**. These non-linearities are what allow neural networks to approximate complex functions beyond simple linear relationships.

Mathematical and Block Diagram Formulation

We can express our function approximation in a more compact, vectorized form. Let's rename the parameters for clarity, associating them with network layers.

- Let the first layer's weights and biases be W^1 (a $d \times 1$ matrix, where each row is a w_i') and $\vec{b}^1$ (a $d \times 1$ vector of b_i').
- Let the second layer's weights and bias be W^2 (a $1 \times d$ matrix of w_i) and b^2 (the scalar bias b).

For a scalar input x, the function becomes:

$$f(x) = b^2 + W^2 \text{ReLU}(W^1 x + \vec{b}^1)$$

Here, the ReLU function is applied element-wise to the vector argument.

This structure can be visualized as a network diagram with distinct layers:

- **Input Layer:** Takes the input x.
- **First Hidden Layer:** Consists of d parallel units. Each unit i computes $h_i = \text{ReLU}(w_i' x + b_i')$. The outputs of this layer, the vector $\vec{h}$, are called the hidden **activations**.
- **Output Layer:** This is typically an affine layer that combines the hidden activations to produce the final output. In this case, it computes $\sum_i w_i h_i + b^2$.

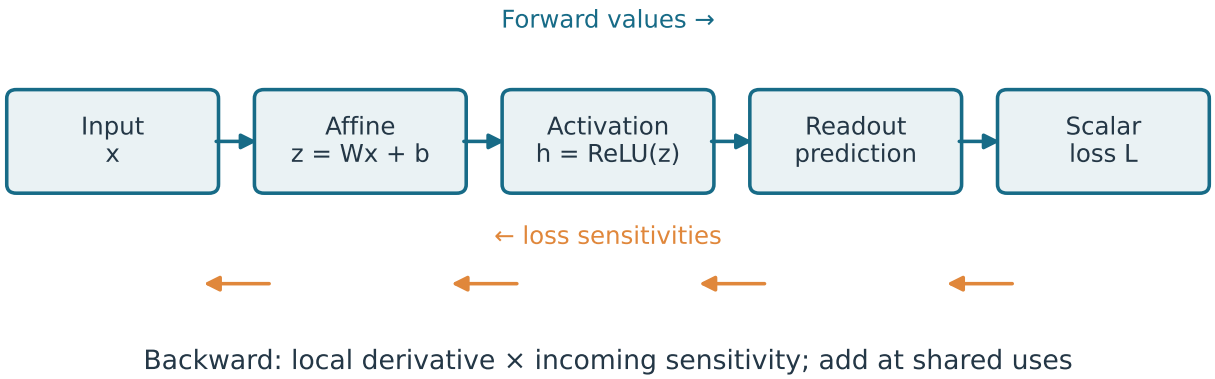


Figure 1.2: Forward values and backward sensitivities in a two-layer network. Shared parameters accumulate gradients from all their uses.

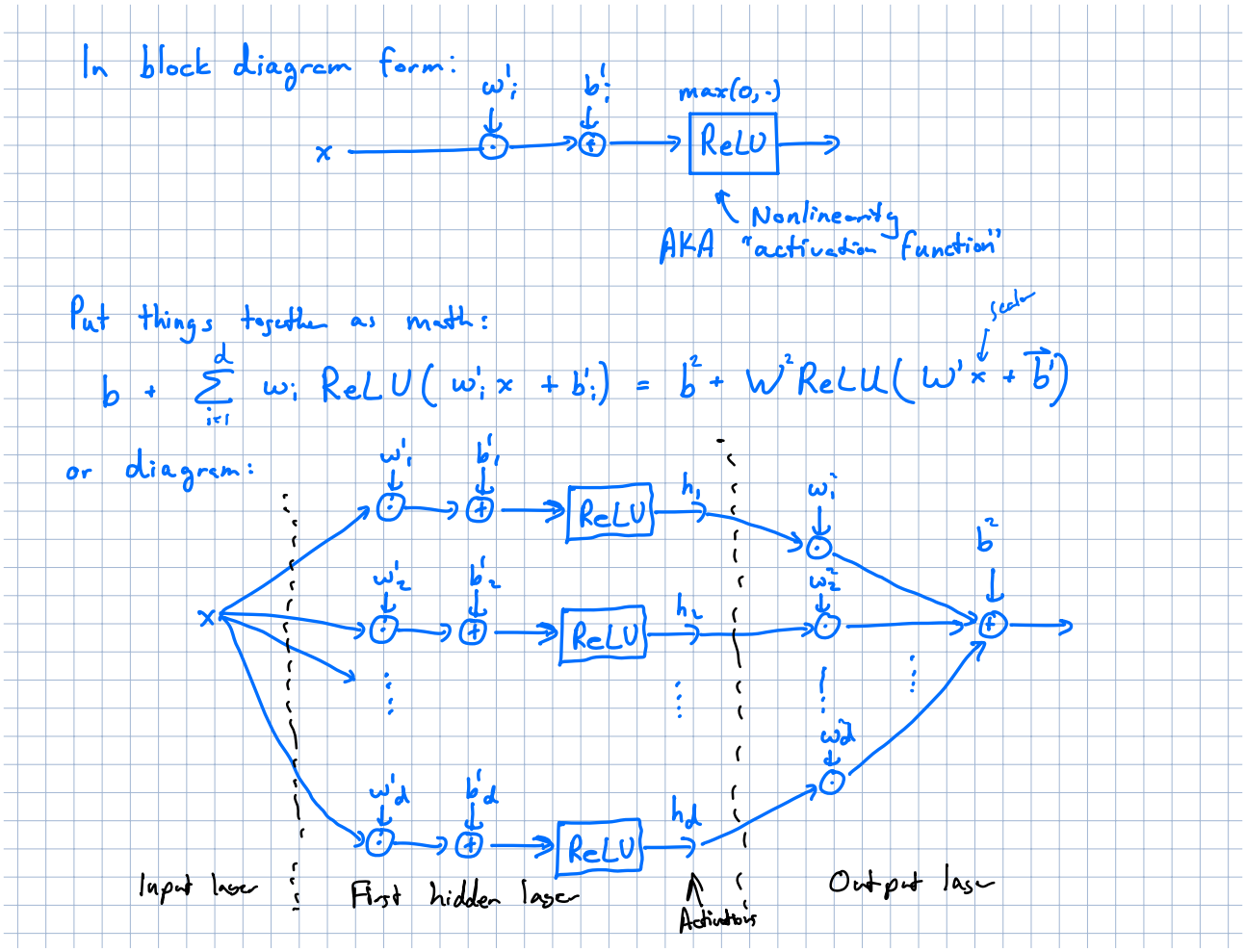


Figure 1.3: A sum of hinges as an actual network. Read each branch as a shifted and scaled hinge, then add the branch outputs. The diagram turns the piecewise-linear formula into parallel computation paths. Original annotated source excerpt; Anant Sahai, faculty handout 01 sahai, p. 3.

Worked derivation: constructing a piecewise-linear function

Lecture basis: faculty handout 01_sahai, pp. 1–3. The steps below expand the lecture argument.

Step 1: specify what must be matched. Let the ordered knots be $a_1 < \cdots < a_K$. A continuous piecewise-linear function has slope m_0 before the first knot and slope m_j on the interval following a_j. Choose a reference point $x_* < a_1$ with value f_*. Define the slope jumps $c_j = m_j - m_{j-1}$.

Step 2: build one change of slope per hidden unit. For $h_j(x) = \max(0, x - a_j)$, away from the knot,

$$h_j'(x) = \begin{cases} 0 & x < a_j, \\ 1 & x > a_j. \end{cases}$$

Thus $c_j h_j$ contributes exactly c_j to the slope after a_j and contributes nothing before it. Set

$$g(x) = f_* + m_0(x - x_*) + \sum_{j=1}^{K} c_j \max(0, x - a_j).$$

Step 3: check every interval, not just the sample points. For $a_k < x < a_{k+1}$, precisely the first k hinges are active:

$$\begin{aligned} g'(x) &= m_0 + \sum_{j=1}^{k} (m_j - m_{j-1}) \\ &= m_0 + (m_1 - m_0) + \cdots + (m_k - m_{k-1}) \\ &= m_k. \end{aligned}$$

Both functions are continuous, have identical slopes on every interval, and agree at x_*. Their difference is therefore zero everywhere. If an architecture has no explicit linear skip, use $x = \mathrm{ReLU}(x) - \mathrm{ReLU}(-x)$ to represent the linear term with two additional units.

Example. Slopes $1, -2, 0$ with knots $0, 2$ give $g(x) = x - 3\,\mathrm{ReLU}(x) + 2\,\mathrm{ReLU}(x - 2)$. Checking the three intervals gives x, $-2x$, and -4, respectively. The hinge coefficients are slope *changes*, not the final slopes.

1.2 Machine Learning vs. Deep Learning Paradigms

Deeper explanation: What a learning curve can diagnose (p. 347)

Classical Machine Learning

Traditional machine learning is often categorized into two main paradigms.

Supervised Learning

- **Data:** Consists of pairs $(\vec{x}, y)$, where $\vec{x}$ is the input/covariate and y is the output/label.
- **Regression:** The task of predicting a continuous, real-valued output y.
- **Classification:** The task of predicting a discrete label y (e.g., binary or multi-class).
- **General Paradigm:** The common approach is to train a specific model for a specific, well-defined problem using a specific dataset.

Unsupervised Learning

- **Data:** Consists of inputs $\vec{x}$ only, with no corresponding labels y. The goal is to find implicit structure in the data.
- **PCA-style (Dimensionality Reduction):** Finding the directions of maximum variation in the data (e.g., Principal Component Analysis).
- **Clustering:** Grouping similar data points together (e.g., k-means).
- **Density Estimation:** Learning the underlying probability distribution from which the data samples $\vec{x}_1, \vec{x}_2, ...$ are drawn.

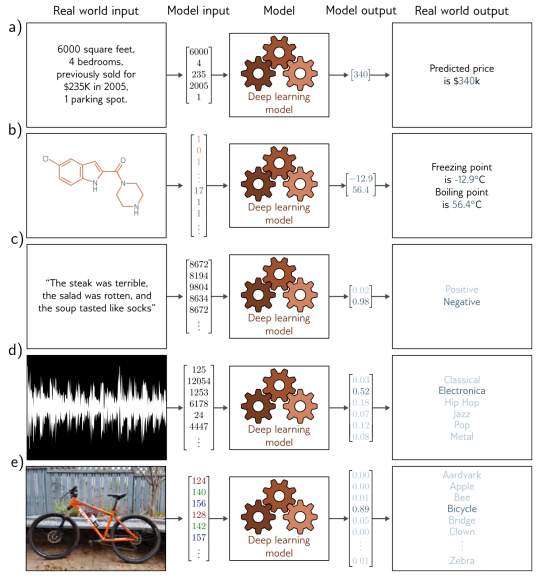
Deep Learning Paradigms

Deep learning extends these classical paradigms and introduces new ones.

Supervised and Unsupervised Deep Learning

Deep networks can be used to solve classical regression and classification problems. They also enable more complex tasks:

- **Localized Annotation (Semantic Segmentation):** A supervised task where the model assigns a label to every pixel in an image, effectively outlining objects (e.g., identifying all pixels corresponding to a cat, a car, or a pedestrian).
- **Learned Embeddings:** An unsupervised task analogous to PCA. The goal is to learn a mapping from complex data (like words or images) into a lower-dimensional vector space, such that similar items are close to each other in this space.
- **Generative Models:** An unsupervised task analogous to density estimation. The goal is to learn a model of the data distribution so that it can generate new, synthetic data samples that resemble the training data (e.g., generating realistic images of cats that have never existed).



Sample supervised learning tasks.
Prince, 2025.

Generative models.



Figure 1.5 Generative models for images. Left: two images were generated from a model trained on pictures of cats. These are not real cats, but samples from a probability model. Right: two images generated from a model trained on images of buildings. Adapted from Karras et al. (2020b).

Figure 1.4: What the input and output of a deep model look like. The examples distinguish prediction from generation by the information supplied to the model and the type of output it produces. Published example panels and their original credits are retained from the handout. Original annotated source excerpt; Gireeja Ranade, faculty handout 01 ranade, p. 4.

Foundation Models

This represents a significant paradigm shift. Instead of training a model from scratch for a specific task, the idea is to:

1. Pre-train a very large model on a massive, general-purpose dataset.
2. This pre-trained model, known as a **foundation model**, learns general representations of the data.
3. This model is then **fine-tuned** on a smaller, task-specific dataset to adapt it to a particular application.

This approach moves away from the ”one model, one task” philosophy towards leveraging large-scale, pre-existing knowledge.

1.3 Optimization for Deep Learning

Empirical Risk Minimization (ERM)

Deeper explanation: Convexity: chords, tangents, and global optimality (p. 355)

Deeper explanation: Lipschitz continuity bounds changes in function value (p. 357)

Deeper explanation: L-smoothness bounds changes in the gradient (p. 357)

Given a dataset of training examples $\{(\vec{x}_i, y_i)\}_{i=1}^n$, our goal is to find the parameters θ of a model $f_\theta(\vec{x})$ that best predict the outputs. We do this by minimizing a loss function $\mathcal{L}$, which measures the discrepancy between the predicted output $f_\theta(\vec{x}_i)$ and the true output y_i.

The standard approach is **Empirical Risk Minimization (ERM)**, where we minimize the average loss over the training data:

$$\hat{\theta} = \arg\min_\theta \frac{1}{n} \sum_{i=1}^n \mathcal{L}(y_i, f_\theta(\vec{x}_i))$$

The True Goal and Key Assumption

The ultimate goal is not just to perform well on the training data, but to generalize to new, unseen data. To formalize this, we make a key assumption: the data pairs $(\vec{x}, y)$ are drawn independently and identically distributed (i.i.d.) from an unknown underlying probability distribution $P(\vec{x}, y)$.

The true objective is to minimize the expected loss, or "risk," over this entire distribution:

$$\theta^* = \arg\min_\theta \mathbb{E}_{(\vec{x},y)\sim P(\vec{x},y)}[\mathcal{L}(y, f_\theta(\vec{x}))]$$

Challenges in Optimization

Deeper explanation: Strong convexity supplies a positive lower curvature (p. 359)

Deeper explanation: Counterexamples keep the assumptions separate (p. 360)

Deeper explanation: Without convexity: descent can certify stationarity only (p. 364)

Deeper explanation: Applying the constants to losses and neural parameterizations (p. 365)

Several challenges arise when trying to achieve this goal.

Challenge 1: We don't know the true distribution $P(\vec{x}, y)$.

Solution: We optimize an empirical training objective, use a validation split for model selection, and retain a held-out test set for final evaluation. The test set estimates generalization; it does not supply the training objective.

Challenge 2: The loss function may not be amenable to optimization.
For instance, it might be non-differentiable.
Solution: We choose a surrogate loss function that is smooth, continuous, and works well with our chosen optimizer (e.g., gradient descent).

Challenge 3: The model performs well on training data but poorly on test data.
This phenomenon is known as **overfitting**. The model has memorized the training examples, including their noise, rather than learning the underlying pattern.
Solution: Regularization.

Regularization and Hyperparameters

Regularization introduces a penalty term to the loss function that discourages overly complex models. A classic example is **Ridge Regression** (L2 regularization), which penalizes large parameter values:

$$\arg\min_{\vec{w}} \|X\vec{w} - \vec{y}\|_2^2 + \lambda\|\vec{w}\|_2^2$$

The term λ is a **hyperparameter** that controls the strength of the regularization.

This introduces a new problem: **hyperparameter search**. Finding the optimal value for λ (and other hyperparameters like learning rate, network architecture, etc.) is critical but can be computationally expensive. Common strategies include:

- **Grid Search / Random Search:** Trying many different hyperparameter values and evaluating their performance on a separate **validation set**. Random search is often more effective than a fixed grid.
- **Couple Hyperparameters:** Scale related hyperparameters together to reduce the dimensionality of the search space.
- **Use What Others Did:** Start with hyperparameter values that have been reported to work well in similar research papers or applications.

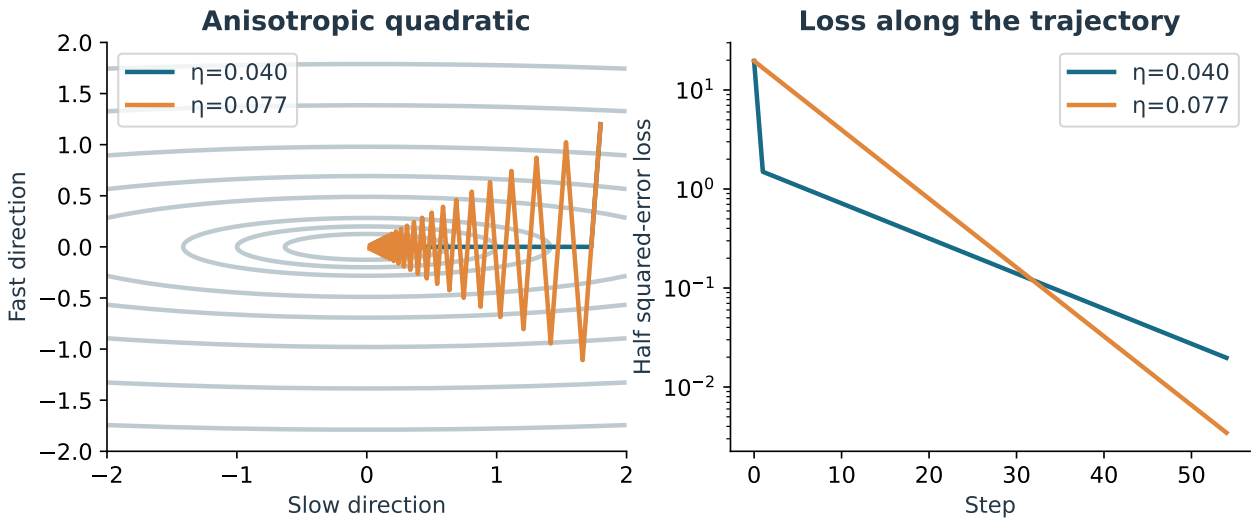


Figure 1.5: Gradient descent on an elongated quadratic: one learning rate must stabilize both directions. The right panel shows the loss over iterations for two stable step sizes.

1.4 Gradient Descent

Deeper explanation: A numerical backpropagation and finite-difference check (p. 342)

Deeper explanation: Read a matrix product as a shape diagram (p. 343)

Deeper explanation: A complete minibatch derivation (p. 343)

Deeper explanation: Numerical stability: preserve the mathematics in finite precision (p. 348)

The Algorithm

Gradient descent is the workhorse optimization algorithm for deep learning. It is an iterative method for finding a local minimum of a differentiable function. Given a loss function $\mathcal{L}(\theta)$ (here, we abbreviate the dependency on training data), the update rule is:

1. Start with an initial guess for the parameters, θ_0.
2. For each step $t = 0, 1, 2, \ldots$, update the parameters by moving in the direction of the negative gradient:
$$\vec{\theta}_{t+1} = \vec{\theta}_t - \eta \nabla_\theta \mathcal{L}(\vec{\theta}_t)$$
where η is the **learning rate**, a hyperparameter that controls the step size.

Analysis for the Least Squares Problem

Let's analyze gradient descent for the simple case of least squares, where the loss is $\mathcal{L}(\vec{w}) = \|X\vec{w} - \vec{y}\|_2^2$. The optimal solution is $\vec{w}^* = (X^T X)^{-1} X^T \vec{y}$.

The gradient of the loss function is:

$$\nabla_{\vec{w}} \mathcal{L}(\vec{w}) = 2X^T(X\vec{w} - \vec{y})$$

The gradient descent update rule becomes:

$$\begin{aligned}\vec{w}_{t+1} &= \vec{w}_t - \eta\left(2X^T(X\vec{w}_t - \vec{y})\right) \\ &= (I - 2\eta X^T X)\vec{w}_t + 2\eta X^T \vec{y}\end{aligned}$$

Convergence and Optimal Learning Rate

Deeper explanation: A safe gradient step comes from the upper model (p. 361)

Deeper explanation: Smooth convex gradient descent has a 1/k gap bound (p. 362)

To analyze convergence, we look at the error vector $\vec{e}_t = \vec{w}_t - \vec{w}^*$. The update rule for the error is:

$$\vec{w}_{t+1} - \vec{w}^* = (I - 2\eta X^T X)(\vec{w}_t - \vec{w}^*)$$

The algorithm converges if the error vector shrinks to zero. This happens if all eigenvalues of the matrix $(I - 2\eta X^T X)$ have an absolute value less than 1.

Let λ_i be an eigenvalue of the symmetric positive semi-definite matrix $X^T X$. The corresponding eigenvalue of the update matrix is $1 - 2\eta\lambda_i$. We require:

$$|1 - 2\eta\lambda_i| < 1 \quad \text{for all } i$$

This inequality must hold for both the smallest eigenvalue, $\lambda_{\min}$, and the largest, $\lambda_{\max}$. The tightest constraint comes from $\lambda_{\max}$:

$$-1 < 1 - 2\eta\lambda_{\max} \implies 2\eta\lambda_{\max} < 2 \implies \eta < \frac{1}{\lambda_{\max}}$$

For the **fastest convergence**, we want to make the largest absolute eigenvalue as small as possible. This occurs when the bounds for $\lambda_{\min}$ and $\lambda_{\max}$ are balanced:

$$|1 - 2\eta\lambda_{\max}| = |1 - 2\eta\lambda_{\min}|$$

Solving this gives the optimal learning rate:

$$\eta^* = \frac{1}{\lambda_{\min} + \lambda_{\max}}$$

The rate of convergence is then determined by $\frac{\lambda_{\max} - \lambda_{\min}}{\lambda_{\max} + \lambda_{\min}}$, which is closely related to the **condition number** $\kappa = \frac{\lambda_{\max}}{\lambda_{\min}}$ of the matrix $X^T X$. A high condition number leads to slow convergence.

Worked derivation: least-squares GD, mode by mode

Lecture basis: faculty handout 01_ranade, pp. 8–10. The steps below expand the lecture argument.

Step 1: differentiate with the loss convention visible. Use $L(w) = \|Xw - y\|_2^2$, with $X \in \mathbb{R}^{n \times d}$. No factor $1/2$ or $1/n$ is included. For a perturbation dw,

$$\begin{aligned} dL &= d\big[(Xw - y)^\top (Xw - y)\big] \\ &= (X\,dw)^\top (Xw - y) + (Xw - y)^\top X\,dw \\ &= 2\big[X^\top (Xw - y)\big]^\top dw, \\ \nabla L(w) &= 2X^\top (Xw - y). \end{aligned}$$

Step 2: subtract a fixed point. A least-squares solution obeys $X^\top X w_* = X^\top y$, even when $Xw_* = y$ is impossible. Hence

$$\begin{aligned} w_{t+1} &= w_t - 2\eta X^\top (Xw_t - y), \\ e_{t+1} := w_{t+1} - w_* &= e_t - 2\eta X^\top X e_t \\ &= (I - 2\eta X^\top X) e_t. \end{aligned}$$

Step 3: diagonalize and solve the scalar recurrences. Write $X^\top X = V \Lambda V^\top$ and $z_t = V^\top e_t$. Orthogonality gives

$$\begin{aligned} z_{t+1} &= V^\top (I - 2\eta V \Lambda V^\top) V z_t \\ &= (I - 2\eta \Lambda) z_t, \\ z_{t,i} &= (1 - 2\eta\lambda_i)^t z_{0,i}. \end{aligned}$$

Every positive-eigenvalue component converges exactly when $|1 - 2\eta\lambda_i| < 1$, or $0 < \eta < 1/\lambda_i$. Simultaneous convergence therefore requires $0 < \eta < 1/\lambda_{\max}$. Zero modes remain unchanged; convergence to a specified w_* additionally requires matching its null-space component.

Step 4: balance the slowest positive modes. On the positive spectrum $[a, b]$ with $a > 0$, the worst error multiplier is

$$q(\eta) = \max\{|1 - 2\eta a|, |1 - 2\eta b|\}.$$

The maximum is at an endpoint because the absolute value of an affine function is convex. Before the two endpoint magnitudes balance, moving toward their intersection reduces the larger one. At the optimum they have opposite signs:

$$\begin{aligned} 1 - 2\eta_* a &= -(1 - 2\eta_* b), \\ 2 &= 2\eta_* (a + b), \\ \eta_* = \frac{1}{a + b}, &\qquad q_* = \frac{b - a}{b + a} = \frac{\kappa - 1}{\kappa + 1}. \end{aligned}$$

Here $\kappa = b/a$ is the condition number on the active subspace. Parameter error contracts by at most q_*^t; excess squared loss contracts by at most q_*^{2t}. With loss $\frac{1}{2}\|Xw - y\|^2$, the optimal learning rate doubles because its gradient is half as large.

1.5 Regularization and Gradient Descent

Deeper explanation: Noise, bias, and variance along one singular direction (p. 353)

Ridge Regression and SVD

The solution to Ridge Regression is given by:

$$\vec{w}^* = (X^T X + \lambda I)^{-1} X^T \vec{y}$$

An alternative form, sometimes called the kernel form, is $\vec{w}^* = X^T (XX^T + \lambda I)^{-1} \vec{y}$. This can be more efficient if the number of data points is smaller than the number of features.

To gain insight, we can use the Singular Value Decomposition (SVD) of the data matrix $X = U\Sigma V^T$. Substituting this into the Ridge solution and simplifying, we get:

$$\vec{w}^* = V(\Sigma^T \Sigma + \lambda I)^{-1} \Sigma^T U^T \vec{y}$$

This can be expressed as a sum over the singular components:

$$\vec{w}^* = \sum_{i=1}^{k} \vec{v}_i \frac{\sigma_i}{\sigma_i^2 + \lambda} (\vec{u}_i^T \vec{y})$$

where $\vec{v}_i$ and $\vec{u}_i$ are the right and left singular vectors and σ_i are the singular values.

This form reveals how Ridge regularization works:

- For a large singular value $\sigma_i \gg \sqrt{\lambda}$, the term $\frac{\sigma_i}{\sigma_i^2 + \lambda} \approx \frac{1}{\sigma_i}$. The component is kept.
- For a small singular value $\sigma_i \ll \sqrt{\lambda}$, the term $\frac{\sigma_i}{\sigma_i^2 + \lambda} \approx \frac{\sigma_i}{\lambda}$, which is small. The component is heavily shrunk towards zero.

Regularization effectively filters out the directions in the data corresponding to small singular values, which are often associated with noise.

This connects to the idea of the **implicit regularization of gradient descent**. As will be discussed later, running gradient descent for a limited number of steps (early stopping) has an effect similar to explicit L2 regularization, where the number of steps plays a role analogous to the inverse of the regularization parameter λ.

Worked derivation: ridge, the dual form, and spectral shrinkage

Lecture basis: faculty handout 01_ranade, pp. 10. The steps below expand the lecture argument.

Step 1: solve the regularized normal equations. For $\lambda > 0$, minimize $J(w) = \|Xw - y\|^2 + \lambda\|w\|^2$. Then

$$\begin{aligned} 0 &= 2X^\top(Xw_\lambda - y) + 2\lambda w_\lambda, \\ (X^\top X + \lambda I_d)w_\lambda &= X^\top y, \\ w_\lambda &= (X^\top X + \lambda I_d)^{-1}X^\top y. \end{aligned}$$

The inverse exists because $v^\top(X^\top X + \lambda I)v = \|Xv\|^2 + \lambda\|v\|^2 > 0$ for $v \neq 0$.

Step 2: derive the smaller dual solve. Associativity gives $(X^\top X+\lambda I_d)X^\top = X^\top(XX^\top+\lambda I_n)$. Multiply on the left and right by the corresponding inverses:

$$(X^\top X + \lambda I_d)^{-1}X^\top = X^\top(XX^\top + \lambda I_n)^{-1}.$$

Both formulas give the same parameters; the dual solve can be smaller when $n < d$.

Step 3: expose active and null coordinates. Let $X = U_r\Sigma_r V_r^\top$ be the compact rank-r SVD and complete V_r with V_0. Write $w = V_r a + V_0 b$. Multiplying the normal equations by $[V_r\ V_0]^\top$ gives

$$\begin{bmatrix} \Sigma_r^2 + \lambda I_r & 0 \\ 0 & \lambda I_{d-r} \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} \Sigma_r U_r^\top y \\ 0 \end{bmatrix}.$$

Thus $b = 0$, $a_i = \sigma_i(u_i^\top y)/(\sigma_i^2 + \lambda)$, and

$$Xw_\lambda = \sum_{i=1}^{r} u_i \frac{\sigma_i^2}{\sigma_i^2 + \lambda}(u_i^\top y).$$

The prediction shrinkage is $\sigma_i^2/(\sigma_i^2 + \lambda)$; the parameter coefficient is $\sigma_i/(\sigma_i^2 + \lambda)$. These are different filters. As $\lambda \downarrow 0$, $w_\lambda \to X^+ y$, the minimum-norm least-squares solution.

Lecture 2

Optimization: implicit regularization, SGD, and momentum

Course sources: lecture_02; 2025-09-04 transcript.

2.1 Implicit Regularization of Gradient Descent

Deeper explanation: Gradient flow and early-stopping time scales (p. 355)

Deeper explanation: Deriving least squares in singular coordinates (p. 352)

Deeper explanation: Strong convexity turns descent into geometric convergence (p. 363)

Deeper explanation: Without smoothness: a subgradient guarantee (p. 364)

Recap: Convergence of Gradient Descent for Linear Regression

Last time, we analyzed the convergence of Gradient Descent (GD) for linear regression. A key result was the condition on the learning rate η:

$$\eta < \frac{1}{\lambda_{\max}(X^T X)} = \frac{1}{\sigma_{\max}(X)^2} \tag{2.1}$$

For GD to converge, the learning rate must be smaller than the inverse of the largest eigenvalue of the covariance matrix $X^T X$.

The main takeaway is that the **geometric properties of the loss function** are crucial. They dictate the practical choices for hyperparameters like the learning rate. Concepts like smoothness and strong convexity are generalizations of these properties for non-quadratic functions, where we can bound the function from above and below by quadratics. The convergence rate then often depends on the ratio of these bounds, analogous to how the convergence of GD for least squares depends on the condition number $\frac{\sigma_{\max}}{\sigma_{\min}}$.

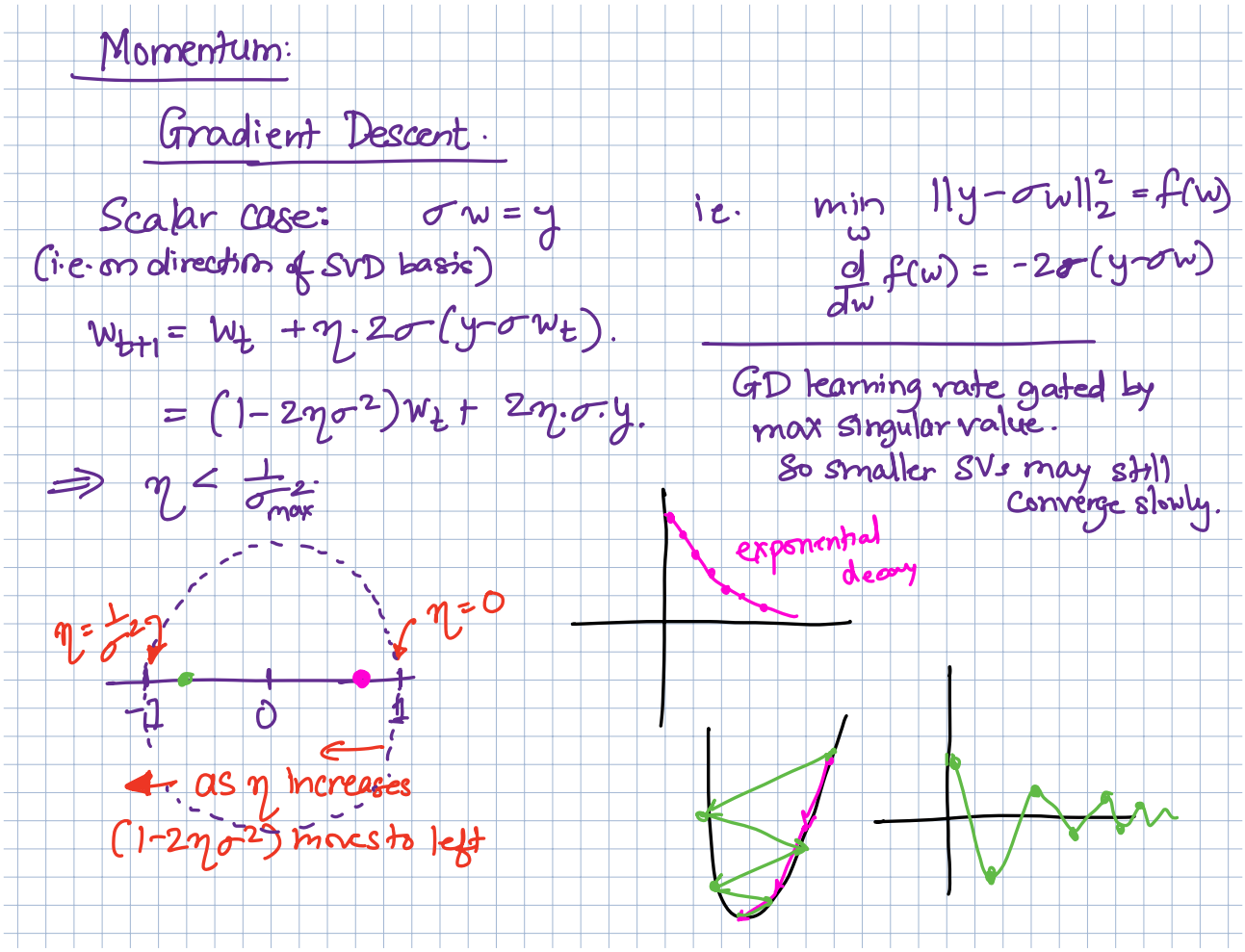


Figure 2.1: Eigenmodes of gradient descent. Each Hessian eigenvector evolves by its own scalar multiplier. Negative multipliers cause alternating signs; a magnitude below one gives decay, while a magnitude above one gives instability. Original annotated source excerpt; Gireeja Ranade, faculty handout 02, p. 8.

Ridge Regression in SVD Coordinates

To understand the connection between optimization and regularization, let's analyze Ridge Regression. The solution for Ridge is given by:

$$\vec{w}^* = (X^T X + \lambda I)^{-1} X^T \vec{y} \tag{2.2}$$

This is the standard form. There is also a "kernel form":

$$\vec{w}^* = X^T (X X^T + \lambda I)^{-1} \vec{y} \tag{2.3}$$

Let's analyze the standard form in the basis defined by the Singular Value Decomposition (SVD) of the data matrix X. Let $X = U\Sigma V^T$. Substituting this into the solution:

$$\vec{w}^* = \left((U\Sigma V^T)^T(U\Sigma V^T) + \lambda I\right)^{-1}(U\Sigma V^T)^T\vec{y} \tag{2.4}$$

$$= \left(V\Sigma^T U^T U\Sigma V^T + \lambda I\right)^{-1} V\Sigma^T U^T\vec{y} \tag{2.5}$$

$$= \left(V\Sigma^T\Sigma V^T + \lambda VV^T\right)^{-1} V\Sigma^T U^T\vec{y} \quad (\text{since } U^TU = I \text{ and } I = VV^T) \tag{2.6}$$

$$= \left(V(\Sigma^T\Sigma + \lambda I)V^T\right)^{-1} V\Sigma^T U^T\vec{y} \tag{2.7}$$

$$= (V^T)^{-1}(\Sigma^T\Sigma + \lambda I)^{-1}V^{-1}V\Sigma^T U^T\vec{y} \quad (\text{using } (ABC)^{-1} = C^{-1}B^{-1}A^{-1}) \tag{2.8}$$

$$= V(\Sigma^T\Sigma + \lambda I)^{-1}\Sigma^T U^T\vec{y} \quad (\text{since } V^{-1} = V^T \text{ and } (V^T)^{-1} = V) \tag{2.9}$$

The matrix $(\Sigma^T\Sigma + \lambda I)$ is diagonal. Its diagonal entries are $\sigma_i^2 + \lambda$ for the non-zero singular values and λ for the rest. The inverse is also diagonal with entries $1/(\sigma_i^2 + \lambda)$.

We can express the solution $\vec{w}^*$ as a linear combination of the right singular vectors $\vec{v}_i$:

$$\vec{w}^* = \sum_{i=1}^{\text{rank}(X)} \vec{v}_i \left(\frac{\sigma_i}{\sigma_i^2 + \lambda}\right)(\vec{u}_i^T\vec{y}) \tag{2.10}$$

The term $\vec{u}_i^T\vec{y}$ is a scalar projection. The key is the scaling factor $\frac{\sigma_i}{\sigma_i^2+\lambda}$.

- If λ is very large compared to σ_i^2 (i.e., for small singular values), the scaling factor approaches 0:
$$\lambda \gg \sigma_i^2 \implies \frac{\sigma_i}{\sigma_i^2 + \lambda} \to 0$$
This means the components of the solution corresponding to small singular values are heavily suppressed.
- If σ_i^2 is very large compared to λ, the scaling factor approaches $1/\sigma_i$:
$$\sigma_i^2 \gg \lambda \implies \frac{\sigma_i}{\sigma_i^2 + \lambda} \approx \frac{\sigma_i}{\sigma_i^2} = \frac{1}{\sigma_i}$$

Intuition: Small singular values correspond to directions of low variance in the data. Ridge regularization prevents the model from fitting patterns in these low-variance directions, which are often dominated by noise. This helps prevent overfitting and improves generalization.

Ridge and null-space coordinates in one block system

For rank r, use $V = [V_r\ V_0]$, $w = V_r a + V_0 b$, and the compact SVD $X = U_r\Sigma_r V_r^\top$. Multiplying the ridge normal equations by $V^\top$ gives, for $\lambda > 0$,

$$\begin{bmatrix}\Sigma_r^2 + \lambda I_r & 0\\ 0 & \lambda I_{d-r}\end{bmatrix}\begin{bmatrix}a\\ b\end{bmatrix} = \begin{bmatrix}\Sigma_r U_r^\top y\\ 0\end{bmatrix}.$$

Consequently $a_i = \sigma_i(u_i^\top y)/(\sigma_i^2 + \lambda)$ and $b = 0$. At $\lambda = 0$, the lower equation is $0 = 0$ and imposes no constraint on b; one cannot invert that zero block. The minimum-norm least-squares solution chooses $b = 0$, while unregularized GD retains the initialization's null component.

Implicit Regularization via Gradient Descent and Early Stopping

Let's analyze the standard GD update for least squares, but this time in the SVD basis. The update rule is: $\vec{w}_{t+1} = \vec{w}_t - 2\eta X^T(X\vec{w}_t - \vec{y})$. Let's change to the basis defined by V by defining $\tilde{\vec{w}}_t = V^T\vec{w}_t$.

$$V^T\vec{w}_{t+1} = V^T\vec{w}_t - 2\eta V^T(V\Sigma^T U^T)(U\Sigma V^T\vec{w}_t - \vec{y}) \tag{2.11}$$

$$\tilde{\vec{w}}_{t+1} = \tilde{\vec{w}}_t - 2\eta\Sigma^T U^T(U\Sigma\tilde{\vec{w}}_t - \vec{y}) \tag{2.12}$$

$$\tilde{\vec{w}}_{t+1} = \tilde{\vec{w}}_t - 2\eta(\Sigma^T\Sigma\tilde{\vec{w}}_t - \Sigma^T U^T\vec{y}) \tag{2.13}$$

Since $\Sigma^T\Sigma$ is a diagonal matrix, this system of equations is decoupled. We can look at the update for each component i individually:

$$\tilde{w}_{t+1,i} = \tilde{w}_{t,i} - 2\eta(\sigma_i^2\tilde{w}_{t,i} - \sigma_i\tilde{y}_i) \tag{2.14}$$

where $\tilde{y}_i$ is the i-th component of $U^T\vec{y}$. The magnitude of the gradient step in the direction of the i-th singular vector is proportional to σ_i^2.

- **Large singular values (σ_i):** GD takes large steps and converges quickly in these directions.
- **Small singular values (σ_i):** GD takes very small steps and converges slowly in these directions.

This leads to the concept of **Early Stopping**. If we run GD but stop it before it fully converges, the weights will have moved significantly in the directions of large singular values but very little in the directions of small singular values.

- **Ridge Regression:** Explicitly suppresses components related to small singular values via the λ parameter.
- **GD with Early Stopping:** Implicitly suppresses these same components because it doesn't have enough time to converge in those slow-moving directions.

Thus, early stopping acts as a form of implicit regularization, achieving a similar effect to Ridge without an explicit penalty term.

Weight Decay The GD update for Ridge regression is:

$$\vec{w}_{t+1} = \vec{w}_t - 2\eta(X^T(X\vec{w}_t - \vec{y}) + \lambda\vec{w}_t) = (1 - 2\eta\lambda)\vec{w}_t - 2\eta X^T(X\vec{w}_t - \vec{y})$$

The term $(1-2\eta\lambda)$ multiplicatively shrinks the weights at each step. This is known as **weight decay**. In practice, sometimes even without an explicit Ridge penalty in the loss, a weight decay term is added directly to the optimizer's update rule to prevent weights from growing too large.

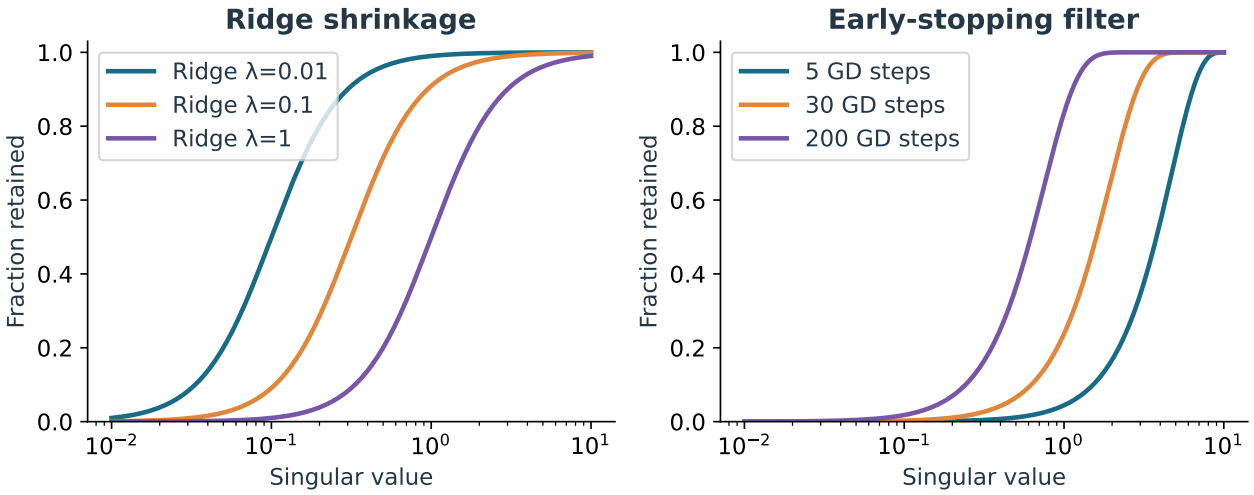


Figure 2.2: Ridge and early stopping act as different spectral filters. Weakly observed directions are learned or retained less strongly.

Worked derivation: solving the GD filter and the momentum recurrence

Lecture basis: faculty handout 02, pp. 1–4, 8–10. The steps below expand the lecture argument.

Step 1: project the affine update, rather than only its error. For zero initialization and unhalved squared loss, let $a_{t,i} = v_i^\top w_t$ and $b_i = u_i^\top y$. Then

$$a_{t+1,i} = (1 - 2\eta\sigma_i^2)a_{t,i} + 2\eta\sigma_i b_i.$$

Put $r_i = 1 - 2\eta\sigma_i^2$. Repeated substitution gives

$$\begin{aligned} a_{t,i} &= 2\eta\sigma_i b_i(1 + r_i + \cdots + r_i^{t-1}) \\ &= 2\eta\sigma_i b_i \frac{1 - r_i^t}{1 - r_i} \\ &= \frac{b_i}{\sigma_i}[1 - (1 - 2\eta\sigma_i^2)^t]. \end{aligned}$$

This calculation is for $\sigma_i > 0$; zero modes stay zero. Therefore

$$Xw_t = \sum_i u_i \underbrace{[1 - (1 - 2\eta\sigma_i^2)^t]}_{\text{prediction filter}} b_i.$$

For $0 < 2\eta\sigma_i^2 \ll 1$, $(1 - 2\eta\sigma_i^2)^t \approx e^{-2\eta t\sigma_i^2}$. Small singular modes need a long time to become visible. Monotone shrinkage requires $\eta \le 1/(2\sigma_{\max}^2)$; the larger stable range allows alternating, overshooting modes. Early stopping resembles ridge qualitatively, but no single λ matches all its mode filters exactly.

Step 2: remove the velocity variable in momentum. For $m_t = \beta m_{t-1} + (1-\beta)\nabla L(w_t)$ and $w_{t+1} = w_t - \eta m_t$, substitute $m_{t-1} = (w_{t-1} - w_t)/\eta$:

$$w_{t+1} = w_t - \beta(w_{t-1} - w_t) - \eta(1-\beta)\nabla L(w_t),$$
$$z_{t+1,i} = [1 + \beta - 2\eta(1-\beta)\sigma_i^2] z_{t,i} - \beta z_{t-1,i}.$$

Unlike GD, each mode has a second-order recurrence. A trial solution $z_{t,i} = r^t$ gives

$$r^2 - [1 + \beta - 2\eta(1-\beta)\sigma_i^2] r + \beta = 0.$$

Convergence requires both roots inside the unit circle. Complex roots explain damped oscillations; momentum is not simply GD with an always larger safe step.

2.2 Stochastic Gradient Descent (SGD)

Deeper explanation: What increasing the batch size changes (p. 370)

While GD is effective for convex problems, training deep neural networks involves optimizing highly non-convex loss landscapes over massive datasets.

The SGD Algorithm

Many loss functions can be decomposed as a sum over the individual data points.

$$f(\theta) = \frac{1}{n} \sum_{i=1}^{n} f_i(\theta) \tag{2.15}$$

where $f_i(\theta)$ is the loss on the i-th data point. Computing the full gradient $\nabla f(\theta) = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\theta)$ requires a full pass over the dataset, which is computationally expensive if n is large.

SGD approximates the full gradient using a small, randomly sampled subset of the data called a **mini-batch**. The SGD update rule is:

$$\theta_{t+1} = \theta_t - \eta \cdot \frac{1}{B} \sum_{i \in \mathcal{B}_t} \nabla f_i(\theta_t) \tag{2.16}$$

where $\mathcal{B}_t$ is a mini-batch of size B sampled uniformly at random from the dataset at step t.

- **Mini-batch SGD:** Standard approach with $1 < B \ll n$.
- **Full-batch SGD:** This is just standard Gradient Descent ($B = n$).

A key property is that the mini-batch gradient is an **unbiased estimator** of the true gradient:

$$\mathbb{E}_{i \sim \text{Uniform}}[\nabla f_i(\theta)] = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\theta) = \nabla f(\theta) \tag{2.17}$$

On average, SGD moves in the correct direction.

Properties of SGD

- **Advantages:**
  1. **Computationally efficient:** Requires processing only a small batch of data per update.
  2. **Escaping local minima:** The noise in the gradient estimate can help the optimizer "jump out" of sharp local minima or move past saddle points, potentially finding a better, more generalizable solution.
- **Disadvantages:**
  1. **Noisy updates:** The convergence path is much more erratic than GD.
  2. **Convergence guarantees:** Traditional optimization theory requires a decaying learning rate schedule (e.g., $\eta_t \propto 1/t$) for SGD to provably converge. However, in deep learning, SGD with a constant step size is often used and can still converge, particularly in over-parameterized regimes.

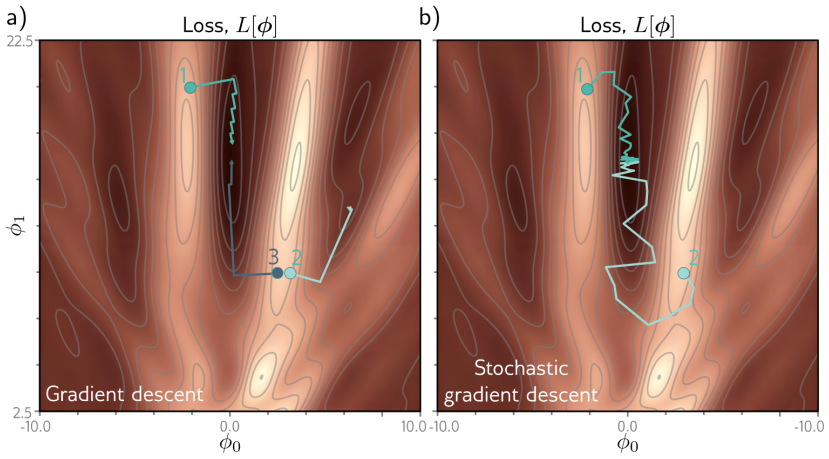


Figure 6.5 Gradient descent vs. stochastic gradient descent. a) Gradient descent with line search. As long as the gradient descent algorithm is initialized in the right "valley" of the loss function (e.g., points 1 and 3), the parameter estimate will move steadily toward the global minimum. However, if it is initialized outside this valley (e.g., point 2), it will descend toward one of the local minima. b) Stochastic gradient descent adds noise to the optimization process, so it is possible to escape from the wrong valley (e.g., point 2) and still reach the global minimum.

Prince 2025.

Figure 2.3: Full-gradient and stochastic paths on a loss surface. The paths visualize how a gradient estimate changes the trajectory. Noise can keep updates moving near a minimum; it does not guarantee escape to a better basin. The published valley plots are preserved from the handout. Original annotated source excerpt; Gireeja Ranade, faculty handout 02, p. 7.

Worked derivation: unbiased gradients and batch variance

Lecture basis: faculty handout 02, pp. 5–8. The steps below expand the lecture argument.

Step 1: declare the objective and sampling law. Let $F(w) = n^{-1} \sum_{i=1}^{n} \ell_i(w)$ and sample I uniformly and independently of the past, conditional on w_t. Set $g_I = \nabla \ell_I(w_t)$ and $\bar{g} = n^{-1} \sum_i g_i$. Then

$$\mathbb{E}[g_I \mid w_t] = \sum_i \frac{1}{n} g_i = \nabla F(w_t).$$

For the *sum* objective, $n g_I$ is unbiased instead. A factor n can be absorbed into a learning rate, but must not disappear midway through a proof.

Step 2: expand the error of an independent mini-batch. For b samples with replacement, $\hat{g} = b^{-1} \sum_{j=1}^{b} g_{I_j}$. Set $\xi_j = g_{I_j} - \bar{g}$, so $\mathbb{E}[\xi_j \mid w_t] = 0$. Independence cancels every cross term:

$$\begin{aligned}\mathbb{E}\|\hat{g} - \bar{g}\|^2 &= \frac{1}{b^2} \sum_{j,k} \mathbb{E}[\xi_j^\top \xi_k] \\ &= \frac{1}{b^2} \sum_{j} \mathbb{E}\|\xi_j\|^2 \\ &= \frac{1}{b} \frac{1}{n} \sum_{i} \|g_i - \bar{g}\|^2.\end{aligned}$$

All expectations here are conditional on w_t. Thus the *variance* scales as $1/b$, while the root-mean-square noise scales as $1/\sqrt{b}$. Sampling without replacement introduces negative correlations and a finite-population correction; it is a different calculation.

2.3 Momentum

Deeper explanation: Momentum as a second-order recurrence (p. 370)

Motivation: The Problem of Oscillation

In GD, the learning rate η is limited by the largest singular value, $\sigma_{\max}$. This choice can lead to slow convergence along directions with smaller singular values. If we try to increase η to speed up convergence, we can introduce oscillations, where the optimizer bounces back and forth across a ravine in the loss landscape instead of moving smoothly along the bottom.

How can we accelerate convergence while damping these oscillations? The key idea is to average gradients over time.

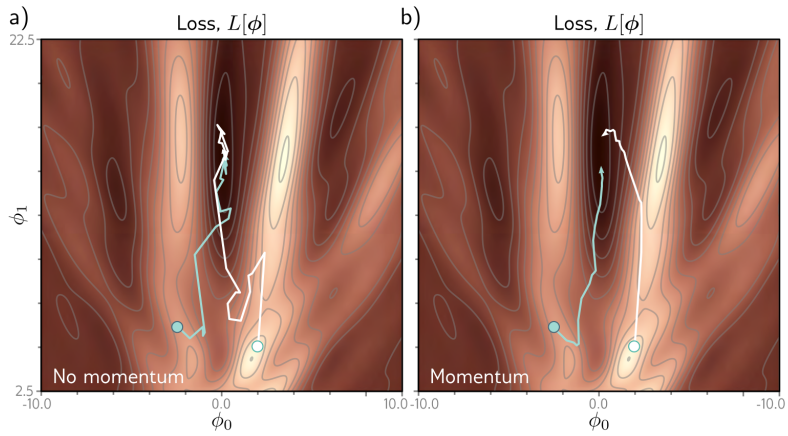


Figure 6.7 Stochastic gradient descent with momentum. a) Regular stochastic descent takes a very indirect path toward the minimum. b) With a momentum term, the change at the current step is a weighted combination of the previous change and the gradient computed from the batch. This smooths out the trajectory and increases the speed of convergence.

Figure 2.4: Why momentum changes the trajectory. The valley plots compare noisy updates with their accumulated direction. Smoothing can reduce transverse oscillation while retaining progress along a narrow valley. Published plot panels are reproduced as they appear in the handout. Original annotated source excerpt; Gireeja Ranade, faculty handout 02, p. 11.

An Exponential Moving Average of Gradients

Instead of a simple moving average, which requires storing many past gradients, we can use a more memory-efficient **exponential moving average**. This idea is analogous to a low-pass filter in an RC circuit, which smooths out high-frequency signals by maintaining a state (the charge on the capacitor).

The Momentum algorithm maintains a "velocity" vector $\vec{z}_k$, which is an exponential moving average of past gradients. The update rules are:

$$\vec{z}_{k+1} = \beta \vec{z}_k + (1-\beta)\nabla f(\vec{w}_k) \quad (\text{with } \vec{z}_0 = \mathbf{0}) \tag{2.18}$$

$$\vec{w}_{k+1} = \vec{w}_k - \eta \vec{z}_{k+1} \tag{2.19}$$

Here, $\beta \in [0, 1)$ is the momentum hyperparameter. A typical value is $\beta = 0.9$. If we expand the recurrence for $\vec{z}_k$, we can see its structure:

$$\begin{aligned}
\vec{z}_1 &= (1-\beta)\nabla f(\vec{w}_0) \\
\vec{z}_2 &= \beta(1-\beta)\nabla f(\vec{w}_0) + (1-\beta)\nabla f(\vec{w}_1) \\
\vec{z}_3 &= \beta^2(1-\beta)\nabla f(\vec{w}_0) + \beta(1-\beta)\nabla f(\vec{w}_1) + (1-\beta)\nabla f(\vec{w}_2)
\end{aligned}$$

The velocity $\vec{z}_k$ is a weighted sum of all past gradients, with weights that decay exponentially into the past.

Intuition: Momentum acts like a ball rolling down a hill.

- In directions where the gradient is consistently pointing the same way, the velocity builds up, and the optimizer accelerates.
- In directions where the gradient oscillates, the velocity averages out these oscillations, leading to smaller steps and a smoother path.

The main benefit is **faster convergence**. Theoretically, for quadratic objectives, momentum changes the dependency of the convergence rate from the condition number κ to $\sqrt{\kappa}$, which is a significant improvement.

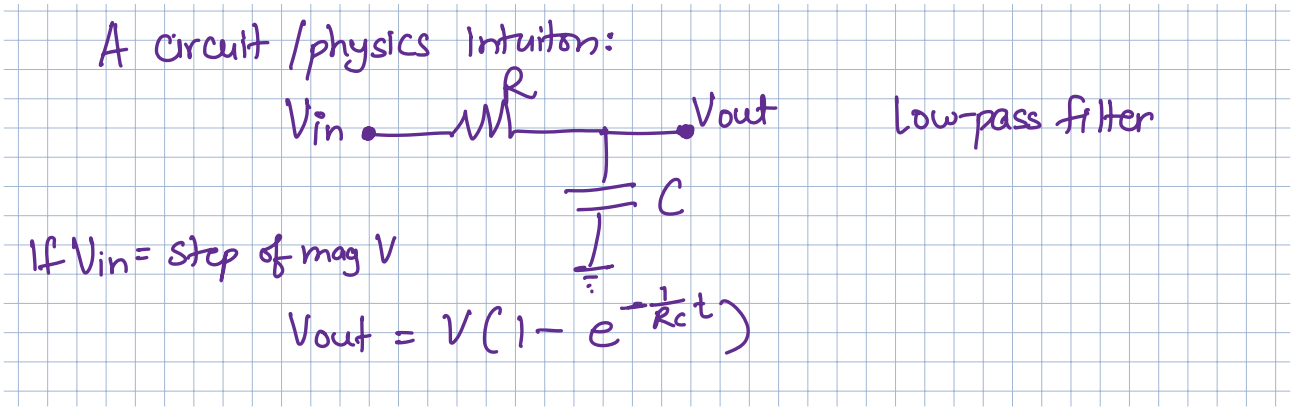


Figure 2.5: Momentum as a low-pass filter. The RC circuit is the lecture analogy for smoothing rapid changes. An exponential moving average keeps persistent gradient directions and attenuates alternating components; its normalization convention can be absorbed into the step size. Original annotated source excerpt; Gireeja Ranade, faculty handout 02, p. 9.

2.4 A Look at SGD Convergence (Underdetermined Case)

Deeper explanation: An alternative proof using squared parameter error (p. 369)

Lecture 2 handout, pages 12–18. We now carry out the proof rather than only describing its strategy. Each stage below identifies the quantity being transformed, the algebraic step, and the assumption used. We use the unhalved squared loss, matching the factor 2η on the slides.

1. Set up the problem and translate to error coordinates

Let $X \in \mathbb{R}^{n\times d}$ have full row rank, with $n < d$. Write its rows as $x_i^\top$, where $x_i \in \mathbb{R}^d$. Full row rank makes $XX^\top$ invertible. Define

$$w^* = X^\top (XX^\top)^{-1} y.$$

First verify that it interpolates the labels:

$$\begin{aligned} Xw^* &= XX^\top (XX^\top)^{-1} y \\ &= I_n y \\ &= y. \end{aligned}$$

It lies in the row space of X. Every other interpolating solution differs from it by a vector in $\ker X$, which is orthogonal to the row space. Consequently $\|w^* + v\|^2 = \|w^*\|^2 + \|v\|^2$ for $v \in \ker X$: this proves that w^* is the minimum-norm solution.

Initialize $w_0 = 0$ and define the error $q_t = w_t - w^*$. The current residual becomes

$$\begin{aligned} Xw_t - y &= Xw_t - Xw^* && \text{because } Xw^* = y \\ &= X(w_t - w^*) \\ &= Xq_t. \end{aligned}$$

The desired endpoint is $Xq = 0$; this equation is *not* assumed to hold during training. Initially,

$$q_0 = w_0 - w^* = -w^*.$$

At each step, sample I_t uniformly from $\{1, \dots, n\}$, independently of the previous samples. Our aim is to find one fixed positive step size for which the expected squared residual tends to zero and then show that the parameters approach w^*.

2. Write the full SVD and expose both coordinate blocks

A full SVD of a wide full-row-rank matrix has the form

$$\underset{n\times d}{\underset{\sim}{X}} = \underset{n\times n}{\underset{\sim}{U}} \underbrace{\begin{bmatrix}\Sigma_1 & 0_{n\times(d-n)}\end{bmatrix}}_{n\times d} \underbrace{\begin{bmatrix}V_r^\top \\ V_0^\top\end{bmatrix}}_{V^\top\in\mathbb{R}^{d\times d}},$$

where $\Sigma_1 = \mathrm{diag}(\sigma_1, \dots, \sigma_n)$ has positive entries. The columns of V_r span the row space and those of V_0 span the null space. In particular,

$$V_r^\top V_r = I_n, \quad V_0^\top V_r = 0, \quad XV_0 = 0.$$

Define the two error-coordinate blocks without discarding either:

$$\begin{bmatrix}z_t \\ b_t\end{bmatrix} = \begin{bmatrix}V_r^\top \\ V_0^\top\end{bmatrix} q_t, \qquad q_t = V_r z_t + V_0 b_t.$$

Here $z_t \in \mathbb{R}^n$ is the slides' reduced error vector $\tilde{z}_t$; $b_t \in \mathbb{R}^{d-n}$ contains the null coordinates. Multiply the blocks explicitly:

$$\begin{aligned} Xq_t &= U\begin{bmatrix}\Sigma_1 & 0\end{bmatrix}\begin{bmatrix}z_t \\ b_t\end{bmatrix} \\ &= U(\Sigma_1 z_t + 0 b_t) \\ &= \underbrace{U\Sigma_1}_{\widetilde{X}\in\mathbb{R}^{n\times n}} z_t. \end{aligned}$$

Thus the residual depends on z_t, but not on b_t. Next compute the initial blocks. Since $XX^\top = U\Sigma_1^2 U^\top$,

$$\begin{aligned} w^* &= V_r\Sigma_1 U^\top (U\Sigma_1^{-2}U^\top)y = V_r\Sigma_1^{-1}U^\top y, \\ \begin{bmatrix}z_0 \\ b_0\end{bmatrix} &= -\begin{bmatrix}V_r^\top V_r \\ V_0^\top V_r\end{bmatrix}\Sigma_1^{-1}U^\top y = \begin{bmatrix}-\Sigma_1^{-1}U^\top y \\ 0\end{bmatrix}. \end{aligned}$$

This establishes $b_0 = 0$. We still need to prove that the updates preserve it; initialization alone is not enough.

3. Derive the sample update and justify the reduction

For one example, $\ell_i(w) = (x_i^\top w - y_i)^2$. Apply the chain rule:

$$\nabla \ell_i(w) = 2(x_i^\top w - y_i)x_i.$$

Because $y_i = x_i^\top w^*$, subtracting w^* from the SGD update gives

$$\begin{aligned} q_{t+1} &= w_t - 2\eta(x_i^\top w_t - y_i)x_i - w^* \\ &= q_t - 2\eta(x_i^\top q_t)x_i \\ &= (I_d - 2\eta x_i x_i^\top)q_t. \end{aligned}$$

Let $a_i = V_r^\top x_i \in \mathbb{R}^n$. Since $V_0^\top x_i = 0$, the full sample outer product in singular coordinates is

$$V^\top x_i x_i^\top V = \begin{bmatrix} a_i \\ 0 \end{bmatrix} \begin{bmatrix} a_i^\top & 0 \end{bmatrix} = \begin{bmatrix} a_i a_i^\top & 0 \\ 0 & 0 \end{bmatrix}.$$

Therefore

$$\begin{bmatrix} z_{t+1} \\ b_{t+1} \end{bmatrix} = \begin{bmatrix} I_n - 2\eta a_i a_i^\top & 0 \\ 0 & I_{d-n} \end{bmatrix} \begin{bmatrix} z_t \\ b_t \end{bmatrix}.$$

Reading the two block rows proves

$$\boxed{z_{t+1} = z_t - 2\eta a_i a_i^\top z_t, \qquad b_{t+1} = b_t = 0.}$$

The reduced system is now justified for every iterate. Row i of $\widetilde{X} = XV_r$ is $a_i^\top$. Set

$$H = \widetilde{X}^\top \widetilde{X} = \Sigma_1 U^\top U \Sigma_1 = \Sigma_1^2.$$

Full-batch GD on the summed loss would use $z_{t+1} = (I_n - 2\eta H)z_t$, which is diagonal in these coordinates. A sampled $a_i a_i^\top$ generally is not diagonal. Uniform sampling instead gives

$$\mathbb{E}[a_{I_t} a_{I_t}^\top \mid z_t] = \frac{1}{n}\sum_{i=1}^{n} a_i a_i^\top = \frac{H}{n}.$$

Hence the sample gradient is unbiased for the *mean* loss gradient. This is the source of the $1/n$ factors below; the summed loss remains a valid quantity with which to measure progress.

4. Expand the loss and bound the linear term

Define the nonnegative Lyapunov function

$$\mathcal{L}(z) = \|\widetilde{X}z\|^2 = z^\top H z = \sum_{j=1}^{n} \sigma_j^2 z_j^2.$$

Condition on the current $z = z_t$, and write

$$\Delta = z_{t+1} - z_t = -2\eta a_{I_t} a_{I_t}^\top z.$$

Expand every term of the new loss:

$$\begin{aligned}\mathcal{L}(z+\Delta) &= (z+\Delta)^\top H(z+\Delta) \\ &= z^\top H z + z^\top H\Delta + \Delta^\top H z + \Delta^\top H\Delta \\ &= \mathcal{L}(z) + \underbrace{2z^\top H\Delta}_{A} + \underbrace{\Delta^\top H\Delta}_{B}.\end{aligned}$$

The two cross terms agree because $H = H^\top$ and each is a scalar. This is an exact quadratic expansion, with no omitted remainder.

For the linear term, z and H are fixed under the conditional expectation:

$$\begin{aligned}\mathbb{E}[A \mid z] &= 2z^\top H \mathbb{E}[\Delta \mid z] \\ &= 2z^\top H\left(-\frac{2\eta}{n} H z\right) \\ &= -\frac{4\eta}{n} z^\top H^2 z \\ &= -\frac{4\eta}{n} \sum_{j=1}^{n} \sigma_j^4 z_j^2.\end{aligned}$$

Let $\sigma_{\min} = \min_j \sigma_j > 0$. Term by term,

$$\sigma_j^4 z_j^2 \geq \sigma_{\min}^2 \sigma_j^2 z_j^2, \quad \text{so} \quad \sum_j \sigma_j^4 z_j^2 \geq \sigma_{\min}^2 \mathcal{L}(z).$$

Multiplying this inequality by $-4\eta/n$ reverses its direction. Thus

$$\boxed{\mathbb{E}[A \mid z] \leq -\frac{4\eta\sigma_{\min}^2}{n} \mathcal{L}(z).}$$

This is the progress term on slide 16. Its coefficient is negative and proportional to η.

5. Bound the quadratic term, one inequality at a time

For a fixed sampled row a_i, substitute the update into B:

$$\begin{aligned} B &= \Delta^\top H \Delta \\ &= (-2\eta a_i a_i^\top z)^\top H (-2\eta a_i a_i^\top z) \\ &= 4\eta^2 z^\top a_i a_i^\top H a_i a_i^\top z \\ &= 4\eta^2 (a_i^\top z)^2 (a_i^\top H a_i). \end{aligned}$$

The last line separates the squared sample residual from the curvature in the sampled direction. To bound that curvature, use $H = \Sigma_1^2$:

$$\begin{aligned} a_i^\top H a_i &= \sum_{j=1}^{n} \sigma_j^2 (a_i)_j^2 \\ &\le \sigma_{\max}^2 \sum_{j=1}^{n} (a_i)_j^2 \\ &= \sigma_{\max}^2 \|a_i\|^2. \end{aligned}$$

Define a bound that works for *every* row,

$$\rho^2 = \max_{1 \le i \le n} \|a_i\|^2.$$

All factors in the expression for B are nonnegative, so substitution preserves the inequality direction:

$$\begin{aligned} B &\le 4\eta^2 \sigma_{\max}^2 \|a_i\|^2 (a_i^\top z)^2 \\ &\le 4\eta^2 \sigma_{\max}^2 \rho^2 (a_i^\top z)^2. \end{aligned}$$

Now average over the uniform sample. The row-norm bound is a constant, while the squared residual is averaged exactly:

$$\begin{aligned} \mathbb{E}[B \mid z] &\le 4\eta^2 \sigma_{\max}^2 \rho^2 \frac{1}{n} \sum_{i=1}^{n} (a_i^\top z)^2 \\ &= \frac{4\eta^2 \sigma_{\max}^2 \rho^2}{n} \|\widetilde{X} z\|^2. \end{aligned}$$

We have obtained the precise slide-17 result:

$$\boxed{\mathbb{E}[B \mid z] \le \frac{4\eta^2}{n} \sigma_{\max}^2 \rho^2 \mathcal{L}(z).}$$

This nonnegative cost is proportional to η^2, so a small enough step lets the linear improvement dominate it.

6. Combine the bounds and choose a fixed step size

Return to the exact expansion, then substitute the two bounds:

$$\begin{aligned}\mathbb{E}[\mathcal{L}(z_{t+1}) \mid z_t] &= \mathcal{L}(z_t) + \mathbb{E}[A \mid z_t] + \mathbb{E}[B \mid z_t] \\ &\leq \mathcal{L}(z_t) - \frac{4\eta\sigma_{\min}^2}{n}\mathcal{L}(z_t) + \frac{4\eta^2\sigma_{\max}^2\rho^2}{n}\mathcal{L}(z_t).\end{aligned}$$

Define the constants used on slide 18:

$$c_1 = \frac{4\sigma_{\min}^2}{n}, \qquad c_2 = \frac{4\sigma_{\max}^2\rho^2}{n}.$$

They depend on the fixed data matrix, not on the iteration or current error. Factoring out the current loss gives

$$\boxed{\mathbb{E}[\mathcal{L}(z_{t+1}) \mid z_t] \leq (1 - c_1\eta + c_2\eta^2)\mathcal{L}(z_t).}$$

Write $\kappa_\eta = 1 - c_1\eta + c_2\eta^2$. For $\eta > 0$, check explicitly when it is below one:

$$\begin{aligned}\kappa_\eta < 1 &\iff -c_1\eta + c_2\eta^2 < 0 \\ &\iff \eta(c_2\eta - c_1) < 0 \\ &\iff 0 < \eta < \frac{c_1}{c_2} = \frac{\sigma_{\min}^2}{\sigma_{\max}^2\rho^2}.\end{aligned}$$

For a convenient, conservative choice that also bounds the factor away from zero, take

$$0 < \eta \leq \min\left\{\frac{c_1}{2c_2}, \frac{1}{2c_1}\right\}.$$

The first restriction gives $c_2\eta^2 \leq c_1\eta/2$. The second gives $c_1\eta \leq 1/2$. Therefore

$$\frac{1}{2} \leq 1 - c_1\eta \leq \kappa_\eta \leq 1 - \frac{c_1\eta}{2} < 1.$$

This is a fixed contraction factor, valid for every state. The bound is sufficient for this proof; it need not be the largest stable step size.

7. Complete the argument: loss, parameters, and iteration count

The slide-18 inequality is the one-step result. To turn it into convergence, use the tower property of conditional expectation:

$$\begin{aligned}\mathbb{E}\mathcal{L}(z_{t+1}) &= \mathbb{E}[\mathbb{E}[\mathcal{L}(z_{t+1}) \mid z_t]] \\ &\le \mathbb{E}[\kappa_\eta \mathcal{L}(z_t)] \\ &= \kappa_\eta \mathbb{E}\mathcal{L}(z_t).\end{aligned}$$

The last equality requires a factor independent of the current state. Iterating gives

$$\mathbb{E}\mathcal{L}(z_t) \le \kappa_\eta \mathbb{E}\mathcal{L}(z_{t-1}) \le \cdots \le \kappa_\eta^t \mathcal{L}(z_0) \longrightarrow 0.$$

Nonnegativity and decreasing expectation alone would not prove this: a decreasing positive sequence can converge to a positive limit. The uniform factor strictly below one supplies geometric decay.

To obtain parameter convergence, use $b_t = 0$, the orthonormal columns of V_r, and the smallest positive singular value:

$$\begin{aligned}\|w_t - w^*\|^2 = \|V_r z_t\|^2 = z_t^\top V_r^\top V_r z_t = \|z_t\|^2, \\ \mathcal{L}(z_t) = \sum_j \sigma_j^2 z_{t,j}^2 \ge \sigma_{\min}^2 \|z_t\|^2.\end{aligned}$$

Combining the two statements and taking expectations yields

$$\boxed{\mathbb{E}\|w_t - w^*\|^2 \le \frac{\kappa_\eta^t \mathcal{L}(z_0)}{\sigma_{\min}^2} \longrightarrow 0.}$$

This proves mean-square convergence to the minimum-norm solution. For a loss tolerance $0 < \varepsilon < \mathcal{L}(z_0)$, use $1 - u \le e^{-u}$:

$$\mathbb{E}\mathcal{L}(z_t) \le e^{-c_1 \eta t/2} \mathcal{L}(z_0) \le \varepsilon \quad \text{if} \quad t \ge \frac{2}{c_1 \eta} \log \frac{\mathcal{L}(z_0)}{\varepsilon}.$$

If the initial loss is zero, all sampled residuals are already zero and no iterations are needed. A nonzero initial null component would be preserved, so the parameter limit would not generally be the minimum-norm solution. Finally, interpolation is essential here: each sample gradient vanishes at w^*, eliminating the persistent additive noise that can prevent exact constant-step convergence in a noninterpolating problem.

Lecture 3

Optimization: Adam, locally linear perspective, features

Course sources: lecture_03; 2025-09-09 transcript.

3.1 Convergence of SGD with Constant Step Size

Deeper explanation: Persistent noise creates a floor (p. 368)

The complete derivation for Lecture 2 handout pages 12–18 is in Lecture 2, §2.4 (p. 35). It develops the error coordinates, rectangular SVD, sample update, both terms of the loss expansion, and the final convergence bound line by line.

The result carried into this lecture is that, for full-row-rank interpolating data, zero initialization, uniform sampling, and a sufficiently small fixed step, SGD converges in mean square to the minimum-norm solution. The proof requires a state-independent contraction factor below one; mere monotonicity does not suffice. The interpolation assumption removes sample-gradient noise at the solution. The linked supplement explains what changes when persistent noise remains.

3.2 The Adam Optimizer

Deeper explanation: Adam's denominator is an uncentered second moment (p. 371)

Motivation: The Challenge of Gradient Descent

Gradient Descent (GD) exhibits a behavior that is both a feature and a bug: it takes larger steps in directions corresponding to large singular values and smaller steps in directions corresponding to small singular values.

- **Advantage**: This behavior leads to **implicit regularization**. By using early stopping, GD tends to find solutions that are simple in the basis of the top singular vectors, often corresponding to low-norm solutions that generalize well.
- **Disadvantage**: It is difficult to choose a single learning rate η. If η is too large, the optimization can become unstable and diverge along the directions of large singular values. If η is too small, progress along directions of small singular values becomes extremely slow, leading to long convergence times.

An ideal (but computationally infeasible) solution would be to compute the SVD of the data matrix and use a different, normalized learning rate for each singular direction.

Normalizing Gradients: Sign SGD

A practical approach is to normalize the gradient updates in the standard coordinate basis. Instead of using the gradient directly, we can use its sign. This ensures that the step size is uniform in every coordinate direction.

$$\vec{w}_{t+1} = \vec{w}_t - \eta \cdot \text{sgn}(\nabla_{\vec{w}} \mathcal{L}(\vec{w}_t)) \tag{3.1}$$

Away from zero, dividing a gradient coordinate by its absolute value gives its sign. The following epsilon-regularized version is a smooth approximation to that rule:

$$\vec{w}_{t+1} = \vec{w}_t - \eta \frac{\nabla_{\vec{w}} \mathcal{L}(\vec{w}_t)}{\sqrt{(\nabla_{\vec{w}} \mathcal{L}(\vec{w}_t))^2 + \epsilon}} \tag{3.2}$$

where the square is element-wise, and ϵ is a small constant for numerical stability. **Problem**: This "Sign SGD" method can be too aggressive. Near the optimum, the gradient magnitude becomes small, but the sign update remains large, causing the algorithm to overshoot and oscillate around the minimum without converging.

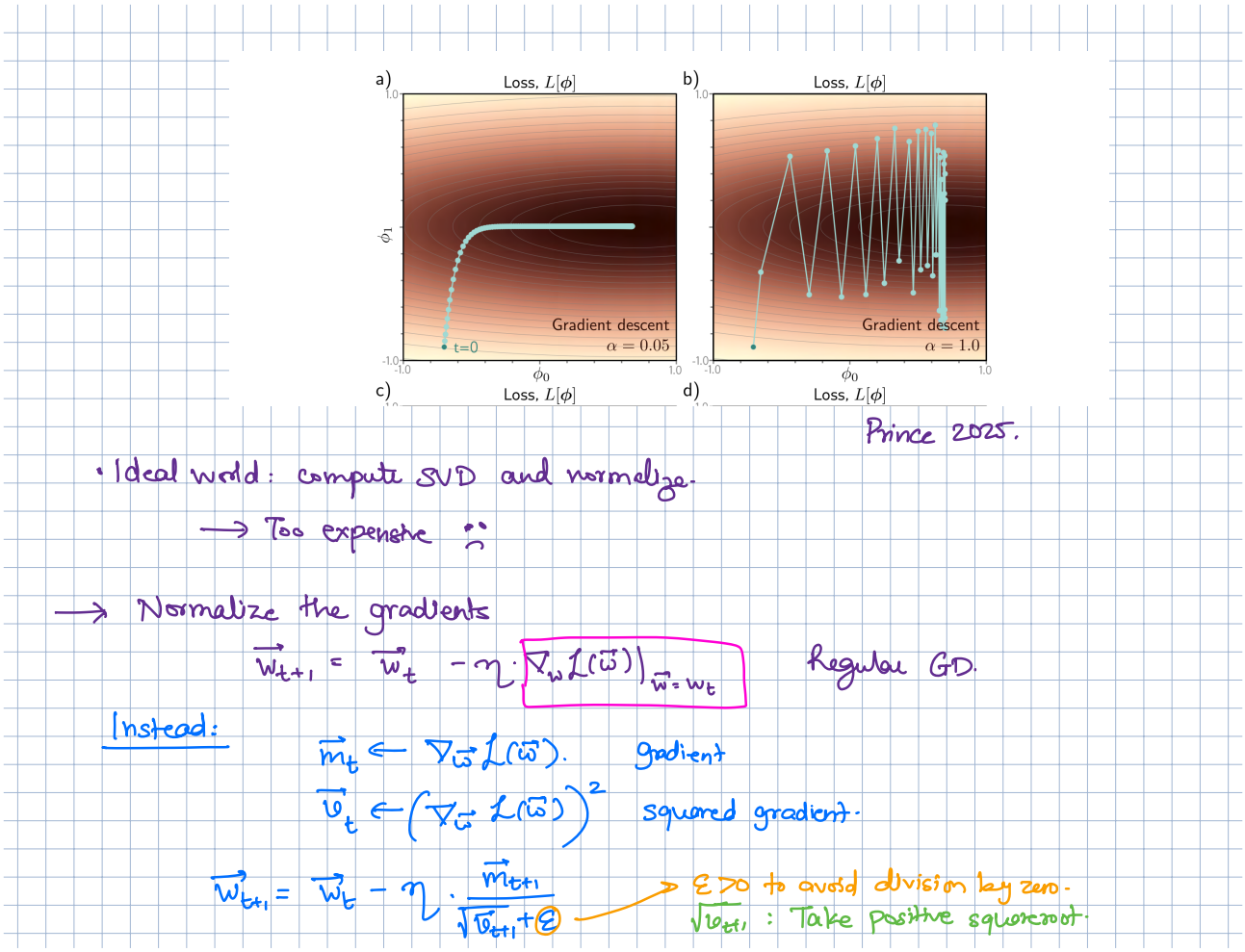


Figure 3.1: Gradient magnitude and coordinatewise normalization. Compare the direction and relative coordinate sizes of the illustrated updates. Normalizing components changes the geometry of descent; it is not simply taking a smaller step along the original gradient. Published panels remain as reproduced in the faculty handout. Original annotated source excerpt; Gireeja Ranade, faculty handout 03, p. 2.

Adam: Adaptive Moment Estimation

Adam (Adaptive Moment Estimation) refines this idea by smoothing the normalization process using momentum. It maintains two exponentially decaying moving averages:

1. **First Moment (Mean)**: An estimate of the mean of the gradients (like in momentum).
2. **Second Moment (Uncentered Variance)**: An estimate of the mean of the element-wise squared gradients.

Let $\vec{g}_t = \nabla_{\vec{w}} \mathcal{L}(\vec{w}_t)$. The moving averages are updated as follows:

$$\vec{m}_t = \beta_1 \vec{m}_{t-1} + (1 - \beta_1)\vec{g}_t \tag{3.3}$$

$$\vec{v}_t = \beta_2 \vec{v}_{t-1} + (1 - \beta_2)(\vec{g}_t \odot \vec{g}_t) \tag{3.4}$$

where β_1 and β_2 are decay rates (e.g., 0.9 and 0.999), and $\odot$ denotes element-wise multiplication.

Bias Correction: Since $\vec{m}_0$ and $\vec{v}_0$ are initialized to zero, the estimates $\vec{m}_t$ and $\vec{v}_t$ are biased towards zero, especially during the initial steps. Adam corrects for this bias:

$$\hat{\vec{m}}_t = \frac{\vec{m}_t}{1-\beta_1^t} \tag{3.5}$$

$$\hat{\vec{v}}_t = \frac{\vec{v}_t}{1-\beta_2^t} \tag{3.6}$$

The final Adam update rule combines these components:

$$\vec{w}_{t+1} = \vec{w}_t - \eta \frac{\hat{\vec{m}}_t}{\sqrt{\hat{\vec{v}}_t} + \epsilon} \tag{3.7}$$

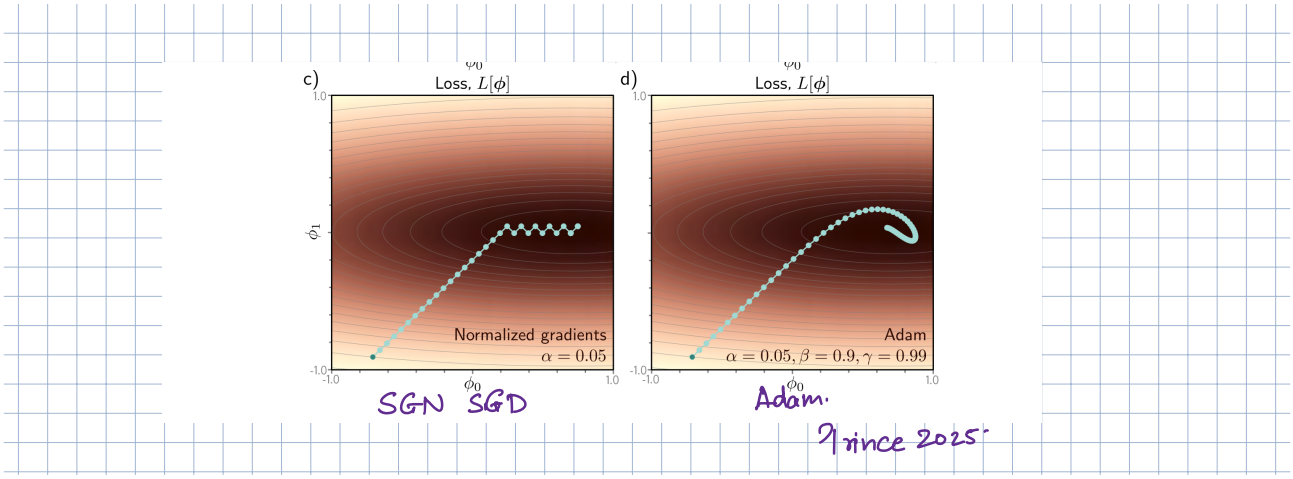


Figure 3.2: Adaptive updates in an elongated valley. Coordinatewise scaling changes the relative step sizes along steep and shallow directions. This geometric illustration explains the motivation for Adam; it is not a guarantee of superiority on every objective. Published panels retain the handout presentation. Original annotated source excerpt; Gireeja Ranade, faculty handout 03, p. 5.

Properties of Adam

- **Vanishing/Exploding Gradients**: The normalization inherent in Adam helps mitigate the issues of vanishing or exploding gradients in deep networks. The step size is less dependent on the gradient's magnitude, making it more stable.
- **Memory Cost**: Adam requires more memory than simpler optimizers.
  - SGD: Stores only the parameters (1x memory).
  - SGD with Momentum: Stores parameters and the first moment vector (2x memory).

– Adam: Stores parameters, first moment vector, and second moment vector (3x memory).

AdamW: Adam with Decoupled Weight Decay

Standard L2 regularization (weight decay) is implemented by adding $\frac{\lambda}{2}||\vec{w}||^2$ to the loss. In GD, this results in an update term that shrinks the weights: $\vec{w}_{t+1} = (1-\eta\lambda)\vec{w}_t - \eta\nabla\mathcal{L}_{\text{data}}$.

This approach does not work well with Adam. The gradient from the L2 penalty ($\lambda\vec{w}$) gets included in the adaptive normalization. This couples the effect of the regularization strength λ with the adaptive learning rate, leading to suboptimal performance.

AdamW solves this by **decoupling** the weight decay from the gradient-based update. The Adam direction uses only the data loss gradient, while a separate decay term acts on the current weights.

$$\vec{w}_{t+1} = (1-\lambda')\vec{w}_t - \eta\frac{\hat{\vec{m}}_t}{\sqrt{\hat{\vec{v}}_t}+\epsilon} \tag{3.8}$$

where λ' is the weight decay rate. This makes the regularization more effective and predictable.

Worked derivation: EMA bias correction and decoupled decay

Lecture basis: faculty handout 03, pp. 3–6. The steps below expand the lecture argument.

Step 1: unroll the initialized average. With $m_0 = 0$ and $m_t = \beta_1 m_{t-1} + (1-\beta_1)g_t$,

$$\begin{aligned} m_t &= (1-\beta_1)g_t + \beta_1(1-\beta_1)g_{t-1} + \cdots \\ &= (1-\beta_1)\sum_{k=1}^{t}\beta_1^{t-k}g_k. \end{aligned}$$

If the gradients have a common expectation μ, linearity of expectation gives

$$\begin{aligned} \mathbb{E}[m_t] &= (1-\beta_1)\mu\sum_{j=0}^{t-1}\beta_1^j \\ &= (1-\beta_1^t)\mu, \\ \widehat{m}_t &= m_t/(1-\beta_1^t). \end{aligned}$$

Likewise, $v_t = (1-\beta_2)\sum_{k=1}^{t}\beta_2^{t-k}g_k^{\odot 2}$ has correction $\hat{v}_t = v_t/(1-\beta_2^t)$. This removes zero-initialization bias under stationary moments; it does not make $\widehat{m}_t/(\sqrt{\widehat{v}_t}+\epsilon)$ an unbiased gradient or remove nonstationarity during training.

Step 2: separate the two sources of parameter motion. Writing $d_t = \widehat{m}_t/(\sqrt{\widehat{v}_t} + \epsilon)$, decoupled decay uses

$$w_{t+1} = w_t - \eta d_t - \eta\lambda w_t = (1 - \eta\lambda)w_t - \eta d_t.$$

The handout can instead denote $\eta\lambda$ by a single decay fraction λ'. If decay were applied to the *already updated* vector, the result would be

$$(1 - \eta\lambda)(w_t - \eta d_t) = (1 - \eta\lambda)w_t - \eta d_t + \eta^2\lambda d_t,$$

which contains an extra term. Coupled L_2 regularization instead feeds $g_t + \lambda w_t$ into Adam's moment estimates and coordinatewise normalization, so it is not equivalent to decoupled decay.

3.3 Neural Networks as Linearized Models

Deeper explanation: The Jacobian explains lazy training and feature learning (p. 346)

To understand how optimizers behave on the complex, non-convex loss landscapes of neural networks, it is useful to adopt a simplified, linear perspective.

Perspective 1: Feature Extraction

A deep neural network can be viewed as a two-stage model:

1. **Feature Extractor**: The initial non-linear layers of the network, $h(\vec{x})$, transform the input data $\vec{x}$ into a high-level feature representation.
2. **Linear Model**: The final layer of the network acts as a simple linear model (e.g., a classifier or regressor) that operates on these learned features.

The key idea is that the network *learns* the optimal features from the data, rather than having them pre-specified.

This perspective gives rise to the practical technique of **linear probing**. For a new task, one can take a pre-trained network, freeze the weights of the feature extraction layers, and only re-train the final linear layer. This is an efficient way to adapt a powerful model to new data.

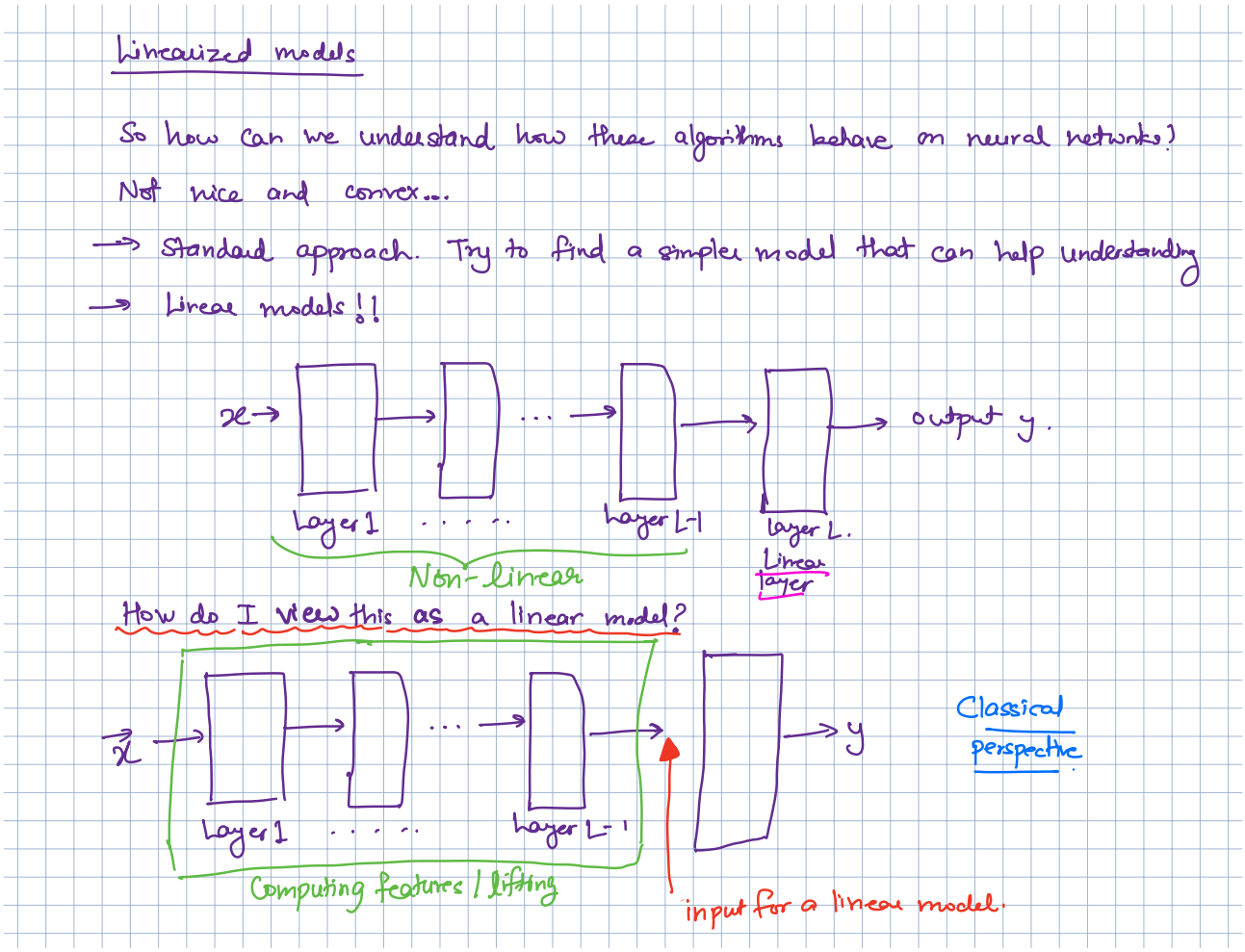


Figure 3.3: Feature extractor followed by a readout. The feature map transforms the input before a linear readout acts on it. Freezing the feature extractor yields a linear learning problem; updating it changes the representation itself. Original annotated source excerpt; Gireeja Ranade, faculty handout 03, p. 7.

Perspective 2: Taylor Expansion

Another way to linearize a network is to consider its first-order Taylor expansion with respect to its parameters, $\vec{\theta}$. Let the network function be $f(\vec{x}, \vec{\theta})$. If we take a small step from $\vec{\theta}_0$ to $\vec{\theta}_0 + \Delta\vec{\theta}$, the network's output changes approximately as:

$$f(\vec{x}, \vec{\theta}_0 + \Delta\vec{\theta}) \approx f(\vec{x}, \vec{\theta}_0) + \langle \nabla_{\vec{\theta}} f(\vec{x}, \vec{\theta}_0), \Delta\vec{\theta} \rangle \tag{3.9}$$

This is a linear model in the parameter update $\Delta\vec{\theta}$. The "features" of this model are the components of the gradient of the network's output with respect to its parameters, $\nabla_{\vec{\theta}} f(\vec{x}, \vec{\theta}_0)$.

This linear approximation is valid under the **lazy training assumption**, which posits that the parameters do not move far from their initialization during training (i.e., $||\Delta\vec{\theta}||$ is small). This regime is often observed in very wide, over-parameterized networks. This perspective

is the foundation for the theory of the **Neural Tangent Kernel (NTK)**, which we will explore in future lectures.

Worked derivation: the linearized model and kernel dynamics

Lecture basis: faculty handout 03, pp. 8–9. The steps below expand the lecture argument.

Step 1: stack predictions and Jacobians. For scalar predictions on n examples, let $f(\theta) \in \mathbb{R}^n$ and $J_0 \in \mathbb{R}^{n \times p}$ have rows $\nabla_\theta f_i(\theta_0)^\top$. Taylor expansion gives

$$f(\theta_0 + \delta) = f(\theta_0) + J_0\delta + O(\|\delta\|^2).$$

Freeze J_0 to define a linearized training problem. For $L = \|f - y\|^2/(2n)$ and $r = f - y$,

$$\begin{aligned} \nabla_\theta L &= \frac{1}{n} J_0^\top r, \\ \dot{\theta} &= -\frac{1}{n} J_0^\top r, \\ \dot{r} &= J_0\dot{\theta} = -\frac{1}{n} J_0 J_0^\top r. \end{aligned}$$

Step 2: solve the dynamics in example space. Define $K = J_0 J_0^\top$. For any a, $a^\top K a = \|J_0^\top a\|^2 \geq 0$, so $K = Q\Lambda Q^\top$ has nonnegative eigenvalues. In coordinates $c = Q^\top r$,

$$\begin{aligned} \dot{c}_i &= -\lambda_i c_i/n, \\ c_i(t) &= e^{-\lambda_i t/n} c_i(0), \\ r(t) &= Q\,\mathrm{diag}(e^{-\lambda_i t/n}) Q^\top r(0). \end{aligned}$$

Positive kernel modes decay; zero modes cannot be fitted by these frozen tangent features. A nonlinear network has $K(\theta_t)$ that generally changes with time. The fixed-kernel exponential solution is exact for the linearized model, not an unconditional claim about finite-width neural networks.

Lecture 4

Optimizers

Course sources: lecture_04; 2025-09-11 transcript.

4.1 Linear Perspectives on Neural Networks

Deeper explanation: The Jacobian explains lazy training and feature learning (p. 346)

We can analyze neural networks through a linear lens in two primary ways. The first perspective, discussed previously, treats the network as a feature extractor followed by a final linear layer.

A Second Perspective: Linearization via Taylor Expansion

A second linear perspective involves analyzing the network as a function of its parameters. Let a neural network be represented by a function $f(x, \theta)$, where x is the input data and $\theta \in \mathbb{R}^m$ are the network parameters (weights and biases).

We can analyze how the network's output changes with a small perturbation $\Delta\theta$ to the parameters using a Taylor expansion around the current parameters θ:

$$f(x, \theta + \Delta\theta) = f(x, \theta) + \langle \nabla_\theta f(x, \theta), \Delta\theta \rangle + \frac{1}{2}\Delta\theta^T H \Delta\theta + \ldots \tag{4.1}$$

where $\nabla_\theta f$ is the gradient of the function with respect to its parameters, and H is the Hessian matrix of second derivatives.

By ignoring the higher-order terms, we obtain a first-order or linear approximation of the network's behavior:

$$f(x, \theta + \Delta\theta) \approx f(x, \theta) + \langle \nabla_\theta f(x, \theta), \Delta\theta \rangle \tag{4.2}$$

The Lazy Training Assumption This linear approximation is only valid when the change in parameters, $\Delta\theta$, is small. The **lazy training assumption** posits that during training, the network's weights move by very small amounts. The final trained parameters θ_{final} remain very close to the initial parameters θ_{initial}. This assumption, often observed in practice for very wide networks, justifies the use of this linear model.

Gradients as Features In the linear term $\langle \nabla_\theta f, \Delta\theta \rangle$, the components of the gradient vector $\nabla_\theta f$ act as coefficients that multiply the changes in parameters $\Delta\theta_j$. By analogy to

standard linear models ($y = Xw$), we can think of the entries of the gradient, $\frac{\partial f}{\partial \theta_j}$, as the **features** of the network with respect to its parameters. This perspective is foundational to the theory of the **Neural Tangent Kernel (NTK)**, which analyzes the training dynamics of infinitely wide neural networks.

4.2 Revisiting Descent Algorithms

Deeper explanation: Preconditioning is a change in the local ruler (p. 373)

This linear perspective provides a new lens through which to view optimization algorithms like Adam and Gradient Descent.

The Core Tension in Optimization

When training a model, a fundamental tension exists:

- We want to take **large steps** to converge to a minimum quickly.
- We must take **small steps** to ensure the local linear approximation of the loss landscape, upon which gradient-based methods rely, remains valid.

Taking a step that is too large can invalidate the gradient information, potentially leading to an increase in the loss.

Optimization as a Constrained Problem

A principled way to resolve this tension is to frame the step selection as a constrained optimization problem. We seek to find the step $\Delta\theta$ that maximally decreases the linearized loss function, subject to a constraint on the size of the step.

$$\Delta\theta^* = \underset{\|\Delta\theta\| \leq \eta}{\arg\min} \quad \mathcal{L}(\theta_i + \Delta\theta) \tag{4.3}$$

Using the first-order approximation for the loss $\mathcal{L}$, this becomes:

$$\Delta\theta^* \approx \underset{\|\Delta\theta\| \leq \eta}{\arg\min} \quad \mathcal{L}(\theta_i) + \left\langle \frac{1}{N} \sum_{l=1}^{N} \nabla_\theta \mathcal{L}(x_l, \theta) \Big|_{\theta=\theta_i}, \Delta\theta \right\rangle \tag{4.4}$$

Since $\mathcal{L}(\theta_i)$ is constant with respect to $\Delta\theta$, we can drop it. Let $\nabla_\theta \mathcal{L}_{\text{batch}} = \frac{1}{N} \sum_{l=1}^{N} \nabla_\theta \mathcal{L}(x_l, \theta_i)$. The problem simplifies to:

$$\Delta\theta^* = \underset{\|\Delta\theta\| \leq \eta}{\arg\min} \quad \langle \nabla_\theta \mathcal{L}_{\text{batch}}, \Delta\theta \rangle \tag{4.5}$$

The solution depends on the choice of norm used for the constraint.

Deriving Adam/SignSGD via the Infinity Norm (L_∞) Let's constrain the step size using the L_∞ norm: $\|\Delta\theta\|_\infty \leq \eta$. This is equivalent to constraining each component individually: $|\Delta\theta_j| \leq \eta$ for all j.

The objective function is $\sum_j (\nabla_\theta \mathcal{L}_{\text{batch}})_j \cdot (\Delta\theta_j)$. Since the objective and constraints are decoupled for each component j, we can solve for each $\Delta\theta_j$ independently:

$$(\Delta\theta_j)^* = \underset{|\Delta\theta_j| \leq \eta}{\arg\min} \quad (\nabla_\theta \mathcal{L}_{\text{batch}})_j \cdot (\Delta\theta_j) \tag{4.6}$$

To minimize this product, we should choose $\Delta\theta_j$ to have the opposite sign of its corresponding gradient component and the largest possible magnitude, which is η.

- If $(\nabla_\theta \mathcal{L}_{\text{batch}})_j > 0$, we choose $\Delta\theta_j = -\eta$.
- If $(\nabla_\theta \mathcal{L}_{\text{batch}})_j < 0$, we choose $\Delta\theta_j = +\eta$.

This can be written concisely as:

$$\Delta\theta_j^* = -\eta \cdot \text{sign}\left((\nabla_\theta \mathcal{L}_{\text{batch}})_j\right) \tag{4.7}$$

This update rule is exactly **SignSGD**. Adam can be viewed as a smoothed version of SignSGD, where momentum is used to average the sign of the gradients over time. This shows that Adam is, in a sense, the optimal descent algorithm under an L_∞ constraint on the step size.

Deriving Gradient Descent via the Euclidean Norm (L_2) If we instead constrain the step size using the L_2 norm, $\|\Delta\theta\|_2 \leq \eta$, the problem becomes:

$$\Delta\theta^* = \underset{\|\Delta\theta\|_2 \leq \eta}{\arg\min} \quad \langle \nabla_\theta \mathcal{L}_{\text{batch}}, \Delta\theta \rangle \tag{4.8}$$

By the Cauchy-Schwarz inequality, the inner product $\langle u, v \rangle$ is minimized when the vectors u and v are anti-parallel (point in opposite directions). Therefore, the optimal step $\Delta\theta^*$ must be in the opposite direction of the gradient $\nabla_\theta \mathcal{L}_{\text{batch}}$. To satisfy the constraint, we take the largest possible step in that direction:

$$\Delta\theta^* = -\eta \frac{\nabla_\theta \mathcal{L}_{\text{batch}}}{\|\nabla_\theta \mathcal{L}_{\text{batch}}\|_2} \tag{4.9}$$

This is a normalized gradient descent step. The update $\theta_{i+1} = \theta_i + \Delta\theta^*$ is standard gradient descent with a learning rate that depends on η and the gradient norm.

Worked derivation: why the constraint determines the descent direction

Lecture basis: faculty handout 04, pp. 3–4. The steps below expand the lecture argument.

Step 1: isolate the local problem. Taylor expansion gives $L(w+\Delta) = L(w) + g^\top \Delta + O(\|\Delta\|^2)$, where $g = \nabla L(w)$. The linear term can decrease without bound unless the step is constrained or penalized.

Step 2: solve the coordinatewise constraint. For $\|\Delta\|_\infty \le \rho$, each coordinate obeys

$$g_i \Delta_i \ge -|g_i||\Delta_i| \ge -\rho|g_i|.$$

Summing yields $g^\top \Delta \ge -\rho\|g\|_1$. Equality is attained by $\Delta_i = -\rho\,\mathrm{sign}(g_i)$; a zero-gradient coordinate may take any feasible value. This is the SignSGD direction.

Step 3: solve the Euclidean ball. Cauchy–Schwarz gives

$$g^\top \Delta \ge -\|g\|_2\|\Delta\|_2 \ge -\rho\|g\|_2.$$

For $g \neq 0$, equality requires $\Delta = -\rho g/\|g\|_2$. This is a *normalized* gradient step of fixed length, not ordinary GD with a constant learning rate.

Step 4: recover ordinary GD from a quadratic penalty. For $\lambda > 0$, minimize $g^\top \Delta + \lambda\|\Delta\|_2^2$. Completing the square,

$$g^\top \Delta + \lambda\|\Delta\|^2 = \lambda\left\|\Delta + \frac{g}{2\lambda}\right\|^2 - \frac{\|g\|^2}{4\lambda}.$$

Thus $\Delta = -g/(2\lambda)$. Equivalently a penalty $\|\Delta\|^2/(2\eta)$ gives $\Delta = -\eta g$. The local linear model chooses a direction; a smoothness bound or line search is needed to guarantee that the actual nonlinear loss decreases.

4.3 Standardization and Normalization

Deeper explanation: Standardizing inputs and preserving the prediction problem (p. 375)

Beyond the choice of optimizer, data preprocessing is crucial for successful training.

Why Standardize Data?

Standardizing input data (e.g., to have zero mean and unit variance) is a standard practice in machine learning for several reasons:

1. **Equal Feature Contribution:** It places all features on a similar scale, preventing features with large magnitudes from dominating the learning process and the loss function.

2. **Numerical Stability:** It avoids numerical precision issues that can arise from very large or very small values in computations. It also improves the conditioning of matrices involved in the optimization.

3. **Effective Use of Non-linearities:** Neural network activation functions are most sensitive or "interesting" in a specific range, typically around zero. Standardizing data ensures that the inputs to the first layer fall into this range, allowing the network to leverage the full expressive power of its non-linearities.

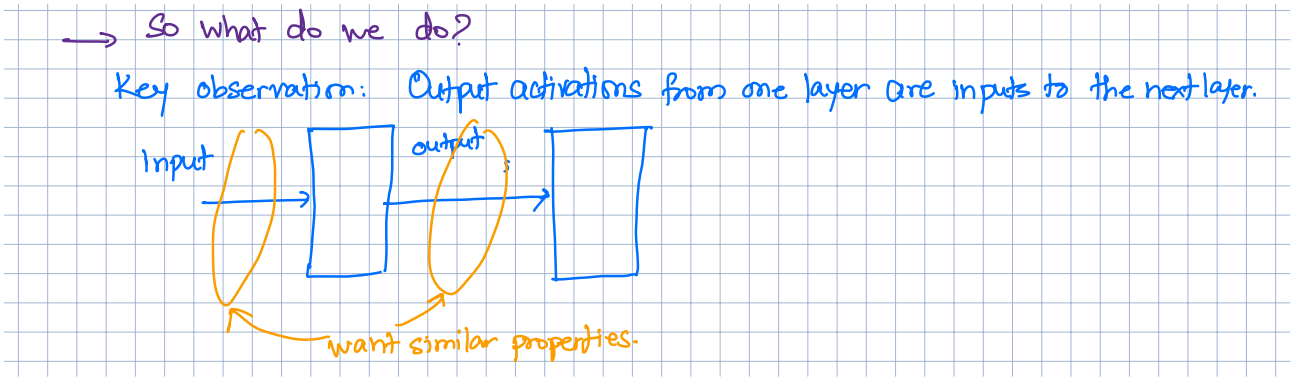


Figure 4.1: Keeping activation scales comparable between layers. An output activation from one layer becomes an input to the next. The sketch motivates controlling the distribution passed between layers, rather than standardizing only the original input. Initialization and normalization address this issue in different ways. Original annotated source excerpt; Gireeja Ranade, faculty handout 04, p. 8.

The Danger of Unstandardized Data: A ReLU Example

Consider a simple neuron with a ReLU activation: $h = \max(0, wx + b)$. The non-linearity (the "elbow") occurs at $x = -b/w$.

Suppose we initialize weights w and biases b from a standard normal distribution, $w, b \sim \mathcal{N}(0, 1)$. The location of the elbow, $-b/w$, follows a **Cauchy distribution**. The PDF of the standard Cauchy distribution is $p(z) = \frac{1}{\pi(1+z^2)}$.

Key properties of the Cauchy distribution:

- Most of its probability mass is concentrated around zero. For instance, $\sim 90\%$ of the mass lies between -7 and 7.
- It has heavy tails, meaning there is a non-trivial probability of sampling very large values. Its mean is undefined and its variance is undefined (its second moment diverges).

Now, suppose our input data lies in a range far from zero, e.g., $x \in [100, 200]$. The probability of an elbow $-b/w$ landing in this range is extremely small (e.g., $\approx 0.159\%$).

Consequence: For the vast majority of neurons, the pre-activation $wx + b$ will be either always positive or always negative for all data points in the training set.

- If $wx + b > 0$ for all x, the ReLU acts as a linear function: $h(x) = wx + b$.
- If $wx + b < 0$ for all x, the ReLU acts as a zero function: $h(x) = 0$.

This effectively "kills" the non-linearity and drastically reduces the network's expressive power. Mathematically, the matrix of hidden activations for all data points becomes **very low-rank**. If all neurons become affine functions $h_i(x) = \alpha_i x + \beta_i$, the resulting feature matrix has a rank of at most 2. This severely constrains the function that the next layer can learn.

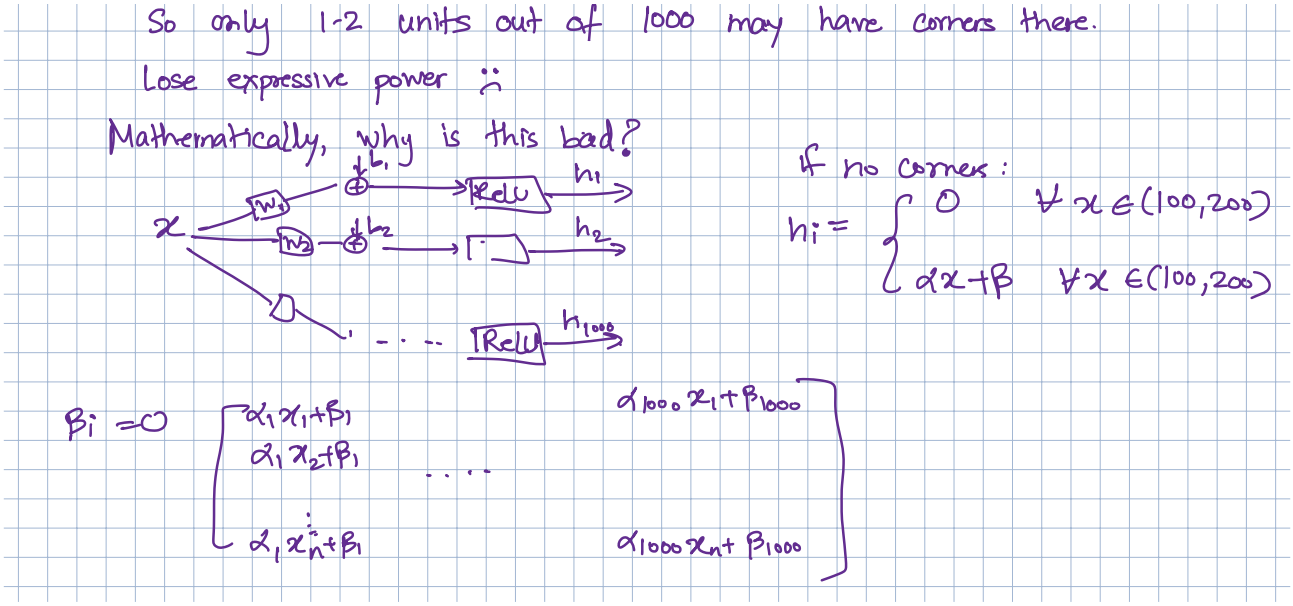


Figure 4.2: An inactive ReLU removes a useful branch. A badly placed activation threshold can suppress a neuron throughout the observed input range. Trace which branch becomes inactive and how this reduces the functions available to the readout. Original annotated source excerpt; Gireeja Ranade, faculty handout 04, p. 7.

Worked derivation: the Gaussian ratio and missing ReLU knots

Lecture basis: faculty handout 04, pp. 6–7. The steps below expand the lecture argument.

Step 1: change variables in the two Gaussian densities. Let b, w be independent standard normals and $K = -b/w$. With $b = -kw$, the absolute Jacobian for $(k, w) \mapsto (b, w)$ is $|w|$. Thus

$$\begin{aligned} p_K(k) &= \int_{-\infty}^{\infty} \frac{|w|}{2\pi} \exp\left[-\frac{(1+k^2)w^2}{2}\right] dw \\ &= \frac{1}{\pi} \int_0^{\infty} we^{-(1+k^2)w^2/2} dw \\ &= \frac{1}{\pi(1+k^2)}. \end{aligned}$$

In the last step substitute $u = (1+k^2)w^2/2$, so $du = (1+k^2)w\,dw$. This is a standard Cauchy density, symmetric about zero but without a finite mean or variance.

Step 2: calculate the useful-knot probability. For an input interval $[a, b]$,

$$P(a \le K \le b) = \frac{\arctan b - \arctan a}{\pi}.$$

For $[100, 200]$ this is approximately 0.00159, whereas for $[-1, 1]$ it is $1/2$. With M independent units, the probability of no knot inside an interval of probability p is $(1-p)^M$.

Step 3: relate absent knots to a rank bound. If no unit changes activation state on the data interval, each hidden feature is affine there: $h_j(x) = a_j x + c_j$, including the zero function. On sample inputs $x_1, \dots, x_n$, the hidden feature matrix is

$$H = \begin{bmatrix} x_1 & 1 \\ \vdots & \vdots \\ x_n & 1 \end{bmatrix} \begin{bmatrix} a_1 & \cdots & a_M \\ c_1 & \cdots & c_M \end{bmatrix}.$$

Its rank is at most two regardless of width. Standardization helps the initialized nonlinearities operate in the data region; it is not merely a cosmetic change of units.

4.4 Weight Initialization Strategies

Deeper explanation: Forward and backward second-moment calculations (p. 377)

Deeper explanation: Why identical hidden units do not diversify by themselves (p. 377)

Deeper explanation: How many random ReLU breakpoints fall in the useful input range? (p. 380)

Standardizing the initial input data is only the first step. We need to ensure that the inputs to *every* layer remain well-behaved, at least at initialization. This is the role of weight initialization.

Goal: Initialize weights such that the statistical properties (mean and variance) of the activations are preserved as they propagate through the network.

Why Not Initialize All Weights to Zero?

Identically initialized hidden units can receive identical gradients and fail to learn distinct features. In a multilayer ReLU network with all weights and biases zero and a zero derivative chosen at the kink, hidden weight gradients vanish. This does not mean every model or every bias gradient vanishes: zero initialization is valid for ordinary linear regression.

Xavier (Glorot) Initialization

Consider the pre-activation z of a neuron in a fully-connected layer:

$$z = \sum_{i=1}^{d_{\text{in}}} w_i h_i + b \tag{4.10}$$

where h_i are the outputs from the previous layer and d_{in} is the "fan-in" (number of inputs).

Assumptions:

- The inputs h_i are standardized: $E[h_i] = 0$ and $Var(h_i) = 1$.
- The weights w_i are initialized with $E[w_i] = 0$.
- Weights w_i and inputs h_i are independent.
- The bias b is initialized to 0.

We want the output pre-activation z to also have zero mean and unit variance.

- **Mean:** $E[z] = \sum E[w_i h_i] + E[b] = \sum E[w_i]E[h_i] + 0 = 0$. The mean is preserved.
- **Variance:**

$$\begin{aligned} Var(z) = E[z^2] - (E[z])^2 &= E\left[\left(\sum_{i=1}^{d_{\text{in}}} w_i h_i\right)^2\right] \\ &= \sum_{i=1}^{d_{\text{in}}} E[w_i^2 h_i^2] \quad \text{(cross-terms are zero due to independence and zero mean)} \\ &= \sum_{i=1}^{d_{\text{in}}} E[w_i^2]E[h_i^2] = \sum_{i=1}^{d_{\text{in}}} Var(w_i)Var(h_i) \end{aligned}$$

$$= \sum_{i=1}^{d_{\text{in}}} Var(w_i) \cdot 1$$

Assuming all weights are sampled from the same distribution, $Var(w_i) = Var(w)$. Then $Var(z) = d_{\text{in}} \cdot Var(w)$. To achieve $Var(z) = 1$, we must set the variance of the weights to:

$$Var(w) = \frac{1}{d_{\text{in}}} \tag{4.11}$$

This is the **Xavier initialization** rule. Weights are typically drawn from a normal or uniform distribution with mean 0 and this variance. Some versions also consider the fan-out to balance forward and backward propagation dynamics, using $Var(w) = \frac{2}{d_{\text{in}}+d_{\text{out}}}$.

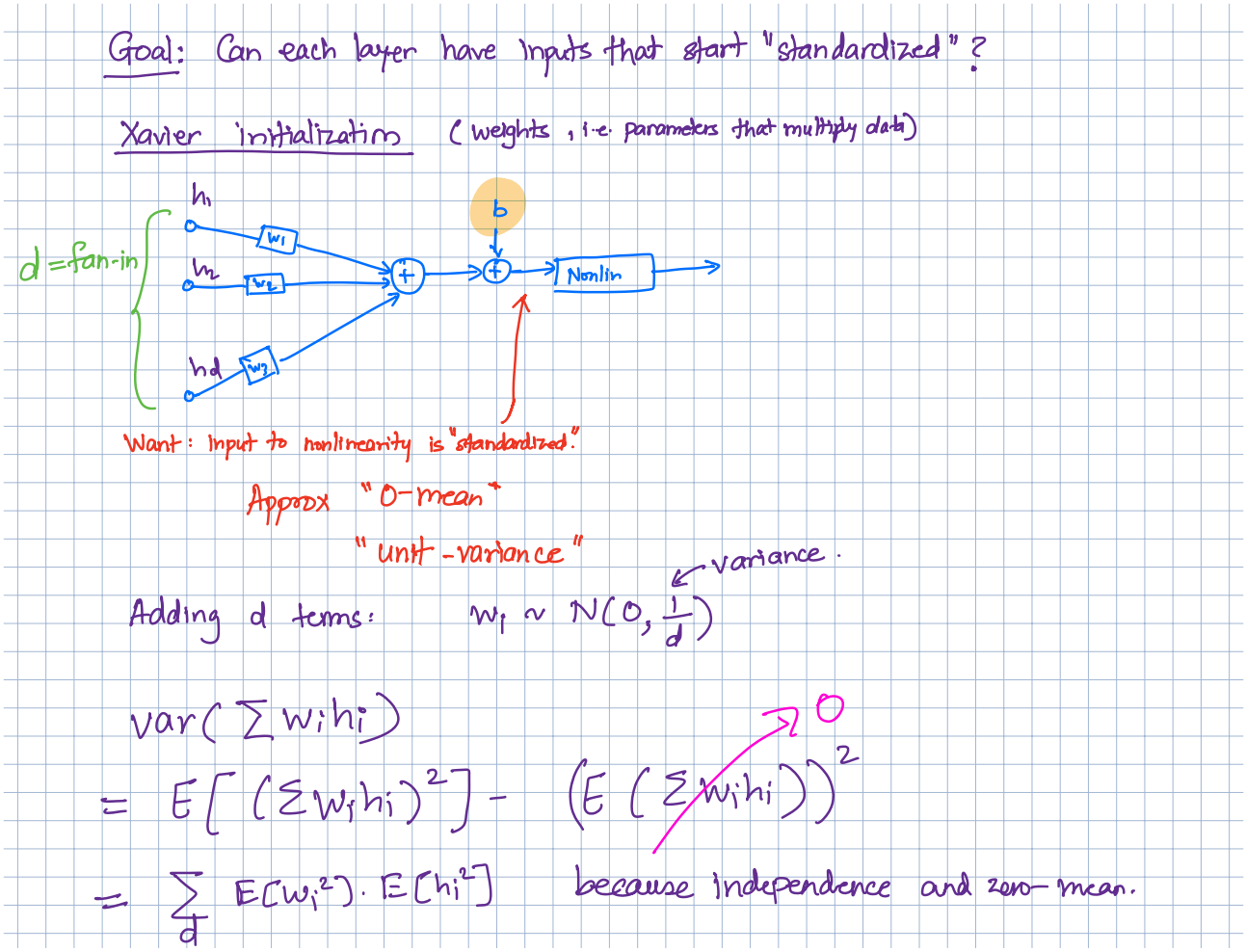


Figure 4.3: Variance propagation through a neuron. The fan-in diagram identifies the terms entering a weighted sum. Under the independence and centering assumptions used in the derivation, their variances add, motivating inverse-fan-in weight variance. Original annotated source excerpt; Gireeja Ranade, faculty handout 04, p. 9.

He Initialization

Xavier initialization was derived assuming a linear activation function. However, non-linearities like ReLU alter the statistics of the activations. A ReLU activation, for zero-mean symmetric inputs, will set approximately half of its outputs to zero. This effectively halves the number of active inputs to the next layer.

The "effective fan-in" for a layer following a ReLU layer is $d_{\text{in}}/2$. To preserve unit variance, we must compensate for this. The variance calculation becomes: $Var(z) \approx (d_{\text{in}}/2) \cdot Var(w)$. To make $Var(z) = 1$, we now need:

$$Var(w) = \frac{2}{d_{\text{in}}} \tag{4.12}$$

This is the **He initialization** rule, which is the standard and recommended initialization for networks using ReLU or its variants (like Leaky ReLU). It accounts for the non-saturating, one-sided nature of these activation functions and has been shown to enable the training of much deeper networks.

Worked derivation: Xavier and He scaling from second moments

Lecture basis: faculty handout 04, pp. 9–10. The steps below expand the lecture argument.

Step 1: compute the preactivation scale. Let $z_i = \sum_{j=1}^{d_{\text{in}}} W_{ij} x_j$, with zero-mean independent weights of variance s^2, independent of x. Conditional on x,

$$\begin{aligned} \mathbb{E}_W[z_i \mid x] &= 0, \\ \mathbb{E}_W[z_i^2 \mid x] &= \sum_{j,k} x_j x_k \mathbb{E}[W_{ij} W_{ik}] \\ &= s^2 \sum_j x_j^2. \end{aligned}$$

The off-diagonal terms vanish because the two weights are independent and centered. If $\mathbb{E}[x_j^2] = q$, averaging over inputs gives $\mathbb{E}[z_i^2] = d_{\text{in}} s^2 q$. Preserving forward second moments in a linear layer calls for $s^2 = 1/d_{\text{in}}$.

Step 2: account for the nonlinearity. If z_i has a symmetric distribution about zero,

$$\begin{aligned} \mathbb{E}[\text{ReLU}(z_i)^2] &= \mathbb{E}[z_i^2 \mathbf{1}_{z_i > 0}] \\ &= \tfrac{1}{2}\mathbb{E}[z_i^2] = \tfrac{1}{2} d_{\text{in}} s^2 q. \end{aligned}$$

Thus He scaling chooses $s^2 = 2/d_{\text{in}}$. This halves the *second moment*, not the variance: the ReLU output usually has positive mean, so its variance is its second moment minus that mean squared.

Step 3: compare the backward condition. For a linear layer, the input gradient is $g_x = W^\top g_z$. Under the corresponding independence approximation,

$$\mathbb{E}[(g_x)_j^2] = d_{\text{out}} s^2 \mathbb{E}[(g_z)_i^2].$$

Backward preservation asks for $s^2 = 1/d_{\text{out}}$. When the widths differ, forward and backward conditions cannot both hold exactly. Xavier's $2/(d_{\text{in}} + d_{\text{out}})$ is a compromise for approximately linear activations. These calculations describe initialization under stated independence assumptions; training introduces correlations.

Lecture 5

Optimizers: insights from induced matrix norms

Course sources: lecture_05; 2025-09-16 transcript.

5.1 Motivation: Why Better Optimizers?

Deeper explanation: Preconditioning is a change in the local ruler (p. 373)

The study of optimizers and initialization schemes is not merely about minor details; these choices have a profound impact on the performance and feasibility of training deep neural networks.

The Practical Importance of Optimization

The primary drivers for developing better optimizers are practical and economic:

- **Faster Training:** A better optimizer leads to faster convergence.
- **Cost Reduction:** Faster training reduces computational resource usage (e.g., GPU time), which translates directly to lower financial costs.

Similarly, principled initialization methods like Xavier and He initialization were significant breakthroughs. Before their development, a common practice was to initialize all weights from a standard normal distribution, $\mathcal{N}(0, 1)$. This often led to networks that would not train due to vanishing or exploding gradients. Figuring out the right initialization was akin to an intractable hyperparameter search over millions of parameters. These methods provided robust rules of thumb that are now standard in deep learning libraries.

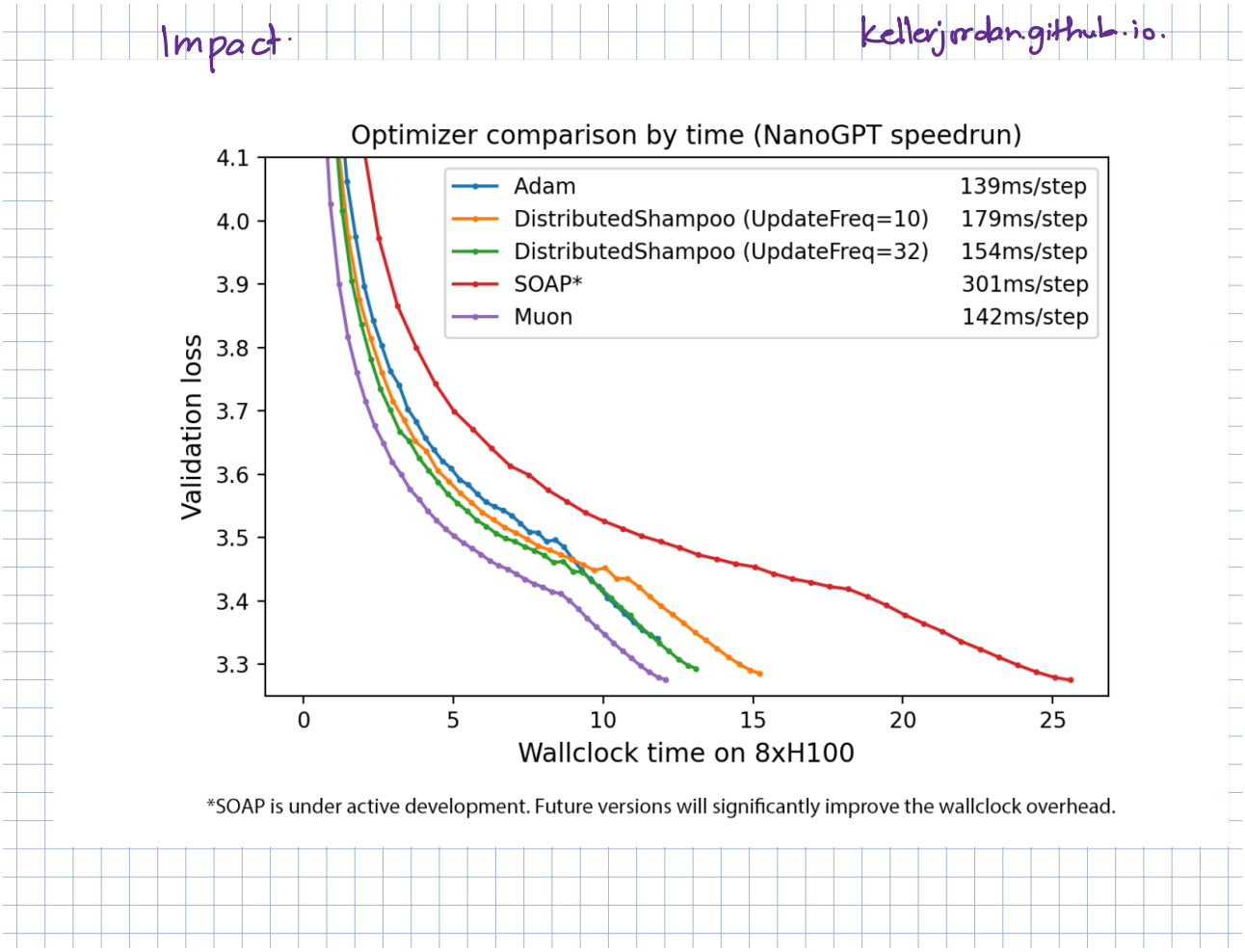


Figure 5.1: Optimizer progress measured in wall-clock time. These are the empirical comparisons discussed in Fall 2025. Read the horizontal axis as computation time rather than an interchangeable count of updates; the result depends on the task, implementation, and experimental setup shown. Original annotated source excerpt; Gireeja Ranade, faculty handout 06, p. 5.

The State of the Art and Its Challenges

Currently, the default optimizer for many deep learning applications is **AdamW**. AdamW is a variant of Adam that decouples weight decay from the adaptive learning rate mechanism, leading to better performance. The deep learning community has invested immense effort in fine-tuning the hyperparameters for AdamW on standard architectures, making it a very strong baseline.

This creates a challenge for new optimizers: a new method must outperform a highly-tuned, well-understood incumbent. Comparing a new optimizer with default hyperparameters against a fine-tuned AdamW is often not a fair comparison.

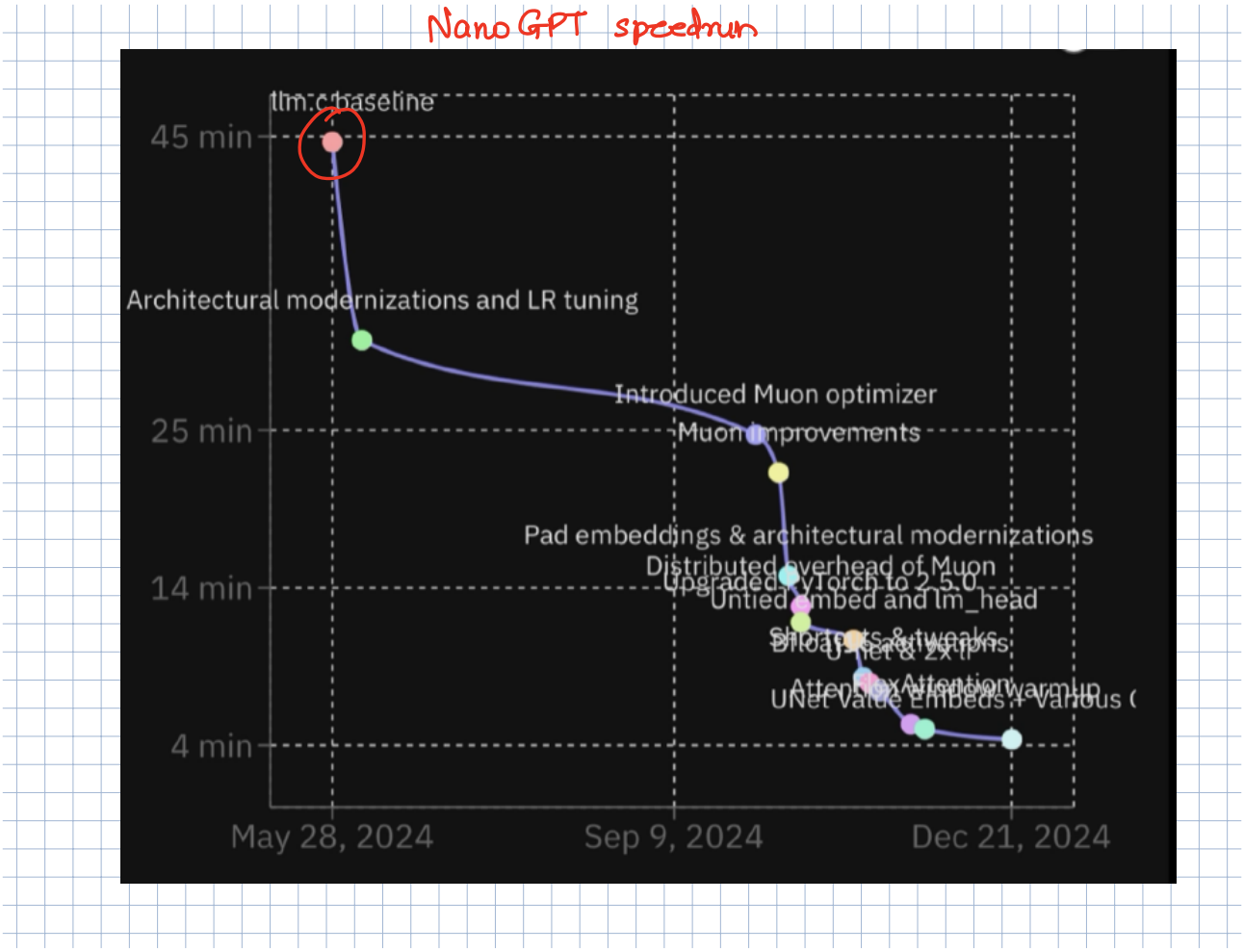


Figure 5.2: The course snapshot of training-speed improvements. This historical timeline motivates the course emphasis on optimization and implementation together. Its speed records describe the snapshot in the handout, not a claim about current records. Original annotated source excerpt; Gireeja Ranade, faculty handout 06, p. 6.

Lazy Training: A Case for Better Optimizers

One phenomenon observed when using optimizers like Adam is **lazy training**. This is an empirical observation where, throughout the training process, the network's parameters change very little from their random initialization, especially in the hidden layers. While the final layer might adapt, the core feature extractors remain mostly static.

This observation can be interpreted in two ways:

1. **The "Magic" Perspective (Kernel/Feature Learning):** Perhaps the random initialization is already good enough. The network is essentially using random, non-linear feature projections, and learning only a linear model (the last layer) on top of these features is sufficient for the task. This connects to ideas like Neural Tangent Kernels (NTK).

2. **The "Inefficiency" Perspective:** It seems highly improbable that a random initialization is optimal. If the parameters are not moving, we are likely not utilizing the full expressive power of the deep network. This suggests that our current optimizers may be stuck in a regime where they cannot effectively move the parameters, pointing to a need for better optimization methods.

5.2 A Local Linear Perspective on Optimization

Deeper explanation: Preconditioning is a change in the local ruler (p. 373)

To develop a more principled approach to creating optimizers, we can analyze the gradient descent step from a local, linearized perspective. This viewpoint follows the work of Bernstein & Newhouse (2024), which builds upon a rich history of prior research.

Let θ be the model parameters, $\vec{x}$ be the data, and $\mathcal{L}(\vec{x}, \theta)$ be the loss function. We can approximate the loss at a nearby point $\theta_t + \Delta\theta$ using a first-order Taylor expansion around the current parameters θ_t:

$$\mathcal{L}(\vec{x}, \theta_t + \Delta\theta) \approx \mathcal{L}(\vec{x}, \theta_t) + \frac{\partial \mathcal{L}}{\partial \theta}\Big|_{\theta=\theta_t}^{T} \Delta\theta$$

This is often written more compactly as:

$$\mathcal{L}(\theta_t + \Delta\theta) \approx \mathcal{L}(\theta_t) + \langle \nabla_\theta \mathcal{L}(\theta_t), \Delta\theta \rangle$$

The fundamental tension in gradient-based optimization is:

- We want to take a **small step** ($\Delta\theta$) so that the local linear approximation remains valid.
- We want to take a **large step** to converge quickly.

This can be formulated as a constrained optimization problem for a single update step. We want to find the step $\Delta\theta$ that minimizes our linear approximation of the loss, while ensuring the step size is not too large.

$$\Delta\theta^* = \underset{||\Delta\theta|| \leq \eta}{\operatorname{argmin}} \left(\mathcal{L}(\theta_t) + \langle \nabla_\theta \mathcal{L}(\theta_t), \Delta\theta \rangle \right)$$

Here, η is the step size (learning rate), and the choice of norm $||\cdot||$ defines the geometry of the constraint region. Since $\mathcal{L}(\theta_t)$ is a constant with respect to the optimization variable $\Delta\theta$, we can simplify the problem to:

$$\Delta\theta^* = \underset{||\Delta\theta|| \leq \eta}{\operatorname{argmin}} \langle \nabla_\theta \mathcal{L}(\theta_t), \Delta\theta \rangle$$

This framework provides a "recipe" for generating optimizers: choose a norm, solve the resulting optimization problem, and the solution defines the update rule.

5.3 A Recipe for Optimizers from Norms

Deeper explanation: Preconditioning is a change in the local ruler (p. 373)

Choice 1: The ℓ_∞ Norm (Sign SGD)

Let's choose the infinity norm for our constraint: $||\Delta\theta||_\infty \leq \eta$. This constraint requires that the maximum absolute value of any component of $\Delta\theta$ is at most η. Geometrically, this defines a hypercube centered at the origin.

The optimization problem is:

$$\underset{||\Delta\theta||_\infty \leq \eta}{\operatorname{argmin}} \langle \nabla_\theta \mathcal{L}(\theta_t), \Delta\theta \rangle = \underset{||\Delta\theta||_\infty \leq \eta}{\operatorname{argmin}} \sum_j (\nabla_\theta \mathcal{L}(\theta_t))_j (\Delta\theta)_j$$

This is a **linear program**: a linear objective with linear constraints. The solution to a linear program on a bounded domain (like a hypercube) must lie at one of its vertices (corners). To minimize the inner product, we want to make each term $(\nabla_\theta \mathcal{L}(\theta_t))_j (\Delta\theta)_j$ as negative as possible. We achieve this by setting the sign of $(\Delta\theta)_j$ to be opposite to the sign of $(\nabla_\theta \mathcal{L}(\theta_t))_j$, and its magnitude to the maximum allowed value, η.

The solution is:

$$(\Delta\theta)_j = \begin{cases} +\eta & \text{if } (\nabla_\theta \mathcal{L}(\theta_t))_j < 0 \\ -\eta & \text{if } (\nabla_\theta \mathcal{L}(\theta_t))_j > 0 \end{cases}$$

This can be written compactly as:

$$\Delta\theta^* = -\eta \cdot \operatorname{sgn}(\nabla_\theta \mathcal{L}(\theta_t))$$

This update rule is known as **Sign SGD**. It forms the basis of the momentum update in the Adam optimizer. This perspective reveals that Adam's update rule is optimal under an ℓ_∞ geometry.

Choice 2: The ℓ_2 Norm (Gradient Descent)

Now, let's choose the more conventional Euclidean or ℓ_2 norm: $||\Delta\theta||_2 \leq \eta$. This constraint defines a hypersphere of radius η.

The problem is:

$$\underset{||\Delta\theta||_2 \leq \eta}{\operatorname{argmin}} \langle \nabla_\theta \mathcal{L}(\theta_t), \Delta\theta \rangle$$

Recall that the inner product can be expressed as $\langle \vec{x}, \vec{y} \rangle = ||\vec{x}||_2 ||\vec{y}||_2 \cos\phi$, where ϕ is the angle between the vectors. To minimize this value, we need $\cos\phi = -1$, which means the vectors must point in opposite directions (anti-aligned).

Therefore, the optimal direction for $\Delta\theta$ is exactly opposite to the gradient vector $\nabla_\theta \mathcal{L}(\theta_t)$. To satisfy the constraint $||\Delta\theta||_2 \leq \eta$, we move as far as possible in that direction, to the boundary of the hypersphere. The solution is:

$$\Delta\theta^* = -\eta \frac{\nabla_\theta \mathcal{L}(\theta_t)}{||\nabla_\theta \mathcal{L}(\theta_t)||_2}$$

This is **Normalized Gradient Descent**. The direction is identical to standard gradient descent, but the step magnitude is constant (η), regardless of the gradient's magnitude.

Alternative Formulation: Regularization An alternative, unconstrained formulation can be derived using a Lagrange multiplier, which is equivalent to adding a regularization term:

$$\underset{\Delta\theta}{\text{argmin}} \langle \nabla_\theta \mathcal{L}(\theta_t), \Delta\theta \rangle + \lambda ||\Delta\theta||_2^2$$

This objective function is convex in $\Delta\theta$. We can find the minimum by setting its gradient with respect to $\Delta\theta$ to zero:

$$\begin{aligned} \nabla_{\Delta\theta} \left(\langle \nabla_\theta \mathcal{L}(\theta_t), \Delta\theta \rangle + \lambda ||\Delta\theta||_2^2 \right) &= 0 \\ \nabla_\theta \mathcal{L}(\theta_t) + 2\lambda \Delta\theta &= 0 \\ \Delta\theta^* &= -\frac{1}{2\lambda} \nabla_\theta \mathcal{L}(\theta_t) \end{aligned}$$

Letting the step size be $\eta' = \frac{1}{2\lambda}$, this is exactly the standard **Gradient Descent** update. The constrained and regularized formulations generate the same family of solutions.

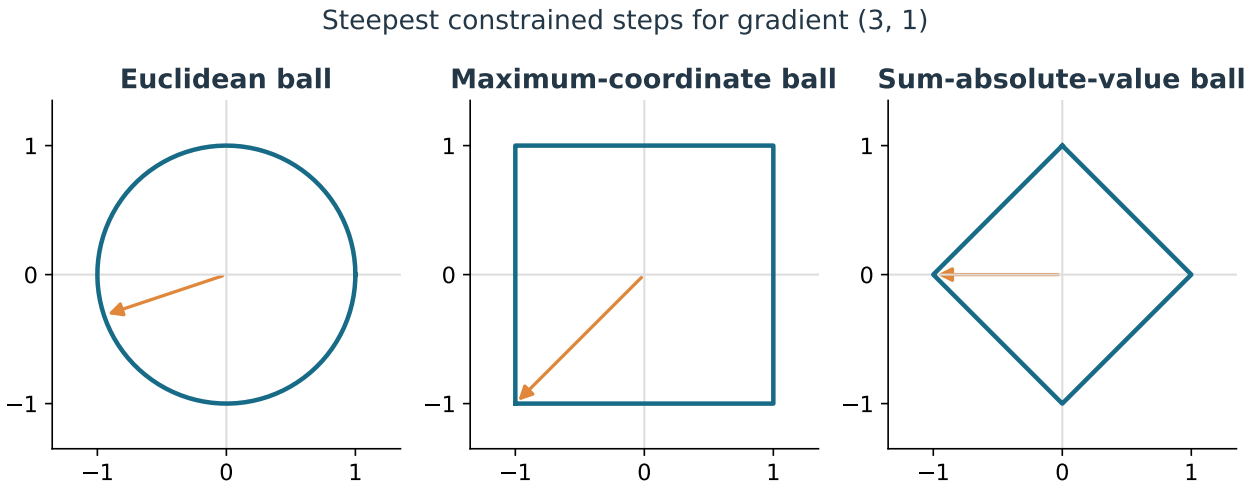


Figure 5.3: The same gradient meets three different unit balls. Euclidean, infinity, and one-norm constraints select different updates.

5.4 Incorporating Network Structure: Matrix-Based Optimization

Deeper explanation: Shampoo: connect the algebra to an adaptive algorithm (p. 373)

Deeper explanation: Nonuniqueness of the spectral-norm optimal step (p. 376)

So far, we have treated the parameters θ as a single, unstructured vector. However, neural network parameters are highly structured, consisting of weight matrices ($\mathbf{W}_l$) and bias vectors ($\vec{b}_l$) for each layer l. Scaling issues, as seen in initialization, primarily arise from the multiplicative interactions of the weight matrices.

This suggests we should reformulate our optimization problem to respect this structure. Let's focus on the weight matrices and define an update step $\Delta\mathbf{W}$ in the space of matrices.

$$\min_{||\Delta\mathbf{W}||\leq\eta} \langle\nabla_{\mathbf{W}}\mathcal{L}(\mathbf{W}), \Delta\mathbf{W}\rangle_F$$

The inner product for matrices is the Frobenius inner product: $\langle\mathbf{A}, \mathbf{B}\rangle_F = \text{trace}(\mathbf{A}^T\mathbf{B})$.

The Shampoo Optimizer and the Spectral Norm

What is a natural norm for a matrix that acts as a linear operator scaling vectors? A good candidate is the **spectral norm** (also known as the operator 2-norm), which measures the maximum possible scaling of a unit vector.

$$||\mathbf{A}||_2 = \sigma_{\max}(\mathbf{A}) = \max_{||\vec{x}||_2=1} ||\mathbf{A}\vec{x}||_2$$

where $\sigma_{\max}(\mathbf{A})$ is the largest singular value of $\mathbf{A}$.

Let's use the spectral norm for our constraint:

$$\operatorname*{argmin}_{||\Delta\mathbf{W}||_2\leq\eta} \langle\nabla_{\mathbf{W}}\mathcal{L}(\mathbf{W}), \Delta\mathbf{W}\rangle_F$$

Solving this problem yields an update rule related to the **Shampoo** optimizer.

Deriving the Shampoo Update Step

To solve the optimization problem, we want to find the matrix $\mathbf{B}$ with $||\mathbf{B}||_2 \leq 1$ that maximizes $\text{trace}(\mathbf{A}^T\mathbf{B})$, where $\mathbf{A} = \nabla_{\mathbf{W}}\mathcal{L}(\mathbf{W})$. The solution to the original minimization problem will then be $-\eta$ times this maximizing $\mathbf{B}$.

Use a compact SVD, retaining the positive singular modes (with a zero update when the gradient is zero). Let the Singular Value Decomposition (SVD) of $\mathbf{A}$ be $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^T$. The

trace term is $\text{trace}((\mathbf{U}\Sigma\mathbf{V}^T)^T\mathbf{B}) = \text{trace}(\mathbf{V}\Sigma^T\mathbf{U}^T\mathbf{B})$. Using the linearity of the trace and the outer product form of the SVD, $\mathbf{A}^T = \sum_i \sigma_i \vec{v}_i \vec{u}_i^T$:

$$\begin{aligned}
\text{trace}(\mathbf{A}^T\mathbf{B}) &= \text{trace}\left(\left(\sum_i \sigma_i \vec{v}_i \vec{u}_i^T\right)\mathbf{B}\right) \\
&= \sum_i \sigma_i \text{trace}(\vec{v}_i \vec{u}_i^T \mathbf{B}) && \text{(Linearity of trace)} \\
&= \sum_i \sigma_i \text{trace}(\vec{u}_i^T \mathbf{B} \vec{v}_i) && \text{(Cyclic property of trace)}
\end{aligned}$$

The term $\vec{u}_i^T\mathbf{B}\vec{v}_i$ is a scalar. By the Cauchy-Schwarz inequality and the definition of the spectral norm:

$$|\vec{u}_i^T\mathbf{B}\vec{v}_i| \leq ||\vec{u}_i||_2||\mathbf{B}\vec{v}_i||_2 \leq ||\vec{u}_i||_2||\mathbf{B}||_2||\vec{v}_i||_2 \leq 1 \cdot 1 \cdot 1 = 1$$

Therefore, $\text{trace}(\mathbf{A}^T\mathbf{B}) \leq \sum_i \sigma_i$.

This upper bound is achieved when $\mathbf{B} = \mathbf{U}\mathbf{V}^T$. With this choice, $\vec{u}_i^T(\mathbf{U}\mathbf{V}^T)\vec{v}_j = \delta_{ij}$, making each term in the sum equal to σ_i. So, the matrix $\mathbf{B}$ that maximizes the inner product is $\mathbf{U}\mathbf{V}^T$.

The optimal update step is:

$$\Delta\mathbf{W}^* = -\eta(\mathbf{U}\mathbf{V}^T)$$

where $\mathbf{U}$ and $\mathbf{V}$ are the singular vectors from the SVD of the gradient matrix $\nabla_{\mathbf{W}}\mathcal{L}(\mathbf{W})$. This is the instantaneous polar direction. Full Shampoo instead accumulates left and right gradient second-moment matrices and applies damped inverse-root preconditioners; it is not generally equal to the current gradient's polar factor. The undamped instantaneous identity uses pseudoinverses on active singular modes.

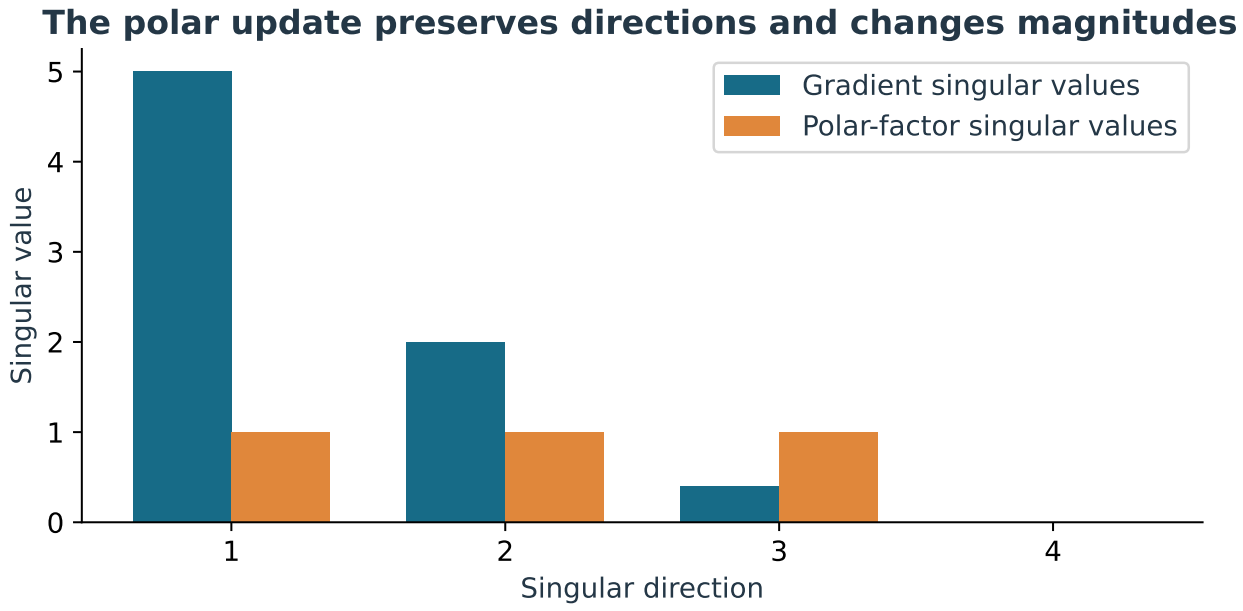


Figure 5.4: A matrix gradient has singular directions and singular magnitudes. A polar update equalizes the nonzero magnitudes while preserving the directions.

5.5 Towards a Principled Choice of Norm for Deep Learning

Deeper explanation: Feature-update scale versus parameter-update scale (p. 379)

Our recipe for creating optimizers depends on the choice of norm. Can we find a norm that is particularly well-suited for deep neural networks by considering their architectural properties?

Recapping Xavier Initialization and Norms

Xavier initialization was designed to preserve the variance of signals as they propagate through the network. If the input to a neuron has variance 1, we want the output to also have variance 1. To achieve this, weights were scaled by $\frac{1}{\sqrt{d_{in}}}$, where d_{in} is the number of incoming connections (fan-in).

This principle suggests that a good "norm" or measure of magnitude should be invariant to the dimension of the vector. The standard ℓ_2 norm does not have this property; a vector in a higher-dimensional space can have a larger ℓ_2 norm even if its components are drawn from the same distribution.

The Root-Mean-Square (RMS) Norm

Let's define a norm that normalizes by dimension. For a vector $\vec{x} \in \mathbb{R}^d$, the **Root-Mean-Square (RMS) norm** is:

$$||\vec{x}||_{\text{RMS}} = \sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2} = \frac{1}{\sqrt{d}}||\vec{x}||_2$$

If $||\vec{x}||_{\text{RMS}} = 1$, it means the "average" squared magnitude of an entry is 1. This captures the spirit of Xavier initialization.

The Induced RMS-to-RMS Norm

We can use this vector norm to define an induced matrix norm, which measures how a matrix scales vectors from an input space with one norm to an output space with another. For a matrix $\mathbf{A} : \mathbb{R}^{d_1} \to \mathbb{R}^{d_2}$, the induced RMS-to-RMS norm is:

$$||\mathbf{A}||_{\text{RMS}\to\text{RMS}} = \max_{||\vec{x}||_{\text{RMS}}=1} ||\mathbf{A}\vec{x}||_{\text{RMS}}$$

Let's expand the definition:

$$\begin{aligned} ||\mathbf{A}||_{\text{RMS}\to\text{RMS}} &= \max_{\frac{1}{\sqrt{d_1}}||\vec{x}||_2=1} \left(\frac{1}{\sqrt{d_2}}||\mathbf{A}\vec{x}||_2\right) \\ &= \max_{||\vec{x}||_2=\sqrt{d_1}} \left(\frac{1}{\sqrt{d_2}}||\mathbf{A}\vec{x}||_2\right) \end{aligned}$$

By scaling the optimization variable, we can rewrite this with a unit-norm constraint:

$$\begin{aligned}||\mathbf{A}||_{\text{RMS}\to\text{RMS}} &= \frac{\sqrt{d_1}}{\sqrt{d_2}} \max_{||\vec{z}||_2=1} ||\mathbf{A}\vec{z}||_2 \\ &= \sqrt{\frac{d_1}{d_2}}||\mathbf{A}||_2\end{aligned}$$

This is a key result: the induced RMS-to-RMS norm is simply the **spectral norm scaled by the square root of the ratio of fan-in to fan-out**.

Worked derivation: the spectral constraint and RMS conversion

Lecture basis: faculty handout 05, pp. 6–12. The steps below expand the lecture argument.

Step 1: reduce matrix descent to singular directions. Let $G = U_r\Sigma_r V_r^\top$ be the compact SVD of the gradient. The first-order change is the Frobenius inner product

$$\begin{aligned}\langle G, \Delta W\rangle_F &= \operatorname{tr}(G^\top \Delta W) \\ &= \operatorname{tr}(\Sigma_r U_r^\top \Delta W V_r) \\ &= \sum_{i=1}^{r} \sigma_i u_i^\top \Delta W v_i.\end{aligned}$$

For $\|\Delta W\|_2 \le \rho$, unit vectors satisfy $|u_i^\top \Delta W v_i| \le \|\Delta W v_i\| \le \rho$. Therefore

$$\langle G, \Delta W\rangle_F \ge -\rho \sum_i \sigma_i.$$

The choice $\Delta W_* = -\rho U_r V_r^\top$ attains every bound simultaneously. Its active singular values equal ρ. In null directions the optimizer need not be unique; this compact polar choice sets those updates to zero.

Step 2: convert the units of the constraint. For $W : \mathbb{R}^{d_{\text{in}}} \to \mathbb{R}^{d_{\text{out}}}$,

$$\begin{aligned}\|W\|_{\text{RMS}\to\text{RMS}} &= \sup_{x\neq 0} \frac{\|Wx\|_2/\sqrt{d_{\text{out}}}}{\|x\|_2/\sqrt{d_{\text{in}}}} \\ &= \sqrt{\frac{d_{\text{in}}}{d_{\text{out}}}}\|W\|_2.\end{aligned}$$

Hence $\|\Delta W\|_{\text{RMS}\to\text{RMS}} \le \eta$ is equivalent to $\|\Delta W\|_2 \le \eta\sqrt{d_{\text{out}}/d_{\text{in}}}$. Substituting that radius into the solved spectral problem gives

$$\boxed{\Delta W_* = -\eta\sqrt{\frac{d_{\text{out}}}{d_{\text{in}}}}\, U_r V_r^\top.}$$

The width factor comes from changing norm units, not from the SVD itself. This is an instantaneous constrained direction; optimizers that accumulate historical statistics are not identical to solving this problem at every step.

5.6 Conclusion and a Look Ahead

By using the induced RMS-to-RMS norm in our optimizer recipe, we can build an optimizer that naturally accounts for the specific geometry of each layer in a neural network.

Consider the constraint $||\Delta \mathbf{W}||_{\text{RMS}\to\text{RMS}} \leq \eta$. This is equivalent to:

$$\sqrt{\frac{d_{in}}{d_{out}}}||\Delta \mathbf{W}||_2 \leq \eta \implies ||\Delta \mathbf{W}||_2 \leq \eta\sqrt{\frac{d_{out}}{d_{in}}}$$

This means that a single, global learning rate hyperparameter η results in layer-specific effective learning rates that are automatically adjusted based on the layer's dimensions (fan-in and fan-out). This can significantly simplify hyperparameter tuning, as it mitigates the problem where different layers require vastly different learning rate scales.

This is the core insight behind the **Maximal Update Parametrization (muP)**, which provides a principled way to parameterize and optimize neural networks of varying widths, a topic for the next lecture.

Lecture 6

muP: maximal update parameterization

Course sources: lecture_06; 2025-09-18 transcript.

6.1 Recap: The RMS to RMS Norm

Deeper explanation: Feature-update scale versus parameter-update scale (p. 379)

We begin by recalling the concept of an induced matrix norm. For a matrix A, the norm induced by vector norms $\|\cdot\|_\alpha$ and $\|\cdot\|_\beta$ is defined as:

$$\|A\|_{\alpha\to\beta} = \max_{\|\vec{x}\|_\alpha=1} \|A\vec{x}\|_\beta$$

Last time, we were motivated by Xavier initialization, which suggests that the activations in a neural network should have a consistent scale, specifically with entries of order one. This led us to the Root Mean Square (RMS) norm for a vector $\vec{x} \in \mathbb{R}^d$:

$$\|\vec{x}\|_{\text{RMS}} = \sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2} = \frac{1}{\sqrt{d}}\|\vec{x}\|_2$$

We can use this to define an induced matrix norm, the RMS to RMS norm. Let $A \in \mathbb{R}^{d_{out}\times d_{in}}$ be a weight matrix that maps an input $\vec{x} \in \mathbb{R}^{d_{in}}$ to an output $\vec{h} \in \mathbb{R}^{d_{out}}$. The RMS to RMS norm of A is:

$$\begin{aligned}
\|A\|_{\text{RMS}\to\text{RMS}} &= \max_{\|\vec{x}\|_{\text{RMS}}=1} \|A\vec{x}\|_{\text{RMS}} \\
&= \max_{\frac{1}{\sqrt{d_{in}}}\|\vec{x}\|_2=1} \frac{1}{\sqrt{d_{out}}}\|A\vec{x}\|_2 \\
&= \max_{\|\vec{x}\|_2=\sqrt{d_{in}}} \frac{1}{\sqrt{d_{out}}}\|A\vec{x}\|_2 \\
&= \frac{\sqrt{d_{in}}}{\sqrt{d_{out}}} \max_{\|\vec{z}\|_2=1} \|A\vec{z}\|_2 \quad (\text{where } \vec{z} = \vec{x}/\sqrt{d_{in}}) \\
&= \sqrt{\frac{d_{in}}{d_{out}}}\|A\|_2
\end{aligned}$$

Here, $\|A\|_2$ is the spectral norm of A, which is its largest singular value. The key takeaway is that the RMS to RMS norm naturally incorporates the dimensions of the matrix (fan-in d_{in} and fan-out d_{out}), which aligns with the structural properties of neural network layers.

6.2 The Optimizer Recipe and Layer-Specific Rates

Deeper explanation: Preconditioning is a change in the local ruler (p. 373)

The General Recipe

Recall our general recipe for designing an optimizer. Given the gradient of the loss with respect to the weights, $\nabla_W \mathcal{L}(W)$, we find the update step ΔW by solving:

$$\arg\min_{\|\Delta W\| \leq \eta} \langle \nabla_W \mathcal{L}(W), \Delta W \rangle$$

The optimizer is defined by two choices:

1. An appropriate norm $\|\cdot\|$ for the constraint.
2. A step size hyperparameter η.

The choice of norm should reflect the geometry of the problem. However, choosing η typically requires an expensive hyperparameter sweep.

Using the RMS to RMS Norm

Let's apply our RMS to RMS norm to this recipe. We want to find an update ΔW such that its norm is bounded by some hyperparameter γ:

$$\|\Delta W\|_{\text{RMS}\to\text{RMS}} \leq \gamma$$

Using our derived relationship, this is equivalent to:

$$\sqrt{\frac{d_{in}}{d_{out}}}\|\Delta W\|_2 \leq \gamma \implies \|\Delta W\|_2 \leq \gamma\sqrt{\frac{d_{out}}{d_{in}}}$$

Key Observation: By choosing the RMS to RMS norm, we can use a single, global hyperparameter γ for the entire network. However, the constraint on the spectral norm of the update for each layer automatically scales with the layer's dimensions (d_{in} and d_{out}). This provides a principled way to achieve layer-specific update magnitudes without needing to tune separate hyperparameters for each layer.

This exact idea, when plugged into the optimizer recipe, leads to a simplified version of the Shampoo optimizer and is the foundation for the **Muon optimizer**. Muon has shown tremendous practical impact, significantly accelerating training time for models like NanoGPT and enabling the training of trillion-parameter models.

6.3 Step-size scaling for Adam and SignSGD

Deeper explanation: Feature-update scale versus parameter-update scale (p. 379)

Instead of building a new optimizer, can we use this insight to improve existing, widely-used optimizers like Adam or SignSGD? We will focus on SignSGD for simplicity, whose update rule is:

$$W_{t+1} = W_t - \eta \cdot \text{sign}(\nabla_W \mathcal{L}(W))$$

Our goal is to find a principled way to choose the learning rate η by factoring in the layer geometry. We will do this by requiring that the update step $\Delta W = -\eta \cdot \text{sign}(\nabla_W \mathcal{L}(W))$ has a bounded RMS to RMS norm:

$$\|\Delta W\|_{\text{RMS}\to\text{RMS}} \leq \gamma$$

Analysis with Mini-Batch Size 1

Let's consider an update based on a single data point (mini-batch size 1).

Gradient Rank For a single data point and common loss functions (e.g., squared error, cross-entropy), the gradient with respect to the weights, $\nabla_W \mathcal{L}$, is a rank-one matrix. This is because the update must lie in the span of the input activations. Using the chain rule, for a layer $\vec{h} = W\vec{x}$, we have $\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial \mathcal{L}}{\partial \vec{h}} \otimes \vec{x}$, which is an outer product and thus rank-one. We can write it as:

$$\nabla_W \mathcal{L} = \sigma \vec{u}\vec{v}^T$$

For a rank-one matrix, the spectral norm and Frobenius norm are equal: $\|\nabla_W \mathcal{L}\|_2 = \|\nabla_W \mathcal{L}\|_F = \sigma$.

Sign Matrix Rank Let $S = \text{sign}(\nabla_W \mathcal{L})$. Since the sign function is applied element-wise and $\text{sign}(ab) = \text{sign}(a)\text{sign}(b)$, we have:

$$S = \text{sign}(\sigma \vec{u}\vec{v}^T) = \text{sign}(\vec{u})\text{sign}(\vec{v}^T)$$

This shows that S is also an outer product of two vectors, and therefore is also a rank-one matrix.

Deriving the Scaling for η Now we can analyze our constraint $\|\Delta W\|_{\text{RMS}\to\text{RMS}} \leq \gamma$.

$$\|-\eta S\|_{\text{RMS}\to\text{RMS}} = \eta \|S\|_{\text{RMS}\to\text{RMS}} = \eta \sqrt{\frac{d_{in}}{d_{out}}} \|S\|_2 \leq \gamma$$

We need to compute the spectral norm of S. Since S is rank-one, its spectral norm is equal to its Frobenius norm. The matrix $S \in \mathbb{R}^{d_{out} \times d_{in}}$ has entries that are all ± 1.

$$\|S\|_F^2 = \sum_{i=1}^{d_{out}} \sum_{j=1}^{d_{in}} S_{ij}^2 = \sum_{i=1}^{d_{out}} \sum_{j=1}^{d_{in}} (\pm 1)^2 = d_{out} \cdot d_{in}$$

Therefore, $\|S\|_2 = \|S\|_F = \sqrt{d_{in} d_{out}}$. Substituting this back into our inequality:

$$\eta \sqrt{\frac{d_{in}}{d_{out}}} \left(\sqrt{d_{in} d_{out}} \right) \leq \gamma$$
$$\eta \cdot d_{in} \leq \gamma$$
$$\eta \leq \frac{1}{d_{in}} \gamma$$

This result provides a principled scaling law: **the learning rate η for a given layer should be scaled inversely with its fan-in dimension, d_{in}.**

Note on Larger Batch Sizes When the batch size is greater than 1, the gradient is no longer guaranteed to be rank-one. However, the inequality $\|A\|_2 \leq \|A\|_F$ always holds. If we use the Frobenius norm as an upper bound for the spectral norm in our derivation, the analysis follows, but the resulting bound on η will be more conservative (i.e., suggest a smaller learning rate).

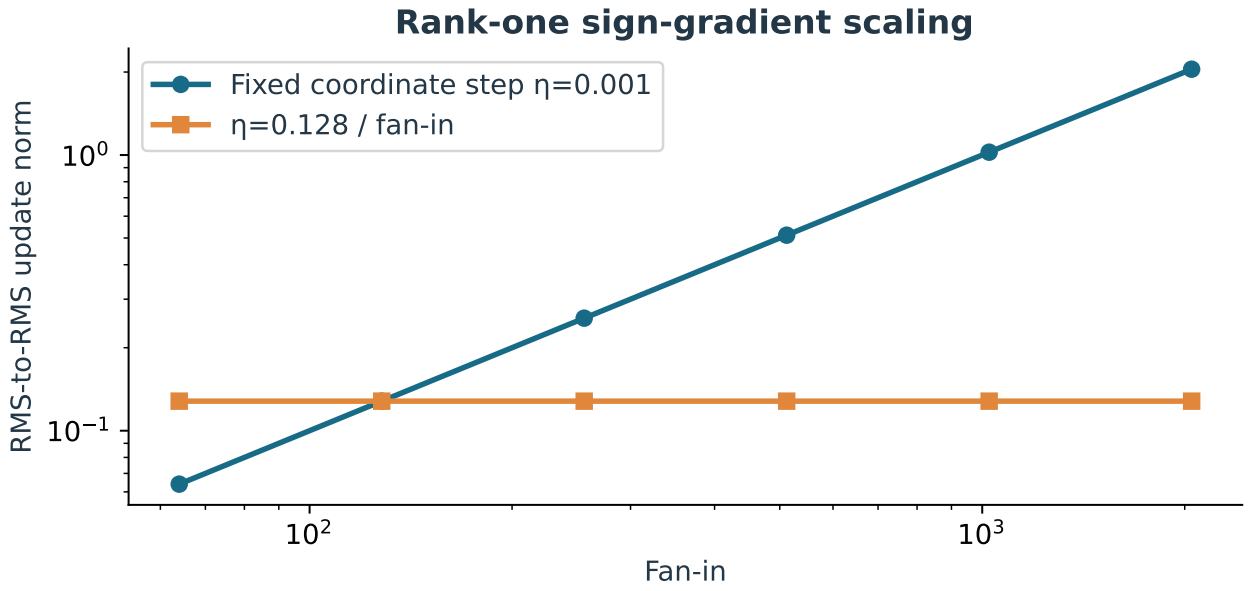


Figure 6.1: Width-aware steps preserve a comparable functional update. A fixed coordinate step grows with fan-in in the rank-one sign-gradient example.

Worked derivation: the one-example SignSGD width rule

Lecture basis: faculty handout 06, pp. 9–13, 18–23. The steps below expand the lecture argument.

Step 1: differentiate a dense layer. Let $z = Wx$, with loss gradient $g = \nabla_z L$. Since $dz = (dW)x$,

$$dL = g^\top (dW)x = \operatorname{tr}(xg^\top dW), \qquad \nabla_W L = gx^\top.$$

For one example, this is an outer product. If all entries of g, x are nonzero, put $a = \operatorname{sign}(g)$, $b = \operatorname{sign}(x)$. Then $S = \operatorname{sign}(gx^\top) = ab^\top$.

Step 2: compute its singular value exactly.

$$\begin{aligned} S^\top S &= ba^\top ab^\top = \|a\|^2 bb^\top, \\ \|S\|_2 &= \|a\|\|b\| = \sqrt{d_{\text{out}} d_{\text{in}}}, \\ \|S\|_{\text{RMS}\to\text{RMS}} &= \sqrt{d_{\text{in}}/d_{\text{out}}}\,\|S\|_2 = d_{\text{in}}. \end{aligned}$$

For $\Delta W = -\eta S$, imposing RMS operator size at most γ therefore gives $\eta \le \gamma/d_{\text{in}}$. For $d_{\text{in}} = 1024$, a raw sign step of size $\gamma/1024$ has RMS operator norm exactly γ under these assumptions.

Step 3: state what survives beyond one example. For a mini-batch, G is a sum of outer products, and $\operatorname{sign}(G)$ need not have rank one. Nevertheless every entry has magnitude at most one, so

$$\|\operatorname{sign}(G)\|_2 \le \|\operatorname{sign}(G)\|_F \le \sqrt{d_{\text{out}} d_{\text{in}}}.$$

The same step rule is a sufficient worst-case bound, but equality need not hold. Adam's history-normalized direction is not generally exactly a sign matrix.

Step 4: retain the product term when both layers and features change. For $z = Wh$,

$$\begin{aligned} \Delta z &= (W + \Delta W)(h + \Delta h) - Wh \\ &= \Delta W h + W\Delta h + \Delta W \Delta h. \end{aligned}$$

Submultiplicativity bounds each term in RMS units. First-order scaling arguments drop the final term only when both changes are small. Norm bounds prove upper bounds; a nonvanishing update also needs alignment and noncancellation, so an $O(1)$ bound alone is not a $\Theta(1)$ feature-learning guarantee.

6.4 Maximal Update Parameterization (μP)

Deeper explanation: Depth creates an additional scale problem (p. 378)

The scaling law we just derived is the essence of Maximal Update Parameterization (μP), a framework for understanding how to scale hyperparameters as model dimensions change.

The Hyperparameter Transfer Problem

Training large neural networks is extremely expensive. A key challenge is hyperparameter tuning. Empirically, the optimal learning rate for a model changes as its width (e.g., d_{model}) changes. Under standard parameterizations (like Xavier initialization alone), the optimal learning rate can shift by orders of magnitude, making it impossible to tune on a small, cheap "proxy" model and transfer the settings to a full-scale model.

μP provides a way to parameterize the model such that the optimal hyperparameters (especially learning rate) remain stable across different model widths. This enables "one-shot transfer": find the best hyperparameters on a small model and use them directly on the large model.

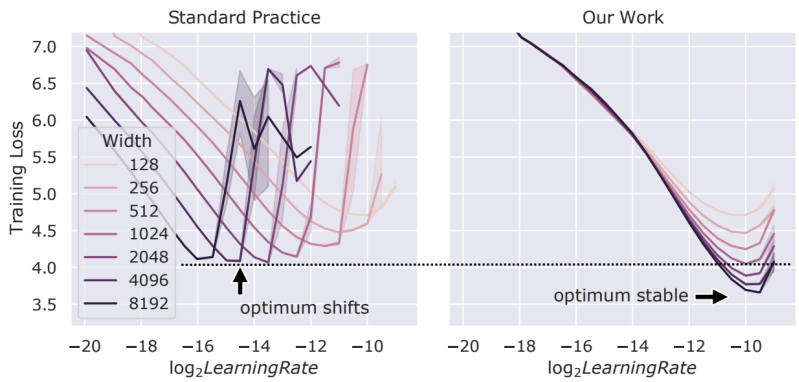


Figure 1: Training loss against learning rate on Transformers of varying d_{model} trained with Adam. Conventionally and in contrast with our technique, different widths do not share the same optimal hyperparameter; wider networks do not always perform better than narrower ones; in fact they underperform the same-width networks in our technique even after tuning learning rate (see dashed line). See Sections 3 and 4 for experimental setup.

Yang et al 2022.

Figure 6.2: Learning-rate transfer across model widths. Compare where the curves attain their best values as width changes. The point of maximal-update parameterization is to make useful hyperparameters transfer more reliably under its scaling assumptions; these are the experiments reproduced in the lecture. Original annotated source excerpt; Gireeja Ranade, faculty handout 06, p. 15.

Analogy: The Central Limit Theorem

To understand the importance of correct scaling, consider the sum of n i.i.d. random variables X_i with zero mean and unit variance.

- $\frac{1}{n}\sum_{i=1}^{n} X_i \xrightarrow{n\to\infty} 0$ (Wrong scaling)
- $\mathrm{Var}(\sum_{i=1}^{n} X_i) = n \to \infty$ (Unstable variance; the centered sum does not converge to positive infinity)
- $\frac{1}{\sqrt{n}}\sum_{i=1}^{n} X_i \xrightarrow{d} \mathcal{N}(0,1)$ (Correct scaling)

For a nondegenerate finite-variance limit, the normalization is $1/\sqrt{n}$. For bounded continuous f, convergence in distribution implies $\mathbb{E}[f(\alpha n^{-1/2}\sum_i X_i)] \to \mathbb{E}[f(\alpha Z)]$ for each fixed α, where $Z \sim \mathcal{N}(0,1)$. Unbounded functions require additional moment control. Pointwise objective convergence alone does not imply convergence of optimal hyperparameters; the lecture uses this scaling argument as motivation for width transfer.

μP aims to find the "right scaling" for neural network hyperparameters with respect to network width, just as $1/\sqrt{n}$ is the right scaling for the sum of random variables.

Worked derivation: why a sum uses square-root width scaling

Lecture basis: faculty handout 06, pp. 15–17. The steps below expand the lecture argument.

Step 1: compute moments before taking a limit. For independent, centered Z_i with variance v, define $S_n = n^{-a}\sum_{i=1}^{n} Z_i$. Then

$$\mathbb{E}[S_n] = 0,$$
$$\mathrm{Var}(S_n) = n^{-2a}\sum_{i=1}^{n}\mathrm{Var}(Z_i) = vn^{1-2a}.$$

An order-one nonzero variance requires $1-2a = 0$, hence $a = 1/2$. With identical distributions and finite variance, the central limit theorem then gives $S_n \Rightarrow \mathcal{N}(0, v)$. Scaling by $1/n$ instead produces variance $v/n \to 0$; omitting scaling gives variance $nv \to \infty$.

Step 2: apply the argument to a dense preactivation. If $z = \sum_{j=1}^{n} W_j h_j$, independent centered W_j and order-one input second moments give $\mathbb{E}[z^2] \sim n\,\mathrm{Var}(W_j)$. Therefore $\mathrm{Var}(W_j) \sim 1/n$ keeps z finite, so the weight standard deviation scales as $1/\sqrt{n}$.

Step 3: delimit the hyperparameter-transfer analogy. Convergence of a normalized random sum motivates choosing coordinates whose scale remains stable across widths. It does not prove that the best learning rate of a trained network converges, or that minimizers of width-dependent validation curves converge. Those claims require additional control of the objectives and their minimizers. The lecture uses this limit as an analogy for principled scaling, not a theorem about all training runs.

6.5 A Principled View: Conditions for Stable Feature Learning

Deeper explanation: Depth creates an additional scale problem (p. 378)

A simplified perspective on μP comes from identifying two desired properties (desiderata) for the network's behavior as its width changes. We want both the activations and their updates to maintain a stable scale.

Let $\vec{h}_l$ be the hidden activations at layer l. The two desiderata are:

1. **Stable Activation Scale:** The RMS norm of the activations should be of constant order with respect to network width: $\|\vec{h}_l\|_{\text{RMS}} = \Theta(1)$.
2. **Stable Update Scale:** The RMS norm of the change in activations per optimizer step should also be of constant order: $\|\Delta\vec{h}_l\|_{\text{RMS}} = \Theta(1)$.

Achieving the Desiderata

Ignoring bias and non-linearities, a layer computes $\vec{h}_l = W_l\vec{h}_{l-1}$.

1. To satisfy the first desideratum, we need to preserve the RMS norm across layers. An inductive argument shows that if $\|\vec{h}_{l-1}\|_{\text{RMS}} = \Theta(1)$, then to get $\|\vec{h}_l\|_{\text{RMS}} = \Theta(1)$, we must have $\|W_l\|_{\text{RMS}\to\text{RMS}} = \Theta(1)$. This is precisely the condition that motivates Xavier initialization.
2. To satisfy the second desideratum, consider the update to the activations: $\Delta\vec{h}_l = (\Delta W_l)\vec{h}_{l-1} + W_l(\Delta\vec{h}_{l-1})$. For $\|\Delta\vec{h}_l\|_{\text{RMS}}$ to remain $\Theta(1)$ throughout the network, the updates from the current layer, $(\Delta W_l)\vec{h}_{l-1}$, must have the same scale as the propagated updates, $W_l(\Delta\vec{h}_{l-1})$. This requires the norm of the weight update, $\|\Delta W_l\|$, to have the same scaling as the weight norm, $\|W_l\|$, with respect to the layer dimensions.

This leads to two fundamental conditions for stable feature learning at scale:

- **Initialization Condition:** $\|W_l\|_{\text{RMS}\to\text{RMS}} = \Theta(1)$
- **Optimization Condition:** $\|\Delta W_l\|_{\text{RMS}\to\text{RMS}} = \Theta(1)$

The second condition provides a fundamental justification for our earlier approach: constraining the optimizer update step in the RMS to RMS norm is a principled way to ensure stable training dynamics as network width grows.

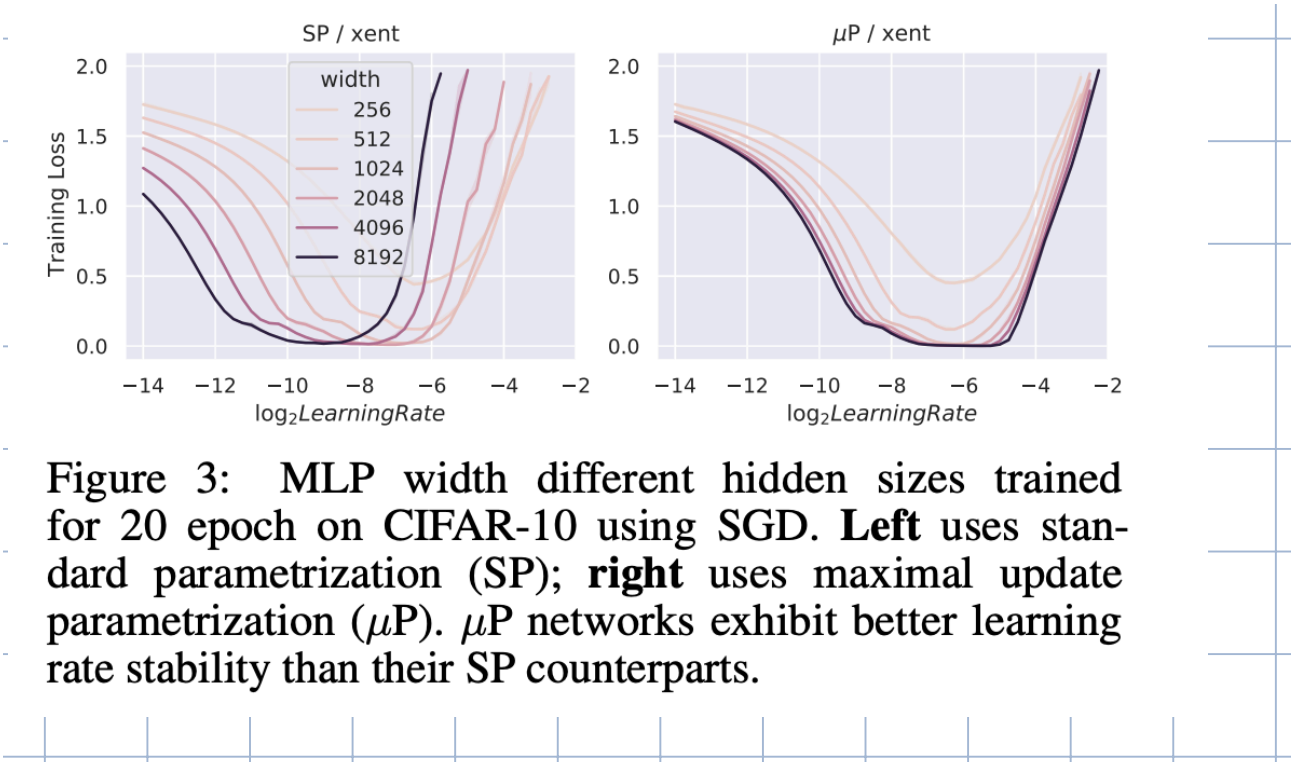


Figure 3: MLP width different hidden sizes trained for 20 epoch on CIFAR-10 using SGD. **Left** uses standard parametrization (SP); **right** uses maximal update parametrization (μP). μP networks exhibit better learning rate stability than their SP counterparts.

Figure 6.3: A second width-transfer experiment. The MLP experiment complements the preceding scaling argument. Agreement of preferred settings across widths is an empirical diagnostic of transfer, not a proof for arbitrary architectures. Original annotated source excerpt; Gireeja Ranade, faculty handout 06, p. 26.

Lecture 7

Muon and Newton–Schulz

Course sources: lecture_07; 2025-09-23 transcript.

7.1 Recap: Maximal Update Parametrization (MUP)

Deeper explanation: Feature-update scale versus parameter-update scale (p. 379)

The primary goal of Maximal Update Parametrization (MUP) is to address the challenge of hyperparameter tuning for very large neural networks. Training these models is computationally expensive, making traditional hyperparameter sweeps infeasible. MUP provides a principled way to find optimal hyperparameters on a smaller, cheaper "proxy" network and then transfer them directly to a much larger target network. This lecture focuses on scaling with respect to network **width**.

A Spectral Perspective on Feature Learning

To build intuition, we adopt a simplified linear algebraic perspective from the paper "A Spectral Condition for Feature Learning" by Bernstein, Simon, and Yang. This perspective leads to the same scaling laws as the original MUP paper, which used more complex tensor algebra.

Consider a single hidden layer L in a network:

$$H_L = \phi(W_L H_{L-1} + b_L)$$

where H_L is the output activation vector, $W_L \in \mathbb{R}^{d_{out} \times d_{in}}$ is the weight matrix, H_{L-1} is the input activation vector, b_L is the bias, and ϕ is a non-linear activation function.

For effective "feature learning" (i.e., the network's parameters move meaningfully from their initialization), we desire two properties to hold as the network width scales:

1. **Stable Activation Magnitudes:** The activations at each layer should have a consistent magnitude, independent of the network width. We formalize this by requiring the Root Mean Square (RMS) norm of the hidden layers to be of order one.

$$\|H_L\|_{RMS} = \sqrt{\frac{1}{d_{out}} \sum_{i=1}^{d_{out}} (H_L)_i^2} \approx O(1)$$

   Since $\|H_L\|_{RMS} = \frac{1}{\sqrt{d_{out}}} \|H_L\|_2$, this is equivalent to requiring the Euclidean norm to be $\|H_L\|_2 \approx O(\sqrt{d_{out}})$.

2. **Meaningful Parameter Updates:** The updates to the parameters during training should not become negligible as the network width increases. If they do, the network remains stuck near its initialization. We formalize this by requiring the RMS norm of the change in activations, ΔH_L, to also be of order one.

$$\|\Delta H_L\|_{RMS} \approx O(1) \implies \|\Delta H_L\|_2 \approx O(\sqrt{d_{out}})$$

Deriving Scaling Conditions

To see what conditions on the weights W_L achieve these properties, we simplify the model by ignoring the bias and non-linearity:

$$H_L = W_L H_{L-1}$$

We proceed inductively, assuming the desired properties hold for layer $L-1$. The paper argues that during training, the hidden state vectors H_{L-1} tend to align with the top singular vectors of the weight matrix W_L. This allows us to approximate the effect of the matrix multiplication on the norm:

$$\|H_L\|_2 \approx \|W_L\|_{spec} \cdot \|H_{L-1}\|_2$$

where $\|W_L\|_{spec} = \sigma_{max}(W_L)$ is the spectral norm.

Using our desired properties for the Euclidean norms, we substitute them into the equation:

$$O(\sqrt{d_{out}}) \approx \|W_L\|_{spec} \cdot O(\sqrt{d_{in}})$$

This implies a condition on the spectral norm of the weight matrix:

$$\|W_L\|_{spec} \approx O\left(\sqrt{\frac{d_{out}}{d_{in}}}\right)$$

This is equivalent to the condition from the previous lecture that the RMS-to-RMS induced norm of W_L should be $O(1)$.

Next, we analyze the update ΔH_L. The update rule for H_L is:

$$\Delta H_L = \Delta W_L \cdot H_{L-1} + W_L \cdot \Delta H_{L-1}$$

Taking the norm and using the triangle inequality (and again, the top singular vector alignment assumption):

$$\|\Delta H_L\|_2 \approx \|\Delta W_L \cdot H_{L-1} + W_L \cdot \Delta H_{L-1}\|_2 \leq \|\Delta W_L\|_{spec}\|H_{L-1}\|_2 + \|W_L\|_{spec}\|\Delta H_{L-1}\|_2$$

We want $\|\Delta H_L\|_2 \approx O(\sqrt{d_{out}})$. Inductively, we assume $\|H_{L-1}\|_2 \approx O(\sqrt{d_{in}})$ and $\|\Delta H_{L-1}\|_2 \approx O(\sqrt{d_{in}})$. We already derived that $\|W_L\|_{spec} \approx O(\sqrt{d_{out}/d_{in}})$. For the equation to balance, the remaining term must also satisfy:

$$\|\Delta W_L\|_{spec} \approx O\left(\sqrt{\frac{d_{out}}{d_{in}}}\right)$$

This implies that both the weight matrix W_L and its update ΔW_L must have an RMS-to-RMS induced norm of $O(1)$. These conditions are the foundation for deriving the MUP scaling laws for hyperparameters like the learning rate.

7.2 Muon: Momentum Orthogonalized by Newton-Schulz

Deeper explanation: Newton–Schulz convergence on nonzero singular modes (p. 380)

Motivation: State-of-the-Art Performance

Muon is a recent, powerful optimizer that has demonstrated significant performance gains. Its introduction was a key factor in a massive improvement in the "NanoGPT speedrun," a competition to train a small language model as quickly as possible. The ideas have also been successfully applied to much larger, trillion-parameter models, indicating their scalability and relevance to frontier models.

Revisiting the Optimizer Recipe and Shampoo

Recall the general recipe for a first-order optimizer. At each step, we want to find an update ΔW that maximally decreases the loss, constrained by some norm:

$$\Delta W^* = \arg\min_{\|\Delta W\| \leq \eta} \langle \nabla_W \mathcal{L}(W), \Delta W \rangle$$

The solution is to align ΔW with the negative gradient: $\Delta W \propto -\nabla_W \mathcal{L}(W)$. The choice of norm $\|\cdot\|$ determines the optimizer's behavior.

The **Shampoo** optimizer can be understood by choosing the **spectral norm**, $\|\cdot\|_2$. If the gradient is $\nabla_W \mathcal{L}(W) = G_t = U\Sigma V^T$, the optimal update is a **semi-orthogonal update**:

$$\Delta W^* = \eta \cdot U_r V_r^T$$

where U_r, V_r correspond to the non-zero singular values. This update removes the influence of the singular values Σ, effectively taking a uniform step in all singular directions. This is the update for Shampoo *without accumulation*.

The actual Shampoo algorithm looks more complex, using left (L_t) and right (R_t) preconditioners:

$$W_{t+1} = W_t - \eta L_t^{-1/4} G_t R_t^{-1/4}$$

where L_t and R_t are typically accumulated statistics of the gradients. Without accumulation, we have $L_t = G_t G_t^T$ and $R_t = G_t^T G_t$. Let's show this is equivalent to the semi-orthogonal update.

$$G_t G_t^T = (U\Sigma V^T)(U\Sigma V^T)^T = U\Sigma V^T V \Sigma^T U^T = U\Sigma\Sigma^T U^T$$
$$G_t^T G_t = (U\Sigma V^T)^T(U\Sigma V^T) = V\Sigma^T U^T U\Sigma V^T = V\Sigma^T\Sigma V^T$$

The matrix square roots act only on the diagonal singular value matrices:

$$(G_t G_t^T)^{-1/4} = (U(\Sigma\Sigma^T)^{1/2}U^T)^{-1/2} = U(\Sigma\Sigma^T)^{-1/4}U^T = U\Sigma^{-1/2}U^T$$
$$(G_t^T G_t)^{-1/4} = V(\Sigma^T\Sigma)^{-1/4}V^T = V\Sigma^{-1/2}V^T$$

(Assuming a square, invertible Σ for simplicity). Combining these:

$$(G_t G_t^T)^{-1/4} G_t (G_t^T G_t)^{-1/4} = (U\Sigma^{-1/2}U^T)(U\Sigma V^T)(V\Sigma^{-1/2}V^T) = U\Sigma^{-1/2}\Sigma\Sigma^{-1/2}V^T = UV^T$$

The fourth roots are used because they can be computed more efficiently than a full SVD using iterative Newton-style methods.

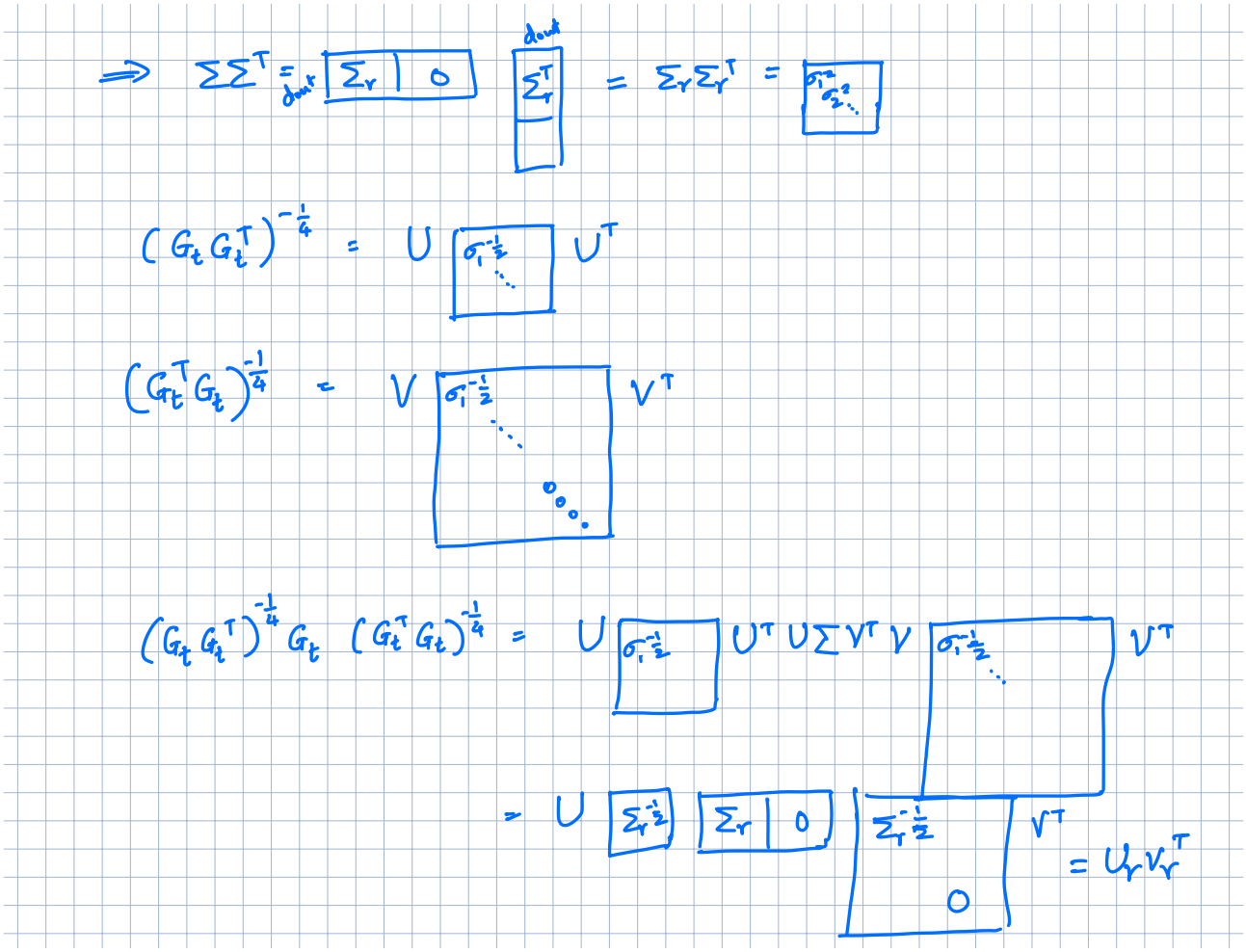


Figure 7.1: Rectangular singular-value factors in the update. The rectangular blocks expose which singular directions survive the normalization. On rank-deficient matrices, inverse powers are interpreted on the nonzero singular subspace, using the pseudoinverse convention. Original annotated source excerpt; Gireeja Ranade, faculty handout 07, p. 4.

Key Idea 1: RMS-to-RMS Norm Optimization

Muon's first key idea is to replace the spectral norm in the optimizer recipe with the **RMS-to-RMS induced norm**, inspired by MUP. The RMS-to-RMS norm is just a scaled version of the spectral norm:

$$\|\Delta W\|_{RMS\to RMS} = \sqrt{\frac{d_{in}}{d_{out}}}\|\Delta W\|_2$$

The optimization problem becomes:

$$\arg\min_{\|\Delta W\|_{RMS\to RMS}\leq\eta} \langle \nabla_W \mathcal{L}(W), \Delta W\rangle \quad \equiv \quad \arg\min_{\|\Delta W\|_2\leq\eta\sqrt{\frac{d_{out}}{d_{in}}}} \langle \nabla_W \mathcal{L}(W), \Delta W\rangle$$

The solution is a scaled semi-orthogonal update:

$$\Delta W^* = \eta\sqrt{\frac{d_{out}}{d_{in}}} U_r V_r^T$$

This scaling factor, derived from MUP principles, is the first major component of Muon.

Key Idea 2: Efficient Semi-Orthogonalization via Newton-Schulz

Muon's second key idea is a computationally efficient method to approximate the semi-orthogonal matrix UV^T without performing an SVD. This is achieved through **Newton-Schulz iterations**.

The goal is to find a function f such that for a matrix $X = U\Sigma V^T$, we have $f(X) \approx UV^T$. The core insight is that **odd polynomials commute with the SVD**. For an odd polynomial $p(x) = a_1 x + a_3 x^3 + \dots$, we have:

$$p(U\Sigma V^T) = U p(\Sigma) V^T$$

This means the polynomial acts directly on the singular values, while leaving the singular vectors U and V unchanged. The problem reduces to finding a scalar polynomial $p(x)$ that, when iterated, drives its input towards 1 (for positive inputs).

A polynomial with this property is:

$$p(x) = \frac{3}{2}x - \frac{1}{2}x^3$$

By repeatedly applying this function, $x_{k+1} = p(x_k)$, any initial value $x_0 \in (0, \sqrt{3})$ will converge to 1. Since singular values are always non-negative, we only care about the positive domain.

The interval $(0, \sqrt{3})$ is a sufficient positive basin for the scalar cubic. Values outside it require separate analysis; they do not all necessarily diverge. A zero singular mode stays zero. To prevent this, Muon employs a simple trick: normalize the gradient matrix by its Frobenius norm before applying the iterations.

$$G_{norm} = \frac{G_t}{\|G_t\|_F}$$

Since $\|G_t\|_F = \sqrt{\sum \sigma_i^2} \geq \sigma_{max} = \|G_t\|_2$, this normalization ensures that all singular values of G_{norm} are less than or equal to 1, placing them safely within the convergence basin of the polynomial iteration.

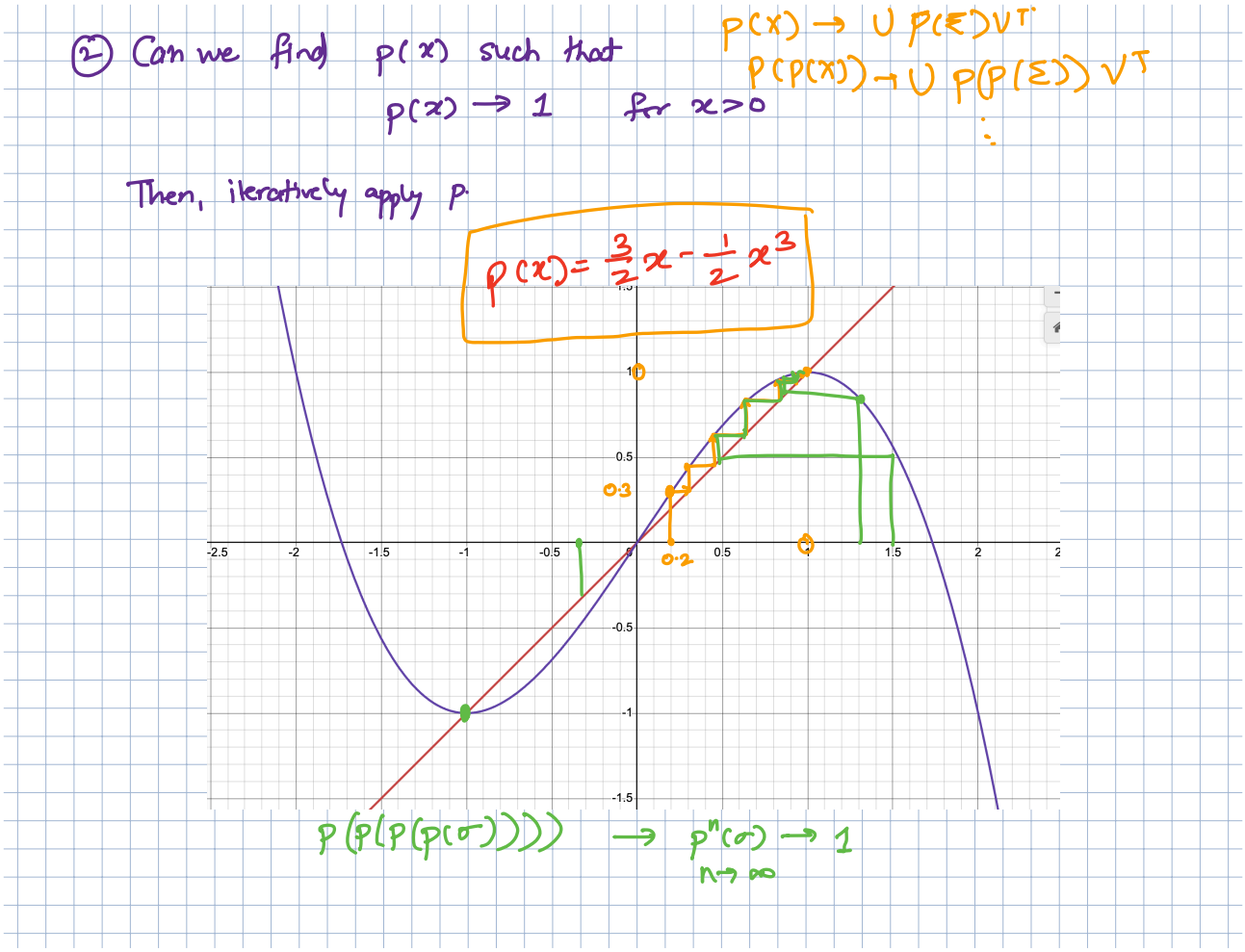


Figure 7.2: Cubic iteration as a cobweb diagram. Alternate between the scalar update curve and the identity line to follow successive singular values. In the appropriate basin, the cubic map drives nonzero singular values toward one; a zero singular value stays zero. Original annotated source excerpt; Gireeja Ranade, faculty handout 07, p. 8.

Rectangular Newton–Schulz in singular blocks

For a rank-r matrix $Z \in \mathbb{R}^{m\times n}$, a full SVD can be partitioned as

$$Z = U \begin{bmatrix} S & 0 \\ 0 & 0 \end{bmatrix} V^\top, \quad S = \operatorname{diag}(s_1, \dots, s_r), \quad ZZ^\top Z = U \begin{bmatrix} S^3 & 0 \\ 0 & 0 \end{bmatrix} V^\top.$$

The outer rectangular block has row sizes $(r, m-r)$ and column sizes $(r, n-r)$; zero-sized blocks are omitted. Thus the cubic iteration gives

$$\tfrac{3}{2}Z - \tfrac{1}{2}ZZ^\top Z = U \begin{bmatrix} \frac{3}{2}S - \frac{1}{2}S^3 & 0 \\ 0 & 0 \end{bmatrix} V^\top.$$

This is why scalar analysis of $s \mapsto 1.5s - 0.5s^3$ controls the matrix iteration. Zero singular modes stay zero. If positive modes converge to one after suitable initial scaling, the limit is

$U_r V_r^\top$: its products are the projectors $U_r U_r^\top$ and $V_r V_r^\top$, not necessarily two identity matrices. A rectangular matrix cannot have both orthonormal rows and orthonormal columns unless it is square.

Worked derivation: inverse fourth roots and Newton–Schulz

Lecture basis: faculty handout 07, pp. 3–9. The steps below expand the lecture argument.

Step 1: simplify the instantaneous preconditioner on its support. For $G = U_r \Sigma_r V_r^\top$ with positive compact singular values,

$$\begin{aligned} GG^\top &= U_r \Sigma_r^2 U_r^\top, \qquad G^\top G = V_r \Sigma_r^2 V_r^\top, \\ [(GG^\top)^+]^{1/4} &= U_r \Sigma_r^{-1/2} U_r^\top, \\ [(G^\top G)^+]^{1/4} &= V_r \Sigma_r^{-1/2} V_r^\top. \end{aligned}$$

The pseudoinverses define zero action off the active subspace. Multiplying in order,

$$\begin{aligned} [(GG^\top)^+]^{1/4} G [(G^\top G)^+]^{1/4} &= U_r \Sigma_r^{-1/2} \Sigma_r \Sigma_r^{-1/2} V_r^\top \\ &= U_r I_r V_r^\top. \end{aligned}$$

Historical Shampoo uses accumulated matrices and damping; those generally do not share the current gradient's singular vectors, so this cancellation is an instantaneous illustration.

Step 2: turn a scalar polynomial into a matrix iteration. Normalize $Z_0 = G/c$ with $c \geq \|G\|_2$ and $c > 0$. Define

$$Z_{k+1} = \tfrac{3}{2} Z_k - \tfrac{1}{2} Z_k Z_k^\top Z_k.$$

If $Z_k = U_r S_k V_r^\top$, orthonormality gives $Z_k Z_k^\top Z_k = U_r S_k^3 V_r^\top$. The singular vectors remain fixed and each positive singular value follows

$$s_{k+1} = \tfrac{3}{2} s_k - \tfrac{1}{2} s_k^3.$$

This multiplication is valid for rectangular matrices; one need not take the ordinary matrix cube of a rectangular Z_k.

Step 3: prove convergence on the normalized interval. For $0 < s \leq 1$,

$$\begin{aligned} f(s) - s &= \tfrac{1}{2} s (1 - s^2) \geq 0, \\ 1 - f(s) &= \tfrac{1}{2} (1 - s)^2 (s + 2) \geq 0. \end{aligned}$$

Thus s_k increases and remains at most one. Its positive limit solves $s = \frac{3}{2} s - \frac{1}{2} s^3$, hence equals one. If $e_k = 1 - s_k$, then

$$e_{k+1} = \tfrac{1}{2} e_k^2 (3 - e_k),$$

which proves quadratic convergence near one. Zero singular values stay zero, so the limit is $U_r V_r^\top$, not an identity matrix on a larger null space. Practical few-step quintic Muon polynomials trade exact convergence for speed and need their own stability analysis; the cubic proof does not automatically apply to them.

The Muon Algorithm and Implementation Details

Muon stands for **M**omentum **O**rthogonalized by **N**ewton-Schulz. The algorithm proceeds as follows:

1. **Momentum Step:** Accumulate gradients using a standard momentum buffer.

$$B_t = \mu B_{t-1} + G_t$$

2. **Orthogonalization Step:** Compute the approximate semi-orthogonal matrix of the momentum buffer using Newton-Schulz iterations. For a rectangular matrix use $P(X) = \frac{3}{2}X - \frac{1}{2}XX^\top X$; the scalar polynomial acts on its singular coefficients, not by ordinary rectangular matrix powers. Let $p^{(k)}$ denote this transpose-aware iteration. If the buffer is zero, return a zero update rather than dividing by its norm.

$$O_t = p^{(k)}\left(\frac{B_t}{\|B_t\|_F}\right)$$

   A typical value for k is 5.

3. **Update Step:** Update the weights using the orthogonalized direction, scaled by the learning rate and the MUP factor.

$$W_{t+1} = W_t - \eta\sqrt{\frac{d_{out}}{d_{in}}}O_t$$

Choosing the polynomial $p(x)$: The choice of polynomial is a hyperparameter. There is a trade-off:

- Higher-order polynomials may converge in fewer iterations (k can be smaller).
- However, each iteration is more computationally expensive (requires more matrix multiplications).

Interestingly, the polynomial used in the record-setting NanoGPT speedrun was a quintic:

$$p(x) = 3.444x - 4.7750x^3 + 2.0315x^5$$

This polynomial does not actually converge to 1 (since $p(1) \neq 1$). It provides only an *approximate* orthogonalization. It was chosen empirically because it has a very steep slope at $x = 0$, which aggressively scales up small singular values. This was found to be crucial for accelerating training, suggesting that perfect orthogonalization is not necessary and that sacrificing convergence for faster initial progress can be beneficial.

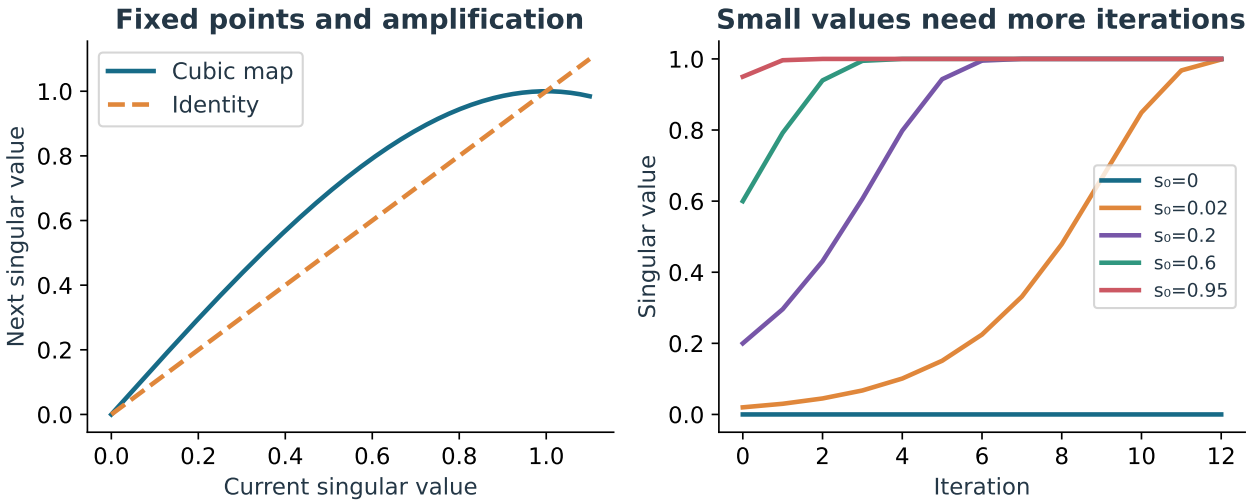


Figure 7.3: Newton–Schulz transforms singular values toward one. Finite iterations lift tiny singular values gradually; a zero singular value remains zero.

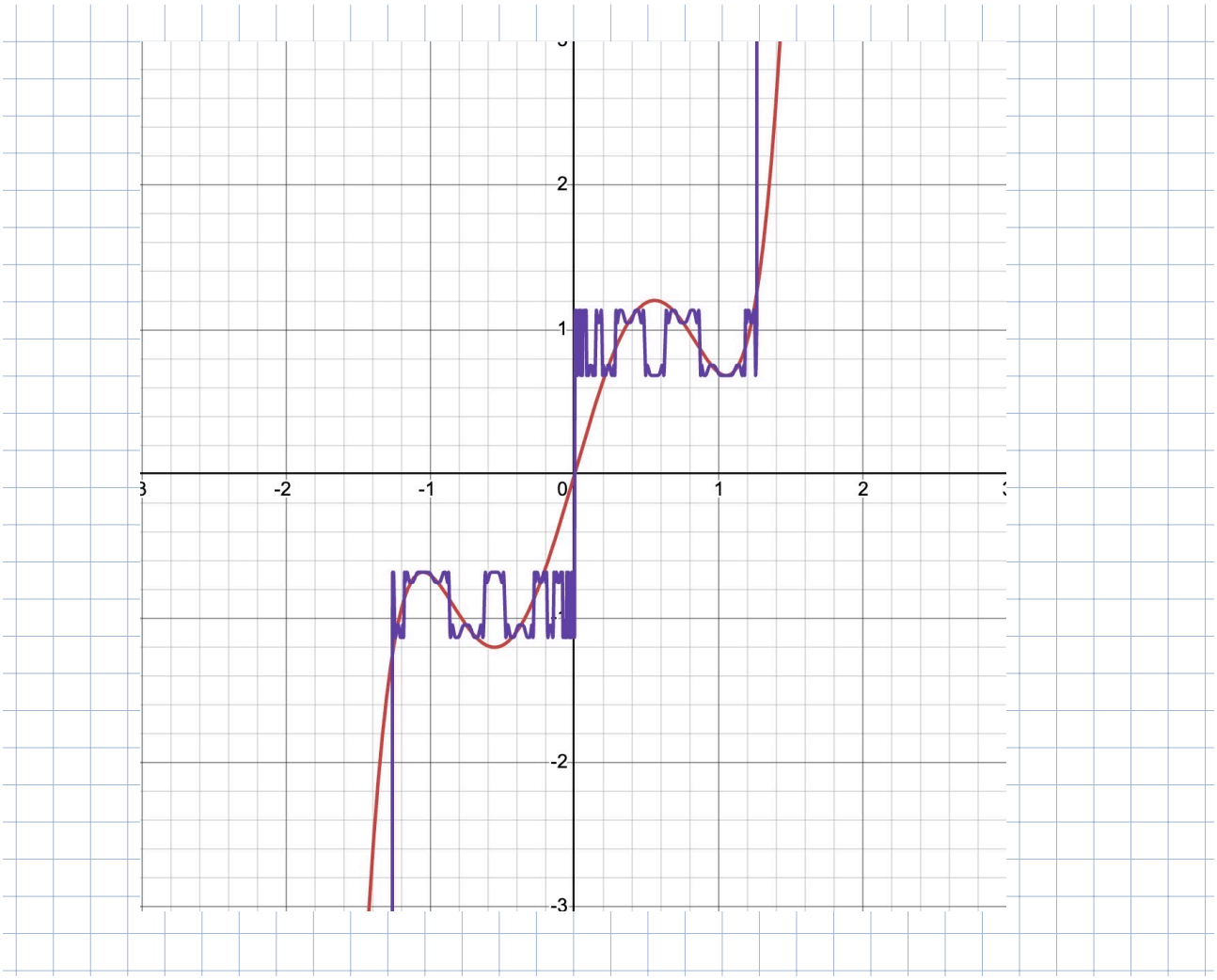


Figure 7.4: Why a practical quintic is run for a fixed budget. The sketch distinguishes rapid approximate flattening from convergence to an exact polar factor. The practical polynomial can oscillate around the target, which is why its finite iteration count is part of the algorithm. Original annotated source excerpt; Gireeja Ranade, faculty handout 07, p. 11.

7.3 Conclusion and Future Directions

Muon was discussed as a competitive optimizer in the Fall 2025 course, building on ideas from Shampoo (semi-orthogonal updates) and MUP (principled hyperparameter scaling). The use of Newton-Schulz iterations provides a computationally feasible way to implement these ideas.

This is a very active area of research. Key practical observations and research questions include:

- **Batch Size:** Larger batches reduce sampling noise under the usual assumptions; whether they improve Muon's compute-adjusted performance is an empirical question.

- **Variants:** Many other variants exist (e.g., Scion, Prawn) that explore different polynomials or implementation details.
- **Fair Comparison:** Comparing optimizers is notoriously difficult. A fair comparison requires extensive, independent hyperparameter tuning for each optimizer on a given task, which is often not done, leading to conflicting reports in the literature.

Lecture 8

Convolutional networks: foundations

8.1 Introduction to Architectures

Course sources: lecture_08; 2025-09-25 transcript.

Deeper explanation: Standard, depthwise, pointwise, and grouped convolutions (p. 385)

This lecture transitions from optimization to the study of neural network architectures. The sequence of architectures covered in this course will be:

1. Multi-Layer Perceptrons (MLPs)
2. Convolutional Neural Networks (CNNs)
3. Graph Neural Networks (GNNs)
4. Recurrent Neural Networks (RNNs) and State-Space Models (SSMs)
5. Transformers

MLPs vs. CNNs: A High-Level View

A key way to understand a new architecture is to contrast it with one we already know.

- **Multi-Layer Perceptron (MLP):** In an MLP layer, every input neuron is connected to every output neuron. This is represented by a dense, unstructured weight matrix where every element is a unique, learnable parameter.
- **Convolutional Neural Network (CNN):** A CNN layer introduces a strong structural prior into the weight matrix. It can be seen as a specialized version of an MLP layer with two defining properties:
    1. **Sparsity (Local Connectivity):** Each output neuron is connected to only a small, local region of the input neurons. This corresponds to a weight matrix that is sparse, with many zero entries representing the absence of connections.
    2. **Parameter (Weight) Sharing:** The same set of weights (known as a filter or kernel) is used for every local connection across the entire input. This means many non-zero entries in the equivalent weight matrix are identical. This drastically reduces the number of unique parameters compared to an MLP.

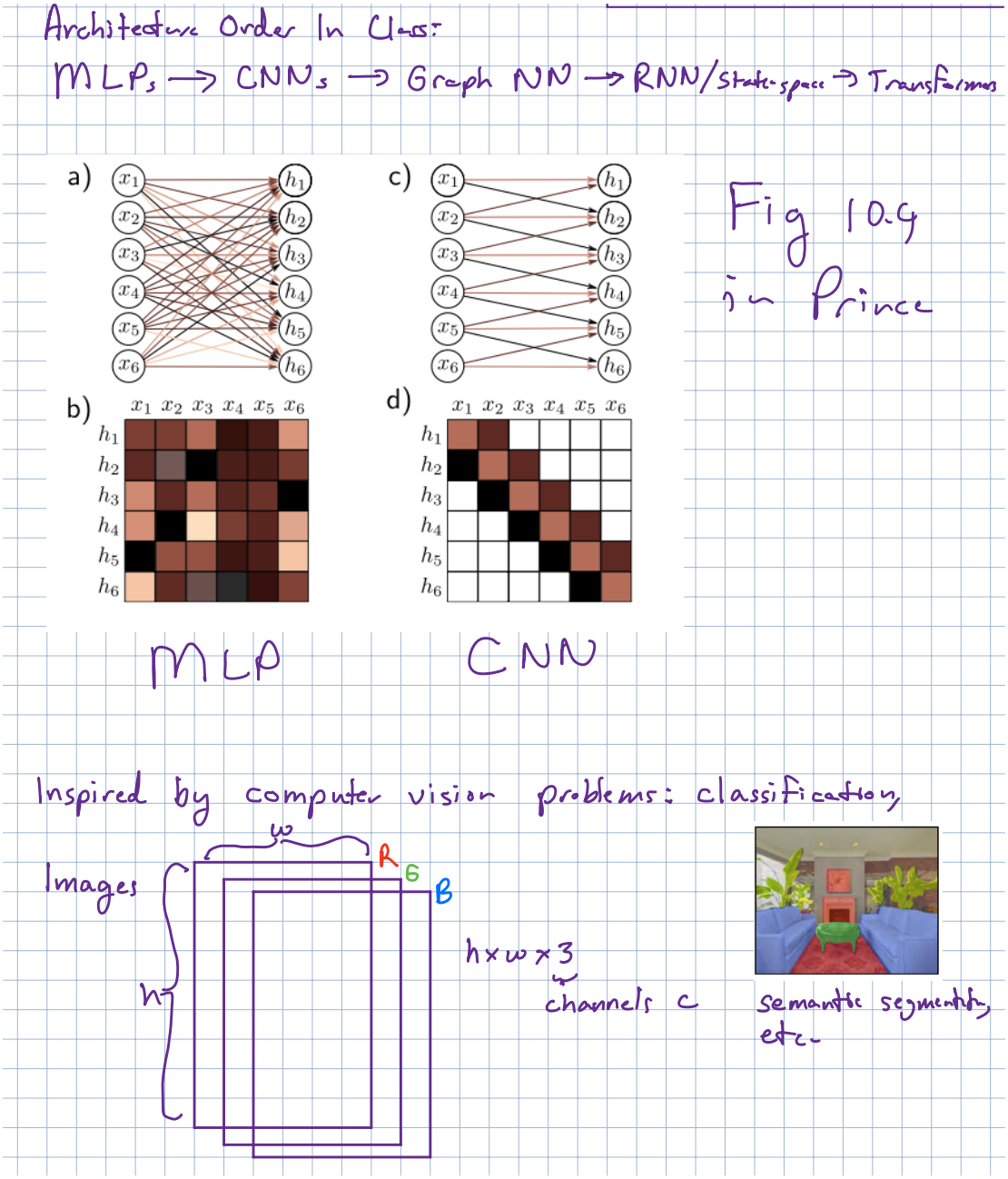


Figure 8.1: Dense connectivity versus structured convolution. The channel sheets and connection patterns contrast an unrestricted dense layer with local, shared interactions. The published diagrams in the faculty handout retain their source presentation. Original annotated source excerpt; Anant Sahai, faculty handout 08, p. 1.

8.2 Motivation and Origins of CNNs

Deeper explanation: Equivariance is not invariance (p. 382)

CNNs were largely inspired by and developed for **computer vision (CV)** tasks.

Input Data: Images

The primary data type for CV is the image. An image is represented as a 3D tensor with dimensions:

- **Height (H)**
- **Width (W)**
- **Channels (C)**

For a standard color image, there are typically 3 channels (Red, Green, Blue), so the shape is $H \times W \times 3$. CNNs are designed to operate on and preserve this spatial tensor structure for as long as possible.

Typical Computer Vision Tasks

- **Image Classification:** Assigning a single label to an entire image. For example, determining if an image from the CIFAR-10 dataset is a 'cat', 'dog', 'car', etc.
- **Semantic Segmentation:** A more complex task where every pixel in the input image is assigned a class label (e.g., 'couch', 'table', 'person'). This requires producing an output that preserves spatial information.

Note: While this is not a computer vision course, understanding the basics of CV and Natural Language Processing (NLP) is crucial because many foundational deep learning advancements originate from these fields.

8.3 Core Principles of CNNs

Deeper explanation: Equivariance is not invariance (p. 382)

The design of CNNs is driven by a set of core ideas. These can be categorized into principles related to **expressivity** (what the model can represent) and principles related to **practicality** (making the model trainable).

Expressivity: Building in Prior Knowledge

These principles are priors about the structure of visual data that are encoded into the architecture.

Locality

- **Premise:** The spatial arrangement of pixels is fundamentally important. Information is local; nearby pixels are strongly correlated, forming meaningful patterns like edges, textures, and corners. Shuffling the pixels of an image destroys its meaning.
- **Architectural Consequence:** The use of a **convolutional structure with small filters** (e.g., 3x3, 5x5). Each filter acts as a local pattern detector.

Invariances, Equivariances, and Symmetries

- **Premise:** The identity of an object or feature is often invariant (or equivariant) to certain transformations. For example, a cat is still a cat if it is shifted 10 pixels to the right in an image. This is known as **translation invariance/equivariance**.
- **Architectural Consequences:**

  1. **Weight Sharing:** This is the direct implementation of translation equivariance. The same filter (a set of weights) is applied at every spatial position in the input. This allows a feature detector (e.g., for a horizontal edge) learned in one part of the image to be immediately reusable across the entire image, without having to re-learn it for each location.
  2. **Data Augmentation:** This is a *training technique*, not an architectural feature. It is used to teach the model other invariances that are not naturally built in, such as invariance to small rotations, scaling, or lighting changes. By showing the model augmented versions of the training data, we force it to learn representations that are robust to these transformations.

Hierarchical Structure

- **Premise:** The visual world is compositional; simple parts combine to form more complex wholes. For instance, edges combine to form corners and contours, which form parts of objects (like an eye), which in turn form larger objects (like a face).
- **Architectural Consequences:**

  1. **Depth:** Stacking multiple convolutional layers allows the network to learn a hierarchy of features. Early layers learn simple features like edges and colors. Deeper layers combine these simple features to detect more complex and abstract concepts.
  2. **Multi-resolution and Filter Banks:** The idea that different features are relevant at different spatial scales. This often manifests in architectures that process the image at multiple resolutions (e.g., through downsampling) and use a varying number of filters (a "filter bank") at different layers, reflecting the increasing complexity and variety of features at higher levels of abstraction.

Practicalities: Getting the Network to Work

These are components and techniques developed primarily to enable stable and effective training of deep networks.

- **Normalization Layers:** (e.g., Batch Normalization, Layer Normalization) These layers do not add expressive power but are crucial for stabilizing training. They control the statistics (e.g., mean and variance) of activations between layers, preventing issues like vanishing or exploding gradients and ensuring that quantities like the RMS norm of layer inputs remain controlled.
- **Dropout:** A regularization technique that randomly sets a fraction of neuron activations to zero during training to prevent overfitting.
- **Residual/Skip Connections:** A powerful technique that adds the input of a layer (or block of layers) to its output. This makes it much easier to train very deep networks by improving gradient flow.

8.4 The Convolution Operation in Deep Learning

Deeper explanation: Backpropagation through shared weights (p. 383)

Convolution vs. Cross-Correlation

In signal processing, the 1D discrete convolution of an input signal x and a filter h is defined as:

$$(x * h)[t] = \sum_{k=-\infty}^{\infty} x[k]h[t-k]$$

The key detail is the $t-k$ term, which implies that the filter h is "flipped" before being slid across the signal.

In deep learning, the operation used is technically **cross-correlation**, where the filter is not flipped:

$$y[t] = \sum_{k} x[t+k]h[k]$$

Since the filter weights $h[k]$ are learned parameters, the network can simply learn the "flipped" version of the filter if that is optimal. Therefore, by convention, this operation is referred to as **convolution** in the deep learning literature.

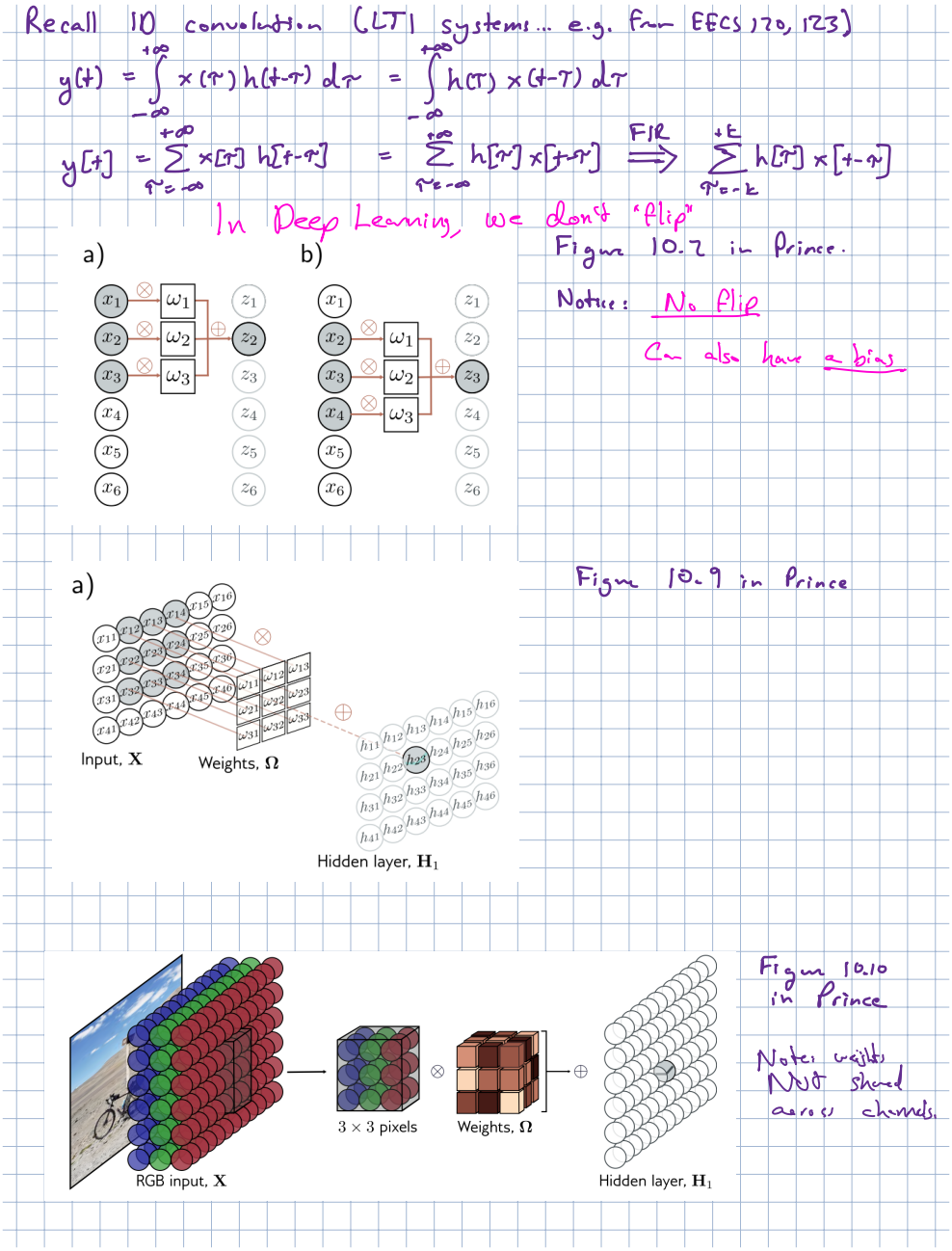


Figure 8.2: From one-dimensional windows to a spatial kernel. The published sequence of convolution diagrams moves from neighboring scalars to a two-dimensional kernel and a multichannel image. Weight sharing refers to reusing the same kernel as the window moves. Original annotated source excerpt; Anant Sahai, faculty handout 08, p. 3.

Mechanics of a Convolutional Layer

A convolutional layer processes an input tensor (called a feature map) to produce an output feature map.

Handling Multiple Channels Let the input feature map have shape $H_{in} \times W_{in} \times C_{in}$.

- **Filters:** A single filter has dimensions $K_h \times K_w \times C_{in}$. Note that the depth of the filter must match the number of input channels.
- **Operation for a single output channel:**
  1. The $K_h \times K_w \times C_{in}$ filter is slid over the input feature map.
  2. At each spatial location (i, j), a dot product is computed between the filter weights and the corresponding $K_h \times K_w \times C_{in}$ patch of the input. This involves performing a 2D convolution for each channel and then summing the results.
  3. A single scalar bias term is added to this sum.
  4. The result is the value at position (i, j) of the single output channel.
- **Multiple Output Channels:** To produce an output feature map with C_{out} channels, we use C_{out} independent filters. Each filter has its own set of weights and its own scalar bias. Each filter produces one channel of the output feature map.

Key Principle: Weights are shared across spatial locations (H, W) but are **not** shared across channels (C).

Special Case: 1x1 Convolution A 1x1 convolution uses a filter of size $1 \times 1 \times C_{in}$.

- It operates on a single pixel location at a time, considering all C_{in} channels for that pixel.
- For a single pixel, the input is a vector $\vec{x}_{\text{pixel}} \in \mathbb{R}^{C_{in}}$.
- To produce an output with C_{out} channels, we use C_{out} different 1×1 filters.
- This entire operation is equivalent to a **dense (fully-connected) linear layer** being applied independently at every pixel location. The weight matrix is $W \in \mathbb{R}^{C_{out} \times C_{in}}$ and the bias is $\vec{b} \in \mathbb{R}^{C_{out}}$.
- The output at a pixel, followed by a non-linearity, is: $\vec{y}_{\text{pixel}} = \text{ReLU}(W\vec{x}_{\text{pixel}} + \vec{b})$.

This perspective is useful for connecting concepts from MLP analysis (like μP) to convolutional networks.

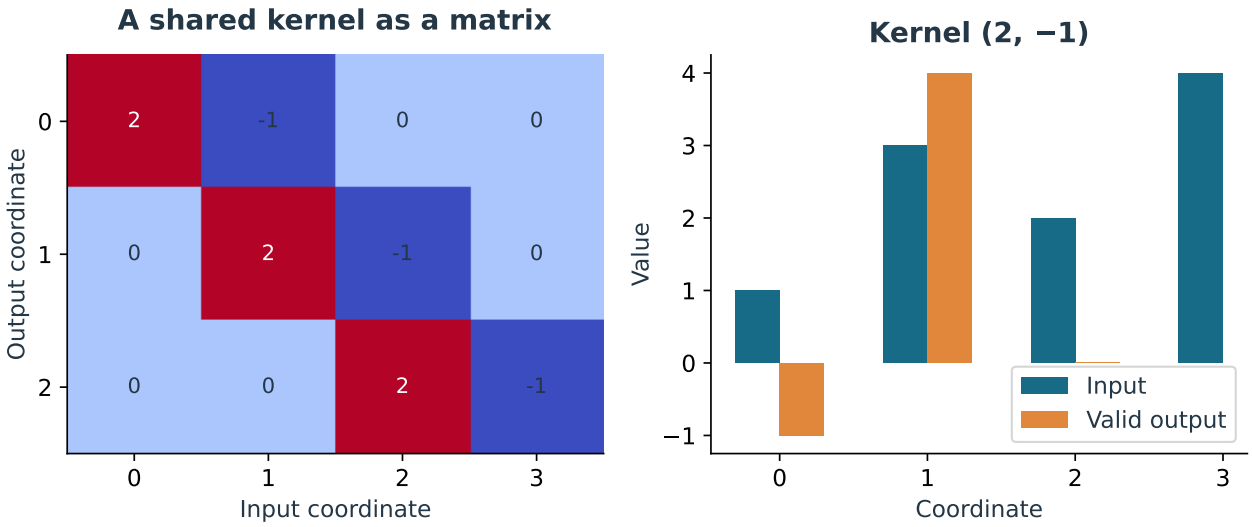


Figure 8.3: Convolution as a sparse matrix. Repeated diagonal entries are the same learned parameter, not independent copies.

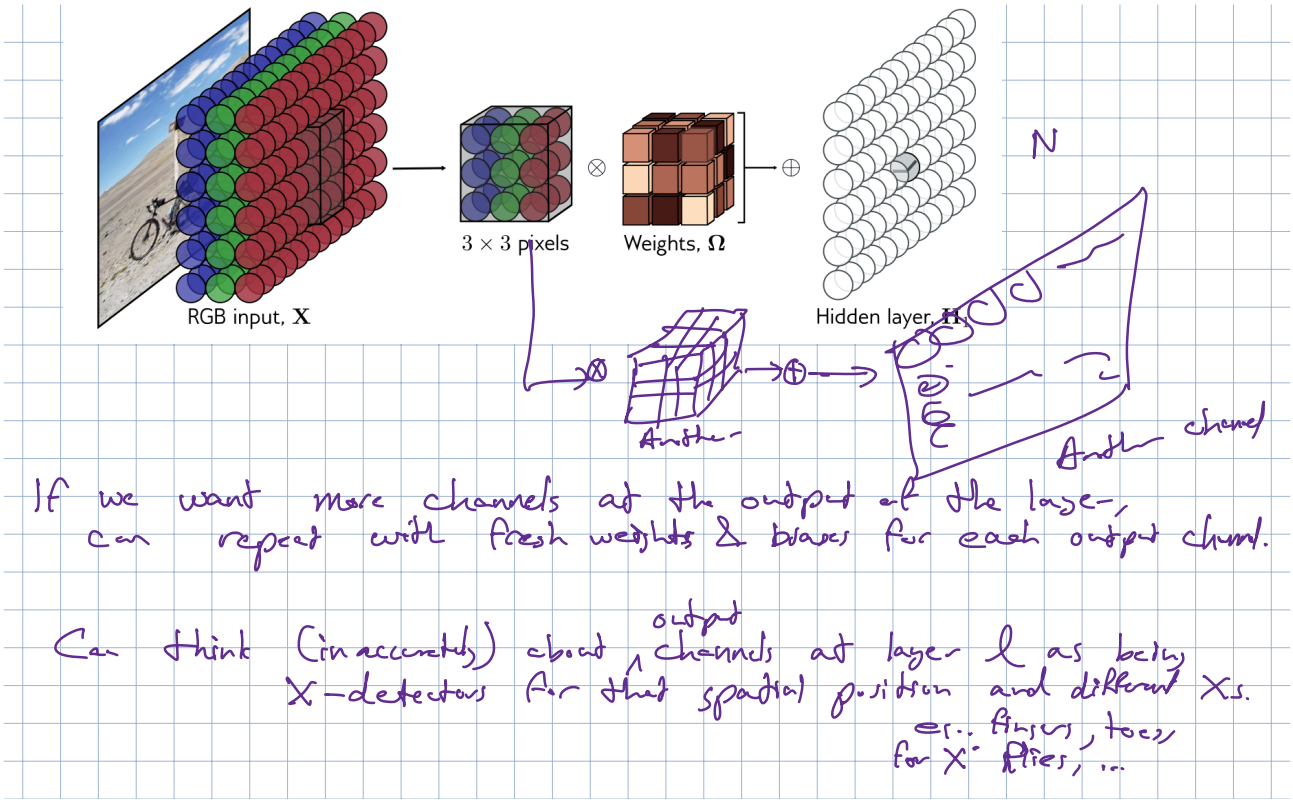


Figure 8.4: From input channels to one output channel. A filter has one spatial kernel per input channel. The channelwise responses are summed to form one output channel; repeating this construction gives multiple output channels. Original annotated source excerpt; Anant Sahai, faculty handout 08, p. 4.

Channel blocks explain convolution and its transpose

Flatten spatial positions separately within each channel. For two input and two output channels, let C_{oi} be the sparse spatial cross-correlation matrix from input i to output o. Then

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \qquad y_o = \sum_{i=1}^{2} C_{oi} x_i.$$

Each block has shape $S_{\text{out}} \times S_{\text{in}}$; repeated entries within a block encode shared kernel weights. The input gradient is

$$\begin{bmatrix} g_{x_1} \\ g_{x_2} \end{bmatrix} = \begin{bmatrix} C_{11}^\top & C_{21}^\top \\ C_{12}^\top & C_{22}^\top \end{bmatrix} \begin{bmatrix} g_{y_1} \\ g_{y_2} \end{bmatrix}.$$

Transposition swaps the channel-block positions as well as transposing each spatial block. Grouped convolution makes cross-group blocks zero. Depthwise convolution is block diagonal across input channels (with possibly several outputs per channel); a following pointwise layer mixes channels at each location. The transpose computes the adjoint, not an inverse of a downsampling operation.

8.5 Architectural Details and Hyperparameters

Deeper explanation: A complete shape-and-cost walkthrough (p. 386)

Handling Boundaries: Padding

When a filter is applied at the edge of an input, it extends beyond the boundary. This leads to two issues:

1. The output feature map is smaller than the input.
2. The pixels at the borders are used less frequently in computations than the interior pixels.

The solution is **padding**: adding extra pixels around the border of the input.

- **Zero Padding:** The most common method, where the padding pixels have a value of zero.
- **Valid Padding:** No padding is used. The output size shrinks. If the input is $W \times W$ and the kernel is $K \times K$, the output is $(W - K + 1) \times (W - K + 1)$.
- **Same Padding:** Padding is added specifically so that the output feature map has the same spatial dimensions (H, W) as the input (assuming a stride of 1).
- **Other types:** Mirror/reflection padding or wrap-around (circular) padding can avoid creating artificial sharp edges at the boundary but are less common in practice.

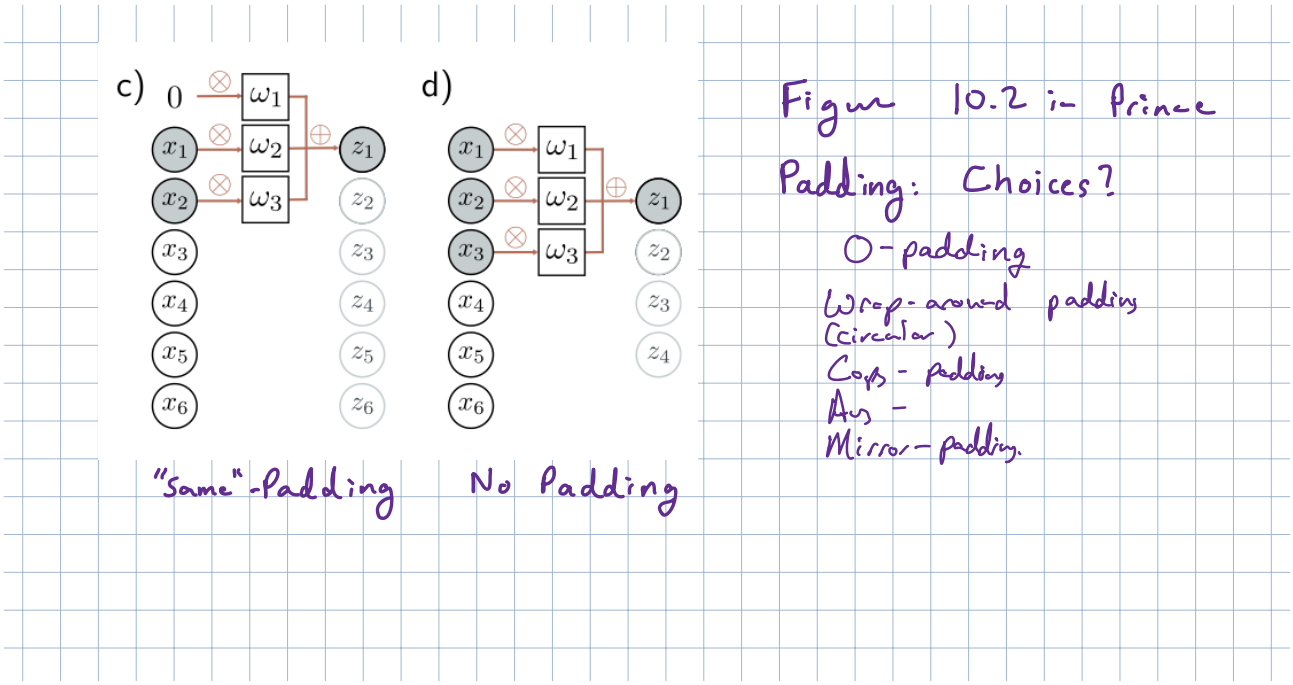


Figure 8.5: Padding changes which neighborhoods are available. Boundary conventions determine which positions can be evaluated and how output size changes. Padding supplies additional values; it does not introduce new learned observations. Published panels are retained from the handout. Original annotated source excerpt; Anant Sahai, faculty handout 08, p. 5.

Receptive Field

The **receptive field** of a neuron in a given layer is the region in the original input image that influences the value of that neuron.

- In a single convolutional layer with a $K \times K$ filter, the receptive field is $K \times K$.
- As layers are stacked, the receptive field grows. For a stack of L layers of 3×3 convolutions (in 1D), the receptive field size is $2L + 1$. This is a **linear growth** with respect to depth.

Downsampling via Stride

To increase the receptive field growth rate and reduce computational cost, we can downsample the feature maps. One way to do this is with a **stride**.

- **Definition:** The stride is the step size the filter takes as it moves across the input. A stride of 1 moves the filter one pixel at a time. A stride of 2 moves it two pixels at a time, skipping every other location.

- **Effect:** A stride greater than 1 results in a smaller output feature map, effectively combining convolution and downsampling into a single efficient operation. This leads to a faster, more multiplicative growth of the receptive field.

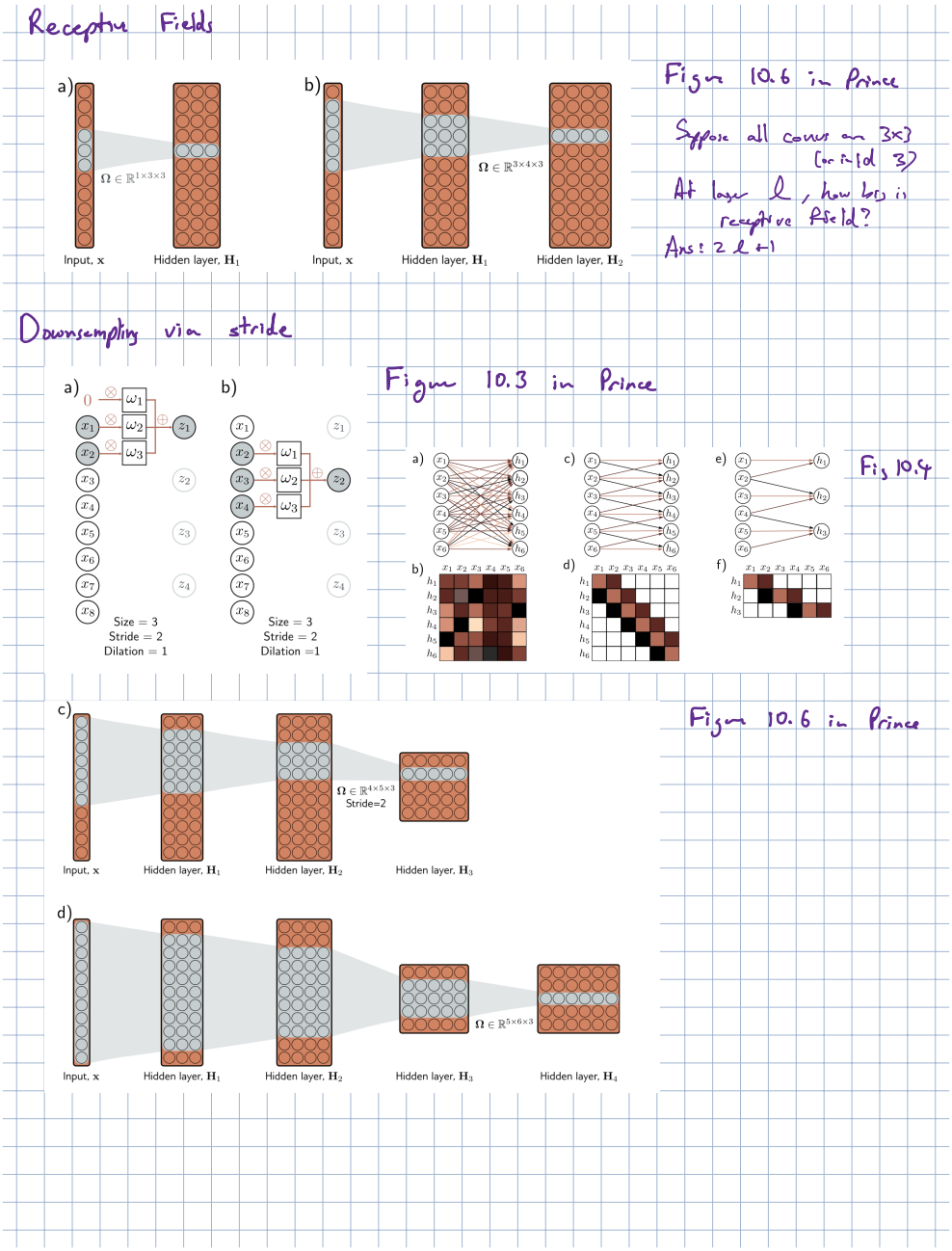


Figure 8.6: Receptive fields, stride, and dilation. Follow the connected input positions to separate three effects: kernel support, spacing between output positions, and spacing within a dilated kernel. The published examples supplement the reader's receptive-field calculation. Original annotated source excerpt; Anant Sahai, faculty handout 08, p. 6.

Worked derivation: output size, receptive fields, and parameter sharing

Lecture basis: faculty handout 08, pp. 3–8. The steps below expand the lecture argument.

Step 1: count valid kernel origins. Consider one spatial axis of input length N, padding p on each side, stride s, kernel size k, and dilation d. The kernel's effective span is $k_{\text{eff}} = d(k-1)+1$. Output index $j \geq 0$ starts at padded coordinate js and ends at $js + k_{\text{eff}} - 1$. It fits exactly when

$$\begin{aligned} js + k_{\text{eff}} - 1 &\leq N + 2p - 1, \\ j &\leq (N + 2p - k_{\text{eff}})/s. \end{aligned}$$

Counting indices starting at zero gives $N_{\text{out}} = \lfloor (N + 2p - k_{\text{eff}})/s \rfloor + 1$, provided at least one placement fits. Apply this independently to height and width.

Step 2: track spacing and receptive-field span separately. Let j_ℓ be the spacing, in original input coordinates, between neighboring outputs of layer ℓ, and r_ℓ the span of one output's receptive field. Initialize $j_0 = r_0 = 1$. Then

$$\begin{aligned} j_\ell &= s_\ell j_{\ell-1}, \\ r_\ell &= r_{\ell-1} + (k_\ell - 1) d_\ell j_{\ell-1}. \end{aligned}$$

The second line follows because the first and last kernel taps are $(k_\ell - 1)d_\ell$ previous-layer positions apart. Three size-3, dilation-1 layers with strides $1, 2, 1$ give $(j, r) = (1, 3), (2, 5), (2, 9)$. Multiplying kernel sizes would give the wrong answer.

Step 3: count distinct weights, not uses. Each output channel needs one $k_h \times k_w$ kernel for each input channel. Thus the weight count is $k_h k_w C_{\text{in}} C_{\text{out}}$, plus C_{out} optional biases. It is independent of image area because the same weight is reused at every spatial position. Depthwise convolution with one output per input channel has $k_h k_w C_{\text{in}}$ weights; a following pointwise mixing layer adds $C_{\text{in}} C_{\text{out}}$. The ratio to an ordinary convolution is $1/C_{\text{out}} + 1/(k_h k_w)$.

Lecture 9

Pooling/Downsampling, Data Augmentation, Normalization Layers

Course sources: lecture_09; 2025-09-30 transcript.

9.1 Recap and Details of Convolutional Layers

Deeper explanation: A complete shape-and-cost walkthrough (p. 386)

The Convolution Operation

Recall the basic picture of a convolution. The topology of the input matters; local neighborhoods are fundamental.

- **1D Case:** A sliding window (filter or kernel) moves across a 1D input. For each position, the values in the window are multiplied by corresponding weights in the filter, summed up, and a bias is added to produce a single output value.
- **2D Case:** This extends the 1D case. The neighborhood is now a 2D patch (e.g., 3×3). The filter is a 2D matrix of weights. The operation is the same: element-wise multiplication and summation over the patch, plus a bias.
- **Multi-Channel Input:** If the input has multiple channels (e.g., an RGB image has 3 channels), the filter must also have a depth corresponding to the number of input channels. For a 3×3 filter on an RGB image, the filter is actually a $3 \times 3 \times 3$ tensor of weights. Each channel of the input is convolved with its corresponding channel in the filter, and the results are all summed together with a single bias to produce a single value for one output channel.

Notation Note: In the context of CNNs, the filters are often called **kernels**. This is a different usage of the word "kernel" than in kernel methods like Support Vector Machines (SVMs).

Understanding The Number of Parameters

Let's break down the number of learnable parameters (weights and biases) in different convolutional layer configurations. Let C_{in} be the number of input channels and C_{out} be the number of output channels.

1. **1x1 convolution with 1 input channel and 1 output channel:**
   - For each pixel, the operation is simply scaling the input value by a constant and adding a bias.
   - **Weights:** 1
   - **Biases:** 1
2. **3x3 convolution with 1 input channel and 1 output channel:**
   - For each pixel, the values in its 3×3 neighborhood are scaled by individual, distinct weights. These are summed, and a single bias is added.
   - The filter consists of 9 weights (one for the center, N, S, E, W, NE, NW, SE, SW neighbors).
   - **Weights:** $3 \times 3 = 9$
   - **Biases:** 1
3. **3x3 convolution with C_{in} input channels and 1 output channel:**
   - The process from the previous case is repeated for each of the C_{in} input channels. Each input channel has its own distinct 3×3 filter.
   - The results from all C_{in} convolutions are summed up, and a single bias is added to produce the final single output value.
   - **Why only one bias?** If we had C_{in} separate biases, one for each channel's convolution, they would all be summed together into a single scalar anyway. In terms of expressive power, a single bias is equivalent. From a backpropagation perspective, the gradient would be identical for all C_{in} hypothetical biases.
   - **Weights:** $9 \times C_{in}$
   - **Biases:** 1
4. **k x k convolution with C_{in} input channels and 1 output channel:**
   - Generalizing the above case.
   - **Weights:** $k^2 \times C_{in}$
   - **Biases:** 1
5. **1x1 convolution with C_{in} input channels and C_{out} output channels:**
   - At each pixel, the input is a vector of size C_{in} (the values across all channels). The output is a vector of size C_{out}.
   - The operation that transforms a vector of size C_{in} to a vector of size C_{out} is a matrix-vector multiplication plus a bias vector. Let $\vec{x}$ be the input vector and $\vec{y}$ be the output vector: $\vec{y} = \mathbf{W}\vec{x} + \vec{b}$, where $\mathbf{W}$ is a $C_{out} \times C_{in}$ matrix and $\vec{b}$ is a C_{out}-dimensional vector.

- **Weights:** $C_{in} \times C_{out}$
- **Biases:** C_{out}

6. **k x k convolution with C_{in} input channels and C_{out} output channels:**
   - This is the most general case. It combines the logic from all previous examples. For each of the C_{out} output channels, we perform a complete $k \times k$ convolution over all C_{in} input channels.
   - Each output channel is generated by its own unique set of filters. There are C_{out} sets of filters, and each set contains C_{in} filters of size $k \times k$.
   - **Weights:** $k^2 \times C_{in} \times C_{out}$. These can be thought of as a 4D tensor of shape (C_{out}, C_{in}, k, k).
   - **Biases:** C_{out} (one for each output channel).

9.2 Backpropagation in CNNs

Deeper explanation: Backpropagation through shared weights (p. 383)

A key feature of CNNs is **weight sharing**: the same filter (parameter w) is used at many different locations in the input. This means the parameter w appears in multiple places in the computational graph. How do we compute the gradient $\frac{\partial \mathcal{L}}{\partial w}$?

The Conceptual Approach

1. **Pretend weights are distinct:** Imagine that each application of the filter uses a unique, independent weight $(w_1, w_2, \dots, w_N)$, which all just happen to have the same initial value, $w_i = w$.
2. **Compute individual gradients:** During backpropagation, you would compute the gradient of the loss with respect to each of these hypothetical independent weights: $\frac{\partial \mathcal{L}}{\partial w_1}, \frac{\partial \mathcal{L}}{\partial w_2}, \dots$.
3. **Sum the gradients:** To get the final gradient for the single shared weight w, you sum up all the individual gradients. This is a direct consequence of the multivariate chain rule.

$$\frac{\partial \mathcal{L}}{\partial w} = \sum_i \frac{\partial \mathcal{L}}{\partial w_i}$$

This summation is analogous to the principle of superposition in physics. Each application of the weight contributes linearly to the total change in the loss, so their effects on the gradient add up.

Practical Considerations: Sum vs. Average

The gradient is a sum, not an average. The reason is that a small change in w affects the output at *every* location it is applied. The total effect on the loss is the sum of these individual effects.

However, this summation can cause the magnitude of the gradient to grow significantly, especially for large images where a filter is applied many times.

- If the individual gradients are uncorrelated, the standard deviation of the sum grows like $\sqrt{N}$, where N is the number of applications.
- If they are correlated, the sum can grow like N.

In standard 32-bit floating-point arithmetic (FP32), this is usually manageable. However, with lower precision formats like FP16 or FP8, this can lead to numerical overflow. Techniques like **mixed-precision training** (where intermediate sums are kept in a higher precision format) are used to mitigate this.

9.3 Pooling and Downsampling

Deeper explanation: Pooling, downsampling, and aliasing (p. 383)

Pooling layers reduce the spatial dimensions of the feature maps, which helps to reduce the number of parameters and computations, and also provides a degree of translation invariance.

- **Striding:** The simplest form of downsampling. Instead of sliding the convolutional filter one pixel at a time, we move it by a stride of $S > 1$. For example, a stride of 2 keeps only one out of every four outputs in a 2x2 region. Backpropagation is straightforward.
- **Mean Pooling:** The output is the average of the values in a pooling window (e.g., a 2×2 region).

  $$\text{Output} = \frac{1}{|\text{Pooling domain}|} \sum_{i \in \text{Pooling domain}} h_i$$

  This is equivalent to a convolution with a fixed (non-learnable) filter where all weights are $\frac{1}{|\text{Pooling domain}|}$. During backpropagation, the gradient from the output is distributed equally among all the inputs in the pooling window.
- **Max Pooling:** The output is the maximum value in the pooling window.

  $$\text{Output} = \max_{i \in \text{Pooling domain}} h_i$$

  The intuition is to capture the presence of a feature. If a "finger detector" fires strongly anywhere in the region, the max pool output will be high.

– **Backpropagation for Max Pooling:** The gradient is passed back only to the input unit that had the maximum value (the 'argmax'). All other inputs in the window receive a gradient of zero. This is because infinitesimal changes to non-maximal values do not affect the output of the max function. The operation acts as a "gradient router".

Worked derivation: shared-weight gradients and pooling gradients

Lecture basis: faculty handout 09, pp. 4–6. The steps below expand the lecture argument.

Step 1: write one indexed cross-correlation. Use one spatial axis for readability, with padded inputs treated as zero:

$$z_{o,t} = b_o + \sum_c \sum_{a=0}^{k-1} W_{o,c,a} x_{c,st+a-p}, \qquad \delta_{o,t} = \frac{\partial L}{\partial z_{o,t}}.$$

For a fixed shared coefficient $W_{o,c,a}$, every output position using it contributes:

$$\begin{aligned} \frac{\partial L}{\partial W_{o,c,a}} &= \sum_t \frac{\partial L}{\partial z_{o,t}} \frac{\partial z_{o,t}}{\partial W_{o,c,a}} \\ &= \sum_t \delta_{o,t} x_{c,st+a-p}, \\ \frac{\partial L}{\partial b_o} &= \sum_t \delta_{o,t}. \end{aligned}$$

For an input coordinate $x_{c,u}$, collect all outputs whose kernel touches it:

$$\frac{\partial L}{\partial x_{c,u}} = \sum_{o,t,a:\, st+a-p=u} \delta_{o,t} W_{o,c,a}.$$

These are sums because of the chain rule. A batch-mean loss introduces its batch normalization factor upstream; weight sharing does not independently justify averaging over image positions.

Step 2: differentiate the pooling map. For mean pooling over a set S, $z = |S|^{-1} \sum_{i \in S} x_i$, so $\partial L / \partial x_i = \delta / |S|$ for every member. For max pooling with a unique maximum i_*,

$$\frac{\partial L}{\partial x_i} = \delta \mathbf{1}_{i=i_*}.$$

At ties the ordinary derivative is not unique; an implementation chooses a routing rule or subgradient. If pooling windows overlap, an input receives the sum of contributions from every window containing it.

9.4 Classic CNN Architectures

Deeper explanation: Standard, depthwise, pointwise, and grouped convolutions (p. 385)

Early CNNs combined these building blocks in a characteristic pattern.

- **AlexNet (2012):** A seminal network that popularized deep CNNs.
  - **Structure:** A series of convolutional layers (often with non-linearity like ReLU), followed by max pooling layers. This pattern reduces the spatial dimensions ($224 \times 224 \to 55 \times 55 \to 27 \times 27 \dots$) while increasing the number of channels ($3 \to 96 \to 256 \dots$).
  - **End:** After several conv/pool blocks, the final feature map is flattened into a single vector and fed into a few fully connected (MLP) layers, ending in a softmax output for classification.
- **VGG (2014):** Followed a similar philosophy but used a more uniform architecture with stacks of small (3×3) convolutional filters, demonstrating that depth was a key component for performance. A stack of 3×3 convolutional layers has more non-linearities than a single larger filter, increasing expressive power.

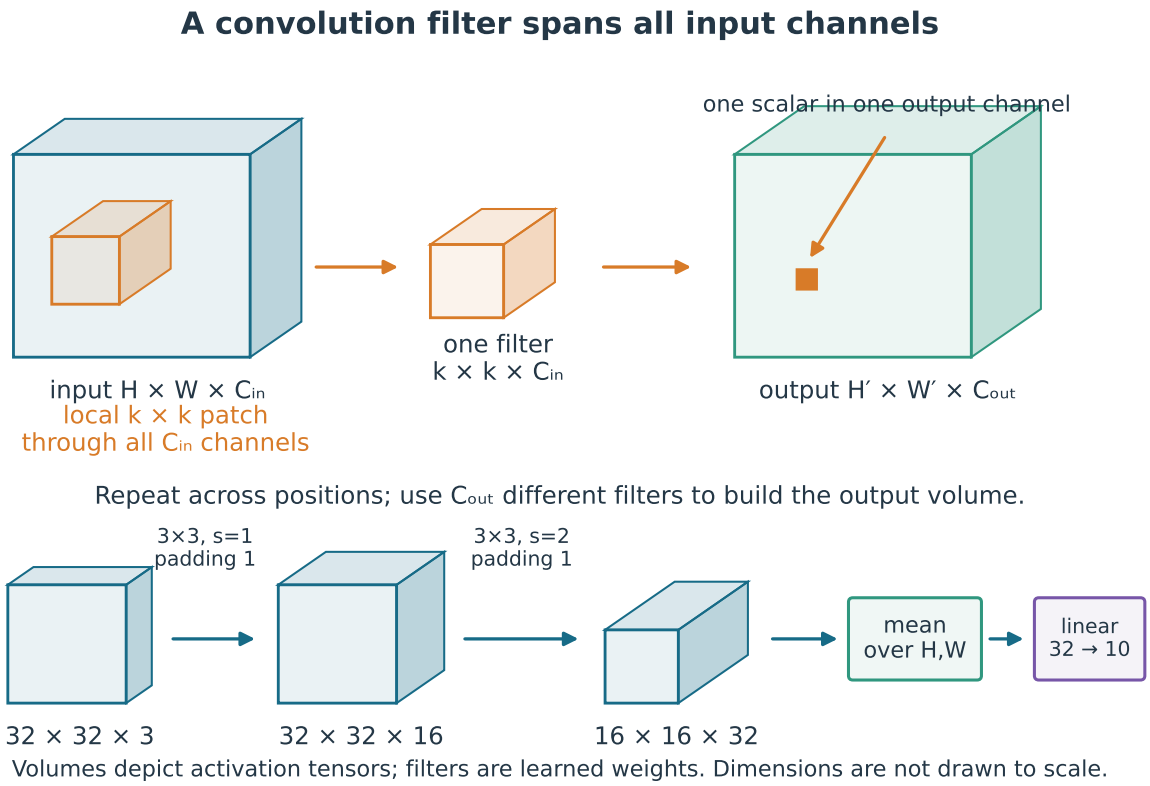


Figure 9.1: A standard convolution uses one full-depth spatial filter per output channel. The orange patch contributes to one output scalar by multiplication and summation over spatial offsets and input channels. The lower path visualizes the chapter's complete CNN example, including global average pooling and a classifier.

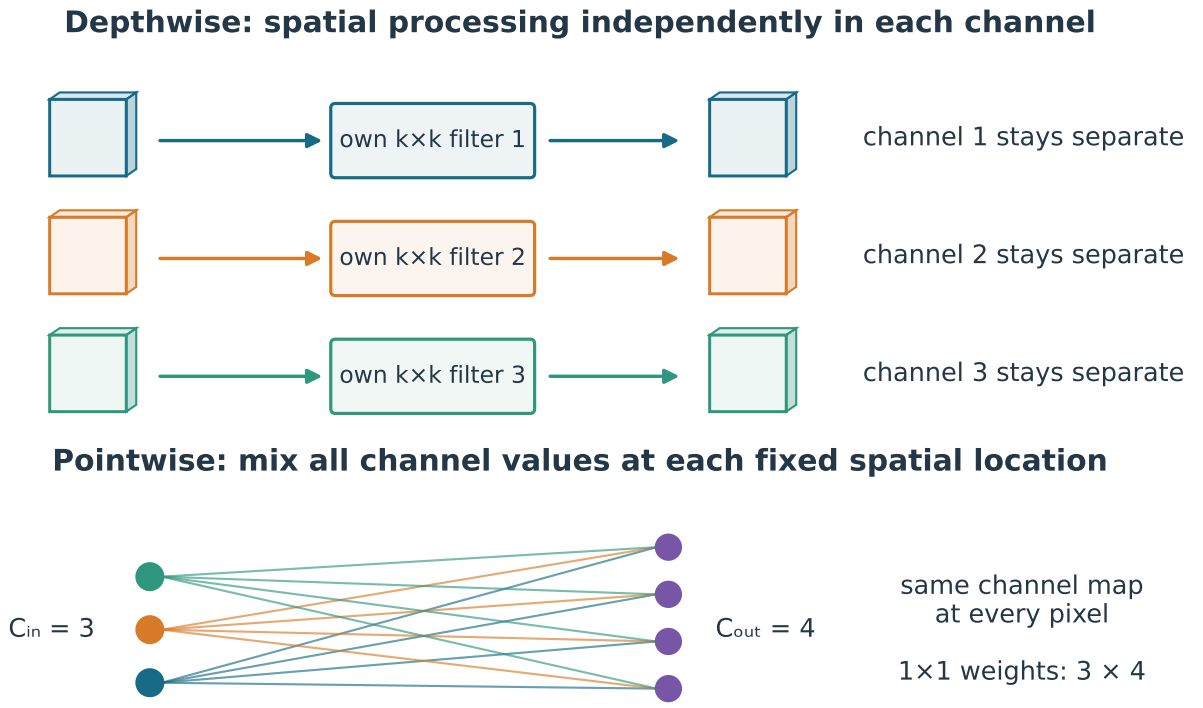


Figure 9.2: Depthwise filtering and pointwise mixing separate two roles of a standard convolution. A depthwise layer with multiplier one preserves channel count and has no cross-channel edges. A pointwise layer reconnects channels independently at each spatial site.

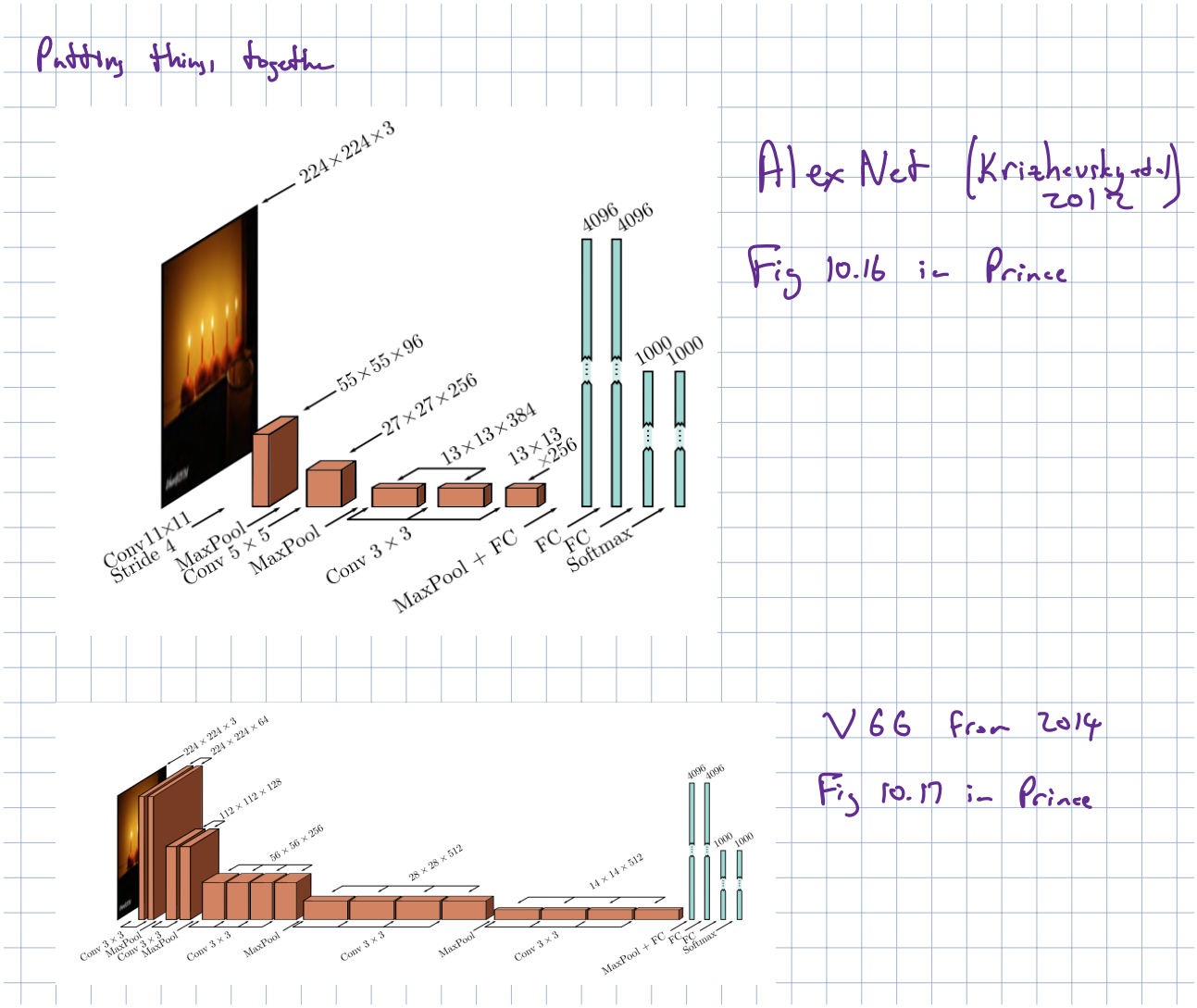


Figure 9.3: AlexNet and VGG as sequences of spatial and channel changes. Track shrinking spatial dimensions and increasing feature depth through the architectures. These historical examples establish the convolution–nonlinearity–downsampling pattern; published architecture panels are preserved from the handout. Original annotated source excerpt; Anant Sahai, faculty handout 08, p. 7.

9.5 Data Augmentations

Deeper explanation: Data augmentation changes the training objective (p. 384)

CNNs build in translation invariance through weight sharing. To achieve invariance to other transformations (like rotation, scaling), we use data augmentation. The idea is to artificially expand the training dataset by creating modified copies of the data.

The goal is to shape the **inductive bias** of the model—the types of patterns it prefers to learn. This can be done via architecture (weight sharing), optimization, or data augmentation.

Basic Augmentations

These augmentations aim to make the training distribution better reflect the true data distribution one might encounter. They are typically applied randomly to each batch during training.

- `autocontrast`, `equalize`: Adjusting image contrast and brightness.
- `rotate`, `shear_x/y`, `translate_x/y`: Geometric transformations.
- `posterize`, `solarize`: Color and pixel value transformations.

With these augmentations, the network rarely sees the exact same input twice, even over many epochs.

Aggressive Augmentations for Robustness

These methods modify the training data in ways not typically expected in the test data. The goal is to make the training task harder, forcing the model to learn more robust and generalizable features.

- **Cutout:** Randomly removes a square patch from an image, forcing the model to use context rather than relying on a single salient feature.
- **Mixup:** Linearly interpolates between two images and their labels. For images x_1, x_2 with one-hot labels y_1, y_2:

$$x_{new} = \lambda x_1 + (1-\lambda)x_2 \quad , \quad y_{new} = \lambda y_1 + (1-\lambda)y_2$$

- **CutMix:** A patch from one image is cut and pasted onto another image. The labels are mixed proportionally to the area of the patches.
- **PixMix:** A more complex method that mixes an image with various "psychedelic" fractal images to act as structured noise, forcing the model to ignore compelling but irrelevant distractors.



Figure 9.4: Image-level examples of aggressive augmentation. Cutout, Mixup, CutMix, and PixMix alter examples in different ways. The images make the distinction visible; label handling must follow the chosen augmentation. Original published panels and credits in the notes are retained. Original annotated source excerpt; Anant Sahai, faculty handout 09, p. 7.

9.6 Training Stability: Normalization Layers

Deeper explanation: Deriving the layer-normalization backward map (p. 389)

A deep network can suffer from training instability. As activations and gradients propagate forward and backward through many layers, their magnitudes can explode (leading to 'NaN's) or vanish. This is especially true for deep networks where effects multiply.

Normalization layers are inserted between other layers to control the distribution of activations. They act as "speed bumps" for activations and gradients, ensuring they stay in a reasonable range. The high-level idea is to perform something akin to input standardization, but for the inputs to intermediate layers.

RMS Norm Layer

This is a simple normalization layer motivated by ensuring the assumptions of schemes like Maximal Update Parametrization (MuP) hold throughout training. MuP assumes inputs have an RMS norm of 1. While initialization can ensure this, the norm can drift during training. An RMS Norm layer explicitly enforces it.

- Given a hidden activation vector $\vec{h}$, the normalized output $\tilde{\vec{h}}$ is:

$$\tilde{\vec{h}} = \frac{\vec{h}}{\sqrt{\frac{1}{d}\sum_{j=1}^{d} h_j^2 + \epsilon}} = \frac{\vec{h}}{\text{RMS}(\vec{h})\sqrt{d} + \epsilon}$$

  where d is the dimension of the vector.
- The output $\tilde{\vec{h}}$ will have an RMS norm of 1 (ignoring ϵ).
- This version has no learnable parameters and is effective with certain modern optimizers.

Batch Normalization (BatchNorm)

One of the first and most widely used normalization layers. It standardizes the activations for each channel by computing the mean and standard deviation over a mini-batch of data.

For a feature map in a CNN, we have dimensions for batch (N), channels (C), height (H), and width (W). BatchNorm computes statistics by averaging over the batch and spatial dimensions (N, H, W) independently for each channel.

Let $\mathcal{B}$ be the set of activations for a single channel over a mini-batch (i.e., across N, H, and W).

1. **Compute mini-batch mean:**

$$\mu_{\mathcal{B}} = \frac{1}{|\mathcal{B}|}\sum_{x \in \mathcal{B}} x$$

2. **Compute mini-batch variance:**

$$\sigma_{\mathcal{B}}^2 = \frac{1}{|\mathcal{B}|}\sum_{x \in \mathcal{B}} (x - \mu_{\mathcal{B}})^2$$

3. **Normalize:**
$$\widehat{x} = \frac{x - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$$
This normalizes the activations to have zero mean and unit variance over the mini-batch.

4. **Scale and Shift:** The network might need to learn a different distribution. To allow this, BatchNorm introduces two learnable parameters per channel, γ (scale) and δ (shift).
$$y = \gamma\widehat{x} + \delta$$
If the network learns $\gamma = \sqrt{\sigma_{\mathcal{B}}^2}$ and $\delta = \mu_{\mathcal{B}}$, it can recover the original activation, so no expressive power is lost.

Lecture 10

Normalization Layers, Dropout, Residual Connections

Course sources: lecture_10; 2025-10-02 transcript.

10.1 Convolutional Neural Networks (Continued)

Deeper explanation: A complete shape-and-cost walkthrough (p. 386)

We continue our discussion of Convolutional Neural Networks (CNNs). The core idea remains using a convolutional architecture for weight sharing, which is particularly effective for image processing. We will first clarify a key concept from the previous lecture: the two perspectives on a $k \times k$ convolution.

Two Perspectives on a $k \times k$ Convolution

Consider a convolutional layer with a $k \times k$ filter, C_{in} input channels, and C_{out} output channels. The number of parameters is identical regardless of perspective: $k^2 \cdot C_{\text{in}} \cdot C_{\text{out}}$ weights and C_{out} biases.

Perspective 1: C_{out} Separate Filters This perspective views the operation as applying C_{out} distinct $k \times k$ convolutional filters. Each of these filters has:

- C_{in} input channels.
- 1 output channel.

The process for a single output channel is as follows:

1. A single $k \times k \times C_{\text{in}}$ filter is applied to a patch of the input tensor.
2. For each of the C_{in} input channels, there is a corresponding $k \times k$ matrix of real-valued weights within the filter.
3. The values in the input patch are multiplied element-wise with the corresponding filter weights, and all results are summed together.
4. A single bias term is added to this sum to produce one value in the output feature map.

This entire process is repeated C_{out} times with C_{out} different filters to generate the C_{out} output channels.

Perspective 2: A Single Filter with Weight Matrices This perspective views the operation as a single $k \times k$ convolution where the components are not scalars, but matrices and vectors.

- Each of the k^2 spatial positions in the filter contains a **weight matrix** of shape $C_{\text{in}} \times C_{\text{out}}$.
- There is a single **bias vector** $\vec{b}$ of dimension C_{out} for the entire convolution.

This perspective treats the values at each pixel as a vector. For an RGB image, this is a 3-dimensional vector. Deeper in the network, it is a C_{in}-dimensional vector. The convolution operation at a single output pixel is then a linear transformation on vectors:

$$\vec{h}_{\text{out}} = \vec{b} + \sum_{i \in \text{neighborhood}} W_i \cdot \vec{h}_{\text{in},i}$$

Where:

- $\vec{h}_{\text{out}}$ is the C_{out}-dimensional output vector at a pixel.
- $\vec{b}$ is the C_{out}-dimensional bias vector.
- The summation index i ranges over the k^2 spatial positions in the convolutional filter's neighborhood (e.g., for a 3×3 filter, this includes center, north, south, east, west, etc.).
- W_i is the $C_{\text{out}} \times C_{\text{in}}$ weight matrix for the i-th spatial position in the filter.
- $\vec{h}_{\text{in},i}$ is the C_{in}-dimensional input vector from the corresponding spatial position in the input feature map.

Both perspectives are computationally identical and represent the same affine transformation from one image representation to another.

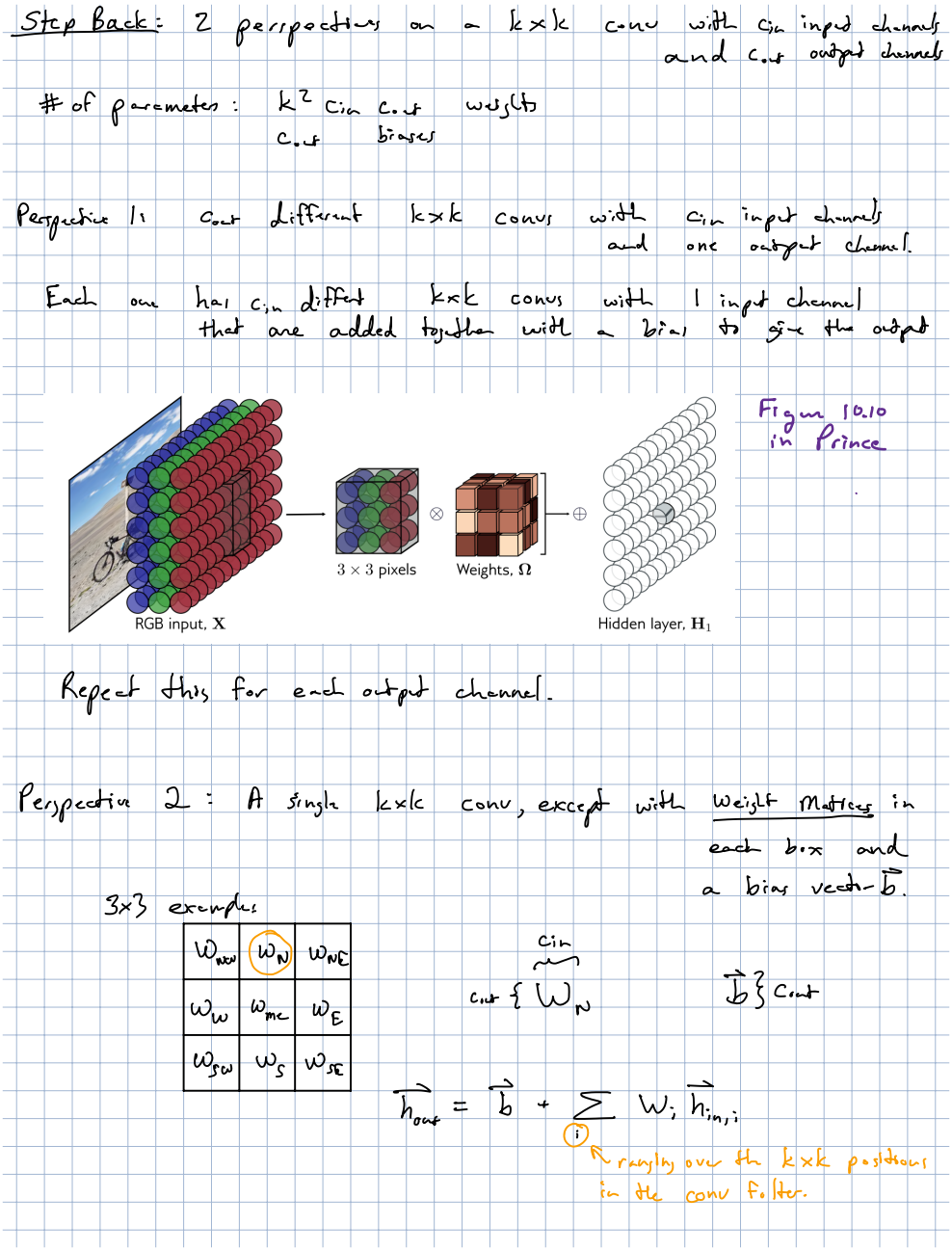


Figure 10.1: Two equivalent views of multichannel convolution. One view groups weights by spatial position; the other groups them by input/output channel pair. Both implement the same contraction over spatial offsets and input channels. Original annotated source excerpt; Anant Sahai, faculty handout 10, p. 2.

10.2 Training: Stability and Effectiveness

Deeper explanation: Batch normalization has two different operating modes (p. 390)

To train deep networks effectively, we must manage the scale of activations and gradients to prevent them from exploding or vanishing. This leads to the idea of inserting normalization layers within the network architecture.

Normalization Layers

The core idea is inspired by input standardization. We want to control the distribution of activations at hidden layers.

RMS Norm Layer A simple approach is to normalize the activations to have a unit Root Mean Square (RMS) norm. For a d-dimensional input vector $\vec{h}$, the normalized output $\vec{\bar{h}}$ is:

$$\vec{\bar{h}} = \frac{\vec{h}}{||\vec{h}||_{\text{rms}}} \quad \text{where} \quad ||\vec{h}||_{\text{rms}} = \sqrt{\frac{1}{d}\sum_{i=1}^{d} h_i^2 + \epsilon}$$

The small constant $\epsilon > 0$ is added for numerical stability to prevent division by zero.

Batch Normalization (Batch Norm) Batch norm standardizes the inputs for each feature channel separately, using statistics computed over the current mini-batch.

- **Averaging Domain**: For a given channel, statistics are computed by averaging over all spatial dimensions (height H, width W) and all examples in the mini-batch (size N).
- Let $\mathcal{B}$ be the set of activations for a single channel across the batch and spatial dimensions. The size of $\mathcal{B}$ is $N \times H \times W$.
- The mean m and standard deviation s are computed as:

$$m = \frac{1}{|\mathcal{B}|}\sum_{h_i \in \mathcal{B}} h_i$$

$$s = \sqrt{\frac{1}{|\mathcal{B}|}\sum_{h_i \in \mathcal{B}} (h_i - m)^2 + \epsilon}$$

- The normalization is then applied element-wise:

$$\hat{h}_i = \frac{h_i - m}{s}$$

- **Learnable Parameters**: Batch norm introduces learnable per-channel scale (γ) and shift (δ) parameters. This allows the network to learn the optimal mean and variance for the activations, rather than being fixed at 0 and 1.

$$h_{\text{out}} = \gamma \hat{h}_i + \delta$$

These are initialized with $\gamma = 1$ and $\delta = 0$. Typically, weight decay is not applied to these parameters.

Layer Normalization (Layer Norm) Layer norm computes statistics differently, making it independent of the batch size.

- **Averaging Domain**: For a single example in the batch, statistics are computed by averaging over all channels (C) and all spatial dimensions (H, W).
- The mean and standard deviation are computed for each example in the batch independently.
- This approach works well even with a batch size of 1.
- Similar to batch norm, layer norm can also have learnable per-channel scale (γ) and shift (δ) parameters.

Other Variants

- **Instance Normalization**: Statistics are computed per-channel and per-example, averaging only over the spatial dimensions (H, W).
- **RMS Norm Variants**: Any of these normalization schemes can be implemented without the recentering step (i.e., without computing and subtracting the mean m). This is often referred to as RMSNorm and is typically done in the style of Layer Norm.

Normalization at Test/Inference Time A key issue arises with methods that depend on batch statistics.

- **Layer Norm**: No issue. Since it operates on a single example at a time, the procedure at test time is identical to training.
- **Batch Norm**: At inference time, there is no batch to compute statistics over. We need a fixed mean m and standard deviation s. Several strategies exist:
  1. **Use Training Statistics**: During training, maintain a running exponential moving average of the mean and variance for each channel. These moving averages are then "frozen" and used at inference time. An optional recalibration procedure is to compute statistics on representative training data after training is complete (e.g., by running an extra epoch with no gradient updates).
  2. **Test-Time Adaptation**: For applications where the data distribution might shift (e.g., autonomous driving), one can continue to update the running average of m and s during inference.
  3. **Learn as Hyperparameters**: Treat m and s as parameters to be learned on a held-out validation set.

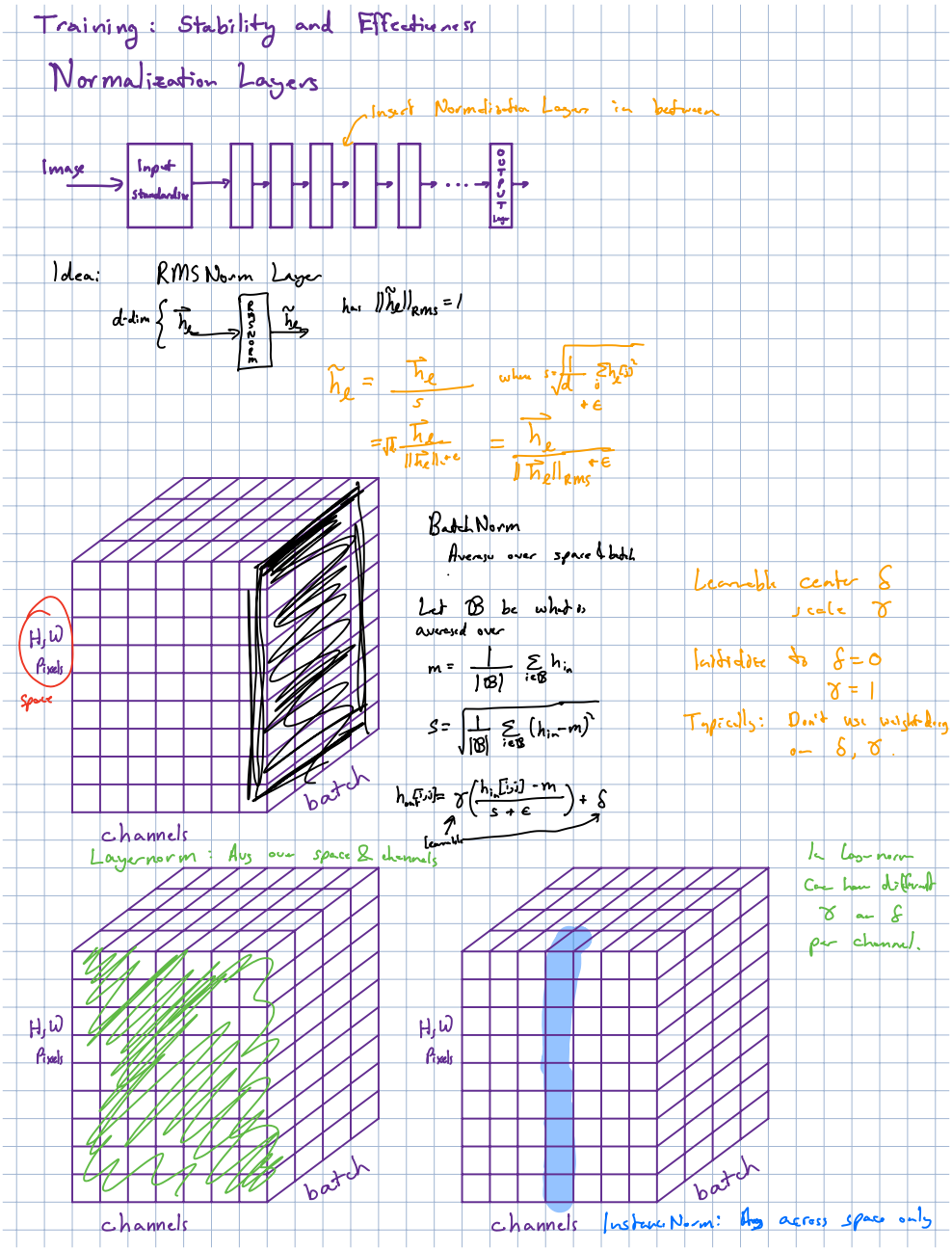


Figure 10.2: The professors' normalization cubes. Each highlighted slice specifies the coordinates used for a statistic. These CNN cubes use channel, spatial, and batch axes; the reader's token-wise LayerNorm and RMSNorm diagrams instead reduce over the feature axis at fixed token and example. Instance normalization is a different choice of slice. Original annotated source excerpt; Anant Sahai, faculty handout 10, p. 3.

Token-wise normalization has a block-diagonal Jacobian

For the token-wise variant, flatten a tensor (B, T, D) into $N = BT$ feature vectors $x_j \in \mathbb{R}^D$. Each normalized output depends only on its own feature vector, so

$$\frac{\partial(y_1, \ldots, y_N)}{\partial(x_1, \ldots, x_N)} = \begin{bmatrix} J_1 & 0 & \cdots & 0 \\ 0 & J_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & J_N \end{bmatrix}.$$

To see what lies inside a LayerNorm block, set $P = I - \mathbf{1}\mathbf{1}^\top/D$, $c = Px$, and $s = \sqrt{c^\top c/D + \epsilon}$. Before learned affine parameters,

$$J_{\text{LN}} = \frac{1}{s}\left(P - \frac{cc^\top}{Ds^2}\right), \qquad J_{\text{RMS}} = \frac{1}{r}\left(I - \frac{xx^\top}{Dr^2}\right), \quad r = \sqrt{x^\top x/D + \epsilon}.$$

Differentiating c/s or x/r produces the outer-product term. Learned feature scales left-multiply each block by $\text{diag}(\gamma)$; biases contribute no input derivative. Features within a token interact, but different tokens do not. This block partition is specific to token-wise normalization: training-mode BatchNorm generally couples examples or spatial positions within its reduction set, whereas fixed inference statistics give an affine map.

Worked derivation: centering, RMS division, and their derivatives

Lecture basis: faculty handout 10, pp. 3–4. The steps below expand the lecture argument.

Step 1: fix the reduction group. Let $x \in \mathbb{R}^D$ be one token's features. For token-wise LayerNorm, neither batch nor token index is reduced. Define

$$P = I - \frac{1}{D}\mathbf{1}\mathbf{1}^\top, \quad c = Px = x - \mu\mathbf{1}, \quad \mu = \frac{1}{D}\mathbf{1}^\top x, \quad s = \sqrt{c^\top c/D + \epsilon}.$$

Then the normalized vector is $u = c/s$, followed by $y = \gamma \odot u + \beta$. The same derivation works for another reduction group after its coordinates are stacked; BatchNorm instead groups examples and spatial positions for a fixed channel during training.

Step 2: differentiate each operation. The centering matrix is symmetric, $P^2 = P$, and $Pc = c$. Therefore

$$\begin{aligned} dc &= P\,dx, \\ ds &= \frac{1}{2s}\,d(c^\top c/D) = \frac{c^\top dc}{Ds} = \frac{c^\top dx}{Ds}, \\ du &= \frac{dc}{s} - \frac{c}{s^2}ds \\ &= \left(\frac{P}{s} - \frac{cc^\top}{Ds^3}\right)dx, \\ J_{\mathrm{LN}} &= \mathrm{diag}(\gamma)\left(\frac{P}{s} - \frac{cc^\top}{Ds^3}\right). \end{aligned}$$

In particular $J_{\mathrm{LN}}\mathbf{1} = 0$: adding a common offset is removed exactly. With $\epsilon > 0$, the radial direction is not removed exactly, since $J_u c = \epsilon c/s^3$.

Step 3: repeat without centering for RMSNorm. Set $r = \sqrt{x^\top x/D + \epsilon}$ and $u = x/r$. Then

$$\begin{aligned} dr &= x^\top dx/(Dr), \\ du &= dx/r - x(x^\top dx)/(Dr^3), \\ J_{\mathrm{RMS}} &= \mathrm{diag}(\gamma)\left(\frac{I}{r} - \frac{xx^\top}{Dr^3}\right). \end{aligned}$$

There is no centering projection P. RMSNorm rescales the common-offset component together with the other components; it does not subtract the mean or preserve the numerical mean unchanged. Stacking tokens gives a block-diagonal Jacobian because different tokens use separate statistics. Training-mode BatchNorm generally couples entries inside its batch reduction group; inference with frozen statistics is an affine map instead.

Dropout: In-Network Data Augmentation

Just as normalization is an in-network analogue to input standardization, dropout can be seen as an in-network analogue to data augmentation.

- **Idea**: During training, randomly set a fraction of neuron activations to zero at each update step.
- **Purpose**: This prevents neurons from co-adapting too much and forces the network to learn more robust features. It encourages the internal representations to be **redundant**, meaning the necessary information is encoded in multiple ways, so that dropping some activations does not destroy the information.
- **Mechanics**: Typically used for MLP-style layers (including 1×1 convolutions).

  - For each element in the activation vector $\vec{h}_L$, it is kept with a probability p (the "keep probability") and set to zero with probability $1-p$.
  - This is equivalent to an element-wise multiplication of the activation vector with a mask vector of i.i.d. Bernoulli random variables.

- **Dropout at Test Time**:
  - **Problem**: During training, the expected output of a layer is scaled down by the keep probability p. If we do nothing at test time (i.e., keep all activations), the scale of the activations will be significantly larger than what the network was trained on. This mismatch in scale disrupts the network's performance.
  - **Solution (Inverted Dropout)**: To compensate for the scale change, the standard approach is to perform the scaling during training. This is known as inverted dropout.
$$\tilde{h}_L[j] = \begin{cases} 0 & \text{with probability } 1-p \\ \frac{h_L[j]}{p} & \text{with probability } p \end{cases}$$
    By scaling up the activations that are kept during training, their expected value remains unchanged: $E[\tilde{h}_L[j]] = p \cdot \frac{h_L[j]}{p} + (1-p) \cdot 0 = h_L[j]$.
  - With inverted dropout, no modification is needed at test time; the dropout layer simply acts as an identity function. This is the standard implementation in frameworks like PyTorch.

10.3 Residual and Skip Connections

Deeper explanation: When an identity skip still has a singular Jacobian (p. 388)

As networks get deeper, a "degradation" problem can occur: adding more layers can lead to higher training error, not just higher test error. This suggests an optimization problem, not an overfitting one. A 56-layer network might perform worse on both training and test sets than a 20-layer network.

The Problem with Deep Networks

Two main hypotheses explain this degradation:

1. **Vanishing/Exploding Gradients**: During backpropagation, gradients are repeatedly multiplied by weight matrices. In deep networks, this can cause the gradient signal to shrink exponentially to zero (vanish) or grow exponentially to infinity (explode), making it difficult for early layers to learn.
2. **Corrupted Gradient Signals ("Confused Gradients")**: In a randomly initialized deep network, layers are not well-configured. The gradient signal passed from a later layer to an earlier one can be noisy and uninformative. This is like a game of telephone

where the useful error signal from the loss function gets corrupted as it propagates backward through many layers.

The Solution: Skip Connections (ResNets)

The core idea is to create "gradient superhighways" that allow the gradient to flow more directly to earlier layers.

- A naive approach might be to connect the output of every earlier layer to the input of every later layer (as in DenseNets). However, this can lead to a blow-up in the number of channels and activations.
- A more minimalist and highly effective approach is the **skip connection** or **residual connection**.
- A layer or a block of layers is reformulated to learn a *residual function* $F(x)$ with respect to its input x. The output of the block becomes:

$$H(x) = F(x) + x$$

  The "$+x$" part is the skip connection, which is an identity mapping that bypasses the layers.

- **Benefit for Gradient Flow**: This architecture creates a direct path for the gradient. Consider the derivative of the loss $\mathcal{L}$ with respect to the input x:

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial H(x)} \frac{\partial H(x)}{\partial x} = \frac{\partial \mathcal{L}}{\partial H(x)} \left(\frac{\partial F(x)}{\partial x} + 1 \right)$$

  The "+1" term ensures that the gradient can flow backward through the identity path unimpeded, even if the gradients through the weight layers $\frac{\partial F(x)}{\partial x}$ become very small. This effectively combats the vanishing gradient problem and allows for the successful training of much deeper networks.

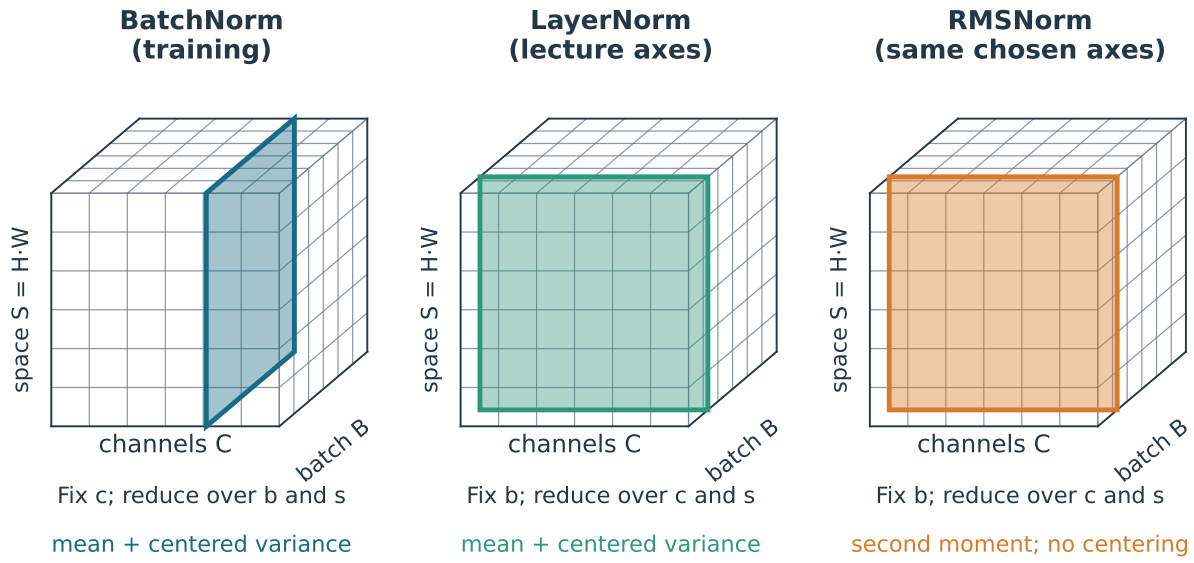


Figure 10.3: Normalization cubes use the faculty's channels–space–batch convention (lecture 10, PDF p. 3; lecture 11, PDF p. 2). Each shaded set supplies one normalization statistic. RMSNorm is shown as an explicit extension using the same axes as the lecture's LayerNorm; its statistic differs even when the shaded set is identical.

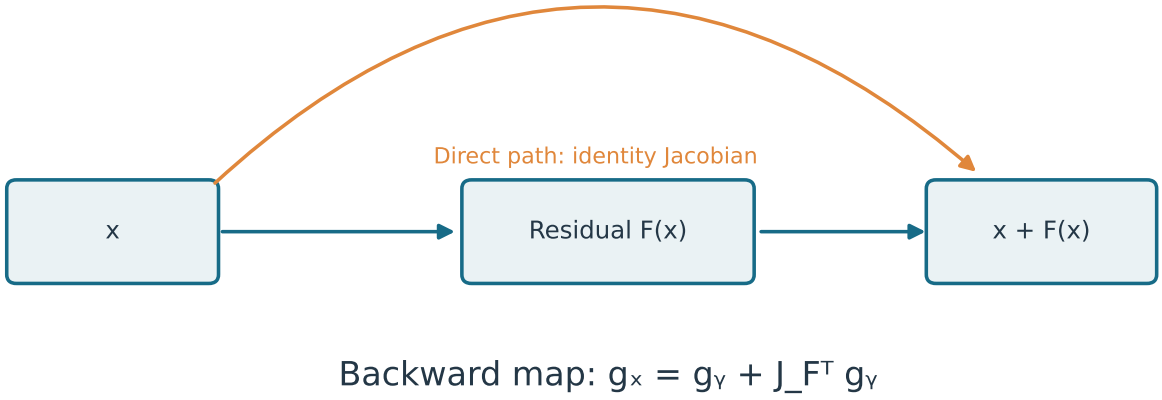


Figure 10.4: A residual block has both an identity path and a learned correction. The Jacobian contains a direct identity contribution.

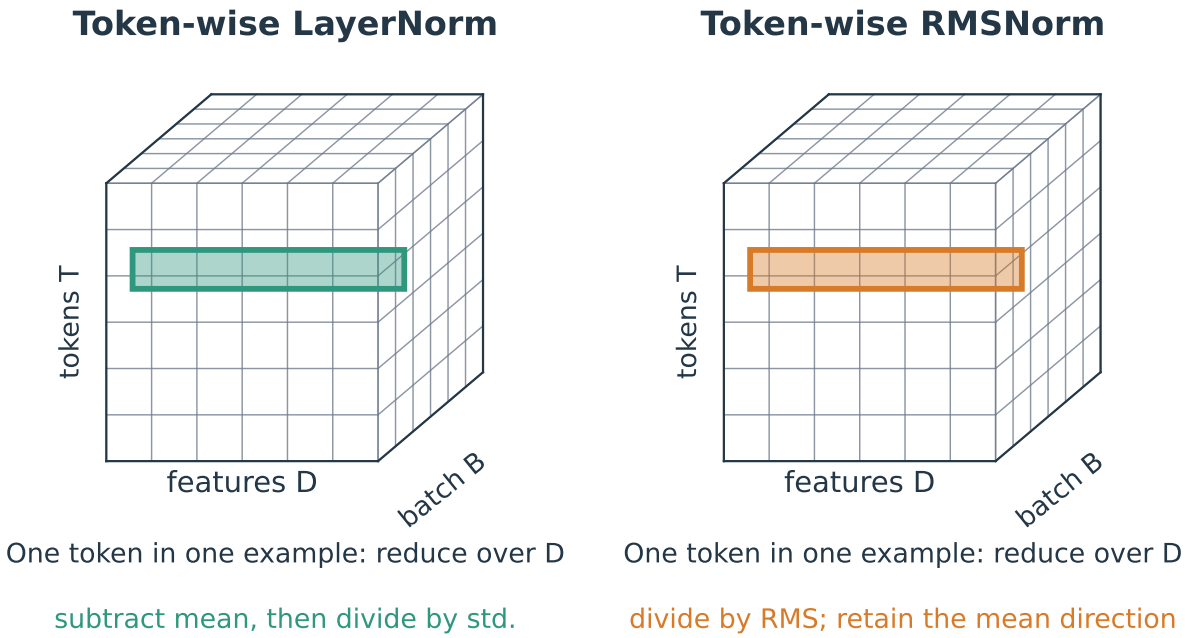


Figure 10.5: For a tensor (B, T, D), token-wise LN and RMSNorm use the same feature fiber, independently for every (b, t). The shaded band represents one fiber; its thickness is only for visibility. Neither operation averages over other tokens or examples.

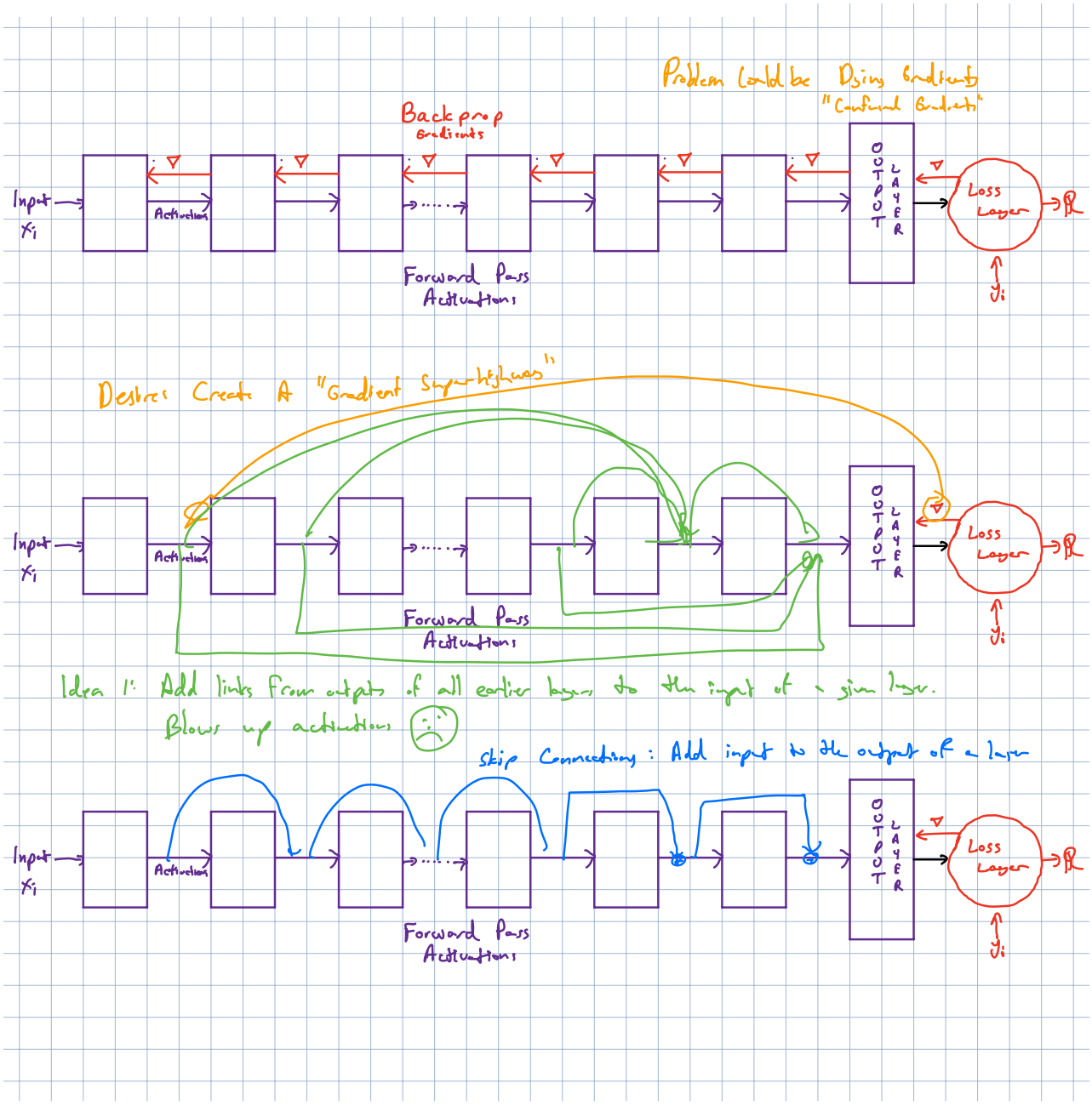


Figure 10.6: Local residual branches versus indiscriminate shortcuts. Follow which activations are added and how often a signal is reused. Residual addition gives a short identity route, but adding many unscaled paths can inflate activation magnitude. Original annotated source excerpt; Anant Sahai, faculty handout 10, p. 6.

Worked derivation: dropout moments and residual gradient paths

Lecture basis: faculty handout 10, pp. 5–6. The steps below expand the lecture argument.

Step 1: preserve the mean of an activation. Let $m \sim \text{Bernoulli}(p)$, where p is the keep probability, and condition on a fixed scalar activation h. Inverted dropout uses $\tilde{h} = mh/p$:

$$\begin{aligned}\mathbb{E}[\tilde{h} \mid h] &= (h/p)\mathbb{E}[m] = h,\\ \mathbb{E}[\tilde{h}^2 \mid h] &= (h^2/p^2)\mathbb{E}[m^2] = h^2/p,\\ \text{Var}(\tilde{h} \mid h) &= h^2(1/p - 1).\end{aligned}$$

Since $m^2 = m$, the second moment increases even though the mean is unchanged. For a whole residual branch $F(h)$ gated by one shared mask, the covariance is $(1/p-1)F(h)F(h)^\top$, so its coordinates are correlated. This is the stochastic-depth calculation used again in Lecture 12.

Step 2: differentiate one residual block. For $h_{\ell+1} = h_\ell + F_\ell(h_\ell)$,

$$dh_{\ell+1} = (I + J_\ell)dh_\ell, \qquad \nabla_{h_\ell} L = (I + J_\ell)^\top \nabla_{h_{\ell+1}} L.$$

For two successive blocks, the forward Jacobian expands as

$$(I + J_2)(I + J_1) = I + J_1 + J_2 + J_2 J_1.$$

The four terms correspond to choosing the skip or residual path at each block. For L blocks, the ordered product contains paths of many lengths. Matrix order must be retained because Jacobians need not commute. The identity path makes short routes available, but other terms can cancel it; residual connections do not prove that every gradient stays large.

Lecture 11

ResNets, ConvNeXt, U-Nets

Course sources: lecture_11; 2025-10-07 transcript.

11.1 Residual and Skip Connections (Recap)

Deeper explanation: When an identity skip still has a singular Jacobian (p. 388)

The Degradation Problem

As neural networks became more successful, researchers observed that simply making them deeper (adding more layers) did not always lead to better performance. In fact, a counter-intuitive phenomenon called the **degradation problem** was discovered: a deeper network (e.g., 56 layers) would perform worse than a shallower one (e.g., 20 layers) on *both* the training and testing sets.

This is puzzling from a machine learning perspective. A deeper model has more parameters and should, in theory, be able to at least learn the identity function for the extra layers, thereby performing no worse than its shallower counterpart. The poor performance on the training set suggests an optimization issue, not overfitting. The hypothesis was that as gradients propagated backward through many layers, they would either diminish (vanish) or become distorted and less informative, making it difficult for the earlier layers to learn effectively.

The Gradient Superhighway

The solution proposed was to create a more direct path for gradients to flow from later layers back to earlier ones. This concept is often called a **gradient superhighway**. Since gradients in backpropagation follow the forward paths of the network in reverse, creating a gradient highway requires adding new paths in the forward pass.

The idea is to add **skip connections** (or residual connections), which take the input to a block of layers, 'x', and add it to the output of that block, 'F(x)'. The new output is then $H(x) = F(x) + x$.

- A **block** is a more abstract concept than a single layer. It can contain multiple layers, such as convolutions, non-linearities, and normalizations.

- The addition operation acts as a gradient distributor. During backpropagation, the gradient flowing into the sum node is passed identically to both the block's output and the input skip path.
- This creates direct gradient paths from the final loss all the way back to the initial layers, bypassing many intermediate blocks. This ensures that even the earliest layers receive clean, informative gradients.

A common initial thought was to connect every layer to every subsequent layer. However, this causes the activations to grow excessively large. Skip connections provide a minimal and effective way to achieve the goal of unimpeded gradient flow. The activations grow arithmetically with depth (due to the summation), which is manageable with standard floating-point precision like FP32, but can become a concern with lower-precision formats (FP16, FP8, FP4).

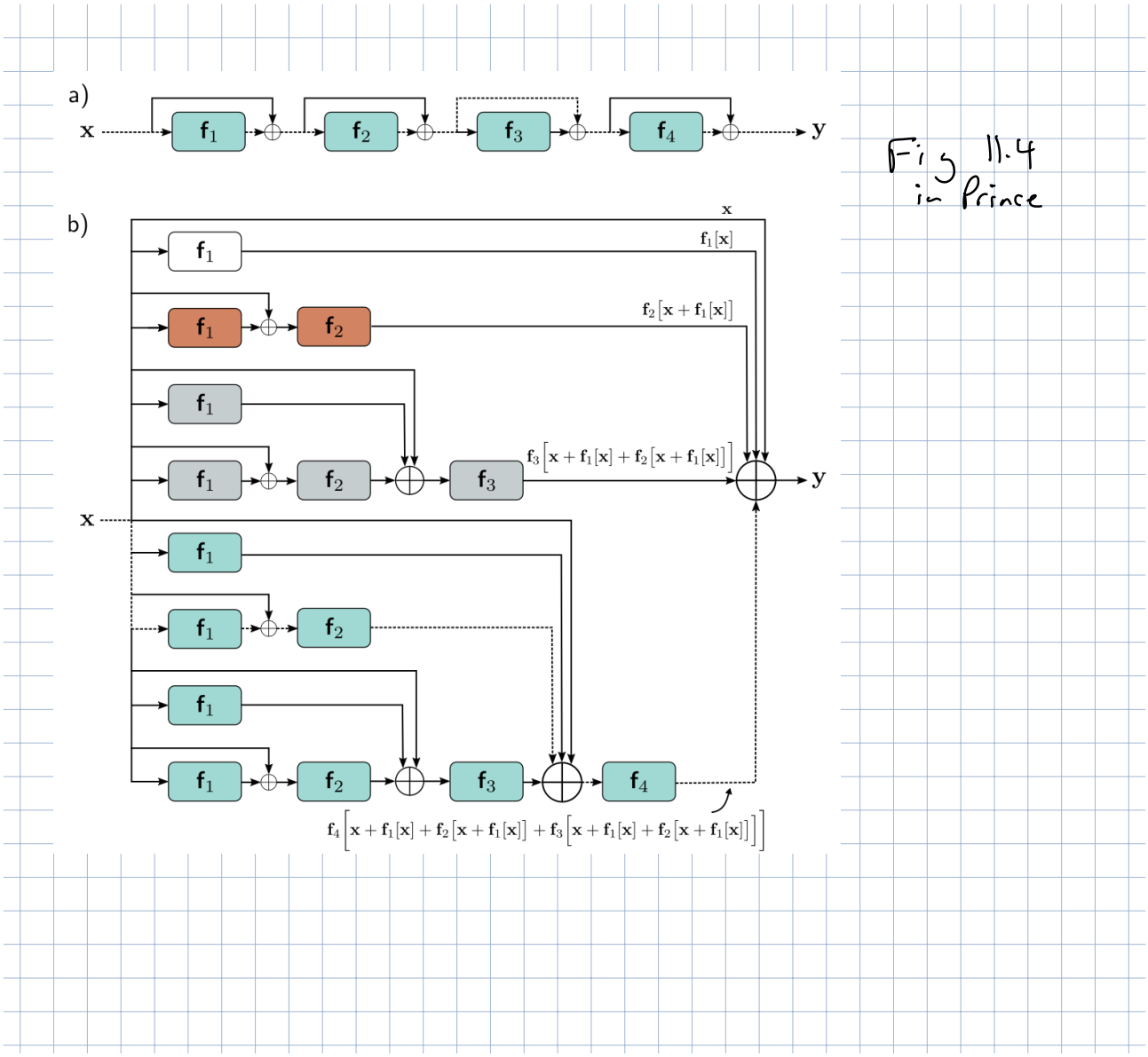


Fig 11.4 in Prince

Figure 11.1: Unrolling a residual network into routes. The expanded published diagram shows how a signal can traverse different subsets of residual branches. This explains shorter routes through a deep network without implying that the branches are independent models. Original annotated source excerpt; Anant Sahai, faculty handout 11, p. 5.

11.2 Inside a Residual Block

Deeper explanation: When an identity skip still has a singular Jacobian (p. 388)

Core Structure and Intuition

The minimal and effective structure for a block, 'F(x)', that is skipped over consists of at least two weight layers. A typical core structure is:

$$\text{Input} \to [\text{Weight Layer 1}] \to [\text{Bias}] \to [\text{ReLU}] \to [\text{Weight Layer 2}] \to [\text{Bias}] \to \text{Output}$$

The reason two layers are needed is rooted in the expressive power of the block. A single layer followed by a ReLU would restrict the block's output 'F(x)' to be non-negative. This is a significant limitation. The second weight layer allows the block to produce both positive and negative values, enabling it to learn more complex functions.

An ODE Perspective The structure of a ResNet changes the fundamental composition of functions in a deep network.

- **Pre-ResNet (Nested Functions):** $y = f_n(f_{n-1}(\dots f_1(x)\dots))$
- **ResNet (Additive Updates):** $x_{i+1} = x_i + g_i(x_i)$

This additive form is reminiscent of a forward Euler discretization of an Ordinary Differential Equation (ODE), such as $\frac{dx}{dt} = g(x(t), t)$. This analogy inspired the field of **Neural ODEs**, where the network is viewed as learning a continuous transformation (a vector field) that flows an input to its desired output. From this perspective, it's clear why $g_i(x_i)$ must be able to produce negative values—to allow the "flow" to move in any direction, not just forward.

This perspective also inspired the idea of **weight sharing across depth**, where the same block function g is used repeatedly, analogous to a time-invariant ODE. While this is effective for many physics-inspired problems, standard ResNets with unique weights at each block generally perform better on computer vision tasks.

Placement of Normalization Layers

A key design choice is where to place normalization layers (like Batch Normalization or Layer Normalization) in relation to the residual block.

- **Old Perspective (Post-Normalization):** Early ResNet papers and related architectures (e.g., the original Transformer) often placed normalization layers at the end of a block, after the addition with the skip connection. The logic was to re-normalize the activations after they might have grown or shifted due to the addition.
- **Modern Default (Pre-Normalization):** The current standard practice is to place the normalization layer *inside* the block, at the very *beginning*.

A modern pre-norm residual block looks like this:

$$\text{Input } x_i \to [\text{Normalization}] \to [\text{Weight Layer 1}] \to [\text{ReLU}] \to [\text{Weight Layer 2}] \to \text{Output } F(x_i)$$

The final output of the entire stage is then $x_{i+1} = x_i + F(x_i)$.

The key advantage of the pre-norm convention is that there are **no operations on the skip path**. The "gradient superhighway" is a clean, uninterrupted path of additions from the end of the network to the beginning. This leads to more stable training, especially for very deep networks.

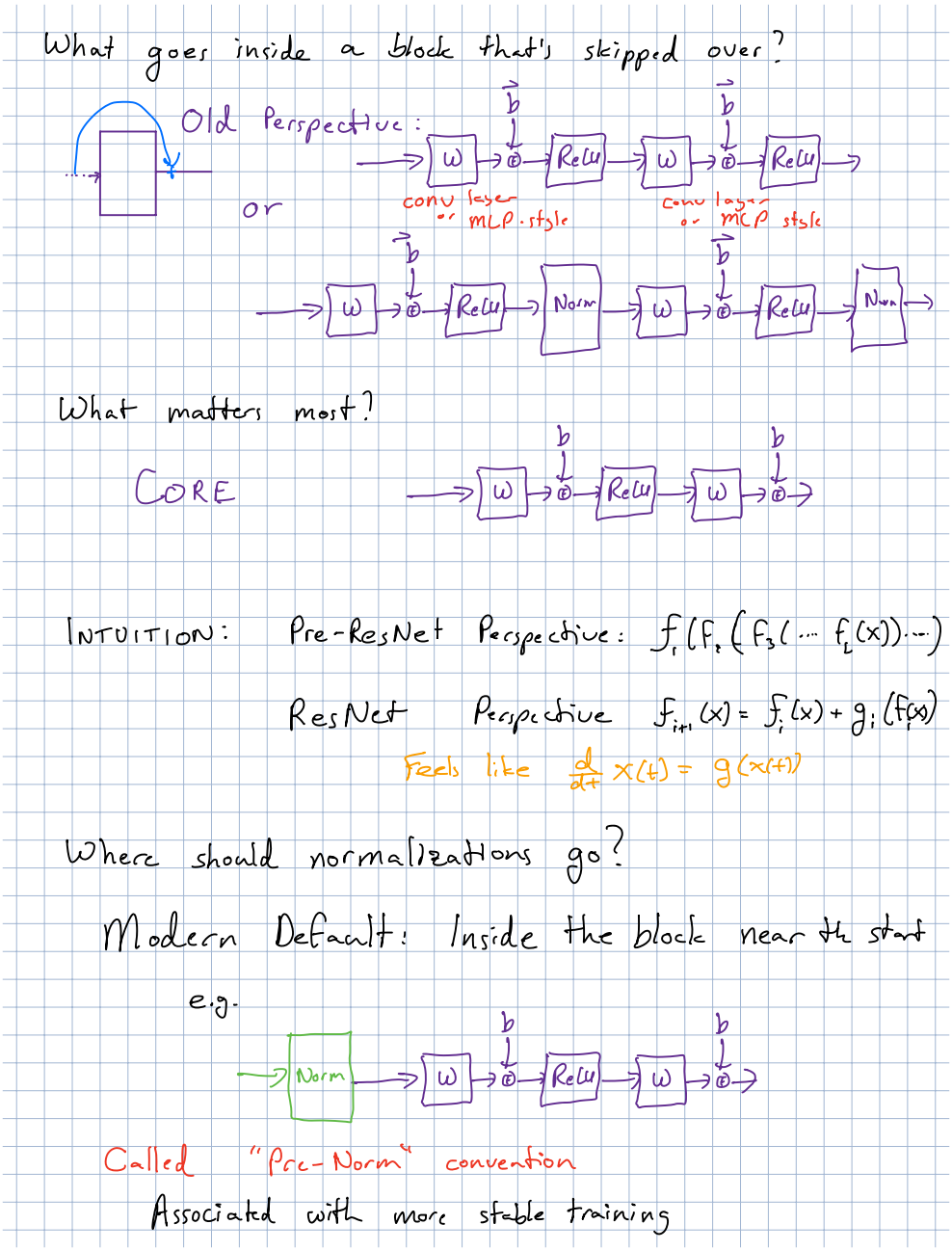


Figure 11.2: Normalization placement inside a residual block. The placement of normalization and nonlinearities determines what lies on the identity route. Compare the drawn orders before treating two residual blocks as equivalent. Original annotated source excerpt; Anant Sahai, faculty handout 11, p. 4.

Handling Dimension Mismatches

Standard skip connections $H(x) = F(x) + x$ require that the input x and the output of the block $F(x)$ have the same shape. This is often not the case, especially at stages where downsampling occurs (e.g., using a strided convolution or a pooling layer inside the block), which reduces spatial dimensions and typically increases the number of channels.

If an input is $100 \times 100 \times 32$ and the block output is $50 \times 50 \times 64$, a simple addition is impossible. To maintain the gradient highway, we must modify the skip connection path to match the dimensions.

- **Goal:** Create a path for gradients to flow back, even if shapes differ.
- **Solution:** Introduce a projection layer, P, on the skip path. The new formulation becomes $H(x) = F(x) + P(x)$.
- **Choices for P:**
  1. **Learnable Projection:** The most flexible approach is to make P a learnable linear layer (e.g., a 1×1 convolution with the appropriate stride and output channels). The network learns the best way to project the identity.
  2. **Fixed Projection:** A simpler method is to use a fixed function. For example, downsample spatially via averaging or subsampling, and handle channel changes by padding with zeros or truncating.

The crucial point is that *some* path must be created. Simply omitting the skip connection at these transition points would break the gradient superhighway.

11.3 A Modern Perspective: ConvNeXt

Deeper explanation: Residual bottlenecks and ConvNeXt-style blocks (p. 391)

In 2022, a paper titled "A ConvNet for the 2020s" revisited the ResNet architecture, modernizing it with design choices inspired by Transformers, resulting in an architecture called **ConvNeXt**. This work demonstrated that with the right details, pure ConvNets could still outperform Transformer-based models on vision tasks.

The key changes from a standard ResNet block to a ConvNeXt block are:

1. **Inverted Bottleneck:**
   - **ResNet:** Uses a "bottleneck" design: 1×1 conv (reduce channels) $\rightarrow 3 \times 3$ conv $\rightarrow 1 \times 1$ conv (restore channels).
   - **ConvNeXt:** Uses an "inverted bottleneck": 1×1 conv (expand channels) $\rightarrow k \times k$ conv $\rightarrow 1 \times 1$ conv (reduce channels). This allows the network to operate on a higher-dimensional feature space internally, increasing expressive power.
2. **Depthwise Separable Convolutions:**
   - The main spatial convolution (e.g., 7×7) is replaced with a **depthwise convolution**.
   - In a depthwise convolution, each input channel is processed independently by its own 2D filter. There is no mixing of information across channels.

- The channel mixing is handled entirely by the preceding and succeeding 1×1 convolutions (also known as pointwise convolutions).
- This is far more parameter-efficient than a standard convolution.

3. **Fewer Normalization and Activation Layers:**
   - Instead of placing normalization (BN) and activation (ReLU) layers after each convolution, ConvNeXt uses a single **Layer Normalization (LN)** at the beginning of the block.
   - It uses a single activation function, **GELU** (Gaussian Error Linear Unit), instead of ReLU. GELU is a smoother alternative to ReLU. Its formula is $GELU(x) = x \cdot \Phi(x)$, where $\Phi(x)$ is the cumulative distribution function (CDF) of the standard normal distribution.

These changes collectively show that architectural details matter significantly, and many "standard" practices can be revised to achieve better performance.

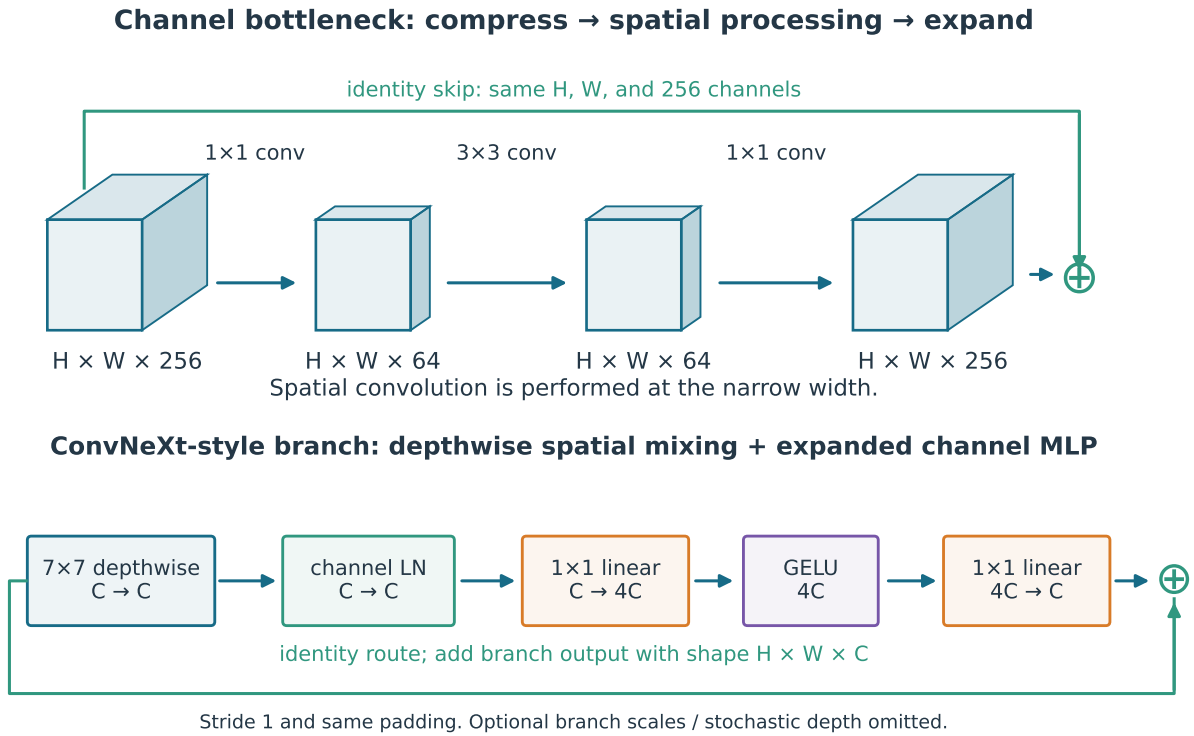


Figure 11.3: A conventional bottleneck and a simplified ConvNeXt-style block organize channel capacity differently. All additions require matching activation shapes. The upper diagram isolates convolution shapes; normalization and activation placement must be specified separately. The lower example explicitly shows channel-only LN, expansion, GELU, projection, and the identity route.

11.4 Upsampling and Transpose Convolutions

Deeper explanation: Uneven overlap in transposed convolution (p. 393)

Many vision tasks, such as semantic segmentation, require the output to be an image of the same (or higher) resolution as the input. This requires **upsampling** operations to reverse the effect of downsampling (pooling, strided convolutions).

Upsampling Methods

- **Max Unpooling:** The reverse of max pooling. Requires storing the indices of the maxima from the forward pass to place the values back in their original locations. Other locations are filled with zeros.
- **Duplication / Nearest Neighbor:** A simple method where each low-resolution pixel value is copied to fill a block of pixels in the high-resolution output.
- **Learnable Interpolation:** The most powerful approach is to learn the upsampling function. This is typically achieved by:
    1. Perform a simple upsampling by inserting zeros between the pixels of the low-resolution input (zero-fill).
    2. Apply a standard learnable convolution to this sparse input. The convolution learns to ”fill in the blanks” by interpolating from the non-zero values.

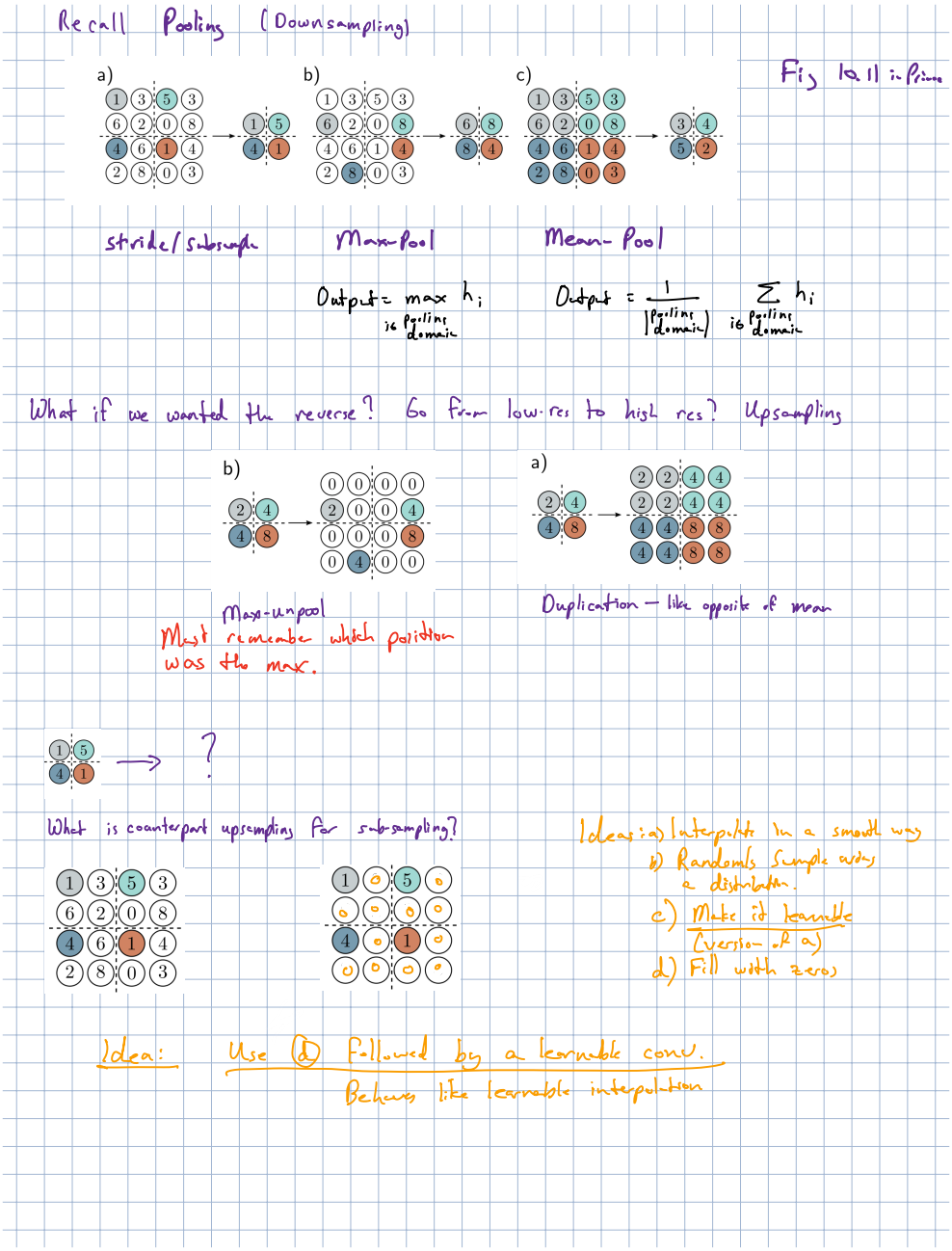


Figure 11.4: Pooling switches, zero insertion, and unpooling. The pictures distinguish where values are placed when resolution increases. A pooling switch can remember an argmax location, but unpooling alone does not recover all information discarded by pooling. Published panels are reproduced from the handout. Original annotated source excerpt; Anant Sahai, faculty handout 11, p. 9.

Transpose Convolution

A **transpose convolution** is a computational optimization for the "zero-fill + convolution" upsampling method. It is *not* a conceptually new operation.

- **Analogy:**
  - A **strided convolution** fuses a standard convolution and a subsequent subsampling step into one efficient operation.
  - A **transpose convolution** fuses a zero-fill upsampling step and a subsequent standard convolution into one efficient operation.
- **Mechanism:** It avoids the explicit creation of a large, zero-filled tensor and skips the wasteful multiplications by zero.
- **Nomenclature:** It is called "transpose" because if the forward pass of a standard convolution is represented as a matrix multiplication $y = Cx$, the corresponding transpose convolution's forward pass can be represented as $z = C^T y$. It is sometimes misleadingly called a "deconvolution," but it does not truly invert the convolution.

Conceptually, there are only standard convolutions. Strided and transpose convolutions are just efficient implementations for downsampling and upsampling, respectively.

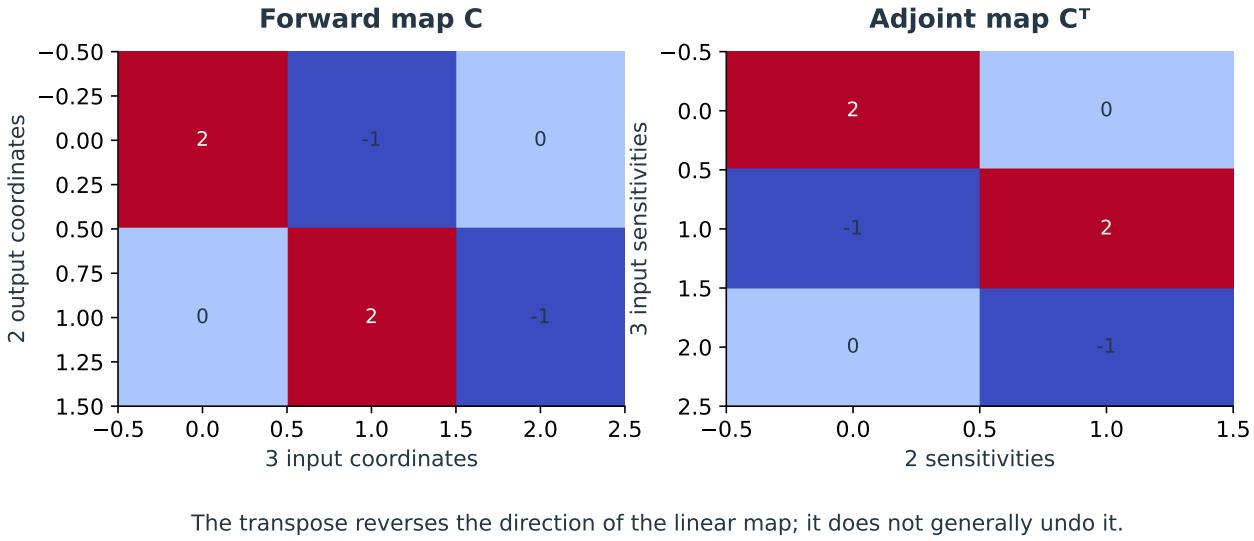


Figure 11.5: A transposed convolution scatters and sums contributions. It reverses the connectivity of the linear operator, not the information loss.

Worked derivation: why transposed convolution is an adjoint

Lecture basis: faculty handout 11, pp. 9–11. The steps below expand the lecture argument.

Step 1: flatten a concrete convolution into a matrix. For a valid width-2, stride-1 cross-correlation on three inputs,

$$y = Kx, \qquad K = \begin{bmatrix} a & b & 0 \\ 0 & a & b \end{bmatrix}.$$

For any output-space vector g,

$$g^\top y = g^\top K x = (K^\top g)^\top x,$$
$$K^\top g = \begin{bmatrix} a & 0 \\ b & a \\ 0 & b \end{bmatrix} \begin{bmatrix} g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} ag_1 \\ bg_1 + ag_2 \\ bg_2 \end{bmatrix}.$$

This is both the adjoint map and the gradient with respect to x. Overlapping kernel contributions are added. It is not an inverse: K maps three numbers to two, so information can be lost.

Step 2: account for stride and shape ambiguity. Forward stride first computes local responses and selects positions. If $K = SC$ with selection matrix S, then $K^\top = C^\top S^\top$. The map $S^\top$ inserts zeros into unselected positions; $C^\top$ spreads and sums them through the adjoint kernel. For effective kernel size k_{eff}, a compatible reconstructed length is

$$N = (N_{\text{out}} - 1)s - 2p + k_{\text{eff}} + o, \qquad 0 \le o < s.$$

The integer o selects among input lengths that collapsed to the same forward length under the floor. The original forward input shape fixes the correct adjoint domain; an independently constructed transposed-convolution layer must be told that shape choice. Enlarging a tensor is not equivalent to recovering the original image.

11.5 Architectures for Image-Level Tasks: U-Nets

Deeper explanation: U-Net skip concatenation: memory and gradient accounting (p. 394)

For tasks like semantic segmentation (classifying every pixel), a common architecture is an **encoder-decoder** network.

- **Encoder:** A contracting path, typically a standard ConvNet (like a ResNet), that progressively downsamples the input image to extract high-level semantic features (the "what").
- **Decoder:** An expansive path that progressively upsamples the feature maps using transpose convolutions to reconstruct a full-resolution output map (the "where").

The Bottleneck Problem

A simple encoder-decoder architecture forces all information to pass through a low-resolution bottleneck between the two paths. This makes it very difficult for the decoder to reconstruct fine-grained spatial details that were lost during downsampling.

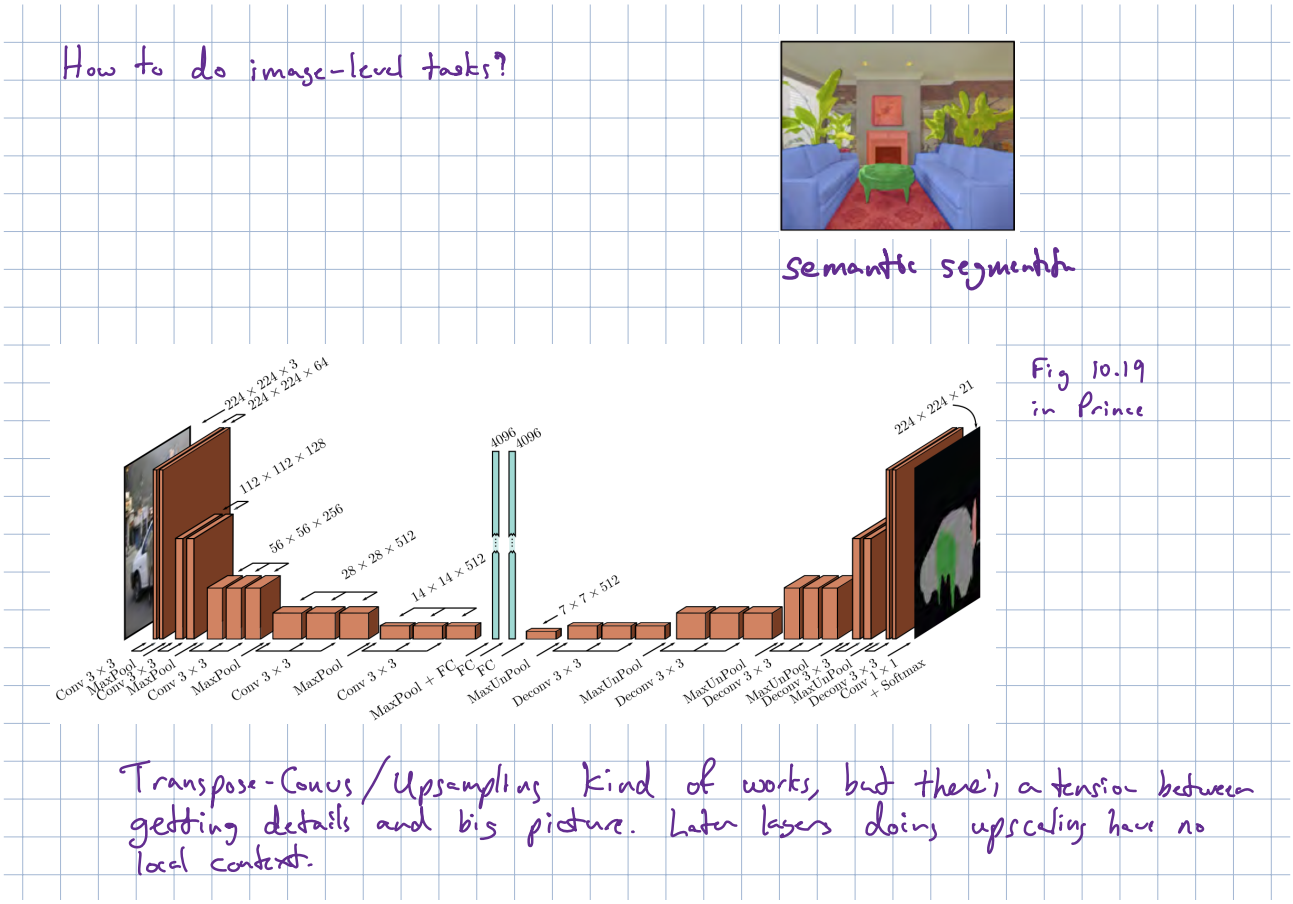


Figure 11.6: A segmentation encoder-decoder before skip connections. Compare the compressed bottleneck with the dense output grid. This published architecture explains why preserving fine spatial detail is difficult if all information must pass through the bottleneck. Original annotated source excerpt; Anant Sahai, faculty handout 11, p. 12.

The U-Net Solution

The **U-Net** architecture solves this problem by introducing long-range skip connections between the encoder and decoder.

- **Mechanism:** The feature maps from an encoder stage are concatenated channel-wise with the feature maps at the corresponding stage in the decoder (after upsampling).
- **Benefit:** This provides the decoder with direct access to high-resolution, low-level feature information from the encoder. The decoder can then combine the high-level

semantic information (from the deeper layers) with this fine-grained spatial information to produce a much more precise output.

- **Structure:** The architecture diagram resembles the letter 'U', giving it its name. These long-range skip connections are used *in addition to* the short-range residual connections within each block of the encoder and decoder.

U-Net-like structures are fundamental to almost any task that requires generating high-resolution image outputs, including generative models.

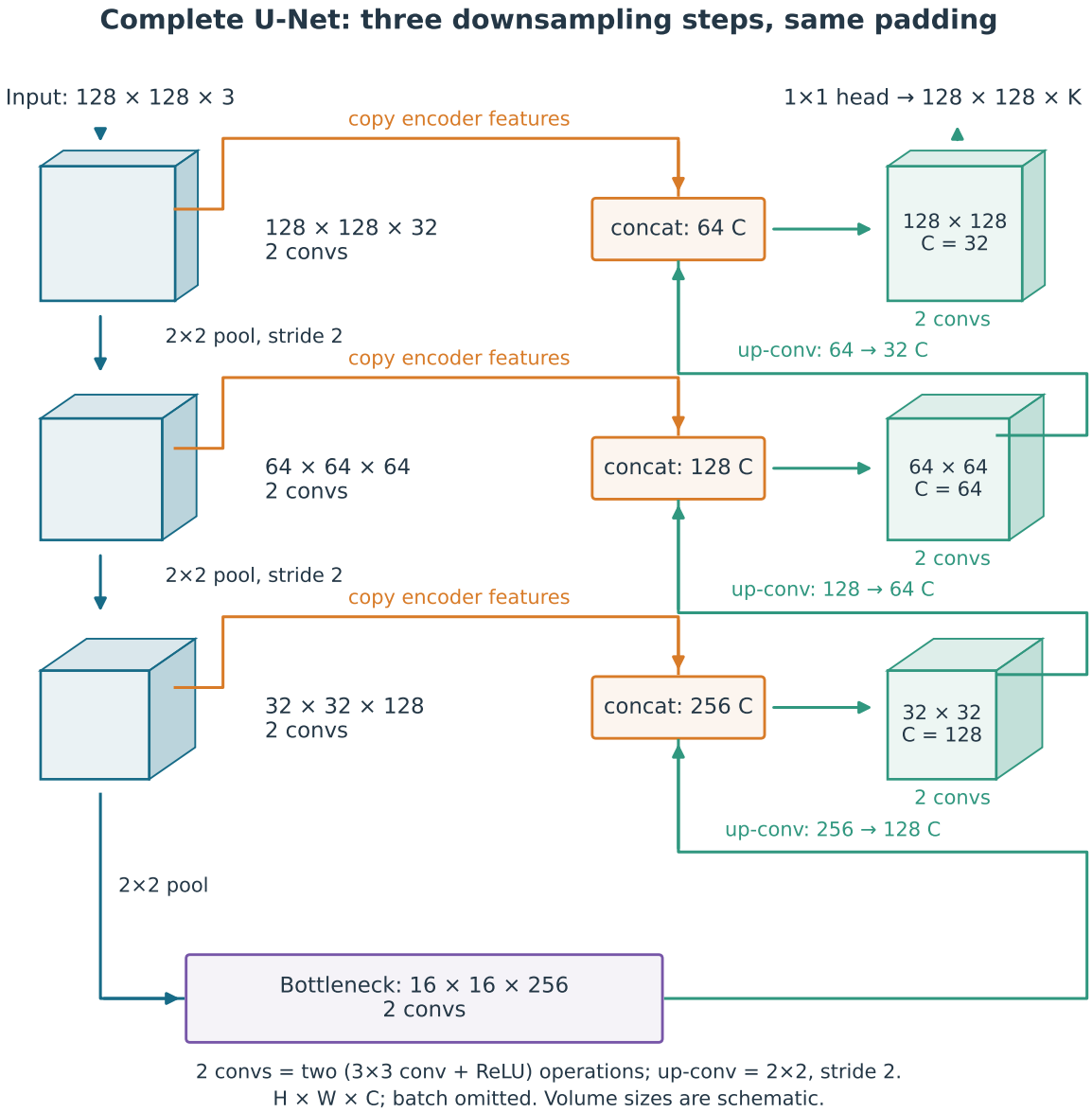


Figure 11.7: A complete shape-consistent U-Net example. Each paired-convolution block uses stride one and same padding; pooling halves spatial size, and a 2×2 stride-two transposed convolution doubles it while reducing channels. Orange skips concatenate channels before the decoder's paired convolutions. The 1×1 head yields K segmentation logits per pixel. This is an illustrative modern same-padding configuration, not an exact reproduction of the original valid-convolution U-Net.

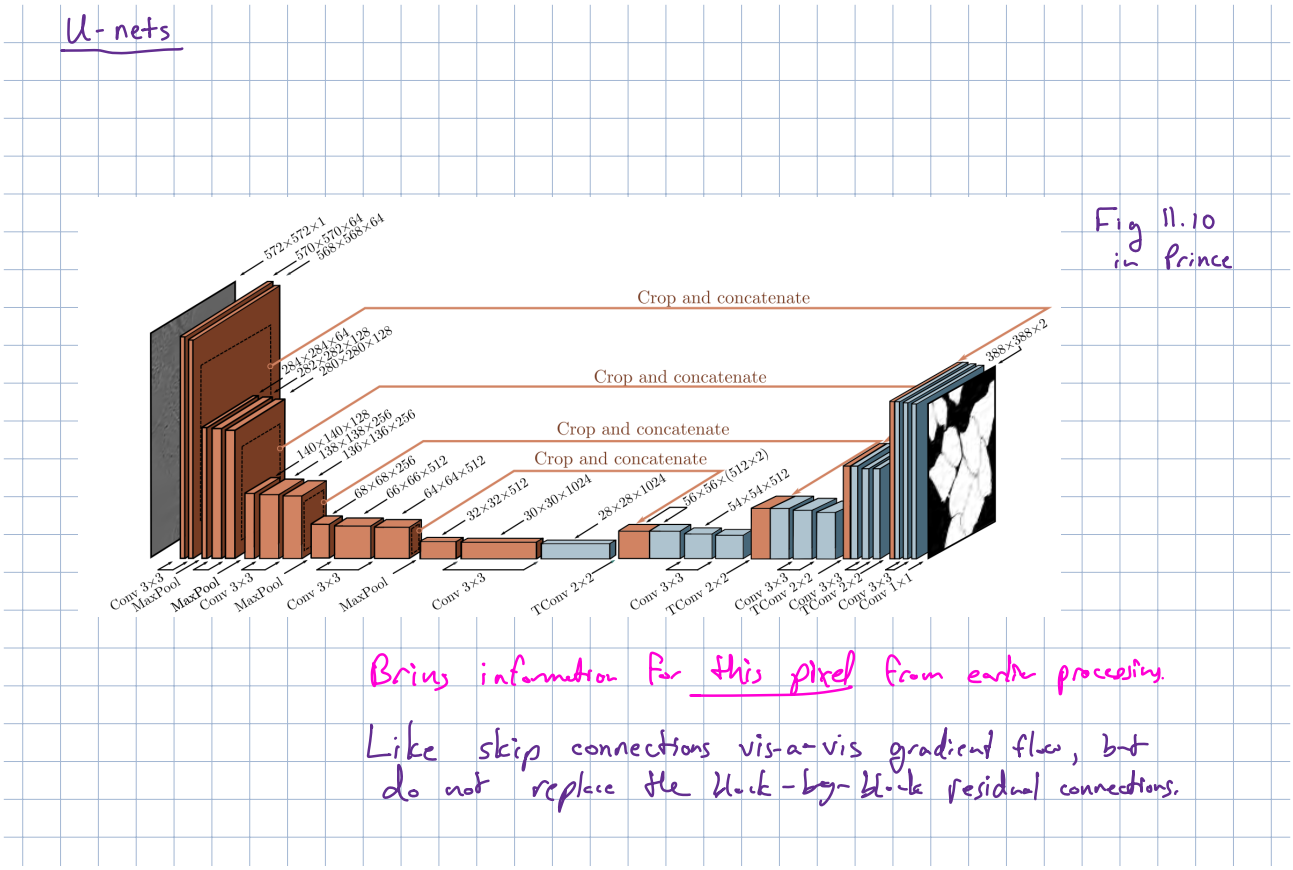


Figure 11.8: The annotated U-Net architecture from the lecture. The long skip connections supply high-resolution encoder features to the decoder. Compare them with the preceding encoder-decoder and the reader's clean tensor-shape diagram. The original published architecture and faculty annotations are preserved. Original annotated source excerpt; Anant Sahai, faculty handout 11, p. 13.

Lecture 12

Graph Neural Networks

Course sources: lecture_12; 2025-10-09 transcript.

12.1 Introduction and Resources

This lecture transitions from Convolutional Neural Networks (CNNs) to Graph Neural Networks (GNNs). GNNs generalize the concept of local structure, moving from the rigid grid of an image to the more flexible structure of a graph.

Key Resources

- **Textbook:** Chapter 13 in "Computer Vision: Models, Learning, and Inference" by Simon Prince.
- **Blog Post:** "A Gentle Introduction to Graph Neural Networks" (Distill, 2021).
- **Software Packages:**
  - PyTorch Geometric (`PyG`)
  - Deep Graph Library (`DGL`)
  - Spektral (for TensorFlow)
- **Further Study:** Stanford's CS224W course on Machine Learning with Graphs.

12.2 Recap and A Loose End from CNNs

Deeper explanation: Coordinate dropout versus branch dropout: the noise covariance (p. 388)

From a pedagogical perspective, understanding GNNs helps clarify which concepts from CNNs are specific to grid-like data and which are more general principles of network architecture.

Stochastic Depth Regularization

Before diving into GNNs, we address one final concept from the CNN module: stochastic depth regularization. This is a form of regularization analogous to Dropout, but applied at the block level in residual networks (ResNets).

- **Idea:** During training, randomly drop entire residual blocks with some probability p. The block is replaced by its identity skip connection.
- **Purpose:**
  - **Regularization:** Helps prevent overfitting.
  - **Functional Redundancy:** Similar to Dropout, it prevents any single block from becoming overly specialized. It encourages information to be distributed across multiple paths in the network.
- **Computational Advantage:** Unlike Dropout in MLPs, dropping an entire block yields a direct computational saving during training, as the forward and backward passes for that block are skipped.
- **Usage:** Typical dropout probabilities are around 10%. Schedules can vary, sometimes dropping earlier layers more frequently than later ones. It is a technique to consider when dealing with overfitting in deep ResNet-style architectures.

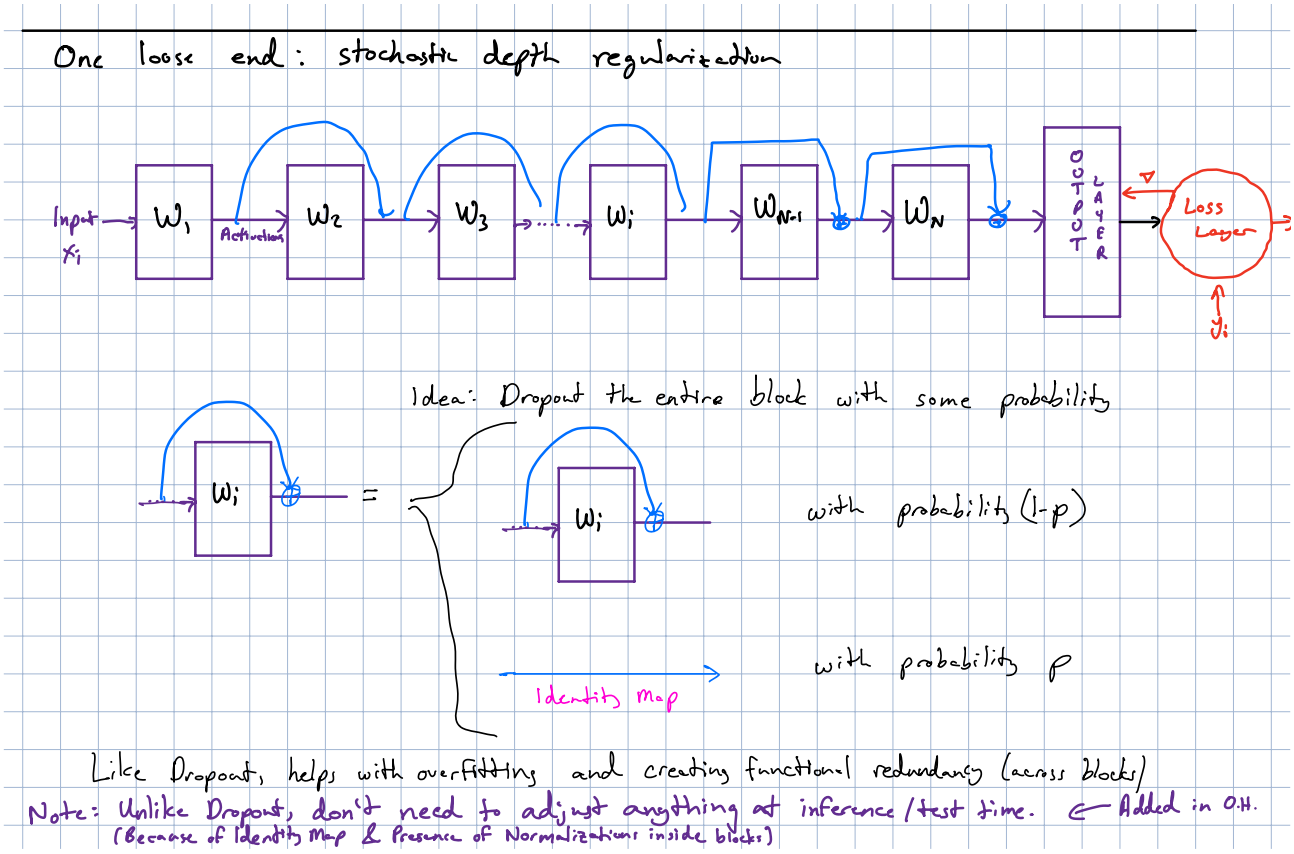


Figure 12.1: Stochastic depth drops a whole residual branch. The random decision acts on an entire branch rather than individual feature entries. The identity path remains, while the surviving branch requires the training/inference scaling convention described in the text. Original annotated source excerpt; Anant Sahai, faculty handout 12, p. 2.

12.3 Generalizing CNNs to Graphs

Deeper explanation: From a pixel grid to an arbitrary graph (p. 396)

The progression of architectures in this course has been from MLPs (fully connected, no assumed input structure) to CNNs (exploiting local grid structure). GNNs are the next step, designed for data where relationships are best represented by a graph.

Why Graphs?

Graphs are a general way to represent local structure and relationships. They appear in numerous applications:

- **Social Networks:** Users as nodes, connections (friendship, follows) as edges.
- **Citation Networks:** Papers as nodes, citations as directed edges.
- **Chemistry and Materials Science:** Atoms as nodes, chemical bonds as edges. Molecules are naturally graphs.
- **Physics and Engineering:** Meshes for 3D modeling, circuit diagrams, and physical systems.
- **Computer Vision:** Scene graphs representing objects and their spatial or semantic relationships.

Analogies Between CNN and GNN Tasks

We can draw direct parallels between common tasks for images and graphs.

- **Image Classification → Graph Classification:** The entire input (image or graph) is processed to produce a single label. For example, classifying a molecule graph based on a chemical property (e.g., "is it smelly?").
- **Semantic Segmentation → Node Classification:** Each component (pixel or node) of the input is assigned a label.
- **Other Tasks:** GNNs can also perform edge-level tasks, such as edge classification or link prediction (predicting if an edge should exist). This lecture will focus on node-level processing.

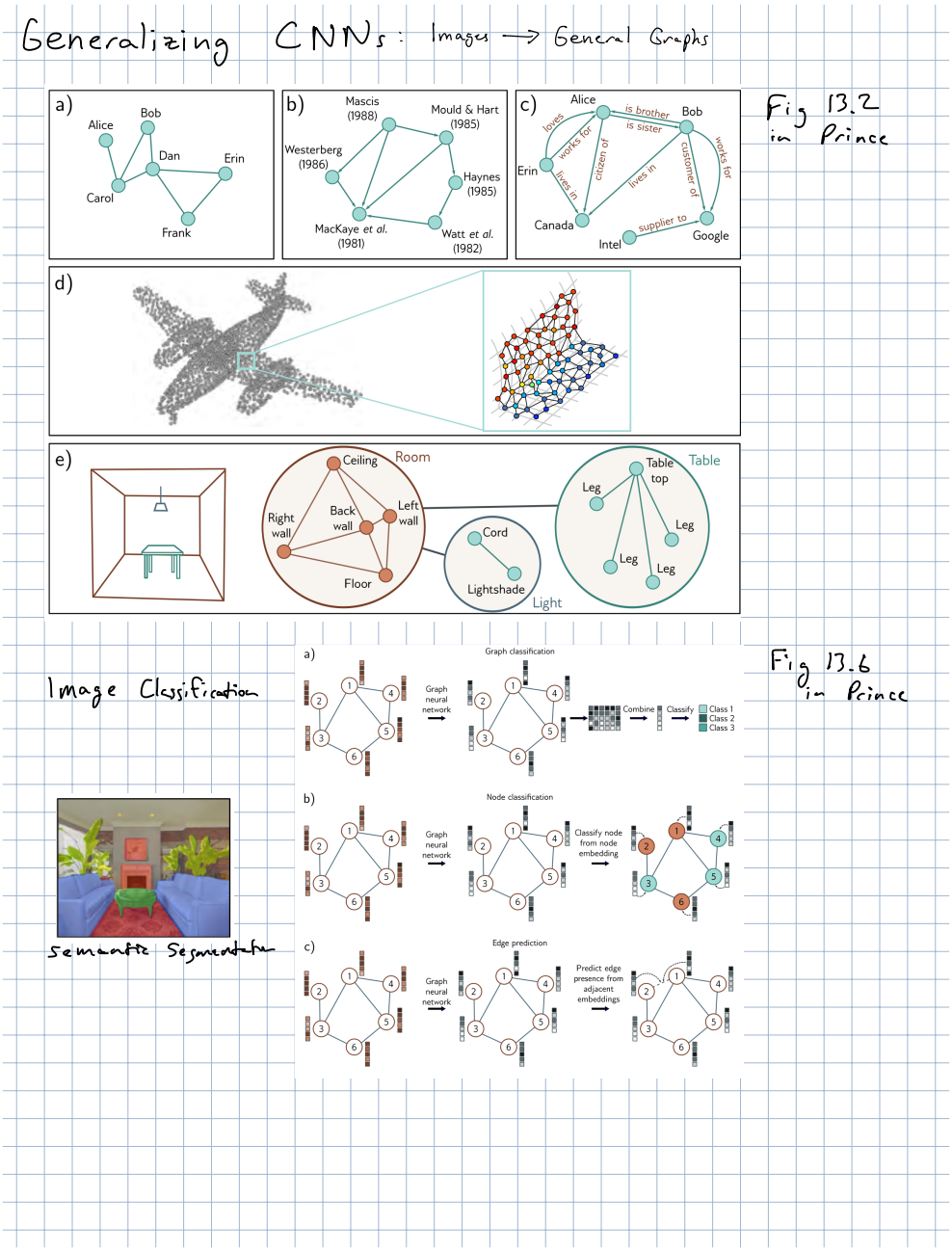


Figure 12.2: Graph domains and levels of prediction. The published examples separate node-level, edge-related, and whole-graph tasks. A graph network's final readout must match the prediction object rather than assuming every task outputs one label per node. Original annotated source excerpt; Anant Sahai, faculty handout 12, p. 3.

12.4 A Conceptual Bridge: Viewing an Image as a Graph

Deeper explanation: From a pixel grid to an arbitrary graph (p. 396)

To understand how to build a GNN, we first reinterpret a CNN's operation through the lens of a graph. Let's consider an image as a specific type of graph.

- **Nodes:** Each pixel in the image is a node in the graph. The value(s) of the pixel (e.g., RGB vector) are the initial features of that node.
- **Edges:** Edges represent the local neighborhood structure. For a pixel, we can draw edges to its 8 neighbors.
- **Graph Properties:**
  - **Directed Graph:** The relationship is directional. The pixel to the north of me has a "North" relationship to me, and I have a "South" relationship to it. These are distinct.
  - **Labeled Edges:** Each edge has a label from a finite set, corresponding to its spatial direction: $\{N, S, E, W, NE, NW, SE, SW\}$.
  - **Self-Loops:** Each node has a special relationship to itself.

A 3x3 Convolution as Message Passing

With this graph perspective, a 3x3 convolution can be seen as a message passing and aggregation step. A central node (pixel) updates its feature vector by receiving "messages" from its neighbors.

The computation for the output feature vector $\vec{h}_{out}$ at a single node is:

$$\vec{h}_{out} = \sigma\left(\vec{b} + W_{\text{self}}\vec{h}_{\text{in,self}} + \sum_{j \in \text{neighbors}} W_{\text{label}(j)}\vec{h}_{\text{in},j}\right) \tag{12.1}$$

where:

- $\vec{h}_{\text{in},j}$ is the input feature vector of neighbor j.
- $W_{\text{label}(j)}$ is a learnable weight matrix associated with the label of the edge from neighbor j (e.g., $W_N, W_S, \dots$).
- W_{self} is a weight matrix for the node's own information.
- $\vec{b}$ is a learnable bias vector.
- σ is a non-linear activation function.

This is mathematically identical to a 3x3 convolution. This re-framing gives us an immediate recipe to generalize CNNs to any directed graph with a finite set of edge labels.

What Generalizes Immediately?

Many core concepts from CNNs carry over directly to this GNN formulation.

- **Weight Sharing:** The same set of weight matrices $(W_N, W_S, \dots)$ is used for every node in the graph, just as a single convolutional filter is applied across all spatial locations in an image.
- **Multiple Channels:** Node features can be vectors, corresponding to channels. The weight matrices W project from input channels to output channels.
- **1x1 Convolutions:** These correspond to applying an MLP-like transformation to each node's feature vector independently.
- **Residual Connections:** The output of a GNN layer can be added to its input.
- **Dropout for MLPs / Stochastic Depth:** These regularization techniques can be applied just as in CNNs.

What Does Not Generalize?

- **Pooling / Downsampling / Upsampling:** These operations are problematic. While downsampling an image to a lower resolution is a natural concept, there is no universal, application-independent way to define a "lower-resolution" version of an arbitrary graph. This means most GNN architectures maintain the same graph structure throughout all layers.

What Needs Discussion: Normalization

Normalization layers need to be adapted to the graph structure.

- **Batch Normalization:** Difficult to apply directly. In a typical batch, different graphs may have different numbers of nodes, so the data tensor is not a simple, uniform block.
- **Layer Normalization:** A feasible option. Normalization statistics (mean, variance) can be computed over all channels and all nodes within a single graph. This assumes the entire graph fits into memory.
- **Per-Embedding (Instance) Normalization:** The simplest and most common choice. Each node's feature vector (embedding) is normalized independently over its channel dimension. The same normalization function (with shared learnable parameters, if any) is applied at each node.

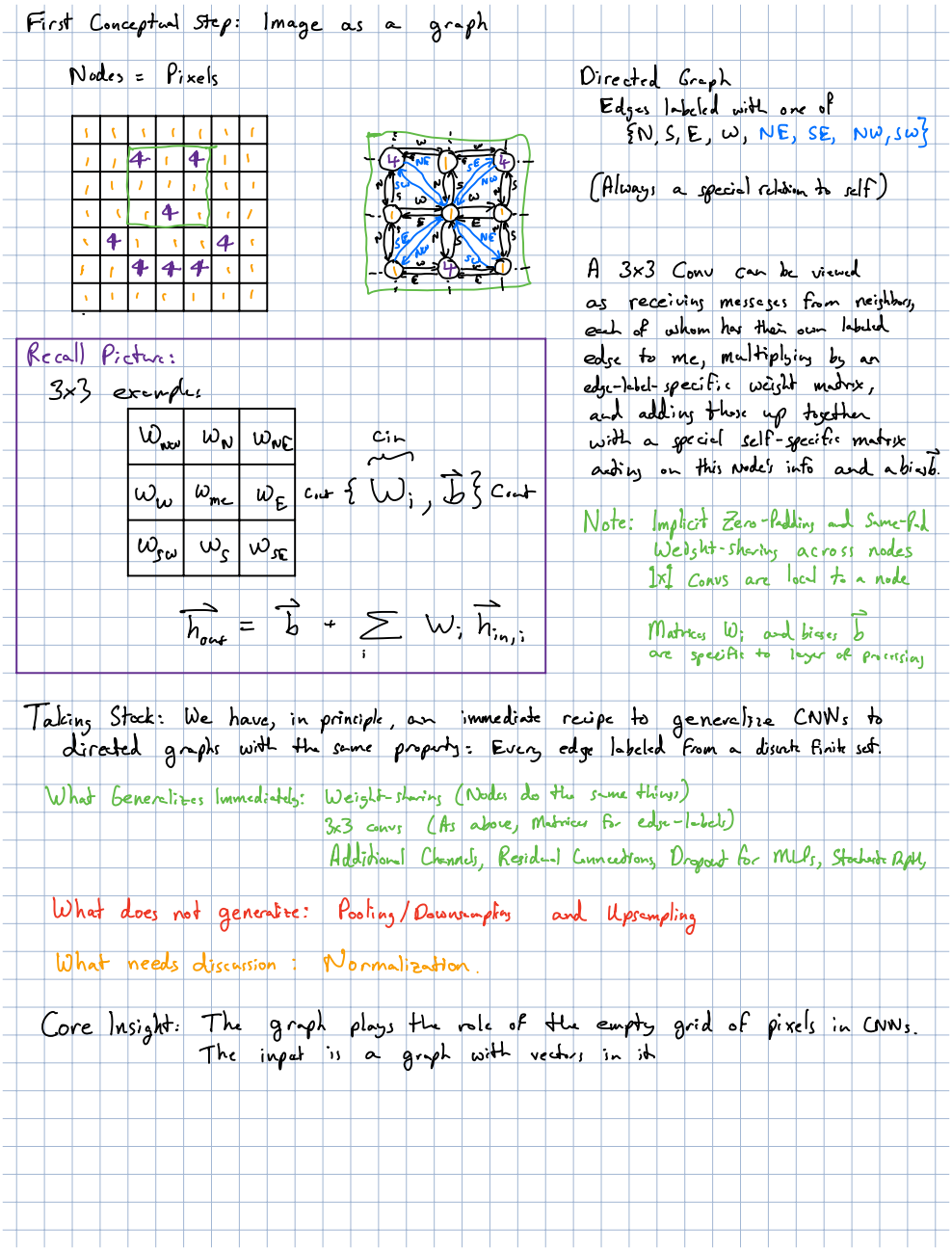


Figure 12.3: An image as a graph with labeled directions. Pixels become nodes and local offsets become edge types. The labels retain relative direction, explaining why an ordinary convolution has more structure than an unlabeled neighbor sum. Original annotated source excerpt; Anant Sahai, faculty handout 12, p. 4.

12.5 Handling General Graph Structures

Deeper explanation: Degree normalization and repeated propagation (p. 398)

Deeper explanation: A numerical message-passing pass (p. 395)

The "image-as-graph" model covers directed graphs with finite edge labels. However, many real-world graphs are different, most notably **undirected graphs with no edge labels**.

A Failed First Attempt

A naive approach is to reduce this case to the one we know.

1. Treat each undirected edge as two directed edges.
2. Assign a single, dummy label ("edge") to all edges.

Following our convolution-as-message-passing formula, the update rule becomes:

$$\vec{h}_{out} = \sigma\left(\vec{b} + W \sum_{j \in \text{neighbors}} \vec{h}_{\text{in},j}\right) \tag{12.2}$$

The single weight matrix W can be pulled out of the sum due to linearity. This means the network can only sum (or average) the feature vectors of its neighbors and then apply a single linear transformation. After a few layers, this repeated averaging causes the features of all connected nodes to converge to a single value. This phenomenon is known as **over-smoothing**, and it severely limits the expressive power of the GNN.

A Principled Approach: Permutation Invariance

To overcome over-smoothing, we must design an architecture that respects the fundamental property of an unlabeled graph: a node's neighborhood is a **set**, not an ordered list. The aggregation function must be **permutation-invariant**.

A general form for a GNN layer respecting this constraint is:

$$\vec{h}_{out} = f_{W_1}\left(\vec{h}_{\text{in,self}}, \bigoplus_{j \in \text{neighbors}} g_{W_2}(\vec{h}_{\text{in},j})\right) \tag{12.3}$$

where $\bigoplus$ is a permutation-invariant aggregation operator.

Choices for Permutation-Invariant Aggregation There is a rich vocabulary of operations that do not depend on the order of their inputs:

- **Simple aggregators:** Sum, Mean, Max, Min, Product.
- **Statistical measures:** Median, Variance, general order statistics.
- **Machine Learning models:** Any ML model that operates on a set of data points is permutation-invariant. This includes powerful options like Least Squares or even kernel methods.

- **Attention-based mechanisms:** Using Softmax to create weighted combinations.

The need to handle this constraint in GNNs spurred creativity and led to the development of more powerful and general aggregation mechanisms.

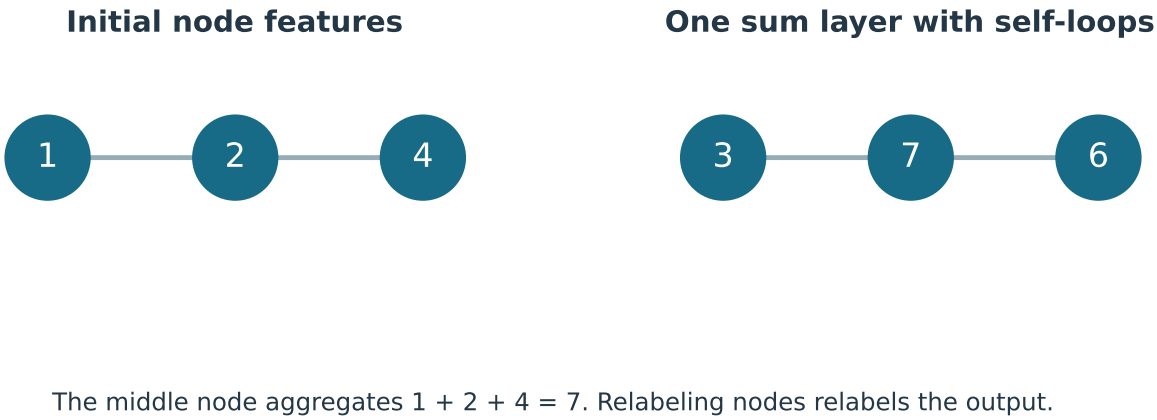


Figure 12.4: A node receives messages from its neighbors, aggregates them without an arbitrary ordering, and updates its feature.

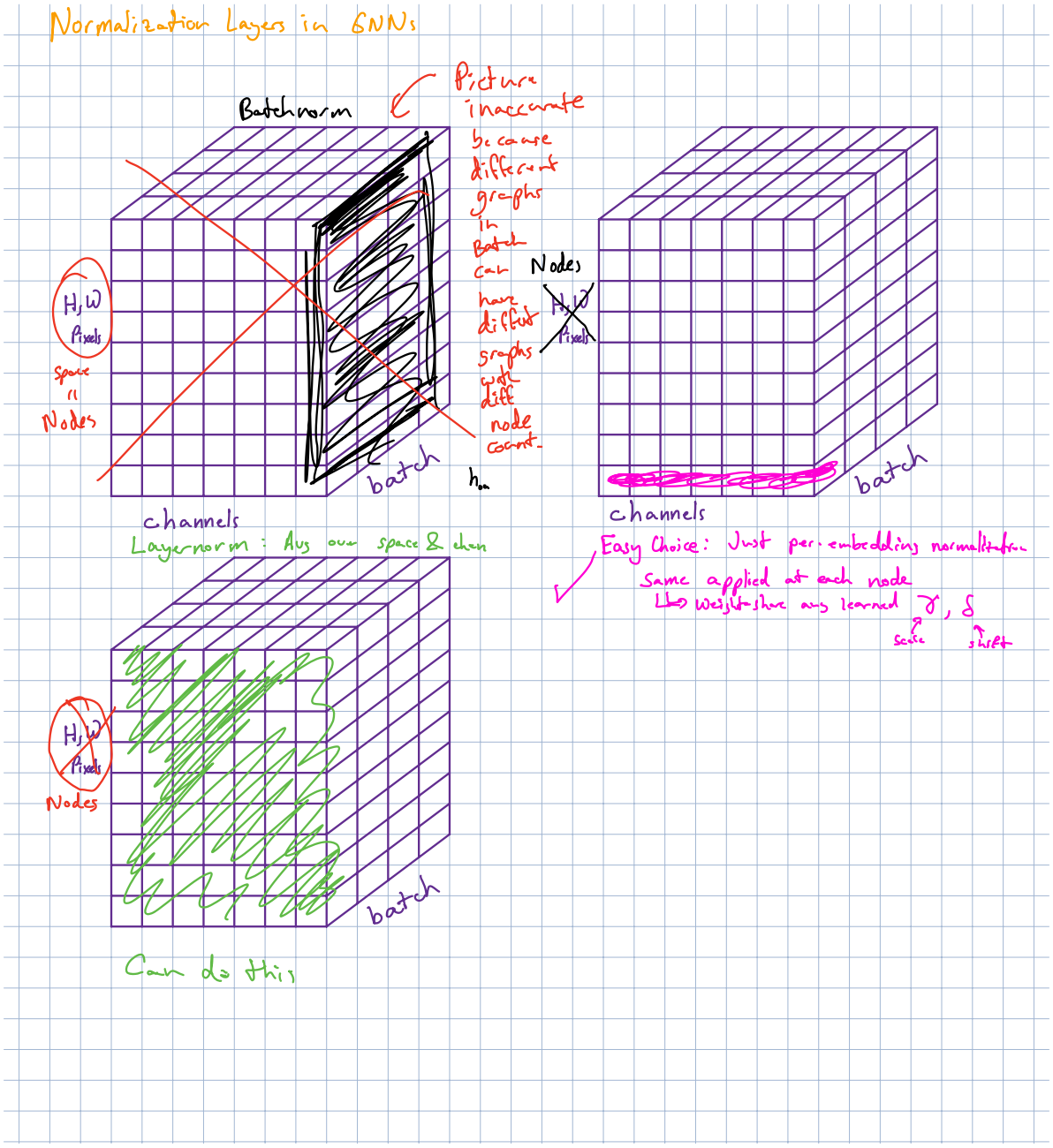


Figure 12.5: Normalization cubes for graph node features. The node axis replaces the image-spatial axis. The per-node feature slice gives a simple LayerNorm convention. The crossed-out BatchNorm sketch flags the lecture's batching inconvenience for variable-size graphs, not a mathematical prohibition on using BatchNorm with graphs. Original annotated source excerpt; Anant Sahai, faculty handout 12, p. 5.

Batching graphs without creating edges between them

For two independent graphs, stack node features and use a block-diagonal adjacency:

$$A_{\text{batch}} = \begin{bmatrix} A_1 & 0 \\ 0 & A_2 \end{bmatrix}, \quad H_{\text{batch}} = \begin{bmatrix} H_1 \\ H_2 \end{bmatrix}, \quad A_{\text{batch}} H_{\text{batch}} W = \begin{bmatrix} A_1 H_1 W \\ A_2 H_2 W \end{bmatrix}.$$

Here $A_g \in \mathbb{R}^{n_g \times n_g}$, $H_g \in \mathbb{R}^{n_g \times d}$, and $W \in \mathbb{R}^{d \times d'}$. Shared W does not create messages across graphs: the zero off-diagonal adjacency blocks prevent that. Adding self-loops and computing degrees within each graph preserves this structure. A pointwise activation also preserves separation; a normalization that pools statistics across all batched nodes can introduce a different kind of cross-example dependence.

Worked derivation: permutation equivariance and graph readout

Lecture basis: faculty handout 12, pp. 5–8. The steps below expand the lecture argument.

Step 1: express a node relabeling. Let $H \in \mathbb{R}^{n \times d}$ hold node features and $A \in \mathbb{R}^{n \times n}$ hold adjacency weights. A permutation matrix P satisfies $P^\top P = I$, and relabeling gives $H' = PH$, $A' = PAP^\top$.

Step 2: push the permutation through a layer. For $F(A, H) = \phi(AHW + \mathbf{1}b^\top)$ with the same pointwise nonlinearity and weights at each node,

$$\begin{aligned} F(A', H') &= \phi(PAP^\top PHW + \mathbf{1}b^\top) \\ &= \phi(PAHW + P\mathbf{1}b^\top) \\ &= P\phi(AHW + \mathbf{1}b^\top) = PF(A, H). \end{aligned}$$

We used $P\mathbf{1} = \mathbf{1}$ and the fact that permuting rows commutes with a shared rowwise operation. Thus the output rows reorder with the input rows: equivariance. For degree normalization, $D' = PDP^\top$, so

$$D'^{-1/2} A' D'^{-1/2} = P(D^{-1/2} A D^{-1/2}) P^\top,$$

with a consistent convention for zero degrees. The same proof follows. Neighbor attention also works when shared edge scores are normalized over the relabeled neighbor set; an arbitrary fixed neighbor ordering would break the argument.

Step 3: remove node labels at graph level. A sum readout $r(H) = \mathbf{1}^\top H$ obeys

$$r(PH) = \mathbf{1}^\top PH = \mathbf{1}^\top H = r(H).$$

The result is invariant rather than equivariant. A sequence of equivariant message-passing layers followed by invariant readout therefore defines an invariant graph predictor. This symmetry guarantee does not imply that all nonisomorphic graphs can be distinguished.

12.6 The Attention Mechanism in GNNs

Deeper explanation: Graph attention is a normalized neighbor-weighting rule (p. 400)

A more expressive and general architecture allows the self-node's features to influence how neighbor information is aggregated. A highly effective way to structure this is through an **attention mechanism**. This is motivated by the idea of using a learned similarity to weight the importance of different neighbors.

A general formulation inspired by kernel methods is:

$$\vec{h}_{out} = f\left(\vec{h}_{\text{in,self}}, \sum_{j \in \text{neighbors}} S(\vec{h}_{\text{in,self}}, \vec{h}_{\text{in},j}) \cdot g(\vec{h}_{\text{in},j})\right) \tag{12.4}$$

Here, S is a learned similarity function. To control the dynamic range of the sum, we can frame it as an expectation. This is achieved by normalizing the similarity scores into probabilities using a **softmax** function.

An Example GNN Block with Attention

Let's assemble a complete GNN block using these design principles, wrapped in a residual connection for stable training.

For each node with input embedding $\vec{h}_0$ and neighbor embeddings $\{\vec{h}_i\}$:

1. **Projections (Query, Key, Value):** Project the embeddings using learnable linear layers (matrices P_q, P_k, P_v).

$$\begin{aligned} \text{query: } \vec{q} &= P_q \vec{h}_0 \\ \text{keys: } \vec{k}_i &= P_k \vec{h}_i \\ \text{values: } \vec{v}_i &= P_v \vec{h}_i \end{aligned}$$

2. **Compute Attention Scores:** Calculate a similarity score between the self-node's query and each neighbor's key, often using a simple dot product.

$$s_i = \vec{q}^T \vec{k}_i \tag{12.5}$$

3. **Normalize Scores to Weights:** Apply softmax over the scores of all neighbors to get attention weights α_i that sum to 1.

$$\vec{\alpha} = \text{softmax}(\vec{s}) \tag{12.6}$$

4. **Aggregate Values:** Compute a weighted sum of the neighbor values using the attention weights. This is the aggregated neighborhood information.

$$\vec{h}_{\text{agg}} = \sum_{i \in \text{neighbors}} \alpha_i \vec{v}_i \tag{12.7}$$

5. **Update Node Representation:** The aggregated information is passed through a final transformation (e.g., an MLP) and added to the original input via a residual connection.

$$\vec{h}_{out} = \vec{h}_0 + \text{MLP}(\vec{h}_{\text{agg}}) \tag{12.8}$$

This entire process of computing query-key scores, applying softmax, and taking a weighted sum of values is the core of the **attention mechanism**. The necessary non-linearity to avoid over-smoothing comes from the softmax function and the multiplications within the attention computation and the final MLP.

Connection to Transformers

This attention mechanism, developed here from the constraints of operating on general graphs, is the fundamental building block of the **Transformer** architecture. A Transformer can be viewed as a GNN operating on a **complete graph**, where every node (e.g., a word in a sentence) is connected to every other node and attends to them to build a context-aware representation.

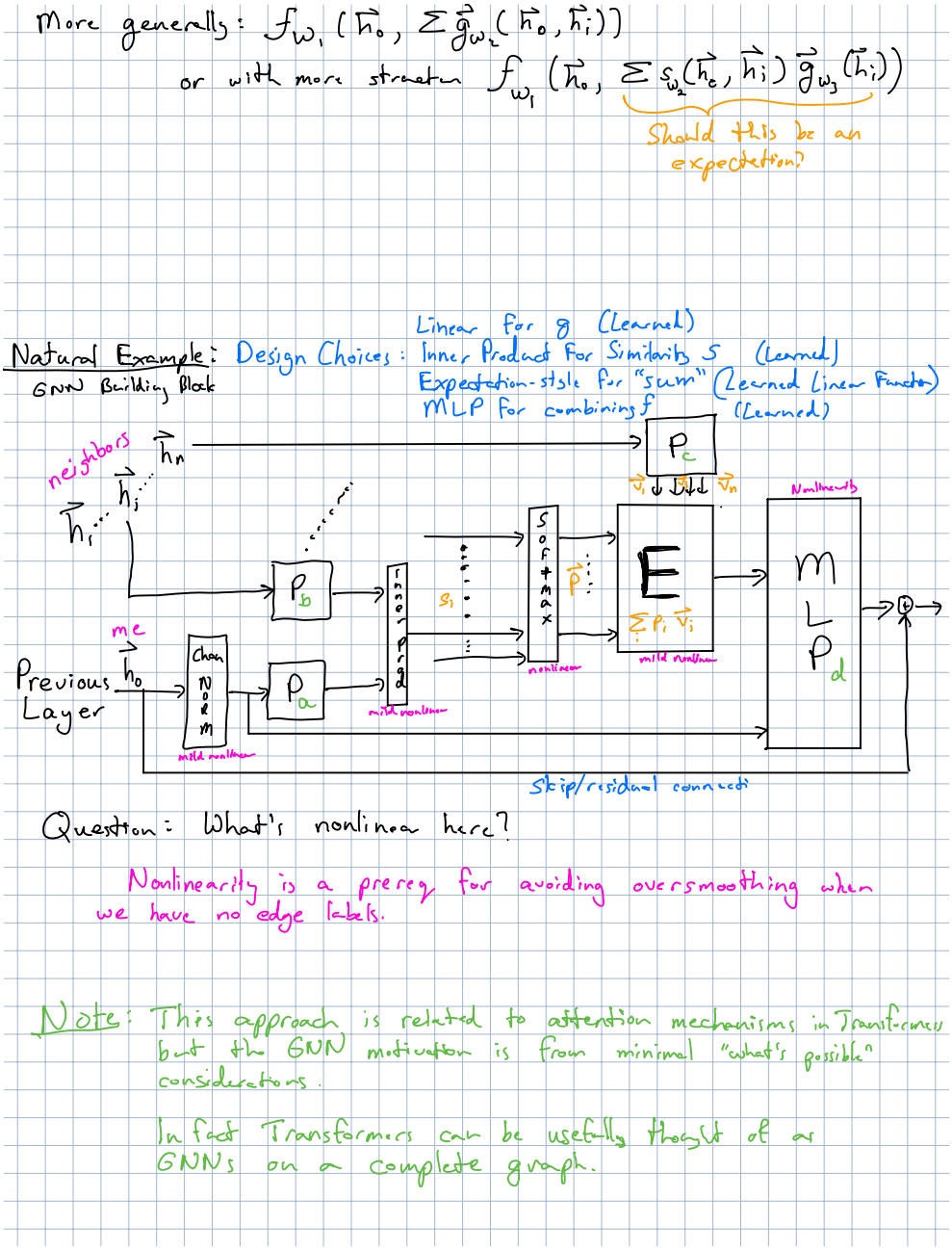


Figure 12.6: The complete graph-attention block. Trace queries and keys into neighbor scores, normalize those scores, and use them to combine values. The surrounding normalization, MLP, and residual route turn that aggregation into a network block. Original annotated source excerpt; Anant Sahai, faculty handout 12, p. 7.

Lecture 13

GNN Pooling and Intro to RNNs

Course sources: lecture_13; 2025-10-14 transcript.

13.1 Administrative Notes (Fall 2025)

These announcements describe the original Fall 2025 lecture context.

Reading Material

- The material on Graph Neural Networks (GNNs) corresponds to Chapter 13 in the Prince textbook (*Understanding Deep Learning*).
- The content in this lecture, particularly on GNN pooling and Recurrent Neural Networks (RNNs), is not covered in the Prince textbook.
- For classic RNN material, a good (though older) reference is the online textbook at `deeplearningbook.org`. The perspectives in this course will be more modern.

Project Information

Key project deadlines and milestones will be announced, but here are the highlights:

- **Team Formation:** Groups of four.
- **Proposal Due:** November 3rd. Details on the proposal format will be provided.
- **Required Meetings:** Every project group must sign up for a meeting slot with the teaching staff in the week leading up to the proposal deadline to discuss their ideas.
- **Feedback Meetings:** Another meeting will occur the week after the proposal submission to get feedback from the TAs.
- **Draft Report:** A draft report will be due the week before Thanksgiving.
- **Peer Review:** You will be expected to conduct peer reviews of other teams' draft reports.
- **Final Report:** The final project report is due on the last day of class.
- **Poster Presentation:** A poster presentation will take place during RRR week.

13.2 Finishing Graph Neural Networks (GNNs)

Deeper explanation: DiffPool gradients and what a soft assignment means (p. 401)

We previously generalized Convolutional Neural Networks (CNNs) to operate on graphs. We established a framework for directed graphs with a finite set of edge labels, where CNN ideas translate directly. The key remaining challenge is handling pooling and downsampling, which does not have a straightforward generalization, especially for undirected graphs with no edge labels.

The Challenge of Pooling and Downsampling in GNNs

Why Do We Pool Anyway? The primary motivation for pooling and downsampling in CNNs is to **grow the receptive fields faster with each layer**. This allows the network to capture attributes of the input (e.g., an image) that exist at a larger spatial scale. Without pooling, a deep stack of small convolutions (e.g., 3x3) would be needed to achieve a large receptive field, which is computationally inefficient and may not capture the desired inductive bias.

This motivation is still highly relevant for GNNs. In a graph, information from distant nodes propagates slowly through message passing, one hop per layer. To make decisions based on global graph structure, we need an effective way to aggregate information from distant parts of the graph quickly.

A Simple Hack: The Global Node A straightforward, albeit blunt, solution to accelerate information flow is to introduce a **fake global node**.

- This node is added to the graph and connected to every other node.
- **Step 1:** In one layer of message passing, all nodes send their information to this global node.
- **Step 2:** The global node aggregates this information. In the next layer, it broadcasts the aggregated information back to all other nodes.

After two layers, every node has access to information from the entire graph, effectively making its receptive field global.

Drawbacks:

- **Coarseness:** This method is an all-or-nothing approach. It provides access to local structure (immediate neighbors) and global structure, but not intermediate, multi-hop structures that might be important.
- **Oversquashing:** This introduces a significant information bottleneck. If the graph has thousands of nodes, all of their information must be compressed ("squashed") into the

single, finite-dimensional embedding of the global node. This can lead to a substantial loss of detail. Oversquashing is a general challenge in GNNs where information from many nodes must be funneled through a smaller-dimensional representation.

Note: Oversquashing is distinct from *oversmoothing*, where repeated averaging operations cause node representations to become indistinguishable.

A Principled Approach: Differentiable Pooling (DiffPool)

To create a more nuanced pooling mechanism, we need to go from many nodes to fewer nodes based on the graph's structure and the nodes' vector contents. This problem is analogous to **clustering**.

The Clustering Analogy Classic clustering algorithms like k-means (Lloyd's algorithm) perform a similar task. However, k-means involves a step that is not differentiable: the hard assignment of each point to its nearest cluster center. This assignment is like a step function; a small change in a center's position can cause a point to abruptly switch clusters, which means the derivative is not well-defined.

Making Clustering Differentiable with Soft Assignments To overcome the non-differentiability of hard assignments, we can use an idea similar to softmax: **soft assignments**.

- Instead of assigning each node to a single cluster, we assign it to *every* cluster with a certain probability.
- Each node will have a probability distribution over the new, coarsened "cluster-nodes".
- A consequence of this is that the resulting downsampled graph will be fully connected, with edges weighted by the probabilistic assignments.

This approach is detailed in the paper "Hierarchical Graph Representation Learning with Differentiable Pooling" by Ying et al., 2018.

The DiffPool Module: A GNN for Pooling Since we don't know the optimal way to cluster nodes for a given task, we follow the deep learning principle: **make it learnable**. We design a learnable module (a "grey-box") that performs this soft clustering.

Module Specification

- **Input:** The current graph with node embeddings. Let's say we have N nodes and are at layer l, with embeddings $H^{(l)} \in \mathbb{R}^{N \times d}$.

- **Output:** A probability vector for each node, indicating its membership in k new clusters. This can be represented by an assignment matrix $S \in \mathbb{R}^{N \times k}$, where S_{ij} is the probability that node i belongs to cluster j. Each row of S sums to 1.

Implementation The task of generating these assignment vectors is essentially a **node-level classification problem** on the graph. The most powerful tool we have for learnable tasks on graphs is a GNN itself. Therefore, the DiffPool module is implemented as a **separate, smaller GNN**.

1. This "pooling GNN" takes the current node embeddings $H^{(l)}$ as input.
2. It performs several layers of message passing to produce new embeddings that are suitable for the clustering task.
3. A final linear layer followed by a 'softmax' operation on each node's output generates the desired assignment matrix S.

The parameters of this pooling GNN are trained jointly with the main GNN.

Constructing the Coarsened Graph Once the assignment matrix S is computed, we can construct the new, smaller graph with k nodes.

1. New Node Embeddings The embedding for each of the k new cluster-nodes is a weighted average of the original node embeddings, where the weights are the assignment probabilities. The content of the original nodes is "poured" into the new clusters.

$$H^{(l+1)} = S^T H^{(l)}$$

Here, $H^{(l+1)} \in \mathbb{R}^{k \times d}$ are the embeddings for the k new nodes.

2. New (Soft) Adjacency Matrix We also need a new adjacency matrix $A^{(l+1)} \in \mathbb{R}^{k \times k}$ for the coarsened graph. This matrix should represent the connection strength between the new cluster-nodes. It can be computed as:

$$A^{(l+1)} = S^T A^{(l)} S$$

The entry $A_{ij}^{(l+1)}$ represents the total strength of connections between the nodes that constitute cluster i and the nodes that constitute cluster j. This results in a dense, weighted adjacency matrix. Our GNN layers must be able to handle these scalar edge weights (e.g., by multiplying messages by the edge weight).

Training with Auxiliary Losses While DiffPool can be trained end-to-end via the main task loss, this can be difficult. The pooling GNN starts with random weights and may produce poor clusterings, leading to noisy gradients. To guide the training, we introduce **auxiliary losses** that encourage desirable properties in the clustering. These losses are added to the main loss during training.

Property 1: Clusters should respect the graph topology. Nodes that are connected in the original graph should be likely to be assigned to the same cluster. This can be encouraged with a **link prediction loss**.

$$\mathcal{L}_{\text{link}} = \|A^{(l)} - SS^T\|_F^2$$

Intuition: The matrix $A^{(l)}$ represents the graph's connectivity. The matrix $SS^T \in \mathbb{R}^{N \times N}$ is a low-rank approximation of connectivity based on cluster assignments. The (i, j)-th entry of SS^T is the inner product of the assignment vectors for nodes i and j. This loss encourages connected nodes (where $A_{ij} = 1$) to have similar assignment vectors (large inner product), thus placing them in the same cluster. This helps the GNN find community structures.

Property 2: Cluster assignments should be close to hard assignments. We introduced soft assignments for differentiability, but we would prefer if each node primarily belongs to one cluster. This can be encouraged by penalizing high entropy in the assignment vectors.

$$\mathcal{L}_{\text{entropy}} = \frac{1}{N} \sum_{i=1}^{N} H(\vec{s}_i)$$

where $H(\cdot)$ is the entropy function and $\vec{s}_i$ is the i-th row of S. Entropy is minimized when a probability distribution is a one-hot vector (i.e., a hard assignment). This loss encourages the softmax outputs to be sharp and decisive.

These auxiliary losses are combined with the main task loss, usually with hyperparameters to weight their contribution. The gradients from all losses flow back to update both the main GNN and the pooling GNN.

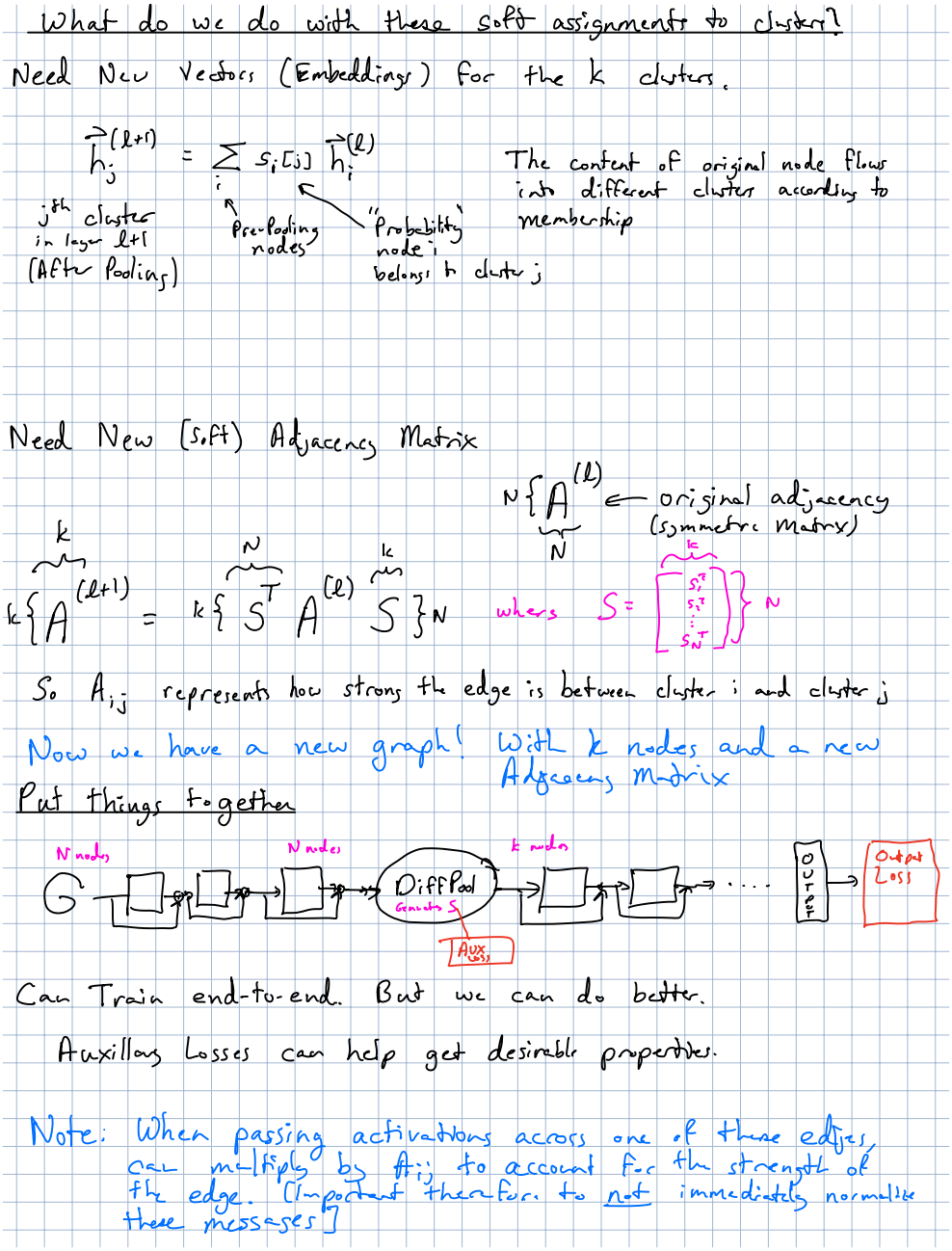


Figure 13.1: Where DiffPool sits in the network. One graph network produces node embeddings while another produces assignment scores. Pooling contracts features and both edge endpoints, producing a smaller graph that can feed subsequent graph layers. Original annotated source excerpt; Anant Sahai, faculty handout 13, p. 6.

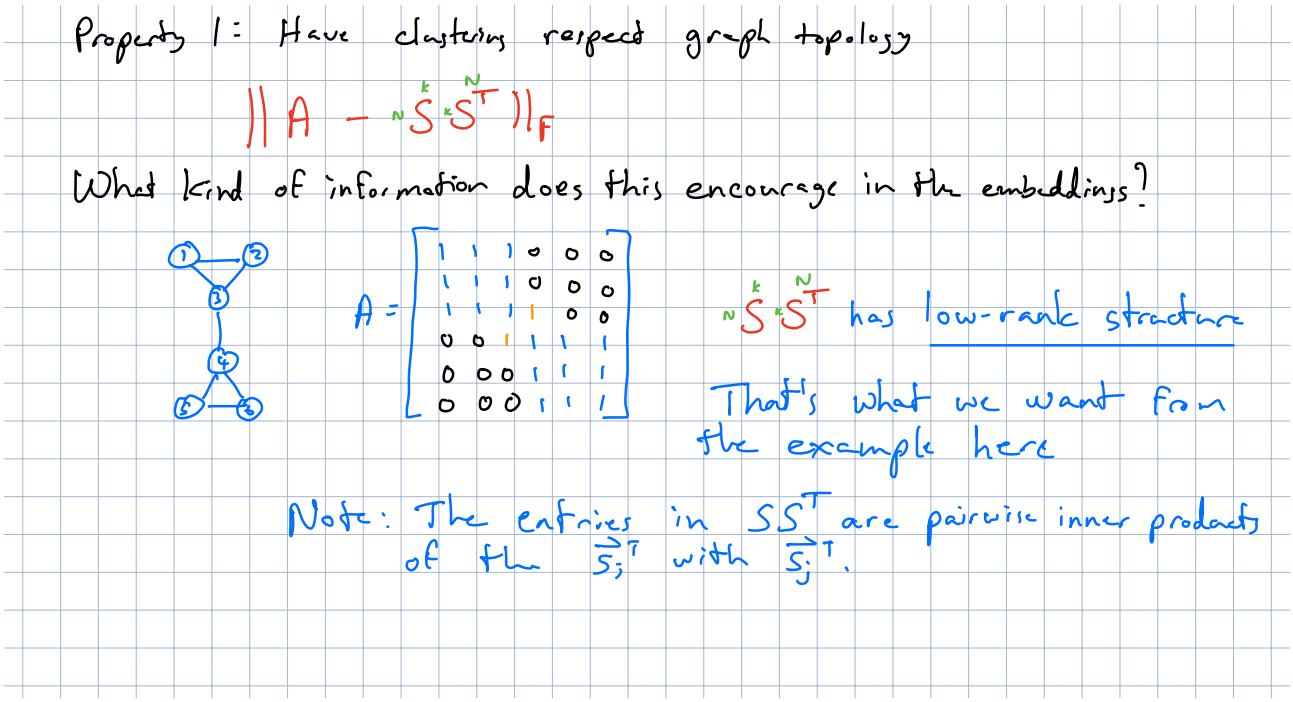


Figure 13.2: Graph clusters and block structure in adjacency. When nodes in the same cluster connect strongly, a suitable node ordering makes blocks visible in the adjacency matrix. The auxiliary reconstruction objective encourages assignments to reflect this topology; it does not force a unique clustering. Original annotated source excerpt; Anant Sahai, faculty handout 13, p. 7.

Worked derivation: the pooled feature and adjacency blocks

Lecture basis: faculty handout 13, pp. 5–7. The steps below expand the lecture argument.

Step 1: distinguish embeddings from assignments. Let $Z \in \mathbb{R}^{n\times d}$ be learned node embeddings and $S \in \mathbb{R}^{n\times k}$ row-softmax assignments, so $S_{ia} \geq 0$ and $\sum_a S_{ia} = 1$. Rows describe original nodes; columns describe learned clusters.

Step 2: derive the feature aggregation entry by entry. For cluster a and feature c,

$$Z'_{ac} = \sum_{i=1}^{n} S_{ia} Z_{ic} = (S^\top Z)_{ac}.$$

This is a weighted sum. It is a weighted mean only if divided by cluster mass $\sum_i S_{ia}$, which standard sum-style DiffPool does not automatically do.

Step 3: aggregate both ends of every edge. An edge of weight A_{ij} contributes $S_{ia}A_{ij}S_{jb}$ to the pooled edge from cluster a to cluster b. Therefore

$$\begin{aligned} A'_{ab} &= \sum_i \sum_j S_{ia} A_{ij} S_{jb} \\ &= (S^\top A S)_{ab}. \end{aligned}$$

For hard assignments, S_{ia} is an indicator and the sum simply counts weighted edges between two clusters. Edges inside a cluster become diagonal entries; pooled adjacency is generally weighted and may have self-loops.

Step 4: verify invariance of the coarsened graph. If the two assignment and embedding networks are equivariant, relabeling gives $\widetilde{S} = PS$ and $\widetilde{Z} = PZ$. Hence

$$\begin{aligned} \widetilde{S}^\top \widetilde{Z} &= S^\top P^\top P Z = S^\top Z, \\ \widetilde{S}^\top (P A P^\top) \widetilde{S} &= S^\top P^\top P A P^\top P S = S^\top A S. \end{aligned}$$

The cluster-level representation is unchanged by an input-node relabeling. The auxiliary matrix $(SS^\top)_{ij} = \sum_a S_{ia} S_{ja}$ measures assignment overlap; matching it to adjacency encourages linked nodes to share clusters. Row entropy $-\sum_a S_{ia} \log S_{ia}$ is minimized at hard assignments, but neither auxiliary objective by itself guarantees useful task-level clusters.

13.3 Final GNN Challenge: Handling Held-Out Data

Deeper explanation: A concrete train/test leakage example on a graph (p. 395)

In standard machine learning, it is crucial to have held-out data for validation and testing. For node-level tasks on a single, large graph (e.g., a social network), we cannot simply hold out an entirely separate graph. We must hold out parts of the single graph we possess.

Approach 1: Excise Entire Nodes

One method is to completely remove a set of nodes and their corresponding edges from the graph during training. These nodes are then used for testing.

- **Challenge:** This fundamentally **alters the graph's connectivity**. Removing nodes can break important information pathways for the remaining nodes. For structured graphs like molecules, removing an atom changes the identity of the molecule itself.

Approach 2: Mask Node Content

A better approach is to remove the *content* (feature vector) from a node but leave the node's structure (its connections) intact within the graph.

- **Problem:** What do we replace the feature vector with? We cannot use a null value, as the network's operations require a vector.
- **Solution:** Make it learnable. We replace the features of any masked node with a single, shared, learnable vector, often called a **mask token**.

Training Process:

1. A learnable vector, $\vec{v}_{\text{mask}}$, is initialized.
2. During training, we perform data augmentation: we randomly select some nodes from the training set, replace their feature vectors with $\vec{v}_{\text{mask}}$, but *keep* the loss calculation on their original labels.
3. The network learns to use the information from a masked node's neighbors to make a correct prediction for that node. In doing so, it learns both a meaningful representation for $\vec{v}_{\text{mask}}$ and how to process it.
4. For the actual held-out (test) nodes, we replace their features with this learned $\vec{v}_{\text{mask}}$ and evaluate the model's predictions.

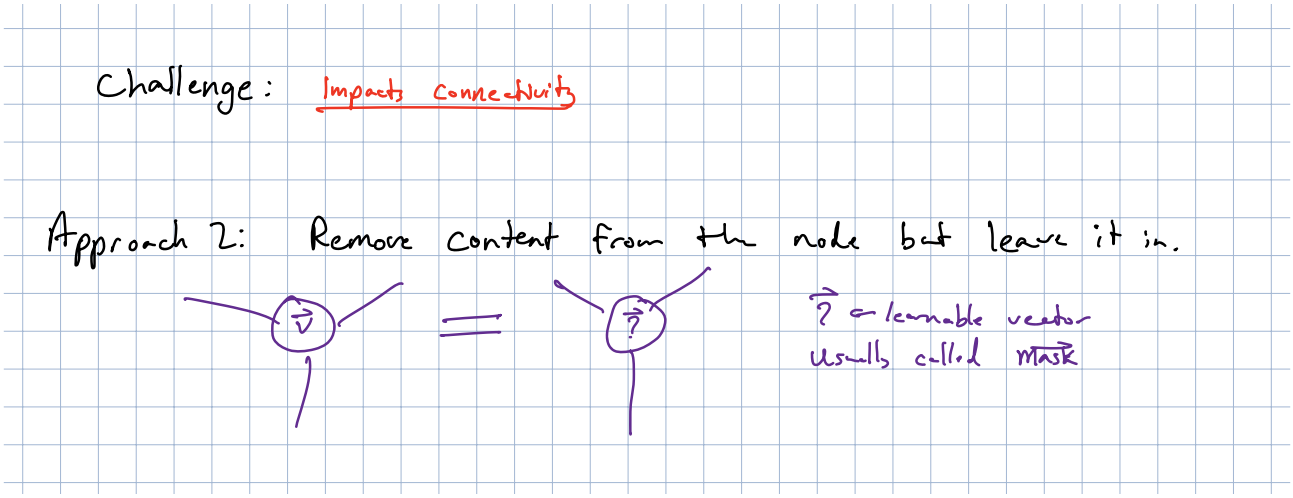


Figure 13.3: Held-out nodes and visible graph structure. The sketch separates removal of a node from hiding its features or label. Which information remains visible defines the evaluation protocol; masking labels alone does not establish an inductive split. Original annotated source excerpt; Anant Sahai, faculty handout 13, p. 8.

13.4 Introduction to Recurrent Neural Networks (RNNs)

Deeper explanation: Backpropagation through time with a loss at every step (p. 406)

Historical Context

Before 2018, the deep learning landscape was largely dominated by two architectures:

- **CNNs** for vision and other spatial data.
- **RNNs** for sequential data like language and speech.

In 2017, the paper "Attention Is All You Need" introduced the Transformer architecture, which quickly began to outperform RNNs on their home turf. For a time, RNNs were considered by many to be obsolete. However, recent state-of-the-art models have begun to re-incorporate ideas from the RNN lineage, showing their continued relevance.

The Core Idea: Sequential and Causal Data

The key characteristic of data like language, speech, and time-series is that it is **sequential**. It has a one-dimensional topology with a strict ordering and a natural **causal** flow: the present depends on the past, but not the future. RNNs are architectures designed specifically to model this structure.

A Signal-Processing Perspective

We can draw an analogy to classical signal processing filters:

- **CNNs** are analogous to **Finite Impulse Response (FIR) filters**. They perform weight sharing across **space**.
- **RNNs** are analogous to **Infinite Impulse Response (IIR) filters**. They maintain a finite-dimensional hidden state and perform weight sharing across **time**.

Our approach to understanding RNNs will be to adapt the design principles we learned from CNNs, but with two key changes:

1. Replace the notion of "space" with "time".
2. Enforce causality, meaning the computation at a given timestep can only depend on current and past inputs.

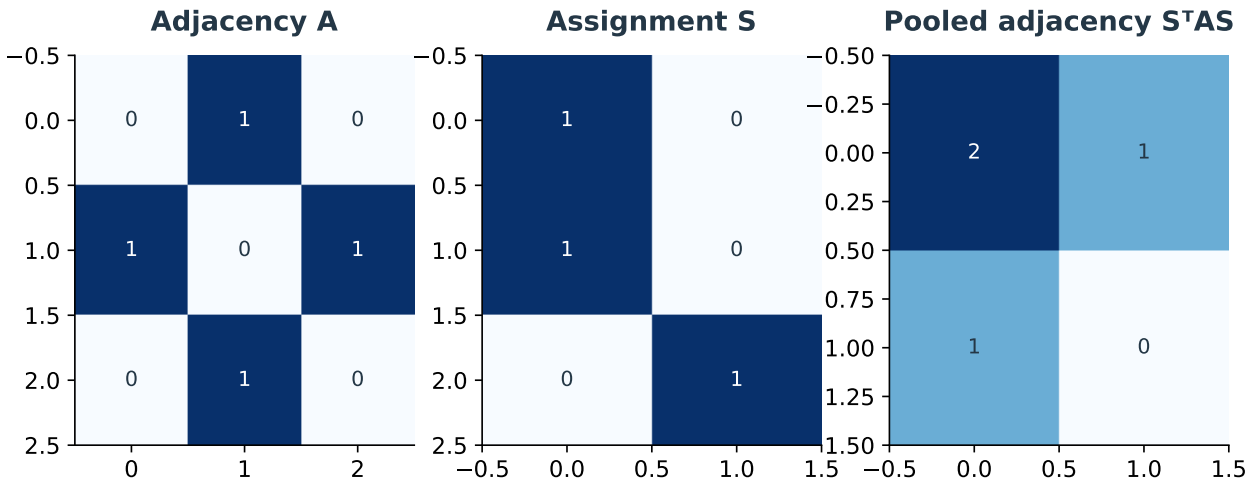


Figure 13.4: Soft assignments turn a graph into a smaller weighted graph. Features and adjacency must both be pooled.

Lecture 14

Recurrent Neural Networks and Self-Supervision

Course sources: lecture_14; 2025-10-16 transcript.

14.1 Introduction to Recurrent Neural Networks (RNNs)

Deeper explanation: Gates create controllable memory paths (p. 408)

Context and History

Recurrent Neural Networks (RNNs) represent a significant class of neural network architectures, historically parallel to Convolutional Neural Networks (CNNs) in their success. Before 2018, the consensus view held two major successes in deep learning:

- **CNNs** for images and computer vision.
- **RNNs** for language, speech, and other Natural Language Processing (NLP) tasks.

The key challenge that RNNs address is the processing of **sequential data**. This includes domains like language, speech, control systems, and time-series analysis, where data points arrive in a specific order and often exhibit causal relationships (i.e., later events depend on earlier ones).

A Signal-Processing Perspective

We can draw a useful analogy between neural network components and classical signal processing filters.

- **Finite Impulse Response (FIR) Filters:** These filters have a finite memory of past inputs. This concept maps directly to **CNNs**, where weight sharing occurs **across space**. The receptive field is finite and grows with network depth.
- **Infinite Impulse Response (IIR) Filters:** These filters, in principle, have an infinite memory of the past. For practical, tractable implementations, they are designed with a **finite-dimensional hidden state**. This concept of a filter with memory maps to **RNNs**. The crucial difference is that weight sharing occurs **across time**. These systems are fundamentally sequential and causal.

Inside a Linear IIR Filter A standard linear IIR filter operates on an input sequence $\vec{u}_t$ to produce an output sequence $\vec{y}_t$. It maintains a hidden state $\vec{h}_t$, which acts as a running summary of the relevant parts of the input history. The update process is defined by two linear equations:

1. **State Update (Causal):** The new hidden state $\vec{h}_{t+1}$ is a linear combination of the previous state $\vec{h}_t$ and the current input $\vec{u}_{t+1}$.

$$\vec{h}_{t+1} = W_h\vec{h}_t + B\vec{u}_{t+1} \tag{14.1}$$

   Here, W_h and B are weight matrices. The hidden state is typically initialized to zero.

2. **Output Generation:** The output $\vec{y}_{t+1}$ is a linear combination of the current hidden state $\vec{h}_{t+1}$ and, optionally, a direct feed-through of the current input $\vec{u}_{t+1}$.

$$\vec{y}_{t+1} = C\vec{h}_{t+1} + D\vec{u}_{t+1} \tag{14.2}$$

   Here, C and D are weight matrices. The dimensions of $\vec{u}$, $\vec{h}$, and $\vec{y}$ can all be different, with the matrices sized accordingly.

The core idea is to treat this IIR filter block as the sequential counterpart to a convolution layer in a CNN.

14.2 Designing Modern RNNs

Deeper explanation: Gates create controllable memory paths (p. 408)

Generalizing from CNN Principles

We adopt the design principles from CNNs and adapt them for sequential data. We replace space with time and enforce causality. The conventional visualization for an RNN architecture is as follows:

- **Horizontal Axis:** Represents time or the sequence order.
- **Vertical Axis:** Represents the depth of the network (layers).

Information flows from left to right (through time) and from bottom to top (through layers).

Many design ideas from CNNs carry over directly in the vertical (depth) dimension:

- **Vertical Residual Connections:** Skip connections can be added between layers at the same time step, just as in ResNets.
- **Vertical "1x1" MLPs:** Multi-Layer Perceptrons can be applied to the feature vector at each time step independently, analogous to 1x1 convolutions.

- **Vertical Normalizations:** Normalization schemes like LayerNorm or RMSNorm can be applied to the feature vector at a specific time step. It's crucial that these normalizations do not "look into the future"; they operate only on the local channels at the current time step.

Traditional "Vanilla" RNNs

A key design question is: **Where do the non-linearities go?** To achieve expressive power, non-linearities are essential.

The traditional RNN perspective places a non-linearity in the state update mechanism. We start with the linear dynamics and modify them:

1. We can drop the feed-through term $D\vec{u}_{t+1}$ from the core RNN block, as this can be implemented as a parallel skip connection if needed.
2. We add learnable biases, a standard practice in neural networks.
3. We wrap the linear state update inside a non-linear activation function σ.

This yields the state update equation for a traditional RNN cell:

$$\vec{h}_{t+1} = \sigma(W_h\vec{h}_t + B\vec{u}_{t+1} + \vec{b}_h) \tag{14.3}$$

The output of an RNN layer is typically just the hidden state itself, $\vec{y}_{t+1} = \vec{h}_{t+1}$. If further transformation is needed, an explicit linear layer can be added on top.

Choice of Non-Linearity Unlike in CNNs where ReLU and its variants dominate, classical RNNs typically use saturating non-linearities like **sigmoid** or **tanh**. The reason for this choice relates to the problem of exploding activations. Since the recurrent update is applied repeatedly over many time steps, an expansive mapping (like ReLU) can cause the state vectors to grow without bound. Saturating functions constrain the activations, helping to maintain stability.

14.3 Training RNNs: Backpropagation Through Time (BPTT)

Deeper explanation: Backpropagation through time with a loss at every step (p. 406)

Gradient Flow

Training requires a loss function. Depending on the task, a loss can be computed at every time step (e.g., sequence tagging) or only at the final time step (e.g., sequence classification).

During backpropagation, gradients flow backward through the computation graph:

- **Vertically:** Gradients flow down through the layers at each time step, similar to a standard feed-forward network.
- **Horizontally:** Gradients also flow backward in time, from $\vec{h}_{t+1}$ to $\vec{h}_t$, through the recurrent connection. This unrolling of the network through time is why the process is called Backpropagation Through Time.

This horizontal gradient path can become extremely long for sequences with many time steps.

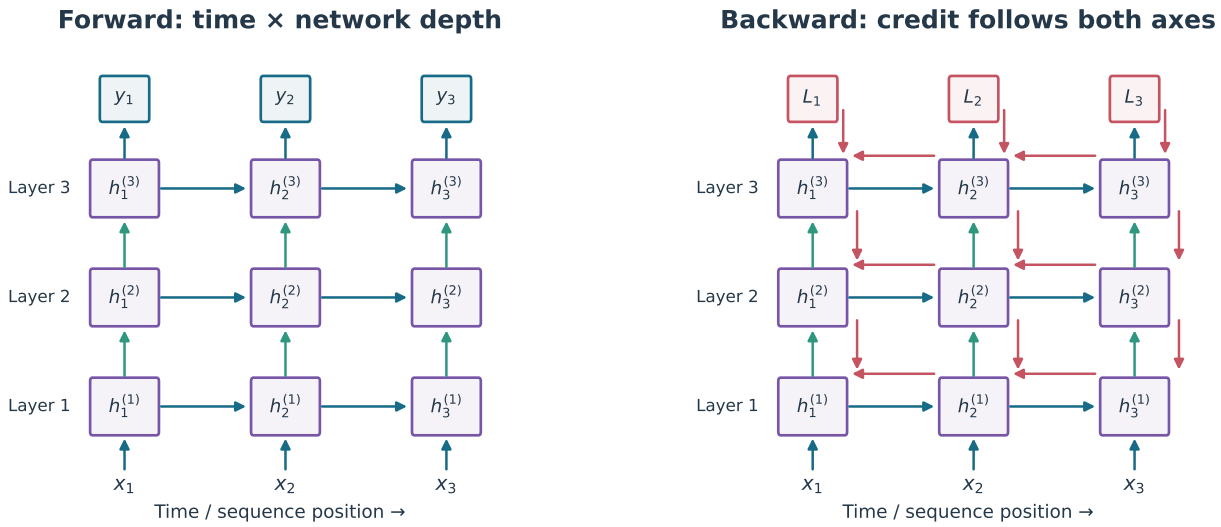


Figure 14.1: Time and depth in a stacked RNN. A structural redraw of the faculty grid. Forward activations move through time and depth; backward sensitivities follow both dependencies. Repeated time positions share parameters. Structural redraw; Anant Sahai, faculty handout 14, p. 2, 3.

BPTT as a triangular block system

Along a fixed forward trajectory, let $J_t = \partial h_t / \partial h_{t-1}$ and $K_t = \partial h_t / \partial x_t$. With fixed h_0, the local perturbation equations for three steps are

$$\begin{bmatrix} I & 0 & 0 \\ -J_2 & I & 0 \\ 0 & -J_3 & I \end{bmatrix} \begin{bmatrix} \delta h_1 \\ \delta h_2 \\ \delta h_3 \end{bmatrix} = \begin{bmatrix} K_1 & 0 & 0 \\ 0 & K_2 & 0 \\ 0 & 0 & K_3 \end{bmatrix} \begin{bmatrix} \delta x_1 \\ \delta x_2 \\ \delta x_3 \end{bmatrix}.$$

Forward substitution yields

$$\delta h_3 = J_3 J_2 K_1 \delta x_1 + J_3 K_2 \delta x_2 + K_3 \delta x_3.$$

For direct loss sensitivities r_t at each state, backward substitution in the transposed system gives $g_3 = r_3$, $g_2 = r_2 + J_3^\top g_3$, and $g_1 = r_1 + J_2^\top g_2$. Products of temporal Jacobians are

therefore forced by the block equations. For nonlinear RNNs these are local derivatives, not a globally linear representation of the forward network. A shared parameter contributes at every time step, and its gradient sums those contributions.

The Problem of Long-Range Dependencies

The long path for gradient propagation in time leads to a significant challenge:

- **Vanishing Gradients:** For the network to be stable, the recurrent dynamics must generally be contractive. This means that during backpropagation, gradients are repeatedly multiplied by the recurrent weight matrix. For a stable system, this causes gradients to shrink exponentially as they flow back in time. As a result, the influence of early time steps on the loss at a much later time step becomes negligible, making it difficult for the network to learn long-range dependencies.
- **Exploding Gradients:** Conversely, if the dynamics are expansive, gradients can grow exponentially, leading to unstable training. This is mitigated by saturating nonlinearities and techniques like gradient clipping.

Historically, architectures like **LSTMs (Long Short-Term Memory)** and **GRUs (Gated Recurrent Units)** were developed to address the vanishing gradient problem. They introduced complex gating mechanisms to better control the flow of information and gradients over long time scales. These were highly effective but also very delicate and complex.

A Modern View: Parallelism during Training

A critical issue with the traditional RNN structure is its inherent sequentiality, which makes it inefficient to train on modern parallel hardware (GPUs/TPUs).

- **Dependency Graph:** To compute the state at layer l and time t, one must have already computed the state at layer $l-1$ at time t AND the state at layer l at time $t-1$.
- **Limited Parallelism:** This creates a diagonal "wavefront" of computation. The maximum number of operations that can be performed in parallel at any given moment is limited by the number of layers in the network, not the length of the sequence. This is a form of "pipeline parallelism" and is often a poor utilization of hardware resources for long sequences.

The fundamental bottleneck to parallelization is the **non-linearity in the recurrent (horizontal) path**. If the recurrence were purely linear, various algebraic techniques could be used to parallelize the computation.

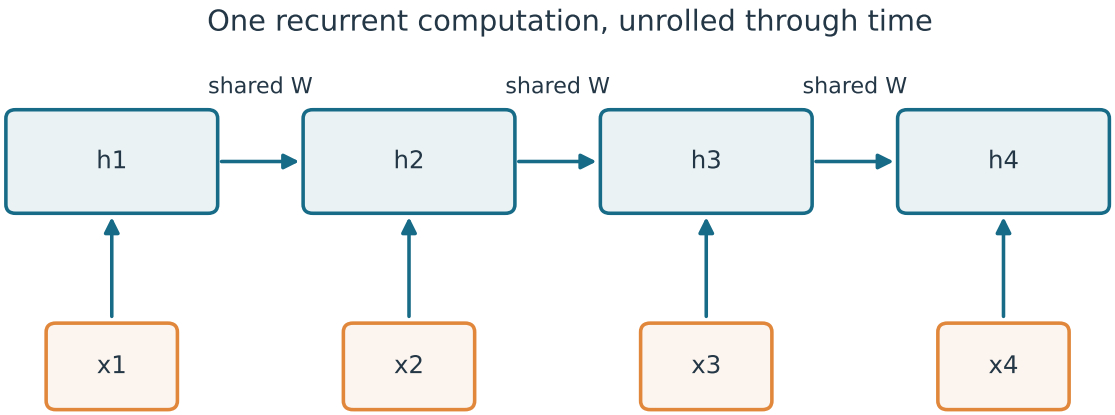


Figure 14.2: Backpropagation through time follows the unrolled state dependencies. The same recurrent matrix receives gradient contributions at every step.

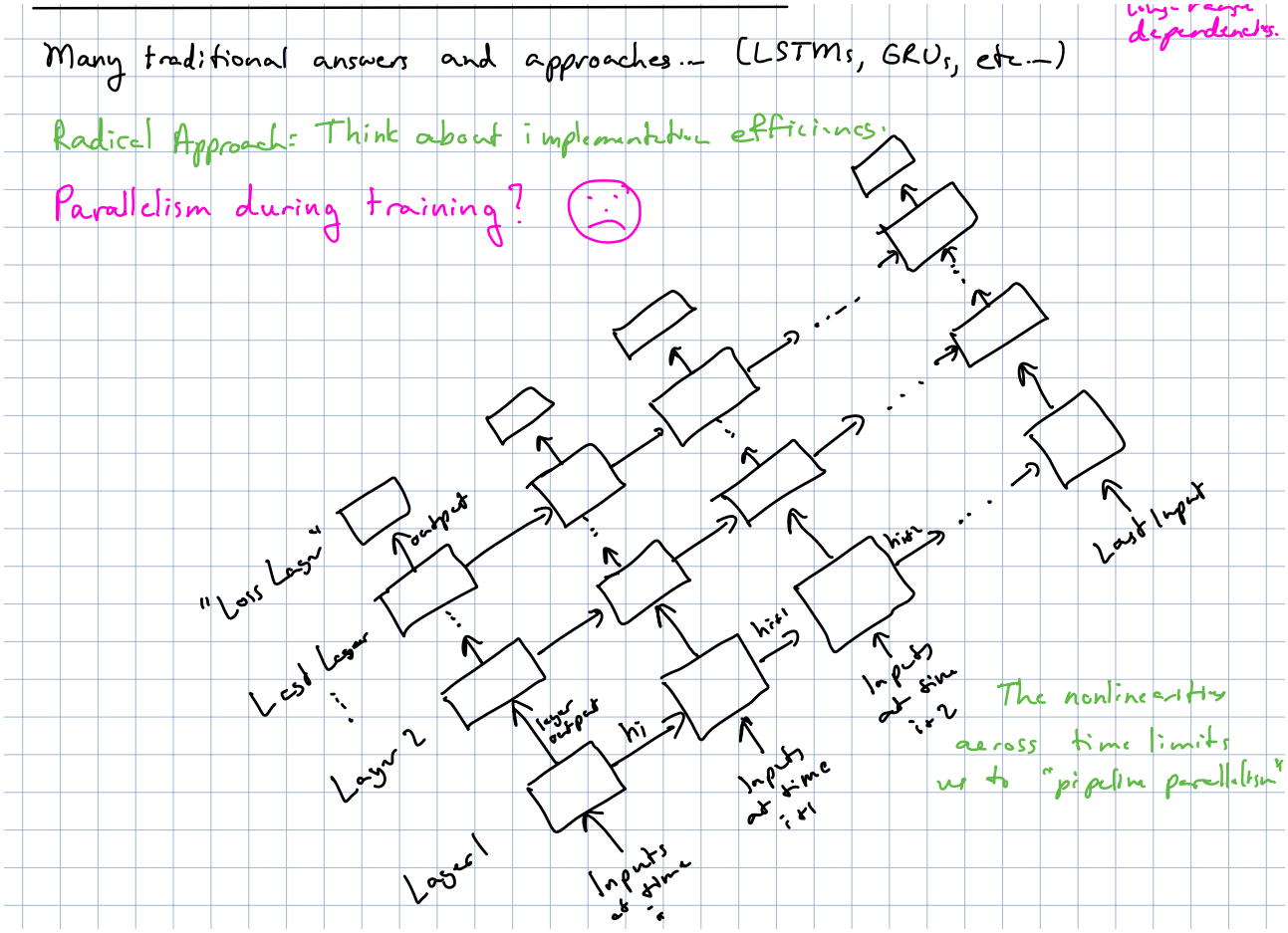


Figure 14.3: A tilted view of the time-depth dependency grid. Tilting the grid exposes the dependency frontier. Nodes can execute together only when their required predecessors are available; a different drawing does not remove the recurrent dependencies. Original annotated source excerpt; Anant Sahai, faculty handout 14, p. 3.

Worked derivation: unrolling the shared-parameter chain rule

Lecture basis: faculty handout 14, pp. 2–3. The steps below expand the lecture argument.

Step 1: isolate direct and recurrent derivatives. Let $h_t = f_\theta(h_{t-1}, x_t)$, $L = \sum_{t=1}^{T} \ell_t(h_t)$, and let h_0 be independent of θ. Define

$$A_t = \frac{\partial f_\theta(h_{t-1}, x_t)}{\partial h_{t-1}}, \quad B_t = \left.\frac{\partial f_\theta(h, x_t)}{\partial \theta}\right|_{h=h_{t-1}}, \quad a_t = \nabla_{h_t} \ell_t.$$

The derivative B_t holds the incoming state fixed; otherwise its recurrence would double-count earlier parameter effects.

Step 2: differentiate forward through the states. Put $D_t = \partial h_t / \partial \theta$. Then

$$\begin{aligned} D_0 &= 0, \\ D_t &= A_t D_{t-1} + B_t, \\ D_1 &= B_1, \quad D_2 = A_2 B_1 + B_2, \\ D_3 &= A_3 A_2 B_1 + A_3 B_2 + B_3, \\ D_t &= \sum_{k=1}^{t} A_t A_{t-1} \cdots A_{k+1} B_k. \end{aligned}$$

The product is the identity for $k = t$. Thus $\nabla_\theta L = \sum_t D_t^\top a_t$ sums every time a shared parameter affects any later loss.

Step 3: reorganize the same sum into backward recursion. Set $\delta_T = a_T$ and

$$\delta_t = a_t + A_{t+1}^\top \delta_{t+1}, \qquad \nabla_\theta L = \sum_{t=1}^{T} B_t^\top \delta_t.$$

This computes the same chain-rule terms without materializing every D_t. Direct parameter dependence in ℓ_t adds its own derivative. For a vanilla RNN, $A_t = \operatorname{diag}(\phi'(W h_{t-1} + U x_t + b))W$. Consequently

$$\|A_t \cdots A_{k+1}\|_2 \le \prod_{j=k+1}^{t} \|A_j\|_2.$$

Uniform factors below one force vanishing long-range sensitivity. Factors above one permit amplification but do not guarantee it in every direction. Truncated BPTT drops some of these paths and therefore generally changes the gradient.

14.4 Example: Learning a Kalman Filter

Deeper explanation: The Kalman filter as an exact recurrent inference program (p. 408)

To build intuition, we consider a purely linear system. The **Kalman Filter (KF)** is the optimal estimator for the state of a linear dynamical system with Gaussian noise.

Problem Setup

Consider an underlying true linear process with state $\vec{x}_t$, input $\vec{u}_t$, and output $\vec{y}_t$.

$$\vec{x}_{t+1} = A\vec{x}_t + B\vec{u}_t + \vec{w}_t \quad \text{(Process dynamics)} \tag{14.4}$$
$$\vec{y}_{t+1} = C\vec{x}_{t+1} + D\vec{u}_{t+1} + \vec{v}_{t+1} \quad \text{(Observation model)} \tag{14.5}$$

where $\vec{w}_t$ and $\vec{v}_t$ are i.i.d. zero-mean Gaussian noise. If all system matrices (A, B, C, D) and noise covariances are known, the steady-state Kalman Filter provides the optimal estimate $\hat{\vec{x}}_t$ of the true state $\vec{x}_t$. The KF itself is a linear recurrent system:

$$\hat{\vec{x}}_{t+1} = \tilde{A}\hat{\vec{x}}_t + \tilde{B}\vec{u}_{t+1} + F\vec{y}_{t+1} \tag{14.6}$$

This is a linear RNN.

Case 1: Supervised Learning

Suppose we do not know the system matrices, but we have a training dataset of input/output traces $\{(\vec{u}^{(j)}, \vec{y}^{(j)})_t\}$ along with the corresponding **ground truth hidden states** $\{\vec{x}_t^{(j)}\}$.

1. We can define a linear RNN with learnable parameter matrices $\tilde{A}, \tilde{B}, F$.
2. We can define a supervised loss function, such as the mean squared error between our estimate and the true state: $\mathcal{L} = \sum_{j,t} \|\hat{\vec{x}}_t^{(j)} - \vec{x}_t^{(j)}\|^2$.
3. We can train this RNN using gradient descent and BPTT to learn the optimal filtering matrices.

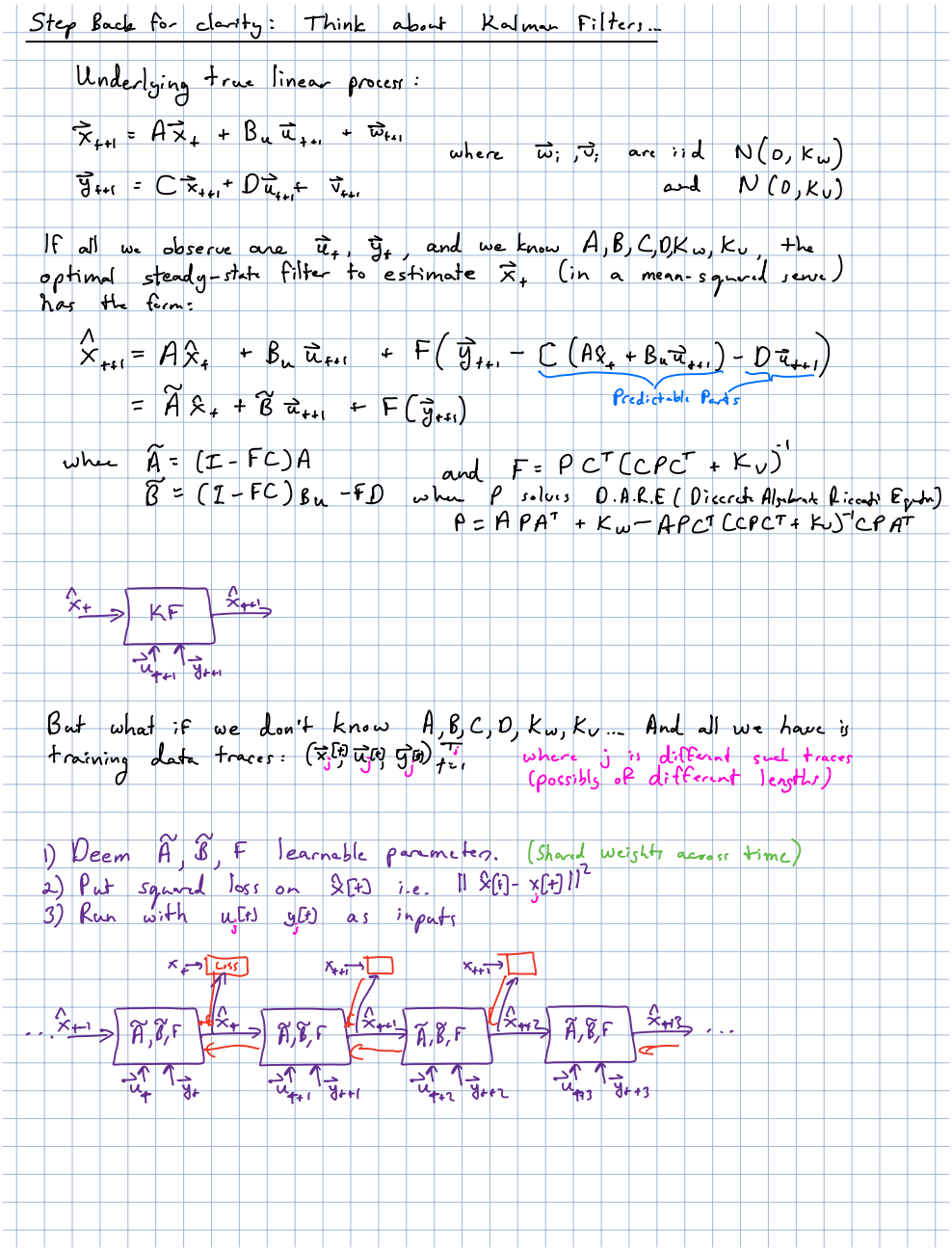


Figure 14.4: Learning a filter with a training scaffold. The physical process produces latent state and noisy observations. When the target is available during training, the loss can supervise the filter directly even if only observations will be available at deployment. Original annotated source excerpt; Anant Sahai, faculty handout 14, p. 4.

Case 2: The Need for Self-Supervision

Now, consider a more realistic scenario: what if we only have the input/output traces $\{(\vec{u}^{(j)}, \vec{y}^{(j)})_t\}$ and **no access to the true hidden states** $\vec{x}_t$?

Question: Can we learn the true Kalman Filter?

Answer: No. The problem is fundamentally underdetermined. The internal state representation $\vec{x}$ is not uniquely identifiable from input-output data alone.

- **Unobservability through Change of Coordinates:** Any invertible linear transformation (change of basis) applied to the state space, $\vec{z} = T\vec{x}$, would result in a different set of system matrices (A', B', C') that produce the *exact same* input-output behavior.
- Since we cannot distinguish between these different coordinate systems from the data, we cannot learn the "true" state $\vec{x}$.
- However, some properties of the system, like the eigenvalues of the dynamics matrix A, are invariant to a change of basis. This suggests we can learn the *essence* or *structure* of the dynamics, but not the specific coordinate system.

This motivates the need for a different learning paradigm where we do not rely on external labels.

Worked derivation: prediction plus innovation as an RNN

Lecture basis: faculty handout 14, pp. 4–5. The steps below expand the lecture argument.

Step 1: write the two stages with one time convention. Suppose the predicted state is $\bar{h}_{t+1} = Ah_t + Bu_{t+1}$, the observation predictor is $C\bar{h}_{t+1} + Du_{t+1}$, and the correction gain is F. Then

$$h_{t+1} = \bar{h}_{t+1} + F(y_{t+1} - C\bar{h}_{t+1} - Du_{t+1}).$$

The innovation is the observed output minus its prediction, not the output alone.

Step 2: collect the three input paths.

$$\begin{aligned} h_{t+1} &= (I - FC)\bar{h}_{t+1} + Fy_{t+1} - FDu_{t+1} \\ &= (I - FC)(Ah_t + Bu_{t+1}) + Fy_{t+1} - FDu_{t+1} \\ &= (I - FC)Ah_t + \big[(I - FC)B - FD\big]u_{t+1} + Fy_{t+1}. \end{aligned}$$

Equivalently,

$$h_{t+1} = \begin{bmatrix} (I - FC)A & (I - FC)B - FD & F \end{bmatrix} \begin{bmatrix} h_t \\ u_{t+1} \\ y_{t+1} \end{bmatrix}.$$

This is a linear recurrent layer driven by controls and observations. A constant F corresponds to a fixed-gain filter; a general Kalman filter may have time-dependent gains. Learning a recurrence from prediction loss does not by itself establish that its hidden coordinates equal the physical latent state, since invertible state-coordinate changes preserve observable predictions.

14.5 Self-Supervision

Deeper explanation: Contrastive prediction and the role of negative examples (p. 410)

General Principle

Self-supervision is a learning paradigm that bridges supervised and unsupervised learning. The core idea is:

> I need labels to train, but I don't have them. So, I will **make my own labels from the data itself**.

We design a "pretext task" where one part of the data is used to predict another part of the same data. This allows us to use the machinery of supervised learning (loss functions, backpropagation) without requiring external annotations.

Self-Supervised Learning for the Filter Problem

Since we cannot predict the unobserved state $\vec{x}_t$, we must choose a target that is available in our data. For a causal system, a natural choice is to predict the future.

- **Pretext Task:** Predict the next observation, $\vec{y}_{t+1}$, using the history of inputs and outputs up to time t.
- **Model:** We use our linear RNN to produce a hidden state $\hat{\vec{x}}_t$, which summarizes the past.
- **Scaffolding:** Since the state $\hat{\vec{x}}_t$ may have a different dimension than the observation $\vec{y}_{t+1}$, we add an extra component, a "prediction head" (e.g., a linear layer), to map the state to a prediction of the output. This is temporary "scaffolding" used only during training.
- **Loss:** The loss is the error between our prediction and the actual next observation from the data: $\mathcal{L} = \sum_{j,t} \|\text{predicted}(\hat{\vec{x}}_t^{(j)}, \vec{u}_{t+1}^{(j)}) - \vec{y}_{t+1}^{(j)}\|^2$.

By training on this task, the network is forced to learn an internal representation ($\hat{\vec{x}}_t$) that captures the essential dynamics of the system, sufficient for predicting future outputs. This learned representation captures the system's structure up to the unobservable invariances (like the change of coordinates).

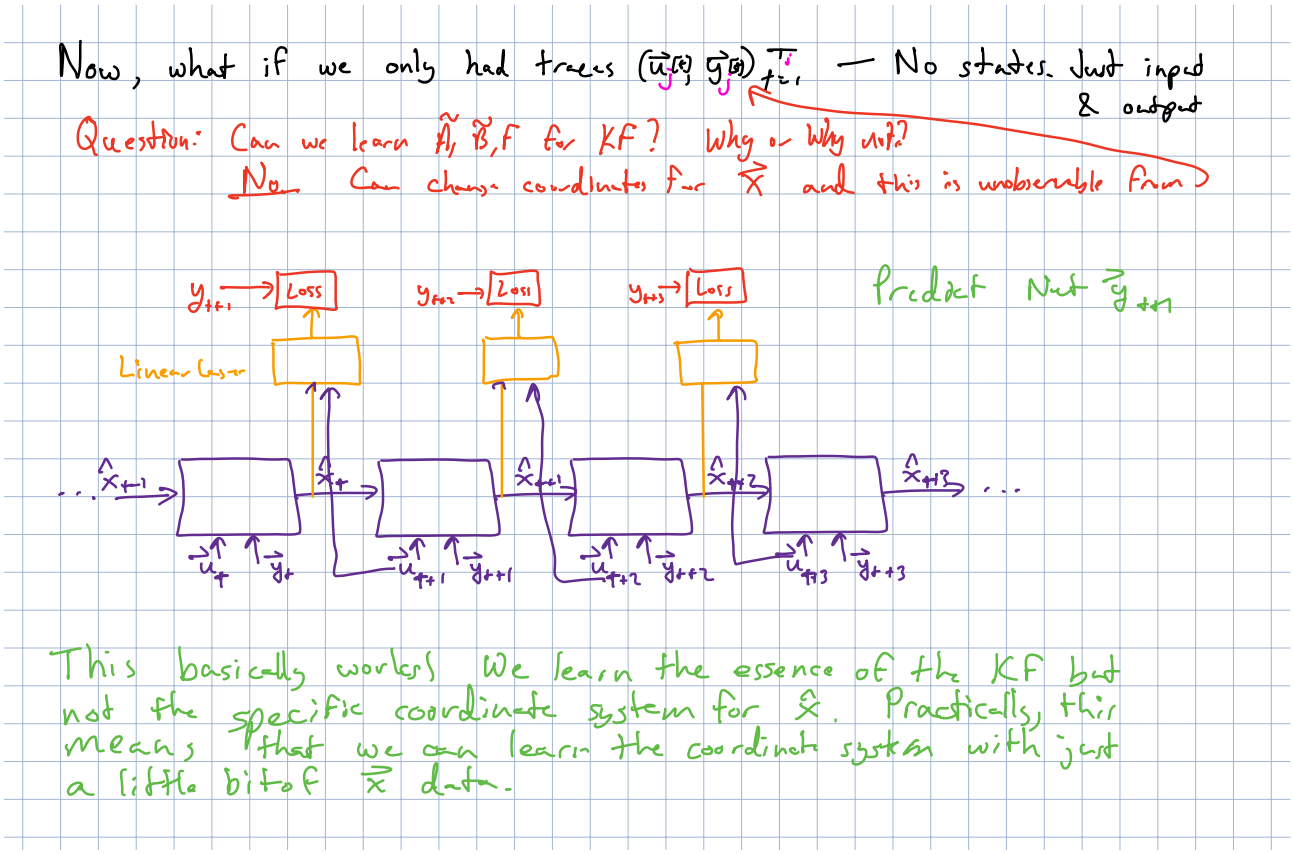


Figure 14.5: Predicting observations supplies self-supervision. The shifted observations create prediction targets from the sequence itself. Compare the training scaffold with the preceding supervised diagram to see what target information has changed. Original annotated source excerpt; Anant Sahai, faculty handout 14, p. 5.

Key Lessons from the Example

1. We can learn a useful **partial pattern** even if the full system is not identifiable.
2. We may need **scaffolding** (e.g., prediction heads) to formulate a trainable loss. These components can be discarded after the primary representation is learned.
3. For causal sequences, **next-step prediction** is a generic and powerful self-supervision strategy.

14.6 Connection to Unsupervised Learning

Deeper explanation: Nonunique latent coordinates and different reconstruction targets (p. 404)

Self-supervision is the modern, deep-learning approach to the classical machine learning problem of **unsupervised learning**. The goal is to discover intrinsic structure in unlabeled data. Classical approaches include:

1. **Dimensionality Reduction** (e.g., PCA): Assumes data lies on a lower-dimensional manifold.
2. **Clustering** (e.g., K-Means): Assumes data is organized into distinct groups or blobs.

Thinking about PCA as Self-Supervision

Principal Component Analysis (PCA) is typically presented as an algorithm (compute covariance, find eigenvectors). Can we reframe it in the language of self-supervision?

- The **Eckart-Young-Mirsky theorem** states that the best rank-k approximation of a matrix X (in terms of Frobenius norm) is given by its truncated Singular Value Decomposition (SVD).

$$\hat{X} = \arg \min_{\text{rank}(Z) \leq k} \|X - Z\|_F^2 \tag{14.7}$$

- This provides a reconstruction loss! We can interpret PCA as trying to find a low-dimensional representation from which the original data can be best reconstructed.
- This framing naturally leads to the concept of an **autoencoder**, a neural network trained to reconstruct its own input after passing it through an information bottleneck. This provides a way to learn dimensionality reduction using gradient-based optimization.

Lecture 15

Self-Supervision and State-Space Models

Course sources: lecture_15_sahai; lecture_16; 2025-10-21 transcript. Lecture numbering follows the original reader; linked faculty handouts identify the overlapping source coverage.

15.1 Self-Supervision

Deeper explanation: Contrastive prediction and the role of negative examples (p. 410)

General Principles

The core idea of self-supervision addresses a fundamental problem in machine learning: training algorithms require labels, but labeled data is often scarce.

> "I need labels to train, I don't have labels, so I make my own labels from data."

This approach allows us to learn meaningful patterns and representations directly from unlabeled data. From the previous lecture's Kalman filtering example, where we tried to learn an underlying state without ground truth labels, we learned several lessons:

- **Learning Partial Patterns:** It is possible to learn useful, albeit partial, patterns. In the Kalman filter case, we could learn the system dynamics up to a change of coordinates.
- **Scaffolding:** We often need to create temporary or "scaffolding" parts of a neural network (like an artificial output layer) to define a loss function for training. These parts may be discarded after training.
- **Next-Thing Prediction:** In causal sequence modeling, a generic and powerful self-supervision technique is to predict the next element in a sequence.

Connecting to Classical Unsupervised Learning

To better understand self-supervision, we can connect it to classical unsupervised learning methods. The two main approaches in classical ML are:

1. **Dimensionality Reduction** (e.g., Principal Component Analysis - PCA)
2. **Clustering** (e.g., K-Means)

Recalling PCA Given a dataset of unlabeled data points $\{\vec{x}_i\}_{i=1}^N$ where $\vec{x}_i \in \mathbb{R}^d$, PCA aims to find a lower-dimensional representation that captures the most variance in the data.

Classic Recipe for PCA:

1. Construct a data matrix $X \in \mathbb{R}^{N \times d}$, where each row is a data point $\vec{x}_i^T$.
2. Compute the Singular Value Decomposition (SVD) of X: $X = U\Sigma V^T$.
3. Keep the top k right singular vectors, $\vec{v}_1, \ldots, \vec{v}_k$, which correspond to the largest singular values. These vectors form an orthonormal basis for the k-dimensional subspace of greatest variance.
4. For a new data point $\vec{x}$, its k-dimensional feature representation is obtained by projecting it onto this basis.

The Deep Learning Perspective: From a modern deep learning viewpoint, this classical recipe feels very different from what we are used to. Specifically, it has:

- No loss function
- No labels
- No gradients
- No optimizer
- No mini-batches

This motivates us to reinterpret PCA in a way that aligns with the deep learning framework.

Reinterpreting PCA: The Autoencoder

A Loss-Based View via Eckart-Young-Mirsky Theorem The Eckart-Young-Mirsky theorem provides a link between the SVD and a loss minimization problem. The theorem states that the best rank-k approximation $\hat{X}$ to a matrix X (in the sense of minimizing the Frobenius norm of the difference) is obtained by truncating the SVD of X at the k-th term.

$$\min_{\text{rank}(\hat{X}) \leq k} \|X - \hat{X}\|_F^2$$

The solution is $\hat{X} = \sum_{i=1}^k \sigma_i \vec{u}_i \vec{v}_i^T$. This gives us something that looks like a loss function: the mean squared error between the original data matrix and its low-rank reconstruction.

Formulating a Neural Network We can now translate this loss minimization problem into a neural network training problem. The goal is to learn a mapping that reconstructs the input vector $\vec{x}$ after passing it through a low-dimensional bottleneck.

- **Input and Label:** The input is a data vector $\vec{x} \in \mathbb{R}^d$. For self-supervision, the label is the input itself, $\vec{x}$.
- **Loss Function:** The reconstruction loss is the mean squared error: $\|\vec{x} - \hat{\vec{x}}\|_2^2$, where $\hat{\vec{x}}$ is the reconstructed vector.
- **Architecture:** To enforce the rank-k constraint, we design an architecture with a bottleneck. We can use two linear layers:

  1. An **encoder**, $W_1 \in \mathbb{R}^{k \times d}$, maps the d-dimensional input to a k-dimensional hidden representation (where $k < d$).
  2. A **decoder**, $W_2 \in \mathbb{R}^{d \times k}$, maps the k-dimensional representation back to the original d-dimensional space.

  The reconstructed vector is $\hat{\vec{x}} = W_2 W_1 \vec{x}$. The product $W_2 W_1$ is inherently at most rank k.

This network can be trained over the entire dataset using standard deep learning techniques like mini-batch gradient descent. The learned encoder W_1 then serves the same purpose as the projection in PCA.

The Autoencoder This neural network architecture is called an **autoencoder**. Its core ingredients are:

- **Labels:** The inputs themselves ($\vec{x}_i$).
- **Architecture:** An encoder, a bottleneck (hidden layer), and a decoder.
- **Goal:** Learn an encoding that captures the most important information needed for reconstruction.

From a traditional perspective, the primary goal is to obtain the encoder for dimensionality reduction on new data. The decoder is merely **scaffolding** needed to define the loss and enable training.

Worked derivation: PCA as a constrained linear autoencoder

Lecture basis: faculty handout 15_sahai, pp. 1–2. The steps below expand the lecture argument.

Step 1: impose an orthogonal bottleneck. Let centered data be columns of $X \in \mathbb{R}^{d\times n}$. Choose $U \in \mathbb{R}^{d\times k}$ with $U^\top U = I_k$. Encoding is $U^\top X$ and reconstruction is $UU^\top X$. The matrix $P = UU^\top$ satisfies $P^2 = P = P^\top$.

Step 2: convert reconstruction error into captured variance.

$$\begin{aligned}\|X - PX\|_F^2 &= \operatorname{tr}\left[X^\top (I-P)^\top (I-P)X\right] \\ &= \operatorname{tr}(X^\top X) - \operatorname{tr}(X^\top PX) \\ &= \|X\|_F^2 - \operatorname{tr}(U^\top XX^\top U).\end{aligned}$$

The first term is fixed, so minimizing error means maximizing the second term.

Step 3: identify the optimal subspace. Let $XX^\top = Q\,\mathrm{diag}(\lambda_1, \dots, \lambda_d)Q^\top$, with descending eigenvalues. Put $a_i = \|U^\top q_i\|^2$. Orthogonal projection gives $0 \le a_i \le 1$ and $\sum_i a_i = k$. Hence

$$\operatorname{tr}(U^\top XX^\top U) = \sum_i \lambda_i a_i \le \sum_{i=1}^{k} \lambda_i.$$

The inequality follows by allocating total mass k to the largest eigenvalues, with at most one unit per coordinate. Equality is attained by $U = [q_1 \ \cdots \ q_k]$. The minimum error is $\sum_{i>k} \lambda_i = \sum_{i>k} \sigma_i(X)^2$.

Step 4: distinguish subspace recovery from coordinate recovery. An unconstrained linear encoder–decoder can attain this reconstruction, but its latent coordinates need not equal PCA coordinates. For invertible $R \in \mathbb{R}^{k\times k}$,

$$D = UR^{-1}, \qquad E = RU^\top \quad \Longrightarrow \quad DE = UU^\top.$$

Thus reconstruction determines a useful subspace, not a unique latent basis. Nonlinear autoencoders need not solve PCA at all.

Variations of Autoencoders

The idea of forcing information through a bottleneck to learn essential features can be generalized. We can encode the desired regularities in our representation using architecture, auxiliary losses, or data augmentation.

Classic (Vanilla) Autoencoder This is the architecture described above. The regularity (low-dimensionality) is enforced by the **architectural bottleneck**, where the hidden layer has fewer neurons than the input/output layers. The encoder and decoder can be arbitrary neural networks (e.g., CNNs for images), not just single linear layers.

Sparse Autoencoder Instead of an architectural bottleneck, we can impose a bottleneck using an **auxiliary loss**.

- The hidden layer can be larger than the input layer.
- We add a sparsity-seeking penalty to the loss function, such as an L1-norm penalty on the hidden activations.
- The total loss becomes: $L = \|\vec{x} - \hat{\vec{x}}\|_2^2 + \lambda\|\vec{h}\|_1$, where $\vec{h}$ is the hidden representation.
- The L1 penalty encourages most hidden units to be zero, forcing the network to use only a small subset of features to represent any given input. The information is in both the values of the non-zero activations and *which* activations are non-zero.

This encourages a representation where data is described by a combination of a few "basis" features from a large dictionary.

Denoising Autoencoder Here, the regularity is enforced through **data augmentation**.

- The input to the encoder is a corrupted version of the original data, $\tilde{\vec{x}}$ (e.g., $\vec{x}$ with added Gaussian noise).
- The network is trained to reconstruct the *original, clean* data $\vec{x}$ from the corrupted input $\tilde{\vec{x}}$.
- To succeed, the encoder must learn to capture the robust, essential features of the data manifold and ignore the noise.
- This prevents the network from learning a trivial identity mapping, even without an architectural bottleneck.

Masked Autoencoder This is another variant based on data augmentation, particularly popular in language (e.g., BERT) and vision.

- A random portion of the input is masked or removed (e.g., setting patches of an image to a gray value).
- The autoencoder's task is to reconstruct the full, original input from the partially masked version.
- This forces the model to understand the context and underlying structure of the data to successfully "fill in the blanks."

- The loss can be computed over the entire reconstructed image or, in some variants, only on the masked regions.

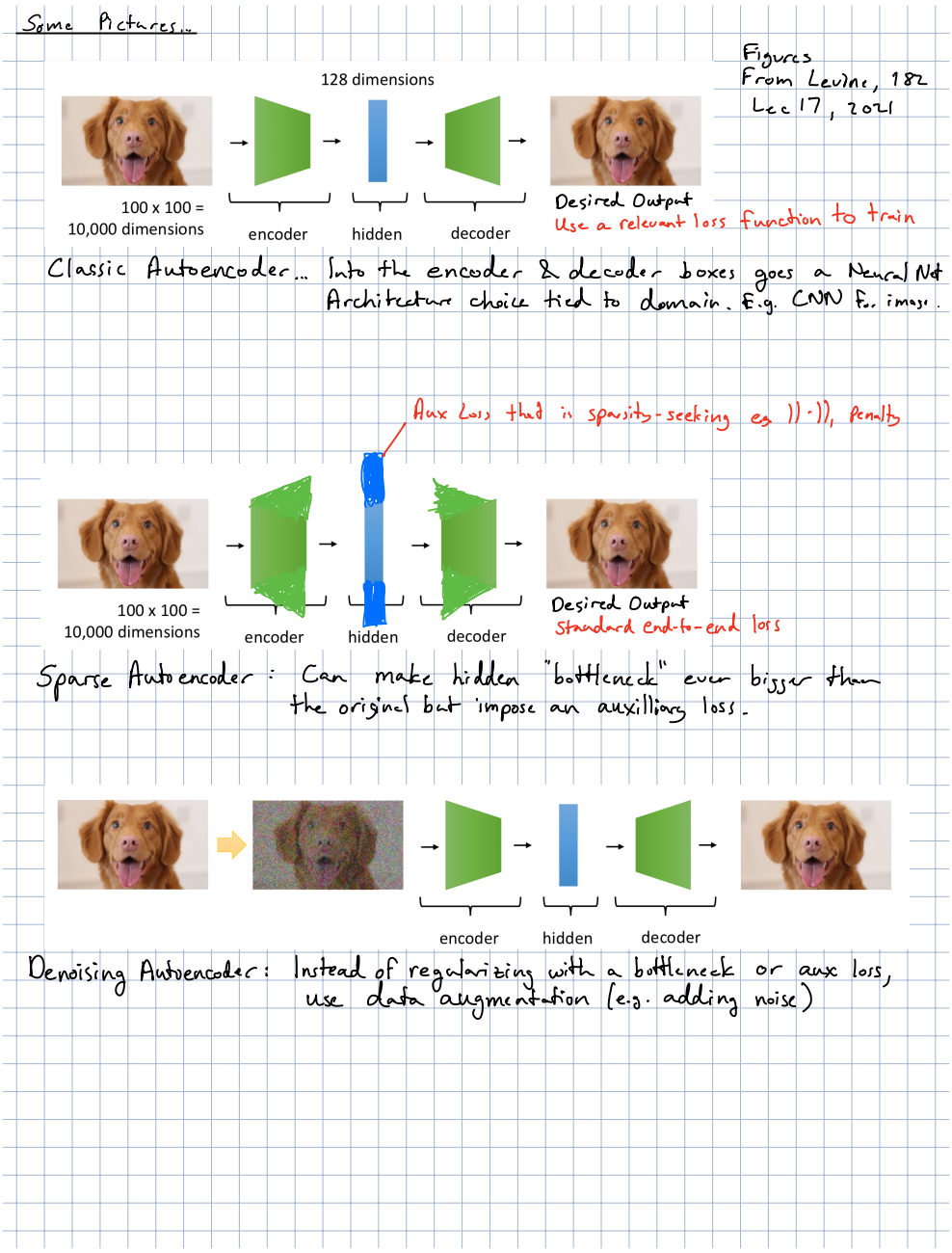


Figure 15.1: Sparse and denoising autoencoder examples. A bottleneck, a sparsity constraint, and input corruption impose different learning pressures. In denoising, the corrupted example is the input and the clean example is the target. Published image examples are retained from the handout. Original annotated source excerpt; Anant Sahai, faculty handout 15 sahai, p. 3.

Contrastive Self-Supervision

Just as autoencoders connect to dimensionality reduction, we can find a self-supervision paradigm that connects to the spirit of **clustering**. This is called **contrastive self-supervision**.

Core Idea:

- An example and its own data augmentations should belong to the same cluster. In an embedding space, this means they should be close to each other. These are called **positive pairs**.
- Fundamentally different examples from the dataset should belong to different clusters. In an embedding space, this means they should be far from each other. These are called **negative pairs**.

Basic Structure:

1. Take a data point $\vec{x}_i$ (the "anchor").
2. Create a positive pair by applying a data augmentation to it: $\text{aug}(\vec{x}_i)$.
3. Select a different data point $\vec{x}_j$ as a negative pair.
4. Pass all three through an encoder to get embeddings: e_i, e_i', and e_j.
5. Define a loss function that rewards closeness for the positive pair (pulls e_i and e_i' together) and penalizes closeness for the negative pair (pushes e_i and e_j apart).

This method learns an embedding space where similar items are clustered together, without needing a decoder. The reliance on negative samples can sometimes be problematic (e.g., pushing apart two different images of the same class), leading to advanced methods that work without explicit negative examples.

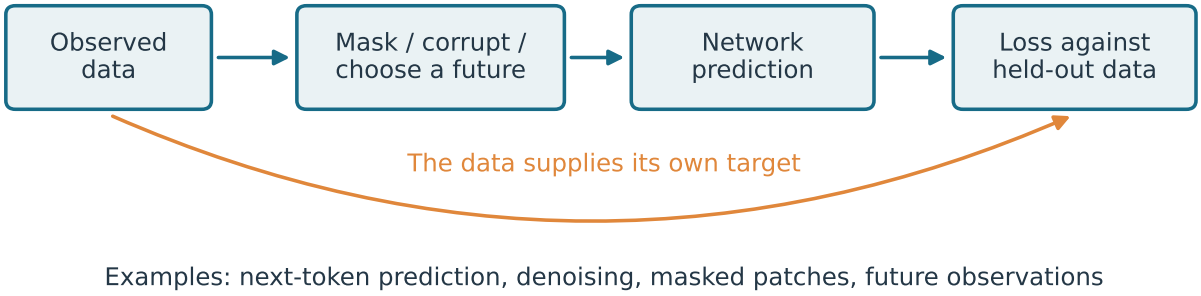


Figure 15.2: Reconstruction, denoising, masking, and contrastive learning create different self-supervised targets.

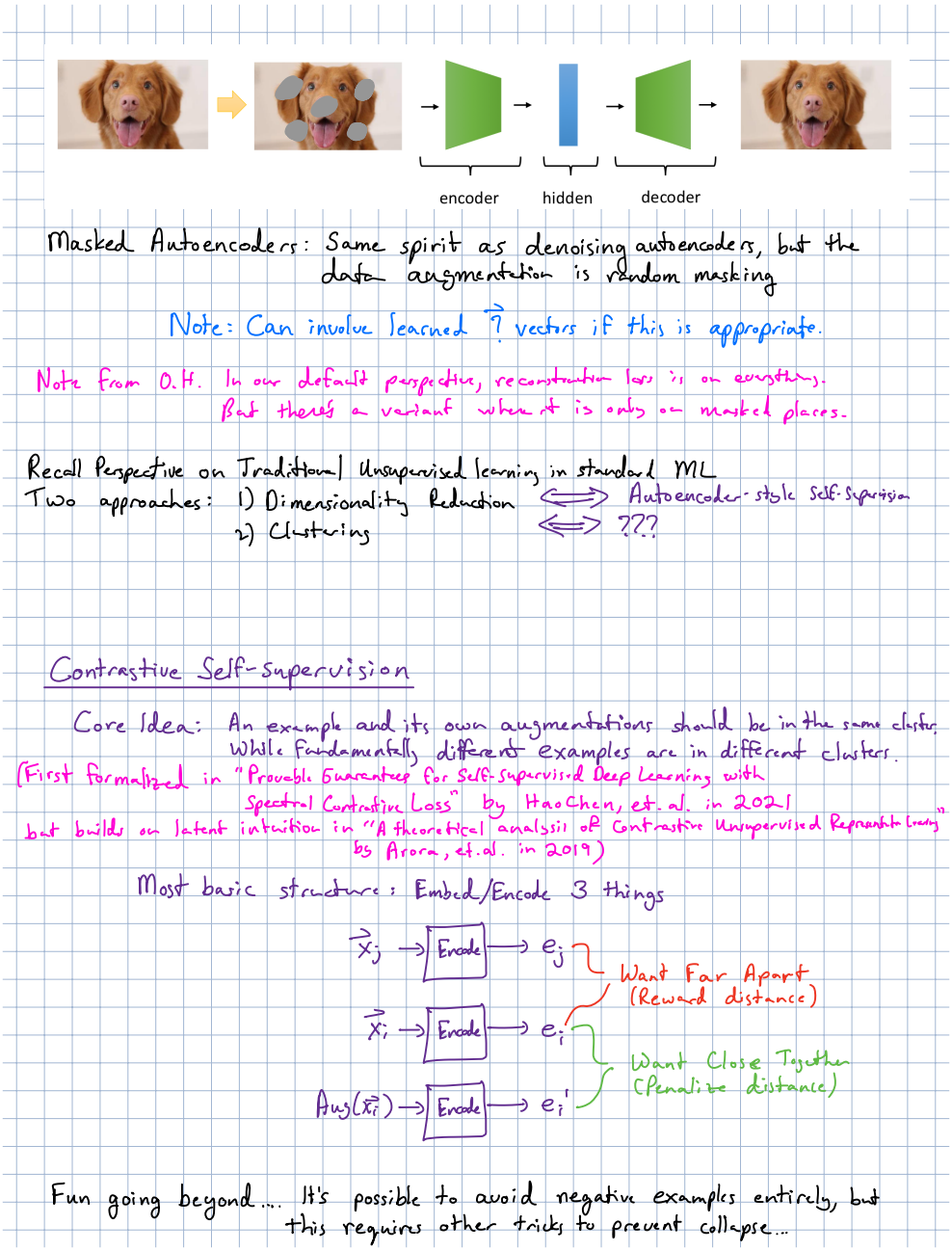


Figure 15.3: Masked reconstruction and contrastive views. Masked reconstruction predicts missing content; contrastive learning compares representations of views. The lecture's paired branches make this distinction concrete. Published panels and faculty annotations are preserved. Original annotated source excerpt; Anant Sahai, faculty handout 15 sahai, p. 4.

15.2 State-Space Models (SSMs)

Deeper explanation: The full convolution derivation, including initial state (p. 412)

Motivation: Problems with Traditional RNNs

Recurrent Neural Networks (RNNs) are designed for sequential data, but they suffer from significant drawbacks:

1. **Difficult to Parallelize Training:** The recurrence relation $h_t = f(h_{t-1}, x_t)$ is inherently sequential. To compute the hidden state at time t, one must first compute it at $t-1$. This prevents parallelization across the time dimension during training.
2. **Difficulty Learning Long-Range Dependencies:** The repeated application of non-linearities and matrix multiplications during backpropagation can lead to vanishing or exploding gradients, making it hard to capture dependencies over long sequences.

The Key Idea: Eliminate Horizontal Non-Linearity

The core innovation of State-Space Models (SSMs) is to address these issues by making the recurrence **linear**. We eliminate the "horizontal" non-linearity that operates along the time axis.

A traditional RNN update looks like: $\vec{h}_{t+1} = \sigma(A\vec{h}_t + B\vec{x}_t + \vec{b})$. The new SSM approach simplifies this to a linear system:

$$\vec{h}_{t+1} = A\vec{h}_t + B\vec{x}_{t+1}$$
$$\vec{y}_t = C\vec{h}_t + D\vec{x}_t$$

Here, A, B, C, D are learned weight matrices that do not depend on the input. This is a Linear Time-Invariant (LTI) system, a well-understood concept from control theory and signal processing. The vector $\vec{h}_t$ is the **state**, which acts as the model's memory, summarizing the history of the sequence up to time t.

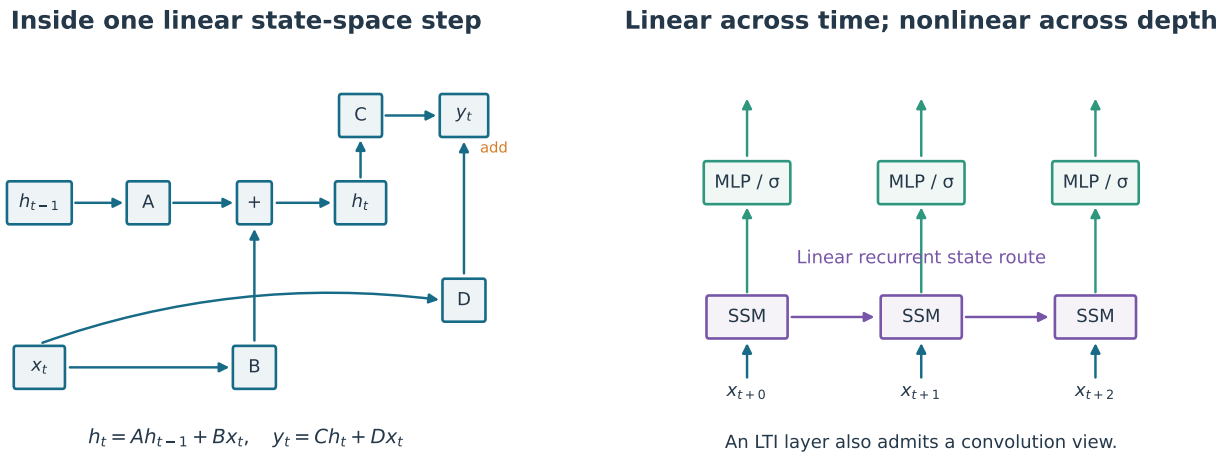


Figure 15.4: Horizontal linearity and vertical nonlinearity. A structural redraw of the lecture's separation of roles. The recurrent state route is linear for an LTI layer; a nonlinear map can still act between layers. The direct input-to-output path uses D on the original input. Structural redraw; Gireeja Ranade, faculty handout 16, p. 1, 2, 4.

The Convolution View for Parallel Training

By removing the non-linearity, we can unroll the recurrence to express the output $\vec{y}_k$ purely in terms of the inputs, without intermediate hidden states.

$$\begin{aligned}
\vec{h}_0 &= B\vec{x}_0 \\
\vec{h}_1 &= A\vec{h}_0 + B\vec{x}_1 = AB\vec{x}_0 + B\vec{x}_1 \\
\vec{h}_2 &= A\vec{h}_1 + B\vec{x}_2 = A^2B\vec{x}_0 + AB\vec{x}_1 + B\vec{x}_2 \\
&\vdots \\
\vec{h}_k &= \sum_{i=0}^{k} A^{k-i}B\vec{x}_i
\end{aligned}$$

Substituting this into the output equation gives:

$$\vec{y}_k = C\vec{h}_k + D\vec{x}_k = C\left(\sum_{i=0}^{k} A^{k-i}B\vec{x}_i\right) + D\vec{x}_k$$

This expression is a **convolution**. The output at each time step is a linear combination of all past inputs. The convolution kernel (or filter) is defined by the matrices $\{CA^kB\}_{k=0,1,\ldots}$.

Since this is a convolution, the entire output sequence can be computed in parallel during training. We no longer need to compute hidden states sequentially.

The finite-sequence SSM is a block Toeplitz map

Use $h_t = Ah_{t-1} + Bx_t$ and $y_t = Ch_t + Dx_t$ with $h_0 = 0$, and start the inputs at $t = 1$. For three steps,

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} CB + D & 0 & 0 \\ CAB & CB + D & 0 \\ CA^2B & CAB & CB + D \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}.$$

If inputs have width p and outputs width q, every entry shown is a $q \times p$ block, and the whole matrix is $3q \times 3p$. The direct path D occurs only on the main diagonal. For a nonzero initial state, add $[CAh_0;\ CA^2h_0;\ CA^3h_0]$. Repeated blocks along diagonals express time invariance; zeros above the diagonal express causality. These are different properties: selective input-dependent dynamics remain causal but generally lose the repeated-diagonal structure.

Worked derivation: from recurrence to a block convolution

Lecture basis: faculty handout 16, pp. 2–3, 7–8. The steps below expand the lecture argument.

Step 1: expand the recurrence before introducing a kernel. For $h_t = Ah_{t-1} + Bx_t$ and $y_t = Ch_t + Dx_t$,

$$\begin{aligned} h_1 &= Ah_0 + Bx_1, \\ h_2 &= A^2h_0 + ABx_1 + Bx_2, \\ h_3 &= A^3h_0 + A^2Bx_1 + ABx_2 + Bx_3, \\ h_t &= A^th_0 + \sum_{j=1}^{t} A^{t-j}Bx_j. \end{aligned}$$

Substituting into the readout gives

$$y_t = CA^th_0 + \sum_{j=1}^{t} K_{t-j}x_j, \quad K_0 = CB + D, \quad K_\ell = CA^\ell B \ (\ell > 0).$$

The initial-state term must be included unless $h_0 = 0$. Constant A, B, C, D make the kernel depend only on lag; input-dependent parameters destroy this fixed convolution representation.

Step 2: work through the handout's two-state example. Take $h_0 = 0$ and

$$A = \begin{bmatrix} 1 & 0 \\ 0 & \alpha \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}, \quad C = \begin{bmatrix} 1/2 & 1/2 \\ 0 & 1 \end{bmatrix}, \quad D = 0.$$

Matrix multiplication, with every coefficient retained, gives

$$\begin{aligned} A^\ell B &= \begin{bmatrix} 1 & 1 \\ 0 & \alpha^\ell \end{bmatrix}, \\ K_\ell = CA^\ell B &= \begin{bmatrix} 1/2 & (1+\alpha^\ell)/2 \\ 0 & \alpha^\ell \end{bmatrix}, \\ y_{t,1} &= \tfrac{1}{2}\sum_{j=1}^{t} x_{j,1} + \tfrac{1}{2}\sum_{j=1}^{t}(1+\alpha^{t-j})x_{j,2}, \\ y_{t,2} &= \sum_{j=1}^{t} \alpha^{t-j}x_{j,2}. \end{aligned}$$

The first state is an accumulator; the second has decaying memory when $|\alpha| < 1$. The factor $1/2$ multiplies both contributions to the first output because of the stated C.

Step 3: identify the computational opportunity. A direct causal sum uses $1 + 2 + \cdots + T = T(T+1)/2$ kernel applications. FFT multiplication can compute convolution in $O(T \log T)$ per scalar channel pair, but sequences of lengths T, T must be zero-padded to at least $2T - 1$ to obtain linear rather than circular convolution. Recurrent inference instead retains h_t and updates one time step at a time.

Improving Computational Efficiency

From $O(T^2)$ to $O(T \log T)$ with FFT A naive implementation of this convolution for a sequence of length T requires $O(T^2)$ total computation. We can significantly speed this up using the **Fast Fourier Transform (FFT)**.

- The convolution theorem states that convolution in the time domain is equivalent to element-wise multiplication in the frequency domain.
- The FFT algorithm can compute the Discrete Fourier Transform of a sequence of length T in $O(T \log T)$ time.
- **Algorithm:**
  1. Compute the FFT of the input sequence $\vec{x}$.
  2. Compute the FFT of the convolution filter (derived from A, B, C).
  3. Multiply the results element-wise in the frequency domain.
  4. Compute the Inverse FFT to get the final output sequence $\vec{y}$.
- This reduces the overall training complexity to $O(T \log T)$, making SSMs highly efficient for long sequences.
- A technical challenge is that the inputs $\vec{x}_t$ are vectors, not scalars. The FFT machinery must be carefully adapted to handle these vector-valued sequences, typically by breaking the computation down component-wise.

Speeding up Kernel Computation with a Diagonal State Matrix Computing the convolution kernel itself involves calculating matrix powers A^k, which can be computationally intensive and adds depth to the computation graph. To simplify this, we can impose structure on the state matrix A.

- We restrict A to be a **diagonal matrix**.
- If $A = \text{diag}(\lambda_1, \dots, \lambda_n)$, then $A^k = \text{diag}(\lambda_1^k, \dots, \lambda_n^k)$, which is trivial to compute.
- This might seem restrictive, but many matrices are diagonalizable. By learning the parameters, the model can effectively learn a basis in which the system dynamics are diagonal, capturing a wide range of behaviors.

Expressive Power and Inference

Expressive Power: A purely linear model lacks the expressive power of non-linear RNNs. To address this, SSM architectures re-introduce non-linearities in the "vertical" path (i.e., within a single time step), while keeping the "horizontal" recurrence linear. This is typically done by placing an MLP or other non-linear activation functions before or after the SSM

layer. This hybrid approach combines the expressive power of non-linearities with the training efficiency of linear recurrences.

Inference: While the parallel convolution view is ideal for training, the recurrent view is highly efficient for inference, especially in auto-regressive tasks.

- To generate the next output, we only need the current state $\vec{h}_t$ and the current input $\vec{x}_{t+1}$.
- We update the state using the simple recurrent formula: $\vec{h}_{t+1} = A\vec{h}_t + B\vec{x}_{t+1}$.
- The entire history of the sequence is compressed into the state vector $\vec{h}_t$. This makes inference very fast (constant time per step, ignoring matrix-vector product costs) and memory-efficient.

This dual formulation gives SSMs the best of both worlds: fast, parallelizable training and efficient, sequential inference. These properties have made them a key component in modern architectures like Mamba, DeltaNet, and large language models such as Qwen.

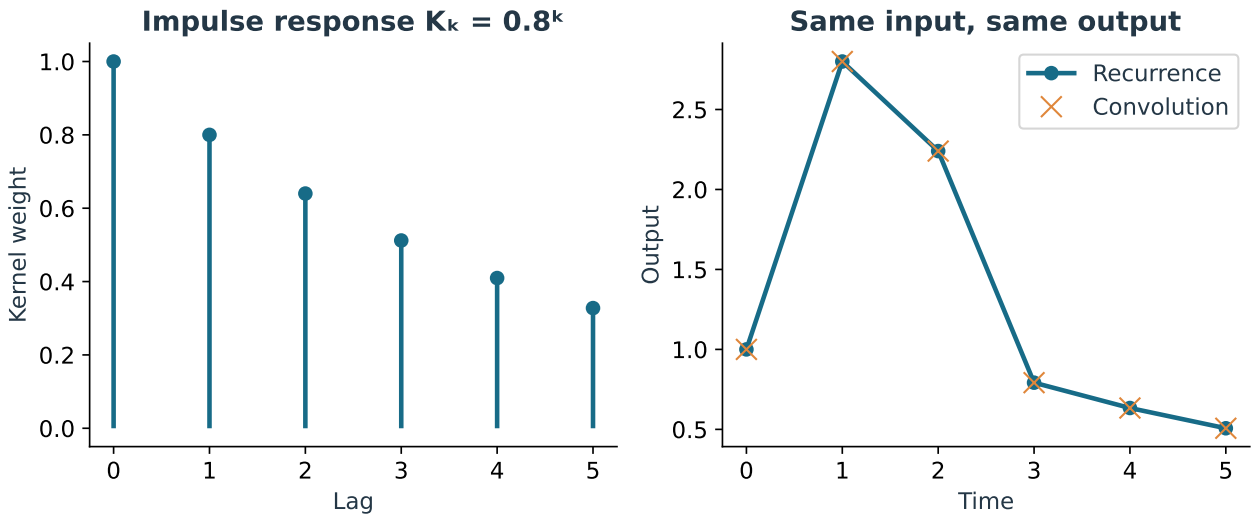


Figure 15.5: The same linear state-space model can run recurrently or as a convolution with its impulse response.

Lecture 16

State-Space Models (S4 and Mamba)

Course sources: lecture_16; lecture_17; 2025-10-23 transcript. Lecture numbering follows the original reader; linked faculty handouts identify the overlapping source coverage.

16.1 Recap: State-Space Models (SSMs) for Sequence Modeling

Deeper explanation: Diagonalization, poles, and memory (p. 413)

We continue our discussion on State-Space Models (SSMs), which are motivated by two key challenges with Recurrent Neural Networks (RNNs):

1. **Parallelization:** RNNs process sequences sequentially, making it difficult to parallelize training over the time dimension. The computation of the hidden state h_t depends on h_{t-1}.
2. **Long-Range Dependencies:** The sequential application of non-linearities and matrix multiplications can lead to vanishing or exploding gradients, making it hard for RNNs to capture dependencies over long time horizons. The influence of past inputs decays over time.

The SSM Approach: Linear Time-Invariant (LTI) Systems

The core idea behind modern SSMs is to eliminate the "horizontal" non-linearity in the recurrence relation.

- A traditional RNN has the form: $\vec{h}_t = \sigma(A\vec{h}_{t-1} + B\vec{x}_t + \vec{b})$
- By removing the non-linearity σ and bias $\vec{b}$, we obtain a linear system: $\vec{h}_t = A\vec{h}_{t-1} + B\vec{x}_t$

This results in a standard **Linear Time-Invariant (LTI) state-space model**:

$$\vec{h}_t = A\vec{h}_{t-1} + B\vec{x}_t \quad \text{(State equation)} \tag{16.1}$$

$$\vec{y}_t = C\vec{h}_t + D\vec{x}_t \quad \text{(Output equation)} \tag{16.2}$$

where A, B, C, D are learned weight matrices that are constant over time.

By unrolling the recurrence, we can express the output $\vec{y}_k$ directly in terms of all previous inputs, without depending on intermediate hidden states:

$$\vec{y}_k = CA^k\vec{h}_0 + \sum_{i=1}^{k} CA^{k-i}B\vec{x}_i + D\vec{x}_k \tag{16.3}$$

Assuming an initial state $\vec{h}_0 = \vec{0}$, this becomes:

$$\vec{y}_k = D\vec{x}_k + CB\vec{x}_{k-1} + CAB\vec{x}_{k-2} + \cdots + CA^{k-1}B\vec{x}_1 \tag{16.4}$$

This is a **convolution**. The output at time k is a weighted sum of all inputs up to time k. The convolution kernel (or filter) is defined by the sequence of matrices $(D, CB, CAB, \ldots, CA^{k-1}B, \ldots)$.

This convolutional form has a major advantage: it can be computed in parallel for all time steps, dramatically speeding up training. A naive convolution over a sequence of length T requires $O(T^2)$ computation. However, we can use the **Fast Fourier Transform (FFT)** to compute the convolution in $O(T \log T)$ time.

16.2 Efficient Parallel Training for SSMs (The S4 Model)

Deeper explanation: Associative scans preserve a different kind of parallelism (p. 415)

The Structured State-Space Model (S4) builds on this foundation by addressing several practical challenges.

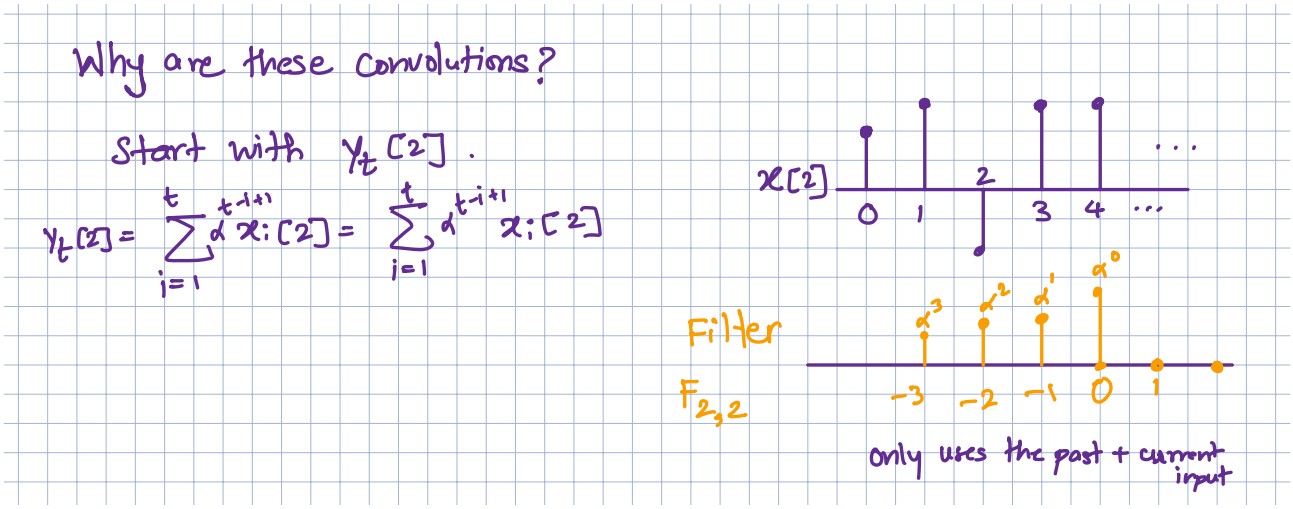


Figure 16.1: The causal convolution kernel aligned with the sequence. The impulse-response coefficients align with past inputs, not future inputs. This convolution computes the same LTI recurrence with the stated boundary and initial-state conventions. Original annotated source excerpt; Gireeja Ranade, faculty handout 16, p. 9.

Challenge 1: Computing Powers of Matrix A

Computing the coefficients of the convolution kernel requires calculating powers of the matrix A (A^k). This is computationally expensive and must be done sequentially.

- **Solution:** Add structure to A. The easiest matrix to exponentiate is a **diagonal matrix**.
- We constrain A to be diagonal. This allows us to compute A^k by simply raising each diagonal element to the power of k.
- While this restricts the model to diagonalizable systems, we can learn the basis (eigenvectors) implicitly, allowing the model to represent a large family of linear systems.

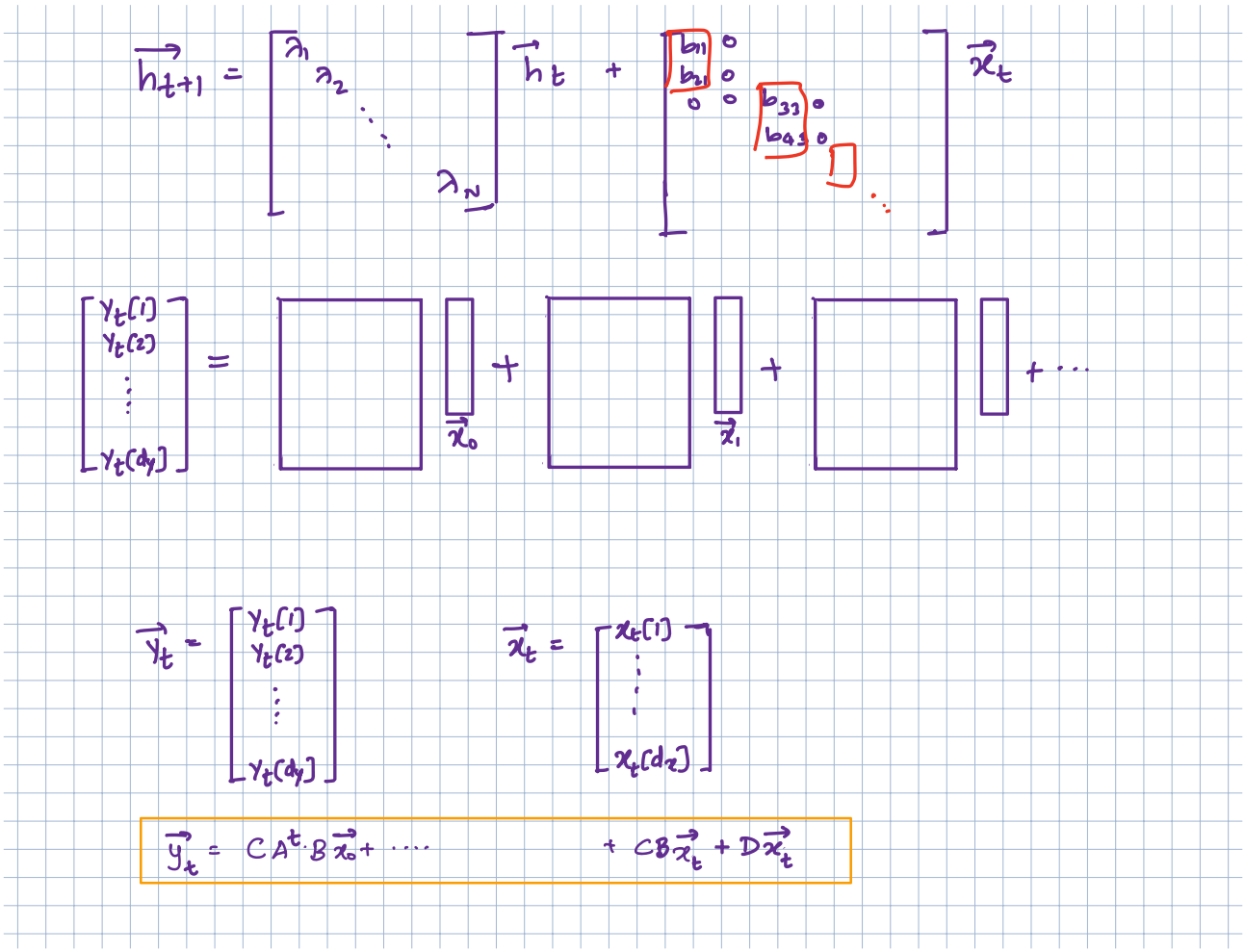


Figure 16.2: Block-diagonal state transitions. Block structure restricts interactions among groups of state coordinates. Powers and structured operations can exploit this organization; it is distinct from assuming every dense transition is already cheap. Original annotated source excerpt; Gireeja Ranade, faculty handout 16, p. 11.

Challenge 2: Applying FFT to Vector Sequences

The standard FFT algorithm is designed for scalar sequences, but our inputs $\vec{x}_t$ and outputs $\vec{y}_t$ are vectors. Let d_x, d_y, and d_h be the dimensions of the input, output, and hidden state, respectively. The hidden state dimension d_h is our "memory" capacity.

- **Solution:** Decompose the problem. We can view a sequence of vectors as a stack of scalar sequences.
- For the output vector $\vec{y}_t = [y_t[1], y_t[2], \ldots, y_t[d_y]]^T$, we can consider the time evolution of each component $y_t[k]$ separately.
- The k-th component of the output, $y_t[k]$, is a linear combination of all components of all past inputs $\vec{x}_i$ for $i \leq t$.
- This means we can express the sequence of the k-th component, denoted $Seq(y[k])$, as a sum of convolutions involving the sequences of each input component, $Seq(x[l])$:

$$Seq(y[k]) = \sum_{l=1}^{d_x} F_{kl} * Seq(x[l]) \tag{16.5}$$

  where F_{kl} is an appropriate scalar filter and $*$ denotes convolution. Each of these scalar convolutions can be computed efficiently using FFT.

Challenge 3: Reducing the Number of Convolutions

The decomposition above results in $d_x \times d_y$ separate convolutions. If $d_x = d_y = 100$, this means 10,000 convolutions, which is computationally intensive.

- **Solution:** Take inspiration from depthwise/channel-wise convolutions in CNNs. We can prevent channels from mixing within the SSM layer.
- We enforce a structure where each input channel only influences a corresponding set of hidden state dimensions and output channels, without interacting with others.
- Since A is already diagonal, this is achieved by imposing a **block-diagonal structure** on matrices B and C.
- This sets many of the filters F_{kl} to zero, drastically reducing the number of required convolutions and speeding up computation.

Challenge 4: Capturing Long-Range Dependencies

The other major goal of SSMs is to avoid the vanishing history problem of RNNs. This is determined by the eigenvalues of the state matrix A. Since we've constrained A to be diagonal, its eigenvalues λ_i are its diagonal entries.

- If $|\lambda_i| > 1$: The corresponding hidden state component will explode exponentially. This leads to numerical instability.
- If $|\lambda_i| < 1$: The state component decays to zero. This is the vanishing history problem we want to avoid.
- If $|\lambda_i| = 1$: The state component's magnitude is preserved over time. This preserves a free mode, but does not guarantee bounded response to persistent inputs. Strictly decaying modes can provide useful long but finite timescales.

If we restrict eigenvalues to be real numbers, the only options are $\lambda = +1$ and $\lambda = -1$. This is a very limited, discrete set, which is not amenable to gradient-based optimization.

- **Solution:** Use **complex numbers** for the eigenvalues. This allows the eigenvalues to lie anywhere on the unit circle in the complex plane, a continuous space perfect for optimization. The complex part introduces rotation, while the magnitude determines scaling.
- To ensure stability (i.e., $|\lambda| \leq 1$), we can parameterize the eigenvalues. For example, by representing them in polar form $re^{i\theta}$ and constraining $r \leq 1$, or by parameterizing the real and imaginary parts. A common parameterization is:

$$\lambda = e^{-\text{ReLU}(p_{\text{real}})+j\cdot p_{\text{imag}}} \tag{16.6}$$

  Here, p_{real} and p_{imag} are learnable parameters. The $-\text{ReLU}(\cdot)$ term ensures the real part of the exponent is non-positive, which guarantees $|\lambda| \leq 1$ for this diagonal model. For a general nondiagonal transition, unit-modulus eigenvalues alone do not rule out growth from Jordan blocks.
- **Initialization:** The performance of S4 is very sensitive to the initialization of these eigenvalues. Techniques derived from the HiPPO (High-order Polynomial Projection Operator) framework provide strong initializations that place eigenvalues strategically near the unit circle. In fact, one can achieve good performance by freezing these HiPPO-initialized eigenvalues and only training matrices B and C.

By carefully engineering the system to maintain information over time, S4 models excel at tasks requiring long-range dependencies, outperforming many Transformer-based models on benchmarks like the Long Range Arena.

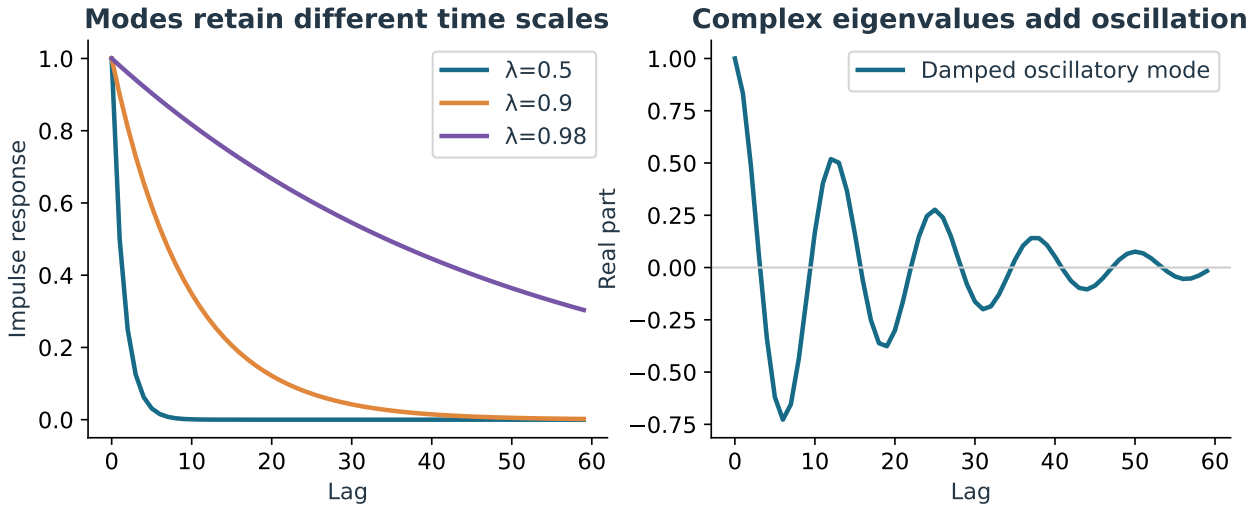


Figure 16.3: State-space kernels combine decay and oscillation. Eigenvalues near the unit circle retain information over longer intervals.

16.3 Mamba: Selective State-Space Models (S6)

Deeper explanation: Exact discretization and the meaning of a learned step size (p. 414)

Deeper explanation: Compare state memory with retrieval memory (p. 415)

While S4 is powerful, its LTI nature is a limitation. The convolution kernel is fixed and does not depend on the input. This means the way the model looks at the past is static for all tokens. Mamba (also known as S6: Structured Selective State-Space) introduces **selectivity**, allowing the system to adapt its behavior based on the input data.

A New Perspective: Continuous-Time Systems

Mamba's key idea comes from viewing the discrete-time SSM as a discretization of an underlying continuous-time linear system:

$$\dot{\vec{h}}(t) = A_c\vec{h}(t) + B_c\vec{x}(t) \tag{16.7}$$

To convert this to a discrete-time system, we need to choose a sampling interval, Δ. Using a standard discretization method (like the zero-order hold), we get a discrete-time system $\vec{h}_k = \bar{A}\vec{h}_{k-1} + \bar{B}\vec{x}_{k-1}$ where the new matrices $\bar{A}$ and $\bar{B}$ are functions of the original continuous matrices A_c, B_c and the sampling interval Δ:

$$\bar{A} = e^{A_c\Delta} \tag{16.8}$$

$$\bar{B} = (e^{A_c\Delta} - I)A_c^{-1}B_c \tag{16.9}$$

The inverse formula assumes A_c is invertible. In general use $\bar{B} = \int_0^\Delta e^{A_c s}B_c\,ds$, which remains valid for singular A_c. This introduces Δ as a new degree of freedom.

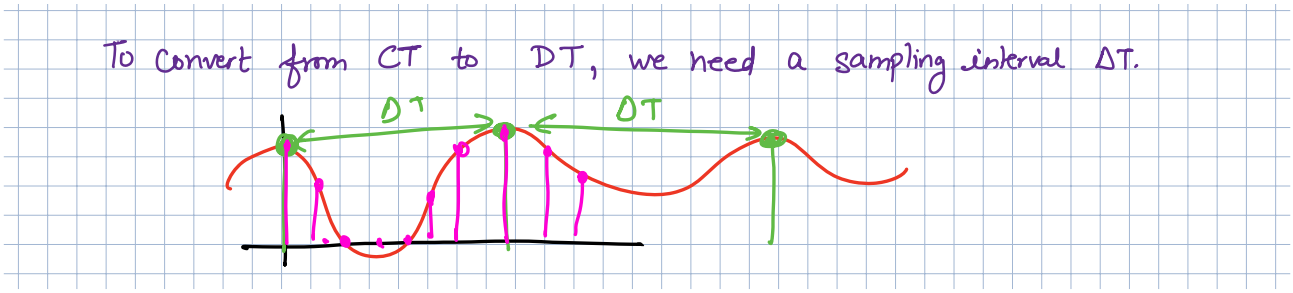


Figure 16.4: Continuous evolution and discrete samples. Sampling a continuous state yields a discrete sequence. The interval between samples becomes part of the discrete transition, so changing the interval changes the effective memory timescale. Original annotated source excerpt; Gireeja Ranade, faculty handout 17, p. 5.

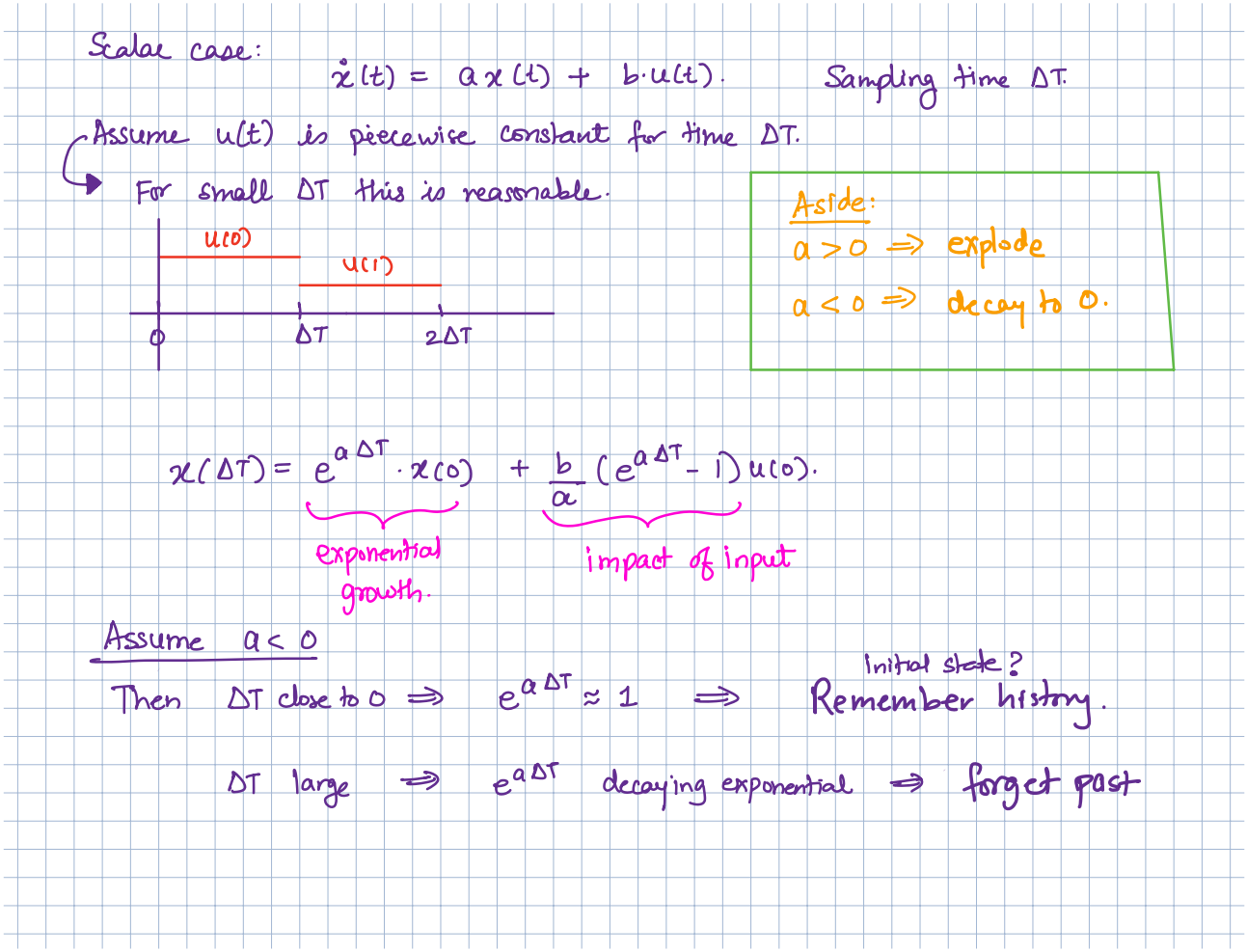


Figure 16.5: Holding the input between samples. The piecewise-constant input is the zero-order-hold assumption used to derive a discrete state update. It is an assumption about the signal between observations, not merely a plotting convention. Original annotated source excerpt; Gireeja Ranade, faculty handout 17, p. 6.

Learning to be Selective with Δ

The sampling interval Δ controls the trade-off between attending to the past state versus the current input. Assume the real parts of the eigenvalues of A_c are negative for stability.

- **Small Δ**:
  - $\bar{A} = e^{A_c \Delta} \approx I$. The new state $\vec{h}_k$ is very close to the old state $\vec{h}_{k-1}$. The system **remembers its history**.
  - $\bar{B} \approx B_c \Delta$. The influence of the current input $\vec{x}_{k-1}$ is small.
- **Large Δ**:
  - $\bar{A} = e^{A_c \Delta} \to 0$. The old state $\vec{h}_{k-1}$ is effectively forgotten. The system can **reset its memory**.
  - $\bar{B} \to -A_c^{-1} B_c$. The influence of the current input is large.

The Mamba Idea: Make Δ input-dependent. By learning a function that maps the input $\vec{x}_t$ to a value Δ_t, the model can dynamically decide whether to remember or forget, for each token.

- Since Δ_t must be positive (it's a time interval), a simple linear layer is insufficient.
- We can use an activation function that outputs positive values. A **softplus** function works well:

$$\Delta_t = \text{softplus}(\text{Linear}(\vec{x}_t)) = \log(1 + \exp(\text{Linear}(\vec{x}_t))) \tag{16.10}$$

Now, the state-space matrices $\bar{A}_t$ and $\bar{B}_t$ change at every time step, making the system **time-varying** and input-dependent. This is the source of selectivity.

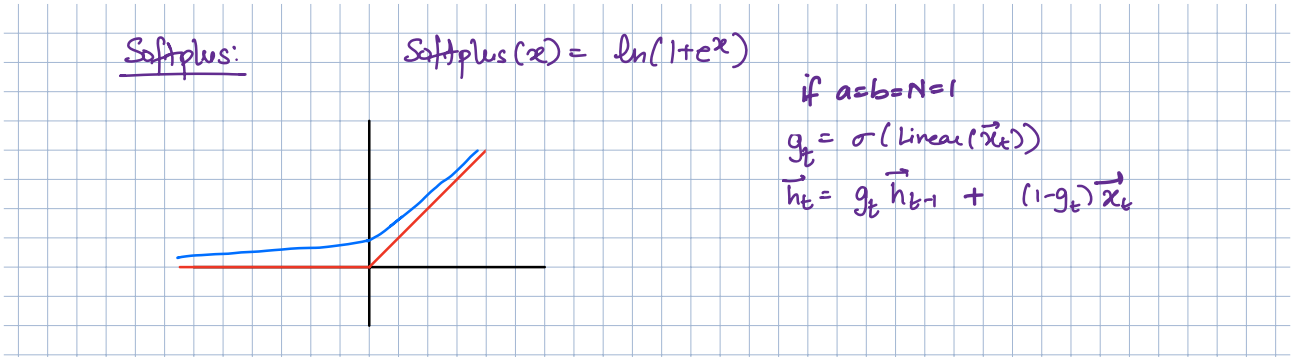


Figure 16.6: A positive learned time step through softplus. Softplus turns an unconstrained input-dependent quantity into a positive interval. Its smooth positive output supports learning how strongly a state should update, without equating the gate with a literal binary switch. Original annotated source excerpt; Gireeja Ranade, faculty handout 17, p. 8.

Implications for Training

The input-dependent nature of Mamba breaks the LTI property.

- **Loss of FFT:** Since the system is no longer a convolution with a fixed kernel, the $O(T \log T)$ FFT trick cannot be used for training.
- **Parallel Scan:** However, the model retains the key property of having no "horizontal" non-linearity. The recurrence can still be computed in parallel. This is done using a hardware-aware **parallel scan** algorithm. While more computationally complex than FFT, it is still highly parallelizable on modern hardware like GPUs and avoids the sequential bottleneck of RNNs.
- **Inference:** At inference time, the model operates just like an RNN. The state $\vec{h}_t$ can be computed from $\vec{h}_{t-1}$ and $\vec{x}_t$, making it very fast and memory-efficient as it only needs to keep track of the current state, not the entire history.

Affine scan as augmented block multiplication

After the input-dependent coefficients are computed, write a selective recurrence as $h_t = A_t h_{t-1} + b_t$, where $b_t = B_t x_t$. Augment the state by a constant coordinate:

$$\begin{bmatrix} h_t \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} A_t & b_t \\ 0^\top & 1 \end{bmatrix}}_{M_t} \begin{bmatrix} h_{t-1} \\ 1 \end{bmatrix}.$$

Composing two steps gives

$$M_2 M_1 = \begin{bmatrix} A_2 A_1 & A_2 b_1 + b_2 \\ 0^\top & 1 \end{bmatrix}.$$

This is the affine-scan rule $(A_2, b_2) \circ (A_1, b_1) = (A_2 A_1, A_2 b_1 + b_2)$. Associativity allows a parallel prefix computation, but the order is chronological and cannot be swapped. The identity is $(I, 0)$, and no inverse or division is required, even when A_t is singular. Diagonal or other structured A_t makes these compositions economical; writing dense augmented matrices explains the algebra and is not a recommendation to materialize them in code.

Worked derivation: continuous-time discretization and selective gates

Lecture basis: faculty handout 17, pp. 2–8. The steps below expand the lecture argument.

Step 1: solve the differential equation over one constant-input interval. Let $\dot{h}(t) = Ah(t) + Bx$, with x held fixed on $[0, \Delta]$. Multiply by the integrating factor e^{-At}:

$$\begin{aligned} \frac{d}{dt}(e^{-At}h(t)) &= e^{-At}Bx, \\ e^{-A\Delta}h(\Delta) - h(0) &= \int_0^{\Delta} e^{-As}Bx\, ds, \\ h(\Delta) &= e^{A\Delta}h(0) + \left[\int_0^{\Delta} e^{Au}\, du\right] Bx. \end{aligned}$$

Thus $\overline{A} = e^{A\Delta}$ and $\overline{B} = (\int_0^{\Delta} e^{Au}du)B$. If A is invertible, $\overline{B} = A^{-1}(e^{A\Delta} - I)B$. The integral is the valid definition even when A is singular.

Step 2: evaluate a scalar mode and its zero limit. For $\dot{h} = ah + bx$,

$$h_{t+1} = e^{a\Delta}h_t + \begin{cases} b(e^{a\Delta} - 1)x_t/a, & a \neq 0, \\ b\Delta x_t, & a = 0. \end{cases}$$

The second case is the limit of the first because $e^{a\Delta} - 1 = a\Delta + O(a^2\Delta^2)$. For $a = -1, b = 1$, set $\Delta = \log(1 + e^u)$:

$$\begin{aligned} e^{-\Delta} &= 1/(1 + e^u) = 1 - \text{sigmoid}(u), \\ 1 - e^{-\Delta} &= \text{sigmoid}(u) = g, \\ h_{t+1} &= (1 - g)h_t + gx_t. \end{aligned}$$

This is a learned interpolation between old state and new input. For a stable scalar mode, the exact input coefficient saturates as Δ grows; the approximation $\overline{B} \approx \Delta B$ is only a small-step approximation.

Step 3: explain memory and parallel scan. A complex continuous-time eigenvalue $-a + i\omega$ becomes $e^{-a\Delta}e^{i\omega\Delta}$: decay times rotation. Positive a gives decay; unit modulus retains a mode but does not guarantee bounded response to arbitrary sustained input. A selective recurrence has the form $h_t = A_t h_{t-1} + b_t$. Composing two updates gives

$$(A_2, b_2) \circ (A_1, b_1) = (A_2A_1, A_2b_1 + b_2).$$

Composing a third in either order yields $(A_3A_2A_1, A_3A_2b_1 + A_3b_2 + b_3)$. This proves associativity, enabling a parallel prefix scan even when the coefficients depend on the input. It does not restore a single time-invariant convolution kernel.

The Mamba Architecture Block

A typical Mamba block combines the selective SSM with other standard neural network components:

1. The input sequence is passed through a linear projection layer.
2. The result is split into two parallel paths.
3. **Path 1 (SSM Path):** The sequence first goes through a 1D convolution (to capture local context), then a SiLU/Swish activation, and finally into the selective SSM layer. The SSM uses the input to dynamically generate the parameters $\Delta_t, \bar{B}_t, \bar{C}_t$.
4. **Path 2 (Gating Path):** The sequence goes through a linear projection and a SiLU/Swish activation. This path acts as a gate.
5. The outputs of the two paths are multiplied element-wise. This gating mechanism allows the model to control the flow of information.

This architecture effectively merges the long-range context capabilities of SSMs with the local processing power of convolutions and the dynamic information control of gating mechanisms.

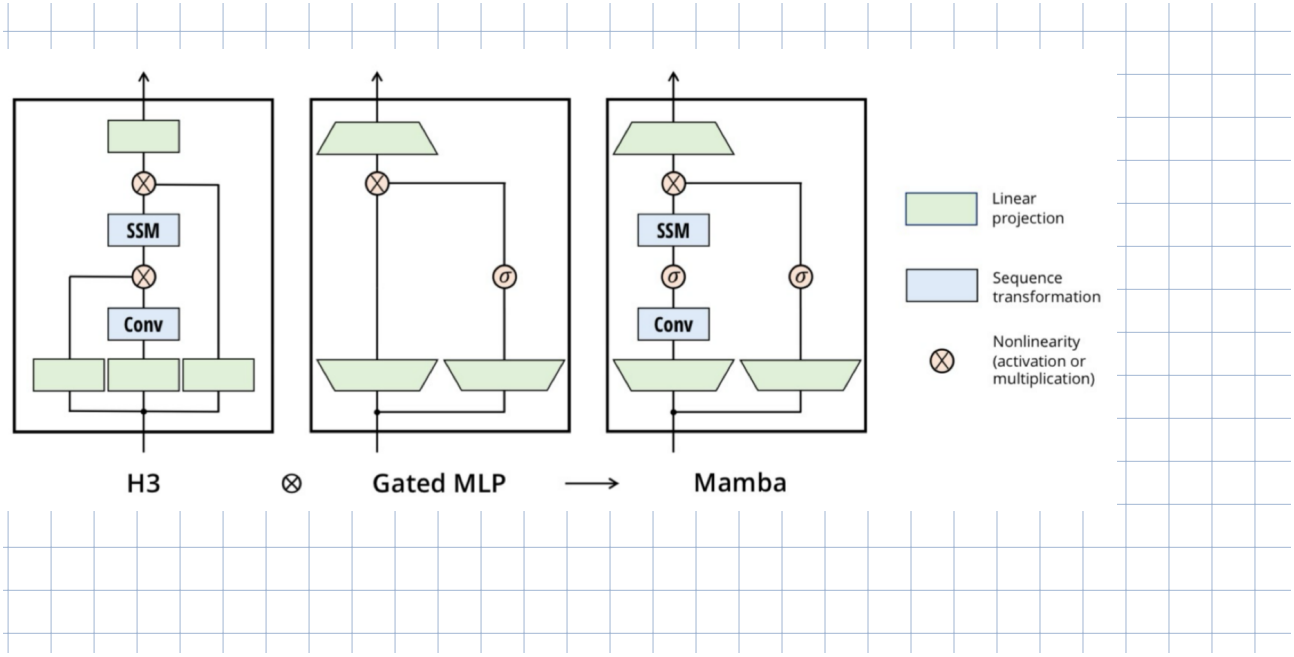


Figure 16.7: H3, gated MLP, and Mamba side by side. Compare sequence mixing, projections, and multiplicative gates rather than only the outer residual wrapper. These are the published architecture diagrams reproduced in the faculty handout. Original annotated source excerpt; Gireeja Ranade, faculty handout 16, p. 12.

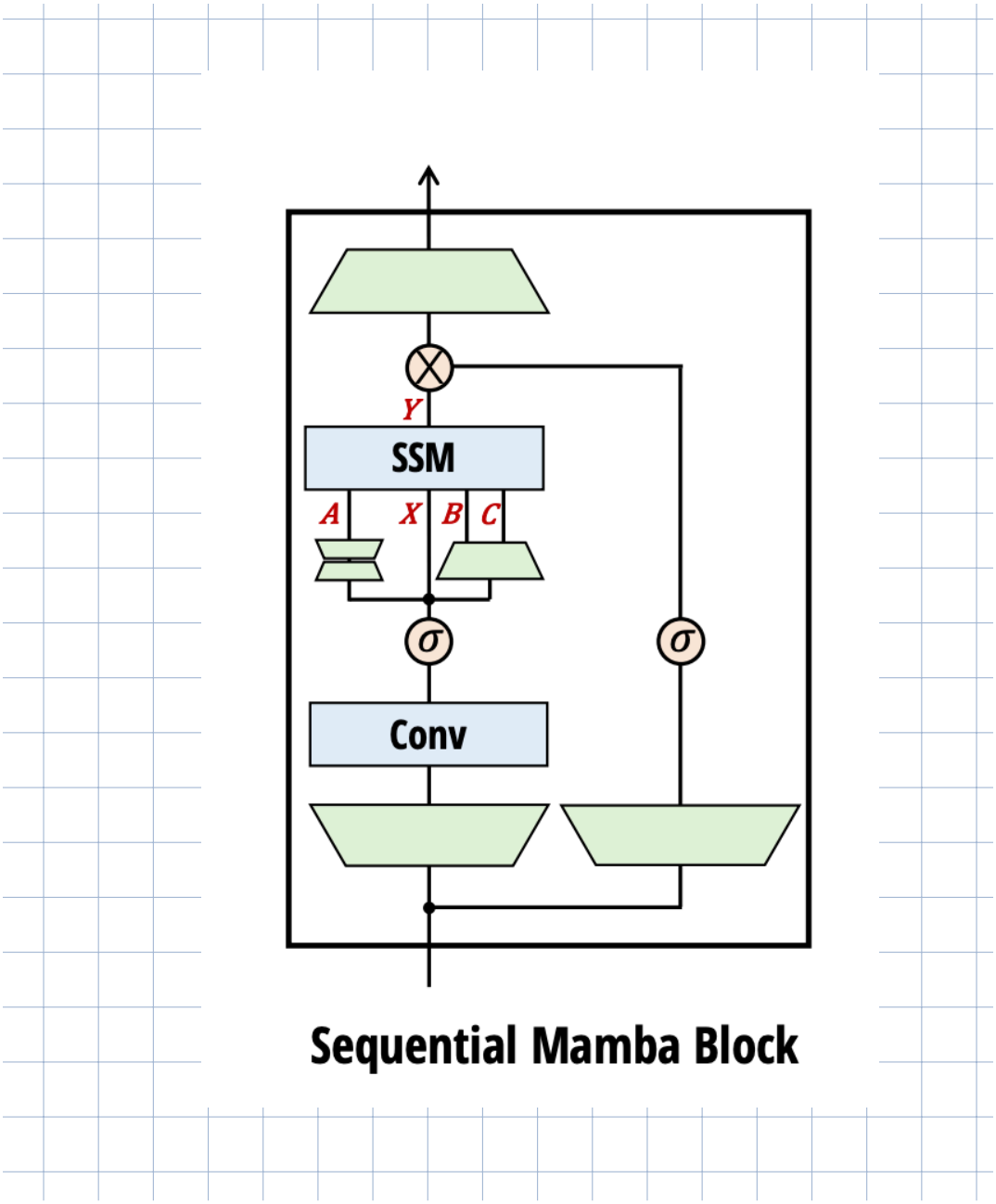


Figure 16.8: The selective state-space block annotated in class. The annotations identify which quantities depend on the current input. That dependence is the reason selective recurrence is more expressive than a fixed LTI convolution, and also why the fixed-kernel FFT shortcut no longer directly applies. Original annotated source excerpt; Gireeja Ranade, faculty handout 17, p. 10.

Lecture 17

Attention Mechanisms

Course sources: lecture_18; 2025-10-28 transcript. Lecture numbering follows the original reader; linked faculty handouts identify the overlapping source coverage.

17.1 Introduction: From State-Space Models to Attention

Deeper explanation: Attention as a learned kernel, with important limits (p. 420)

We previously discussed State-Space Models (SSMs) as a way to address challenges with Recurrent Neural Networks (RNNs), particularly the difficulty of parallelizing training. SSMs leverage a convolution perspective, which allows for efficient, parallel training. Their strength lies in the ability to use a very large hidden state, h, to store a vast amount of information about the sequence history. The output matrix, C, can then project this high-dimensional state back to a lower-dimensional output, y, managing the model's parameter count.

Today, we explore another approach to overcoming the limitations of RNNs: **attention mechanisms**. Historically, attention was developed to improve RNN performance before the famous "Attention is All You Need" paper demonstrated that the RNN structure could be discarded entirely. Attention and SSMs represent two distinct evolutionary paths stemming from the challenges of RNNs.

17.2 Sequence-to-Sequence Problems with RNNs

Deeper explanation: Attention as a learned kernel, with important limits (p. 420)

The Encoder-Decoder Architecture

A common application for sequence-to-sequence tasks, such as machine translation, is the **encoder-decoder architecture**.

- **Encoder**: This part of the network processes the input sequence (e.g., an English sentence). It is typically composed of a stack of RNN layers. As it processes the input token by token, it builds up a hidden state that is intended to summarize the entire input sequence.
- **Decoder**: This part of the network generates the output sequence (e.g., the translated Chinese sentence). It is also a stack of RNN layers. It begins with a special "start" token and generates the output sequence one token at a time.

The core idea is to transfer the ”meaning” or ”context” captured by the encoder to the decoder. In a standard RNN-based architecture, this is done by using the final hidden state of the encoder's layers to initialize the hidden states of the decoder's corresponding layers.

Visualizing the Architecture and the Bottleneck

We can visualize this process by ”unrolling” the RNNs through time.

- The horizontal axis represents the time or sequence dimension.
- The vertical axis represents the layers of the network.

The encoder processes the input sequence of length T, $x_1, x_2, \dots, x_T$. After processing the final token x_T, the encoder's final hidden state, h_T, contains a summary of the entire input sentence. This single state vector is then passed to initialize the decoder.

This creates a significant **information bottleneck**. All information about the input sentence—its meaning, grammar, and nuances—must be compressed into a single, fixed-size state vector. For long sentences, this compression is lossy and makes it very difficult for the decoder to generate an accurate translation. The only way to cram more information in is to make the hidden state dimension larger, which makes the network much more difficult and computationally expensive to train. The length of the input sequence processed by the encoder does not have to be the same as the length of the output sequence generated by the decoder.

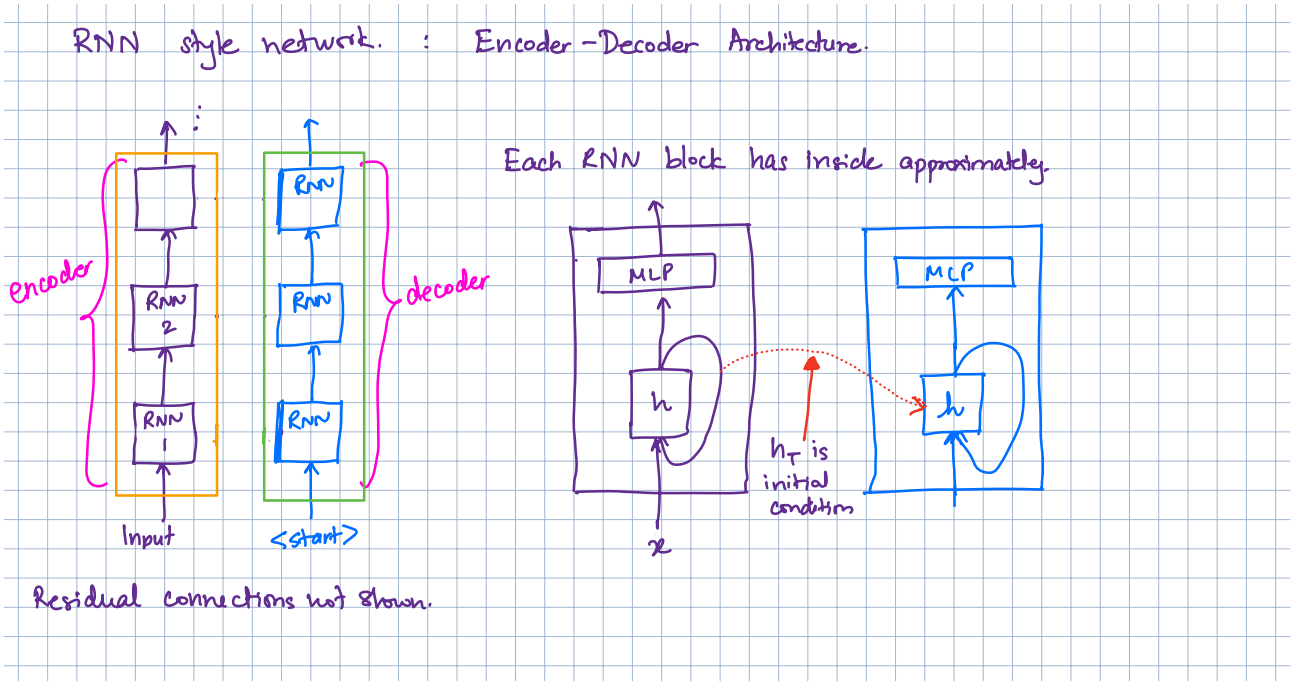


Figure 17.1: The sequence-to-sequence state bottleneck. The encoder compresses a variable-length source into the state supplied to the decoder. The drawing reveals where access to individual source positions is lost, motivating cross-attention. Original annotated source excerpt; Gireeja Ranade, faculty handout 18, p. 1.

17.3 Training Sequence-to-Sequence Models

Deeper explanation: A two-token numerical attention example (p. 418)

Teacher Forcing

During training, we have access to the ground truth output sequence (the correct translation). Instead of feeding the decoder's own (potentially incorrect) previous output as the input for the next step, we can use the ground truth token. This technique is called **teacher forcing**.

- **Process**: At each time step t of the decoder, we provide the correct token from the training data, y_{t-1}, as input to generate the prediction $\hat{y}_t$.
- **Advantage**: This prevents the model from compounding its own errors during training. If the model generates a wrong word at an early step, feeding that wrong word back in could lead it down a completely incorrect path, making it difficult for gradients to correct the behavior. Teacher forcing keeps the model on track and stabilizes training.

Autoregressive Generation (Inference Time)

At test (or inference) time, we do not have the ground truth translation. The model must generate the output sequence on its own. This is done **autoregressively**.

- **Process**:
  1. The decoder starts with a learned ”start” token.
  2. It generates a probability distribution over the vocabulary for the first output token.
  3. A token is chosen from this distribution (e.g., by sampling).
  4. This chosen token is then fed back as the input for the next time step.
  5. This process repeats until the model generates a special ”end” token.
- **Sampling**: Instead of greedily picking the token with the highest probability at each step, it is often better to *sample* from the output distribution. This introduces randomness, which can lead to more diverse and often higher-quality outputs. A ”temperature” parameter can control the level of randomness: low temperature approaches greedy selection, while high temperature increases randomness.

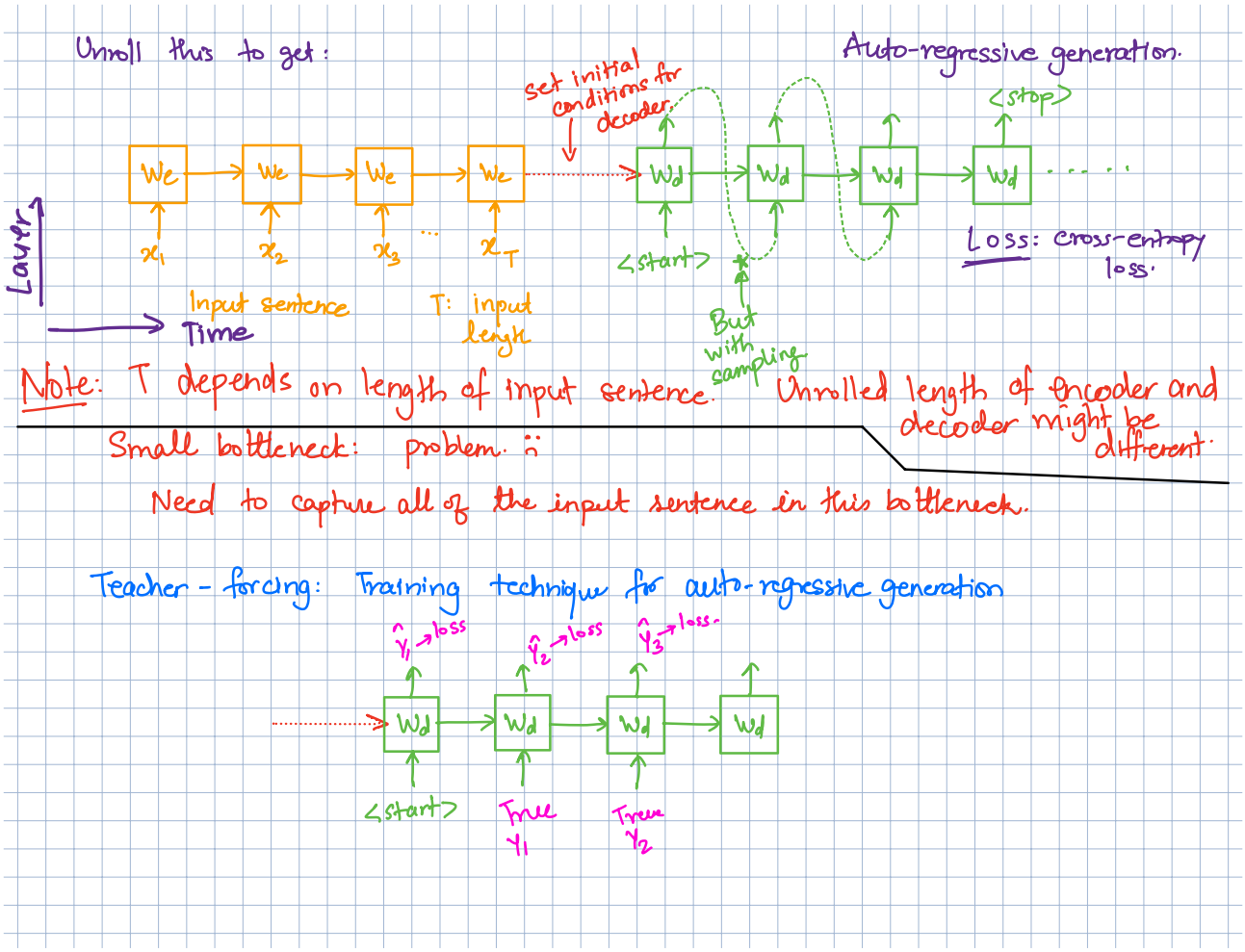


Figure 17.2: Teacher forcing versus feeding back a generated token. During training, the preceding ground-truth token is available; at inference, the preceding model output supplies the next input. The unrolled topology makes the change in input distribution explicit. Original annotated source excerpt; Gireeja Ranade, faculty handout 18, p. 2.

Worked derivation: sequence likelihood and the token loss

Lecture basis: faculty handout 18, pp. 1–3. The steps below expand the lecture argument.

Step 1: apply the probability chain rule. For conditioning input x and output sequence $y_1, \dots, y_T$ including its stop token,

$$p_\theta(y \mid x) = \prod_{t=1}^{T} p_\theta(y_t \mid x, y_{<t}),$$

$$-\log p_\theta(y \mid x) = -\sum_{t=1}^{T} \log p_\theta(y_t \mid x, y_{<t}).$$

Training evaluates each factor with the observed prefix. This is teacher forcing and is an exact evaluation of this autoregressive likelihood, not an approximation that replaces sampled tokens inside the same objective.

Step 2: derive the cross-entropy gradient. For logits $z \in \mathbb{R}^V$, $p_j = e^{z_j}/\sum_k e^{z_k}$, and correct class c,

$$\begin{aligned}\ell(z, c) &= -z_c + \log \sum_k e^{z_k}, \\ \frac{\partial \ell}{\partial z_j} &= -\mathbf{1}_{j=c} + \frac{e^{z_j}}{\sum_k e^{z_k}} \\ &= p_j - \mathbf{1}_{j=c}.\end{aligned}$$

The target class is pushed upward and other classes downward, with total logit gradient zero. At inference the prefix contains the model's own sampled outputs, so the distribution of prefixes can differ from training. This exposure difference does not invalidate the likelihood factorization.

17.4 Overcoming the Bottleneck: The Motivation for Attention

Deeper explanation: A complete decoder training example (p. 422)

The information bottleneck is the central problem. How can the decoder access information from the entire input sequence, not just the final hidden state?

Analogy to U-Nets

This problem is analogous to one seen in U-Nets for image segmentation. In a U-Net, an image is progressively downsampled (the "encoder" path) to capture global context, creating a bottleneck at the lowest resolution. It is then upsampled (the "decoder" path) to produce a pixel-level segmentation map. To recover fine-grained spatial information lost during downsampling, U-Nets use **skip connections**, linking layers of the same spatial resolution from the encoder path to the decoder path.

These skip connections work because there is a natural, fixed correspondence: a pixel at position (i, j) in a higher-resolution layer corresponds directly to a feature at position (i, j) in the corresponding lower-resolution layer.

The Challenge in Language

For language translation, we cannot use fixed skip connections. The "appropriate position" to look at in the input sentence changes depending on the word being generated. Word order

is not preserved across languages. For example, the third word in a French sentence might correspond to the first or fifth word in its English translation.

The connections cannot be predefined; they must be **dynamic** and depend on the input itself. We need a mechanism that allows the decoder, at each step, to look back at all the encoder's hidden states and decide which ones are most relevant for generating the next word.

17.5 Developing the Attention Mechanism: A Step-by-Step Intuition

Deeper explanation: Softmax temperature controls sensitivity as well as selection (p. 419)

We need a learnable, differentiable mechanism that functions like a dynamic lookup table or a "soft" hash table. Let's build up the idea. We'll use the concepts of a **Query**, **Key**, and **Value**.

- **Query (Q)**: Represents the current state of the decoder. It's asking, "What information should I pay attention to in the source sentence?"
- **Keys (K)**: Associated with each token in the source sentence. They are the "indices" we can look up.
- **Values (V)**: Also associated with each token in the source sentence. They contain the actual information content of that token.

Idea -1: Exact Match

Look for an exact match of the Query among all Keys and return the corresponding Value.

- **Problem**: In high-dimensional vector spaces, exact matches are virtually impossible, especially with continuous embeddings and across different languages. At initialization, no matches would be found, yielding no gradient signal to learn from. This approach is a non-starter.

Idea 0: Closest Match

Find the Key that is "closest" (e.g., highest similarity) to the Query and return its corresponding Value.

- **Problem**: This is not differentiable. A small change in the Query vector will likely not change which Key is the closest. This results in piecewise constant behavior, leading to zero gradients almost everywhere, which makes learning via gradient descent impossible.

Idea 1: Weighted Average

Instead of a hard selection of one Key, we can compute a weighted average of all Values. The weights should reflect the similarity between the Query and each Key. This makes the entire operation "soft" and differentiable.

A small change in the Query will now smoothly change its similarity to all Keys, which in turn smoothly changes all the weights and the final output. This provides a rich gradient signal for learning.

- **Similarity Score**: A simple and effective way to measure the similarity between the query vector Q and a key vector K_i is the **dot product**: $Q^T K_i$.
- **Normalization**: The dot products can result in very large numbers. To turn these scores into a well-behaved probability distribution (weights that sum to 1), we apply a **softmax** function across all the similarity scores.

$$\alpha_i = \text{softmax}(Q^T K_i) = \frac{\exp(Q^T K_i)}{\sum_j \exp(Q^T K_j)}$$

- **Final Output**: The output of the attention mechanism is the weighted sum of the Value vectors:

$$\text{Attention}(Q, K, V) = \sum_i \alpha_i V_i$$

17.6 The Attention Formula and its Components

Deeper explanation: Derive all projection gradients in one place (p. 417)

Deeper explanation: Gradients through the retrieval weights (p. 416)

Deeper explanation: Track attention with rectangular matrices (p. 416)

Making it Learnable: Q, K, V Projections

Where do the Query, Key, and Value vectors come from? They are learned linear projections of the input embeddings. For an input vector x, we learn weight matrices W_Q, W_K, and W_V:

$$Q = W_Q x$$
$$K = W_K x$$
$$V = W_V x$$

These matrices are the learnable parameters of the attention mechanism, optimized via back-propagation.

Scaled Dot-Product Attention

A key issue with the dot product is that as the dimension of the key vectors, d_k, grows, the magnitude of the dot products can also grow large. This can push the softmax function into regions where it has extremely small gradients, hurting the learning process. To counteract this, we scale the dot products before the softmax:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Here, Q, K, V are matrices containing the queries, keys, and values for all tokens in the sequence. This is the core formula for **Scaled Dot-Product Attention**.

Types of Attention

- **Cross-Attention**: This is the original form used in encoder-decoder models. The **Queries** come from the decoder's state, while the **Keys** and **Values** come from the encoder's output states for all input tokens. This allows every decoder step to attend to every part of the input sentence.
- **Self-Attention**: Here, the Queries, Keys, and Values all come from the **same sequence**. For example, within the encoder, each word can look at all other words in the input sentence to build a more context-rich representation.
- **Masked Self-Attention**: Used in decoders. When generating the i-th word, the model should only be able to attend to words $1, \ldots, i$. It cannot "look into the future." This is achieved by masking (setting to $-\infty$) the attention scores for future positions before the softmax is applied.

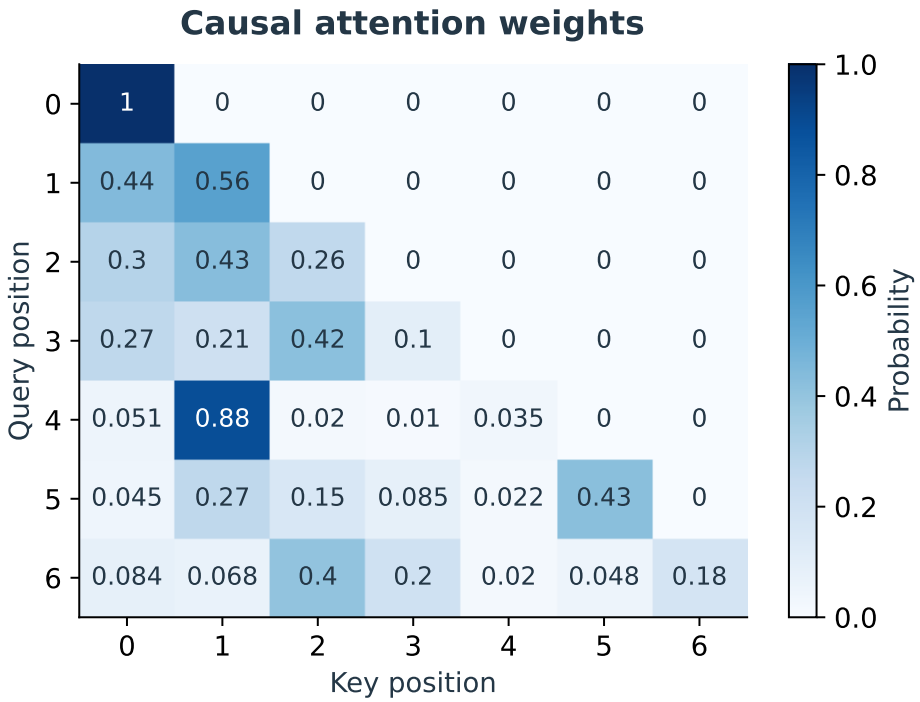


Figure 17.3: A causal attention matrix permits each query to read only the current and earlier positions. Rows are probability distributions over available keys.

17.7 Multi-Head Attention

Deeper explanation: Where quadratic cost enters (p. 420)

Instead of performing a single attention calculation with high-dimensional Q, K, V vectors, it can be beneficial to perform multiple attention calculations in parallel, each with lower-dimensional vectors. This is called **Multi-Head Attention**.

- **Intuition**: This allows the model to jointly attend to information from different representation subspaces at different positions. A single attention head might learn to focus on syntactic relationships, while another focuses on semantic relationships.
- **Mechanism**:
  1. The original Q, K, and V are each linearly projected into h lower-dimensional representations (one for each "head"). The dimension is typically split, so if the original embedding dimension is d_{model}, each head works with dimension $d_k = d_{model}/h$.
  2. Scaled dot-product attention is applied in parallel for each head, producing h output vectors.
  3. These h output vectors are concatenated and passed through a final linear projection to produce the final output.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$

where

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Multi-head attention is a standard and powerful component of modern neural network architectures, particularly Transformers, which we will discuss next.

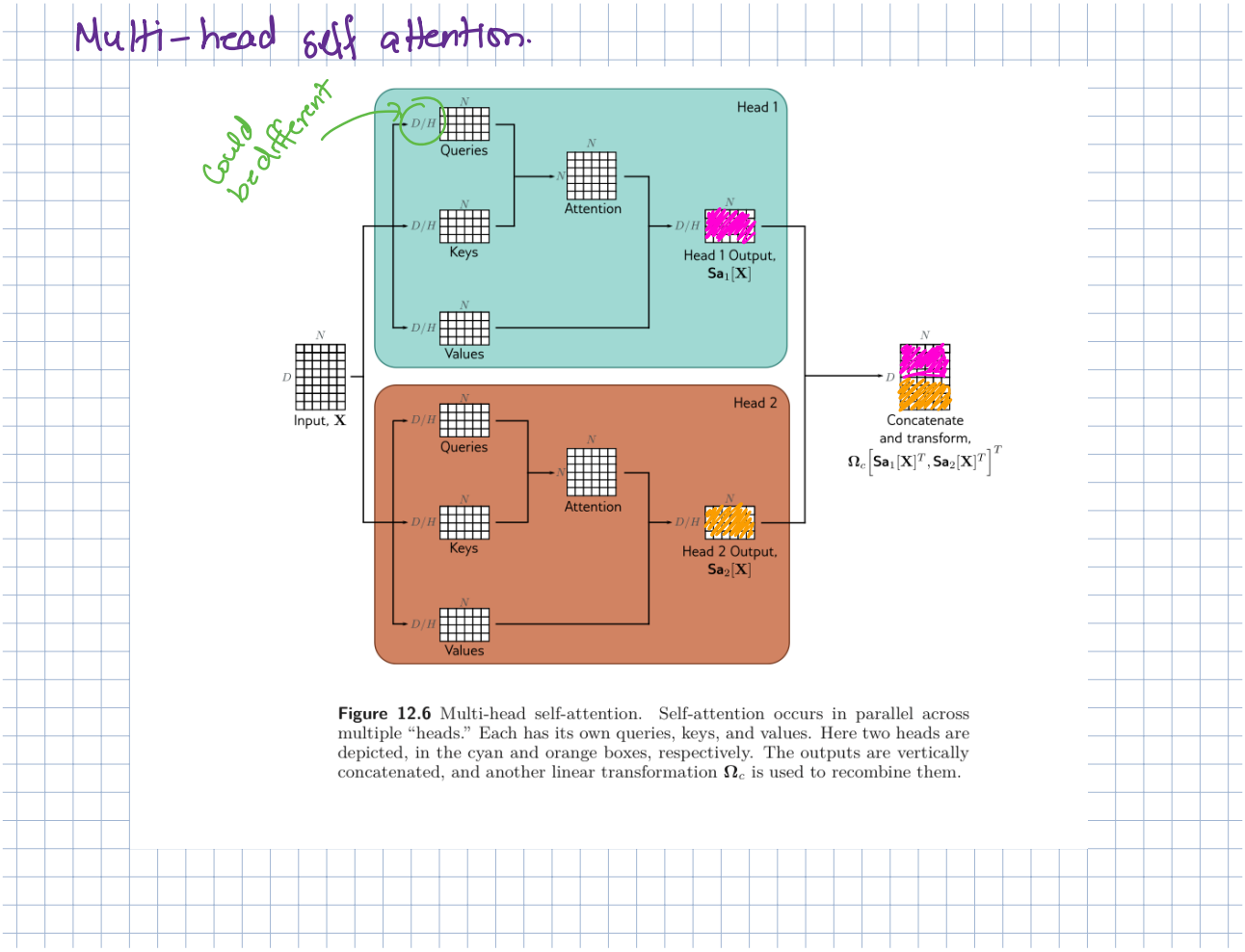


Figure 12.6 Multi-head self-attention. Self-attention occurs in parallel across multiple "heads." Each has its own queries, keys, and values. Here two heads are depicted, in the cyan and orange boxes, respectively. The outputs are vertically concatenated, and another linear transformation $\mathbf{\Omega}_c$ is used to recombine them.

Figure 17.4: Parallel heads, concatenation, and projection. Each head forms its own weighted mixture. Their outputs are concatenated and projected, rather than averaging independently normalized attention scores across heads. The published diagram is retained from the handout. Original annotated source excerpt; Gireeja Ranade, faculty handout 18, p. 11.

Lecture 18

Attention and Transformers

Course sources: lecture_18; lecture_19; 2025-10-30 transcript. Lecture numbering follows the original reader; linked faculty handouts identify the overlapping source coverage.

18.1 Recap: Sequence-to-Sequence (Seq2Seq) Models

Deeper explanation: A complete decoder training example (p. 422)

We begin by recapping the sequence-to-sequence problems, particularly in the context of Recurrent Neural Networks (RNNs).

Encoder-Decoder Architecture

A common architecture for tasks like machine translation is the encoder-decoder model.

- **Encoder:** An RNN processes the input sequence (e.g., a sentence in English) one token at a time, summarizing the information into a final hidden state vector.
- **Decoder:** Another RNN takes the encoder's final hidden state as its initial state. It then generates the output sequence (e.g., the translated sentence) auto-regressively.
  - **Auto-regressive Generation:** At each time step, the decoder produces a probability distribution over the vocabulary. A token is sampled from this distribution and fed as the input to the next time step. This process begins with a special '<start>' token.

The Bottleneck Problem

A significant limitation of this architecture is the **bottleneck**. The entire meaning and context of the input sequence must be compressed into a single, fixed-size hidden state vector.

- For long sequences, this compression is lossy and makes it difficult for the decoder to access relevant information from early parts of the input.
- Increasing the size of the hidden state to pass more information makes the model larger and harder to train.

This limitation motivated the development of the attention mechanism, which allows the decoder to look back at the entire input sequence at every step of the generation process.

Training Seq2Seq Models

- **Loss Function:** These models are typically trained using a cross-entropy loss on the predicted probability distributions for the next token.
- **Teacher Forcing:** During training, instead of feeding the model's own (potentially incorrect) prediction to the next time step, we feed the ground-truth token from the target sequence. This stabilizes training and helps the model learn the correct sequence structure more quickly.

18.2 The Attention Mechanism

Deeper explanation: A two-token numerical attention example (p. 418)

Attention provides a way to overcome the bottleneck problem by creating direct connections between the decoder and all encoder states.

Conceptual Overview: Queries, Keys, and Values

The attention mechanism is analogous to a differentiable dictionary lookup.

- **Query (Q):** Represents the current piece of information we are focused on (e.g., the current hidden state of the decoder). It's asking a question: "What information from the source is relevant right now?"
- **Keys (K):** A set of vectors associated with the source information (e.g., all encoder hidden states). Each key acts as a label or an identifier for a piece of information.
- **Values (V):** A set of vectors, where each value is paired with a key. Values contain the actual content or information.

The process is as follows:

1. The query is compared with all keys to compute a **similarity score**.
2. These scores are converted into a probability distribution (weights) using a softmax function. These weights determine how much "attention" to pay to each value.
3. The final output is a weighted sum of all the values, based on the computed attention weights.

Calculating Q, K, and V

We don't assume Q, K, and V are given. Instead, we learn them from our input data. Given an input vector $\vec{x}$, we generate Q, K, and V using learned linear transformations:

$$\vec{q}_n = W_q \vec{x}_n + \vec{\beta}_q$$
$$\vec{k}_m = W_k \vec{x}_m + \vec{\beta}_k$$
$$\vec{v}_l = W_v \vec{x}_l + \vec{\beta}_l$$

Here, W_q, W_k, W_v are learnable weight matrices, and $\vec{\beta}_q, \vec{\beta}_k, \vec{\beta}_l$ are learnable bias vectors. Gradient descent determines their values during training.

Types of Attention

Cross-Attention:

This is the form of attention used in the original encoder-decoder context.

- **Queries** come from the decoder (representing the target sequence state).
- **Keys and Values** come from the encoder (representing the source sequence).
- This allows the decoder, at each step, to look across the entire input sequence to decide what information is most relevant for generating the next output token.

Self-Attention:

In self-attention, the queries, keys, and values are all derived from the *same* source sequence.

- This allows each position in a sequence to attend to all other positions in the same sequence.
- It helps the model build richer representations by understanding dependencies and relationships within the sequence itself (e.g., pronoun resolution).
- **Masked (Causal) Self-Attention:** Used in decoders. To prevent the model from "cheating" by looking at future tokens during training, we apply a mask to the attention scores. The mask sets the scores for all subsequent positions to $-\infty$ before the softmax, ensuring a position can only attend to itself and previous positions.

18.3 Scaled Dot-Product Attention

Deeper explanation: Softmax temperature controls sensitivity as well as selection (p. 419)

The most common implementation of attention is scaled dot-product attention.

The Attention Formula

Given a query $\vec{q}$ and a set of keys $\vec{k}_j$ and values $\vec{v}_j$, the attention output is calculated as:

1. **Compute scores:** The similarity score is the dot product of the query with each key: $s_j = \langle \vec{q}, \vec{k}_j \rangle$.
2. **Apply Softmax:** The scores are converted to weights using the softmax function: $\alpha_j = \frac{\exp(s_j)}{\sum_i \exp(s_i)}$.
3. **Compute weighted sum:** The output is the weighted sum of the values: $\text{Attention}(\vec{q}, K, V) = \sum_j \alpha_j \vec{v}_j$.

The Need for Scaling

The simple dot product doesn't work well in practice. If the dimension of the key/query vectors, d_k, is large (e.g., 1024), the magnitude of the dot product $\langle \vec{q}, \vec{k}_j \rangle$ can become very large.

- A dot product is a sum of d_k terms. If the components of $\vec{q}$ and $\vec{k}$ have zero mean and unit variance, the variance of their dot product is d_k.
- Large input values to the softmax function push it into regions where the gradient is extremely small (saturation). This makes learning very difficult.

Solution: Scale the dot product scores before the softmax. Inspired by the Central Limit Theorem, we scale by the standard deviation of the sum, which is $\sqrt{d_k}$.

The **Scaled Dot-Product Attention** formula is:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

This scaling prevents the softmax inputs from getting too large, ensuring stable gradients and more effective training.

Worked derivation: attention scaling and its backward pass

Lecture basis: faculty handout 18, pp. 4–6. The steps below expand the lecture argument.

Step 1: justify the scale under explicit assumptions. For independent, centered coordinates q_i, k_i with unit variances and independent products across i, the raw score $s = \sum_{i=1}^{d_k} q_i k_i$ satisfies

$$\begin{aligned} \mathbb{E}[s] &= 0, \\ \mathrm{Var}(s) &= \sum_i \mathbb{E}[q_i^2]\mathbb{E}[k_i^2] = d_k, \\ \mathrm{Var}(s/\sqrt{d_k}) &= 1. \end{aligned}$$

This motivates the scale; learned query and key distributions need not satisfy these assumptions exactly.

Step 2: differentiate a softmax row. For $p_j = e^{s_j}/Z$, $Z = \sum_k e^{s_k}$,

$$\frac{\partial p_j}{\partial s_i} = \frac{\delta_{ij} e^{s_j} Z - e^{s_j} e^{s_i}}{Z^2} = p_j(\delta_{ij} - p_i).$$

Thus $J_p = \mathrm{diag}(p) - pp^\top$. For an upstream row gradient g, the score gradient is

$$\nabla_s L = p \odot (g - (p^\top g)\mathbf{1}).$$

Every score gradient row sums to zero because adding a common constant to logits does not change softmax.

Step 3: propagate through the attention matrices. Let $Q \in \mathbb{R}^{T_q \times d_k}$, $K \in \mathbb{R}^{T_k \times d_k}$, $V \in \mathbb{R}^{T_k \times d_v}$, $S = QK^\top/\sqrt{d_k}$, $P = \mathrm{softmax}_{\mathrm{row}}(S)$, and $O = PV$. For $G = \nabla_O L$,

$$\begin{aligned} \nabla_V L &= P^\top G, & \nabla_P L &= GV^\top, \\ (\nabla_S L)_{i,:} &= P_{i,:} \odot \left[(\nabla_P L)_{i,:} - \sum_j P_{ij} (\nabla_P L)_{ij} \right], \\ \nabla_Q L &= (\nabla_S L) K/\sqrt{d_k}, & \nabla_K L &= (\nabla_S L)^\top Q/\sqrt{d_k}. \end{aligned}$$

The scalar row sum in the middle line is broadcast across that row. A causal mask excludes forbidden keys from the row softmax; their probabilities and derivatives are zero. This derivation shows both routes through attention: changing values and changing the weights assigned to them.

18.4 Multi-Head Attention

Deeper explanation: Where quadratic cost enters (p. 420)

Instead of performing a single attention calculation, it's beneficial to allow the model to jointly attend to information from different representation subspaces. This is the idea behind Multi-Head Attention.

Mechanism

1. The input embedding dimension D is split among H different "attention heads". Each head operates on a smaller dimension of D/H.
2. For each head $i \in \{1, \dots, H\}$, we learn separate weight matrices $(W_{q,i}, W_{k,i}, W_{v,i})$ to project the input into a lower-dimensional query, key, and value space.
3. Each head computes its attention output in parallel:

$$\text{head}_i = \text{Attention}(XW_{q,i}, XW_{k,i}, XW_{v,i})$$

4. The outputs of all heads are concatenated back together.
5. This concatenated result is passed through a final linear projection layer with weight matrix W_o to produce the final output, which has the same dimension as the input.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_H)W_o$$

This allows different heads to specialize and focus on different types of information or relationships within the sequence (e.g., one head for syntactic relationships, another for semantic ones).

Head concatenation and the output projection in blocks

Use token rows: $X \in \mathbb{R}^{T\times d}$. For two heads, the combined projection is

$$X\begin{bmatrix} W_Q^{(1)} & W_Q^{(2)} \end{bmatrix} = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix}, \qquad W_Q^{(j)} \in \mathbb{R}^{d\times d_k}.$$

Construct keys and values in the same way, but compute a separate token-mixing matrix in each head:

$$P_j = \text{softmax}_{\text{row}}\left(\frac{Q_j K_j^\top}{\sqrt{d_k}} + M\right), \qquad O_j = P_j V_j.$$

Here M is the additive mask and $V_j \in \mathbb{R}^{T\times d_v}$. Splitting the output matrix vertically makes the final operation explicit:

$$\begin{bmatrix} O_1 & O_2 \end{bmatrix}\begin{bmatrix} W_{O,1} \\ W_{O,2} \end{bmatrix} = O_1 W_{O,1} + O_2 W_{O,2} \in \mathbb{R}^{T\times d}, \quad W_{O,j} \in \mathbb{R}^{d_v\times d}.$$

Concatenation is not summation: the sum appears only after multiplication by the corresponding output blocks. Also, a single softmax of the combined scores would not reproduce two independently normalized heads. For causal attention, $M_{ts} = -\infty$ for $s > t$, so P_j is lower triangular, not generally diagonal or Toeplitz.

Variations for Efficiency

Multi-Query Attention:
To save computation and memory, use the same keys and values for all attention heads. Only the query is projected differently for each head.

Grouped-Query Attention:
A compromise where heads are divided into groups. Heads within the same group share keys and values.

18.5 The Transformer Architecture

Deeper explanation: A complete decoder training example (p. 422)

The seminal paper "Attention is All You Need" introduced the Transformer, a model architecture that dispenses with recurrence and convolutions entirely, relying solely on self-attention mechanisms.

Overall Structure

The Transformer follows an encoder-decoder structure, but each component is built from stacked attention-based layers.

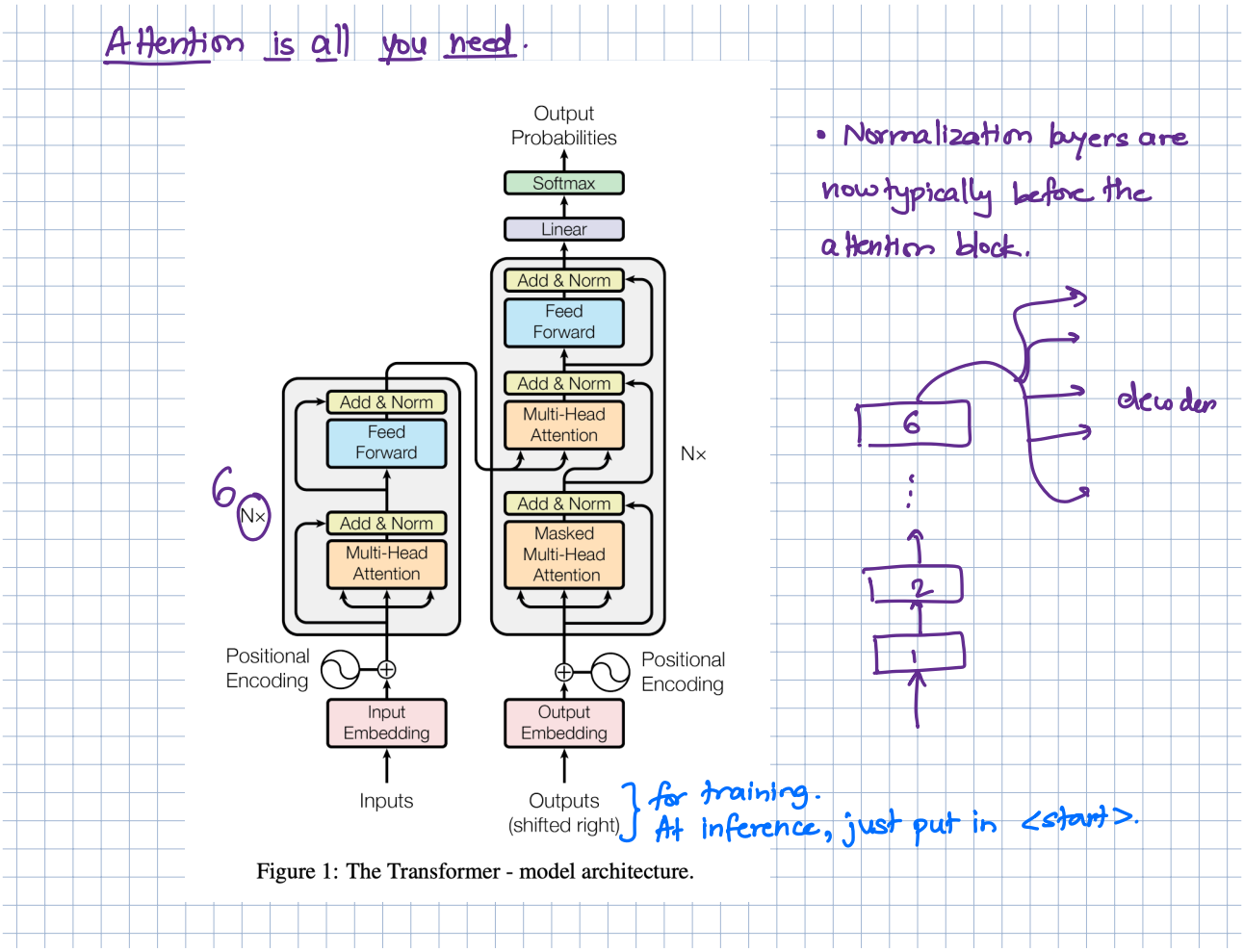


Figure 18.1: The full encoder-decoder Transformer. The original published architecture, with faculty annotations, distinguishes self-attention, causal decoder attention, and cross-attention to encoder outputs. The reader's decoder-only diagram is a different specialization. Original annotated source excerpt; Gireeja Ranade, faculty handout 19, p. 2.

The Encoder Block

Each encoder block consists of two main sub-layers:

1. **Multi-Head Self-Attention:** The input sequence attends to itself.
2. **Position-wise Feed-Forward Network (FFN):** A two-layer MLP applied independently to each position.

A residual connection is applied around each of the two sub-layers, followed by layer normalization. The output of a sub-layer is $\text{LayerNorm}(x + \text{Sublayer}(x))$.
Note: More recent architectures often use Pre-LN, where layer normalization is applied *before* the sub-layer: $x + \text{Sublayer}(\text{LayerNorm}(x))$. This can lead to more stable training.

The Decoder Block

Each decoder block has three sub-layers:

1. **Masked Multi-Head Self-Attention:** The decoder attends to the output sequence it has generated so far. Masking ensures it cannot see future tokens.
2. **Multi-Head Cross-Attention:** The queries come from the previous sub-layer, but the keys and values come from the output of the **final encoder block**. This is where the decoder incorporates information from the source sentence.
3. **Position-wise FFN:** Same as in the encoder.

Similar to the encoder, residual connections and layer normalization are used around each sub-layer.

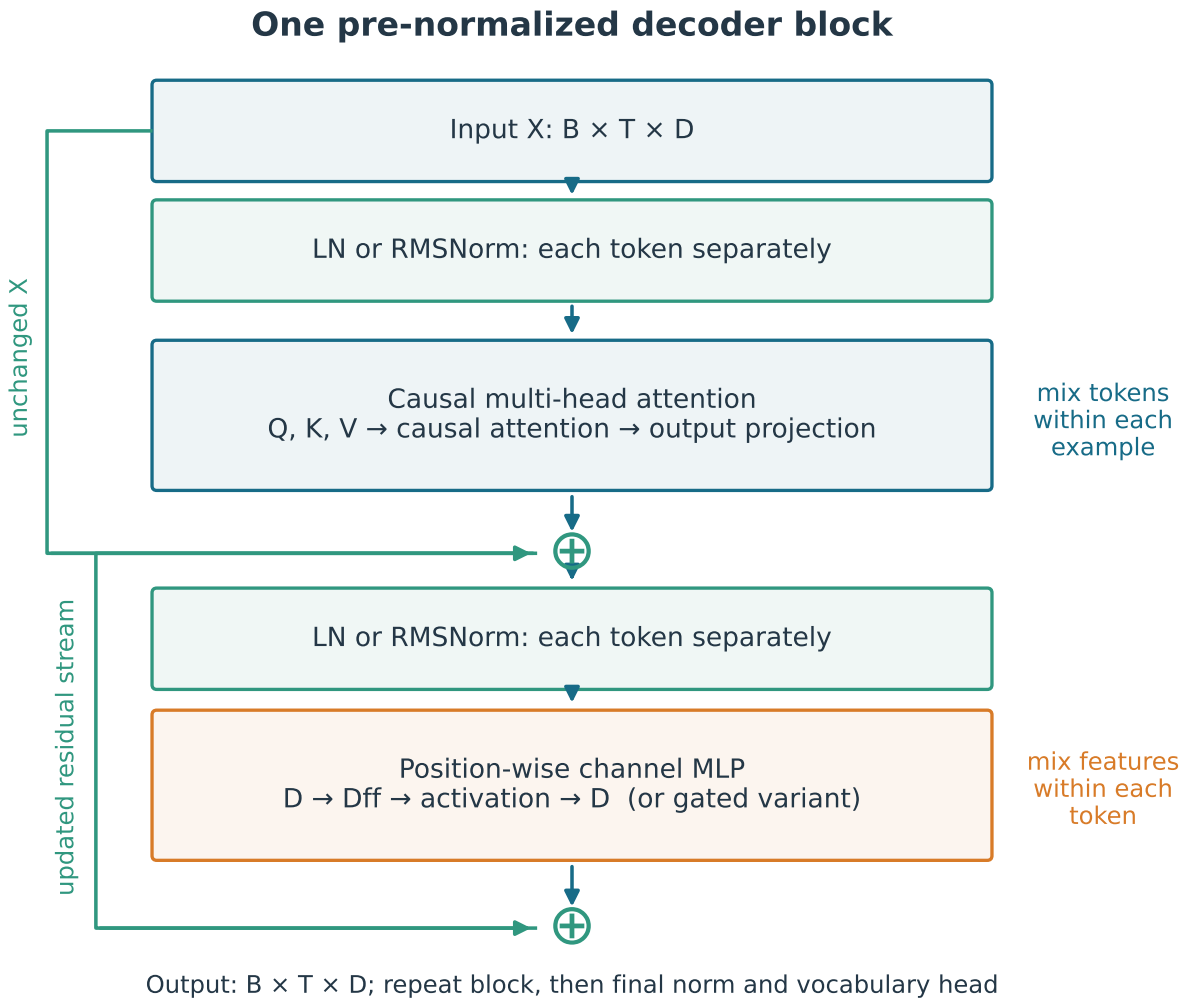


Figure 18.2: An expanded decoder block separates token mixing from feature mixing and shows both residual additions. The causal mask restricts each query's available keys; it does not delete the residual stream. This schematic uses pre-normalization and omits dropout. LN versus RMSNorm and the exact MLP are architecture choices.

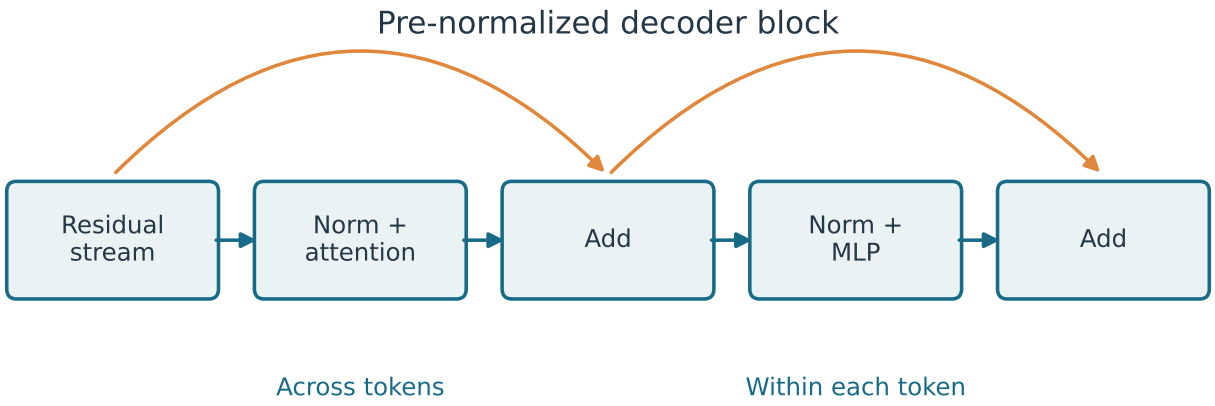


Figure 18.3: A decoder block alternates communication across tokens with nonlinear processing within each token. The residual stream supplies a direct path through both sublayers.

18.6 Embedding and Positional Encoding

Deeper explanation: Position aliasing and information supplied by a causal boundary (p. 424)

Input/Output Embeddings

- **Input:** The model receives input tokens as one-hot encoded vectors of vocabulary size (e.g., 32,000).
- **Input Embedding:** A learnable linear layer projects these high-dimensional one-hot vectors into a lower-dimensional, dense embedding space (e.g., dimension $d_{\text{model}} = 1024$). Since the input is one-hot, this operation is equivalent to a lookup in an embedding matrix. For good initialization, the columns of this matrix should have an RMS norm of approximately 1.
- **Output De-embedding:** After the final decoder block, another linear layer projects the model's output representation back to the vocabulary size to produce logits for each possible next token. A softmax is then applied to get probabilities.
  - **Initialization:** The input to this layer is normalized. To prevent the output logits from being too large (which would saturate the softmax), the rows of this "de-embedding" matrix should be initialized with small values. A standard approach is to initialize entries with a standard deviation of $1/\sqrt{d_{\text{model}}}$.

Worked derivation: one-hot embeddings have different fan-in statistics

Lecture basis: faculty handout 19, pp. 4–6. The steps below expand the lecture argument.

Step 1: an embedding selects a column. Use column convention $E \in \mathbb{R}^{d\times V}$, where V is vocabulary size. For token j, $x = e_j$ has one nonzero entry, so

$$h = Ee_j = E_{:,j}, \qquad h_i = E_{ij}.$$

If embedding entries have second moment s^2, then $\mathbb{E}[h_i^2] = s^2$, not Vs^2. There is no sum of V independent contributions; copying a dense fan-in variance $1/V$ would unnecessarily shrink the selected embedding.

Step 2: contrast the dense output projection. For an untied readout $z = Uh$, $U \in \mathbb{R}^{V\times d}$, each logit is $z_j = \sum_{i=1}^{d} U_{ji}h_i$. Under the same centered-weight independence assumptions as the initialization calculation,

$$\mathbb{E}[z_j^2] = d\,\mathrm{Var}(U_{ji})\mathbb{E}[h_i^2].$$

Thus order-one logits motivate readout variance of order $1/d$ when hidden coordinates have order-one second moment. This is a dense sum and has a different scaling calculation from column lookup.

Step 3: identify what parameter tying changes. If the readout is tied to $E^\top$, the same entries serve both roles and the independence assumptions may fail. One can introduce an explicit output scale, but the untied calculation does not automatically prove an appropriate tied-model initialization. Forward activation scale, backward gradients, and feature-update scale must all be checked when transferring width rules.

Positional Encoding

Unmasked self-attention without positional information is permutation-equivariant: permuting the input token rows permutes the output rows. It has no inherent token-position signal. A causal mask already introduces an order-dependent structure. Since word order is crucial in language, we must explicitly inject this information into the model.

Original Sinusoidal Encoding (Obsolete):

The original Transformer paper added fixed vectors to the input embeddings. These vectors were generated using sine and cosine functions of different frequencies, uniquely encoding each position.

No Positional Encoding (NOPE):

It was discovered that for many tasks, models without explicit positional encoding perform surprisingly well. It's hypothesized that the model can infer relative positions through other means (e.g., the causal mask in the decoder implies order).

Learned Relative Positional Encoding:

This approach modifies the attention score calculation directly. A learnable bias term b_{t-i} is added to the attention logit for a query at position t and a key at position i. The bias depends only on the relative distance $t - i$, and these biases are learned during training.

$$\text{score}(q_t, k_i) = \frac{\vec{q}_t^T \vec{k}_i}{\sqrt{d_k}} + b_{t-i}$$

Rotary Positional Encoding (RoPE):

The current state-of-the-art method. Instead of adding to the scores, RoPE modifies the queries and keys directly using rotation matrices.

- The embedding vector at each position is treated as a sequence of 2D vectors (or complex numbers).
- Each 2D component of the query and key vectors is rotated by an angle that is a function of its absolute position.
- A key property of rotations is that the inner product between two vectors rotated by angles θ_m and θ_n depends only on the relative angle difference, $\theta_m - \theta_n$.
- This elegantly injects relative position information into the attention score $\langle R_t \vec{q}, R_i \vec{k} \rangle$ without changing the norms of the vectors. This has been shown to be very effective.

Lecture 19

Advanced Transformer Architectures

Course sources: lecture_20; 2025-11-04 transcript. Lecture numbering follows the original reader; linked faculty handouts identify the overlapping source coverage.

19.1 Recap and Introduction

Deeper explanation: Cache correctness and generation boundaries (p. 427)

This lecture aims to deconstruct modern deep learning architectures, such as the Qwen model, so that every component is understood. We will build upon our previous discussion of positional encoding and explore various architectural innovations that have led to the current state-of-the-art in large language models.

19.2 Rotary Positional Encoding (RoPE)

Deeper explanation: Position aliasing and information supplied by a causal boundary (p. 424)

Attention mechanisms are permutation-invariant, meaning they do not inherently understand the order of tokens in a sequence. Positional encodings are introduced to solve this. We previously discussed several methods:

- **Absolute Positional Encoding:** Adding a unique vector to each token embedding based on its absolute position. This was found to be suboptimal.
- **NoPE (No Positional Encoding):** Simply not adding any positional information. This surprisingly outperformed absolute encodings in some cases.
- **Learned Relative Positional Encoding:** Modifying the attention score directly with a learned bias that depends on the relative distance between the query and key tokens. This was an improvement but has been superseded.

The current state-of-the-art is **RoPE (Rotary Positional Encoding)**.

The Core Idea of RoPE

The key idea behind RoPE is to modify the query vector $\mathbf{q}$ and key vector $\mathbf{k}$ based on their absolute positions, such that their inner product (which determines the attention score) depends only on their *relative* position.

The attention score is computed via an inner product, $\langle \mathbf{q}, \mathbf{k} \rangle$. We know that for two vectors, this can be expressed as:

$$\langle \mathbf{q}, \mathbf{k} \rangle = \|\mathbf{q}\|_2 \|\mathbf{k}\|_2 \cos(\theta)$$

where θ is the angle between them. RoPE leverages this by using positional information to *rotate* the query and key vectors. If we can modulate the angle between them based on their positions, we can encode positional information into the attention score.

RoPE in Two Dimensions

Consider a 2D case where $\mathbf{q}$ and $\mathbf{k}$ are two-dimensional vectors. A rotation by an angle ϕ can be achieved by multiplying with a rotation matrix:

$$\mathbf{R}_\phi = \begin{bmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{bmatrix}$$

Let the query be at position t and the key at position i. We apply rotations to them based on their positions:

- Query at position t is transformed to $\mathbf{q}'_t = \mathbf{R}_{\omega t}\mathbf{q}_t$.
- Key at position i is transformed to $\mathbf{k}'_i = \mathbf{R}_{\omega i}\mathbf{k}_i$.

The new inner product becomes:

$$\langle \mathbf{q}'_t, \mathbf{k}'_i \rangle = \langle \mathbf{R}_{\omega t}\mathbf{q}_t, \mathbf{R}_{\omega i}\mathbf{k}_i \rangle = (\mathbf{R}_{\omega t}\mathbf{q}_t)^T(\mathbf{R}_{\omega i}\mathbf{k}_i) = \mathbf{q}_t^T \mathbf{R}_{\omega t}^T \mathbf{R}_{\omega i}\mathbf{k}_i$$

Since rotation matrices are orthogonal, $\mathbf{R}_{\omega t}^T = \mathbf{R}_{-\omega t}$. The product of two rotation matrices is a rotation by the sum of their angles:

$$\mathbf{R}_{-\omega t}\mathbf{R}_{\omega i} = \mathbf{R}_{\omega(i-t)}$$

Thus, the inner product becomes:

$$\langle \mathbf{q}'_t, \mathbf{k}'_i \rangle = \mathbf{q}_t^T \mathbf{R}_{\omega(i-t)}\mathbf{k}_i$$

This shows that the final score depends only on the relative position, $i - t$.

Extending RoPE to Higher Dimensions

Transformer embedding dimensions are much larger than 2 (e.g., 1024, 4096). To apply this rotation idea to a d-dimensional vector, RoPE groups the vector's dimensions into pairs.

- A d-dimensional vector is treated as a concatenation of $d/2$ two-dimensional vectors (or complex numbers).
- A different rotation is applied to each 2D block.

This is implemented using a block diagonal matrix $\mathbf{M}_t$, where each block $\mathbf{M}_j$ on the diagonal is a 2D rotation matrix:

$$\mathbf{M}_t = \begin{bmatrix} \mathbf{M}_1 & & & \\ & \mathbf{M}_2 & & \\ & & \ddots & \\ & & & \mathbf{M}_{d/2} \end{bmatrix} \quad \text{where} \quad \mathbf{M}_j = \begin{bmatrix} \cos(\theta_j t) & -\sin(\theta_j t) \\ \sin(\theta_j t) & \cos(\theta_j t) \end{bmatrix}$$

The rotation frequencies θ_j are pre-selected and fixed; they are not learned parameters. They are chosen across a range of scales (often logarithmically spaced).

- **Low Frequencies:** For small θ_j, the angle $\theta_j t$ changes slowly with position t. These channels are less sensitive to distance and can capture long-range dependencies.
- **High Frequencies:** For large θ_j, the angle changes rapidly. These channels are very sensitive to small changes in position and capture local, short-range information.

The model learns which information to place in which frequency channels by learning the query and key weights, effectively specializing different parts of the embedding space for different positional scales.

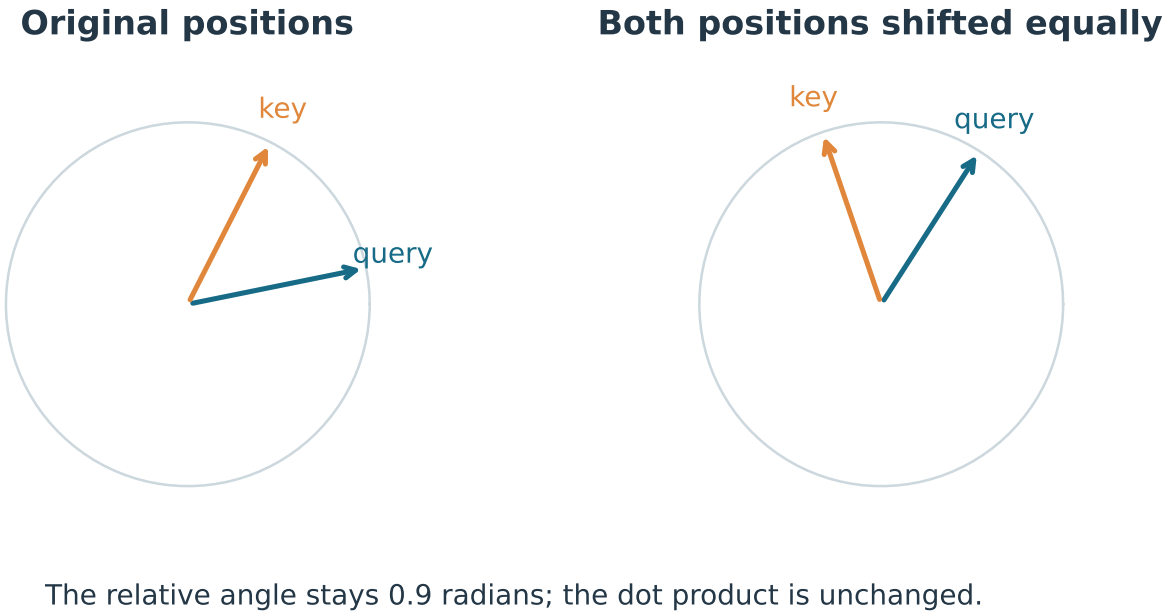


Figure 19.1: Rotating both vectors by their positions makes the dot product depend on their relative rotation. Different frequency pairs distinguish different distance scales.

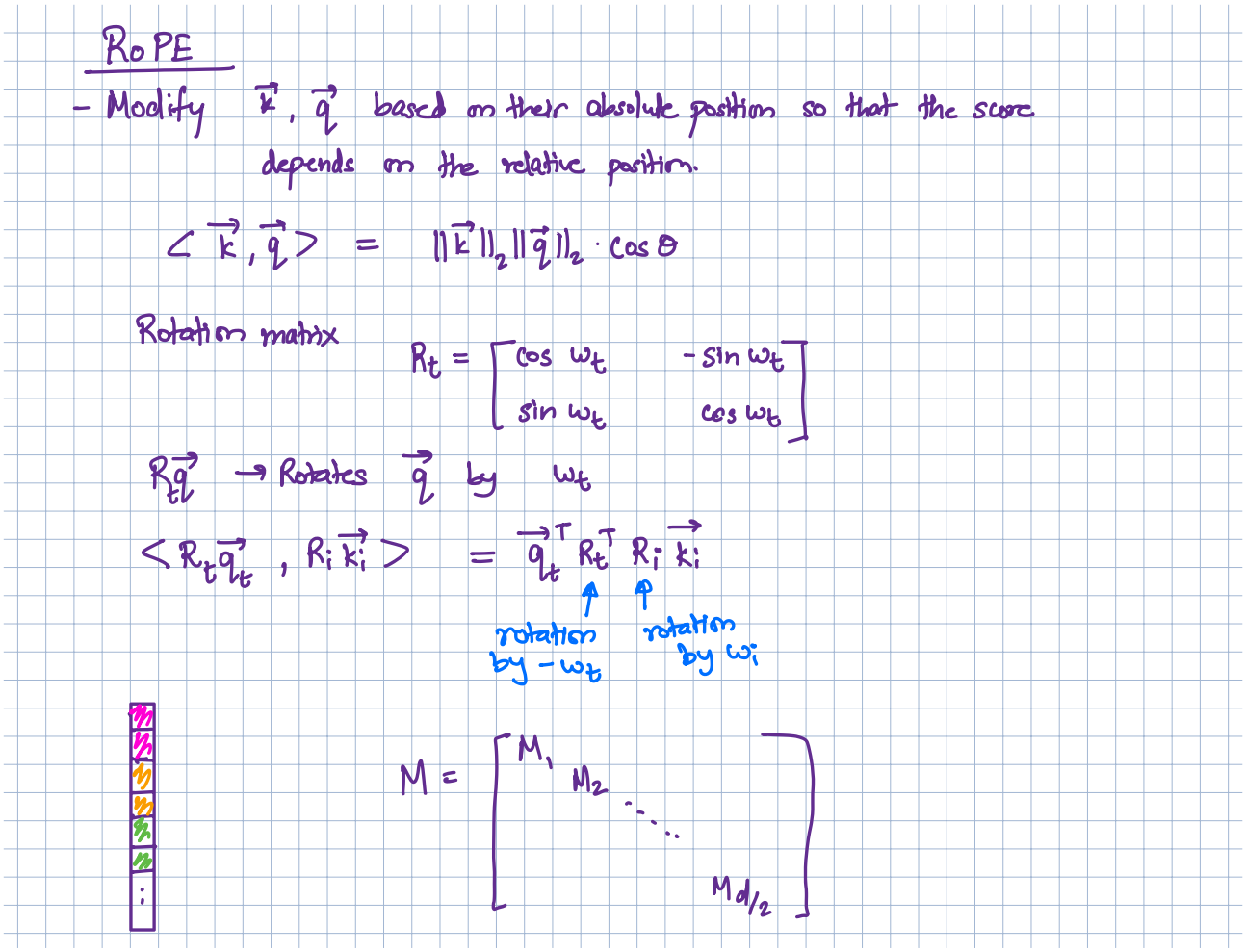


Figure 19.2: RoPE as a block-diagonal matrix of rotations. Each two-dimensional pair receives its own rotation frequency. The block matrix explains how the two-dimensional relative-position identity extends to a full feature vector. Original annotated source excerpt; Gireeja Ranade, faculty handout 20, p. 2.

RoPE rotates independent feature pairs

For four features, define $R(\alpha) = \left[\begin{smallmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{smallmatrix}\right]$ and

$$\mathcal{R}_t = \begin{bmatrix} R(t\omega_1) & 0_{2\times 2} \\ 0_{2\times 2} & R(t\omega_2) \end{bmatrix}, \qquad q = \begin{bmatrix} q^{(1)} \\ q^{(2)} \end{bmatrix}.$$

Then

$$(\mathcal{R}_t q)^\top (\mathcal{R}_s k) = \sum_{j=1}^{2} (q^{(j)})^\top R((s-t)\omega_j) k^{(j)},$$

because each 2×2 block satisfies $R(t\omega_j)^\top R(s\omega_j) = R((s-t)\omega_j)$. Off-diagonal zero blocks prevent mixing between frequency pairs. The relative displacement appears inside each pair, while the final scalar dot product adds their contributions. Different storage layouts may permute which coordinates form a pair without changing this underlying algebra.

Worked derivation: why rotations produce relative position

Lecture basis: faculty handout 20, pp. 2–3. The steps below expand the lecture argument.

Step 1: multiply the two-dimensional rotations explicitly. For $R(\theta) = \left[\begin{smallmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{smallmatrix}\right]$, $R(\theta)^\top = R(-\theta)$. The top-left entry of $R(-a)R(b)$ is $\cos a \cos b + \sin a \sin b = \cos(b-a)$; the top-right is $-\cos a \sin b + \sin a \cos b = -\sin(b-a)$. The other two entries similarly give

$$R(a)^\top R(b) = R(b-a).$$

Step 2: substitute into an attention score. Rotate a query at position t by $t\omega$ and a key at position s by $s\omega$:

$$\begin{aligned} \tilde{q}_t^\top \tilde{k}_s &= (R(t\omega)q_t)^\top R(s\omega)k_s \\ &= q_t^\top R(t\omega)^\top R(s\omega)k_s \\ &= q_t^\top R((s-t)\omega)k_s. \end{aligned}$$

The positional part depends on $s-t$. The content vectors can still depend on the entire context; this identity does not say that attention depends only on distance.

Step 3: assemble the feature-pair blocks. For even d_k, define the block matrix

$$R_t = \operatorname{diag}(R(t\omega_1), \dots, R(t\omega_{d_k/2})).$$

Then

$$R_t^\top R_s = \operatorname{diag}(R((s-t)\omega_1), \dots, R((s-t)\omega_{d_k/2})),$$

so the full score is the sum of pairwise relative-position scores. Orthogonality also gives $\|R_t q\| = \|q\|$. Rotating queries and keys changes their interaction while preserving their individual norms.

19.3 Key Transformer Architectures

Deeper explanation: Encoder, decoder, and encoder-decoder information sets (p. 425)

GPT-3: The Decoder-Only Model

The Generative Pre-trained Transformer (GPT) family represents decoder-only models, designed primarily for text generation.

- **Architecture:** It consists of a stack of transformer decoder blocks. Unlike the original transformer, there is no encoder and no cross-attention. Each block contains:

1. **Masked (Causal) Self-Attention:** This is crucial. To predict the next token, a token at position i can only attend to tokens at positions $j \leq i$. This prevents the model from "cheating" by looking at future tokens during training.
2. **Feed-Forward Network (FFN):** A standard multi-layer perceptron (MLP) that processes each position independently.
3. **Normalization and Residual Connections:** Used throughout the block for stable training.

- **Training:** GPT is trained on a next-token prediction task in an auto-regressive manner.
  - The input is a sequence of tokens, and the target is the same sequence shifted one position to the left.
  - For an input token at position i, the model is trained to predict the token at position $i + 1$.
  - The loss function is typically cross-entropy, calculated at every position in the sequence. This makes training very data-efficient, as every token contributes to the loss.
- **Inference (Generation):**
  - Generation is sequential. The model is given a starting prompt.
  - It predicts the next token, which is then appended to the input sequence.
  - This new, longer sequence is fed back into the model to generate the subsequent token, and so on.
  - **KV-Caching:** During this process, the computed keys and values for past tokens can be cached and reused, making inference much more efficient.
  - **Sampling Strategies:** To select a token from the output probability distribution, common methods include:
    - **Top-k Sampling:** Sample only from the k most probable tokens to avoid generating very rare or nonsensical words.
    - **Beam Search:** Instead of greedily picking the best next token, maintain a "beam" of k most probable sequences at each step. This allows the model to explore different generation paths and potentially find a globally more probable sequence.
- **Scale of GPT-3 (Largest Version):**
  - **Parameters:** $\sim$175 billion
  - **Layers:** 96
  - **Attention Heads:** 96
  - **Embedding Dimension:** 12,288

– **Context Length:** 2048 tokens

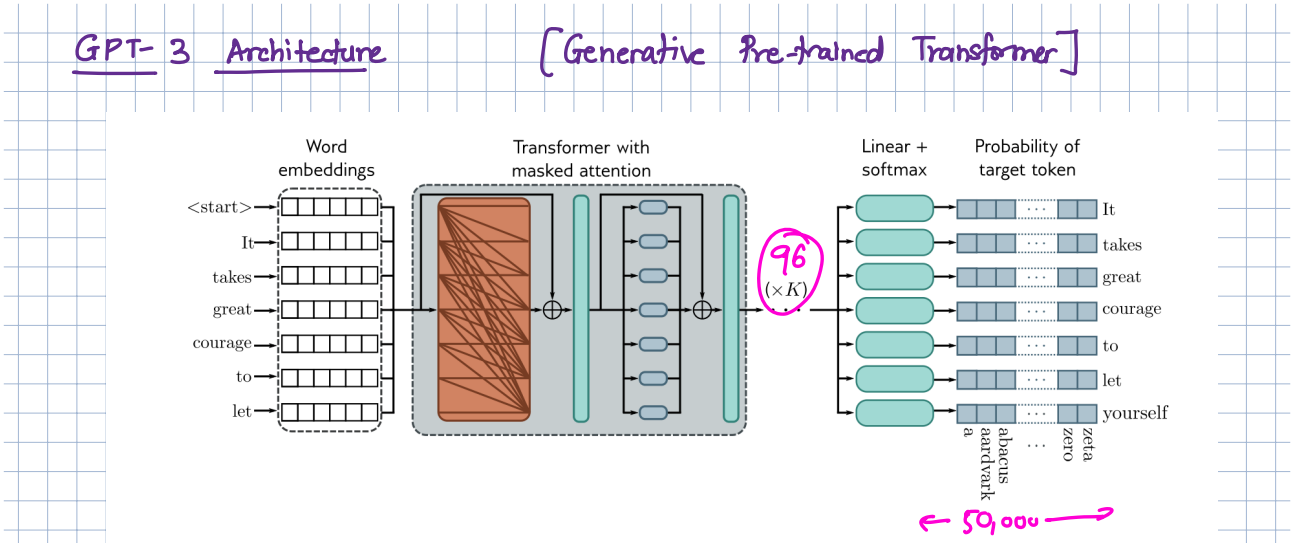


Figure 12.12 Training GPT3-type decoder network. The tokens are mapped to word embeddings with a special <start> token at the beginning of the sequence. The embeddings are passed through a series of transformer layers that use masked self-attention. Here, each position in the sentence can only attend to its own embedding and those of tokens earlier in the sequence (orange connections). The goal at each position is to maximize the probability of the following ground truth token in the sequence. In other words, at position one, we want to maximize the probability of the token It; at position two, we want to maximize the probability of the token takes; and so on. Masked self-attention ensures the system cannot cheat by looking at subsequent inputs. The autoregressive task has the advantage of making efficient use of the data since every word contributes a term to the loss function. However, it only exploits the left context of each word.

Figure 19.3: GPT-style input, decoder, and output wiring. The published diagram identifies which embeddings enter the decoder and how the output distribution is formed. It complements the encoder-decoder architecture by showing the removed encoder and cross-attention routes. Original annotated source excerpt; Gireeja Ranade, faculty handout 20, p. 4.

BERT: The Encoder-Only Model

BERT (Bidirectional Encoder Representations from Transformers) is an encoder-only model designed to learn deep bidirectional representations of language. It is not for generation but for creating powerful embeddings for downstream tasks.

- **Architecture:** A stack of transformer encoder blocks. Crucially, it uses **full (non-causal) self-attention**, allowing every token to attend to every other token in the sequence (past and future).
- **Pre-training Tasks:** BERT is pre-trained on two self-supervised tasks:
  1. **Masked Language Modeling (MLM):**

  - About 15% of the input tokens are randomly replaced with a special ‘[MASK]‘ token.
  - The model's objective is to predict the original identity of these masked tokens based on the surrounding unmasked context. This forces the model to learn a deep understanding of linguistic context.

2. **Next Sentence Prediction (NSP):**
  - The model is given two sentences, A and B, separated by a ‘[SEP]‘ token.
  - The task is to predict whether sentence B is the actual sentence that follows A in the original text, or if it is a random sentence from the corpus.
  - This was intended to help the model understand sentence relationships, but later research found it to be less impactful, and many modern encoder models have dropped it.

- **Fine-tuning:** After pre-training, BERT can be fine-tuned for various downstream tasks. This typically involves adding a small, task-specific output layer on top of the pre-trained BERT encoder.
  - **Sentence Classification (e.g., Sentiment Analysis):** The final hidden state corresponding to the special ‘[CLS]‘ token (prepended to every input) is fed into a classifier. The model is fine-tuned to predict a label for the entire sentence (e.g., positive/negative review).
  - **Token Classification (e.g., Named Entity Recognition):** The final hidden state for each token is fed into a classifier to predict a label for that specific token (e.g., person, location, organization).

BERT - Encoder only model.

Bidirectional Encoder Representations from Transformers.

Goal: Learn general information about the statistics of Language

Use fine-tuning to adapt to solve specific task.

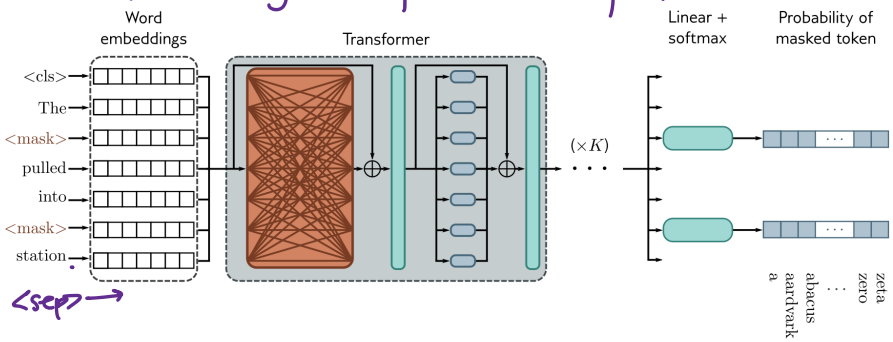


Figure 12.10 Pre-training for BERT-like encoder. The input tokens (and a special <cls> token denoting the start of the sequence) are converted to word embeddings. Here, these are represented as rows rather than columns, so the box labeled "word embeddings" is $\mathbf{X}^T$. These embeddings are passed through a series of transformer layers (orange connections indicate that every token attends to every other token in these layers) to create a set of output embeddings. A small fraction of the input tokens are randomly replaced with a generic <mask> token. In pre-training, the goal is to predict the missing word from the associated output embedding. To this end, the outputs corresponding to the masked tokens are passed through softmax functions, and a multiclass classification loss (section 5.24) is applied to each. This task has the advantage that it uses both the left and right context to predict the missing word but has the disadvantage that it does not make efficient use of data; here, seven tokens need to be processed to add two terms to the loss function.

Figure 19.4: BERT masked-token pretraining. Bidirectional context supports predicting selected masked positions. The special tokens and loss locations in the published diagram differ from causal next-token training. Original annotated source excerpt; Gireeja Ranade, faculty handout 20, p. 7.

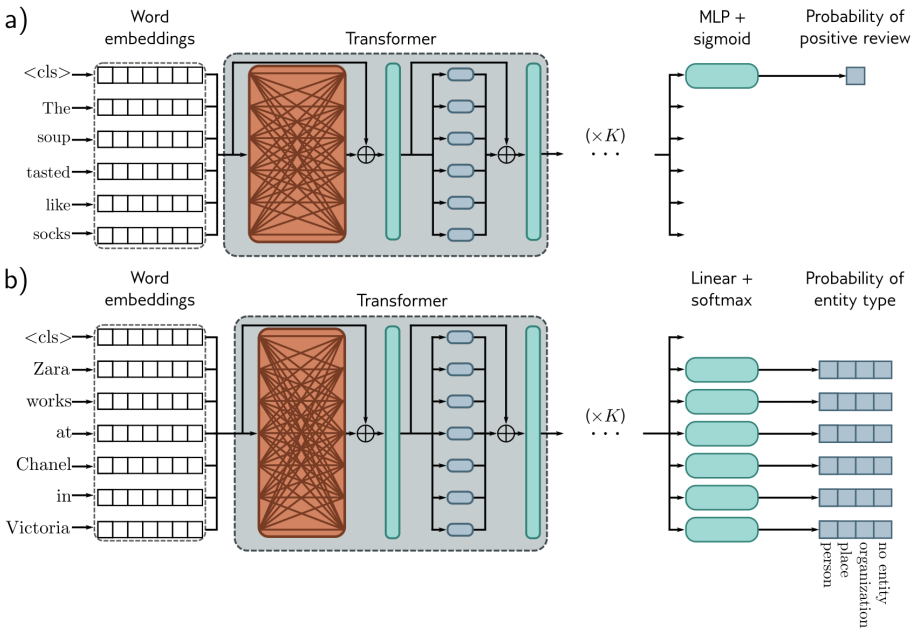


Figure 12.11 After pre-training, the encoder is fine-tuned using manually labeled data to solve a particular task. Usually, a linear transformation or a multi-layer perceptron (MLP) is appended to the encoder to produce whatever output is required. a) Example text classification task. In this sentiment classification task, the <cls> token embedding is used to predict the probability that the review is positive. b) Example word classification task. In this named entity recognition problem, the embedding for each word is used to predict whether the word corresponds to a person, place, or organization, or is not an entity.

Figure 19.5: Sequence-level and token-level heads on BERT. A sequence classifier reads an aggregate representation, whereas a token task attaches predictions to token positions. The backbone can be shared while the supervised readout and loss change. Published panels are retained from the handout. Original annotated source excerpt; Gireeja Ranade, faculty handout 20, p. 9.

19.4 Modern Architectural Components

Deeper explanation: Parameter count versus active computation (p. 425)

Deeper explanation: The KV cache and the cost of generation (p. 421)

Gated Linear Units (GLU)

The standard Feed-Forward Network (FFN) in a transformer block is often replaced by a Gated Linear Unit (GLU) or its variants.

- **Mechanism:** The input $\mathbf{x}$ is passed through two parallel linear layers. The output of one layer is passed through a gating activation function (like sigmoid), and the result is element-wise multiplied with the output of the other linear layer.

$$\text{GLU}(\mathbf{x}) = (\mathbf{x}\mathbf{W} + \mathbf{b}) \odot \sigma(\mathbf{x}\mathbf{V} + \mathbf{c})$$

  Here, $\odot$ denotes element-wise multiplication, and σ is the sigmoid function. The second term acts as a "gate," controlling which information from the first term passes through.

- **SwiGLU:** A popular and effective variant that uses the Swish activation function as the gate. Swish is defined as:

$$\text{Swish}_\beta(x) = x \cdot \sigma(\beta x)$$

  where β can be a learned parameter. The SwiGLU operation is:

$$\text{SwiGLU}(\mathbf{x}) = (\mathbf{x}\mathbf{W} + \mathbf{b}) \odot \text{Swish}(\mathbf{x}\mathbf{V} + \mathbf{c})$$

- **Intuition:** GLUs have been empirically shown to improve performance. While a full theoretical explanation is elusive (leading to the famous "divine benevolence" quote from a paper by Shazeer), one intuition is that they can more easily model multiplicative interactions and approximate piecewise quadratic functions, offering more expressive power than the piecewise linear ReLU.

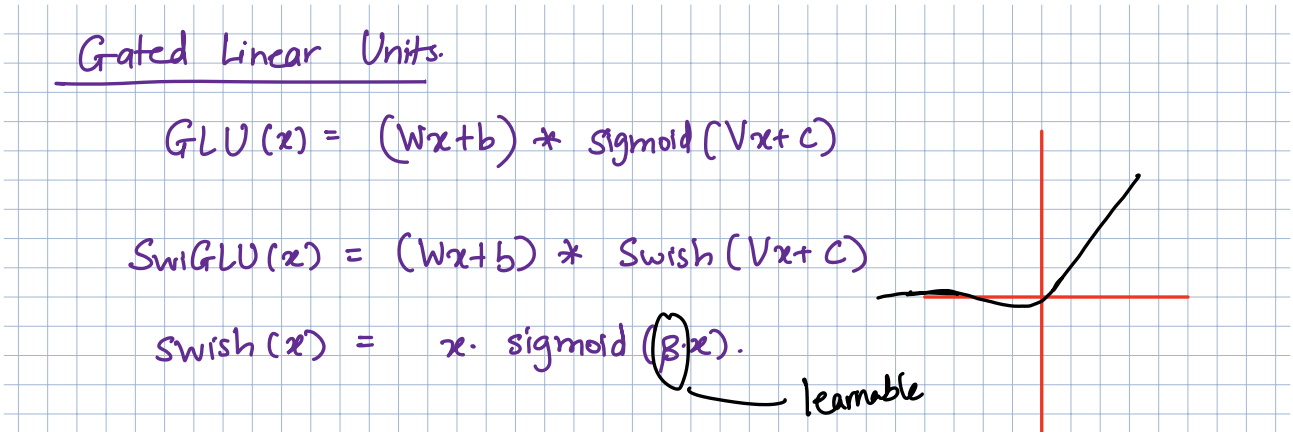


Figure 19.6: GLU, SwiGLU, and the smooth gate. Read the multiplication as an elementwise interaction between two projected branches. The Swish curve is smooth and can take small negative values, unlike ReLU; the gate therefore changes both shape and gradient flow. Original annotated source excerpt; Gireeja Ranade, faculty handout 20, p. 10.

Mixture of Experts (MoE)

Mixture of Experts (MoE) is a technique to increase model capacity without a proportional increase in computational cost for each forward pass.

- **Motivation:** Standard transformer models are "dense," meaning all parameters are used for processing every token. This can be computationally expensive.
- **Core Idea:** Replace the single large FFN in a transformer block with a set of smaller FFNs, called "experts." For each token, a lightweight "router" network decides which expert(s) should process it.

$$\text{Token} \xrightarrow{\text{Router}} \text{Selects Top-K Experts} \rightarrow [\text{Expert}_1, \text{Expert}_2, \dots] \xrightarrow{\text{Combine}} \text{Output}$$

- **Mechanism:**
  1. The router is typically a small linear layer that outputs a probability distribution over the available experts for a given token.
  2. In modern MoE models, the router selects the top-K experts (e.g., K=2 or 4) with the highest probabilities.
  3. The token is passed through each of these K experts.
  4. The final output is a weighted sum of the outputs from the selected experts, where the weights are the normalized probabilities from the router.
- **Advantages:** This allows for a massive increase in the total number of parameters in the model, while keeping the number of active parameters (and thus the computational cost) for any single token constant. This leads to the concept of "sparse activation."
- **Challenges:** Training MoE models can be complex due to challenges like load balancing (ensuring all experts receive a similar amount of work) and training stability.

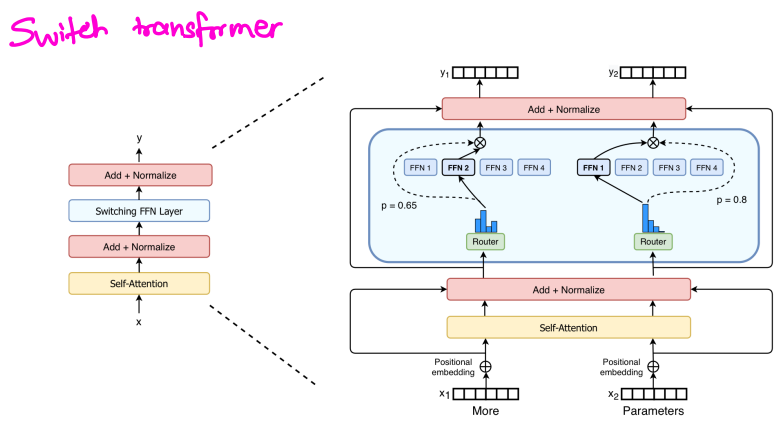


Figure 2: Illustration of a Switch Transformer encoder block. We replace the dense feed forward network (FFN) layer present in the Transformer with a sparse Switch FFN layer (light blue). The layer operates independently on the tokens in the sequence. We diagram two tokens (x_1 = "More" and x_2 = "Parameters" below) being routed (solid lines) across four FFN experts, where the router independently routes each token. The switch FFN layer returns the output of the selected FFN multiplied by the router gate value (dotted-line).

Fedus et al 2021.

Figure 19.7: Switch Transformer routes tokens to experts. The router selects feedforward computation for each token. Sparse activation changes computation per token without making the stored expert parameters disappear. The handout credits Fedus and collaborators (2021). Original annotated source excerpt; Gireeja Ranade, faculty handout 20, p. 11.

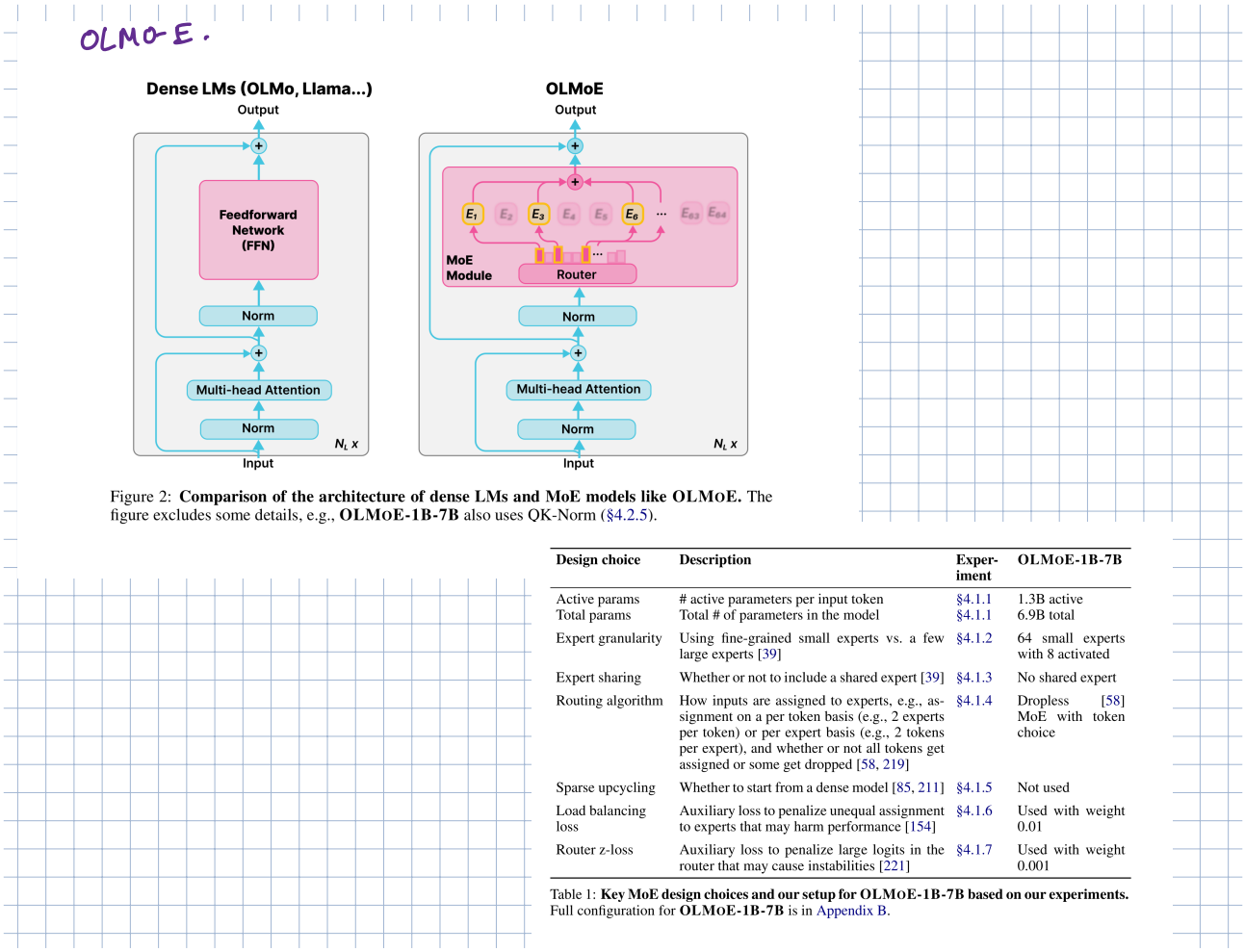


Figure 2: **Comparison of the architecture of dense LMs and MoE models like OLMoE.** The figure excludes some details, e.g., **OLMoE-1B-7B** also uses QK-Norm (§4.2.5).

Design choice	Description	Experiment	OLMoE-1B-7B
Active params	# active parameters per input token	§4.1.1	1.3B active
Total params	Total # of parameters in the model	§4.1.1	6.9B total
Expert granularity	Using fine-grained small experts vs. a few large experts [39]	§4.1.2	64 small experts with 8 activated
Expert sharing	Whether or not to include a shared expert [39]	§4.1.3	No shared expert
Routing algorithm	How inputs are assigned to experts, e.g., assignment on a per token basis (e.g., 2 experts per token) or per expert basis (e.g., 2 tokens per expert), and whether or not all tokens get assigned or some get dropped [58, 219]	§4.1.4	Dropless [58] MoE with token choice
Sparse upcycling	Whether to start from a dense model [85, 211]	§4.1.5	Not used
Load balancing loss	Auxiliary loss to penalize unequal assignment to experts that may harm performance [154]	§4.1.6	Used with weight 0.01
Router z-loss	Auxiliary loss to penalize large logits in the router that may cause instabilities [221]	§4.1.7	Used with weight 0.001

Table 1: **Key MoE design choices and our setup for OLMoE-1B-7B based on our experiments.** Full configuration for **OLMoE-1B-7B** is in Appendix B.

Figure 19.8: Dense versus mixture-of-experts blocks. The published OLMoE comparison locates the router and expert bank inside a familiar Transformer residual block. The configuration table is a historical example, not a universal recipe. Original annotated source excerpt; Gireeja Ranade, faculty handout 20, p. 12.

19.5 Putting It All Together: A Modern Architecture

Deeper explanation: Parameter count versus active computation (p. 425)

Modern LLM architectures like Qwen, Llama 3, and Gemini are not monolithic but are compositions of the advanced components discussed. A typical modern block might feature:

- **Pre-Normalization:** Normalization layers (e.g., RMSNorm) are applied *before* the attention and FFN sub-layers, which has been found to improve training stability.
- **Rotary Positional Encoding (RoPE):** This is the standard for positional information. Some models use variants like *Partial RoPE*, where RoPE is only applied to a fraction of the embedding dimensions.

- **Gated Linear Units (GLU):** SwiGLU is commonly used in place of the standard ReLU-based FFN.
- **Mixture of Experts (MoE):** Increasingly, dense FFN layers are being replaced with MoE layers to scale up model size efficiently.
- **Hybrid Structures:** Some architectures, like Qwen, are hybrids, mixing standard attention layers with other sequence-modeling mechanisms like state-space models (e.g., Mamba, Gated Delta Net) within the same network.

By understanding these individual components, we can deconstruct and comprehend the design principles behind the latest and most powerful large language models. The field continues to evolve, with new combinations and variations of these fundamental building blocks constantly being explored.

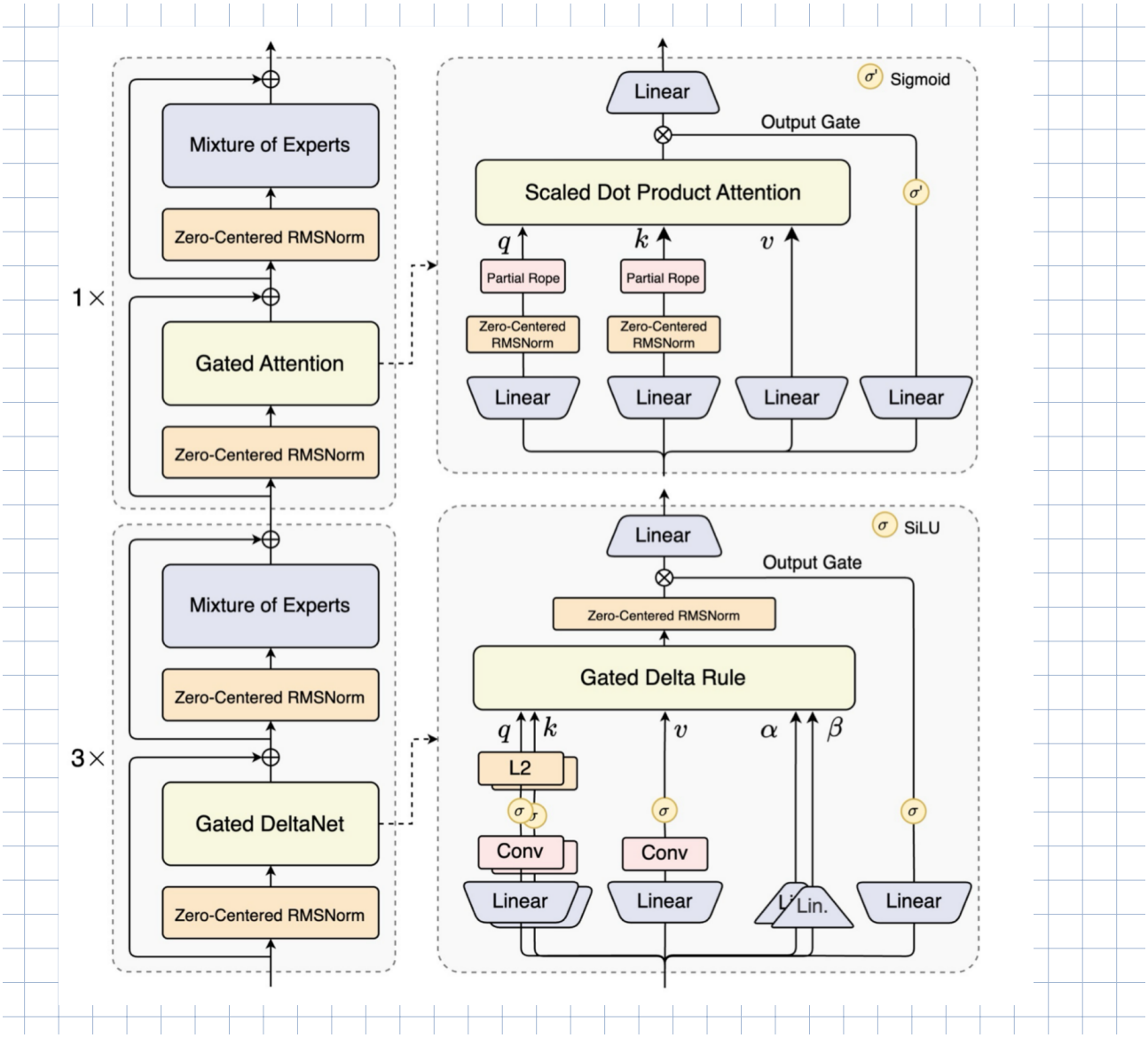


Figure 19.9: A hybrid attention and gated-recurrence architecture. The hybrid alternates different sequence-mixing mechanisms. Trace the attention and recurrent branches separately before following their shared residual stack. This published design is reproduced as a Fall 2025 course example. Original annotated source excerpt; Gireeja Ranade, faculty handout 16, p. 13.

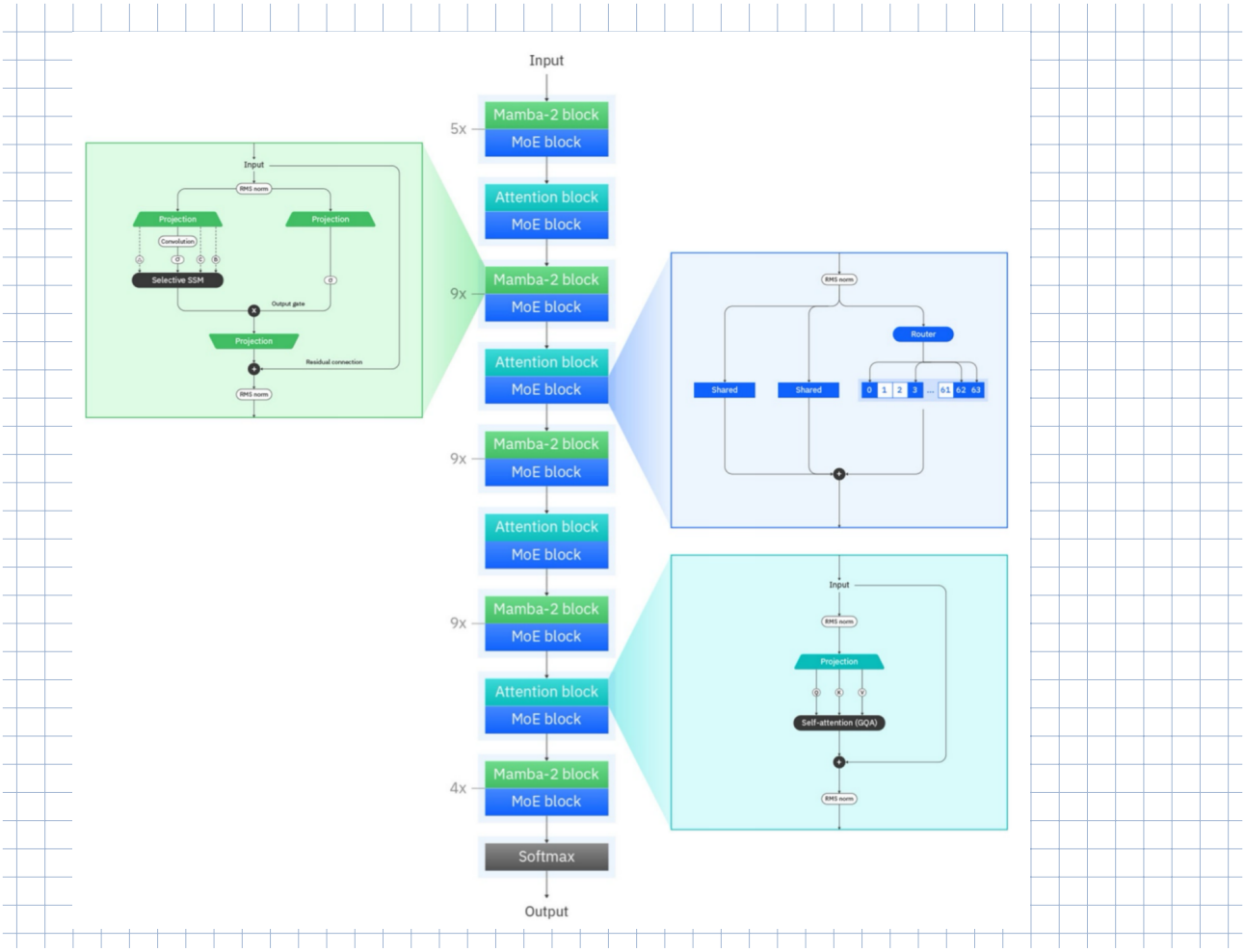


Figure 19.10: Combining state-space layers, attention, and experts. The full stack shows that the choice of sequence mixer and the choice of feedforward expert routing are separate design decisions. The published architecture is retained from the handout, including its model-specific arrangement. Original annotated source excerpt; Gireeja Ranade, faculty handout 16, p. 14.

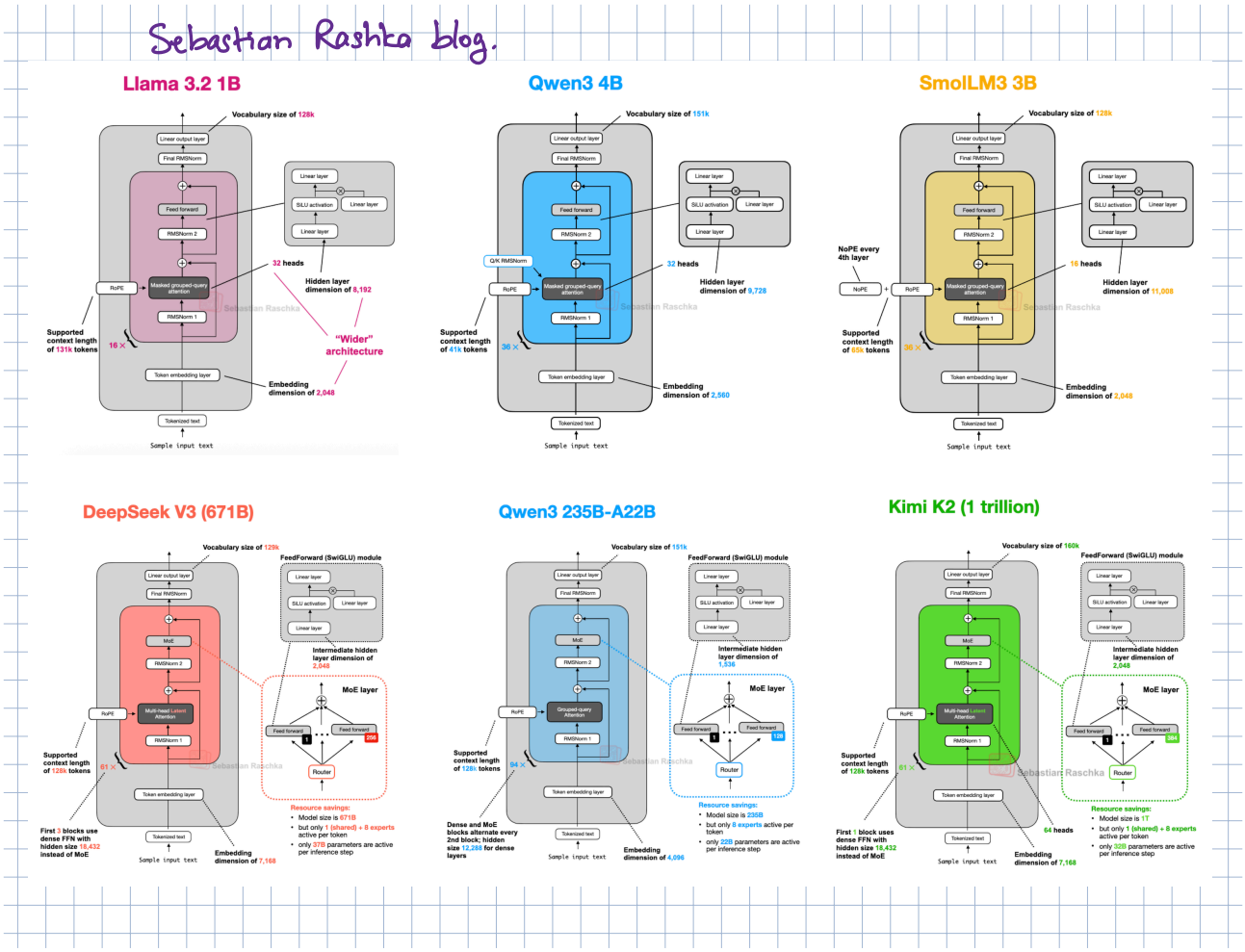


Figure 19.11: Comparing complete language-model architectures. The lecture's architecture atlas, credited there to Sebastian Raschka, exposes differences in attention, normalization, positional encoding, and experts. Model names and sizes describe the Fall 2025 handout snapshot. Original annotated source excerpt; Gireeja Ranade, faculty handout 20, p. 14.

Lecture 20

In-Context Learning, Prompting, and Fine-Tuning

Course sources: lecture_21; 2025-11-06 transcript. Lecture numbering follows the original reader; linked faculty handouts identify the overlapping source coverage.

20.1 Recap of Large Language Model Mechanics

Deeper explanation: A complete decoder training example (p. 422)

Pre-training GPT-Style Models

The pre-training process for Generative Pre-trained Transformer (GPT) style models is foundational. The objective is to learn general patterns and knowledge from a vast corpus of text data.

Architecture Overview The model architecture can be broken down into several key components, all of which contain learnable parameters:

1. **Tokenizer**: Converts raw text into a sequence of discrete tokens from a fixed vocabulary (e.g., 32,000 tokens).
2. **Embedder**: A learnable lookup table (a linear layer) that maps each discrete token ID to a high-dimensional continuous vector (the embedding). For example, it might convert a 32,000-dimensional one-hot vector into a 4,096-dimensional embedding.
3. **Transformer Layers**: A stack of identical blocks, each typically containing:
   - Causal Self-Attention mechanisms.
   - Multi-Layer Perceptrons (MLPs).
   - Residual connections and layer normalization.
   - Variations can include State Space Models (SSMs) or Mixture of Experts (MoE) layers.

   A critical feature is that the weights of these layers are **shared across all time steps** (positions in the sequence). Interactions across time are handled by the attention mechanism or by state-carrying components in SSMs.

4. **Dembedder** (or Unembedder): A final linear layer that projects the output embeddings from the last transformer layer back into the vocabulary space. It maps the 4,096-dimensional vector back to a 32,000-dimensional vector of logits (scores).
5. **Softmax**: Converts the logits into a probability distribution over the entire vocabulary for the next token.

Training Objective: Next Token Prediction The model is trained on the task of predicting the next token in a sequence.

- **Loss Function**: The standard loss function is the **cross-entropy loss**, which compares the model's predicted probability distribution with the actual next token (the ground truth).
- **Teacher Forcing**: This is the training strategy used. The model is fed the entire ground-truth sequence as input, and at each position t, it is trained to predict the token at position $t + 1$. The model's own predictions are not fed back during training; it is always "forced" to condition on the correct preceding sequence. This allows for massive parallelization during training.
- **Example**: Given the start of the poem "Jabberwocky": *'Twas brillig, and the slithy toves...*
  - After processing the '<start>' token, the model's target is ''Twas'.
  - After processing '<start> 'Twas', the target is 'brillig'.
  - After processing '<start> 'Twas brillig', the target is ','.
  - ...and so on, until the end of the document, where the target is an '<end-of-sequence>' token.

The quality and mixture of the pre-training data are crucial. Recent work, such as the paper "Textbooks Are All You Need," has shown that high-quality, and even synthetically generated, data can significantly improve model performance.

Inference: Autoregressive Generation

At inference time, the model is used to generate new text in an autoregressive (autocomplete) fashion.

1. **Prompt**: The process begins with a user-provided sequence of tokens called a **prompt**.
2. **Context Processing**: The model processes the entire prompt in one forward pass to compute the internal states (i.e., the key-value cache for all attention layers). No sampling occurs during this phase.
3. **First Token Generation**: Using the state computed from the last token of the prompt, the model generates a probability distribution for the next token.

4. **Sampling**: A token is chosen from this distribution using a sampling strategy (e.g., top-k, nucleus sampling). The **temperature** parameter of the softmax function is a key knob here.

$$P(token_i) = \frac{e^{\text{logit}_i/T}}{\sum_j e^{\text{logit}_j/T}}$$

   - A high temperature ($T > 1$) makes the distribution more uniform, leading to more random and creative outputs.
   - A low temperature ($T \to 0$) makes the distribution sharper, approaching a greedy selection of the most likely token (argmax).

5. **Iteration**: The sampled token is appended to the sequence, and the model performs another forward pass (incrementally, using the cached state) to generate the next token. This loop continues until a special '<stop>' token is generated or a maximum length is reached.

20.2 A Brief History of Prompting and In-Context Learning

The way we interact with and utilize large language models has evolved significantly.

- **2017: Transformer**: The "Attention Is All You Need" paper introduced the transformer architecture, laying the groundwork for modern LLMs.
- **2018: BERT and GPT-1**:
  - **BERT** (encoder-only) revolutionized NLP by showing that pre-trained models could produce powerful contextual embeddings, which could be fine-tuned for downstream tasks, leading to state-of-the-art results across the board.
  - **GPT-1** (decoder-only) demonstrated that next-token prediction pre-training also produced high-quality embeddings, positioning it as an alternative to BERT for feature extraction.
- **2019: GPT-2**: By significantly scaling up the model size, GPT-2 demonstrated surprising **emergent abilities**.
  - **Coherent Text Generation**: It could generate shockingly natural prose, poetry, and stories that maintained style and context.
  - **Zero-Shot Task Performance**: The model could perform tasks it was not explicitly trained for, simply by being prompted correctly. This showed it had absorbed factual knowledge.
    * **Example Prompt**: 'The capital of France is'
    * **Completion**: 'Paris'

- **2020: GPT-3**: The paper "Language Models are Few-Shot Learners" introduced the concept of **In-Context Learning (ICL)**.
  - **Concept**: By providing a few examples of a task within the prompt itself, the model can learn to perform the task for a new input without any weight updates.
  - **Example**: The model can infer the desired relationship (gemstones vs. fruits) from the examples.

```
Prompt 1:
red: Ruby; blue:

Completion:
Sapphire

---
Prompt 2:
red: raspberry; blue:

Completion:
blueberry
```

  - **Significance**: ICL allows solving ML problems with just a few examples, no gradient descent, and no fine-tuning, provided the examples fit within the model's context window. This sparked the field of "prompt engineering."

20.3 Instruction Following and Supervised Fine-Tuning (SFT)

Deeper explanation: Differentiate a masked response loss through the prompt (p. 429)

While ICL is powerful, a natural next step was to make models more directly useful by having them follow explicit instructions.

The Rise of Instruction Tuning

- Pre-trained models like GPT-3, when prompted with an instruction like "Write me a poem about a cat," would often complete the instruction rather than execute it (e.g., generating "...and make it rhyme.").
- **InstructGPT (2021)**: This pivotal paper from OpenAI showed how to align a model to follow user instructions. It used a combination of techniques, most notably **Reinforcement Learning from Human Feedback (RLHF)**, to train a model to be a helpful and harmless assistant. This research directly led to the development of ChatGPT.

Mechanism: Supervised Fine-Tuning (SFT)

A core component of instruction tuning, and a simpler alternative to full RLHF, is Supervised Fine-Tuning. SFT adapts a pre-trained model to the instruction-following format.

- **Data**: SFT requires a dataset of high-quality instruction-response pairs. (e.g., {prompt: "Translate 'hello' to French", completion: "bonjour"}).
- **Process**: The training process is similar to pre-training but with a crucial difference in how the loss is calculated.
    1. The model is given the instruction (prompt) as input.
    2. It is then trained to generate the desired response using teacher forcing.
    3. **Masked Loss**: The cross-entropy loss is calculated **only on the response tokens**. No loss is backpropagated from the prompt tokens.
- **Why Mask the Loss?**: The goal is to teach the model how to *respond* to instructions, not how to *generate* them. Training on the entire sequence could lead to confusing behaviors, like the model asking a follow-up question instead of providing an answer.

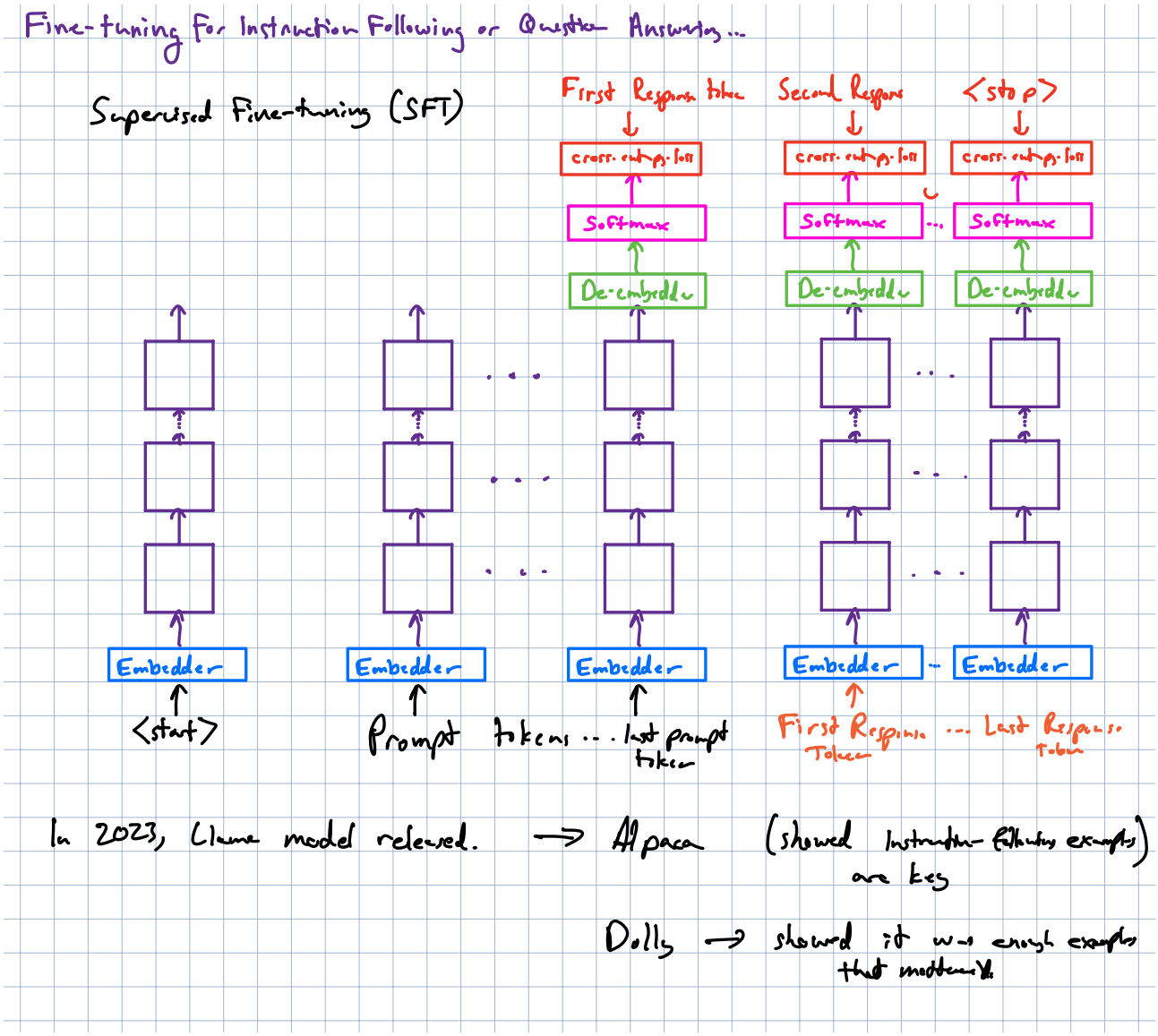


Figure 20.1: The response-only SFT training scaffold. The diagram places losses on response positions while retaining the question as context. No direct loss on question tokens does not mean no gradient through the computations that use those tokens. Original annotated source excerpt; Anant Sahai, faculty handout 21, p. 4.

The Democratization of SFT: The Llama/Alpaca Story (2023)

Initially, instruction tuning was seen as an expensive process requiring massive human annotation effort, accessible only to large labs.

- **LLaMA Release**: Meta released the LLaMA family of models, providing a powerful open-source pre-trained base model.
- **Alpaca (Stanford)**: Researchers hypothesized that the key to instruction following was not the complex RLHF process, but simply having enough high-quality examples.

  - They created a dataset by scraping 52,000 instruction-response pairs generated by OpenAI's powerful models (e.g., GPT-3.5).
  - They performed SFT on the base LLaMA model using this synthetic dataset.
  - **Result**: The resulting model, Alpaca, exhibited impressive instruction-following capabilities, demonstrating that SFT on high-quality synthetic data was remarkably effective. This was a paradigm shift.

- **Dolly (Databricks)**: A follow-up question was whether the success was due to the instruction format or some latent property ("GPT-ism") of the synthetic data. Furthermore, using OpenAI's output to train a competing model raised licensing concerns.
  - Databricks employees created a smaller, fully human-generated dataset of 15,000 instruction-following examples.
  - They fine-tuned an open model on this dataset.
  - **Result**: The model also acquired strong instruction-following abilities, proving that the **distribution of data** was the key. Pre-training corpora lack sufficient instruction-response examples, and SFT remedies this distributional mismatch.

20.4 Parameter-Efficient Fine-Tuning (PEFT)

Deeper explanation: Rank, placement, and merging are separate choices (p. 432)

Full fine-tuning updates all model parameters (billions of them), which is computationally expensive and requires storing a full model copy for each task. PEFT methods offer a compromise, achieving strong performance by tuning only a tiny fraction of the parameters.

The Spectrum of Model Adaptation

We can think of methods for adapting a pre-trained model on a spectrum:

0. **Feature Extraction (Linear Probing)**: Freeze the entire model and train a small prediction head (e.g., a linear classifier) on top of its embeddings. (0% of parameters tuned).
1. **Full Fine-Tuning**: Update all weights in the model via SFT. (100% of parameters tuned).
2. **Pure Prompting / In-Context Learning**: Update no weights at all; all task specification happens in the prompt.

PEFT methods explore the space between (0) and (1).

Soft Prompting (Prompt Tuning)

Instead of manually engineering a discrete text prompt, soft prompting learns an optimal prompt directly in the continuous embedding space.

- **Problem**: Gradients cannot flow back to discrete tokens to optimize a prompt.
- **Solution**: Freeze the entire pre-trained model and learn a sequence of continuous embedding vectors (a "soft prompt") that are prepended to the input.
- **Mechanism**:
  1. A fixed number of "special prompt tokens" are designated (e.g., 100).
  2. These tokens are associated with learnable embedding vectors, which are the **only trainable parameters**.
  3. For each training example, these learnable prompt embeddings are prepended to the (frozen) embeddings of the actual input.
  4. The model processes this combined sequence, and the loss on the response is back-propagated to update *only* the soft prompt embeddings.
- **Parameter Count**: For a model with embedding dimension D_{emb} and a soft prompt of length L_p, the number of trainable parameters is just $L_p \times D_{emb}$.
  - **Example**: For an 8B parameter model with $D_{emb} = 4096$ and $L_p = 100$, we only train $100 \times 4096 \approx 400,000$ parameters, which is $\sim$0.005% of the total.
- **Performance**: Empirically, as the base model size increases, the performance of prompt tuning approaches that of full fine-tuning.

Prefix Tuning

Prefix tuning is a more expressive extension of soft prompting. The insight is that the prompt influences downstream computations via the key-value (KV) states in the attention layers. Prefix tuning learns these intermediate states directly.

- **Mechanism**:
  1. Freeze the entire pre-trained model.
  2. For each transformer layer, introduce a small set of learnable "prefix" key and value vectors.
  3. During the forward pass, for each layer, these learnable KV vectors are prepended to the actual keys and values computed from the input sequence.
  4. The attention mechanism then attends over this combined set of keys and values.
  5. The loss is backpropagated to update only these prefix KV vectors across all layers.

- **Parameter Count**: For a model with N_{layers}, prefix length L_p, and model dimension D_{model}:
  - Trainable parameters $= N_{layers} \times L_p \times (D_{key} + D_{value})$. Since keys and values typically span the model dimension, this is roughly $N_{layers} \times L_p \times 2 \times D_{model}$.
  - **Example**: For a 32-layer model with $D_{model} = 4096$ and $L_p = 100$, we train $32 \times 100 \times 2 \times 4096 \approx 26$ million parameters. This is more than prompt tuning but still a tiny fraction (<0.5%) of a full fine-tune.

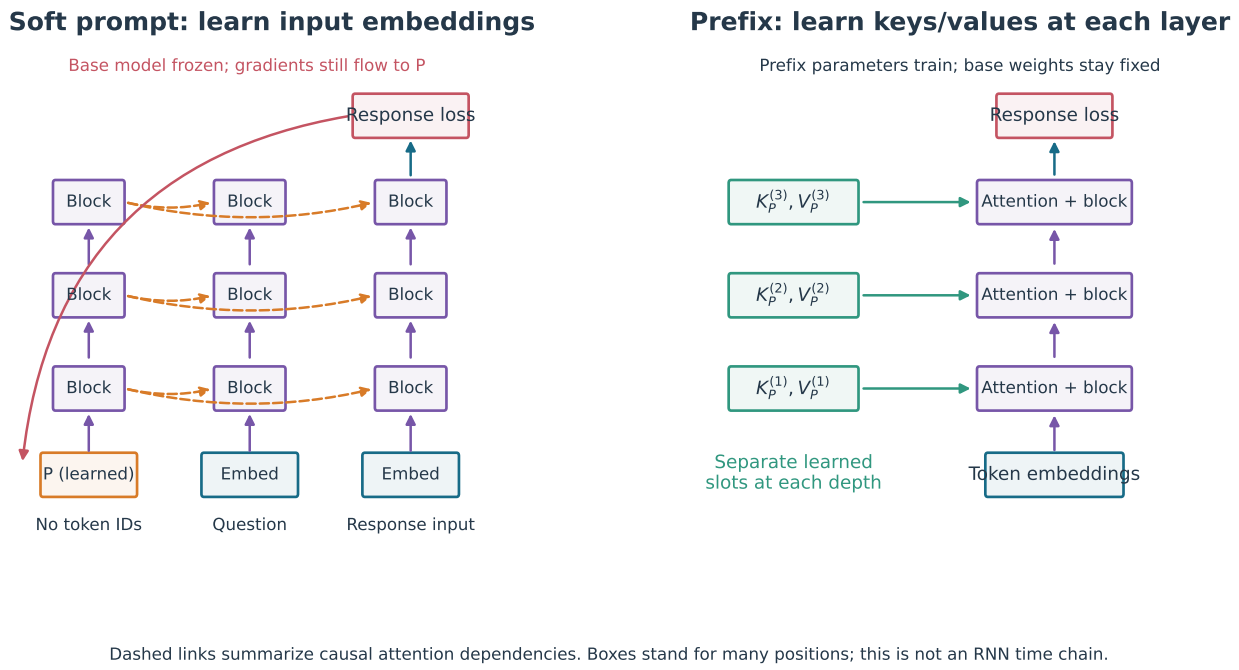


Figure 20.2: Soft prompts and per-layer prefixes. A structural redraw of the two training scaffolds. Soft prompts learn input embeddings; prefix tuning supplies learned key/value slots at each layer. Selected dashed attention dependencies are shown, not every edge. Gradients traverse the frozen model to reach the trainable parameters. Structural redraw; Anant Sahai, faculty handout 22, p. 3.

Worked derivation: masked SFT loss and trainable prompt paths

Lecture basis: faculty handout 21, pp. 4–7. The steps below expand the lecture argument.

Step 1: distinguish the loss mask from the attention mask. For a prompt followed by a response, let $m_t = 1$ on response prediction positions and zero on prompt-only positions. The supervised loss is

$$L = -\frac{1}{\sum_t m_t} \sum_t m_t \log p_\theta(y_t \mid y_{<t}).$$

A causal attention mask restricts what a position can see; this loss mask restricts which predictions are scored. A prompt position can have no direct prediction loss and still affect every later response prediction.

Step 2: follow the gradient to a soft prompt. Let $P \in \mathbb{R}^{m \times d}$ be trainable prompt embeddings and freeze the model weights θ. For response logits $z_t = F_\theta(P, x)_t$,

$$\begin{aligned} dL &= \sum_t (\nabla_{z_t} L)^\top dz_t, \\ dz_t &= J_{P,t}\, d\,\mathrm{vec}(P), \\ \nabla_{\mathrm{vec}(P)} L &= \sum_t J_{P,t}^\top \nabla_{z_t} L. \end{aligned}$$

Freezing θ means not updating those parameters. It does not mean discarding the Jacobians through the frozen network: they are needed to reach P. Prompt inputs that were detached from the computation graph would receive no such gradient.

Step 3: count the trainable entries. An input soft prompt has md parameters. Prefix tuning with m trainable key and value vectors at each of L layers has $2mLd_{\mathrm{KV}}$ entries when d_{KV} is the total key/value width per layer. With $m = 100$, $d = d_{\mathrm{KV}} = 4096$, $L = 32$,

$$md = 409{,}600, \qquad 2mLd_{\mathrm{KV}} = 26{,}214{,}400.$$

Grouped-query attention may have $d_{\mathrm{KV}} < d$, and some methods parameterize prefixes through an auxiliary network. Those choices change the count; the handout's example assumes the stated full KV width.

Lecture 21

Fine-Tuning and Meta-Learning

Course sources: lecture_22; 2025-11-13 transcript. Lecture numbering follows the original reader; linked faculty handouts identify the overlapping source coverage.

21.1 Recap: Fine-Tuning and Gradient Flow

Deeper explanation: Differentiate a masked response loss through the prompt (p. 429)

When fine-tuning a transformer model for a task like question-answering, a common setup involves providing a prompt (the question) and training the model to generate a specific answer. The loss, typically a cross-entropy loss, is calculated only on the generated answer tokens.

A key point of clarification is the flow of gradients. During backpropagation, the gradient originates from the loss on the answer part. However, due to the nature of the attention mechanism, which connects all tokens to all previous tokens, the gradient flows "back in time" to all preceding parts of the sequence, including the prompt/question tokens.

- The attention blocks create computational paths from the answer tokens back to the question tokens.
- Therefore, even though there is no explicit loss computed on the question part, its representations and the parameters that produced them are updated during fine-tuning. This allows the model to learn how to better process prompts for the given task.

21.2 A Spectrum of Approaches for New Tasks

Deeper explanation: A controlled adaptation experiment (p. 433)

Given a pre-trained foundation model, there are several ways to adapt it for a new task. These methods exist on a spectrum of how much the model's parameters are changed.

Pure Prompting (0-shot, Few-shot)

This approach requires no gradient updates. It leverages the model's in-context learning capabilities.

- **Pure In-Context Learning:** Provide the model with a few examples (demonstrations) of the task within the prompt itself, followed by the new query.

- **Instruction Tuning:** For models that have been instruction-tuned (like InstructGPT), you can provide a combination of natural language instructions and examples.
- **Prompt Optimization:** The process of finding the optimal prompt (either instructions or examples) can be automated. This involves using a validation set to score different prompts and employing optimization techniques to search the discrete space of tokens for the best-performing prompt.

Full Fine-Tuning

In full fine-tuning, all (or a substantial portion) of the pre-trained model's weights are unfrozen and updated via gradient descent on the new task's data. This is parameter-intensive but often yields high performance.

Parameter-Efficient Fine-Tuning (PEFT)

PEFT methods aim to match the performance of full fine-tuning while only updating a very small number of parameters. This significantly reduces computational and storage costs.

Soft Prompting (Prompt Tuning) Instead of finding optimal "hard" prompts (sequences of discrete tokens), we introduce a "soft prompt" which is a sequence of continuous embedding vectors.

- **Mechanism:** A sequence of learnable embedding vectors is prepended to the input sequence (before the question). These vectors are not tied to any specific tokens in the vocabulary.
- **Parameters:** The only learnable parameters are these new embedding vectors. The entire pre-trained model remains frozen.
- **Gradient Flow:** Gradients from the loss on the answer flow back through the frozen model and are used exclusively to update the soft prompt embeddings.
- **Memory:** While the number of learnable parameters is tiny, the memory required for the forward and backward pass is unchanged. You still need to store all the intermediate activations of the large model for backpropagation (or recompute them). PEFT primarily saves memory on the optimizer states (e.g., momentum and variance terms in Adam), which are only needed for the parameters being updated.

Soft Prefixes (Prefix Tuning) This is a more expressive variant of soft prompting. Instead of learning embeddings at the input layer, we directly insert learnable vectors into the intermediate layers of the transformer.

- **Mechanism:** For the prefix tokens, instead of computing their key (K) and value (V) vectors from the input, we treat these K and V vectors themselves as learnable parameters directly in each attention block.
- When the model processes the question and answer tokens, their queries (Q) attend to these learnable K and V pairs from the prefix.
- **Parameters:** The learnable parameters are the K and V vectors for the prefix sequence, across all layers. These parameters are independent for each layer.
- **Gradient Flow:** Gradients flow back from the answer tokens. When they reach the attention computation involving the prefix, they stop at the learnable K and V parameters, as these parameters do not depend on any earlier computations.

PEFT Parameter Counts Example For a typical large model (e.g., Llama, 7-8B parameters) with an embedding dimension of 4096:

- **Soft Prompt:** A prompt of length 100.

$$\text{Params} = \text{length} \times \text{embedding\_dim} = 100 \times 4096 = 409{,}600$$

- **Soft Prefix:** A prefix of length 100, assuming 32 layers and K, V vectors also of size 4096.

$$\text{Params} = \text{length} \times \text{layers} \times (\text{dim}_K + \text{dim}_V) = 100 \times 32 \times (4096 + 4096) \approx 26\text{M}$$

Both are substantially smaller than the 7-8 billion parameters of the full model.

Why Can PEFT Outperform Full Fine-Tuning?

Empirically, it has been observed that PEFT methods can sometimes achieve better generalization on the fine-tuning task than a full fine-tune.

- **Regularization/Overfitting:** The classic machine learning explanation is related to the bias-variance tradeoff. A full fine-tune adjusts billions of parameters, giving it immense capacity to overfit to the specifics of a potentially small fine-tuning dataset.
- By severely restricting the number of learnable parameters (e.g., 410k or 26M), PEFT acts as a strong regularizer, forcing the model to find a more general solution and preventing it from fitting noise in the training data. This is analogous to fitting a line (few parameters) being better for generalization than a 500-degree polynomial (many parameters) on a small dataset.

21.3 Low-Rank Adaptation (LoRA): A General PEFT Method

Deeper explanation: Choose the initial LoRA subspace deliberately (p. 433)

Deeper explanation: Checking the first LoRA gradient numerically (p. 429)

Deeper explanation: LoRA as a bilinear parameterization (p. 431)

Deeper explanation: Adapter composition and embedding matrices (p. 434)

Observing that different PEFT methods work well for different tasks, researchers sought a more general and powerful recipe. This led to LoRA.

The Core Idea

The central hypothesis of LoRA is that the change in weights needed to adapt a pre-trained model to a new task has a low "intrinsic rank." That is, the adaptation can be represented by a low-rank matrix.

- Instead of updating the original weight matrix $W_0 \in \mathbb{R}^{d\times k}$, we keep it frozen.
- We learn a low-rank update matrix ΔW, such that the new effective weight matrix is $W = W_0 + \Delta W$.
- To enforce the low-rank constraint, ΔW is decomposed into the product of two smaller matrices: $\Delta W = BA$, where $B \in \mathbb{R}^{d\times r}$ and $A \in \mathbb{R}^{r\times k}$.
- The rank r is a hyperparameter, typically very small (e.g., 4, 8, 16), which is much smaller than d or k.

The forward pass is modified as:

$$h = Wx = (W_0 + BA)x = W_0x + B(Ax) \tag{21.1}$$

This means we can implement LoRA as a parallel path to the original frozen layer, whose output is added to the original layer's output.

LoRA as a stacked branch and a block readout

For column input $x \in \mathbb{R}^n$, let $W_0 \in \mathbb{R}^{m\times n}$, $A \in \mathbb{R}^{r\times n}$, and $B \in \mathbb{R}^{m\times r}$. The two branches can be written as

$$u = \underbrace{\begin{bmatrix} W_0 \\ sA \end{bmatrix}}_{(m+r)\times n} x, \qquad y = \underbrace{\begin{bmatrix} I_m & B \end{bmatrix}}_{m\times(m+r)} u = (W_0 + sBA)x.$$

Block multiplication recovers the merged inference weight directly. For output gradient g, the trainable factors have gradients

$$\nabla_B L = s\, g(Ax)^\top, \qquad \nabla_A L = s\, B^\top g\, x^\top.$$

With $B = 0$ and a nonzero A, the B gradient can be nonzero while the A gradient is initially zero. Initializing both factors to zero blocks both gradients. The stacked form is an algebraic explanation of the branches, not an extra set of trainable parameters.

Training and Inference

- **Training:** During training, W_0 is frozen. Only the parameters of A and B are learnable and receive gradient updates.
- **Inference:** After training, the update can be merged back into the original weights: $W_{new} = W_0 + BA$. The resulting model has the exact same architecture and inference latency as the original model. There is no overhead. This allows for creating multiple "adapted" versions of a model by simply swapping out the main weight matrix.

Note: LoRA can adapt attention projections, MLP matrices, and embedding or output-projection matrices. An embedding lookup is a linear map on a one-hot vector; tied weights require an explicit sharing convention.

Initialization and Optimization: Best Practices

The initialization of A and B is critical for successful LoRA fine-tuning.

1. **Failed Idea: Initialize A=0, B=0.** This doesn't work because the gradients for both A and B will be zero at the first step, and the model will never learn.
2. **Failed Idea: Initialize A and B with Xavier/Gaussian.** This is also problematic. At the start of fine-tuning, $\Delta W = BA$ would be a random non-zero matrix. This changes the initial function by a size depending on the factor scales. A compensated frozen branch $W_0 - B_0A_0$ can preserve the initial function even with two nonzero factors. The goal is to *adapt*, not to restart from a random point.
3. **Correct Approach: Initialize B=0, A with Xavier/Gaussian.** This ensures that at initialization, $\Delta W = BA = 0$. The fine-tuning process starts exactly from the pre-trained model's state. As training progresses, B moves away from zero, and the model adapts smoothly.

Optimization Details:

- **Optimizer:** Use a standard optimizer like AdamW. Mup-style optimizers are generally not recommended by default for LoRA, as the theoretical assumptions may not hold for this specific architecture.
- **Learning Rates:** Use two different learning rates for A and B. The lecture discusses experiments using a larger learning rate for B than for A ($\text{lr}_B = \lambda \cdot \text{lr}_A$, with $\lambda \sim 4-10$). This can be loosely motivated by Mup principles, as A and B have vastly different

fan-ins (k vs r), but the exact Mup scaling doesn't work perfectly because the rank r is often too small for the theoretical guarantees to hold.

Worked derivation: LoRA gradients and why only one factor starts at zero

Lecture basis: faculty handout 22, pp. 5–6. The steps below expand the lecture argument.

Step 1: fix shapes and the scale convention. Let $W_0 \in \mathbb{R}^{m\times n}$ be frozen, $A \in \mathbb{R}^{r\times n}$, $B \in \mathbb{R}^{m\times r}$, and $y = W_0x + sBAx$. The scalar s is a chosen adapter scale (often α/r); the handout's BA notation corresponds to $s = 1$ or to absorbing the scale into a factor.

Step 2: take the differential with respect to both factors. For $g = \nabla_y L$ and $u = Ax$,

$$\begin{aligned} dy &= s(dB)u + sB(dA)x, \\ dL &= sg^\top(dB)u + sg^\top B(dA)x, \\ \nabla_B L &= sgu^\top = sg(Ax)^\top, \\ \nabla_A L &= sB^\top gx^\top. \end{aligned}$$

For a batch, sum these outer products with the normalization specified by the loss.

Step 3: evaluate the initial gradients. If $A = B = 0$, both gradients are zero and the adapter cannot start learning through ordinary first-order updates. If $B = 0$ but A is nonzero, the forward update BA is still zero, while

$$\nabla_A L = 0, \qquad \nabla_B L = sg(Ax)^\top$$

can be nonzero. Once B changes, gradients can reach A as well. This starts from the base model's predictions without killing both gradient paths.

Step 4: count and merge. The adapter has $rn + mr = r(n + m)$ entries rather than mn. Its matrix update has rank at most r because every output of BA lies in the column space of B. At inference,

$$W_0x + sB(Ax) = (W_0 + sBA)x.$$

The equality is exact for this linear branch with fixed factors and scale. Training-only adapter dropout must be disabled to use the same deterministic merged inference map.

Advanced Initialization Strategies for A

While random Gaussian initialization for A works well, more targeted initializations can potentially improve performance.

- **SVD of Gradients:**
  1. Compute a batch of gradients for the full weight matrix W_0 on the fine-tuning data. Let this be G_W.
  2. Perform Singular Value Decomposition (SVD) on G_W.
  3. Initialize the rows of matrix A with the top r right singular vectors of G_W. This aligns the initial updates with the principal directions of the gradient, potentially speeding up convergence.
- **SVD of Weights:**
  1. Perform SVD on the pre-trained weight matrix W_0 itself.
  2. Initialize the rows of matrix A with the top r right singular vectors of W_0. This is based on the hypothesis that the most important directions for the new task are related to the most salient directions (principal components) of the original pre-trained weights.

Implications: Model Arithmetic

The success of LoRA, which treats adaptations as simple additions to the weight space, has bolstered the field of model merging. One can take a base model, fine-tune two different LoRAs on it for two different tasks (e.g., Task 1 and Task 2), and then create a new model by adding both adaptations: $W_{new} = W_0 + B_1A_1 + B_2A_2$. The resulting matrix is well defined, but retaining both task behaviors is an empirical question; parameter addition does not guarantee compatible functions.

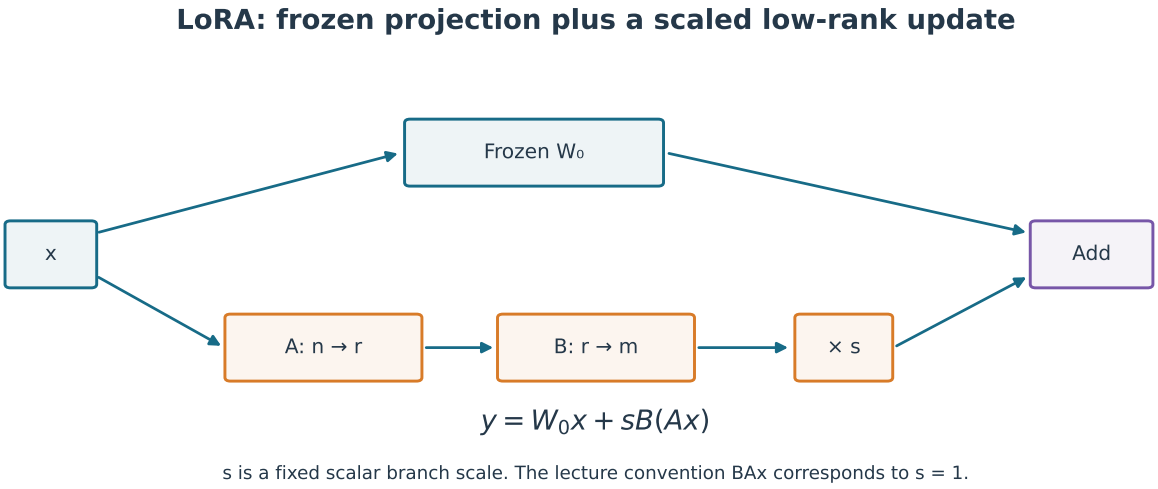


Figure 21.1: LoRA adds a narrow trainable branch around a frozen projection. The low-rank factors can be merged into the base matrix for an ordinary linear forward pass.

The scalar s explicitly controls the strength of the LoRA branch. With $s = 1$, $sB(Ax) = BAx$, exactly the unscaled convention in the faculty diagram. A fixed nonzero scale can be absorbed into a factor algebraically, although the parameterization affects optimization when training the factors.

21.4 Meta-Learning: Learning to Fine-Tune

Deeper explanation: Several inner steps require a product of Jacobians (p. 436)

The goal of meta-learning is to train a model that is not just good at a task, but is specifically good at being **fine-tuned** for a new set of tasks. We want to optimize the model's initial state to be a better starting point for future adaptation.

Model-Agnostic Meta-Learning (MAML)

MAML provides a general, conceptual framework for solving the meta-learning problem. While not always used in practice today for LLMs, its core ideas are highly influential.

Problem Formulation: The Ingredients To solve the meta-learning problem, we must first define it formally. We need:

1. **A collection of tasks:** We need a training set of tasks. Each task T_i is itself defined by its own training data and a loss function, e.g., $\{(\mathbf{x}_{i,j}, \mathbf{y}_{i,j}), \mathcal{L}_i\}$.
2. **A defined fine-tuning procedure:** We must specify exactly how adaptation will occur. For example: "take 3 gradient steps with an SGD optimizer and a learning rate of 0.01".
3. **A defined evaluation procedure:** We must specify how we measure success after fine-tuning. For example: "evaluate the performance (loss) on a held-out test set for that task".

The MAML Insight and Algorithm The key insight of MAML is to **train like you test** and make the entire process differentiable. The "test" procedure is fine-tuning. Therefore, we should simulate the fine-tuning process during our meta-training and optimize the initial parameters based on post-fine-tuning performance.

The algorithm can be summarized as:

1. Start with a set of initial (meta) parameters, θ. These are the parameters we want to learn.
2. For a batch of tasks $\{T_i\}$ from our collection:
   (a) For each task T_i, perform a simulated fine-tuning. This is the **inner loop**.

- Copy the meta-parameters: $\theta_i' \leftarrow \theta$.
- Take one or more gradient descent steps on the *training data* of task T_i to get updated parameters: $\theta_i'' = \theta_i' - \alpha \nabla_{\theta'} \mathcal{L}_{T_i}(\theta_i')$.

(b) Now, evaluate the performance of the fine-tuned parameters θ_i'' on the *test data* of task T_i. This gives a meta-loss, $\mathcal{L}_{T_i}(\theta_i'')$.

3. The total meta-loss is the sum of losses over the batch of tasks, $\sum_i \mathcal{L}_{T_i}(\theta_i'')$.
4. This entire process—from θ to θ_i' to θ_i'' to the final loss—is one large computation graph. We can compute the gradient of the meta-loss with respect to the original meta-parameters, $\nabla_\theta \sum_i \mathcal{L}_{T_i}(\theta_i'')$.
5. Update the meta-parameters θ using this "meta-gradient". This is the **outer loop**.

Essentially, MAML views the fine-tuning process (a sequence of SGD steps) as a recurrent neural network. The initial parameters θ are the initial hidden state, and the gradients are backpropagated through the unrolled steps of the learning process itself to update this initial state. The result is a model θ that is optimized to be just a few gradient steps away from high performance on a wide range of tasks from the target family.

Lecture 22

Meta-Learning and Generative Models Intro

Course sources: lecture_23; 2025-11-18 transcript. Lecture numbering follows the original reader; linked faculty handouts identify the overlapping source coverage.

22.1 Recap: Adapting Pre-trained Models

Deeper explanation: A controlled adaptation experiment (p. 433)

When adapting a pre-trained model to a new task, several approaches are available, ranging from zero-parameter changes to full model updates.

1. **Prompting:** For models that are "promptable" (e.g., instruction-tuned LLMs), one can simply provide a natural language prompt describing the new task. This is a zero-shot or few-shot approach that leverages in-context learning.
    - **Practical Tip:** If you have data for your task, use a prompt optimizer to find the best-performing prompt instead of relying on manual prompt engineering.
2. **Linear Probing (Feature Extraction):** Treat the pre-trained model as a fixed feature extractor.
    - The process involves passing task data through the frozen pre-trained model to get embeddings.
    - A new, simple task-specific "head" (usually a linear classifier or regressor) is trained on top of these embeddings.
    - **Advantages:** Computationally cheap and fast. Requires only task-specific data.
    - **Disadvantages:** May not perform as well as full fine-tuning.
    - **Observation:** Interestingly, for some tasks, linear probing can outperform prompting, even on large models. This suggests the model's internal representations contain the necessary information, even if it cannot be elicited through prompting alone.
3. **Full Fine-Tuning:** Update all (or a significant portion) of the pre-trained model's parameters, along with a new task-specific head, using the new task's data.
    - **Advantages:** Typically yields the highest performance on the new task.

- **Disadvantages:** Computationally expensive, requiring significant resources for training and hyperparameter tuning. It also carries a risk of overfitting and *catastrophic forgetting* (discussed later).

4. **Parameter-Efficient Fine-Tuning (PEFT):** Methods like LoRA or soft-prompting/prefix-tuning that are intermediate between linear probing and full fine-tuning. They involve training a small number of additional parameters while keeping the base model frozen.

Practical Tips for Full Fine-Tuning

When performing a full fine-tune with a new task-specific head, the initialization strategy is crucial.

- **Bad Practice:** Initializing the new head with random weights. At the start of training, a random head will produce meaningless outputs and send noisy, unhelpful gradients back into the pre-trained model. This can disrupt the valuable pre-trained weights before any meaningful learning occurs.
- **Better Practice:** Initialize the new head with all zeros. This ensures that in the initial training steps, zero or near-zero gradients are passed back to the pre-trained model, protecting its weights from random perturbations.
- **Best Practice (Two-Stage Tuning):**
    1. **Stage 1 (Linear Probing):** Freeze the pre-trained model's weights and train only the new task-specific head on your task data. This allows the head to learn a reasonable mapping from the model's features to the task outputs.
    2. **Stage 2 (Full Fine-Tune):** Unfreeze the pre-trained model's weights and continue training the entire model (body and head) with a lower learning rate. At this stage, the gradients flowing from the head are much more meaningful, leading to a more stable and effective fine-tuning process.

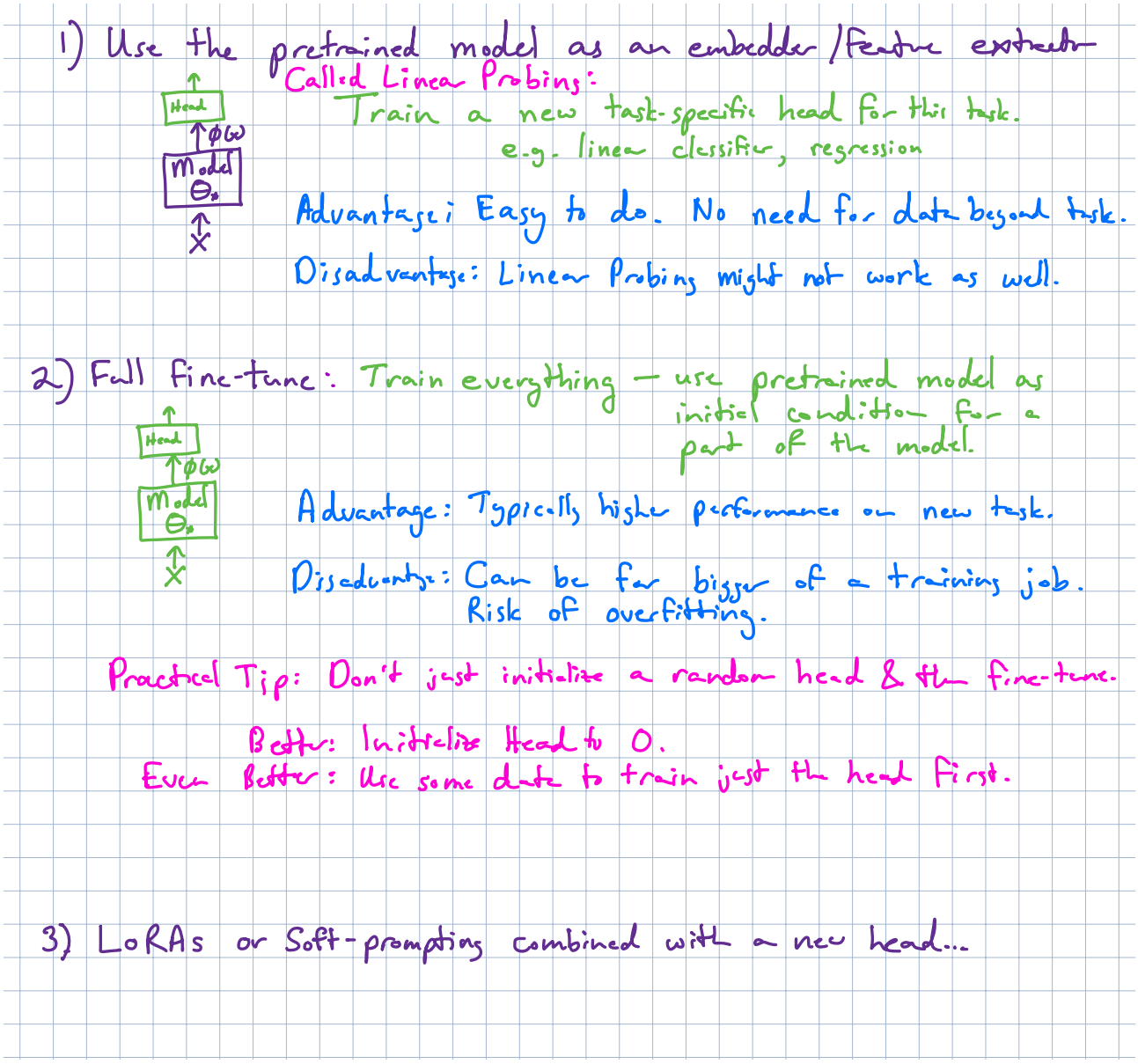


Figure 22.1: Training a new head versus updating the whole model. The marked trainable portions explain which parameters receive updates. Backpropagation through frozen computation is still possible when an earlier trainable input or module requires its gradient. Original annotated source excerpt; Anant Sahai, faculty handout 23, p. 1.

22.2 Meta-Learning: Learning to Fine-Tune

Deeper explanation: Exact, first-order, and implicit differentiation (p. 438)

Deeper explanation: A differentiable closed-form learner (p. 436)

Deeper explanation: Meta-learning an initialization on a family of quadratics (p. 438)

Meta-learning aims to train a model that is not just good at a task, but is specifically good at being *fine-tuned* for a family of new, unseen tasks. The goal is to find a model initialization that can adapt quickly and effectively.

Model-Agnostic Meta-Learning (MAML)

MAML is a general framework for learning a good parameter initialization. It relies on two key insights:

1. **Train like you will test:** At test time, we will perform gradient descent (fine-tuning) on a new task. Therefore, the meta-training process should itself involve simulating this fine-tuning process.
2. **Learning as an RNN:** The process of performing multiple steps of Stochastic Gradient Descent (SGD) is a sequential computation. It can be viewed as an unrolled Recurrent Neural Network (RNN), where the model's parameters are the "hidden state." Since RNNs are differentiable, we can backpropagate through the entire learning process.

The MAML Algorithm MAML operates with two loops: an inner loop for task-specific adaptation and an outer loop for meta-optimization.

- **Setup:** We need a distribution of tasks. For each task $\mathcal{T}_i$, we have training data D_i (for the inner loop) and held-out test data $\tilde{D}_i$ (for the outer loop).
- **Outer Loop (Meta-Training):**
  1. Sample a batch of tasks $\{\mathcal{T}_i\}$ from the task distribution.
  2. For each task $\mathcal{T}_i$:
     (a) **Inner Loop (Task Adaptation):**
     - Create a copy of the current meta-parameters, $\theta'_0 = \theta$.
     - Perform a few steps of gradient descent on the task's training data D_i to find task-specific parameters θ'_k.

       $$\theta'_{j+1} = \theta'_j - \alpha \nabla_{\theta'_j} \mathcal{L}_{\mathcal{T}_i}(\theta'_j, D_i)$$

       This inner loop is performed for k steps, resulting in final adapted parameters θ'_k.

     (b) **Meta-Gradient Calculation:** Evaluate the performance of the adapted parameters θ'_k on the task's held-out data $\tilde{D}_i$. The loss on this held-out data, $\mathcal{L}_{\mathcal{T}_i}(\theta'_k, \tilde{D}_i)$, is the meta-loss for this task.

     (c) The gradient of this meta-loss is computed with respect to the initial meta-parameters θ. This involves backpropagating through all k steps of the inner-loop optimization.

3. **Outer Loop Update (Meta-Optimization):** The meta-parameters θ are updated by taking a step in the direction of the averaged meta-gradients from the batch of tasks.

$$\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i} \mathcal{L}_{\mathcal{T}_i}(\theta'_k, \tilde{D}_i)$$

Here, α is the inner-loop learning rate and β is the outer-loop (meta) learning rate.

This process is computationally demanding because it requires computing gradients of gradients ("backprop through backprop") and storing the entire computation graph of the inner loop.

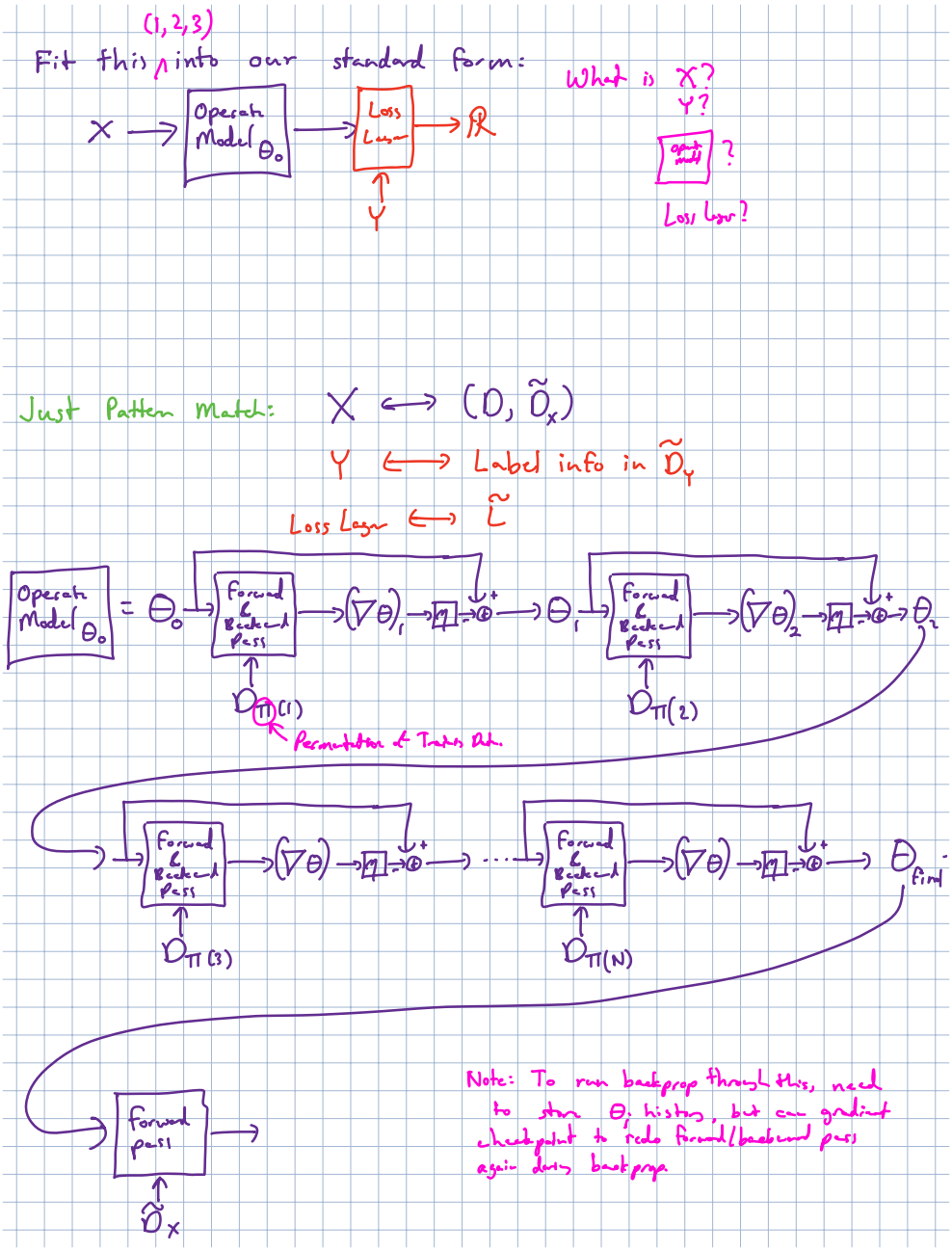


Figure 22.2: The optimizer unrolled as a computation graph. Inner-loop updates become nodes in the outer objective's graph. Differentiating through them measures how the initialization changes performance after adaptation, not just before adaptation. Original annotated source excerpt; Anant Sahai, faculty handout 23, p. 3.

Worked derivation: MAML through one and several inner steps

Lecture basis: faculty handout 23, pp. 2–5. The steps below expand the lecture argument.

Step 1: separate support fitting from query evaluation. For one task, let $L_S(\theta)$ be the support loss and L_Q the held-out query loss. One inner update is

$$\theta' = \theta - \alpha \nabla L_S(\theta), \qquad F(\theta) = L_Q(\theta').$$

The outer objective evaluates the adapted parameters, not the original parameters on the support set.

Step 2: differentiate the update itself. Assuming twice differentiability,

$$\begin{aligned} d\theta' &= d\theta - \alpha \nabla^2 L_S(\theta) d\theta \\ &= (I - \alpha H_S) d\theta, \\ dF &= \nabla L_Q(\theta')^\top (I - \alpha H_S) d\theta, \\ \nabla F(\theta) &= (I - \alpha H_S)^\top \nabla L_Q(\theta'). \end{aligned}$$

The Hessian term accounts for how changing the initialization changes the *adaptation step*. First-order MAML drops that term; it is an approximation, not the exact chain rule.

Step 3: unroll several updates in the correct order. Let $\theta_{k+1} = \theta_k - \alpha \nabla L_{S,k}(\theta_k)$, $\theta_0 = \theta$, and $J_k = I - \alpha H_{S,k}(\theta_k)$. Then

$$\begin{aligned} \frac{\partial \theta_K}{\partial \theta_0} &= J_{K-1} J_{K-2} \cdots J_0, \\ \nabla_{\theta_0} L_Q(\theta_K) &= J_0^\top J_1^\top \cdots J_{K-1}^\top \nabla L_Q(\theta_K). \end{aligned}$$

These products can be evaluated as Hessian-vector products without storing a dense Hessian. Reptile instead moves initialization toward task-adapted weights; that update is not this exact query-loss derivative.

Example: a scalar quadratic. If $L_S(\theta) = \frac{1}{2}(\theta - a)^2$, then $\theta' = (1 - \alpha)\theta + \alpha a$. For $L_Q(\theta') = \frac{1}{2}(\theta' - b)^2$,

$$\frac{dF}{d\theta} = (1 - \alpha)\big[(1 - \alpha)\theta + \alpha a - b\big].$$

The factor $1 - \alpha$ is precisely the update Jacobian. Omitting it changes the gradient even in this simplest example.

MAML Variants

Reptile Reptile is a first-order approximation of MAML that avoids the expensive second-order derivatives.

- **Idea:** Instead of backpropagating through the inner-loop optimization, approximate the meta-gradient as the difference between the final adapted parameters and the initial meta-parameters.
- **Algorithm:**
  1. Sample a task $\mathcal{T}_i$.
  2. Starting with meta-parameters θ, run SGD for k steps on the task data D_i to obtain adapted parameters θ'_k.
  3. Update the meta-parameters by moving them a small fraction of the way towards the adapted parameters:
$$\theta \leftarrow \theta + \beta(\theta'_k - \theta)$$
- **Benefits:** Computationally much cheaper, requires less memory, and allows for many more inner-loop steps. The intuition is that by taking small steps in the direction of task-specific optima, the meta-parameters will drift towards a point that is, on average, close to the optima of all tasks.

Meta-Learning for Linear Probing If the goal is to learn a model that serves as a good feature extractor (i.e., for linear probing), the inner loop can be made much more efficient.

- **Idea:** For linear models, the optimal head can often be found via a closed-form solution or a fast convex optimization algorithm, both of which are differentiable.
- For regression tasks with a squared-error loss, the optimal linear head has a closed-form solution (Normal Equations). We can differentiate through this closed-form calculation to get a gradient on the features, and thus on the feature extractor's weights.
- For classification tasks, problems like training an SVM or logistic regression are convex. They can be solved rapidly with a few steps of an algorithm like Newton's method. These optimization steps are also differentiable.
- **Benefits:** This approach avoids the iterative nature of the SGD inner loop, replacing it with a single, highly stable, and efficient computation. It eliminates issues with exploding/vanishing gradients in the inner loop and is highly parallelizable.

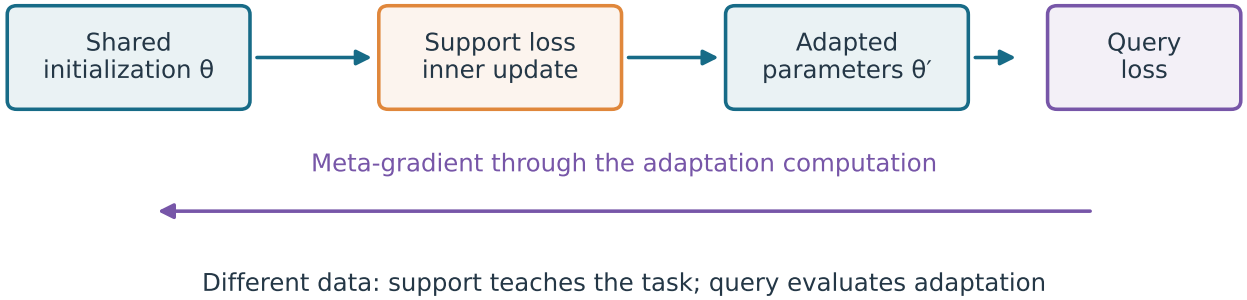


Figure 22.3: Meta-learning differentiates through an adaptation step. Support data drives the inner update; query data evaluates the resulting parameters.

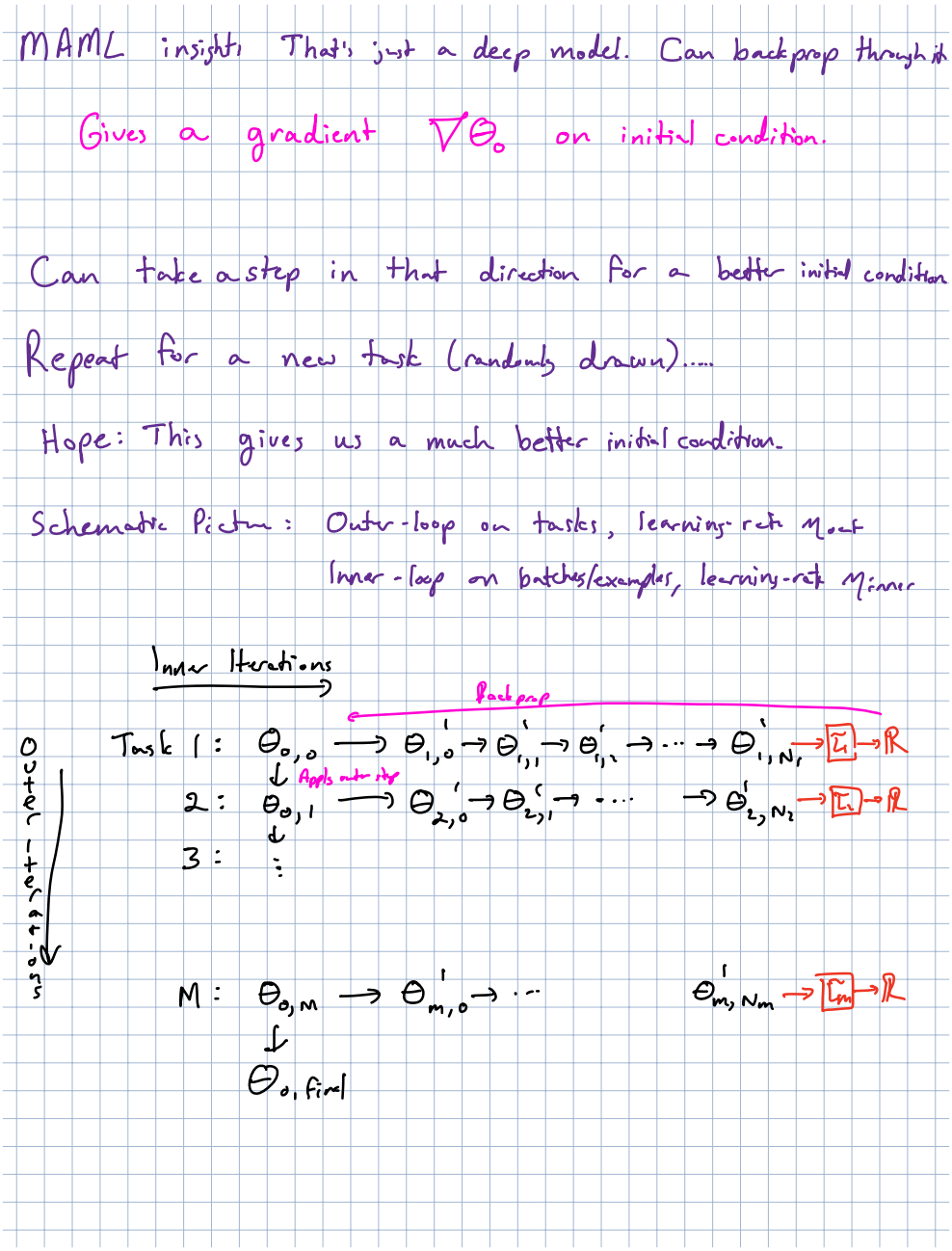


Figure 22.4: Tasks and adaptation steps are different axes. Read one axis as inner updates for a task and the other as the collection of tasks contributing to meta-training. Shared initialization links the task-specific trajectories. Original annotated source excerpt; Anant Sahai, faculty handout 23, p. 4.

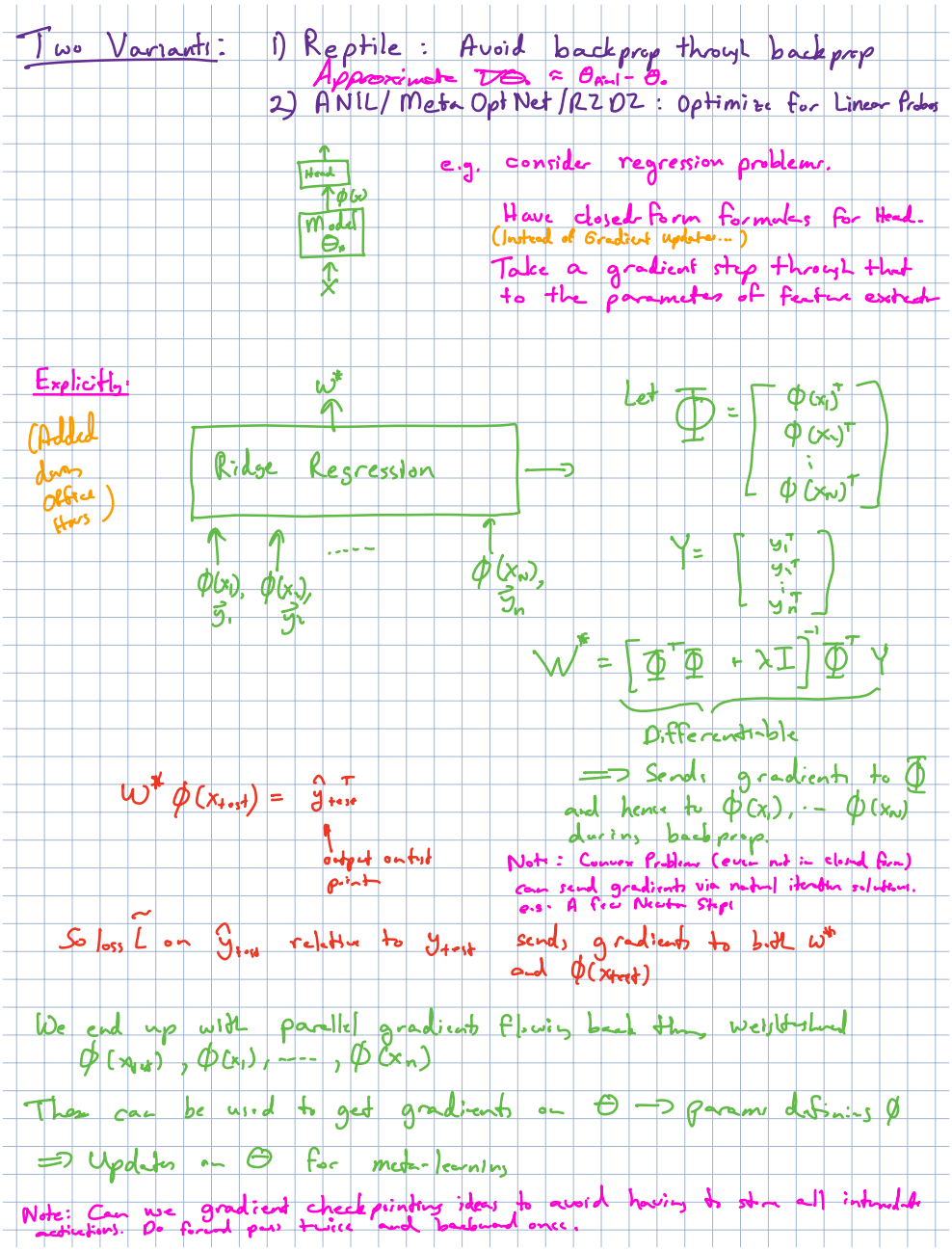


Figure 22.5: A differentiable ridge-regression adaptation layer. The inner learner can be a closed-form solve rather than a long optimizer unroll. Gradients must account for the solve's dependence on the learned representation and regularization. Original annotated source excerpt; Anant Sahai, faculty handout 23, p. 5.

Worked derivation: backpropagating through a fitted ridge head

Lecture basis: faculty handout 23, pp. 5. The steps below expand the lecture argument.

Step 1: express fitting as a linear system. Let learned support features be rows of $\Phi \in \mathbb{R}^{n\times d}$, with fixed labels y. For $\lambda > 0$, the fitted head solves

$$Mw = b, \qquad M = \Phi^\top \Phi + \lambda I, \qquad b = \Phi^\top y.$$

Step 2: differentiate the equation instead of an inverse formula. For fixed y, λ,

$$\begin{aligned}(dM)w + M\,dw &= db,\\ M\,dw &= (d\Phi)^\top y - \big[(d\Phi)^\top \Phi + \Phi^\top d\Phi\big]w\\ &= (d\Phi)^\top (y - \Phi w) - \Phi^\top (d\Phi) w,\\ dw &= M^{-1}\big[(d\Phi)^\top (y - \Phi w) - \Phi^\top (d\Phi) w\big].\end{aligned}$$

This displays both effects of changing a feature: changing the right-hand side and changing the fitted system matrix.

Step 3: use an adjoint solve to obtain a gradient. For an outer loss with head gradient $g = \nabla_w L_Q$, solve $M^\top v = g$. Put $e = y - \Phi w$. Then

$$\begin{aligned}dL_Q &= v^\top\big[(d\Phi)^\top e - \Phi^\top (d\Phi) w\big],\\ \nabla_\Phi L_Q\big|_{\text{fit path}} &= ev^\top - (\Phi v) w^\top.\end{aligned}$$

If query features also depend on the feature-network parameters, their direct gradient path must be added. If λ is learned, its fit-path derivative is $-v^\top w$. Solving the inner regression exactly does not prevent differentiating through its result.

22.3 Catastrophic Forgetting

Deeper explanation: Forgetting as a multi-objective tradeoff (p. 439)

Catastrophic forgetting is the phenomenon where a model, when fine-tuned on a new task, loses its ability to perform previously learned tasks.

- **The Problem:** This is particularly problematic when the goal of fine-tuning is to add a general capability (e.g., instruction following, safety alignment) without sacrificing the model's vast pre-trained knowledge. Empirically, aligning a model for safety can make it "dumber" on unrelated, safe tasks.
- **Intuition:**

  - **Leaky Balloon Analogy:** Think of the model's capabilities as different parts of an inflatable structure. The training data for a specific capability acts as an air pump, keeping that part inflated. If you stop "pumping" data for one capability and only pump for another, the first one deflates due to factors like weight decay or gradient interference.
  - **Gradient Interference:** The gradients for the new task, while optimizing for that task, introduce effectively random noise into the weight directions responsible for other tasks. Without a counter-balancing signal from the old task data, the model's weights drift, and performance on old tasks degrades.

- **Standard Solution: Data Replay/Mixing** The most common and effective solution is to mix a small amount (e.g., 5-10%) of the original pre-training data (or data from a similar distribution) into the fine-tuning dataset. This ensures that while the model learns the new task, it continuously receives a "refresher" signal that prevents it from forgetting its general pre-trained capabilities.
- **Forgetting and PEFT:** Restricting trainable parameters can limit changes, but does not guarantee less forgetting. Penalizing low-rank factors does not directly constrain every old-task output; evaluate old and new tasks together.

22.4 Introduction to Generative Models

Deeper explanation: A linear-Gaussian model as a solvable reference (p. 451)

Generative models aim to learn the underlying probability distribution of a dataset, $p(\mathbf{x})$, to be able to generate new, unseen samples from that distribution.

Types of Generation

- **Unconditional Generation:** Sample $\mathbf{x} \sim p(\mathbf{x})$. The goal is simply to generate a new, valid sample from the data distribution (e.g., "generate an image of a cat").
- **Conditional Generation:** Sample $\mathbf{x} \sim p(\mathbf{x}|c)$. The generation is conditioned on some input c (e.g., "generate an image corresponding to the text 'a photo of an astronaut riding a horse'").

Naive Approaches (That Don't Work)

Inverting a Classifier

- **Idea 1 (Rejection Sampling):** Train a classifier $p(y|\mathbf{x})$. To generate a "cat," sample a random image $\mathbf{x}$ (e.g., from uniform noise) and accept it if the classifier assigns a high probability to the "cat" class. This fails because the space of natural images is

astronomically small compared to the space of all possible pixel combinations. The probability of randomly sampling a coherent image is effectively zero.

- **Idea 2 (Gradient Ascent):** Start with a random noise image $\mathbf{x}$ and use gradient ascent to iteratively modify $\mathbf{x}$ to maximize the classifier's output for the desired class, e.g., $\max_{\mathbf{x}} p(y = \text{cat}|\mathbf{x})$. This also fails to produce realistic images, often resulting in adversarial examples or high-frequency noise patterns that exploit the classifier without resembling a real image.

Using a Standard Autoencoder (AE)

- **Idea:** Train an autoencoder consisting of an encoder $E(\mathbf{x}) = \mathbf{z}$ and a decoder $D(\mathbf{z}) = \hat{\mathbf{x}}$. For generation, discard the encoder and feed random vectors $\mathbf{z}$ (e.g., sampled from a Gaussian) into the decoder.
- **Why It Fails - The Manifold Hypothesis:**
  - Natural data (like images) is assumed to lie on a low-dimensional manifold embedded in a very high-dimensional pixel space.
  - The encoder learns to map points from this data manifold in the input space to a corresponding manifold in the latent space $\mathcal{Z}$.
  - The decoder learns the reverse mapping. However, it only ever sees latent vectors $\mathbf{z}$ that come from the encoder—i.e., points on the latent manifold.
  - When we sample a random $\mathbf{z}$ from a simple distribution (like a Gaussian), it will almost certainly fall into a "hole" or an empty region of the latent space, far from the manifold the decoder was trained on.
  - The decoder's behavior in these unseen regions is undefined and typically results in blurry or nonsensical outputs. The AE is not incentivized during training to produce good outputs for these "off-manifold" latent codes.

Towards Variational Autoencoders (VAEs)

To fix the issue with standard AEs for generation, we need to address two key problems: the lack of robustness in the decoder and the unstructured nature of the latent space.

1. **Inject Randomness During Training:** The "train like you test" principle applies. If we want to feed random $\mathbf{z}$ vectors to the decoder at test time, we must introduce randomness into the latent space during training.
2. **Regularize the Latent Space:** We need to force the distribution of encoded vectors from the training data to match a simple, known prior distribution that we can easily sample from (e.g., a standard normal distribution, $\mathcal{N}(0, I)$). This can be achieved by adding a regularization term (a loss) that penalizes divergence from this prior.

The combination of these ideas leads to the Variational Autoencoder, which trains the decoder to be robust to small perturbations in the latent space, effectively ”filling in the holes” and learning a smoother mapping from the latent space to the data manifold.

Lecture 23

Generative Models and Post-Training

Course sources: lecture_24; 2025-11-20 transcript. Lecture numbering follows the original reader; linked faculty handouts identify the overlapping source coverage.

23.1 Generative Models

Deeper explanation: A linear-Gaussian model as a solvable reference (p. 451)

Introduction

Generative modeling aims to create models that can generate new, unseen data samples that appear to be drawn from the same distribution as a given training dataset. Unlike discriminative models (e.g., classifiers) that learn to map inputs to labels ($p(y|x)$), generative models aim to learn the underlying data distribution itself ($p(x)$).

This can be conceptualized as learning to sample from an unknown distribution. The training process involves two main challenges:

1. Devising an approach that allows for sampling from a distribution.
2. Learning a parameterized form of this distribution from training data.

Types of Generation

There are two primary categories of generative tasks:

- **Unconditional Generation:** The model takes a source of randomness (e.g., a random vector) and produces a new example from the learned distribution. There is no input other than the randomness.

$$\text{Randomness} \rightarrow \boxed{\text{Generative Model (G)}} \rightarrow \text{New Example}$$

- **Conditional Generation:** The model takes both a source of randomness and some conditioning information (e.g., a text prompt, a class label) to produce a new example that is consistent with the conditioning. This is the more practically important case.

$$\text{Conditioning} + \text{Randomness} \rightarrow \boxed{\text{Generative Model (G)}} \rightarrow \text{New Example}$$

  Examples include text-to-image generation (conditioning is the text prompt) and language models generating text based on a starting prompt.

Initial Ideas That Don't Work

Using a Classifier A naive idea is to leverage a pre-trained classifier.

- **Idea:** To generate an image of a cat, we can use a cat classifier. An image is a "cat image" if the classifier outputs a high score for the "cat" class.
- **Approach:** Start with a random noise image and perform gradient ascent on the pixels of the image to maximize the cat classifier's output score.
- **Problem:** The space of all possible images is vast, while the "manifold" of natural-looking images is extremely thin and low-dimensional. Most images are just random noise. Gradient ascent on a classifier is not constrained to stay on this manifold.
- **Result:** The process finds an image that is adversarial; it maximizes the classifier's confidence but looks like random noise to a human. The model moves the random image to a nearby point in image space that is confidently "cat", but this point is not on the natural image manifold.

Using an Autoencoder An autoencoder learns a compressed latent representation of data, which seems promising for generation.

- **Architecture:** An autoencoder consists of an **encoder** (E) that maps an input x to a low-dimensional latent code z, and a **decoder** (D) that reconstructs the input $\hat{x}$ from z. It's trained to minimize the reconstruction loss, e.g., $\|x - \hat{x}\|^2$.

$$x \to \boxed{\text{Encoder}} \to z \to \boxed{\text{Decoder}} \to \hat{x}$$

- **Generative Idea:** Train an autoencoder on a dataset (e.g., cat images). Then, discard the encoder and use only the decoder. To generate a new image, sample a random vector z from a simple distribution (e.g., a Gaussian) and pass it through the decoder.
- **Problem:** Similar to the classifier issue, the latent codes z produced by the encoder for the training data also lie on a very low-dimensional manifold within the latent space. A randomly sampled z is almost certain to be far from this manifold. The decoder has never seen such a z during training and will extrapolate poorly.
- **Result:** The output is typically a noisy or blurry, non-sensical image, not a coherent sample from the data distribution.

23.2 Variational Autoencoders (VAEs)

The Variational Autoencoder (VAE) addresses the shortcomings of the naive autoencoder approach by adding a probabilistic spin. The goal is to make the decoder robust to variations in the latent space so that it can handle randomly sampled z vectors.

Key Ideas

Deeper explanation: Joint distributions, conditioning, and expectation (p. 440)

Deeper explanation: Conditional expectation and the tower property (p. 441)

Deeper explanation: Total variance and total covariance (p. 442)

Deeper explanation: Derive the Gaussian posterior by completing the square (p. 452)

The VAE introduces three core components to the autoencoder framework:

1. **Make z random during training:** Instead of the encoder producing a deterministic z, it will produce the parameters of a probability distribution over z. A sample is then drawn from this distribution to be passed to the decoder. This introduces noise and variation, making the decoder more robust.
2. **Add a loss on the distribution of z:** We enforce that the distribution of z produced by the encoder for a given input x should be "close" to a pre-specified prior distribution (e.g., a standard normal distribution, $\mathcal{N}(0, I)$). This encourages the latent space to be well-structured and dense, preventing gaps.
3. **Make this work with SGD:** The sampling step is inherently non-differentiable. A special technique, the "reparameterization trick," is used to allow gradients to flow back through the sampler to the encoder.

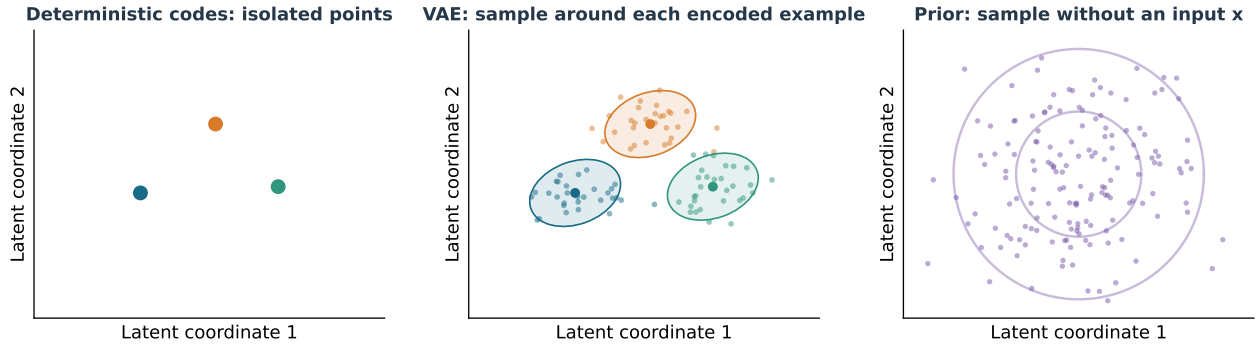


Figure 23.1: Latent codes, conditional distributions, and the prior. An explanatory visualization of the lecture idea, with illustrative geometry rather than a facsimile. Each input has its own conditional distribution. The KL tradeoff encourages compatibility with the sampling prior; it does not force every conditional distribution to coincide with it. Explanatory redraw; illustrative geometry; Anant Sahai, faculty handout 24, p. 2, 3.

VAE Architecture and Loss

Deeper explanation: Derive the evidence lower bound (p. 451)

Deeper explanation: Jensen's inequality connects curvature to probability (p. 443)

Deeper explanation: Cross-entropy from maximum likelihood (p. 448)

Deeper explanation: The ELBO is exact likelihood minus an inference gap (p. 454)

Deeper explanation: Amortized inference shares computation across examples (p. 454)

Deeper explanation: Reconstruction, interpolation, and unconditional sampling (p. 455)

The encoder, $q_\phi(z|x)$, takes an input x and outputs the parameters of a distribution for the latent code z. Typically, this is a Gaussian distribution, so the encoder outputs a mean vector μ_x and a covariance matrix Σ_x. A sample z is then drawn from $\mathcal{N}(\mu_x, \Sigma_x)$. This z is fed into the decoder, $p_\theta(x|z)$, which produces the reconstruction $\hat{x}$.

The VAE is trained to optimize a loss function with two components:

1. **Reconstruction Loss:** Encourages the decoder to accurately reconstruct the input. This is typically the Mean Squared Error (MSE) or binary cross-entropy.

$$\mathcal{L}_{\text{recon}} = \|x - \hat{x}\|^2$$

2. **KL Divergence Loss (Regularizer):** Forces the distribution $q_\phi(z|x)$ produced by the encoder to be close to a chosen prior distribution $p(z)$, usually $p(z) = \mathcal{N}(0, I)$. This is measured by the Kullback-Leibler (KL) divergence.

$$\mathcal{L}_{\text{KL}} = D_{KL}(q_\phi(z|x)\|p(z))$$

For a Gaussian likelihood, squared reconstruction error must use its variance-dependent coefficient to equal negative log likelihood. With that convention and $\beta = 1$, the following loss is the negative ELBO; other weights define a modified objective. The total loss is a weighted sum: $\mathcal{L} = \mathcal{L}_{\text{recon}} + \beta\mathcal{L}_{\text{KL}}$, where β is a hyperparameter that balances the two objectives. There is an inherent tension: the reconstruction loss pushes the model to encode as much information as possible into z, while the KL loss pushes z to be uninformative (close to the prior).

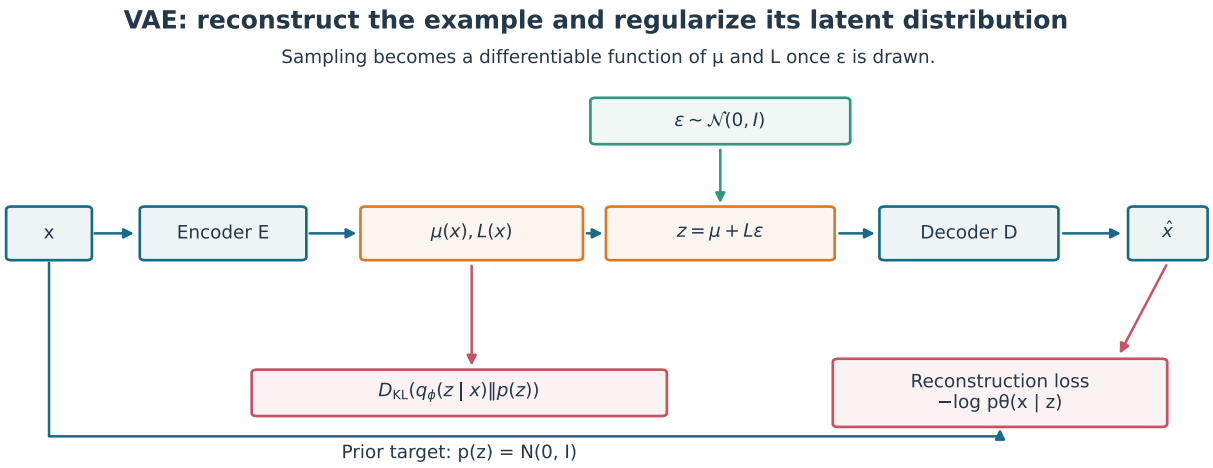


Figure 23.2: The two VAE losses and the reparameterized sample. A structural redraw of the lecture's encoder–sampler–decoder. Reconstruction trains both networks; the KL term constrains the encoder distribution. External noise is sampled independently, and its affine transformation remains differentiable. Structural redraw; Anant Sahai, faculty handout 24, p. 2, 3.

Worked derivation: the ELBO and its exact inference gap

Lecture basis: faculty handout 24, pp. 2–3. The steps below expand the lecture argument.

Step 1: introduce the encoder distribution inside the evidence. For a fixed observation x, choose a density $q_\phi(z \mid x)$ with suitable support. Then

$$\begin{aligned}\log p_\theta(x) &= \log \int q_\phi(z \mid x) \frac{p_\theta(x,z)}{q_\phi(z \mid x)}\, dz \\ &= \log \mathbb{E}_q\left[\frac{p_\theta(x,z)}{q_\phi(z \mid x)}\right] \\ &\geq \mathbb{E}_q\left[\log \frac{p_\theta(x,z)}{q_\phi(z \mid x)}\right].\end{aligned}$$

The last inequality is Jensen's inequality for the concave logarithm.

Step 2: separate decoder fit from prior matching. Using $p_\theta(x,z) = p_\theta(x \mid z)p(z)$,

$$\begin{aligned}\mathcal{E}(x) &= \mathbb{E}_q[\log p_\theta(x \mid z) + \log p(z) - \log q_\phi(z \mid x)] \\ &= \mathbb{E}_q \log p_\theta(x \mid z) - D_{\mathrm{KL}}(q_\phi(z \mid x)\|p(z)).\end{aligned}$$

Thus minimizing $-\mathcal{E}$ combines a negative log-likelihood reconstruction term with a KL term. For $p_\theta(x \mid z) = \mathcal{N}(f_\theta(z), \tau^2 I)$ of fixed variance, the reconstruction term is $\|x - f_\theta(z)\|^2/(2\tau^2)$ plus a constant. Its coefficient is determined by the likelihood.

Step 3: derive exactly what the lower bound leaves out. Bayes' rule gives $\log p_\theta(z \mid x) = \log p_\theta(x, z) - \log p_\theta(x)$. Therefore

$$\begin{aligned} D_{\mathrm{KL}}(q \| p_\theta(z \mid x)) &= \mathbb{E}_q[\log q - \log p_\theta(x, z) + \log p_\theta(x)] \\ &= \log p_\theta(x) - \mathcal{E}(x). \end{aligned}$$

The bound is tight precisely when the encoder matches the model posterior almost everywhere. Replacing the KL coefficient by an arbitrary β changes the objective; the ordinary evidence identity corresponds to $\beta = 1$. These steps explain the probabilistic objective underlying the handout's two-loss construction.

The KL Divergence

Deeper explanation: Entropy and cross-entropy have different arguments (p. 444)

Deeper explanation: Nonnegativity and asymmetry (p. 445)

Deeper explanation: Differential entropy and KL under coordinate changes (p. 449)

The KL divergence measures the difference between two probability distributions. For continuous distributions Q and P, it is defined as:

$$D_{KL}(Q \| P) = \int q(z) \log \frac{q(z)}{p(z)} dz = \mathbb{E}_{z \sim Q}\left[\log \frac{q(z)}{p(z)}\right]$$

Note that KL divergence is not symmetric ($D_{KL}(Q \| P) \neq D_{KL}(P \| Q)$). In the VAE context, we use $D_{KL}(q_\phi(z|x) \| p(z))$. This form penalizes q for placing probability mass where p has very little, which is desirable because we want the distributions seen during training to resemble the prior we will sample from at generation time.

For two multivariate Gaussian distributions, $q(z) = \mathcal{N}(\mu, \Sigma)$ and $p(z) = \mathcal{N}(0, I)$, the KL divergence has a convenient closed-form solution:

$$D_{KL}(\mathcal{N}(\mu, \Sigma) \| \mathcal{N}(0, I)) = \frac{1}{2}\left(\mathrm{tr}(\Sigma) + \mu^T \mu - k - \log \det(\Sigma)\right)$$

where k is the dimensionality of the latent space. This formula is fully differentiable with respect to μ and Σ.

Making Sampling Differentiable: The Reparameterization Trick

Deeper explanation: Reparameterization moves randomness outside the parameters (p. 447)

Deeper explanation: Gradients with a log-variance parameterization (p. 454)

To train the VAE with gradient descent, we must backpropagate through the sampling step $z \sim q_\phi(z|x)$. This is a stochastic node, which blocks gradients. The reparameterization trick reframes the sampling process to isolate the randomness.

Instead of sampling z directly from $\mathcal{N}(\mu, \Sigma)$, we:

1. Sample a random vector ϵ from a fixed, standard normal distribution: $\epsilon \sim \mathcal{N}(0, I)$.
2. Transform this random vector using the learned mean and covariance to get z: $z = \mu + L\epsilon$, where $\Sigma = LL^T$ (the Cholesky decomposition of Σ).

In this formulation, the randomness is an external input (ϵ), and the path from the encoder's outputs (μ, L) to z is deterministic and fully differentiable. Gradients can now flow from the decoder loss, through z, back to the parameters of the encoder. The stochasticity is treated like any other input data in an SGD setting; over many mini-batches, the expectation of the gradient is what we want.

Parameterizing the Covariance: The network must output a valid covariance matrix (positive definite when evaluated as a full-dimensional Gaussian density). A common technique is to have the network output a lower-triangular matrix L and compute the covariance as $\Sigma = LL^T$. An even simpler approach is to assume a diagonal covariance matrix, in which case the network just outputs log variances and exponentiates them to obtain positive diagonal entries.

Reparameterization as a diagonal covariance factor

For two latent coordinates with positive scales σ_1, σ_2, write

$$z = \begin{bmatrix}\mu_1 \\ \mu_2\end{bmatrix} + \underbrace{\begin{bmatrix}\sigma_1 & 0 \\ 0 & \sigma_2\end{bmatrix}}_{S} \begin{bmatrix}\epsilon_1 \\ \epsilon_2\end{bmatrix}, \qquad \epsilon \sim \mathcal{N}(0, I_2).$$

Conditioned on x, the covariance is $SS^\top = \operatorname{diag}(\sigma_1^2, \sigma_2^2)$. Diagonal covariance is an encoder modeling choice; a full-covariance Gaussian could instead use a triangular factor L and $z = \mu + L\epsilon$. Holding the sampled ϵ fixed during differentiation,

$$\frac{\partial z}{\partial(\mu, \sigma)} = \begin{bmatrix} I_2 & \operatorname{diag}(\epsilon) \end{bmatrix}.$$

If the network predicts log variances ℓ, then $\sigma_j = e^{\ell_j/2}$ and the second block becomes $\frac{1}{2}\operatorname{diag}(\sigma \odot \epsilon)$. This displays the two gradient paths explicitly. Conditional independence of latent coordinates given x does not imply that the aggregate latent distribution over different examples is independent.

Worked derivation: Gaussian KL and reparameterized gradients

Lecture basis: faculty handout 24, pp. 2–3. The steps below expand the lecture argument.

Step 1: expand the log-density ratio. Let $q = \mathcal{N}(\mu, \Sigma)$ with $\Sigma \succ 0$ and $p = \mathcal{N}(0, I_k)$. Gaussian normalization constants cancel except for the determinant:

$$\log \frac{q(z)}{p(z)} = -\tfrac{1}{2} \log\det \Sigma - \tfrac{1}{2}(z-\mu)^\top \Sigma^{-1}(z-\mu) + \tfrac{1}{2} z^\top z.$$

Step 2: evaluate the two quadratic expectations. Using $\mathbb{E}[(z-\mu)(z-\mu)^\top] = \Sigma$ and cyclic trace,

$$\begin{aligned} \mathbb{E}_q[(z-\mu)^\top \Sigma^{-1}(z-\mu)] &= \operatorname{tr}(\Sigma^{-1}\Sigma) = k, \\ \mathbb{E}_q[z^\top z] &= \operatorname{tr} \mathbb{E}_q[zz^\top] \\ &= \operatorname{tr}(\Sigma + \mu\mu^\top) = \operatorname{tr}\Sigma + \|\mu\|^2. \end{aligned}$$

Substitution yields

$$\boxed{D_{\mathrm{KL}}(q\|p) = \tfrac{1}{2}(\operatorname{tr}\Sigma + \|\mu\|^2 - k - \log\det\Sigma).}$$

For diagonal log variances $\ell_j = \log \sigma_j^2$, this becomes $\frac{1}{2}\sum_j (e^{\ell_j} + \mu_j^2 - 1 - \ell_j)$.

Step 3: differentiate a sample while fixing its external noise. Sample $\epsilon \sim \mathcal{N}(0, I)$ and set $z_j = \mu_j + e^{\ell_j/2}\epsilon_j$. Its mean is μ and its covariance is $\operatorname{diag}(e^\ell)$, so it has the required encoder distribution. For reconstruction gradient $g_j = \partial L_{\mathrm{recon}}/\partial z_j$,

$$\begin{aligned} \frac{\partial L_{\mathrm{recon}}}{\partial \mu_j} &= g_j, \\ \frac{\partial L_{\mathrm{recon}}}{\partial \ell_j} &= \tfrac{1}{2} g_j e^{\ell_j/2}\epsilon_j, \\ \frac{\partial D_{\mathrm{KL}}}{\partial \mu_j} &= \mu_j, \\ \frac{\partial D_{\mathrm{KL}}}{\partial \ell_j} &= \tfrac{1}{2}(e^{\ell_j} - 1). \end{aligned}$$

Add the reconstruction and appropriately weighted KL gradients, then backpropagate to the encoder parameters. Under conditions allowing differentiation under the expectation, averaging these noise-sample gradients gives the objective gradient. A straight-through derivative for rounding is a different, generally biased surrogate; it is not justified by this reparameterization identity.

Techniques from VAEs in Other Contexts

The ideas developed for VAEs are broadly applicable.

Quantization-Aware Training (QAT)

- **Problem:** To deploy large models on resource-constrained hardware, we often need to use lower-precision arithmetic (e.g., 8-bit integers instead of 32-bit floats). This process of converting high-precision weights to low-precision is called quantization. A naive post-training quantization can significantly degrade performance.
- **Challenge:** The rounding operation at the core of quantization is non-differentiable, so we cannot naively incorporate it into the training loop.
- **Solution (Straight-Through Estimator):** This technique is analogous to the reparameterization trick.
  - **Forward Pass:** The model's weights are quantized (e.g., rounded to the nearest representable value) before being used in computations.
  - **Backward Pass:** During backpropagation, the gradient is passed through the quantization node as if it were an identity function. We pretend the non-differentiable quantization step didn't happen.

  This allows the model to learn weights that are robust to the effects of quantization, improving the performance of the final quantized model.

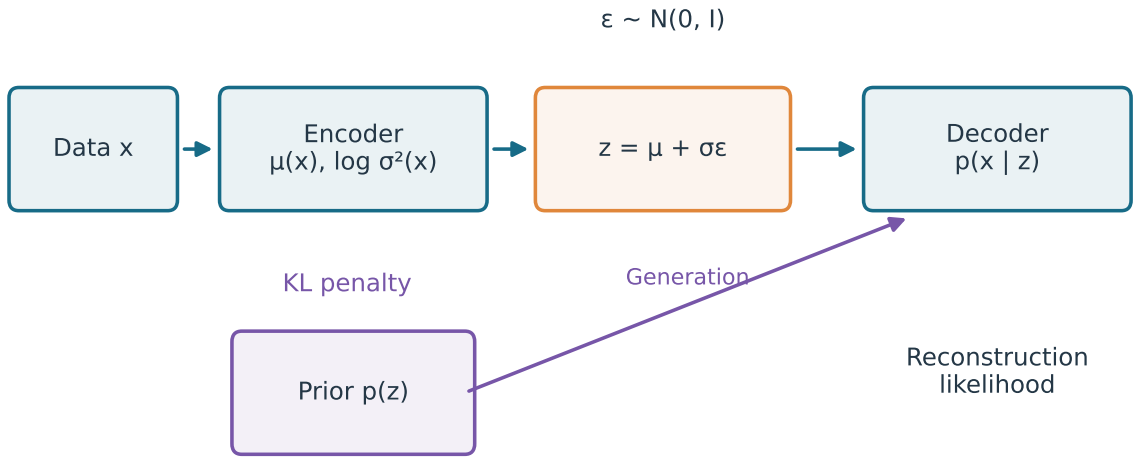


Figure 23.3: The encoder supplies a distribution over latent variables during training. Generation starts from the prior and uses only the decoder.

23.3 Post-Training of Large Language Models

Deeper explanation: Deriving the trajectory policy gradient (p. 464)

After a base language model is pre-trained via next-token prediction, it undergoes several stages of "post-training" to align it with human preferences and improve its problem-solving abilities.

Recap: Supervised Fine-Tuning (SFT)

The first step is often SFT, where the model is fine-tuned on a high-quality dataset of instruction-response pairs. This teaches the model to follow instructions and act as a helpful assistant.

Improving Problem-Solving Capabilities

Simply training on question-answer pairs is often not enough for complex reasoning tasks. Two key dimensions for improvement have emerged:

1. **Test-Time Compute Scaling:** Spending more computational effort during inference to generate a better answer.
2. **Training for Better Compute Utilization:** Training the model specifically to make better use of this increased inference-time compute. This is where methods like Reinforcement Learning from Verifiable Rewards (RLVR) and Reinforcement Learning from Human Feedback (RLHF) come in.

Test-Time Compute Scaling These are methods that improve performance without changing the model's weights.

- **Chain-of-Thought (CoT) Prompting:** Encouraging the model to "think step by step" by including this phrase in the prompt. This causes the model to generate a rationale before the final answer, which often improves accuracy on reasoning tasks.
- **Best-of-N Sampling:** Generate N different responses to the same prompt (e.g., by using a non-zero temperature) and then select the best one. The challenge is the selection criterion when the ground truth answer is unknown:
  - **Majority Vote:** If the answer format is constrained (e.g., multiple choice, a single number), pick the answer that appears most frequently among the N samples.
  - **Model-as-Critic:** Use another (or the same) LLM to rate or critique the N responses and select the one with the highest score.
  - **Highest Probability:** Select the full sequence (response) to which the model assigned the highest cumulative probability during generation.

Advanced Sampling and The Role of RL

Recent research investigates the connection between post-training and sampling.

- **Observation:** Models that have undergone RLHF/RLVR tend to produce samples that were already among the higher-probability sequences of the original, base model. This suggests that RL might be functioning as a sophisticated sampler, shifting probability mass towards pre-existing "good" regions of the output space.
- **Question:** Can we achieve the benefits of RL post-training with just better sampling methods at inference time, without retraining the model?
- **New Approach: Distribution Tilting:** Sample from a modified probability distribution that amplifies the probability of high-likelihood sequences. This can be done by sampling from a distribution proportional to $p(x)^\gamma$, where $p(x)$ is the original model's sequence probability and $\gamma > 1$.
  - This is related to, but distinct from, low-temperature sampling. Temperature affects each token's probability distribution individually, whereas this tilting applies to the entire sequence probability.
  - Sampling from this new (unnormalized) distribution is non-trivial and may require techniques like Markov Chain Monte Carlo (MCMC).
- **Early Results:** Recent work has shown that these advanced sampling techniques alone can significantly boost model performance on reasoning benchmarks, closing a large portion of the gap between the base SFT model and a fully RL-tuned model. This opens up a new avenue of research teasing apart the effects of training versus sampling in aligning LLMs.

Lecture 24

Post-Training and Test-Time Compute

Course sources: lecture_25; 2025-11-25 transcript. Lecture numbering follows the original reader; linked faculty handouts identify the overlapping source coverage.

24.1 Announcements (Fall 2025)

These announcements describe the original Fall 2025 lecture context.

- **Course Survey:** Please fill out the course survey. If 75% of the class completes it by Friday, everyone will receive 3% extra credit. This feedback is valuable.
- **Project Peer Reviews:** Draft projects have been submitted. Everyone has been assigned one review. Please complete your review carefully and provide constructive feedback. The deadline is today, so that feedback can be released before the Thanksgiving break. The goal is to help improve everyone's projects for the final submission.

24.2 Recap and Lecture Context

Deeper explanation: An evaluation table that separates the mechanisms (p. 461)

We have covered various architectures and general deep learning concepts. We began discussing generative models, starting with Variational Autoencoders (VAEs). We are now taking a brief interlude to discuss a critical and current topic: **post-training of models and test-time compute**. These concepts will also be relevant when we discuss diffusion models next week.

The context for this discussion is Large Language Models (LLMs). A model is pre-trained on a vast dataset, but this pre-training distribution may not align perfectly with desired real-world applications, such as instruction following. To bridge this gap, post-training techniques are employed.

Supervised Fine-Tuning (SFT)

One of the first steps in post-training is Supervised Fine-Tuning. This is similar to the pre-training objective (cross-entropy loss), but with a crucial difference: the loss is only computed on the *answer* tokens, not the prompt (question) tokens. This is often called a **masked loss**.

$$\textit{Prompt Tokens} \rightarrow \textbf{Model} \rightarrow \textit{First Response Token, Second Response Token, ...}$$

(No Loss) (Compute Cross-Entropy Loss on each generated token)

24.3 Improving Problem-Solving in Models

Deeper explanation: Best-of-many requires a selection mechanism (p. 460)

A key advance for LLMs was the concept of "Chain-of-Thought," where models are prompted or trained to show their work. The lecture discussed settings in which generating intermediate reasoning improved measured performance; longer output alone is not a guarantee of correctness. This leads to a central question: **How can we make a model better at solving problems?**

The answer has two parts:

A) Be willing to spend more computational resources at inference time. This is known as **test-time compute**.

B) Train the model to be better at this reasoning process.

24.4 Test-Time Compute Strategies

Deeper explanation: Beam search with explicit termination (p. 459)

Spending more compute when a query is made can yield better answers from a pre-trained model.

1. **Prompting:** The simplest approach. Prompt the model with phrases like "Think step by step" or "Be careful." This often encourages more detailed and accurate responses.
2. **Repeated Generation (Best-of-N):** Instead of generating one response, generate N responses by sampling from the model. Since sampling introduces randomness, the responses will differ. Then, select the best one using a criterion:
   - **Majority/Plurality Vote:** If the problem has a discrete, verifiable answer (e.g., A, B, C, D), you can take the most frequent answer among the N generations.
   - **Take Highest Probability:** Choose the generated sequence $\vec{x}$ that has the highest probability $P(\vec{x})$ under the model.
   - **Train a Reward Model:** Train a separate model to score or "grade" each of the N responses. Then, select the response with the highest score from this reward model.
3. **Sample Better:** Instead of just sampling and picking, can we modify the sampling process itself to inherently produce better responses?

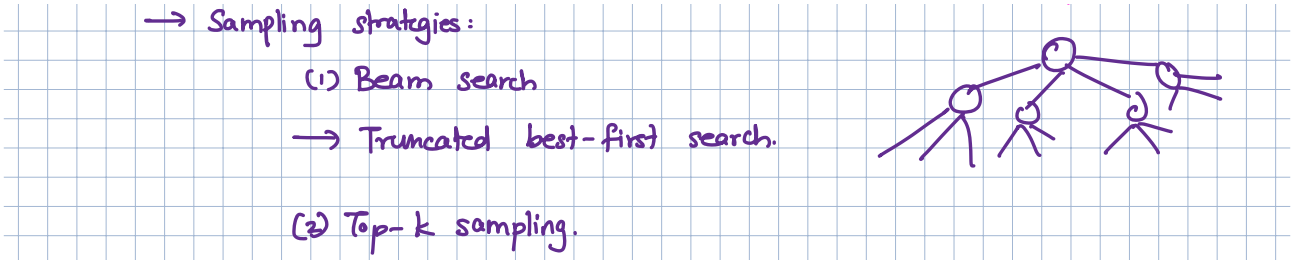


Figure 24.1: Beam search as a branching prefix tree. The tree tracks candidate prefixes as new tokens are appended. Keeping several prefixes is a search strategy; it is not equivalent to lowering token temperature or independently drawing several complete responses. Original annotated source excerpt; Gireeja Ranade, faculty handout 20, p. 5.

24.5 Reasoning with Sampling

Deeper explanation: Why cut-specific power sampling preserves its target (p. 461)

Deeper explanation: The exact next-token marginal of a powered sequence model (p. 457)

Deeper explanation: Detailed balance explains Metropolis-Hastings (p. 458)

A recent paper, "Reasoning with Sampling: Your base model is smarter than you think" (Karan & Du, 2025), explores the idea of improving reasoning without further training.

Empirical Observation and Intuition

It was empirically observed that when models were fine-tuned for better reasoning (e.g., with reinforcement learning), the improved responses they generated often had a *higher probability under the original base model* as well.

This leads to the core idea of the paper: Can we bypass the fine-tuning and get better reasoning simply by finding higher-probability sequences from the base model directly?

Intuition: When a human (or a model) makes a reasoning error, they might "talk themselves into a corner." Subsequent steps become difficult and awkward. For an LLM, this translates to a sequence of low-probability token choices. The entire sequence, being a product of these probabilities, will thus have a very low total probability. Conversely, a correct and coherent line of reasoning might correspond to a higher-probability path.

Goal: Sample from a Sharpened Distribution

The goal is to sample from a distribution that puts more mass on high-probability sequences. We can achieve this by modifying the original probability distribution $P(\vec{x})$:

$$P_{\text{desired}}(\vec{x}) \propto P(\vec{x})^{\alpha}, \quad \text{where } \alpha > 1$$

Since probabilities are less than 1, raising them to a power $\alpha > 1$ suppresses smaller probabilities more than larger ones. The result must be re-normalized to be a valid distribution.

Is This Just Low-Temperature Sampling?

No. Low-temperature sampling and sampling from $P(\vec{x})^\alpha$ are not the same. Let's analyze this mathematically.

To generate a sequence token by token, we need the conditional probability of the next token x_t given the past $x_{0..t-1}$.

The **desired distribution** for x_t requires marginalizing (summing over) all possible future continuations of the sequence:

$$P_{\text{desired}}(x_t|x_{0..t-1}) \propto \sum_{x_{t+1..T}} (P(x_{0..T}))^\alpha = \sum_{x_{t+1..T}} \left(\prod_{i=0}^{T} P(x_i|x_{<i})\right)^\alpha$$

The normalization constant would be a sum over all possible choices for $\tilde{x}_t$.

$$P_{\text{desired}}(x_t|x_{0..t-1}) = \frac{\sum_{x_{t+1..T}} (P(x_{0..T}))^\alpha}{\sum_{\tilde{x}_t} \sum_{x_{t+1..T}} (P(x_{0..t-1}, \tilde{x}_t, x_{t+1..T}))^\alpha}$$

Notice that the powers are inside the sum over future continuations.

In contrast, **low-temperature sampling** works differently. It sharpens the distribution at each step greedily, without considering the full sequence probability:

$$P_{\text{low-temp}}(x_t|x_{0..t-1}) \propto (P(x_t|x_{0..t-1}))^\alpha$$

This is equivalent to raising each conditional probability term to the power α inside the product:

$$P_{\text{low-temp}}(x_t|x_{0..t-1}) \propto \frac{(P(x_t|x_{0..t-1}))^\alpha \left(\sum_{x_{t+1..T}} \prod_{i=t+1}^{T} P(x_i|x_{<i})\right)^\alpha}{\dots}$$

The key difference is that raising a sum to a power is not the same as the sum of powers: $(\sum a_i)^\alpha \neq \sum (a_i)^\alpha$. Therefore, low-temperature sampling is an approximation that does not correctly sample from the desired distribution over full sequences.

MCMC for Sampling

Since computing the normalization constant for P_{desired} is intractable, we need a different approach. **Markov Chain Monte Carlo (MCMC)** is a class of algorithms designed for sampling from a distribution when we only have access to a function proportional to its probability density.

The idea is to construct a Markov chain whose stationary distribution is our target distribution, P_{desired}. Under suitable irreducibility and aperiodicity conditions, the distribution of the state approaches the stationary distribution. A finite run is generally approximate and can mix slowly.

The "compute" in test-time compute is spent taking these MCMC steps.

The Power Sampling Algorithm

The paper proposes an algorithm that uses MCMC with a block-wise generation strategy to make it more efficient.

1. **Initialization:** Start with a short prefix.
2. **Block-wise Iteration:** Grow the sequence in blocks of size B. For each block $k = 0, 1, \ldots$:
   (a) **Extend:** Initialize the next block by sampling autoregressively from the base model.
   (b) **MCMC Steps:** For N_{MCMC} steps:
      i. Pick a random position m within the current total length $(k+1)B$.
      ii. **Propose a move:** Create a new candidate sequence x' by resampling from position m to the end using the base model's proposal distribution P_{prop}.
      iii. **Accept/Reject:** Compute the acceptance probability $A(x', x)$ using a Metropolis-Hastings update rule.

$$A(x', x) = \min\left(1, \frac{\pi_k(x')P_{prop}(x|x')}{\pi_k(x)P_{prop}(x'|x)}\right)$$

      where $\pi_k(x) \propto P(x)^\alpha$ is our unnormalized target distribution. When the suffix proposal is exactly the base autoregressive model, the proposal ratio cancels one power of the target ratio, leaving:

$$A(x', x) = \min\left(1, \left(\frac{P(x')}{P(x)}\right)^{\alpha-1}\right)$$

      For the base-model proposal and $\alpha > 1$, this means a move to a higher probability sequence ($P(x') > P(x)$) is **always accepted**. A move to a lower probability sequence is accepted stochastically.
      iv. Accept the new sequence x' with probability $A(x', x)$, otherwise keep the old sequence x.
   (c) **Grow again:** Retain the current sequence as a warm start and extend its length. Later cuts may revisit any previously generated position; only the conditioning prompt is permanently fixed.
3. **Return:** Output the final generated sequence.

Worked derivation: why sequence powering is not token temperature

Lecture basis: faculty handout 25, pp. 2–3. The steps below expand the lecture argument.

Step 1: compute the actual conditional of a powered sequence law. On a finite set of complete sequences, let $q(y) = p(y)^\alpha / Z$, $\alpha > 0$. Given prefix h and a candidate next token a, sum over all completions z:

$$\begin{aligned} q(a \mid h) &= \frac{\sum_z p(h, a, z)^\alpha}{\sum_{b,z} p(h, b, z)^\alpha} \\ &= \frac{p(a \mid h)^\alpha C(h, a)}{\sum_b p(b \mid h)^\alpha C(h, b)}, \\ C(h, a) &= \sum_z p(z \mid h, a)^\alpha. \end{aligned}$$

The common prefix factor $p(h)^\alpha$ cancels. Tokenwise temperature uses only $p(a \mid h)^\alpha$ and omits the completion factor $C(h, a)$. Equality requires those factors to be the same across candidate next tokens.

Step 2: calculate a two-step counterexample. Suppose first tokens A, B each have probability $1/2$. After A, there are two equiprobable completions; after B, one completion is certain. The three complete-sequence probabilities are $1/4, 1/4, 1/2$. With $\alpha = 2$, their unnormalized powers are $1/16, 1/16, 1/4$, so

$$q(A) = \frac{1/8}{3/8} = \frac{1}{3}, \qquad q(B) = \frac{2}{3}.$$

Tokenwise temperature leaves the first-token probabilities at $1/2, 1/2$. The difference appears even before generating any completion.

Step 3: derive an MCMC acceptance ratio. For a proposal $r(y' \mid y)$, Metropolis–Hastings uses

$$a(y, y') = \min\left\{1, \frac{p(y')^\alpha r(y \mid y')}{p(y)^\alpha r(y' \mid y)}\right\}.$$

The unknown normalization Z cancels. Indeed both directions have stationary probability flow

$$q(y) r(y' \mid y) a(y, y') = \min\{q(y) r(y' \mid y), q(y') r(y \mid y')\},$$

which is symmetric in y, y'. This proves detailed balance. With independent base-model proposals $r(y' \mid y) = p(y')$, the ratio simplifies to $[p(y')/p(y)]^{\alpha-1}$. Finite runs still have mixing and initialization error; a valid stationary distribution is not a guarantee of rapid sampling.

Results and Critique

The paper shows significant performance gains on reasoning benchmarks without any fine-tuning. For example, on MATH500, the base model (Qwen2.5-7B) achieves 49.6% accuracy, while this sampling method boosts it to 74.8%, which is comparable to a model fine-tuned with RL (GRPO) that gets 78.5%.

Critique (What's missing?): A critical analysis of the paper's results raises questions about missing comparisons that would provide a fuller picture:

- **Best-of-N Sampling:** The most direct baseline is to generate N samples and pick the one with the highest probability. How does MCMC compare to this simpler method for the same amount of compute?
- **Beam Search:** Another standard method for finding high-probability sequences.
- **Prompt Optimization:** Could a better prompt have elicited similar gains from the base model?
- **Relative Compute Cost:** The comparison should be normalized by the amount of compute used for each method.
- **Error Bars:** To assess statistical significance.

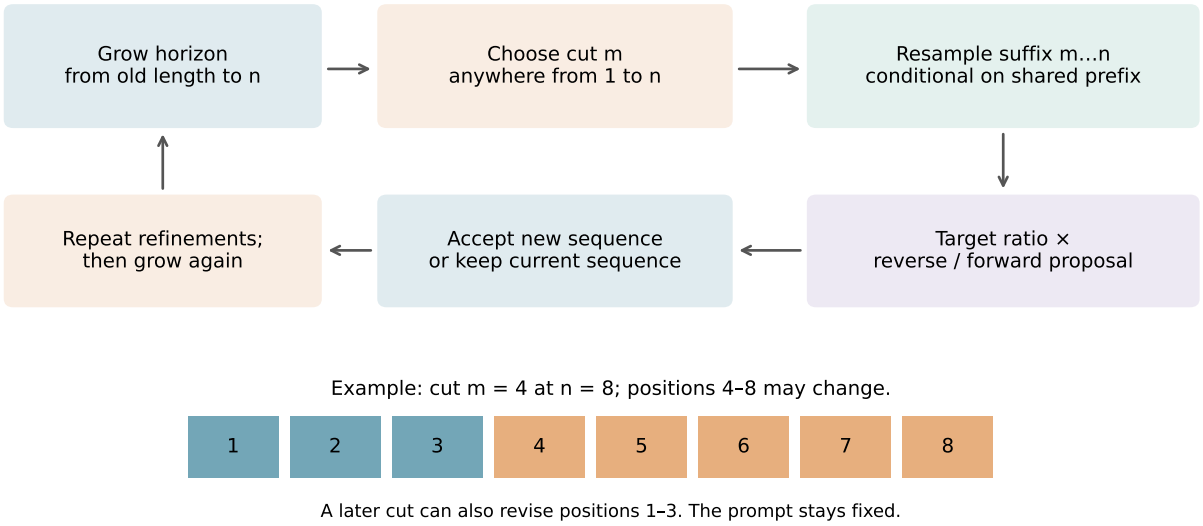


Figure 24.2: Grow the horizon, then refine with cuts anywhere in the generated sequence. Earlier generated blocks can be revised at later stages; only the conditioning prompt remains fixed.

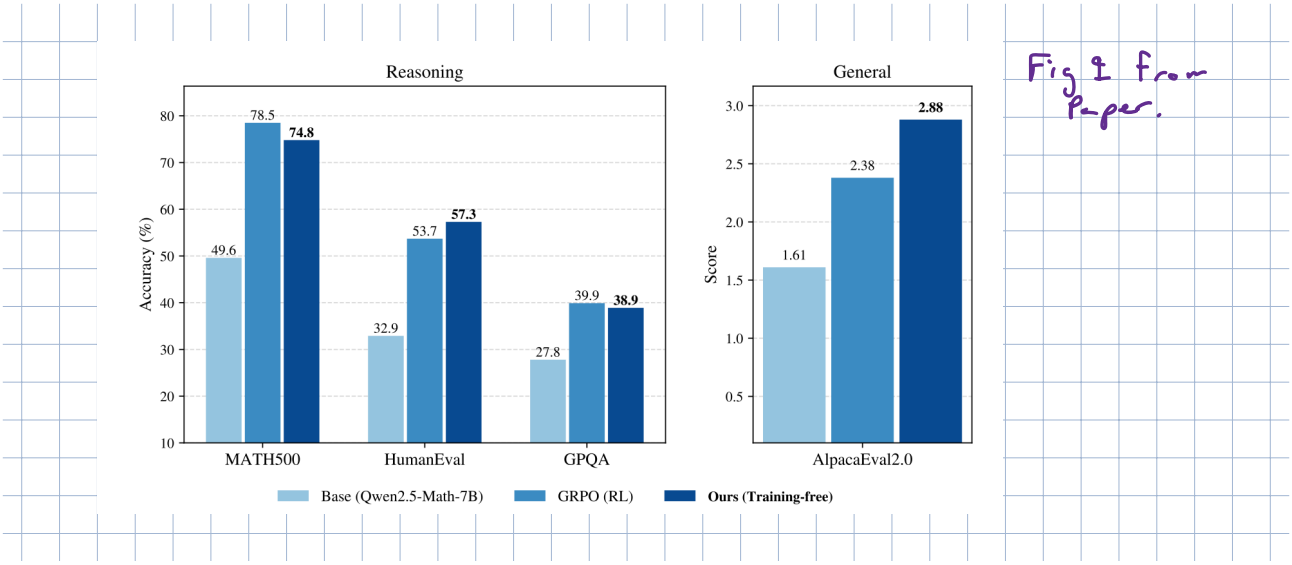


Figure 24.3: The reported power-sampling comparison. The bars summarize the specific experiment discussed in the lecture. Compare sampling budget and evaluation setup before reading a score improvement as a general replacement for reinforcement learning. Original annotated source excerpt; Anant Sahai, faculty handout 25, p. 4.

24.6 Reinforcement Learning with Verifiable Rewards (RLVR)

Deeper explanation: PPO clipping and stopped weighting have different derivatives (p. 465)

Deeper explanation: Standard PPO is a useful comparison, with a different formula (p. 463)

As distinct from Reinforcement Learning from Human Feedback (RLHF), RLVR is a setting where we have an automated, programmatic way to verify if a model's response is correct. This is common in domains like math (checking the final answer) or code generation (running unit tests). This simpler setting is a good way to understand the mechanics of RL-based fine-tuning.

Scaling Laws for RL

A key paper, "The Art of Scaling Reinforcement Learning for LLMs" (discussed in the Fall 2025 lecture), investigates how RL performance scales with compute. They find that the pass rate (a measure of performance) often follows a predictable S-shaped curve when plotted against GPU hours on a log scale.

$$R_c = \frac{A - R_0}{1 + (C_{mid}/C)^B} + R_0$$

The fitted curve provides an empirical extrapolation under the measured setup, not a convergence guarantee. This is crucial for deciding which methods are worth scaling up.

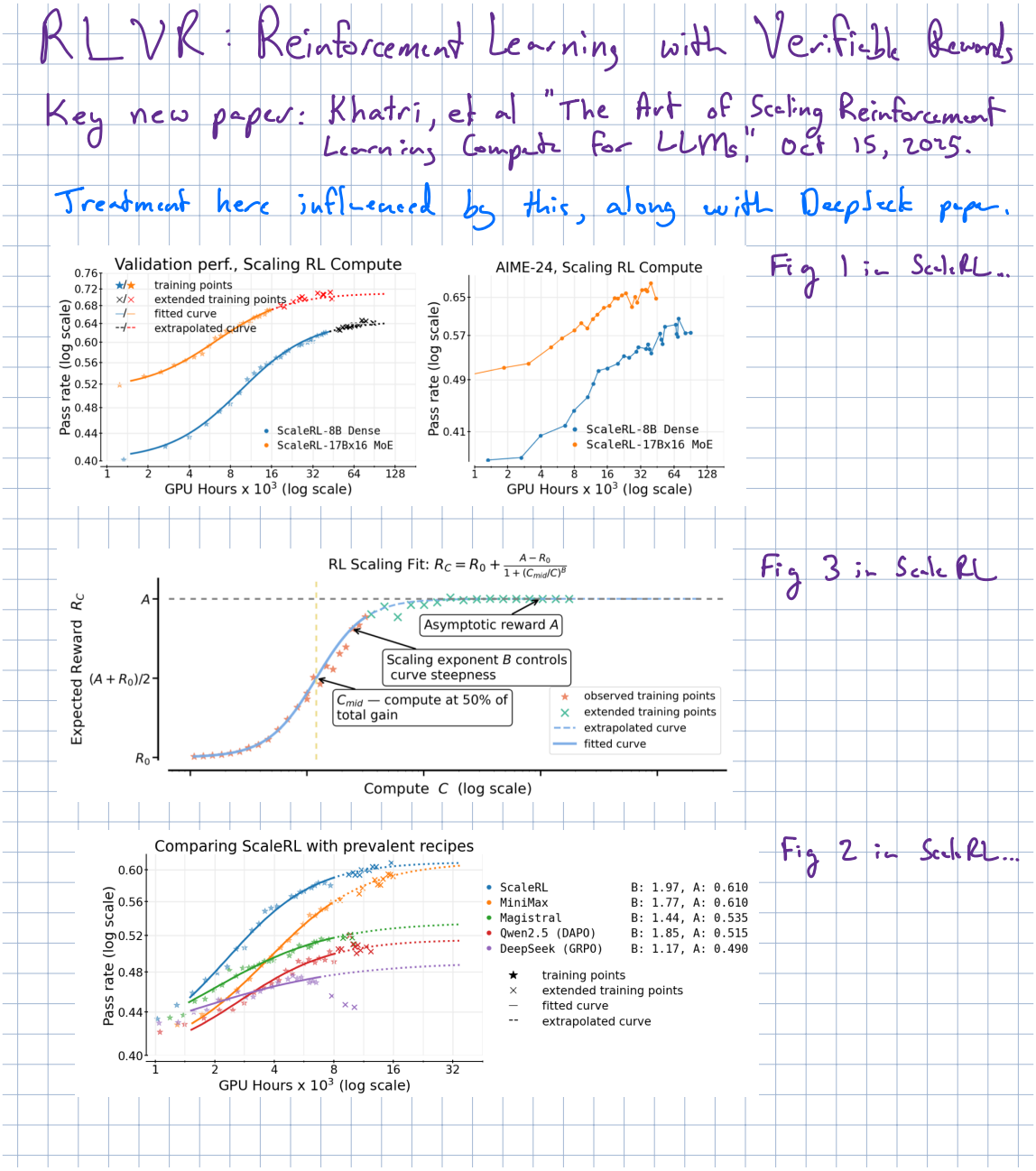


Figure 24.4: The RLVR scaling curves shown in class. The panels compare performance against training compute and fitted trends. These empirical curves motivate the lecture discussion; extrapolation depends on the reward, data, optimization recipe, and compute regime. Original annotated source excerpt; Anant Sahai, faculty handout 25, p. 5.

The ScaleRL Loss Function

The paper proposes a recipe called ScaleRL. Its loss function looks complex, but we can break it down.

$$J_{\text{ScaleRL}}(\theta) = \mathbb{E}_{\substack{x\sim D,\\ \{y_i\}_{i=1}^{G}\sim\pi_{\text{gen}}^{\theta}(\cdot|x)}}\left[\frac{1}{|G|}\frac{1}{\sum_{g=1}^{G}|y_g|}\sum_{g=1}^{G}\sum_{t=1}^{|y_g|}\text{sg}(\min(\rho_{g,t},\epsilon))\hat{A}_g^{\text{norm}}\log\pi_{\text{train}}^{\theta}(y_{g,t}|y_{g,<t})\right]$$

The lecture discusses filtering easy prompts using a pass-rate threshold. Such filtering changes the sampled prompt distribution. The displayed reduction convention includes both a group and token normalization; implementations must state their chosen weighting. Guard zero reward variance when computing normalized advantages.

Let's unpack the terms:

- $\mathbb{E}[\dots]$: The expectation is taken over prompts x from the dataset and generations (rollouts) y_i from a **generation policy** π_{gen}.
- $\log\pi_{\text{train}}^{\theta}(\dots)$: This is the log-probability of a token under the current **model being trained**. The gradient of the loss will be taken with respect to this term's parameters θ. This is the standard "log-probability trick" (from the REINFORCE algorithm) that allows gradients to be estimated for non-differentiable rewards.
- $\hat{A}_g^{\text{norm}}$: **Normalized Advantage**. This is the core reward signal. For each generation y_g in a batch of G generations:

  1. Compute its reward r_g (e.g., 1 if correct, 0 if incorrect).
  2. Compute the mean $\bar{r}$ and standard deviation σ_r of rewards across the batch.
  3. The normalized advantage is $\hat{A}_g^{\text{norm}} = (r_g - \bar{r})/\sigma_r$.

  This normalization centers the rewards, so correct answers get a positive signal (increasing their probability) and incorrect answers get a negative signal (decreasing their probability).

- $\rho_{g,t}$: **Importance Sampling Ratio**. A problem arises because the generations y_g are sampled from π_{gen}, but the gradient is applied to π_{train}. If these policies are different (e.g., π_{gen} is a slightly older, "stale" version of the model), the expectation is mismatched. Importance sampling corrects for this by re-weighting each sample:

  $$\rho_{g,t} = \frac{\pi_{\text{train}}^{\theta}(y_{g,t}|y_{g,<t})}{\pi_{\text{gen}}^{\theta}(y_{g,t}|y_{g,<t})}$$

  This ratio accounts for how much more or less likely a given token is under the current model versus the model that generated it.

- $\min(\rho_{g,t},\epsilon)$: **Clipping**. The minimum caps the importance ratio from above at ϵ. It limits unusually large corrective weights; it does not impose a lower floor. The course objective is maximized, or its negative is minimized.

- $\mathrm{sg}(\dots)$: **Stop Gradient**. This instructs the automatic differentiation framework (e.g., PyTorch) not to backpropagate gradients through the importance sampling term. This term is just a corrective weight, not part of the objective to be optimized via gradient descent. The gradient should only flow into the $\log \pi^{\theta}_{\text{train}}$ term.
- **Bag-of-Tokens Perspective:** The loss is calculated per-token and averaged over all tokens in all generations, rather than treating each sequence as a single unit.

This is the course's clipped reward-weighted training recipe. Its stability and effectiveness depend on reward quality, sampling, normalization, and the optimization settings.

Worked derivation: score-function gradients and importance ratios

Lecture basis: faculty handout 25, pp. 6. The steps below expand the lecture argument.

Step 1: differentiate an expectation over sampled outputs. For fixed reward $R(y)$ and a differentiable discrete policy with fixed support,

$$\begin{aligned} J(\theta) &= \sum_y p_\theta(y) R(y), \\ \nabla J &= \sum_y R(y) \nabla p_\theta(y) \\ &= \sum_y p_\theta(y) R(y) \nabla \log p_\theta(y) \\ &= \mathbb{E}_{p_\theta}[R(y) \nabla \log p_\theta(y)]. \end{aligned}$$

This is a score-function estimator, not a pathwise reparameterization of discrete tokens. If the reward explicitly depends on θ, its derivative adds another term.

Step 2: expand the log probability across tokens. For an autoregressive sequence,

$$\nabla \log p_\theta(y \mid x) = \sum_t \nabla \log p_\theta(y_t \mid x, y_{<t}).$$

A terminal reward can therefore weight every sampled token's score. A prompt-only baseline $b(x)$ does not change the expectation because

$$\mathbb{E}[b(x) \nabla \log p_\theta(y \mid x)] = b(x) \nabla \sum_y p_\theta(y \mid x) = 0.$$

A baseline depending on the sampled action need not cancel. For a group mean including the same reward, centering can introduce a scale bias; a leave-one-out baseline avoids this dependence under independent samples.

Step 3: change the sampling distribution. If samples come from p_{old} with support covering the new policy,

$$\mathbb{E}_{p_\theta}[f(y)] = \sum_y p_{\text{old}}(y) \frac{p_\theta(y)}{p_{\text{old}}(y)} f(y).$$

The full sequence ratio is a product of token ratios. Tokenwise clipped surrogates use a different, deliberately modified objective to control update size. Clipping is not an algebraic identity that preserves the original unbiased importance estimator. Treating reward, sampled data, old-policy probabilities, or advantages as stop-gradient quantities must be stated when differentiating such a surrogate.

Lecture 26

RL Post-Training and Diffusion Models

Course sources: lecture_26; 2025-12-02 transcript.

26.1 Announcements and Logistics (Fall 2025)

These announcements describe the original Fall 2025 lecture context.

- **Course Topics:** This week finishes RL Post-Training and introduces Diffusion Models, the final topic for the course. A new homework on diffusion models will be released.
- **Survey Extra Credit:** The class survey participation reached 40%, below the 75% target. However, as a thank you, students who completed the survey will receive 1.5% extra credit (prorated from the originally proposed 3%).
- **Final Project Deadlines:**
  - **Final Project Report:** Deadline extended to Sunday, December 7th.
  - **Poster Session:** Tuesday, December 9th, during class time in the Wozniak Lounge. This is a required event. Teams should be present at their posters, especially when staff members visit.
  - **Peer Reviews:** You will be assigned peer reviews of other final reports. These are due by Thursday, December 11th. The feedback is crucial for the final submission.
  - **Final Final Report:** Due Sunday, December 14th. This is the latest possible date before final exam week.
- **Final Exam:**
  - **Date and Time:** Wednesday, December 17th, at 8 AM. Room details to be announced.
  - **Format:** 3 hours long. No notes, no clarifications will be given during the exam. All necessary information will be provided on the exam paper itself.
  - **Preparation:** The best way to prepare is to have internalized the material from homeworks and discussions throughout the semester. The final exam from the previous semester will be posted, but most problem types will have been seen on homeworks.
- **Participation Grading:** A self-grading form will be released. It will involve uploading a picture of yourself and indicating which discussions you attended. For ED participa-

tion credit, you can attach screenshots of your posts. Special participation reports should be uploaded to the designated Gradescope assignments.

26.2 Finishing RL Post-Training

Deeper explanation: Reward learning from paired comparisons (p. 466)

Deeper explanation: What post-training can be evaluated to improve (p. 468)

We recall the basic pipeline for large language models:

1. **Pre-training:** Unsupervised learning on a massive corpus of text.
2. **Supervised Fine-Tuning (SFT):** Fine-tuning the model on instruction-response pairs to teach it to follow instructions. This typically involves maximizing the log-probability of the desired response tokens using a cross-entropy loss (often with masking so there is no loss on the prompt tokens).
3. **Post-training:** Further training to make the model better at solving problems, often using Reinforcement Learning (RL) techniques. This phase focuses on improving quality, alignment, and helpfulness, and often involves spending more compute at generation time.

RL with Verifiable Rewards (RLVR) Recap

Deeper explanation: Why a same-group mean is not an independent baseline (p. 464)

Deeper explanation: The score-function identity (p. 446)

Deeper explanation: Importance sampling and the meaning of stop-gradient (p. 448)

Deeper explanation: Variance reduction for score-function estimators (p. 449)

Deeper explanation: Importance weighting, support, and effective sample size (p. 450)

RLVR is used when we have an automated way (an "autograder") to score the quality of a model's generation for a given prompt. The core idea is to generate many candidate responses, score them, and then update the model to increase the probability of high-scoring responses and decrease the probability of low-scoring ones.

The **ScaleRL** paper proposes the following objective function:

$$J_{\text{ScaleRL}}(\theta) = \mathbb{E}_{\substack{x\sim D_x \\ \{y_i\}_{i=1}^{G}\sim\pi^{\theta}_{\text{gen}}(\cdot|x)}}\left[\sum_{i=1}^{G}\sum_{t=1}^{|y_i|}\text{sg}\big(\min(\rho_{i,t},\epsilon)\big)\,\hat{A}_i^{\text{norm}}\log\pi^{\theta}_{\text{train}}(y_{i,t}|y_{i,<t})\right] \tag{26.1}$$

This objective is maximized; minimizing its negative gives the equivalent loss. The displayed sum is a convention: averaging tokens or sequences changes the weighting and must be specified. If a reward group has zero standard deviation, define a guarded normalization or skip its zero-advantage update.

Let's unpack the components:

- $\pi^{\theta}_{\text{train}}$ is the policy (model) we are currently training. $\pi^{\theta}_{\text{gen}}$ is the policy used to generate the responses. In practice, these are different due to engineering constraints (generation servers lag behind the training server). $\pi^{\theta}_{\text{gen}}$ tracks $\pi^{\theta}_{\text{train}}$ with a delay.
- $\hat{A}_i^{\text{norm}}$ is the normalized advantage for generation y_i. It is calculated by taking the reward r_i for generation y_i, de-meaning it across all G generations for the same prompt, and normalizing by the standard deviation. This tells us how much better or worse a generation is compared to the average.

$$\hat{A}_i = r_i - \text{mean}(\{r_j\}_{j=1}^{G}), \quad \hat{A}_i^{\text{norm}} = \frac{\hat{A}_i}{\text{std}(\{r_j\}_{j=1}^{G})}$$

- The model takes a "bag of tokens" perspective, where every token in a successful generation is considered "good" and inherits the positive advantage of the whole sequence.
- $\rho_{i,t}$ is the importance sampling ratio. It corrects for the mismatch between the generation policy and the training policy.

$$\rho_{i,t} = \frac{\pi^{\theta}_{\text{train}}(y_{i,t}|y_{i,<t})}{\pi^{\theta}_{\text{gen}}(y_{i,t}|y_{i,<t})}$$

  The generation policy is held fixed. In the course surrogate, the entire clipped weighting factor is stopped, so differentiation acts only on the log-policy term.

- $\min(\rho_{i,t}, \epsilon)$ clips the importance weight. If the training policy assigns a much higher probability to a token than the generation policy did, this ratio can become very large, leading to unstable updates. Clipping prevents this.

KL Regularization

A common technique in RL-based fine-tuning is to add a KL-divergence penalty to the objective. This penalizes divergence from a reference policy, which helps mitigate catastrophic forgetting. The objective becomes finding a policy π that maximizes the expected reward while staying close to a reference policy π_{ref}:

$$\arg\max_{\pi} \mathbb{E}_{\substack{x, \\ y\sim\pi(\cdot|x)}}[r(x,y)] - \beta D_{KL}[\pi(\cdot|x)||\pi_{\text{ref}}(\cdot|x)] \tag{26.2}$$

where β is a hyperparameter controlling the strength of the regularization.

The solution to this optimization problem has the form:

$$\pi(y|x) = \frac{1}{Z(x)}\pi_{\text{ref}}(y|x)e^{\frac{1}{\beta}r(x,y)} \tag{26.3}$$

Here, $Z(x)$ is a normalization constant. This solution intuitively states that the optimal policy should be the reference policy, re-weighted to put more probability mass on responses with higher rewards.

Worked derivation: the KL-regularized optimal policy

Lecture basis: faculty handout 26, pp. 4. The steps below expand the lecture argument.

Step 1: write the constrained optimization for one prompt. Let $p(y) = \pi_{\text{ref}}(y \mid x) > 0$ on a finite support, $r_y = r(x, y)$, and $\beta > 0$. Optimize

$$J(q) = \sum_y q_y r_y - \beta \sum_y q_y \log(q_y/p_y), \qquad \sum_y q_y = 1.$$

Step 2: differentiate the Lagrangian. For multiplier λ, use $\mathcal{L} = J(q) + \lambda(\sum_y q_y - 1)$. At an interior optimum,

$$\begin{aligned} 0 &= r_y - \beta[\log(q_y/p_y) + 1] + \lambda, \\ \log(q_y/p_y) &= r_y/\beta + \lambda/\beta - 1, \\ q_y &= p_y e^{r_y/\beta} e^{\lambda/\beta - 1}. \end{aligned}$$

Normalization determines $e^{\lambda/\beta-1} = 1/Z$, where $Z = \sum_y p_y e^{r_y/\beta}$. Thus $q^*_y = p_y e^{r_y/\beta}/Z$.

Step 3: verify global optimality without relying only on stationarity. Substitute $\log q^*_y = \log p_y + r_y/\beta - \log Z$:

$$\begin{aligned} \beta D_{\text{KL}}(q \| q^*) &= \beta \sum_y q_y \log(q_y/p_y) - \sum_y q_y r_y + \beta \log Z, \\ J(q) &= \beta \log Z - \beta D_{\text{KL}}(q \| q^*) \le \beta \log Z. \end{aligned}$$

Equality holds exactly at $q = q^*$. This proves the reward-tilting formula and its uniqueness on the support. Zero reference probabilities cannot gain positive mass while keeping this KL finite. On infinite supports the partition function must also be finite.

RL from Human Feedback (RLHF)

In many cases, we don't have a verifiable, automated reward function. RLHF uses human preferences to guide the model.

Key Challenges & Solutions:

1. **Humans are expensive:** It's not feasible to get human feedback for every single generation during RL training. **Solution:** Use human feedback to train an AI **reward model** (RM). This RM then acts as a proxy for the human, providing learned reward scores during RL; these may be terminal sequence scores rather than dense per-token rewards.

2. **Humans are bad at giving absolute scores:** People are much better at comparative judgments (e.g., "Response A is better than Response B"). **Solution:** Collect data as binary comparisons and train the reward model to predict which response a human would prefer.

The Classic RLHF Pipeline:

1. Start with a pre-trained, SFT model.
2. Generate multiple responses for a set of prompts.
3. Have humans rank these responses or pick the best one (creating preference pairs).
4. Use this preference data to train or update a reward model.
5. Use the trained reward model in an RL loop (such as PPO) to fine-tune the language model.
6. This is an iterative process. As the language model drifts, the reward model may become inaccurate (out-of-distribution), requiring a new round of data collection and RM training.

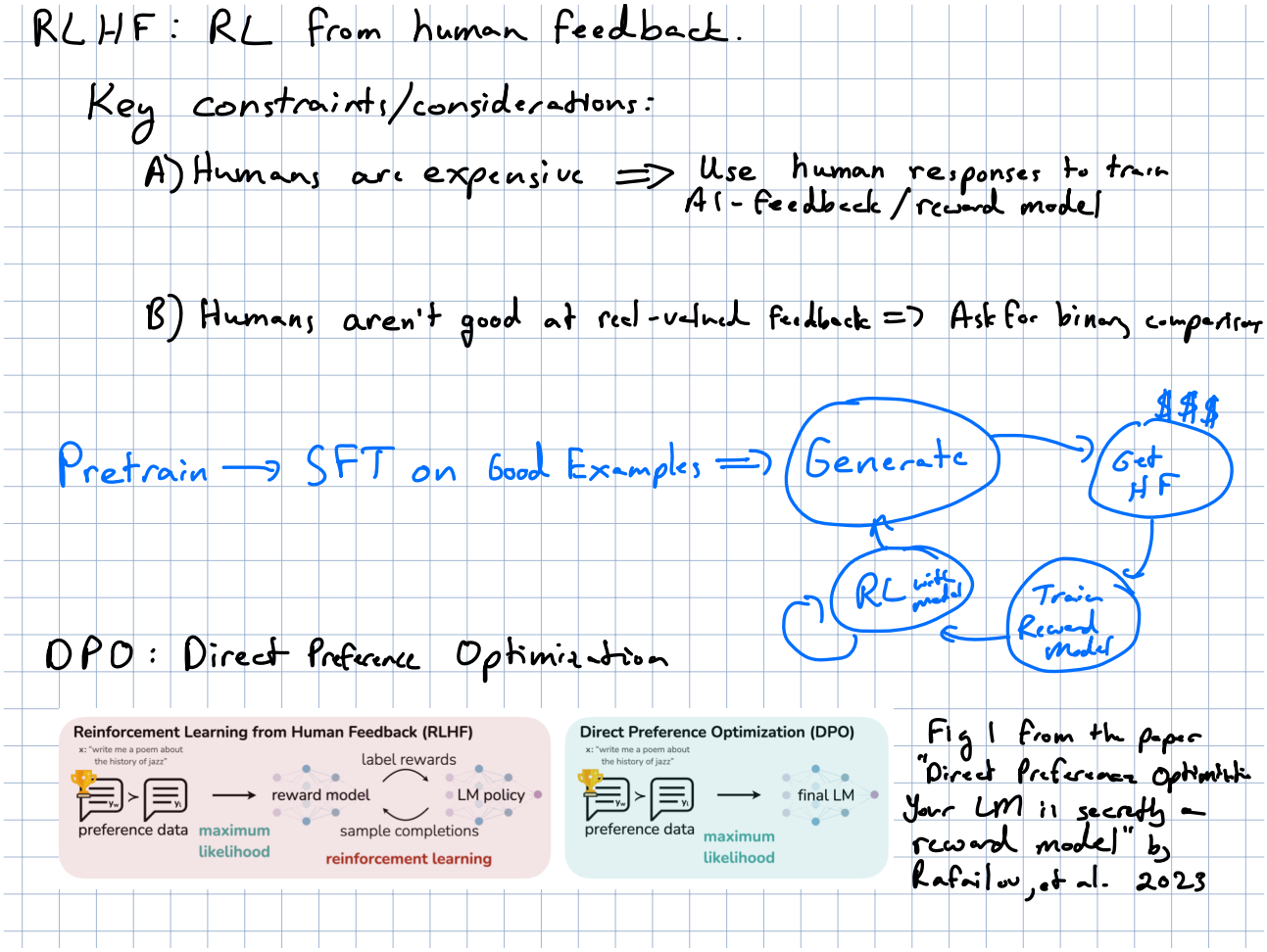


Figure 26.1: RLHF and DPO training workflows. The human-preference pipeline separates collecting comparisons, fitting a reward model, and policy optimization. The published DPO comparison highlights which stages direct preference training avoids; original figure credits in the notes are retained. Original annotated source excerpt; Anant Sahai, faculty handout 26, p. 5.

Direct Preference Optimization (DPO)

Deeper explanation: A numerical DPO example (p. 467)

DPO is a more recent technique that bypasses the need for an explicit reward model. It fine-tunes the language model directly on the preference data.

The Problem with Naive Approaches: A simple idea is to treat the preferred response (y_w, for "winner") as a positive example and the dispreferred response (y_l, for "loser") as a negative one. One might try to maximize $\log \pi_\theta(y_w)$ and minimize $\log \pi_\theta(y_l)$.

- **Result:** This often leads to a "total meltdown." The model's generations become incoherent. By pushing down the probability of the loser response, which is often grammatically correct and reasonable but just slightly worse, the model can learn to be un-grammatical or nonsensical to satisfy the objective.

The DPO Approach: DPO proposes a specific loss function derived from a model of human preferences.

1. **Idea 1 (Baseline):** Choose a fixed reference policy, commonly an SFT model. Winner-only SFT is one possible initialization discussed in the lecture, not a requirement of the DPO derivation.
2. **Idea 2 (DPO Loss):** Starting from $\pi_\theta = \pi_{\text{ref}}$, minimize the following loss function using SGD on preference pairs (y_w, y_l):

$$\mathcal{L}_{\text{DPO}}(\pi_\theta; \pi_{\text{ref}}) = -\mathbb{E}_{(x,y_w,y_l)\sim D}\left[\log \sigma\left(\beta \log \frac{\pi_\theta(y_w|x)}{\pi_{\text{ref}}(y_w|x)} - \beta \log \frac{\pi_\theta(y_l|x)}{\pi_{\text{ref}}(y_l|x)}\right)\right] \tag{26.4}$$

where σ is the sigmoid function and β is a temperature parameter.

Intuition for the DPO Loss:

- The core of the loss is the difference between two log-probability ratios. This can be interpreted as the difference in the implicit reward between the winner and loser responses. DPO implicitly defines a reward model:

$$\hat{r}_\theta(x, y) = \beta \log \frac{\pi_\theta(y|x)}{\pi_{\text{ref}}(y|x)}$$

  This means "what is good is what the current model likes to generate, relative to the reference model."

- **Why does π_{ref} matter?** Gradients do not flow into π_{ref}. So how does it affect training? The terms involving π_{ref} act as a constant offset inside the sigmoid function. This offset shifts the operating point of the sigmoid. If the current model π_θ is already much better than π_{ref} at preferring y_w over y_l, the argument to the sigmoid will be large and positive, putting it in the saturation region where gradients are small. This effectively tells the model "you've learned enough on this example." It focuses the training effort on pairs where the model's preference does not yet align with the human preference.

- **Gradient View:** The gradient of the DPO loss can be shown to be a weighted version of the naive approach:

$$\nabla_\theta \ell_{\text{DPO}} = -\beta\sigma(-z)\left[\nabla_\theta \log \pi_\theta(y_w \mid x) - \nabla_\theta \log \pi_\theta(y_l \mid x)\right]$$

  Here z is the beta-scaled difference of log probability ratios inside the loss. The factor $\sigma(-z)$ downweights pairs already assigned a large positive margin.

Idea 3 (TR-DPO): A discussed variant is to periodically update the reference policy π_{ref} to be the current policy π_θ every N training steps (e.g., N=512). This allows the baseline for "good enough" to adapt as the model improves.

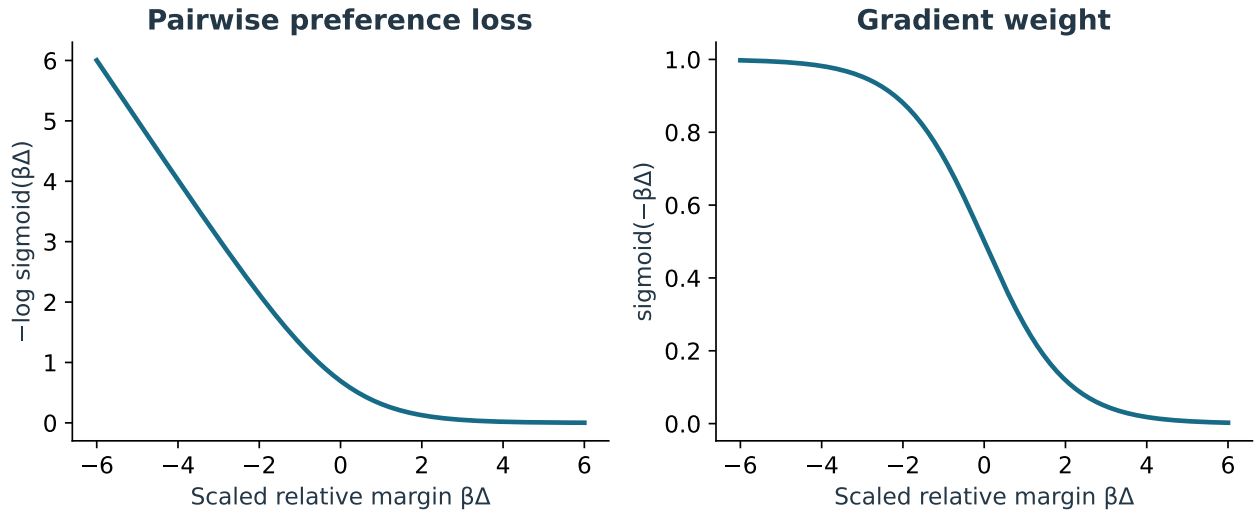


Figure 26.2: Preference learning operates on a difference of policy-to-reference log ratios. Confidently satisfied pairs contribute smaller gradients than reversed pairs.

Worked derivation: reward differences, DPO, and its gradient

Lecture basis: faculty handout 26, pp. 5–6. The steps below expand the lecture argument.

Step 1: eliminate the prompt-dependent normalizer. The previous optimum identity gives $r(x, y) = \beta \log[\pi^*(y \mid x)/\pi_{\text{ref}}(y \mid x)] + \beta \log Z(x)$. For a winner y_w and loser y_l, the same $Z(x)$ cancels:

$$r(x, y_w) - r(x, y_l) = \beta \left[\log \frac{\pi^*(y_w \mid x)}{\pi_{\text{ref}}(y_w \mid x)} - \log \frac{\pi^*(y_l \mid x)}{\pi_{\text{ref}}(y_l \mid x)} \right].$$

Step 2: use a pairwise preference model. Assume the Bradley–Terry law $P(y_w \succ y_l \mid x) = \text{sigmoid}(r_w - r_l)$. Parameterize the implicit policy by π_θ and define

$$z_\theta = \beta \left[\log \frac{\pi_\theta(y_w \mid x)}{\pi_{\text{ref}}(y_w \mid x)} - \log \frac{\pi_\theta(y_l \mid x)}{\pi_{\text{ref}}(y_l \mid x)} \right].$$

The observed-preference negative log likelihood is $L_{\text{DPO}} = -\log \text{sigmoid}(z_\theta)$.

Step 3: differentiate and identify the reference model's role. The scalar derivative is

$$\frac{d}{dz}[-\log \text{sigmoid}(z)] = -\text{sigmoid}(-z).$$

Therefore

$$\nabla_\theta L_{\text{DPO}} = -\beta \, \text{sigmoid}(-z_\theta)[\nabla_\theta \log \pi_\theta(y_w \mid x) - \nabla_\theta \log \pi_\theta(y_l \mid x)].$$

The frozen reference has no parameter derivative, but it changes z_θ and therefore the weight multiplying this gradient. An already well-separated preferred pair gets a smaller update. For language models each sequence log probability is a sum of response-token log probabilities; an arbitrary length average defines a different objective. The derivation depends on the KL-regularized policy identity and preference model, not merely on increasing the winner's likelihood in isolation.

26.3 Diffusion Models for Generation

Deeper explanation: A Gaussian-data denoiser can be solved exactly (p. 472)

We now switch back to generative modeling, introducing a new paradigm: diffusion models.

Motivation: Test-Time Compute

- Models like VAEs and GANs perform a single forward pass for generation: noise in, sample out. This is very fast.
- Autoregressive models spend more compute at test time, generating a sequence one token at a time. This iterative process allows for more complex generation. This works well for domains with inherent sequentiality (text, audio).
- **Question:** How can we spend more compute at test time for non-sequential data like images?

We recall an idea that *didn't* work: starting with random noise and performing gradient ascent on the output of a classifier (e.g., a "cat" classifier). This fails because the classifier is trained on the manifold of natural images, and random noise is far off this manifold. The gradients in this out-of-distribution space are not meaningful for generating realistic images.

The insight from VAEs was that the decoder must be trained on noisy inputs to learn to map a noisy latent space to clean outputs. Diffusion models take this idea to an extreme.

The Core Idea of Diffusion

Instead of just one level of noise, what if we train the model on a whole spectrum of noise?

1. We start with a **clean desired sample** (e.g., a real image).
2. At the other extreme, we have **total noise** (e.g., a sample from a standard Gaussian).
3. The bridge between these two is a continuum of samples with varying levels of noise. We can generate these by taking a clean sample and progressively adding more and more noise.

The model will be trained on data from this entire continuum. The generative process will then involve starting from total noise and iteratively *removing* noise to arrive at a clean sample.

The Forward and Reverse Processes

- **Forward Process (Diffusion):** This is a fixed process that we define. We start with a data sample $x_0 \sim q(x)$ and gradually add Gaussian noise over a series of T timesteps, producing a sequence of latent variables $x_1, x_2, \dots, x_T$.

  $$q(x_t|x_{t-1}) = \mathcal{N}(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t\mathbf{I})$$

  The β_t are small positive constants. After many steps, x_T becomes approximately an isotropic Gaussian $\mathcal{N}(0, \mathbf{I})$. For conceptual simplicity, one can think of this as simply adding noise: $x_t = x_{t-1} + \text{noise}$. The scaling factors $\sqrt{1-\beta_t}$ are a practical detail to keep the variance bounded.

- **Reverse Process (Generation):** The goal is to learn the reverse process, $p_\theta(x_{t-1}|x_t)$. If we can model this distribution, we can start with a sample $x_T \sim \mathcal{N}(0, \mathbf{I})$ and iteratively sample backwards: $x_{T-1}, x_{T-2}, \dots, x_0$ to generate a new data point.

The analogy is a movie of a drop of dye diffusing in water. The forward process is the movie playing forward (dye spreads out to equilibrium). The reverse process is learning to run the movie backward (randomly distributed dye particles coalesce back into a drop).

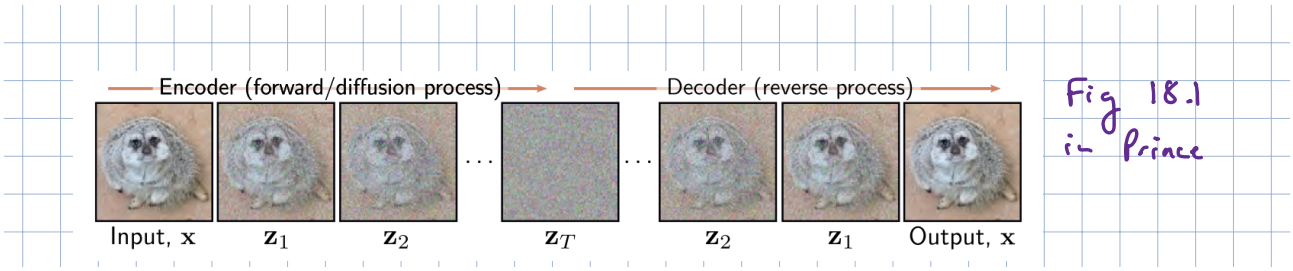


Figure 26.3: An image progressively noised and denoised. The image strip from Prince makes the forward and reverse directions visible. Forward corruption is prescribed; the reverse transitions must be learned. The reverse row is a schematic of generation, not a promise to recover the particular original image. Original annotated source excerpt; Anant Sahai, faculty handout 26, p. 8.

Key Design Question: Randomness in the Reverse Path

When we generate by moving backward, does each step need to be random?

1. **Stochastic Sampling (DDPM - Denoising Diffusion Probabilistic Models):** Yes, the backward path is stochastic. This is analogous to simulating the reverse of a physical process.

2. **Deterministic Sampling (DDIM - Denoising Diffusion Implicit Models):** No, the backward path can be deterministic. We will discuss this in the next lecture.

Stochastic Case (DDPM)

To build intuition, consider a simple discrete-state Markov chain with 3 states (A, B, C) that converges to a uniform stationary distribution $(1/3, 1/3, 1/3)$.

- Suppose our "data distribution" is non-stationary, e.g., $p_0(A) = 1/2, p_0(B) = 0, p_0(C) = 1/2$.
- **Forward process:** We run the Markov chain forward. Over time, any initial distribution will converge to the stationary uniform distribution.
- **Reverse process:** If we watch a "movie" of this process in reverse, the transition probabilities are different. For example, after one step, if we find ourselves in state B, we know we must have come from A or C, not B. The reverse process is time-varying; it has a "drift" that pulls the distribution back towards the initial, non-stationary one.

Applying this to the continuous case of images, a key result is that if the forward noising step is a Gaussian, the reverse step is *also* approximately Gaussian, as long as the noise added (β_t) is small.

$$p(x_{t-1}|x_t) \approx \mathcal{N}(x_{t-1}; \mu_\theta(x_t, t), \Sigma_\theta(x_t, t))$$

Usually, the variance Σ_θ is fixed, so the learning task is to train a neural network $\mu_\theta(x_t, t)$ to predict the mean of the distribution for the previous, less noisy state, given the current noisy state x_t and the timestep t. This network is trained to "denoise" the input by one step. The generation process then involves calling this network repeatedly to move from pure noise back to a clean image.

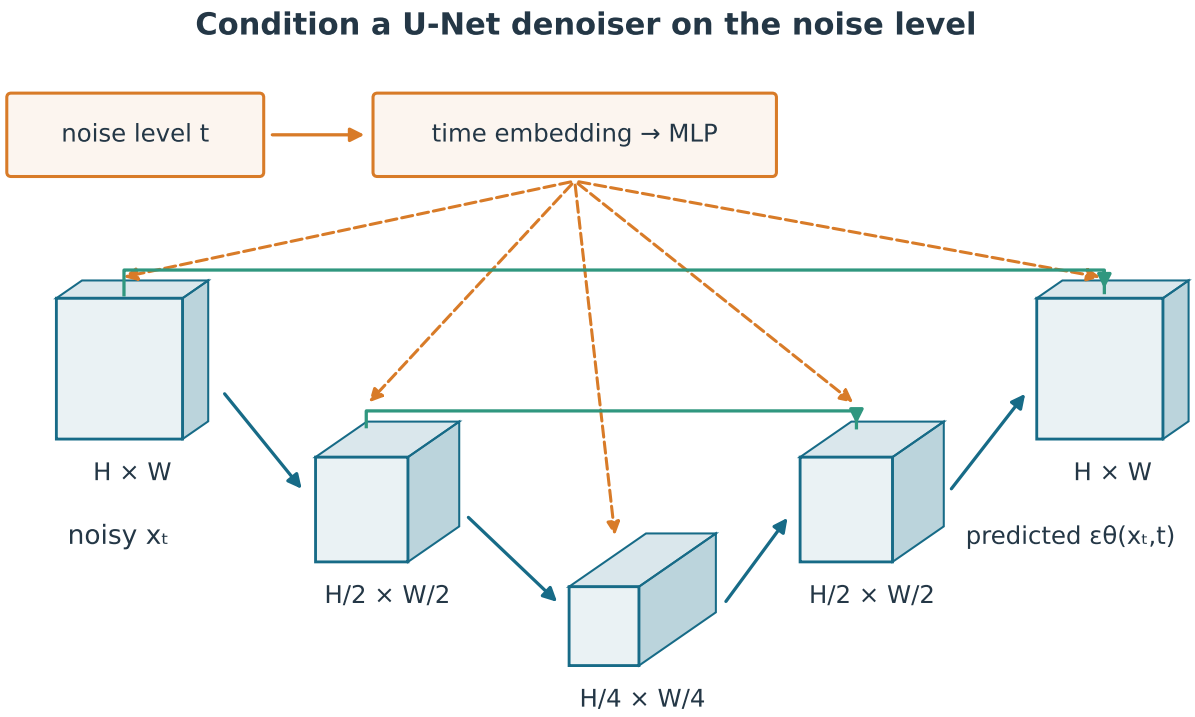


Figure 26.4: A U-Net can predict a spatial noise field while receiving timestep information at multiple resolutions. Dashed orange arrows are conditioning paths; green arrows carry encoder features. Shown: three resolutions and an additive time-conditioning example. Precise widths and block counts are model choices; see the earlier complete U-Net for a full shape walkthrough.

Lecture 27

Diffusion Models

Course sources: lecture_27; 2025-12-04 transcript.

27.1 Announcements and Logistics (Fall 2025)

These announcements describe the original Fall 2025 lecture context. This is the final lecture of the course. Key reminders and deadlines are as follows:

- **Final Project Report:** The deadline has been extended to Sunday, December 7th.
- **Poster Session:** Tuesday, December 9th.
  - Students are responsible for printing their own posters. It is perfectly acceptable to print slides on regular paper and arrange them together. The focus is on clear communication of your work, not on professional printing quality.
  - It is recommended to bring a printed copy of your report and a laptop to show additional plots, code, or simulations during discussions.
- **Peer Reviews:** Assigned on Monday, December 8th, and due on Thursday, December 11th. Your individual grade for conducting peer reviews will be a multiplicative factor on your group's project grade.
- **Final Report Submission:** After incorporating feedback from peer reviews and the poster session, the final version of the report is due Sunday, December 14th. The final grade is based on the quality of this final report, the poster session presentation, and the quality of your peer reviews.
- **Final Exam:** Wednesday, December 17th, at 8:00 AM.
  - The exam is closed-book: no notes, no questions, and no clarifications will be permitted during the exam.
  - You will only need a pencil and an eraser.
- **Recommended EECS Colloquium:**
  - Wednesday, December 10th, 4-5 PM by Ece Kamar from Microsoft Research, who leads the group releasing models like Phi.
  - The recent colloquium by Bill Dally from NVIDIA is also highly recommended for understanding the interplay between hardware (GPUs) and deep learning architectures.

27.2 Introduction to Diffusion Models

Deeper explanation: A Gaussian-data denoiser can be solved exactly (p. 472)

This lecture covers diffusion models for generative tasks. The primary readings are Chapter 18 of the Prince textbook and the paper arXiv:2406.08929, "Step-by-step Diffusion: An elementary tutorial," which provides a clear and modern perspective.

Recap: The Challenge of Generation

The goal of generative modeling is to sample from the distribution of "natural" data, such as images. This distribution forms a very thin, low-dimensional manifold within the vast space of all possible pixel combinations.

- A naive approach, like starting with random noise and performing gradient ascent on a classifier's score (e.g., "cat score"), fails. It produces noise-like images that maximally activate the classifier but do not resemble natural images.
- The insight from Variational Autoencoders (VAEs) is that the model needs to see noisy versions of the data during training to learn how to navigate the high-dimensional space and move towards the data manifold.

The Core Idea of Diffusion

Diffusion models embrace this idea by systematically corrupting data with noise and then learning to reverse the process.

- **Analogy:** Imagine a drop of red dye (the manifold of natural images) placed in a glass of water (the space of all possible images). Over time, the dye diffuses until it is uniformly spread throughout the water.
- **Forward Process (Diffusion):** This corresponds to gradually adding noise to an image, step-by-step, until it becomes pure, unstructured noise (like the fully diffused dye). This process maps a complex data distribution to a simple, known distribution (e.g., a standard Gaussian).
- **Reverse Process (Generation):** This corresponds to running the diffusion movie in reverse. We start with a sample from the simple noise distribution and learn a process to gradually denoise it, step-by-step, until it becomes a sample from the original data distribution.

The key design question is whether the reverse path needs to be stochastic or if it can be deterministic. We will explore both approaches.

1. **Stochastic Sampling:** Denoising Diffusion Probabilistic Models (DDPM).

2. **Deterministic Sampling:** Denoising Diffusion Implicit Models (DDIM).

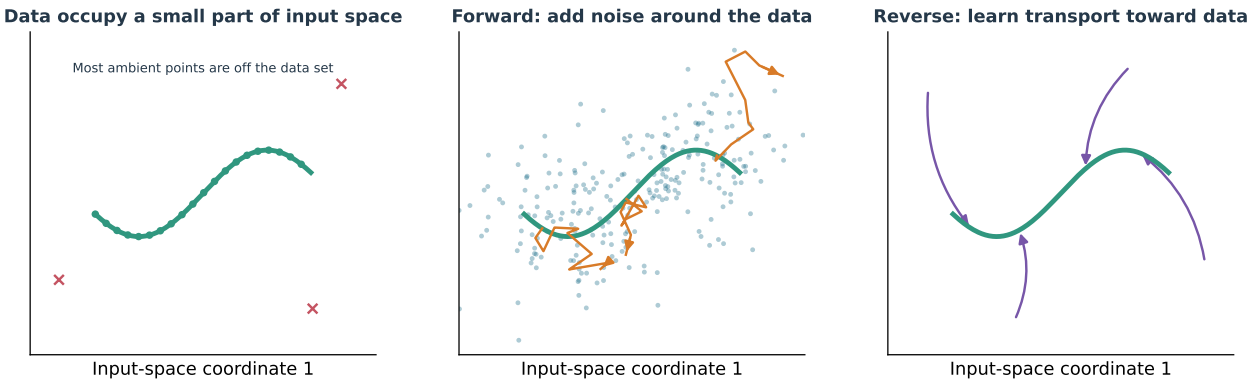


Figure 27.1: Data geometry, noise, and learned reverse transport. An explanatory redraw of the lecture's manifold sketches. The curve and trajectories are schematic. Learning to return noisy points toward data differs from replaying the exact random increments of one forward path. Explanatory redraw; illustrative geometry; Anant Sahai, faculty handout 27, p. 1, 2.

27.3 Stochastic Case: Denoising Diffusion Probabilistic Models (DDPM)

The first approach is to simulate the exact reverse of the forward stochastic process.

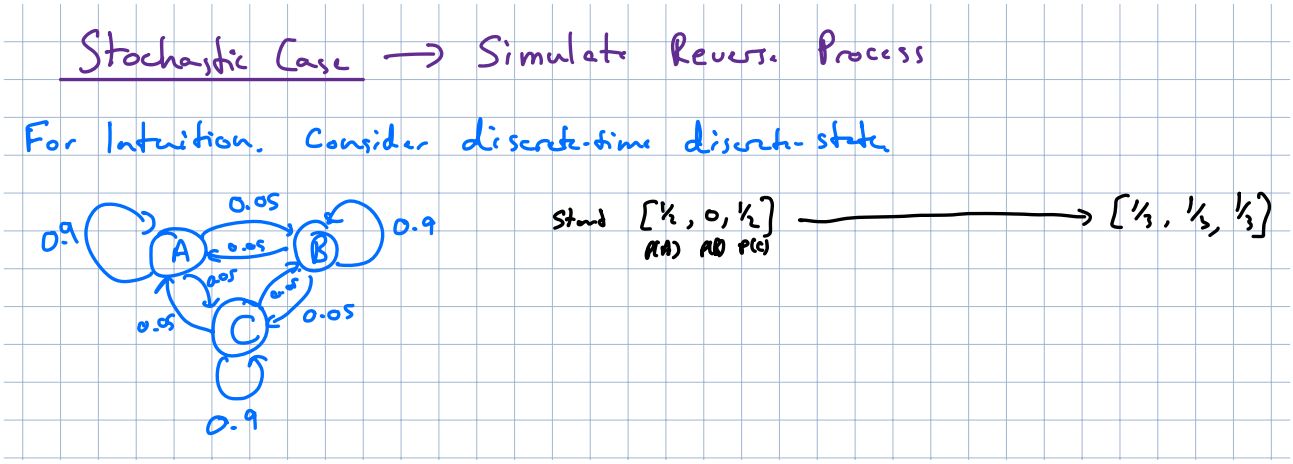


Figure 27.2: A finite-state analogy for forward and reverse diffusion. Forward transition probabilities and reverse conditional probabilities answer different questions. Reverse probabilities also depend on the current marginal distribution, so reversing arrows does not simply reuse their labels. Original annotated source excerpt; Anant Sahai, faculty handout 27, p. 3.

Continuous-Time Perspective

For clarity, we adopt a continuous-time perspective where the process evolves from clean data at $t = 0$ to a chosen terminal noise level at $t = 1$. The discrete steps are of size Δt.

- Let X_0 be a sample from the data distribution p_0.
- The forward process adds Gaussian noise at each step. In a small time step Δt, the added noise is $\mathcal{N}(0, \sigma^2 \Delta t I)$. The variance scales with Δt to ensure a well-defined limit. The added-noise variance after time t is $t\sigma^2$; the marginal also includes the original data variance. Additive Brownian corruption at finite time does not make arbitrary data exactly standard Gaussian.
- We define a sequence of noisy images $X_0 \to X_{\Delta t} \to X_{2\Delta t} \to ... \to X_1$.

The Reverse Process

Our goal is to learn the conditional distribution $p(X_{t-\Delta t}|X_t)$.

Key Claim: As long as the data density p_0 is sufficiently smooth and the noise step $\sigma^2 \Delta t$ is small enough, the reverse conditional distribution is also approximately Gaussian:

$$p(X_{t-\Delta t}|X_t = z) \approx \mathcal{N}(\mu_t(z), \sigma^2 \Delta t I) \tag{27.1}$$

To leading order in a small step, the reverse covariance equals the forward covariance; finite-step conditionals need not share it exactly. The mean $\mu_t(z)$ is a function of the current state z and time t, and this is what we must learn from data using a neural network.

Argument for the Gaussian Reverse Process

Deeper explanation: Complete the square for the variance-preserving posterior (p. 473)

Deeper explanation: The Brownian bridge gives an exact conditional mean (p. 470)

We can justify this claim with an argument based on Bayes' rule and Taylor expansions.

1. **Bayes' Rule:** Start by expressing the reverse conditional in terms of the forward conditional:

$$p(x_{t-\Delta t}|x_t) = \frac{p(x_t|x_{t-\Delta t})p_{t-\Delta t}(x_{t-\Delta t})}{p_t(x_t)} \propto p(x_t|x_{t-\Delta t})p_{t-\Delta t}(x_{t-\Delta t})$$

   The denominator $p_t(x_t)$ is a normalization constant that does not depend on $x_{t-\Delta t}$.

2. **Log-Probabilities:** Taking the logarithm simplifies the product into a sum:

$$\log p(x_{t-\Delta t}|x_t) = \log p(x_t|x_{t-\Delta t}) + \log p_{t-\Delta t}(x_{t-\Delta t}) + \text{const}$$

3. **Substitute Known Terms:** The forward process is Gaussian, so $\log p(x_t|x_{t-\Delta t}) = -\frac{1}{2\sigma^2\Delta t}\|x_t - x_{t-\Delta t}\|_2^2 + \text{const}$. For sufficiently small Δt, $p_{t-\Delta t} \approx p_t$, so we can approximate $\log p_{t-\Delta t}(x_{t-\Delta t}) \approx \log p_t(x_{t-\Delta t})$.

$$\log p(x_{t-\Delta t}|x_t) \approx -\frac{1}{2\sigma^2\Delta t}\|x_t - x_{t-\Delta t}\|_2^2 + \log p_t(x_{t-\Delta t}) + \text{const}$$

4. **Taylor Expand around x_t:** We expand $\log p_t(x_{t-\Delta t})$ around the point x_t:

$$\log p_t(x_{t-\Delta t}) \approx \log p_t(x_t) + \langle \nabla_{x_t} \log p_t(x_t), x_{t-\Delta t} - x_t \rangle + O(\|x_{t-\Delta t} - x_t\|^2)$$

The term $\log p_t(x_t)$ is constant with respect to $x_{t-\Delta t}$ and can be absorbed into the normalization constant. The gradient term $\nabla_{x_t} \log p_t(x_t)$ is known as the **score**.

5. **Complete the Square:** Combining terms, we have an expression that is quadratic in $x_{t-\Delta t}$:

$$\log p(x_{t-\Delta t}|x_t) \approx -\frac{1}{2\sigma^2\Delta t}\|x_t - x_{t-\Delta t}\|_2^2 - \langle \nabla_{x_t} \log p_t(x_t), x_t - x_{t-\Delta t} \rangle + \text{const}$$

This is the classic setup for completing the square. By rearranging terms, we can see this is the log-density of a Gaussian.

$$\begin{aligned}\log p(x_{t-\Delta t} \mid x_t) &\approx -\frac{1}{2\sigma^2\Delta t}\|u\|^2 + s^\top u + C_1(x_t) \\ &= -\frac{1}{2\sigma^2\Delta t}\|u - \sigma^2\Delta t\, s\|^2 + C_2(x_t),\end{aligned}$$

where $u = x_{t-\Delta t} - x_t$ and $s = \nabla \log p_t(x_t)$. The constants absorb terms independent of the earlier state. This reveals the form of a Gaussian with:

- **Mean:** $\mu_t(x_t) = x_t + \sigma^2\Delta t \nabla_{x_t} \log p_t(x_t)$
- **Variance:** $\sigma^2\Delta t I$

Worked derivation: completing the reverse-step square carefully

Lecture basis: faculty handout 27, pp. 3–4. The steps below expand the lecture argument.

Step 1: use an increment variable and retain the approximation sign. Let $h > 0$, $v = \sigma^2 h$, $z = X_t$, and $u = X_{t-h} - z$. Brownian corruption gives $p(z \mid z+u) \propto \exp(-\|u\|^2/(2v))$. Bayes' rule therefore gives exactly

$$\log p(u \mid z) = -\frac{\|u\|^2}{2v} + \log p_{t-h}(z+u) + C(z).$$

For a smooth positive density in the local region, approximate the earlier log density by its first-order expansion. Writing $s = \nabla \log p_t(z)$, this yields

$$\log p(u \mid z) \approx -\frac{\|u\|^2}{2v} + s^\top u + C'(z).$$

Replacing the earlier-time score by the current one and dropping curvature are small-step approximations, not exact equalities for general data.

Step 2: complete the square with its constant term shown.

$$\begin{aligned} -\frac{\|u\|^2}{2v} + s^\top u &= -\frac{1}{2v}(\|u\|^2 - 2vs^\top u) \\ &= -\frac{1}{2v}(\|u - vs\|^2 - v^2\|s\|^2) \\ &= -\frac{\|u - vs\|^2}{2v} + \frac{v}{2}\|s\|^2. \end{aligned}$$

The final term depends on z but not u, so it is absorbed by normalization. Reading the Gaussian density gives

$$X_{t-h} \mid X_t = z \;\approx\; \mathcal{N}(z + \sigma^2 h \nabla \log p_t(z),\, \sigma^2 h I).$$

Step 3: see where the covariance correction enters. If the Hessian term is retained, the local precision is $v^{-1}I - \nabla^2 \log p_t(z)$. When invertible and positive definite its inverse is

$$(v^{-1}I - H)^{-1} = v(I - vH)^{-1} = vI + v^2 H + O(v^3).$$

Thus equal forward and reverse covariance is a leading-order statement. A finite-step reverse conditional can be non-Gaussian or multimodal. Gaussian source data permit the exact block-conditioning calculation alongside this local argument.

Joint Gaussian blocks explain the reverse conditional

As an exactly solvable illustration, let $x_0 \sim \mathcal{N}(0, \Sigma_0)$ and $x_t = x_0 + W_t$, where W_t is independent Brownian noise with covariance tI. Set $S = \Sigma_0 + (t-h)I$ and $T = \Sigma_0 + tI$ for $0 < h \le t$. Then

$$\begin{bmatrix} x_{t-h} \\ x_t \end{bmatrix} \sim \mathcal{N}\left(0, \begin{bmatrix} S & S \\ S & T \end{bmatrix}\right).$$

The off-diagonal block is S because the earlier state is shared and the increment $W_t - W_{t-h}$ is independent. Gaussian conditioning yields

$$\mathbb{E}[x_{t-h} \mid x_t] = ST^{-1}x_t, \qquad \mathrm{Cov}(x_{t-h} \mid x_t) = S - ST^{-1}S.$$

For $\Sigma_0 = I$, $t = 3$, and $h = 1$, these become $3x_t/4$ and $3I/4$. The covariance expression is the Schur complement of T in the joint block matrix. Marginals S, T alone would not determine the conditional: the cross-covariance block is essential. For general non-Gaussian data, this Gaussian example is not an exact finite-step reverse model; local reverse approximations and the conditional-on-clean-data training target require their own assumptions.

Training and Generation

Deeper explanation: Sample averages and differentiation under expectations (p. 444)

Deeper explanation: Binary diffusion makes the same ideas visible (p. 469)

Deeper explanation: Binary reverse targets average to the desired marginals (p. 474)

Deeper explanation: Late binary reverse targets are not generally one-half (p. 474)

Generation (DDPM Sampling): With a trained model $f_\theta(x_t, t)$ that approximates the mean $\mu_t(x_t)$, we can generate samples.

1. Start with pure noise: $\hat{x}_1 \sim \mathcal{N}(0, \sigma^2 I)$.
2. Iterate backwards from $t = 1$ down to $t = \Delta t$: For each step, compute the next state by taking a sample from the reverse conditional Gaussian:

$$\hat{x}_{t-\Delta t} = f_\theta(\hat{x}_t, t) + \eta, \quad \text{where } \eta \sim \mathcal{N}(0, \sigma^2 \Delta t I)$$

3. The final output is $\hat{x}_0$.

Training: We train the network f_θ using stochastic gradient descent.

1. Sample a clean image x_0 from the training set.
2. Sample a random time t uniformly from $[0, 1]$.
3. Construct a noisy version x_t by adding Gaussian noise with variance $t\sigma^2$ to x_0.
4. The network $f_\theta(x_t, t)$ is trained to predict the mean of the reverse step. The ideal target is $\mathbb{E}[X_{t-\Delta t} | X_t = x_t]$. We can use $x_{t-\Delta t}$ (generated from x_0) as a target.
5. The loss function is the squared error:

$$L_\theta = \| f_\theta(x_t, t) - x_{t-\Delta t} \|_2^2$$

An Improved Training Objective

Deeper explanation: Training targets and reverse samplers are distinct objects (p. 474)

Instead of predicting $x_{t-\Delta t}$, we can re-parameterize the prediction target. Note that the mean $\mu_t(x_t)$ can be rewritten as $\mu_t(x_t) = x_t + (\mu_t(x_t) - x_t)$, where $\mu_t(x_t) - x_t$ is the expected negative of the noise added in the last step.

$$\mu_t(x_t) - x_t \text{ is an estimate of } \mathbb{E}[x_{t-\Delta t} - x_t | x_t] \tag{27.2}$$

The key insight comes from the symmetry of the i.i.d. noise added over time. Given only the final noisy image x_t, there is no way to distinguish the noise added in the last step from noise added in any other step. Therefore, the expectation of the last noise increment is simply the average of all noise increments.

$$\mathbb{E}[x_{t-\Delta t} - x_t | x_t] = \frac{\Delta t}{t} \mathbb{E}[x_0 - x_t | x_t] \tag{27.3}$$

This implies that learning to predict the last noise step is equivalent to learning to predict the total noise added $(x_t - x_0)$, which is in turn equivalent to predicting the original clean image x_0.

Benefits of Predicting x_0:

- **Inductive Bias:** Neural network architectures like U-Nets and ConvNets are designed with strong inductive biases for processing and generating natural images, not for generating unstructured noise. Whether clean-image prediction is more effective depends on the parameterization, architecture, and weighting across noise levels.
- **Variance Reduction:** Replacing a sampled previous-state target by its conditional expectation given (x_t, x_0) removes avoidable target noise. This Rao–Blackwellization motivates a lower-variance target; it does not establish that every x_0 parameterization has lower gradient variance than every noise parameterization.

Thus, one useful training objective is to have the network $f_\theta(x_t, t)$ predict an estimate of x_0, and use the loss $\|f_\theta(x_t, t) - x_0\|_2^2$.

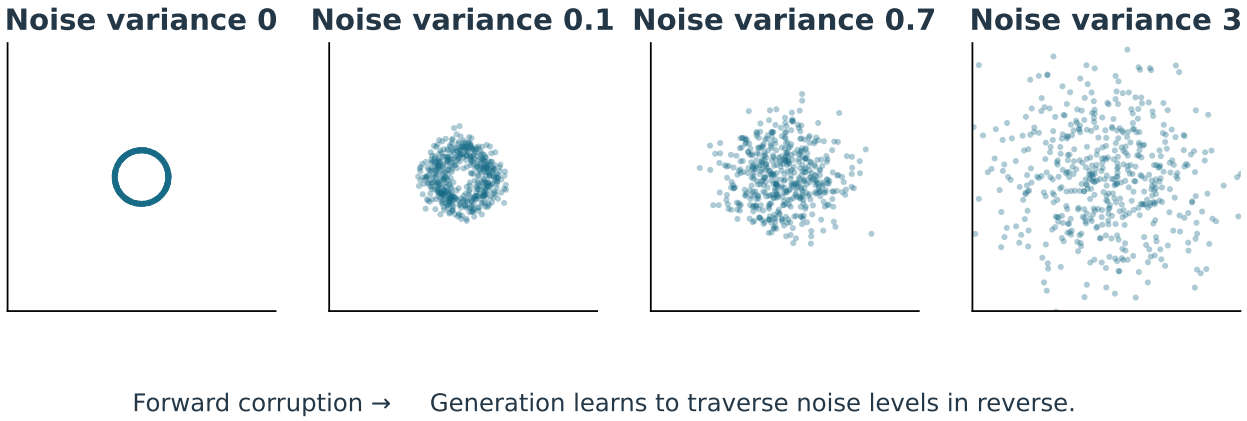


Figure 27.3: Known forward noise produces training pairs at any time. The learned reverse dynamics use a noise-level-conditioned denoiser or score.

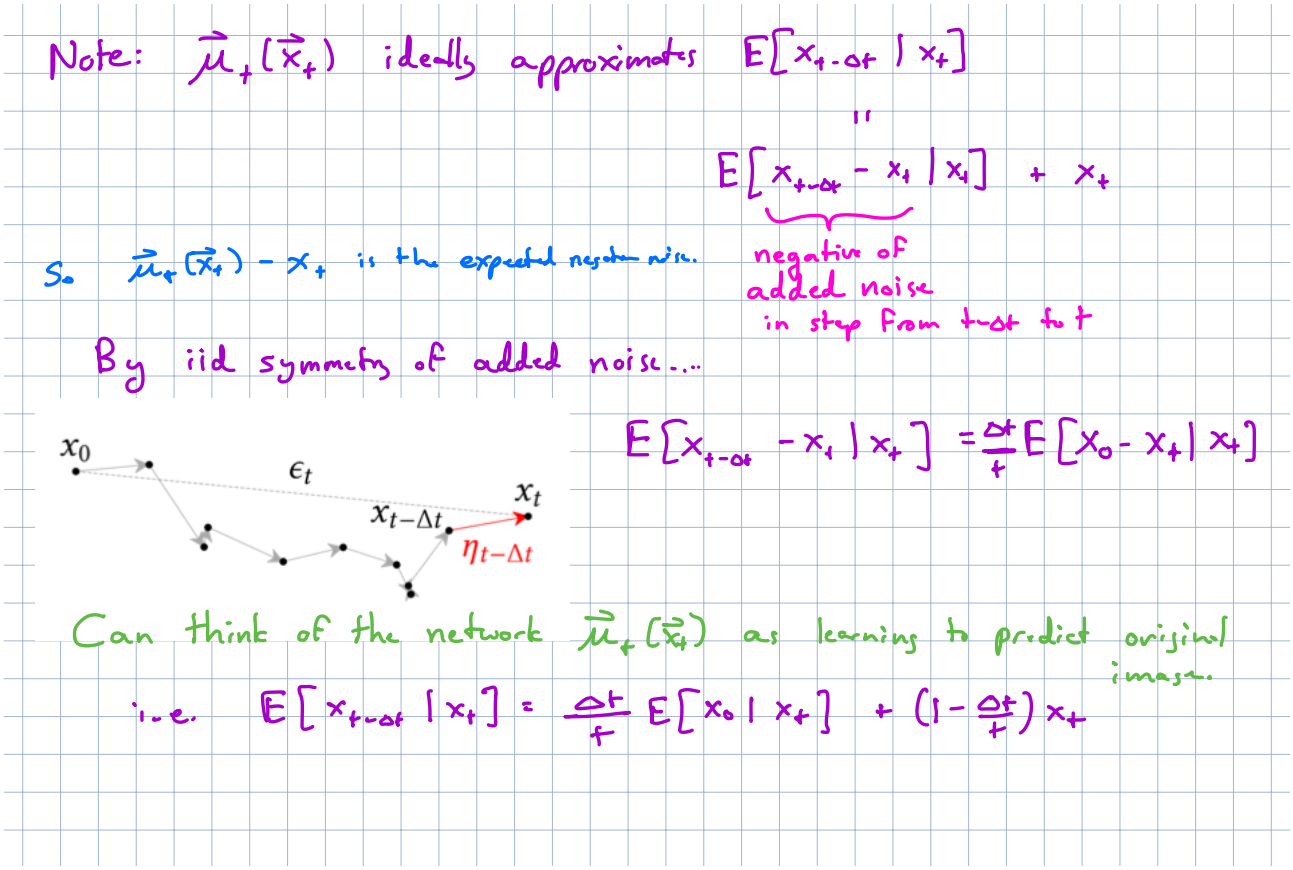


Figure 27.4: The noisy-input prediction training scaffold. A clean sample and a chosen noise level generate the supervised pair. Squared loss targets a conditional mean; this estimator is not itself the entire reverse sampling distribution. Original annotated source excerpt; Anant Sahai, faculty handout 27, p. 5.

Worked derivation: Brownian bridge targets and the denoiser–score identity

Lecture basis: faculty handout 27, pp. 5. The steps below expand the lecture argument.

Step 1: condition first on the clean sample. For $X_t = X_0 + \sigma W_t$ and $0 < h \le t$, the pair of Brownian noises at $t-h$ and t has covariance blocks $(t-h)I, (t-h)I, tI$. Gaussian conditioning gives

$$\begin{aligned}\mathbb{E}[X_{t-h} \mid X_t = z, X_0 = x_0] &= x_0 + \frac{t-h}{t}(z - x_0) \\ &= \left(1 - \frac{h}{t}\right) z + \frac{h}{t} x_0.\end{aligned}$$

Its conditional covariance is $\sigma^2 h(t-h)I/t$. Averaging the conditional mean over $X_0 \mid X_t = z$ gives, by the tower property,

$$m_{t,h}(z) := \mathbb{E}[X_{t-h} \mid X_t = z] = z + \frac{h}{t}[D_t(z) - z], \quad D_t(z) = \mathbb{E}[X_0 \mid X_t = z].$$

This mean identity is exact for additive Brownian corruption; it does not make the full reverse conditional Gaussian.

Step 2: derive why squared-error denoising learns D_t. For any prediction a at fixed z, write $X_0 - a = (X_0 - D_t) + (D_t - a)$. Squaring and averaging conditionally,

$$\mathbb{E}[\|X_0 - a\|^2 \mid z] = \mathbb{E}[\|X_0 - D_t\|^2 \mid z] + \|D_t - a\|^2,$$

because the cross term contains $\mathbb{E}[X_0 - D_t \mid z] = 0$. The unique conditional minimizer is $a = D_t(z)$. Replacing a sampled earlier-state target by its mean conditional on (X_0, X_t) preserves the regression optimum and removes conditional target noise.

Step 3: differentiate the corrupted-data density. With Gaussian noise variance $\sigma^2 t$ and sufficient integrability to exchange derivative and integral,

$$\begin{aligned}\nabla_z p_t(z) &= \int p_0(x)\, \nabla_z \mathcal{N}(z; x, \sigma^2 t I)\, dx \\ &= \int p_0(x) \mathcal{N}(z; x, \sigma^2 t I) \frac{x - z}{\sigma^2 t}\, dx, \\ \nabla_z \log p_t(z) &= \frac{\mathbb{E}[X_0 \mid X_t = z] - z}{\sigma^2 t}.\end{aligned}$$

Consequently $m_{t,h}(z) = z + \sigma^2 h \nabla \log p_t(z)$. Clean prediction, noise prediction, and reverse-mean prediction can be converted into each other, but their training weights and sampling formulas must use the matching scale factors.

27.4 Deterministic Case: Denoising Diffusion Implicit Models (DDIM)

Deeper explanation: Deriving the half drift from density evolution (p. 472)

The DDPM sampling process walks backward through the same joint distribution as the forward process. However, to generate a sample from p_0, we only need the marginal distributions p_t to be correct at each step. This opens the possibility of finding a different, deterministic path that connects the same sequence of marginals.

Intuition: A Simple Gaussian Example

Consider two Gaussian marginal distributions, $X \sim \mathcal{N}(0, 1)$ and $Y \sim \mathcal{N}(0, 2)$.

- **Stochastic Connection (DDPM-like):** Define $Y = X + \epsilon$, where $\epsilon \sim \mathcal{N}(0, 1)$ is independent of X. The reverse process is stochastic: given Y, the distribution of X is $\mathcal{N}(Y/2, 1/2)$. Sampling X from Y requires adding new noise.

- **Deterministic Connection (DDIM-like):** Define $Y = \sqrt{2}X$. This preserves the marginals. The reverse process is purely deterministic: $X = Y/\sqrt{2}$. No additional randomness is needed.

This example shows that it is possible to connect the same marginals deterministically.

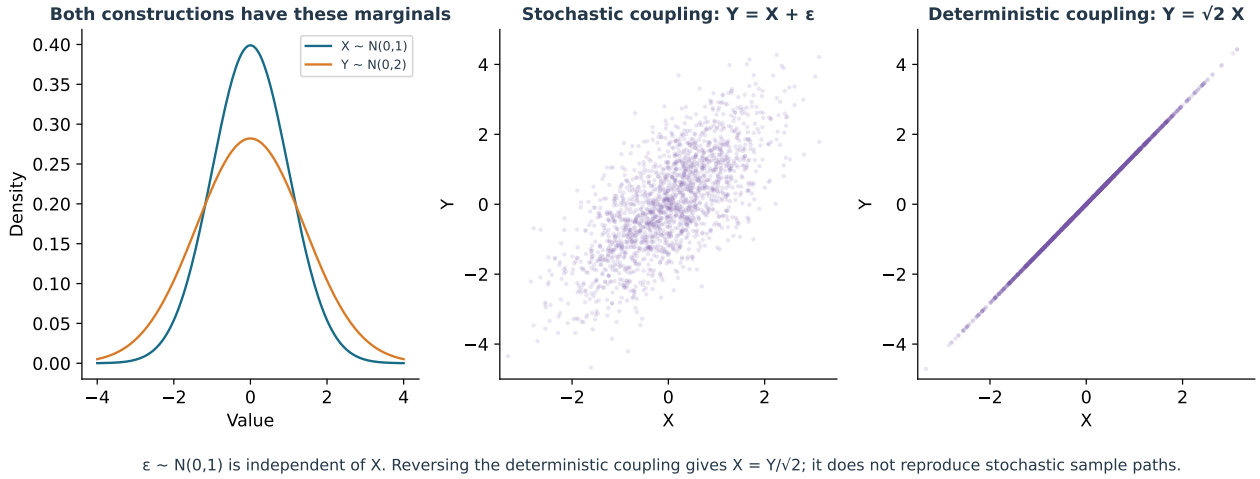


Figure 27.5: Same marginals, different couplings. The lecture's Gaussian example made visual. Both constructions have the same X and Y marginals, but different joint distributions: one has conditional randomness and the other is deterministic. This distinction explains why deterministic generation need not recover stochastic sample paths. Explanatory redraw; illustrative geometry; Anant Sahai, faculty handout 27, p. 6.

Guess-and-Verify Strategy for the Deterministic Step

We want to find a deterministic update rule. Let's guess a form for the reverse step. We know that $(\mu_t(x_t) - x_t)$ is the predicted direction towards a less noisy state. A natural guess is to move in this direction:

$$\hat{x}_{t-\Delta t} = \hat{x}_t + \lambda(\mu_\theta(\hat{x}_t, t) - \hat{x}_t) \tag{27.4}$$

The problem is to find the correct scaling factor λ. We use the strategy of solving for λ in the simplest possible case and hoping the solution generalizes.

The Easiest Example: Assume the data distribution is a single point at the origin: $p_0 = \delta_0$.

- In this case, we know all marginals exactly: $x_t \sim \mathcal{N}(0, t\sigma^2)$.
- The true conditional mean is also known: $\mathbb{E}[X_{t-\Delta t}|X_t = z] = \frac{t-\Delta t}{t} z$.

- Substitute this into our guessed update rule:

$$\begin{aligned}\hat{x}_{t-\Delta t} &= \hat{x}_t + \lambda\left(\frac{t-\Delta t}{t}\hat{x}_t - \hat{x}_t\right)\\ &= \hat{x}_t\left(1-\lambda\frac{\Delta t}{t}\right)\end{aligned}$$

Matching Variances: For the marginals to be correct, the variance of our generated $\hat{x}_{t-\Delta t}$ must match the true variance, which is $(t-\Delta t)\sigma^2$.

$$\begin{aligned}\text{Var}(\hat{x}_{t-\Delta t}) &= \text{Var}\left(\hat{x}_t\left(1-\lambda\frac{\Delta t}{t}\right)\right)\\ (t-\Delta t)\sigma^2 &= \left(1-\lambda\frac{\Delta t}{t}\right)^2\text{Var}(\hat{x}_t)\\ (t-\Delta t)\sigma^2 &= \left(1-\lambda\frac{\Delta t}{t}\right)^2(t\sigma^2)\end{aligned}$$

Solving this equation for λ yields:

$$\lambda = \frac{t}{\Delta t}\left(1-\sqrt{\frac{t-\Delta t}{t}}\right) \tag{27.5}$$

For small $\Delta t/t$, $\sqrt{1-\frac{\Delta t}{t}} \approx 1-\frac{1}{2}\frac{\Delta t}{t}$, so $\lambda \approx \frac{t}{\Delta t}(\frac{1}{2}\frac{\Delta t}{t}) = 1/2$. This means for most of the reverse process, we step about half the predicted distance.

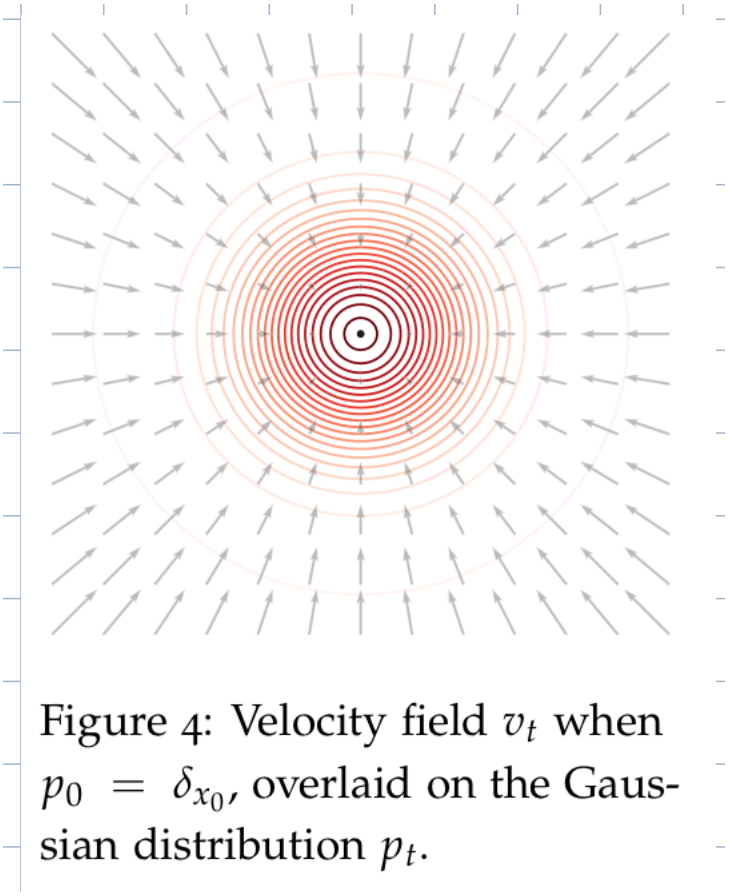


Figure 27.6: A probability-flow vector field. In this point-mass example, forward noise produces Gaussian marginals. The radial velocity field describes deterministic transport between these distributions. Matching marginals does not require retracing individual stochastic paths. The published illustration and original caption are retained from the handout. Original annotated source excerpt; Anant Sahai, faculty handout 27, p. 8.

Worked derivation: the half drift and the finite-step coefficient

Lecture basis: faculty handout 27, pp. 6–8. The steps below expand the lecture argument.

Step 1: rationalize the coefficient from the point-mass example. For $X_0 = 0$, $\mathrm{Var}(X_t) = \sigma^2 t$ and $m_{t,h}(z) = (1 - h/t)z$. The update $z' = z + \lambda(m_{t,h}(z) - z)$ has multiplier $1 - \lambda h/t$. Matching the earlier variance while retaining the positive branch requires

$$\begin{aligned}(1 - \lambda h/t)^2 t &= t - h,\\ 1 - \lambda h/t &= \sqrt{1 - h/t},\\ \lambda &= \frac{t}{h}(1 - \sqrt{1 - h/t})\\ &= \frac{1}{1 + \sqrt{1 - h/t}}.\end{aligned}$$

The last line multiplies numerator and denominator by $1 + \sqrt{1 - h/t}$. Thus $\lambda \to 1/2$ as $h/t \to 0$, but $\lambda = 1$ at the final step $h = t$. The small parameter is h/t, not merely whether t is numerically large.

Step 2: convert to a clean-data predictor without losing a factor. Since $m_{t,h}(z) - z = (h/t)(D_t(z) - z)$,

$$z' = z + \underbrace{\left(1 - \sqrt{1 - h/t}\right)}_{\text{coefficient for a clean predictor}} (D_t(z) - z).$$

The coefficient multiplying $D_t - z$ is $\lambda h/t$, not λ. For general non-Gaussian data this finite step is a denoising-based approximation, not an exact finite-step marginal transport theorem.

Step 3: derive the general continuous-time half drift. Brownian corruption has heat equation $\partial_t p_t = (\sigma^2/2)\Delta p_t$. A deterministic flow $\dot{x} = v_t(x)$ transports its density by $\partial_t p_t = -\nabla \cdot (p_t v_t)$. Choose

$$v_t(x) = -\frac{\sigma^2}{2} \nabla \log p_t(x).$$

Then, using $p_t \nabla \log p_t = \nabla p_t$,

$$-\nabla \cdot (p_t v_t) = \frac{\sigma^2}{2} \nabla \cdot (\nabla p_t) = \frac{\sigma^2}{2} \Delta p_t.$$

The two density evolutions agree under suitable regularity and the same initial marginal. Integrating this forward-time ODE backward by h gives

$$x_{t-h} \approx x_t + \frac{\sigma^2 h}{2} \nabla \log p_t(x_t) = x_t + \frac{h}{2t}(D_t(x_t) - x_t).$$

This establishes the half drift for general smooth marginals. Correct density evolution still requires an accurate score, a suitable terminal distribution, and numerical integration; additive noise at finite time does not erase the data distribution exactly.

DDIM Sampling and the ODE Perspective

The point-mass calculation motivates a deterministic denoising step. For general data, its small-step limit agrees with the probability-flow ODE derived above; exact finite-step marginal matching is not implied.

Generation (DDIM Sampling):

1. Start with pure noise: $\hat{x}_1 \sim \mathcal{N}(0, \sigma^2 I)$.

2. Iterate backwards from $t = 1$ down to $t = \Delta t$:

$$\hat{x}_{t-\Delta t} = \hat{x}_t + \left(1 - \sqrt{1 - \Delta t/t}\right)\left(f_\theta(\hat{x}_t, t) - \hat{x}_t\right)$$

   where f_θ predicts x_0. The coefficient here is $\lambda(t)\Delta t/t$ because the predictor estimates clean data rather than the mean of just one reverse step.

3. The final output is $\hat{x}_0$.

This process can be viewed as solving an Ordinary Differential Equation (ODE). The update rule is a discretization of a continuous flow defined by a velocity field $v_t(x_t)$:

$$\frac{d\hat{x}}{dt} = v_t(\hat{x}_t) \tag{27.6}$$

The model learns a vector field that flows from any point in space towards the data manifold. The provided figure of a spiral dataset illustrates this: with enough training samples, the learned trajectories (velocity field) correctly flow towards the spiral structure, effectively learning the geometry of the data manifold. This also highlights that the quality of the learned manifold depends on the density of the training data.

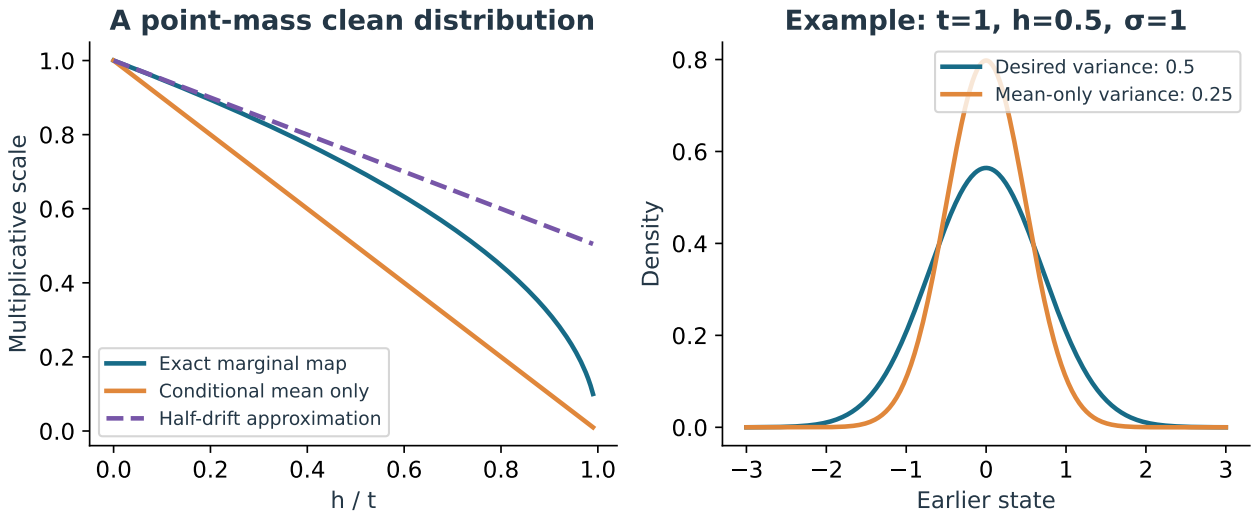


Figure 27.7: For a point-mass data distribution, the correct deterministic map preserves the desired earlier-time variance. Simply using the reverse conditional mean contracts too far.

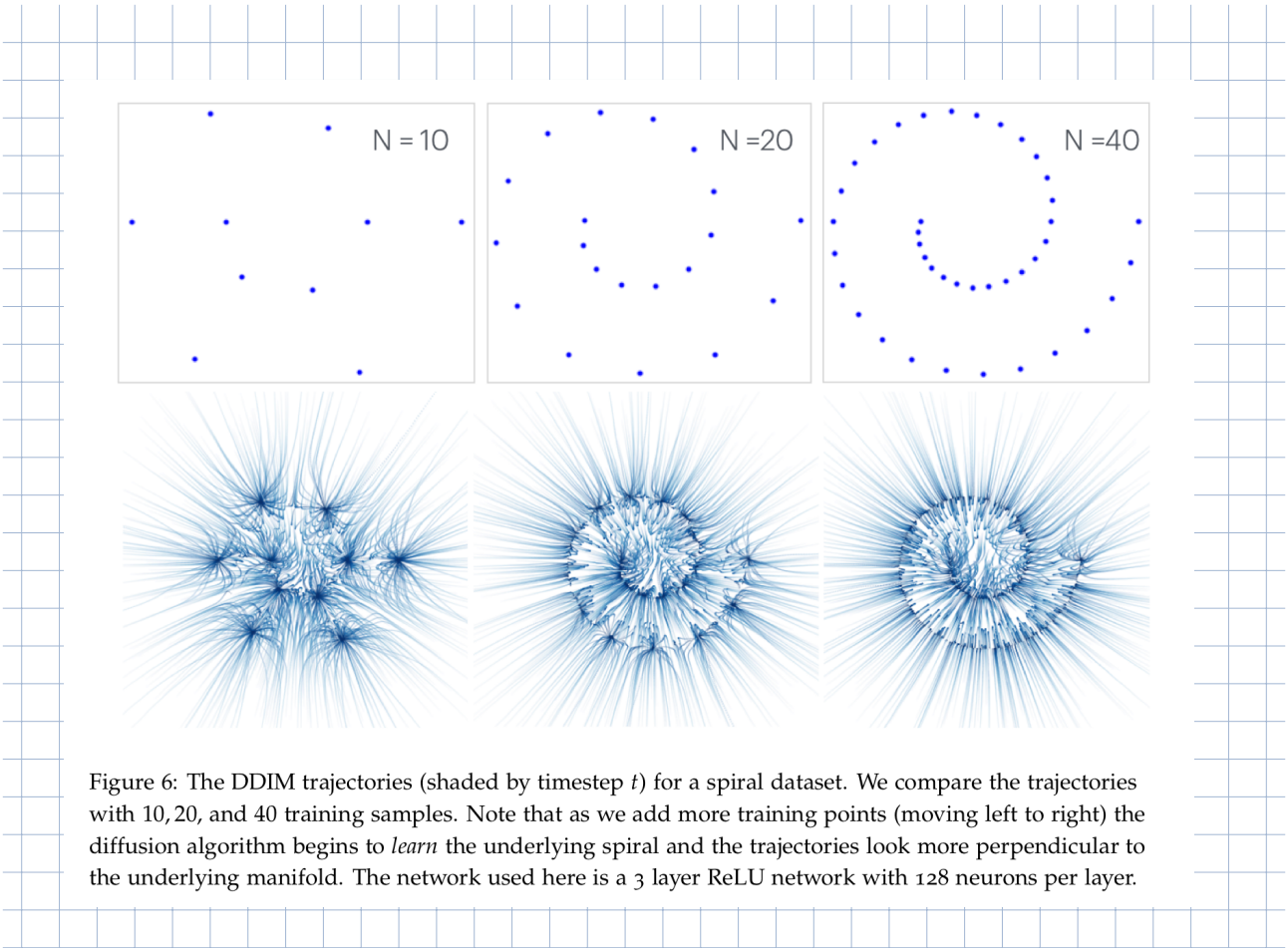


Figure 6: The DDIM trajectories (shaded by timestep t) for a spiral dataset. We compare the trajectories with 10, 20, and 40 training samples. Note that as we add more training points (moving left to right) the diffusion algorithm begins to *learn* the underlying spiral and the trajectories look more perpendicular to the underlying manifold. The network used here is a 3 layer ReLU network with 128 neurons per layer.

Figure 27.8: More training samples reveal the underlying spiral. These columns compare training sets of 10, 20, and 40 points; they are not successive time steps. With more samples, the learned DDIM trajectories better reflect the spiral geometry. The shading along trajectories indicates time. The published figure and its explanatory caption are retained from the handout. Original annotated source excerpt; Anant Sahai, faculty handout 27, p. 9.

Part II

Detailed supplements and worked explanations

Supplement A

Learning functions and differentiating programs

Source trail: Lectures 0–1; Sahai's lecture 1 PDF, Ranade's lecture 1 PDF; optimization draft §§1–4; discussions 1–2. Lecture 0 is available as the August 28 recording; its Ed post contains no separate faculty PDF.

A.1 Constructing a prescribed piecewise-linear function

Return to lecture: Lecture 0, function approximation, Lecture 1, §1.1

Worked construction. Suppose a curve starts at zero with slope 1, changes to slope -2 at $x = 1$, and changes to slope $1/2$ at $x = 3$. Its slope changes are -3 and $5/2$, so

$$f(x) = x - 3\,\mathrm{ReLU}(x - 1) + \frac{5}{2}\,\mathrm{ReLU}(x - 3).$$

For $x < 1$, $f(x) = x$. For $1 \le x < 3$, $f(x) = 3 - 2x$. For $x \ge 3$, $f(x) = \frac{1}{2}x - \frac{9}{2}$. The formulas agree at both knots. The representation exposes the trainable decisions: where to place bends, how much to change each slope, and where to place the whole curve vertically.

View the corresponding lecture diagram (Figure 1, p. 9).

A.2 A numerical backpropagation and finite-difference check

Return to lecture: Lecture 1, §1.4

Numerical example. Let $x = (1, -2)^\top$, $w = (2, 1)^\top$, $b = 1$, $h = \mathrm{ReLU}(w^\top x + b) = 1$, and $\hat{y} = 3h$. For target $y = 1$ and $L = \frac{1}{2}(\hat{y} - y)^2$, the output derivative is $\hat{y} - y = 2$. Therefore $\partial L/\partial h = 6$, $\nabla_w L = (6, -12)^\top$, $\partial L/\partial b = 6$, and the derivative with respect to the output weight is $2h = 2$. Each number is the product of the downstream sensitivity and the local derivative.

At the ReLU kink, the classical derivative is undefined. Implementations select a subgradient, commonly zero. Away from a kink, centered finite differences provide a useful implementation check:

$$\frac{L(\theta + \epsilon e_j) - L(\theta - \epsilon e_j)}{2\epsilon} \approx \frac{\partial L}{\partial \theta_j}.$$

Do not choose ϵ so small that floating-point subtraction dominates, and do not expect agreement across a nondifferentiable kink. A directional check compares $\nabla L^\top v$ with a finite difference along v and can test large parameter sets without one evaluation per parameter.

A.3 Read a matrix product as a shape diagram

Return to lecture: Lecture 1, §1.4

A matrix has rows and columns: an $m \times n$ matrix maps an n-component column vector to an m-component column vector. For $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times p}$, the product $C = AB$ has shape $m \times p$. The repeated inner dimension is summed over; it does not become a dimension of the output. Specifically, $C_{ij} = \sum_{k=1}^{n} A_{ik}B_{kj}$.

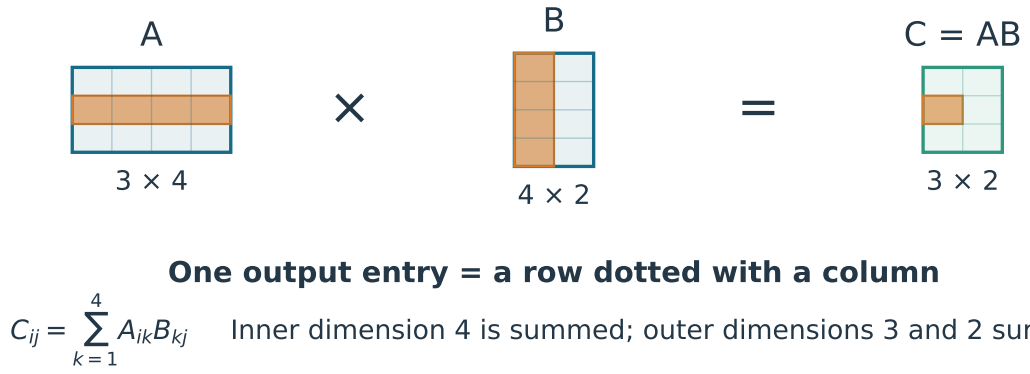


Figure A.1: Rectangles make the contraction visible. The highlighted row of the first matrix and column of the second determine one highlighted output entry.

For example, dotting the row $(1, 2, -1, 0)$ with the column $(3, 1, 2, -2)^\top$ gives $3+2-2+0 = 3$. Repeating that calculation for every row–column pair fills the output. Alternatively, each column of AB is A applied to one column of B. This second interpretation makes a batch of examples natural.

Shape compatibility is necessary but does not prove correctness. Swapping two equal-length axes can preserve a shape while changing the computation. Always name each axis: feature, example, token, key, or output channel. Transposition swaps the two axes; it does not invert a transformation.

A.4 A complete minibatch derivation

Return to lecture: Lecture 1, §1.4

The single-example formulas become more useful when extended to the tensor convention used in an implementation. Put examples in columns, so $X \in \mathbb{R}^{d \times B}$ contains a batch of B inputs. For a hidden layer of width m and a scalar output,

$$Z = WX + b\mathbf{1}_B^\top, \qquad H = \mathrm{ReLU}(Z), \tag{A.1}$$

$$\widehat{Y} = a^\top H + c\mathbf{1}_B^\top, \qquad L = \frac{1}{2B}\|\widehat{Y} - Y\|_F^2. \tag{A.2}$$

Here W is $m \times d$, b and a are m-vectors, and Y is $1 \times B$. Let $R = (\widehat{Y} - Y)/B$. Backpropagation gives

$$\nabla_a L = HR^\top, \qquad \nabla_c L = R\mathbf{1}_B, \tag{A.3}$$

$$G_Z = (aR) \odot \mathbf{1}_{Z>0}, \qquad \nabla_W L = G_Z X^\top, \tag{A.4}$$

$$\nabla_b L = G_Z \mathbf{1}_B, \qquad \nabla_X L = W^\top G_Z. \tag{A.5}$$

Every matrix product has a role. aR distributes each example's output residual to its hidden units. The activation mask blocks inactive routes. Multiplication by $X^\top$ sums the outer-product weight gradients over the batch. Multiplication by $\mathbf{1}_B$ sums the gradients of a bias that is broadcast to every example.

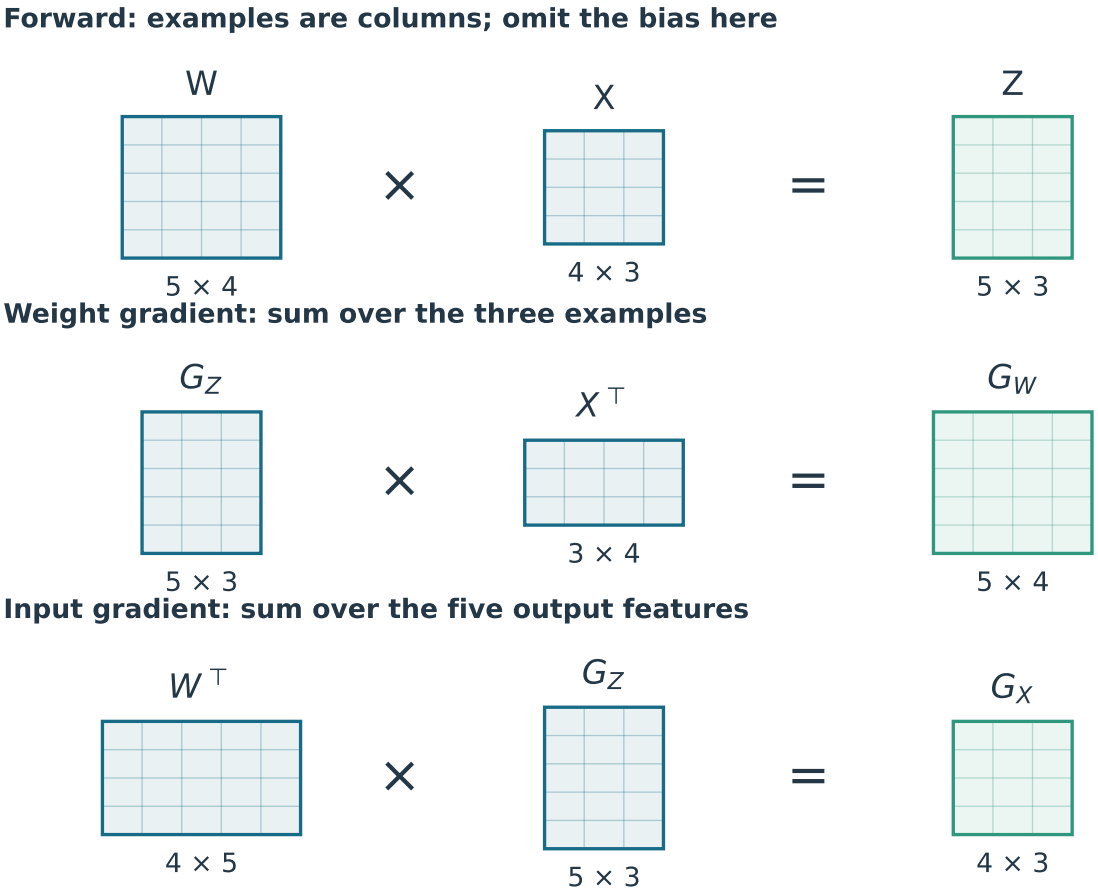


Figure A.2: One dense layer, three products. The forward map contracts input features; the weight gradient contracts examples; the input gradient contracts output features. Bias addition and the activation derivative are handled separately.

For an explicit forward pass, use

$$X = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad W = \begin{bmatrix} 1 & -1 \\ 2 & 1 \end{bmatrix}, \quad b = 0, \quad a = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \quad Y = \begin{bmatrix} 0 & 1 \end{bmatrix}.$$

Then $H = [\begin{smallmatrix} 1 & 0 \\ 2 & 1 \end{smallmatrix}]$, $\widehat{Y} = (-1, -1)$, and $R = (-1/2, -1)$. Thus

$$G_Z = \begin{bmatrix} -1/2 & 0 \\ 1/2 & 1 \end{bmatrix}, \quad \nabla_W L = G_Z, \quad \nabla_b L = \begin{bmatrix} -1/2 \\ 3/2 \end{bmatrix}, \quad \nabla_a L = \begin{bmatrix} -1/2 \\ -2 \end{bmatrix}, \quad \nabla_c L = -3/2.$$

The batch factor appears exactly once. If a software loss already returns a mean, dividing gradients by B again changes the effective learning rate. If a vector is broadcast across both examples and spatial positions, its derivative sums over both sets of repeated uses.

A.5 The Jacobian explains lazy training and feature learning

Return to lecture: Lecture 3, §3.3, Lecture 4, §4.1

For a scalar prediction on N examples, stack predictions into $f(\theta) \in \mathbb{R}^N$ and parameter derivatives into $J \in \mathbb{R}^{N \times P}$. Under squared loss, $\nabla_\theta L = J^\top (f - y)$. A small gradient step changes predictions approximately by

$$\Delta f \approx -\eta J J^\top (f - y).$$

The matrix $K = JJ^\top$ is positive semidefinite, since $v^\top K v = \|J^\top v\|^2 \geq 0$. It describes which residual patterns training can reduce quickly near the current parameters. Large eigenvalues correspond to output directions that respond strongly to parameter updates. A null direction of K cannot be changed by this local linearization.

If J remains nearly fixed, training resembles a kernel or fixed-feature method. If J changes substantially, the accessible directions themselves evolve. This is a precise way to distinguish lazy training from feature learning. Small parameter changes do not always imply small functional changes: the scale and width of the Jacobian matter. Conversely, a large Euclidean parameter change can have little functional effect along a symmetry direction.

For $f = a\,\mathrm{ReLU}(wx)$ with $x > 0, w > 0$, the transformation $(a, w) \mapsto (a/c, cw)$ for $c > 0$ preserves the function. Along this family, parameter norms and coordinate gradients change while predictions do not. A geometry chosen in raw parameter coordinates therefore encodes an implicit preference among equivalent representations. This observation motivates the matrix and activation-based norms developed later.

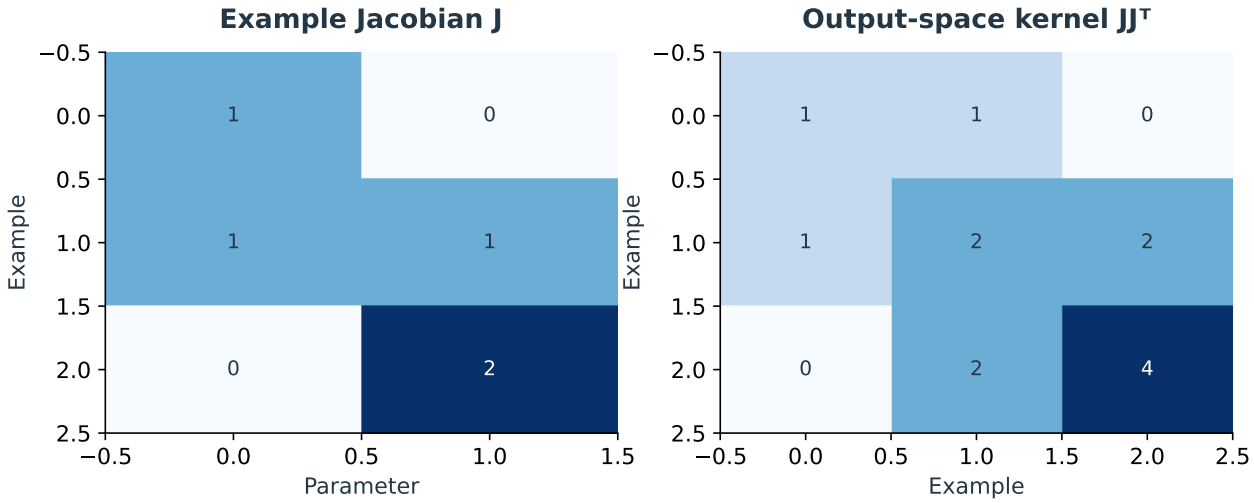


Figure A.3: A local Jacobian maps parameter changes to predictions. Its Gram matrix describes how those changes couple the training examples.

A.6 What a learning curve can diagnose

Return to lecture: Lecture 1, §1.2

Read the gap and the absolute level separately. The two panels are illustrative patterns, not measurements or a guarantee of one cause. In the left pattern, continuing to minimize training loss eventually worsens validation performance; validation selects a checkpoint, and an untouched test set estimates final performance. In the right pattern, first check whether the implementation can fit a tiny dataset. If it cannot, inspect labels, gradients, capacity, and optimization before attributing the result to generalization. Curves should use comparable evaluation conditions: training-mode dropout or augmentation can make a reported training loss harder than validation loss.

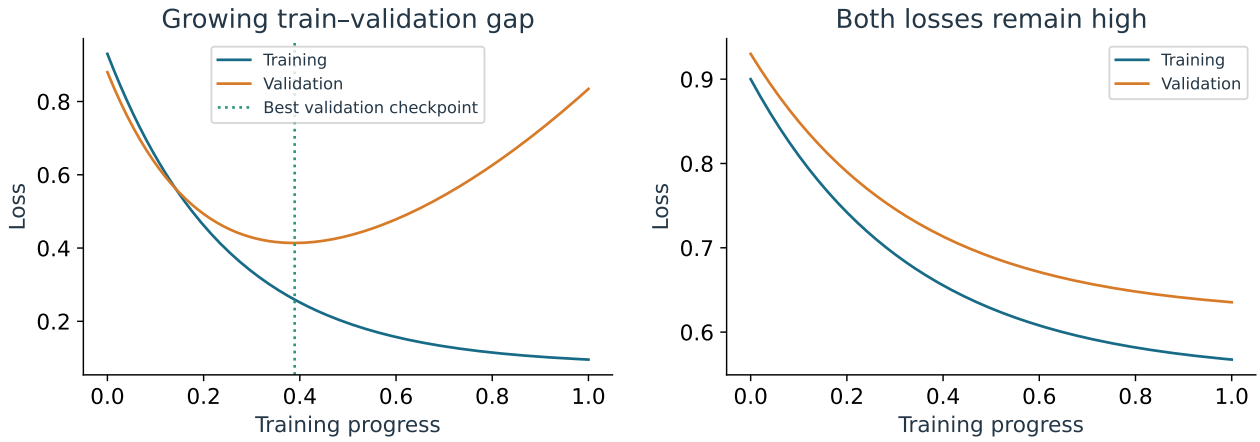


Figure A.4: Learning curves suggest experiments rather than uniquely diagnosing a fault. The plotted curves are synthetic and their scales are illustrative.

A training curve measures the objective on data used to optimize. A validation curve measures a different set of examples under the same evaluation protocol. Low training loss and high validation loss can indicate overfitting, a distribution mismatch, or leakage in the way the experiment is organized. High losses on both sets can indicate inadequate features, an optimization problem, or a faulty input/target pipeline.

A useful controlled experiment is to fit a very small dataset with a deliberately expressive model. If it cannot fit a handful of examples, investigate shapes, labels, loss reduction, gradients, and learning rate. This test does not establish generalization, but it isolates whether the training mechanism works at all.

Standardization statistics must be fitted on the training split. Computing a mean or vocabulary from the entire dataset can expose held-out information, even if labels are not used directly. For grouped or temporal data, split at the level at which future performance is meant to generalize. Randomly separating adjacent frames of the same video can create an easier evaluation than predicting genuinely new videos.

Reasoning exercise. Equal training predictions imply equal empirical loss, even when functions disagree elsewhere. Architecture, regularization, optimization, or additional data can distinguish them; observed loss values alone cannot.

A.7 Numerical stability: preserve the mathematics in finite precision

Return to lecture: Lecture 1, §1.4

A correct symbolic derivative is useful only if the forward and backward calculations remain meaningful on a computer. Floating-point numbers have limited range and precision. Overflow produces an infinite magnitude; underflow can round a tiny nonzero number to zero; cancellation can discard significant digits when nearly equal numbers are subtracted. Changing the algebra can avoid these failures without changing the real-valued function. The examples here are implementation supplements to the course's differentiation and probabilistic-loss calculations.

Log-sum-exp and cross-entropy directly from logits

Let $z \in \mathbb{R}^K$ be finite logits and $q_k = e^{z_k} / \sum_j e^{z_j}$. With $m = \max_j z_j$,

$$\mathrm{LSE}(z) = \log \sum_j e^{z_j} = m + \log \sum_j e^{z_j - m}.$$

Every shifted exponent is nonpositive, so none of the exponentials can overflow. At least one is exactly one. The identity follows by factoring e^m out of the sum. For a normalized target p,

$$\ell(z, p) = \mathrm{LSE}(z) - \sum_k p_k z_k = \log \sum_j e^{z_j - m} - \sum_k p_k (z_k - m).$$

The second form also avoids subtracting two large common offsets. Compute log probabilities as $(z_k - m) - \log \sum_j e^{z_j - m}$, rather than taking the logarithm of already rounded probabilities. Differentiating the original smooth expression gives

$$\frac{\partial \ell}{\partial z_i} = \frac{e^{z_i}}{\sum_j e^{z_j}} - p_i = q_i - p_i.$$

Although the auxiliary maximum is not differentiable at a tie, the complete expression represents a smooth function of finite logits; the shift terms cancel. An implementation must differentiate the complete stable expression consistently.

For $z = (1000, 1001, 999)$, the shifted logits are $(-1, 0, -2)$, the exponential sum is 1.503215, and

$$q \approx (0.244728, 0.665241, 0.090031).$$

For target class 2, the loss is 0.407606 and the gradient is approximately

$$(0.244728, -0.334759, 0.090031).$$

Directly exponentiating the original logits overflows in common floating-point formats, despite this modest loss and gradient.

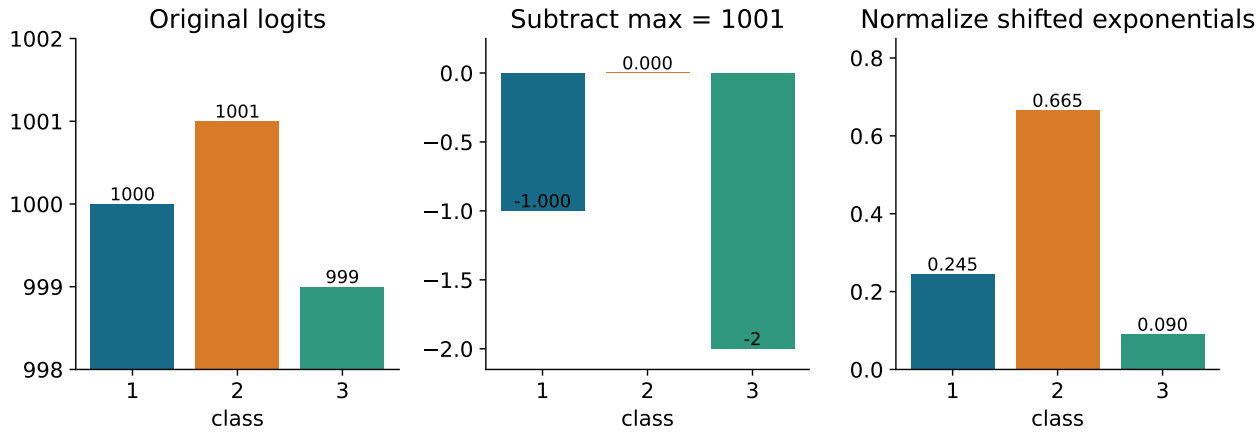


Figure A.5: Subtracting a shared logit offset changes numerical range while preserving every probability ratio. Cross-entropy should use the shifted log probabilities directly.

For logits $(0, -1000)$ and target class 2, the stable loss is approximately 1000. A stored probability for class 2 may underflow to zero; computing $-\log q_2$ afterward then returns infinity. Adding an arbitrary probability floor would cap the loss and change its gradient. The log-domain calculation preserves the intended objective much better.

Binary losses, masking, and long sequences

With binary target $y \in \{0, 1\}$ and logit z, combine sigmoid and binary cross-entropy algebraically:

$$\ell(z, y) = \log(1 + e^z) - yz = \max(z, 0) - yz + \log(1 + e^{-|z|}).$$

The final logarithm is best evaluated with a `log1p` operation, which is designed for small increments. The derivative remains $\sigma(z) - y$. At $z = 1000$, the loss is approximately zero for $y = 1$ and 1000 for $y = 0$. Avoid evaluating separate terms such as $0 \log 0$ in a naive probability-space formula.

In masked attention, assign disallowed positions zero probability by excluding them from the normalization. The $-\infty$ logit convention implements this when there is at least one allowed finite logit in the row. An entirely masked row has no probability distribution to normalize: subtracting its maximum gives $-\infty - (-\infty)$, an undefined operation. Define how padded query rows are handled and exclude them from the loss; do not silently treat this as an ordinary softmax row. A zero target times an infinite negative log probability

also needs explicit masking, since machine arithmetic does not apply the limiting convention automatically.

For autoregressive generation,

$$\log p(x_{1:T}) = \sum_{t=1}^{T} \log p(x_t \mid x_{<t}).$$

Store this sum instead of multiplying thousands of probabilities. Importance ratios can be formed from log-probability differences, and Metropolis–Hastings can compare $\log u$ with a log acceptance ratio. A mathematically impossible event with true probability zero is still impossible: stable arithmetic cannot repair missing support.

The reduction defines the learning objective

For N equally weighted examples, the sum loss has gradient N times that of the mean loss. This factor changes a plain gradient-descent step unless the learning rate is adjusted. With masks $m_{it} \in \{0, 1\}$, a token-average language-model loss is

$$\mathcal{L} = \frac{\sum_{i,t} m_{it} \ell_{it}}{\sum_{i,t} m_{it}}.$$

An example-average loss instead averages each example's token mean, then averages over examples. The two objectives differ when valid lengths differ. Both are legitimate choices; name the intended weighting.

Suppose two microbatches contain 2 and 6 valid tokens, with token-mean losses 1 and 3. The joint token mean is $(2 \cdot 1 + 6 \cdot 3)/8 = 2.5$, whereas averaging the two means gives 2. For gradient accumulation, weight the two mean gradients by $2/8$ and $6/8$, or accumulate token-loss sums and divide by 8. This equivalence assumes the forward function is unchanged by splitting the batch: training BatchNorm, batch-dependent objectives, and changed randomness can break it.

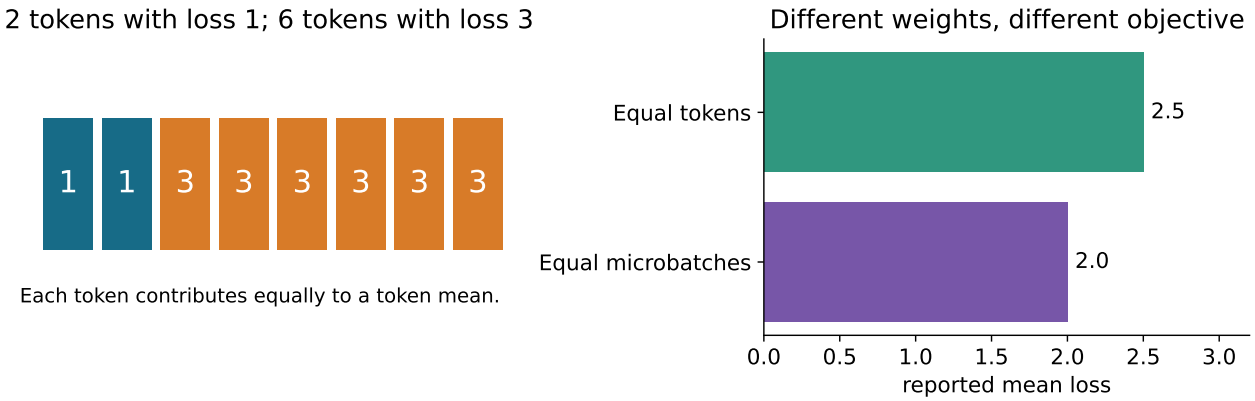


Figure A.6: Equal weight per microbatch and equal weight per valid token answer different questions. The denominator belongs to the objective, not just to its display.

Loss scaling, clipping, and a diagnostic sequence

A lower-precision format can save memory while making small gradients harder to represent. Scaling the scalar loss by $s > 0$ scales its mathematical gradient by s. Recover the original gradient by dividing by s before applying the optimizer and before clipping to the intended norm threshold. In general,

$$\text{clip}(sg, c)/s \neq \text{clip}(g, c).$$

For $g = (3, 4)$, $s = 8$, and $c = 1$, clipping the scaled vector first gives norm $1/8$ after unscaling; unscaling first gives the desired clipped norm 1. Nonfinite intermediate values cannot be repaired merely by dividing the final gradient. Check them, choose an appropriate scale or precision, and recompute a failed step under the implementation's stated rules.

When a derivative check fails, first compare forward values in higher precision, then compare a directional finite difference over several perturbation sizes. Large perturbations incur approximation error; extremely small ones amplify rounding. Fix random draws and the model's state for all evaluations. Changing training/evaluation mode between the analytic and finite-difference calculations changes the function, especially with dropout and BatchNorm. At a ReLU kink, test away from the kink or reason with a one-sided derivative. Finally inspect loss normalization, masked positions, tensor shapes, and nonfinite values before interpreting a training failure as a model limitation.

Supplement B

Least squares, spectral dynamics, and regularization

Source trail: Lectures 1–2; lecture 2; optimization draft §§5–6; discussion 1. This chapter consistently uses a factor of 1/2 in quadratic losses. The draft uses an unhalved loss in several places; learning-rate constants must be adjusted accordingly.

B.1 Deriving least squares in singular coordinates

Return to lecture: Lecture 2, §2.1

Let $X = U_r \Sigma_r V_r^\top$ have rank r. Extend V_r to an orthonormal basis $[V_r\ V_0]$ and decompose $w = V_r a + V_0 b$. Then

$$Xw = U_r \Sigma_r a, \qquad \|Xw - y\|^2 = \|\Sigma_r a - U_r^\top y\|^2 + \|(I - U_r U_r^\top) y\|^2.$$

The three spaces in an SVD. In the rank-three example below, $V_r^\top$ extracts three identifiable coordinates from a four-dimensional parameter vector, Σ_r scales them, and U_r places the result into the five-dimensional observation space. A vector in the missing parameter direction is annihilated by X. Compact SVD discards zero singular modes without pretending that X is square or invertible.

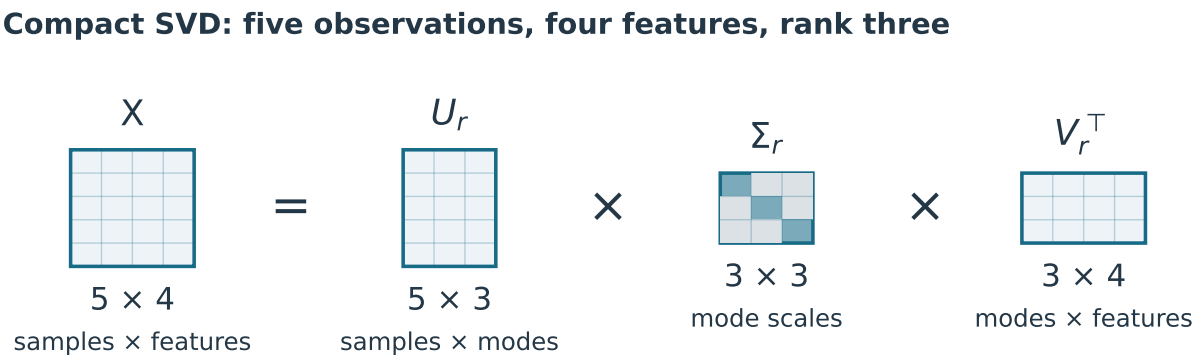


Figure B.1: Compact SVD separates observation, feature, and singular-mode axes. Only the diagonal entries of Σ_r are nonzero; the pictured dimensions are illustrative.

The second term is independent of w: it is the part of the target outside the column space of X. The first term separates into r scalar problems. Its minimizer is $a_i = (u_i^\top y)/\sigma_i$, while b is arbitrary. Setting $b = 0$ gives the minimum-norm solution because the two parameter components are orthogonal.

This decomposition separates three issues that are often conflated. A residual can remain because the target is not representable by the linear model. Nonuniqueness can remain because parameters contain unobserved null directions. Instability can arise because dividing by a very small nonzero singular value amplifies noise. Rank, approximation error, and conditioning are different properties.

Underdetermined example. For one observation $w_1 + 2w_2 = 3$, every point on that line interpolates. The row-space direction is $(1, 2)$, so gradient descent from zero stays on $w = c(1, 2)$. Solving $5c = 3$ gives $w = (0.6, 1.2)$. A null vector is $(-2, 1)$, and starting with an added component $d(-2, 1)$ retains that component throughout training. All such limits fit the observation, but they predict differently on inputs outside the observed row direction.

B.2 Noise, bias, and variance along one singular direction

Return to lecture: Lecture 1, §1.5

Suppose $y = Xw_{\text{true}} + \varepsilon$, with $\mathbb{E}\varepsilon = 0$ and $\text{Cov}(\varepsilon) = \sigma_\varepsilon^2 I$. In a nonzero singular direction, unregularized least squares gives

$$\hat{a}_i = a_{\text{true},i} + \frac{u_i^\top \varepsilon}{\sigma_i}, \qquad \text{Var}(\hat{a}_i) = \frac{\sigma_\varepsilon^2}{\sigma_i^2}.$$

Weakly observed directions have large parameter variance. Ridge produces

$$\hat{a}_{\gamma,i} = \frac{\sigma_i^2}{\sigma_i^2 + \gamma} a_{\text{true},i} + \frac{\sigma_i}{\sigma_i^2 + \gamma} u_i^\top \varepsilon.$$

Ridge is a sequence of shape-preserving coordinate operations. Write $b = U_r^\top y$ and $D_\gamma = \text{diag}(\sigma_i/(\sigma_i^2 + \gamma))$. Then $\widehat{w}_\gamma = V_r D_\gamma b$. The diagonal filter maps singular-mode coordinates to singular-mode coefficients; it neither mixes examples nor changes the number of modes. The fitted observations are $U_r \Sigma_r D_\gamma b$, a different vector with a different axis.

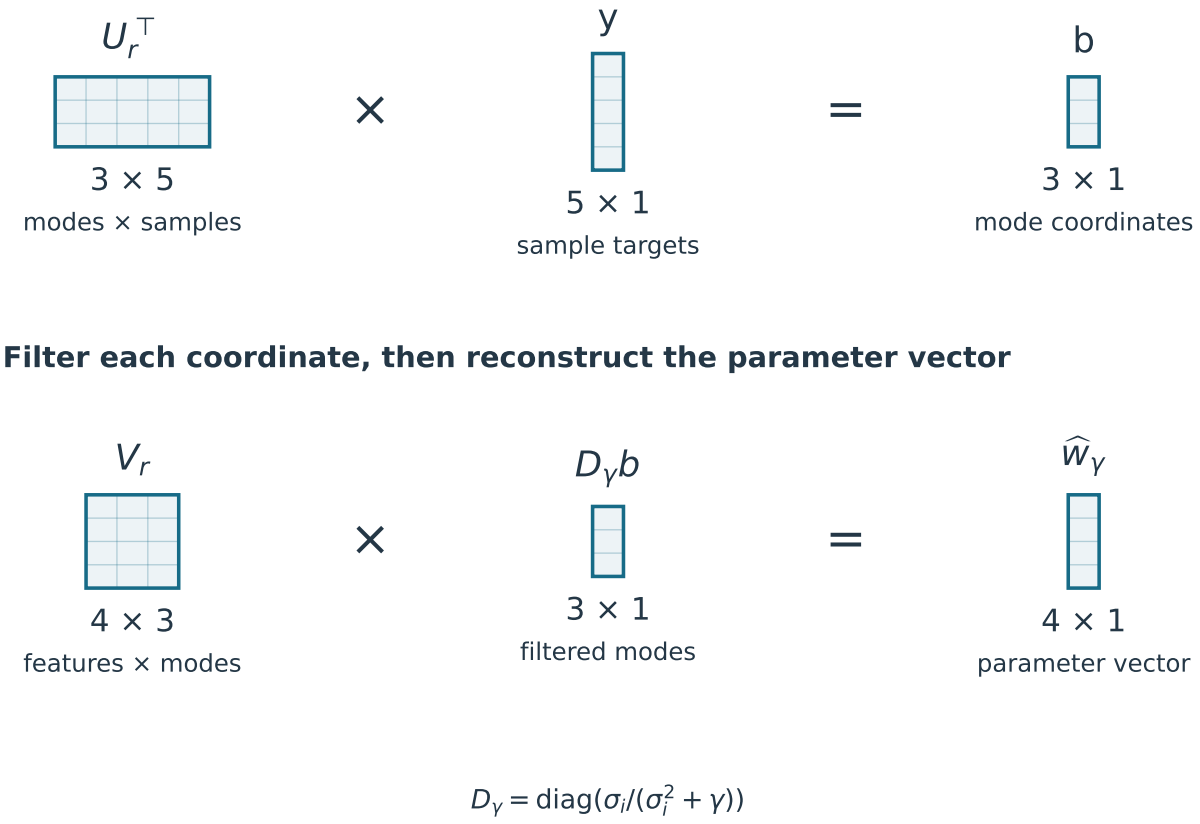


Figure B.2: Ridge regression travels from sample coordinates through singular coordinates to parameter coordinates. The target and the parameter estimate need not have equal lengths.

Its bias is $-\gamma a_{\text{true},i}/(\sigma_i^2+\gamma)$ and its variance is $\sigma_\varepsilon^2\sigma_i^2/(\sigma_i^2+\gamma)^2$. The coordinate mean squared error is therefore

$$\frac{\gamma^2 a_{\text{true},i}^2 + \sigma_\varepsilon^2\sigma_i^2}{(\sigma_i^2+\gamma)^2}.$$

For $a_{\text{true},i} = 1$, $\sigma_i = 0.1$, and $\sigma_\varepsilon = 0.1$, unregularized parameter MSE is 1. Choosing $\gamma = 0.01$ produces squared bias 0.25 and variance 0.25, for total 0.5. Regularization deliberately introduces bias to reduce instability. The same choice need not help a different signal-to-noise ratio.

Parameter error is not the same as prediction error. On the training design, a parameter error in direction v_i is multiplied by σ_i before being observed. A nearly invisible parameter error may become important under a different test-input distribution. This is why generalization claims require a distributional model rather than only a training matrix calculation.

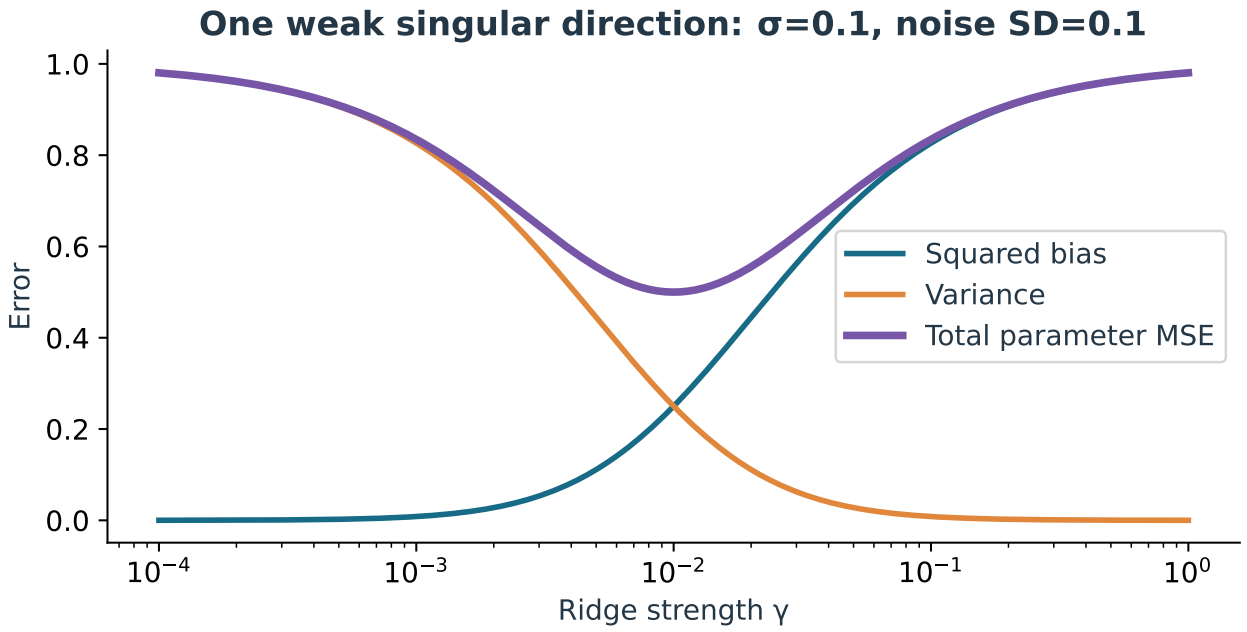


Figure B.3: Ridge trades squared bias against variance along a weakly observed singular direction. The sum can improve even though the estimator becomes biased.

B.3 Gradient flow and early-stopping time scales

Return to lecture: Lecture 2, §2.1

The continuous-time idealization of gradient descent is $\dot{e} = -He$. In eigen-coordinates,

$$z_i(t) = e^{-\lambda_i t} z_i(0).$$

Starting from zero, the prediction filter is $1 - e^{-\sigma_i^2 t}$. Its characteristic learning time is proportional to $1/\sigma_i^2$. A direction with singular value ten times smaller takes approximately one hundred times longer to learn under this model.

For a finite-step algorithm, $(1 - \eta\lambda)^k \approx e^{-\eta k \lambda}$ when $\eta\lambda$ is small. Thus physical training time corresponds roughly to $t = \eta k$ in the gradient-flow limit. Large steps near the stability boundary can oscillate, so they are not well described by a purely monotone exponential approximation.

Early stopping and ridge have different filters. Matching their behavior around one singular value does not make them identical on the whole spectrum. At small $\sigma_i^2 t$, early stopping retains about $\sigma_i^2 t$; ridge retains about σ_i^2/γ when $\sigma_i^2 \ll \gamma$. This gives a rough correspondence $t \sim 1/\gamma$ in the weak-direction regime, not an exact equality of estimators.

B.4 Convexity: chords, tangents, and global optimality

Return to lecture: Lecture 1: empirical risk minimization

Notation and scope. Write the objective as f so that L can denote a smoothness constant without also naming the loss. All norms in this section are Euclidean unless stated otherwise. Definitions refer to a specified convex domain $\mathcal{D}$: the segment joining any two points in $\mathcal{D}$ must stay in $\mathcal{D}$. The gradient-descent rate proofs below concern unconstrained optimization on $\mathbb{R}^d$ with their stated global assumptions. They do not automatically apply to an iterate that leaves a region where a bound was checked.

This mathematical bridge extends the course's least-squares and optimization-draft examples. The explanations, numerical examples, and figures here are independently developed study material. For standard reference treatments, see Boyd and Vandenberghe, *Convex Optimization*, §§3.1 and 9.1–9.3, and Bubeck, *Convex Optimization: Algorithms and Complexity*, Supplements C–D. These are supplementary references, not additional faculty lecture PDFs.

A function is **convex** if, for $x, y \in \mathcal{D}$ and $\alpha \in [0, 1]$,

$$f((1-\alpha)x + \alpha y) \leq (1-\alpha)f(x) + \alpha f(y).$$

The left side evaluates the interpolated input; the right side interpolates the two output values. In one dimension, the graph lies below each chord. This is a global requirement over all pairs, not a judgment based on whether one small part of a plot looks bowl-shaped.

When f is differentiable on an open convex domain, convexity is equivalent to the first-order inequality

$$f(y) \geq f(x) + \nabla f(x)^\top (y - x).$$

Thus every tangent hyperplane is a global lower bound. To see one direction, apply the chord inequality at $x + \alpha(y - x)$, subtract $f(x)$, divide by α, and let $\alpha \downarrow 0$. The resulting directional derivative satisfies $\nabla f(x)^\top (y - x) \leq f(y) - f(x)$.

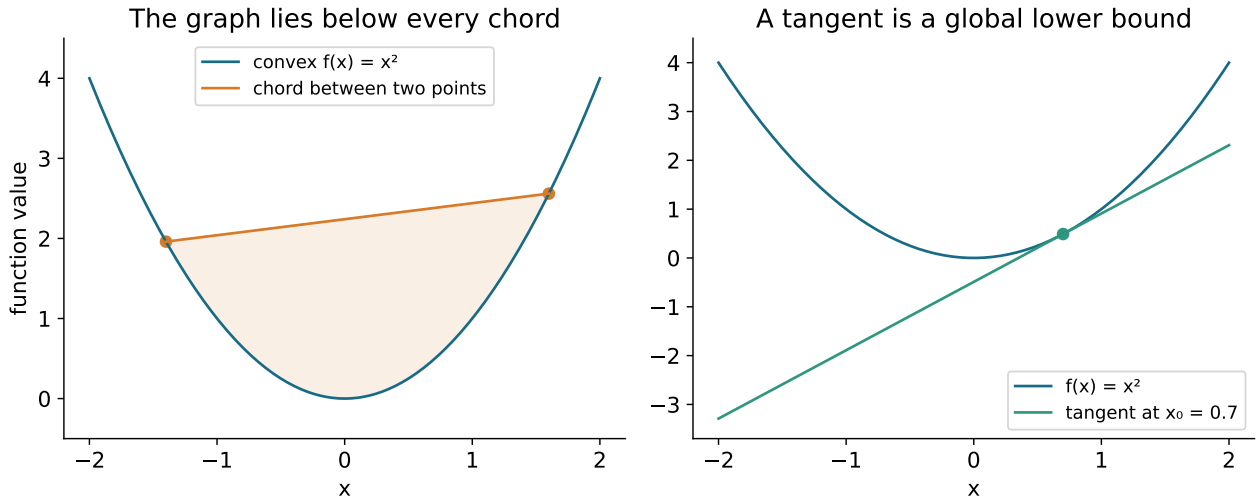


Figure B.4: Two views of convexity for $f(x) = x^2$: chords lie above the graph, while tangents lie below it. The definitions require these relations for every permitted pair or anchor, not just the displayed ones.

Why this matters for training. If $\nabla f(x^*) = 0$ and f is convex, the tangent inequality gives $f(y) \geq f(x^*)$ for every y: a stationary point is globally optimal. This statement is for an unconstrained differentiable problem. At a constrained optimum the gradient need not vanish; for example, minimizing $f(x) = x$ on $[0, 1]$ gives $x^* = 0$ with derivative 1.

Convexity does not promise that a minimizer exists. The convex function e^x has infimum zero on $\mathbb{R}$, but reaches it nowhere. Nor does convexity promise uniqueness: a rank-deficient least-squares problem can have an entire affine family of minimizers. **Strict convexity** makes the chord inequality strict for distinct endpoints and $0 < \alpha < 1$, so a minimizer, if it exists, is unique. Strict convexity still supplies no uniform positive lower curvature: x^4 is the standard example.

B.5 Lipschitz continuity bounds changes in function value

Return to lecture: Lecture 1: empirical risk minimization

A function is G-**Lipschitz** on $\mathcal{D}$ if

$$|f(y) - f(x)| \leq G\|y - x\| \qquad (x, y \in \mathcal{D}).$$

Moving the input by distance r changes the value by at most Gr. The absolute value controls increases and decreases. We use G for this constant to distinguish it from gradient smoothness. If a bound holds with G, it also holds with any larger constant; a quoted constant need not be the smallest possible one.

For a differentiable function on an open convex domain, a uniform gradient bound $\|\nabla f(z)\| \leq G$ gives Lipschitz continuity. Set $d = y - x$ and integrate along the segment:

$$f(y) - f(x) = \int_0^1 \nabla f(x + td)^\top d \, dt, \qquad |f(y) - f(x)| \leq \int_0^1 G\|d\| \, dt = G\|d\|.$$

Conversely, if a differentiable function is G-Lipschitz on an open domain, take directional difference quotients in any unit direction v to get $|\nabla f(x)^\top v| \leq G$. Maximizing over v yields $\|\nabla f(x)\| \leq G$. Convexity of the function is not needed for either implication; the segment and domain conditions do matter.

A corner can still be Lipschitz. The reverse triangle inequality gives $\big||y| - |x|\big| \leq |y - x|$, so $|x|$ is 1-Lipschitz even though it is not differentiable at zero. ReLU is also 1-Lipschitz. A slope bound does not prevent an abrupt change in slope.

The domain changes the answer. For $f(x) = x^2/2$, the derivative is x. There is no finite global function-Lipschitz constant on $\mathbb{R}$: setting $y = x + 1$ gives $|f(y) - f(x)| = |x + 1/2|$, which is unbounded. On $[-R, R]$, however, the derivative is bounded by R, so f is R-Lipschitz there. Always attach a domain to claims about global constants.

B.6 L-smoothness bounds changes in the gradient

Return to lecture: Lecture 1: empirical risk minimization

A differentiable function is L-**smooth** if its gradient is L-Lipschitz:

$$\|\nabla f(y) - \nabla f(x)\| \le L\|y - x\|.$$

This compares two gradients; it does not say that either gradient has norm at most L. The function $x^2/2$ is globally 1-smooth because $|f'(y) - f'(x)| = |y - x|$, despite its unbounded derivative and lack of global function-Lipschitz continuity.

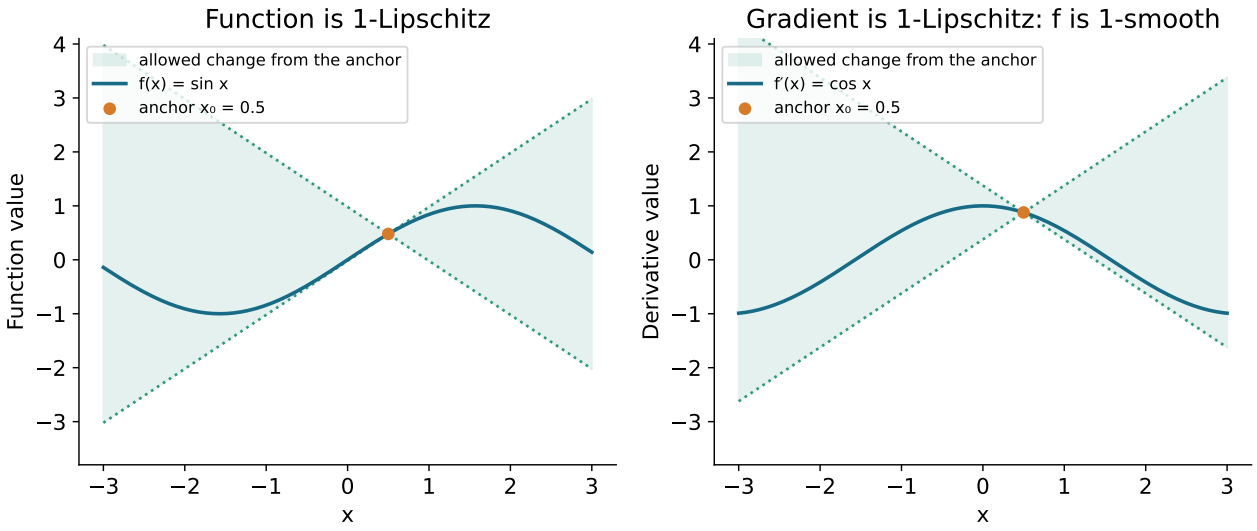


Figure B.5: Lipschitz continuity controls vertical changes in the function; smoothness controls changes in the derivative. For $f(x) = \sin x$, both constants can be 1. Each cone is drawn around one anchor; the inequalities must hold around all anchors.

For a twice continuously differentiable function on an open convex domain, a sufficient and necessary uniform smoothness condition is

$$\|\nabla^2 f(x)\|_{\text{op}} \le L.$$

The Hessian is symmetric, so this bounds the absolute value of every eigenvalue:

$$-LI \preceq \nabla^2 f(x) \preceq LI.$$

Smoothness permits negative curvature. For a *convex* twice differentiable function the Hessian is positive semidefinite, so the combined condition becomes $0 \preceq \nabla^2 f(x) \preceq LI$. Writing only an upper Hessian bound for a general nonconvex function misses potentially large negative eigenvalues.

Deriving the quadratic approximation error. With $d = y - x$, subtract the linear term from the line integral:

$$f(y) - f(x) - \nabla f(x)^\top d = \int_0^1 \left(\nabla f(x + td) - \nabla f(x)\right)^\top d \, dt,$$

$$\left|f(y) - f(x) - \nabla f(x)^\top d\right| \le \int_0^1 Lt\|d\|^2 \, dt = \frac{L}{2}\|d\|^2.$$

The factor $1/2$ comes from integrating t. In particular, the **descent lemma** is

$$f(y) \le f(x) + \nabla f(x)^\top (y - x) + \frac{L}{2}\|y - x\|^2.$$

This upper quadratic model holds without convexity. The corresponding lower model has $-L\|y - x\|^2/2$; it is not the positive-curvature lower model supplied by strong convexity.

B.7 Strong convexity supplies a positive lower curvature

Return to lecture: Lecture 1: optimization challenges

A differentiable function is **μ-strongly convex**, with $\mu > 0$, when

$$f(y) \ge f(x) + \nabla f(x)^\top (y - x) + \frac{\mu}{2}\|y - x\|^2.$$

Equivalently, $f(x) - \mu\|x\|^2/2$ is convex. The chord formulation subtracts a positive interpolation correction:

$$f((1-\alpha)x + \alpha y) \le (1-\alpha)f(x) + \alpha f(y) - \frac{\mu}{2}\alpha(1-\alpha)\|x - y\|^2.$$

For a twice continuously differentiable function, strong convexity is equivalent to $\nabla^2 f(x) \succeq \mu I$ throughout the domain. If it is also L-smooth, then

$$\mu I \preceq \nabla^2 f(x) \preceq L I, \qquad 0 < \mu \le L.$$

Around each anchor, a lower quadratic of curvature μ and an upper quadratic of curvature L enclose the function. At an unconstrained minimizer,

$$f(x) - f(x^*) \ge \frac{\mu}{2}\|x - x^*\|^2.$$

Thus a small objective gap controls parameter distance. For a finite differentiable strongly convex function on all of $\mathbb{R}^d$, the quadratic lower bound forces growth at infinity and ensures an attained, unique minimum. Domain restrictions can change existence: x^2 on $(0, 1)$ does not attain its infimum.

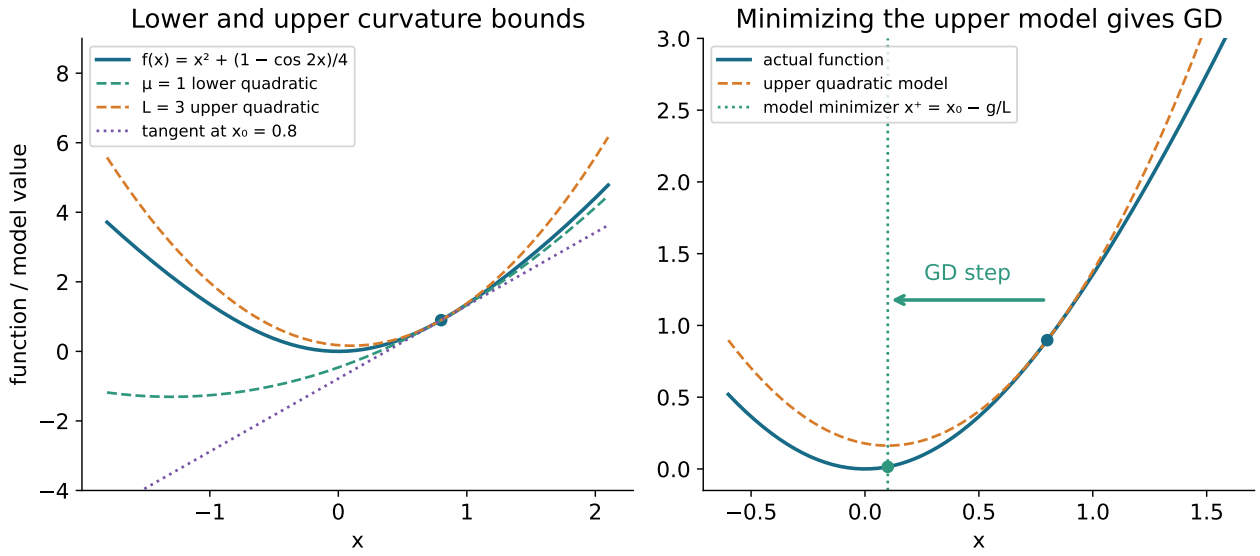


Figure B.6: For $f(x) = x^2 + (1 - \cos 2x)/4$, the second derivative is $2 + \cos 2x \in [1, 3]$. Hence $\mu = 1$ and $L = 3$ give valid global lower and upper quadratic models. The right panel minimizes the upper model at a single anchor; this produces one gradient-descent step.

Strict is weaker than strong. The derivative $4x^3$ of x^4 is strictly increasing, so the function is strictly convex. But $f''(0) = 0$ rules out every $\mu > 0$ on any interval containing zero. It also has no finite global smoothness constant on $\mathbb{R}$, since $f''(x) = 12x^2$ is unbounded. On $[-R, R]$ it is $12R^2$-smooth. Its visually flat minimum is exactly the feature a positive strong-convexity constant would prohibit.

B.8 Counterexamples keep the assumptions separate

Return to lecture: Lecture 1: optimization challenges

The four properties answer different questions. Convexity is a global shape condition; strong convexity adds a minimum amount of curvature; Lipschitz continuity bounds slopes where derivatives exist; smoothness bounds how quickly those slopes change.

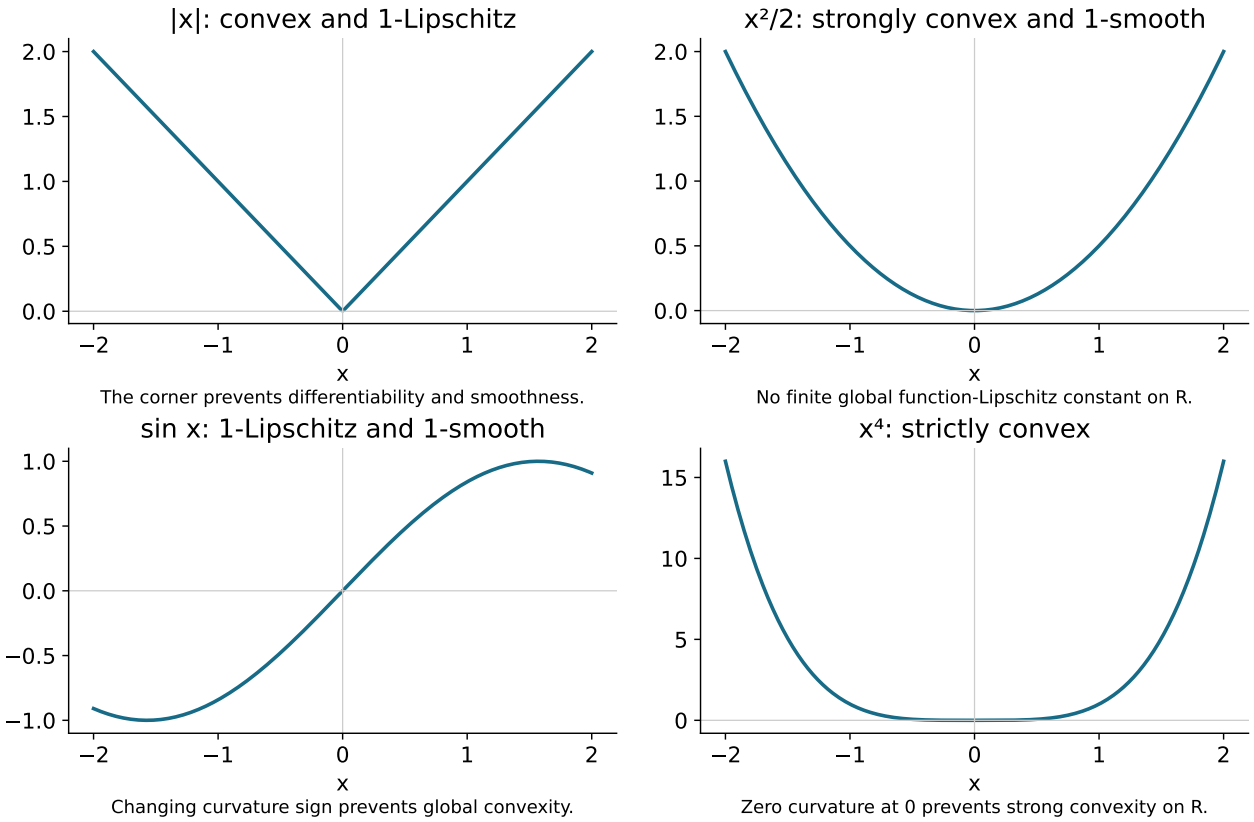


Figure B.7: Four one-dimensional counterexamples. The displayed interval is finite, but the claims in the titles and notes concern the whole real line. In particular, a finite plot cannot establish a global Lipschitz or smoothness constant.

For $\sin x$, $|f'(x)| = |\cos x| \le 1$ and $|f''(x)| = |\sin x| \le 1$. It is both globally 1-Lipschitz and 1-smooth, but its Hessian changes sign, so it is not convex on $\mathbb{R}$. At $x = \pi/2$ its derivative vanishes at a maximum. This is a concrete warning that a small gradient need not certify a small optimality gap.

Another useful incompatibility is that a strongly convex differentiable function on all of $\mathbb{R}^d$ cannot have a finite global function-Lipschitz constant. Along a ray away from its minimizer, strong convexity requires growth at least $\mu r^2/2$, whereas Lipschitz continuity would allow at most Gr. Both can hold on a bounded domain, but not for arbitrarily large r.

B.9 A safe gradient step comes from the upper model

Return to lecture: Lecture 1: convergence and step size

Let $g = \nabla f(x)$ and $x^+ = x - \eta g$. Substitute this update into the descent lemma:

$$\begin{aligned} f(x^+) &\le f(x) + g^\top(-\eta g) + \frac{L}{2}\|\eta g\|^2 \\ &= f(x) - \eta\left(1 - \frac{L\eta}{2}\right)\|g\|^2. \end{aligned}$$

For $L > 0$, every $0 < \eta < 2/L$ ensures strict decrease when $g \neq 0$. At $\eta = 2/L$, the bound permits no decrease; for $f(x) = Lx^2/2$, the iterate changes sign and its objective remains unchanged. Larger steps can increase the objective. A step outside the sufficient interval may still decrease a particular function at a particular point; the uniform guarantee is what has been lost.

Choosing $\eta = 1/L$ gives the especially useful bound

$$f(x^+) \leq f(x) - \frac{1}{2L}\|g\|^2.$$

It also minimizes $g^\top u + L\|u\|^2/2$, the change predicted by the upper quadratic model. This connects smoothness to the chapter's local-model interpretation of gradient descent. An overestimate of L is safe but conservative; an underestimate may invalidate the step guarantee. When a global constant is unknown, a line search can test an appropriate decrease condition instead of treating an estimated curvature as a proof.

Concrete step. For $f(x) = x^2 + (1 - \cos 2x)/4$ at $x = 0.8$, $g = 1.6 + 0.5 \sin 1.6 \approx 2.09979$. With $L = 3$, the model-minimizing step gives $x^+ \approx 0.100071$. The actual function drops from approximately 0.897300 to 0.0150046; the upper bound only promises $f(x^+) \leq 0.162449$. The guarantee can be conservative without being wrong.

B.10 Smooth convex gradient descent has a 1/k gap bound

Return to lecture: Lecture 1: convergence and step size

Assume f is convex and L-smooth on $\mathbb{R}^d$, with an attained minimizer x^*, and use $x_{k+1} = x_k - g_k/L$. Define $\Delta_k = f(x_k) - f(x^*)$ and $D_k = \|x_k - x^*\|^2$. Smoothness gives

$$\Delta_{k+1} \leq \Delta_k - \frac{1}{2L}\|g_k\|^2,$$

while convexity gives $\Delta_k \leq g_k^\top (x_k - x^*)$. Combine them and expand the squared distance:

$$\begin{aligned}
\Delta_{k+1} &\leq g_k^\top (x_k - x^*) - \frac{1}{2L}\|g_k\|^2, \\
D_{k+1} &= D_k - \frac{2}{L} g_k^\top (x_k - x^*) + \frac{1}{L^2}\|g_k\|^2, \\
\Delta_{k+1} &\leq \frac{L}{2}(D_k - D_{k+1}).
\end{aligned}$$

Summing from $k = 0$ to $T - 1$ telescopes the distances:

$$\sum_{k=0}^{T-1} \Delta_{k+1} \leq \frac{L}{2}(D_0 - D_T) \leq \frac{L}{2} D_0.$$

The objective decreases at each step, so every gap in this sum is at least Δ_T. Therefore

$$\boxed{f(x_T) - f(x^*) \leq \frac{L\|x_0 - x^*\|^2}{2T}}, \qquad T \geq 1.$$

This is a last-iterate objective-gap guarantee. It says that $T \geq L\|x_0 - x^*\|^2/(2\varepsilon)$ suffices for gap at most ε. It does not say every trajectory decays exactly as $1/T$, nor does it control distance to a particular minimizer without additional structure. A rank-deficient quadratic may converge quickly in its observed subspace while leaving its null-space component unchanged.

B.11 Strong convexity turns descent into geometric convergence

Return to lecture: Lecture 2, §2.1

Assume in addition that f is μ-strongly convex. Minimize its lower quadratic model over y:

$$f(x^*) \geq \min_y \left\{ f(x) + g^\top (y - x) + \frac{\mu}{2}\|y - x\|^2 \right\} = f(x) - \frac{\|g\|^2}{2\mu}.$$

Consequently $\|g\|^2 \geq 2\mu(f(x) - f(x^*))$. Insert this into the smoothness descent bound:

$$\Delta_{k+1} \leq \Delta_k - \frac{\|g_k\|^2}{2L} \leq \left(1 - \frac{\mu}{L}\right)\Delta_k, \qquad \boxed{\Delta_T \leq \left(1 - \frac{\mu}{L}\right)^T \Delta_0}.$$

For $\kappa = L/\mu > 1$, the inequality $1 - 1/\kappa \leq e^{-1/\kappa}$ gives the sufficient iteration count $T \geq \kappa \log(\Delta_0/\varepsilon)$ when $0 < \varepsilon < \Delta_0$. This is often called **linear convergence**: the error is multiplied by a fixed factor each iteration, so it decreases geometrically, not linearly as a function of iteration number. If $L = \mu$, the bound gives a zero gap after one step.

Combining with quadratic growth yields $\|x_T - x^*\|^2 \leq 2\Delta_T/\mu$. Notice which quantity is being bounded: objective gap, squared distance, distance, and gradient norm have different conversions and potentially different exponents.

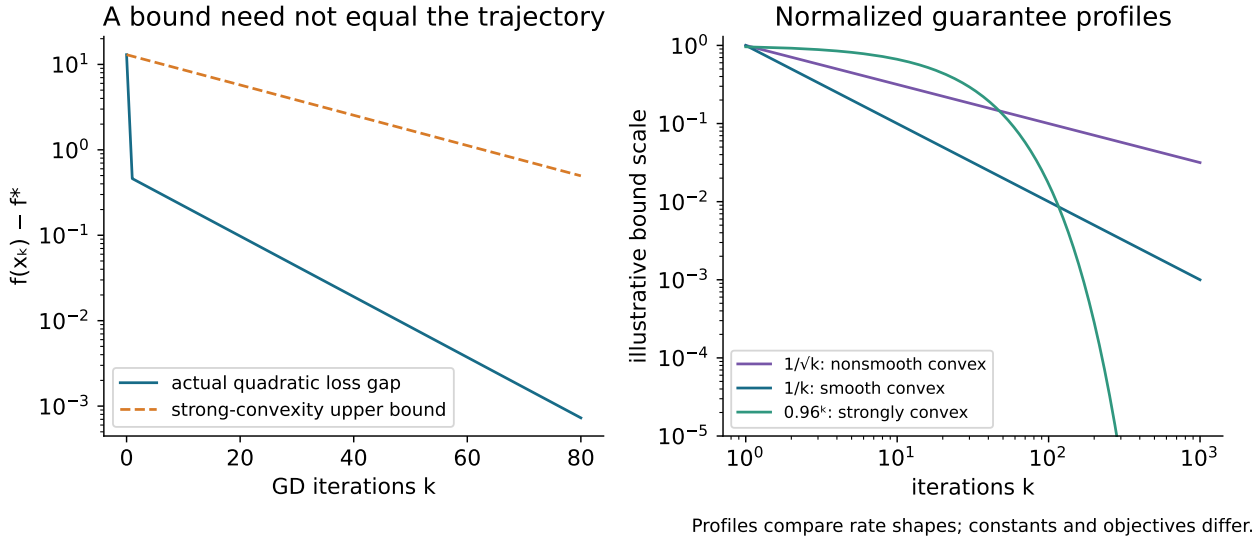


Figure B.8: Left: actual GD on $f(x) = \frac{1}{2}(x_1^2 + 25x_2^2)$ from $(1, 1)$, with step $1/25$, compared with the valid general strong-convexity bound. Right: illustrative normalized rate profiles; these are not measurements on a common objective and do not remove the constants in the proved bounds.

Recovering the earlier spectral example. For $H = \mathrm{diag}(1, 25)$, $\mu = 1$, $L = 25$, and $\kappa = 25$. With step $1/L$, the coordinate factors are $(0.96, 0)$. Starting at $(1, 1)$, $\Delta_0 = 13$ and, for $k \geq 1$, the exact gap is $\frac{1}{2}(0.96)^{2k}$. The general bound is $13(0.96)^k$, which is looser but valid.

The earlier quadratic-specific step $2/(L + \mu)$ instead gives worst-case *distance* factor $(\kappa - 1)/(\kappa + 1)$. For a positive-definite quadratic the objective gap contracts by at most the square of that factor. This is a sharper quadratic spectral calculation than the general $1/L$ proof above; comparing the factors without also comparing the step size and measured error is misleading.

B.12 Without smoothness: a subgradient guarantee

Return to lecture: Lecture 2, §2.1

For a convex function that may have corners, a subgradient $g \in \partial f(x)$ satisfies $f(y) \geq f(x) + g^\top(y - x)$ for every y. For $|x|$, the subgradient set at zero is $[-1, 1]$. A subgradient is a supporting slope, not necessarily a direction whose negative produces immediate decrease: from the minimum of $|x|$, selecting $g = 1$ and taking a positive step moves away from the minimum.

Suppose an attained minimizer exists, $\|x_0 - x^*\| \leq R$, and the selected subgradients satisfy $\|g_k\| \leq G$. For unconstrained updates $x_{k+1} = x_k - \eta g_k$,

$$\|x_{k+1} - x^*\|^2 \leq \|x_k - x^*\|^2 - 2\eta(f(x_k) - f(x^*)) + \eta^2 G^2.$$

Sum for $k = 0, \ldots, T - 1$ and divide by $2\eta T$. Convexity at the averaged iterate $\bar{x}_T = T^{-1} \sum_{k=0}^{T-1} x_k$ gives

$$f(\bar{x}_T) - f(x^*) \leq \frac{R^2}{2\eta T} + \frac{\eta G^2}{2}.$$

For $R, G > 0$, the horizon-dependent choice $\eta = R/(G\sqrt{T})$ balances the two terms and gives $f(\bar{x}_T) - f(x^*) \leq RG/\sqrt{T}$. The best observed objective also obeys this bound. This proof does not give the same bound for the last iterate. Projecting onto a closed convex constraint set permits a corresponding analysis, but unprojected updates must not be assumed to remain inside a bounded domain where a Lipschitz constant was established.

B.13 Without convexity: descent can certify stationarity only

Return to lecture: Lecture 1: optimization challenges

Suppose f is L-smooth on $\mathbb{R}^d$ and bounded below by $f_{\inf} > -\infty$, but may be nonconvex. The $1/L$ descent bound still telescopes:

$$\frac{1}{2L} \sum_{k=0}^{T-1} \|\nabla f(x_k)\|^2 \leq f(x_0) - f(x_T) \leq f(x_0) - f_{\inf}.$$

Thus

$$\boxed{\min_{0\le k<T} \|\nabla f(x_k)\|^2 \le \frac{2L(f(x_0)-f_{\inf})}{T}}.$$

To guarantee that some iterate has gradient norm at most ε, this bound requires order $1/\varepsilon^2$ iterations. It controls the *squared* gradient norm at the best iterate in this sense, not the objective gap of the last iterate. A stationary point can be a saddle or maximum, as $\sin x$ already demonstrated. Smooth nonconvex descent alone is not a global-optimality theorem.

The assumptions form a useful guide: smoothness supplies descent; convexity converts a tangent into an optimality comparison; strong convexity ties the gradient and distance to the objective gap. Removing an assumption changes the conclusion that is justified, not merely a constant in the same conclusion.

Read the guarantee together with its assumptions and iterate. In the table, $R = \|x_0 - x^*\|$ and $\Delta_0 = f(x_0) - f(x^*)$ when a minimizer is assumed. Smooth cases use step $1/L$; the subgradient case uses the horizon-dependent step derived above.

Assumptions	Certified quantity	Bound after T steps
Convex, bounded subgradients	Average-iterate objective gap	$RG/\sqrt{T}$
Convex and L-smooth	Last-iterate objective gap	$LR^2/(2T)$
μ-strongly convex and L-smooth	Last-iterate objective gap	$(1-\mu/L)^T\Delta_0$
L-smooth and bounded below	Smallest squared gradient norm among the first T iterates	$2L(f(x_0)-f_{\inf})/T$

B.14 Applying the constants to losses and neural parameterizations

Return to lecture: Lecture 1: optimization challenges

Logistic regression is a useful bridge. With $y_i \in \{-1, 1\}$, define the mean binary logistic loss

$$f(w) = \frac{1}{N}\sum_{i=1}^{N} \log(1 + e^{-y_i x_i^\top w}).$$

Let $p_i = \sigma(y_i x_i^\top w)$ and let X have rows $x_i^\top$. Differentiating twice gives

$$\nabla^2 f(w) = \frac{1}{N} X^\top D(w) X, \qquad D_{ii} = p_i(1-p_i) \in [0, 1/4].$$

Hence f is convex and globally L-smooth with valid constant $L = \|X\|_{\text{op}}^2/(4N)$. Its gradient is a sum of feature vectors with coefficients of magnitude at most $1/N$, so $G = N^{-1}\sum_i \|x_i\|$

is a valid global function-Lipschitz constant. Both properties hold here, but they follow from different bounds.

There need not be a uniform positive strong-convexity constant: probabilities can saturate and make D_{ii} arbitrarily small. On separable data, an unregularized logistic loss can approach its infimum as parameter norms diverge without attaining a finite minimizer. Adding $\lambda\|w\|^2/2$ with $\lambda > 0$ to all coordinates yields $\mu = \lambda$ as a valid lower constant and $L = \|X\|_{\text{op}}^2/(4N) + \lambda$ as a valid upper constant. An unregularized intercept must be handled separately.

Worked constants. For two scalar examples $x_1 = 1, x_2 = 2$, the unregularized bounds are $G = (1+2)/2 = 1.5$ and $L = (1^2 + 2^2)/(4 \cdot 2) = 5/8$. With $\lambda = 1/8$, valid constants become $\mu = 1/8$, $L = 3/4$, and $\kappa = 6$. Step $1/L = 4/3$ is valid under this objective's scaling, with gap factor at most $5/6$. The added quadratic removes global function-Lipschitz continuity, despite retaining smoothness. If the loss were summed instead of averaged, its data-dependent constants would be multiplied by N.

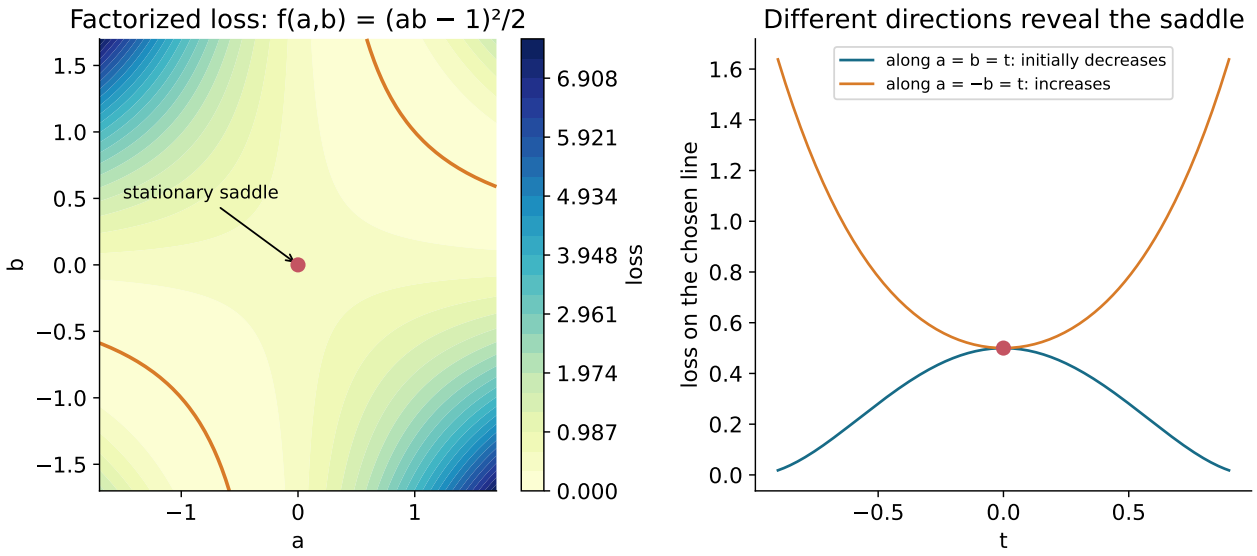


Figure B.9: A convex squared loss in the prediction becomes nonconvex under a two-layer scalar parameterization. The orange curves satisfy $ab = 1$ and attain zero loss. At the origin the gradient vanishes, but opposite directions reveal negative and positive curvature.

For prediction z, the loss $\ell(z) = \frac{1}{2}(z-1)^2$ is convex. Parameterize $z = ab$ and obtain $f(a,b) = \frac{1}{2}(ab-1)^2$. Its gradient is $((ab-1)b, (ab-1)a)$ and

$$\nabla^2 f(a,b) = \begin{pmatrix} b^2 & 2ab-1 \\ 2ab-1 & a^2 \end{pmatrix}, \qquad \nabla^2 f(0,0) = \begin{pmatrix} 0 & -1 \\ -1 & 0 \end{pmatrix}.$$

The eigenvalues at zero are -1 and 1. Convexity in the network's output therefore does not establish convexity in its weights. A neural network's Lipschitz constant with respect to *input data* also differs from the smoothness of its training objective with respect to *parameters*. ReLU networks can have parameter-space kinks; even smooth activations do not automatically give a finite global Hessian bound over unbounded weights.

Checks for understanding. First, explain why x^4 is smooth on $[-1, 1]$ but not globally smooth: the respective Hessian bounds are 12 and unbounded. Second, identify the guarantee available for globally smooth $\sin x$: the stationarity bound can apply because the function is bounded below, but convex global-optimality reasoning cannot. Third, consider rank-deficient least squares: it is convex and smooth, but a positive global strong-convexity constant is unavailable; adding a positive ridge penalty on every coordinate supplies one. Finally, distinguish $O(1/T)$ in an objective-gap bound from $O(1/T)$ in a squared-gradient bound before translating either into an iteration budget.

Supplement C

Stochastic gradients, momentum, Adam, and weight decay

Source trail: lecture 2, lecture 3; optimization draft §§7–9; discussion 2. The faculty correction in Ed #58 emphasizes a uniform contraction factor, not merely a loss that decreases.

C.1 Persistent noise creates a floor

Return to lecture: Lecture 3, §3.1

If individual examples disagree, their gradients need not vanish at the empirical minimizer. A scalar local model is

$$e_{t+1} = (1 - \eta\lambda)e_t - \eta\xi_t, \quad \mathbb{E}[\xi_t] = 0, \quad \operatorname{Var}(\xi_t) = \sigma^2.$$

For $\lambda > 0$ and $0 < \eta < 2/\lambda$, the deterministic mode is stable. Under the simplifying assumption that the noise is independent across steps and of e_t, a stationary variance satisfies

$$V = (1 - \eta\lambda)^2 V + \eta^2\sigma^2,$$

hence

$$V = \frac{\eta\sigma^2}{2\lambda - \eta\lambda^2}.$$

Small constant steps produce a small neighborhood, not exact mean-square convergence. Diminishing steps, variance reduction, or interpolation can change that conclusion. The mean iterate may approach the optimum while individual iterates continue to fluctuate.

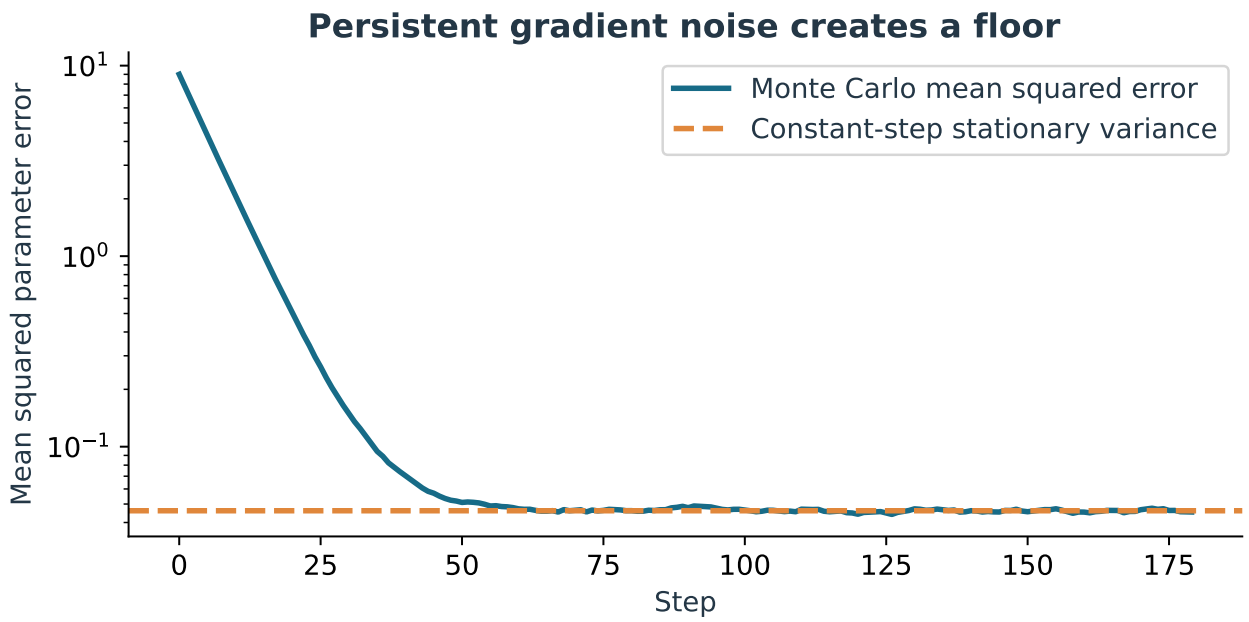


Figure C.1: Noisy updates near a quadratic optimum. Smaller steps and larger batches reduce the stationary fluctuations in this illustrative model.

C.2 An explicit contraction proof on the identifiable subspace

Return to lecture: Lecture 2, §2.4

Assume every example is interpolated by w^*, so $y_i = x_i^\top w^*$, and define $e = w - w^*$. A single-example half-squared-loss update is

$$e^+ = (I - \eta x_i x_i^\top)e.$$

Expanding rather than invoking a general theorem gives

$$\|e^+\|^2 = \|e\|^2 - 2\eta(x_i^\top e)^2 + \eta^2\|x_i\|^2(x_i^\top e)^2.$$

If $\|x_i\|^2 \le R^2$ and $0 < \eta < 2/R^2$, then

$$\mathbb{E}[\|e^+\|^2 \mid e] \le \|e\|^2 - \eta(2 - \eta R^2)e^\top \overline{H} e, \qquad \overline{H} = \mathbb{E}[x_i x_i^\top].$$

On the span of the data, suppose the smallest positive eigenvalue of $\overline{H}$ is $\mu > 0$. Then the row-space component contracts by at most

$$C = 1 - \eta(2 - \eta R^2)\mu < 1.$$

Iterating conditional expectation yields $\mathbb{E}\|e_t\|^2 \le C^t\|e_0\|^2$ on that subspace. The use of a uniform $C < 1$ is the key step. Saying only that the distance decreases cannot rule out convergence to a positive residual.

For underdetermined data, a null-space component is unchanged by every sample update. The proof therefore applies either after projecting onto the data span or after selecting w^* with the same null component as the initialization. It does not show convergence to an arbitrarily chosen interpolating parameter vector.

If interpolation fails, write a sample gradient at an aggregate minimizer as $\xi_i = \nabla \ell_i(w^*)$. Although $\mathbb{E}\xi_i = 0$, individual ξ_i need not vanish. Expanding the update then introduces additive noise terms. The interpolating proof cannot simply be reused by replacing an exact per-example equation with an average normal equation.

C.3 What increasing the batch size changes

Return to lecture: Lecture 2, §2.2

Let independent sample gradients have mean g and covariance Σ. For a batch mean $\hat{g} = B^{-1}\sum_{i=1}^{B} g_i$,

$$\mathbb{E}\hat{g} = g, \qquad \mathrm{Cov}(\hat{g}) = \Sigma/B.$$

The cross terms vanish because independent centered gradients have zero cross covariance. For a dataset of size $N > 1$, define $\Sigma = N^{-1}\sum_i (g_i - g)(g_i - g)^\top$. A uniform size-$B$ batch sampled without replacement has covariance $\frac{N-B}{B(N-1)}\Sigma$. It vanishes at $B = N$, whereas sampling with replacement still introduces randomness. Correlated sampling requires its own cross-covariance terms.

A larger batch reduces gradient noise per update, but it also uses more examples per update. Comparing methods after the same number of updates and after the same number of processed examples answers different questions. In a quadratic, raising the learning rate can initially compensate for fewer updates, but the curvature stability bound remains. A linear batch-size learning-rate rule cannot hold without limit.

Gradient accumulation reproduces a larger-batch mean only if the microbatch losses are weighted correctly and the computation behaves independently across microbatches. Batch normalization can make the forward computation differ between one large batch and several small ones. Stochastic layers and optimizer timing also need to be held consistent.

C.4 Momentum as a second-order recurrence

Return to lecture: Lecture 2, §2.3

With the heavy-ball convention $v_{t+1} = \beta v_t + g_t$ and $w_{t+1} = w_t - \eta v_{t+1}$, eliminate v_t to obtain

$$e_{t+1} = (1 + \beta - \eta\lambda)e_t - \beta e_{t-1}$$

for a quadratic eigenmode of curvature λ. The characteristic roots solve

$$r^2 - (1 + \beta - \eta\lambda)r + \beta = 0.$$

When the roots are complex, their product is β and their magnitude is $\sqrt{\beta}$. The trajectory oscillates with an envelope determined by this magnitude. Real roots instead give a combination of two exponential modes. Momentum therefore changes the order and damping of the dynamics, not merely the size of a gradient step.

For the exponential-average convention $m_{t+1} = \beta m_t + (1-\beta)g_t$, the effective heavy-ball gradient coefficient is $\eta(1-\beta)$. Copying a learning rate between these conventions without adjustment changes the recurrence.

For a positive-definite quadratic with known extremes μ, L_s, the classical heavy-ball parameter choice

$$\eta = \frac{4}{(\sqrt{L_s} + \sqrt{\mu})^2}, \qquad \beta = \left(\frac{\sqrt{L_s} - \sqrt{\mu}}{\sqrt{L_s} + \sqrt{\mu}}\right)^2$$

illustrates how momentum can improve the spectral contraction scale. These quadratic parameters do not establish a universal safe setting for nonquadratic stochastic training.

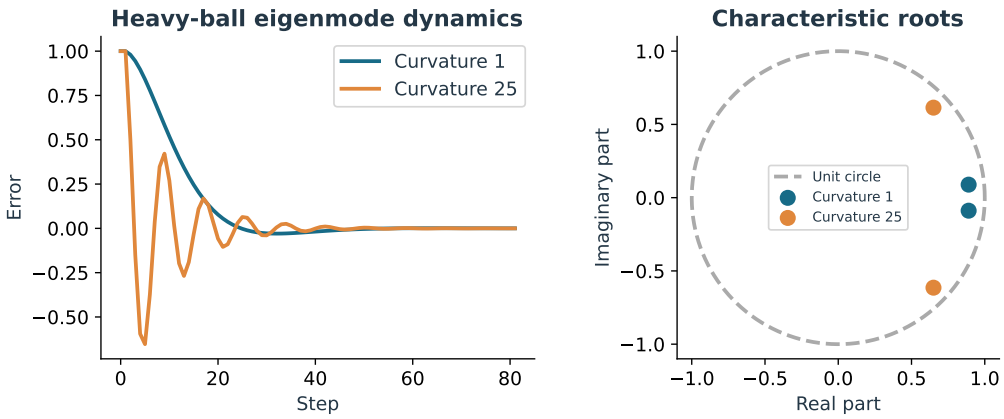


Figure C.2: Heavy-ball dynamics have two characteristic roots per eigenmode. Stable complex roots produce decaying oscillation.

C.5 Adam's denominator is an uncentered second moment

Return to lecture: Lecture 3, §3.2

Adam tracks $v_t \approx \mathbb{E}[g_t^2]$ coordinatewise, not the variance $\mathbb{E}[g_t^2] - (\mathbb{E}[g_t])^2$. A coordinate with a large persistent mean gradient can therefore have a large denominator even if its gradient barely fluctuates. Calling v_t a variance obscures what the update normalizes.

For a constant scalar gradient g, starting moments at zero gives $m_t = (1-\beta_1^t)g$ and $v_t = (1-\beta_2^t)g^2$. Bias correction recovers g and g^2, so the initial normalized direction is approximately $\text{sign}(g)$ when $|g| \gg \epsilon$. This does not mean every later Adam step is SignGD: the two moments average different histories.

Positive gradient rescaling approximately cancels between Adam's moments when ϵ is negligible. It does not confer invariance to arbitrary parameterizations or remove learning-rate sensitivity. Coordinatewise and matrix-aware geometries remain distinct.

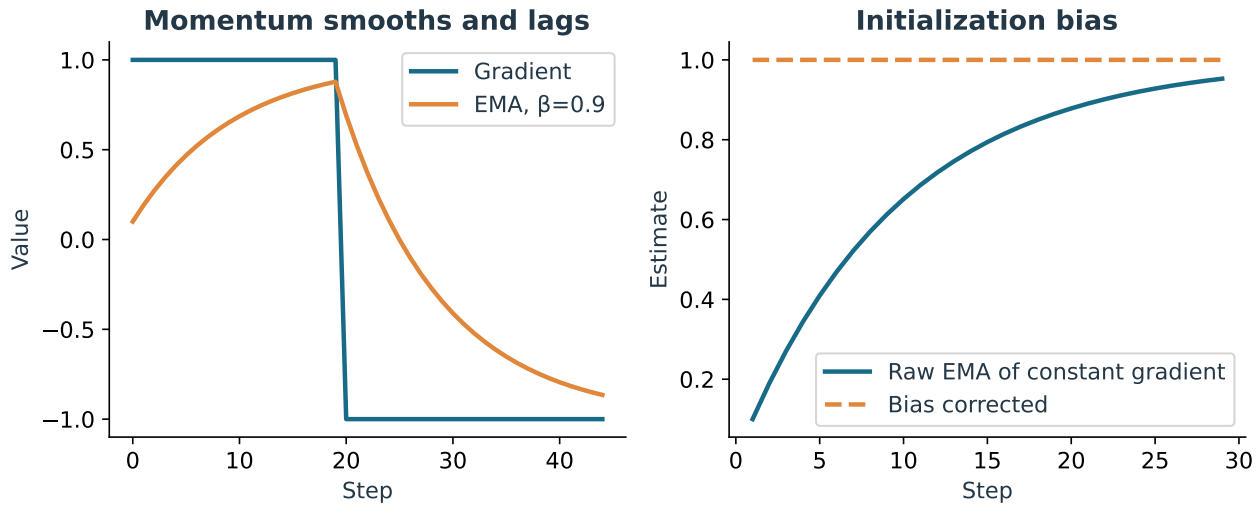


Figure C.3: An exponential moving average smooths a noisy gradient but lags when the underlying direction changes. Bias correction removes startup attenuation for a constant signal.

Why coupled regularization and decoupled decay differ. With a fixed positive diagonal preconditioner P, putting a penalty $\lambda\|w\|^2/2$ in the objective produces a step $-\eta P(g+\lambda w)$. Decoupled decay produces $-\eta P g - \eta\lambda w$. Their difference is $-\eta\lambda(P-I)w$. For $P = \mathrm{diag}(1, 10)$ and $w = (1, 1)$, the penalty-induced shrinkage differs tenfold between coordinates, while decoupled shrinkage is equal. In Adam, including the penalty gradient also changes future moments, so a fixed-P calculation only isolates one source of inequivalence.

Supplement D

Optimizers as choices of geometry

Source trail: lecture 4, lecture 5; optimization draft §10; discussion 3. The distinction between a constrained linear step and a penalized linear step is essential.

D.1 Shampoo: connect the algebra to an adaptive algorithm

Return to lecture: Lecture 5, §5.4

On the nonzero singular subspace,

$$(GG^\top)^{-1/4}G(G^\top G)^{-1/4} = U_r V_r^\top.$$

To verify this, insert the SVD. The left factor contributes $\Sigma_r^{-1/2}$, the gradient contributes Σ_r, and the right factor contributes $\Sigma_r^{-1/2}$, leaving identity on the nonzero singular values. Rectangular or rank-deficient matrices require pseudoinverse powers on the appropriate subspaces; ordinary inverse powers may not exist.

A practical Shampoo-style method instead accumulates statistics, for example

$$L_t = \epsilon I + \sum_{s \le t} G_s G_s^\top, \qquad R_t = \epsilon I + \sum_{s \le t} G_s^\top G_s,$$

and preconditions with $L_t^{-1/4} G_t R_t^{-1/4}$. Moving-average variants and cached root computations are also possible. Accumulating past gradients means this is not generally the exact polar factor of the current gradient. The single-gradient identity explains the geometry, while the accumulated algorithm introduces memory and numerical regularization.

There is a computational tradeoff. A matrix layer stores two structured statistics rather than a full covariance of its flattened parameter vector, but matrix roots can still be expensive. Blocking, infrequent root updates, and damping are implementation choices that change both cost and behavior.

D.2 Preconditioning is a change in the local ruler

Return to lecture: Lecture 4, §4.2, Lecture 5, §5.3, Lecture 5, §5.2, Lecture 5, §5.1, Lecture 6, §6.2

Let M be symmetric positive definite. Define the squared update norm $\|u\|_M^2 = u^\top M u$. Minimizing a linearized loss plus this penalty gives

$$\min_u \; g^\top u + \frac{1}{2\eta} u^\top M u, \qquad g + \eta^{-1} M u = 0, \qquad u = -\eta M^{-1} g.$$

Two-sided matrix geometry. For a layer gradient $G \in \mathbb{R}^{m \times n}$, a two-sided preconditioned step has the form $-\eta LGR$, where L is $m \times m$ and R is $n \times n$. These symbols denote the actual preconditioning factors, such as appropriately regularized inverse roots, rather than the raw accumulated statistics. Left multiplication acts on output-feature coordinates; right multiplication acts on input-feature coordinates. Flattening the matrix hides this useful separation.

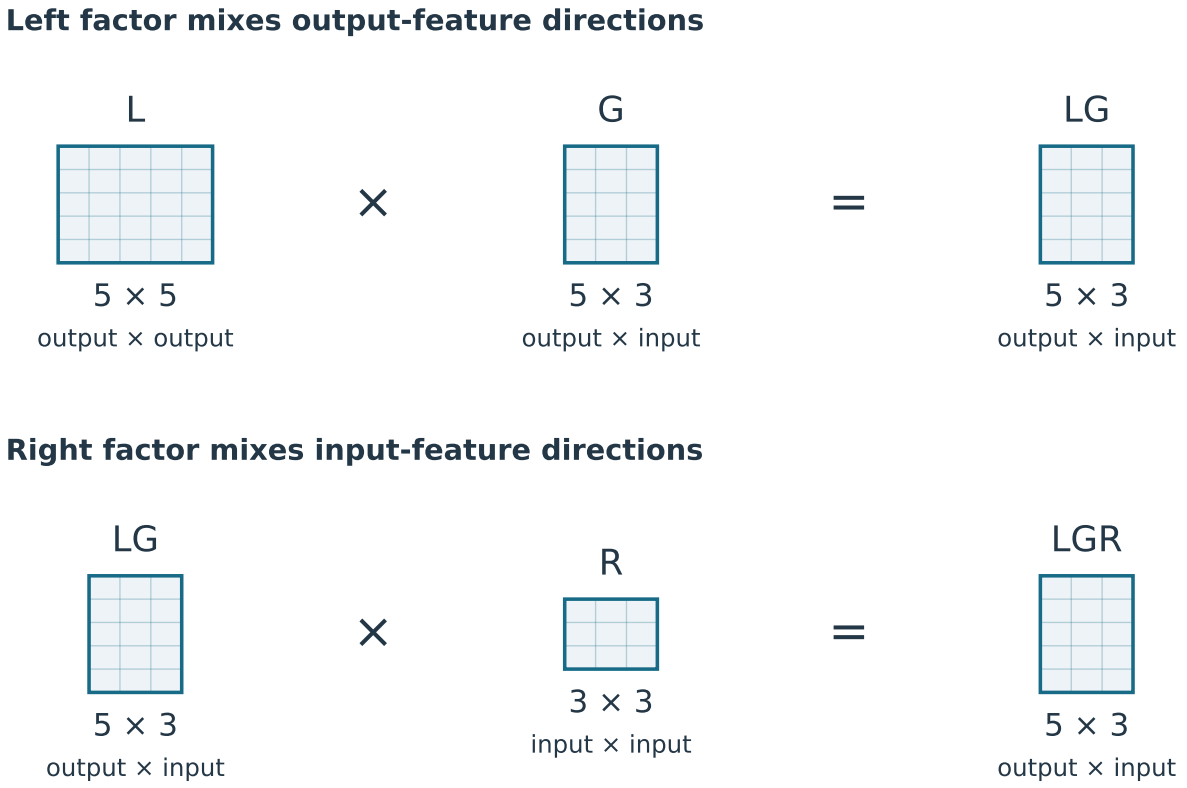


Figure D.1: A matrix preconditioner can treat the two feature spaces separately. The gradient and final update remain $m \times n$.

Directions expensive under M are changed less. If $M = H$ for a positive-definite quadratic Hessian, one step with $\eta = 1$ reaches the minimizer. For a nonconvex loss, an indefinite Hessian does not define a norm; damping or another positive-definite approximation is needed.

A coordinate change makes this geometry concrete. If $\theta = Az$ for an invertible matrix A, then $\nabla_z L = A^\top g$. Ordinary gradient descent in z produces $\Delta\theta = -\eta A A^\top g$. Thus Euclidean

descent in transformed coordinates corresponds to preconditioned descent in the original coordinates. The raw gradient is not independent of the parameterization.

For $H = \mathrm{diag}(1, 100)$, ordinary descent must respect the high-curvature direction, slowing progress in the first coordinate. Choosing $M = H$ makes the effective Hessian in the transformed geometry the identity. A diagonal approximation can remove axis-aligned scale differences but cannot generally remove rotated correlations. A full matrix approximation is more expressive and more expensive.

D.3 Standardizing inputs and preserving the prediction problem

Return to lecture: Lecture 4, §4.3

For a scalar feature x_j, define $\tilde{x}_j = (x_j - \mu_j)/s_j$ with $s_j > 0$. A linear predictor can represent the same function after adjusting its weights and intercept, but its optimization coordinates change. Features measured in dollars and fractions can otherwise produce gradients with very different magnitudes simply because of units.

Centering changes which role the bias must play. Scaling changes the curvature of the loss. Whitening additionally removes correlations under an estimated covariance matrix, but requires a more expensive and potentially unstable inverse square root. Standardization does not guarantee a well-conditioned deep network because later nonlinear layers create new correlations.

A ReLU example. Suppose a hidden preactivation is $z = w_1 x_1 + w_2 x_2 + b$ and x_1 has magnitude around 10^3 while x_2 has magnitude around 1. Equal-sized initial weights can make the first term dominate the activation mask. A small change in w_1 can then have a much larger functional effect than the same change in w_2. Standardizing inputs helps make the initialization's intended scale assumptions plausible.

Near-constant features require care: dividing by a tiny empirical standard deviation can amplify noise. Statistics must be fitted on training data, and the same transformation must be reused at validation and inference. Recomputing statistics independently on a test batch changes the model's input map.

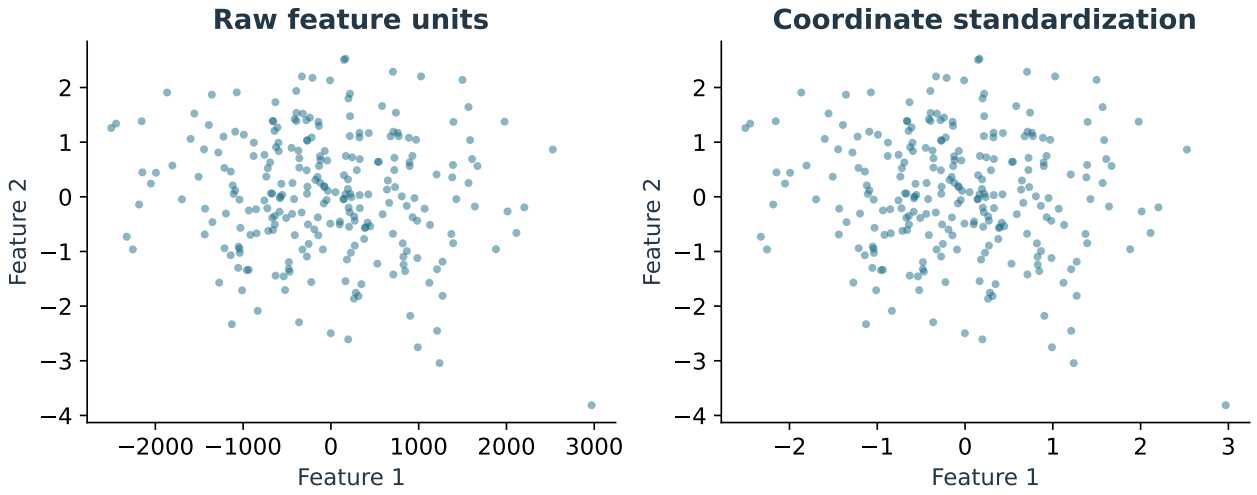


Figure D.2: Coordinate standardization changes feature units. It does not generally remove correlations or solve all downstream conditioning problems.

D.4 Nonuniqueness of the spectral-norm optimal step

Return to lecture: Lecture 5, §5.4

Take $G = \mathrm{diag}(2, 0)$ and radius $\rho = 1$. The lecture's compact polar choice is $D_0 = \mathrm{diag}(-1, 0)$, with objective $\langle G, D_0 \rangle_F = -2$. But every $D_a = \mathrm{diag}(-1, a)$ with $|a| \le 1$ is feasible and attains the same value. The linearized objective is indifferent to changes in a null direction of G.

This does not make the choices equivalent for the nonlinear loss. For example, at the origin let $L(u_1, u_2) = 2u_1 + \frac{1}{2}cu_2^2$ with $c > 0$. Its gradient is $(2, 0)$, yet $L(-\eta, \eta a) = -2\eta + \frac{1}{2}c\eta^2 a^2$. The null-direction update adds a second-order cost invisible to the linear model. Setting it to zero is a meaningful convention, even though first-order optimality alone does not require it.

Supplement E

Initialization, maximal updates, and Muon

Source trail: lecture 6, lecture 7, parts of lectures 4–5; optimization draft §§11–15; discussion 4. The derivations here concern the course's width-scaling model, not a claim that one formula covers every practical Muon implementation.

E.1 Why identical hidden units do not diversify by themselves

Return to lecture: Lecture 4, §4.4

Consider two hidden units with identical incoming weights and biases, identical activation functions, and identical outgoing weights. They produce the same activations on every example. Their backward sensitivities are also identical, so a deterministic gradient update keeps their parameters identical. By induction, the two units remain copies throughout training.

This is a symmetry argument. Initializing all hidden units identically wastes the width because the optimizer has no information that distinguishes their roles. Random initialization breaks the symmetry. It need not inject a large signal; it only needs to give units distinct starting directions while preserving a sensible activation scale.

Zero initialization has several different failure modes. In a network with zero weights on every layer, early-layer gradients can vanish because a downstream zero matrix blocks them. With ReLU and zero biases, preactivations sit at a nondifferentiable point where implementations commonly choose derivative zero. In a single linear regression layer, however, zero initialization is perfectly reasonable: its gradient $X^\top(0 - y)$ need not vanish. The rule is about the computation graph, not the number zero in isolation.

The same reasoning explains LoRA's initialization. Two zero factors in a product block each other's gradients. One random factor and one zero factor preserve the original function while allowing one factor to learn immediately. This is a controlled way to break symmetry without perturbing the base model's initial predictions.

E.2 Forward and backward second-moment calculations

Return to lecture: Lecture 4, §4.4

Let a layer have independent zero-mean weights with variance v and inputs with common second moment q. Under the usual independence assumptions,

$$\mathbb{E}[z_i^2] = d_{\text{in}} v q.$$

For a symmetric preactivation distribution and ReLU,

$$\mathbb{E}[\mathrm{ReLU}(z_i)^2] = \frac{1}{2}\mathbb{E}[z_i^2].$$

Preserving the activation second moment therefore suggests $v = 2/d_{\text{in}}$. The output's mean is positive, so its variance is smaller than its second moment. For a zero-mean Gaussian z with variance s^2, the ReLU mean is $s/\sqrt{2\pi}$ and variance is $s^2(1/2 - 1/(2\pi))$.

For a leaky ReLU with negative slope a, symmetry gives

$$\mathbb{E}[\phi(z)^2] = \frac{1+a^2}{2}\mathbb{E}[z^2],$$

so the analogous second-moment-preserving variance is $2/[(1+a^2)d_{\text{in}}]$. The ordinary ReLU factor should not be copied unchanged to every activation variant.

Backward propagation through a linear layer multiplies by $W^\top$. Under a similar independence approximation, the gradient second moment is multiplied by $d_{\text{out}}v$. A single variance cannot exactly preserve both forward and backward moments when fan-in and fan-out differ. Xavier-style balancing uses both dimensions; He-style fan-in or fan-out choices emphasize one direction. These calculations identify assumptions and tradeoffs, rather than proving that a deep trained network remains statistically independent.

E.3 Depth creates an additional scale problem

Return to lecture: Lecture 6, §6.5, Lecture 6, §6.4

A width-preserving initialization does not automatically preserve scale over arbitrary depth. If every layer multiplies a second moment by c, then L layers multiply it by c^L. Even a small systematic mismatch compounds. For example, $1.02^{100} \approx 7.24$ and $0.98^{100} \approx 0.133$.

In a residual stack $h_{\ell+1} = h_\ell + \alpha F_\ell(h_\ell)$, a simplified uncorrelated-branch model gives

$$\mathbb{E}\|h_{\ell+1}\|^2 \approx \mathbb{E}\|h_\ell\|^2 + \alpha^2\mathbb{E}\|F_\ell(h_\ell)\|^2.$$

If branch scales stay comparable, choosing α of order $1/\sqrt{L}$ can control accumulated variance in this model. Correlated branches can have cross terms and different scaling. Normalization and learned residual scales alter the analysis further. The point is that width and depth are distinct asymptotic axes.

A practical signal audit tracks activation RMS, gradient RMS, and update-to-weight ratios by layer. A single global gradient norm can hide one exploding layer and many nearly inactive ones. Compare these statistics at initialization and after several updates, because training changes the correlations underlying the initialization argument.

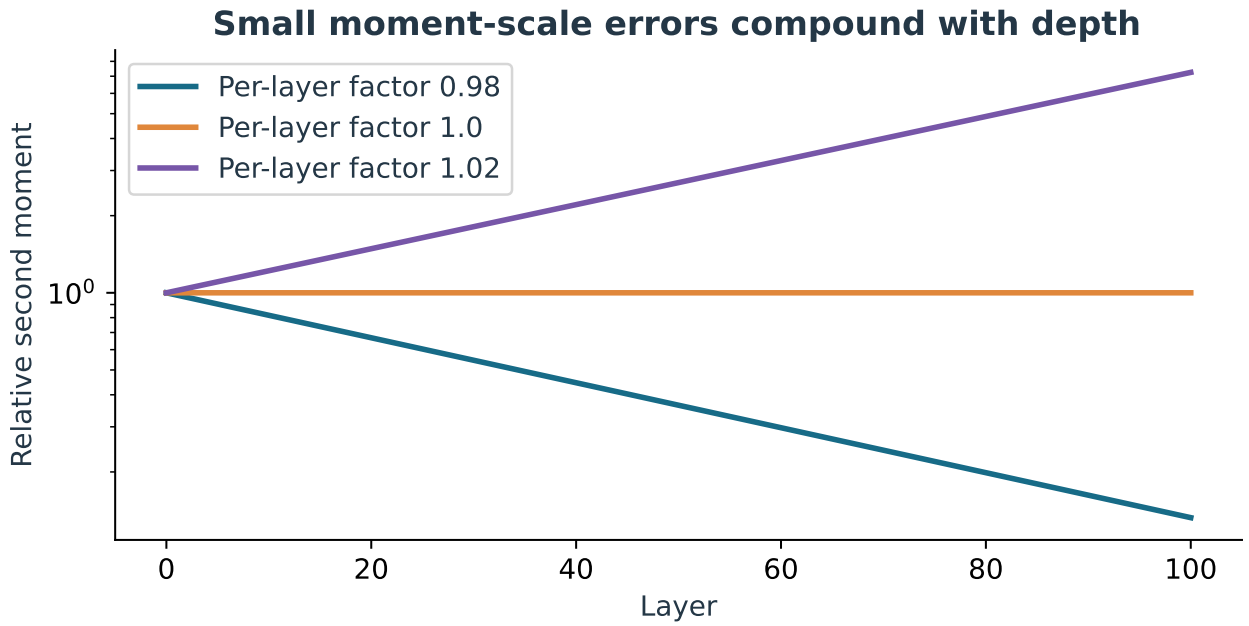


Figure E.1: A small per-layer moment mismatch compounds exponentially with depth, motivating explicit checks of signal scale.

E.4 Feature-update scale versus parameter-update scale

Return to lecture: Lecture 5, §5.5, Lecture 6, §6.3, Lecture 6, §6.1, Lecture 7, §7.1

For a rank-one gradient $G = gx^\top$, ordinary gradient descent produces

$$\Delta W\, x = -\eta g \|x\|_2^2.$$

If x has order-one RMS, then $\|x\|_2^2$ is of order d_{in}. A fixed coordinate learning rate can therefore produce a feature change that grows with width under this simplified model. The appropriate scaling also depends on how g scales, which is set by downstream parameterization and the loss.

For SignGD, the exact action is

$$\Delta W\, x = -\eta\, \text{sign}(g) \|x\|_1$$

when the gradient factorization and nonzero-coordinate assumptions hold. With typical order-one absolute input coordinates, $\|x\|_1$ also grows with fan-in. This gives an input-aligned explanation of the inverse-fan-in rate, complementing the worst-case induced-norm bound.

The output layer is special because its job is to aggregate many hidden features into a prediction of fixed dimension. Embeddings are special because their input is sparse and categorical. Biases are special because they add directly without a fan-in sum. Width transfer requires treating these roles explicitly; one scalar rule applied to every tensor does not constitute a full maximal-update parameterization.

E.5 How many random ReLU breakpoints fall in the useful input range?

Return to lecture: Lecture 4, §4.4

The lecture identifies the breakpoint $K = -b/w$ as Cauchy with scale $s = \sigma_b/\sigma_w$ for independent centered Gaussian bias and weight. For an input interval $[-a, a]$, $a > 0$, the useful-range probability is

$$p = \Pr(|K| \le a) = \frac{2}{\pi} \arctan(a/s).$$

With m independently initialized units, the number N of breakpoints in that interval is binomial: $\mathbb{E}[N] = mp$, $\mathrm{Var}(N) = mp(1-p)$, and $\Pr(N = 0) = (1-p)^m$. For $a = s$, $p = 1/2$; ten units give probability 2^{-10} of no breakpoint in range. More generally, making that probability at most δ requires $m \ge \log \delta / \log(1-p)$, rounded up.

This criterion guarantees only at least one initial kink, not a good approximation or successful training. It also exposes a bias-scale tradeoff: small s puts more kinks inside a narrow input range, but overly concentrated kinks can leave other relevant regions poorly covered. The Cauchy mean and variance do not exist; interval probabilities remain well-defined and are the useful quantities here.

E.6 Newton–Schulz convergence on nonzero singular modes

Return to lecture: Lecture 7, §7.2

For the cubic iteration from the lecture, a singular value evolves as $s^+ = \frac{3}{2}s - \frac{1}{2}s^3$. If $0 < s \le 1$, write $e = 1 - s$. Direct factorization yields

$$e^+ = 1 - s^+ = \tfrac{1}{2}(1-s)^2(s+2) = \tfrac{1}{2}e^2(s+2).$$

Hence $0 \le e^+ < e$ when $0 < e < 1$, and $e^+ \approx \frac{3}{2}e^2$ near the limit. A tiny positive singular value initially grows by roughly $3/2$ per iteration; reaching the near-one regime can therefore require many steps for an ill-conditioned matrix. Zero singular values remain exactly zero.

For a rectangular rank-r matrix, the limiting compact polar factor is $U_r V_r^\top$. Its Gram matrices are projectors $V_r V_r^\top$ and $U_r U_r^\top$, not both full identities. The spectral error to the compact polar factor is $\max_{1 \le i \le r} |1 - s_i^{(k)}|$ when the singular vectors are preserved. Testing the wrong full-identity residual incorrectly reports failure for rank-deficient inputs. These statements apply to this cubic iteration and its stated initialization range; other polynomial coefficients require their own stability analysis.

Why rectangular orthogonalization uses Gram products. For $Z \in \mathbb{R}^{m \times n}$, a cubic singular-value iteration uses $ZZ^\top Z$, which is well-defined even when $m \ne n$. The expression Z^3 is generally undefined for a rectangular matrix. Associativity permits either $(ZZ^\top)Z$ or $Z(Z^\top Z)$; forming the smaller Gram matrix can reduce work and storage. In both cases the final update stays in the original layer's parameter space.

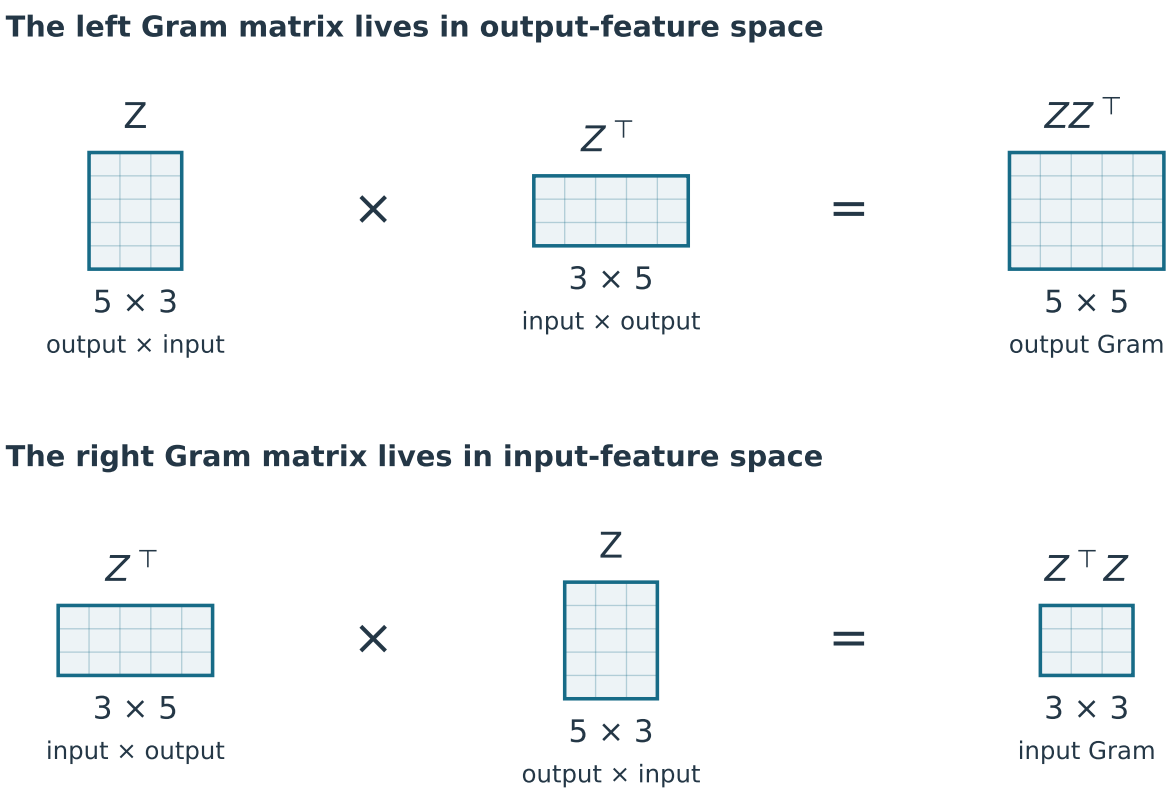


Figure E.2: The two Gram matrices have different shapes but share the same nonzero eigenvalues. Their cubic actions on Z agree.

Supplement F

Convolution, locality, and weight sharing

Source trail: lecture 8, lecture 9; discussion 5. The course develops CNNs from structural assumptions about images rather than treating convolution as a mysterious primitive.

F.1 Equivariance is not invariance

Return to lecture: Lecture 8, §8.3, Lecture 8, §8.2

A shift should move a feature map before pooling removes location. For a circular sequence, let $(Tx)_i = x_{i-2}$ and $F(x)_i = x_i + 2x_{i-1} + x_{i-2}$. Then $F(Tx) = TF(x)$ exactly: this is equivariance. The feature locations change, so $F(Tx)$ need not equal $F(x)$. A global sum $g(x) = \sum_i F(x)_i$ is invariant to the circular shift because it sums the same values in a different order. Ordinary padding, cropping, and striding can alter these equalities at boundaries or for particular shifts.

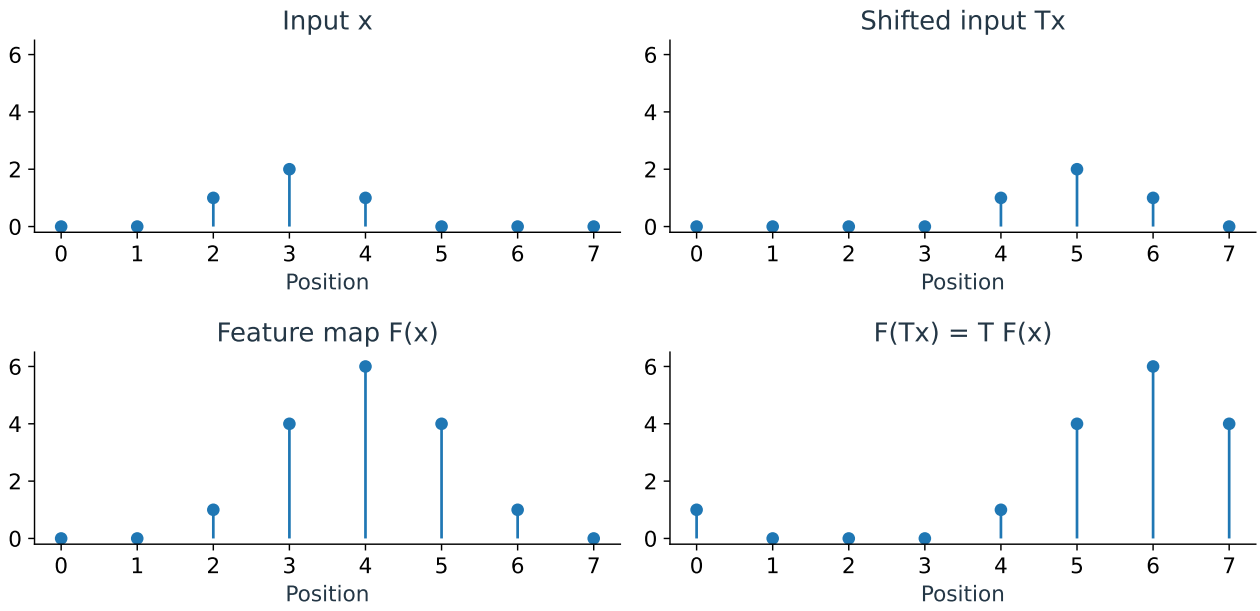


Figure F.1: Equivariance means that transforming the input transforms the output correspondingly. This example uses circular boundaries; global pooling can turn the shifted feature map into an invariant summary.

Let T_a translate an input by a positions and let C denote stride-one convolution on an infinite grid. Shared weights imply

$$C(T_a x) = T_a(Cx).$$

This is equivariance: the output moves with the input. Invariance would mean $C(T_a x) = Cx$, with no output change. Image classification may seek approximate translation invariance, while segmentation should preserve spatial correspondence and is naturally equivariant.

Finite padding introduces boundaries, and striding changes which shifts preserve exact equivariance. A stride-s layer generally respects shifts aligned to that stride, with compatible boundary conventions. A one-pixel shift can change which samples are retained. Data augmentation can encourage robustness beyond exact architectural symmetries, but does not turn an approximate property into a theorem.

F.2 Backpropagation through shared weights

Return to lecture: Lecture 8, §8.4, Lecture 9, §9.2

For a one-channel valid cross-correlation $y_t = \sum_r w_r x_{t+r}$ and downstream sensitivities $g_t = \partial L/\partial y_t$,

$$\frac{\partial L}{\partial w_r} = \sum_t g_t x_{t+r}.$$

Every use of w_r contributes to the same parameter gradient. Averaging or summing additionally over a batch follows the loss convention. Treating each occurrence as a separate parameter destroys weight sharing.

Worked example. Let $x = (1, 2, 3, 4)$, $w = (2, -1)$, and valid stride-one cross-correlation. Then $y = (0, 1, 2)$. If the downstream gradient is $(1, 2, 3)$, the kernel gradient is $(1+4+9, 2+6+12) = (14, 20)$. This explicit calculation is a useful check against automatic differentiation.

Discussion 5 also asks you to compare CNN and MLP behavior before training. Translation structure comes from the architecture, not solely from learned weights. This illustrates an inductive bias: the model already treats certain transformations in a particular way before seeing data.

F.3 Pooling, downsampling, and aliasing

Return to lecture: Lecture 9, §9.3

A stride-s convolution computes a filtered response and keeps every sth spatial location. Average pooling computes local means before subsampling. Max pooling returns the largest value in each region. All reduce spatial size, but they discard different information and have different derivatives.

For a pooling window with entries $x_1, \ldots, x_k$, the average is $k^{-1} \sum_i x_i$, so the incoming gradient is distributed equally. For a unique maximum at index j, max pooling sends the incoming

gradient only to x_j. Ties require a subgradient convention. A max-pooling layer has no learned weights, but it still participates in backpropagation.

Downsampling can alias high-frequency patterns. In one dimension, the alternating signal $(1, -1, 1, -1, ...)$ becomes a constant sequence if every second entry is retained. Shifting it by one position produces the opposite constant. A prefilter can suppress high frequencies before subsampling, although it also removes detail. This explains why exact one-pixel translation equivariance generally fails after striding.

Local pooling provides limited tolerance to shifts, not unrestricted invariance. Global average pooling of a spatially equivariant feature map can produce a translation-invariant summary under appropriate boundary conditions. That can suit classification but is unsuitable when the output must preserve exact object locations. Architecture should match whether the task asks *what is present* or *where it is*.

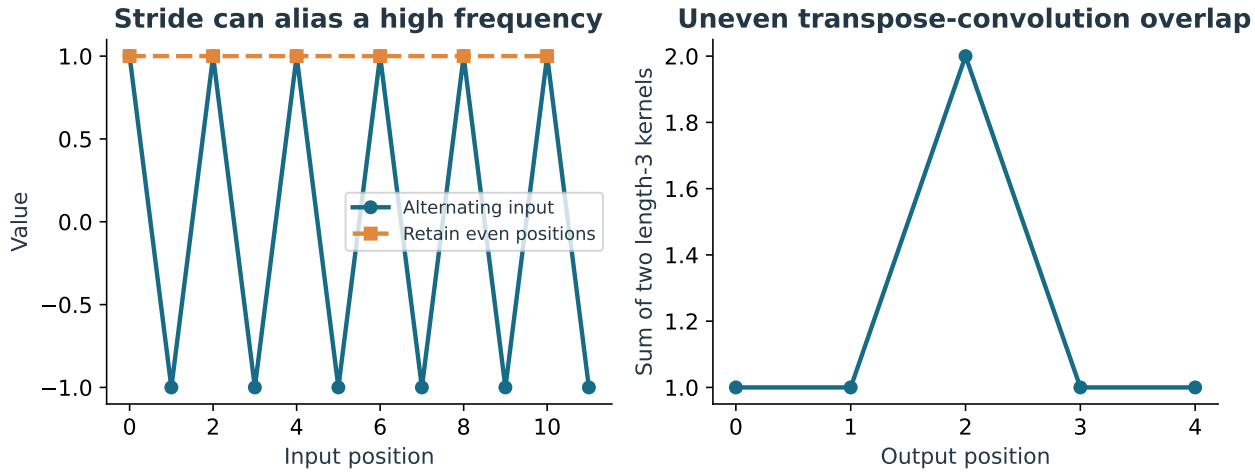


Figure F.2: Subsampling can turn an alternating input into a constant. A transposed convolution can create unequal overlap even from equal low-resolution inputs.

F.4 Data augmentation changes the training objective

Return to lecture: Lecture 9, §9.5

Let T be a random transformation sampled from a distribution of augmentations. Training on transformed inputs minimizes

$$\mathbb{E}_{(x,y),T}\, \ell(f_\theta(Tx), y)$$

when the label is unchanged. This teaches consistency across a chosen set of variations. It is justified only when those transformations preserve the relevant label semantics. A horizontal flip may be appropriate for an object category but can reverse a directional instruction. A crop may remove the evidence needed for the original label.

For structured outputs, transform the target too. A segmentation mask should undergo the same geometric crop or rotation as the image. Applying independent transformations to

input and target corrupts supervision. Photometric changes can leave the mask unchanged because they alter appearance rather than location.

Mixup forms $\tilde{x} = \lambda x_i + (1-\lambda)x_j$ and a corresponding soft target $\tilde{y} = \lambda y_i + (1-\lambda)y_j$. For cross-entropy, the soft-target loss is the same convex combination of the two hard-target cross-entropies at the mixed input. This imposes an interpolation preference; it is not a theorem that real-world labels are literally linear in pixel space. Cut-and-paste augmentations use a different spatial assumption and may weight labels by area as an approximation.

Test-time augmentation averages predictions over transformed views. It spends inference computation and changes the prediction rule. Compare it separately from training augmentation. If an augmentation destroys task information, averaging more such views does not recover the missing evidence automatically.

F.5 Standard, depthwise, pointwise, and grouped convolutions

Return to lecture: Lecture 8, §8.1, Lecture 9, §9.4

Follow which entries can communicate. In the top half, a feature in channel 1 cannot use channel 2, regardless of the spatial kernel's size. The pointwise map below mixes the channel vector but reads no adjacent pixel. With no intermediate nonlinearity, a depthwise-then-pointwise pair has effective weights $K_{u,v,c,o} = D_{u,v,c}P_{c,o}$: for a fixed input channel, every output reuses one spatial pattern scaled by a mixing coefficient. This explains both the saved parameters and the constraint relative to an arbitrary standard convolution.

View the corresponding lecture diagram (Figure 9.2, p. 114).

For a $k \times k$ kernel, standard convolution has $k^2 C_{\text{in}} C_{\text{out}}$ weights. A depthwise convolution uses a separate spatial kernel per input channel, giving $k^2 C_{\text{in}}$ weights for one output per channel. A following 1×1 pointwise convolution mixes channels with $C_{\text{in}} C_{\text{out}}$ weights.

The depthwise-separable total is

$$k^2 C_{\text{in}} + C_{\text{in}} C_{\text{out}},$$

which can be much smaller. For $k = 3$ and $C_{\text{in}} = C_{\text{out}} = 64$, the standard count is 36,864, whereas the separated count is $576 + 4096 = 4672$. The operations impose different parameter-sharing constraints, so this is a capacity/computation tradeoff, not an algebraically identical replacement for every standard convolution.

A grouped convolution partitions channels into groups. With g equal groups, the weight count is $k^2 C_{\text{in}} C_{\text{out}}/g$, assuming both channel counts are divisible by g. Depthwise convolution is an extreme grouping case; pointwise mixing can reconnect information across groups.

A 1×1 convolution is a shared linear map on the channel vector at each spatial position. It changes channel dimension without enlarging the spatial receptive field. A sequence of 1×1, spatial, and 1×1 layers can therefore separate channel compression, spatial processing, and channel expansion.

F.6 A complete shape-and-cost walkthrough

Return to lecture: Lecture 8, §8.5, Lecture 9, §9.1, Lecture 10, §10.1

Read a volume as spatial locations carrying feature vectors. The front face is a spatial grid, and depth counts channels; this differs from the normalization cubes, whose oblique axis is the batch. A standard filter has depth C_{in} and produces one scalar at a chosen location. Sliding the same weights generates one feature map; a bank of C_{out} filters generates the output volume. The batch consists of independent copies of this computation with shared weights.

View the corresponding lecture diagram (Figure 9.1, p. 113).

A convolution as patch-matrix multiplication. Unfold each receptive field into one row of $P \in \mathbb{R}^{N_{\text{pos}} \times K}$, with $K = k_h k_w C_{\text{in}}$. Flatten each output filter into one column of $F \in \mathbb{R}^{K \times C_{\text{out}}}$. Then $Y = PF$ contains one output row per spatial position. Biases broadcast across those rows. Patches overlap, so entries of P can refer to the same original pixel; unfolding is not a new set of independent trainable inputs. Efficient implementations may avoid explicitly materializing this large temporary matrix.

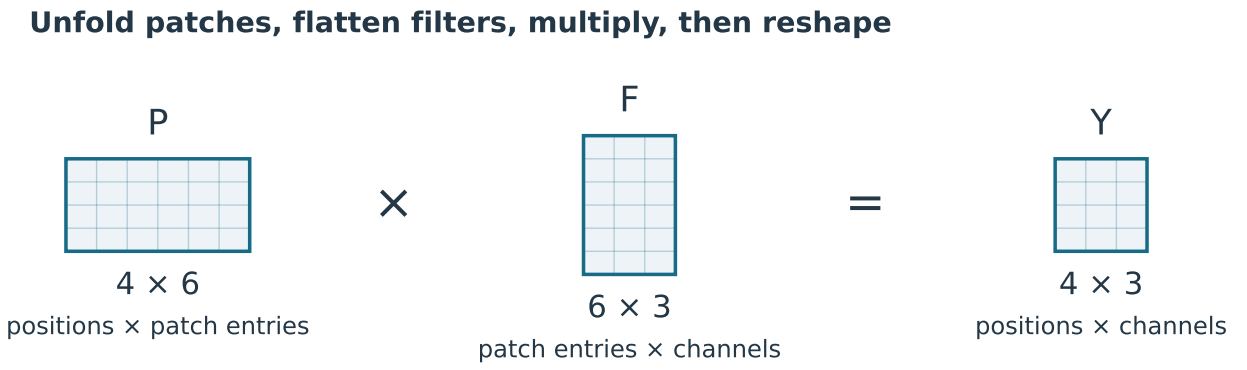


Figure F.3: Convolution contracts patch coordinates while preserving output positions and output channels. Reshaping the result restores the spatial grid.

Consider an input of shape $32 \times 32 \times 3$. A padded 3×3 stride-one convolution with 16 output channels preserves spatial size and has $3 \cdot 3 \cdot 3 \cdot 16 = 432$ weights. A second padded 3×3 stride-two convolution with 32 outputs produces $16 \times 16 \times 32$ and has 4608 weights. A global spatial mean produces a 32-vector, followed by a 10-class affine head with 320 weights and 10 biases.

Ignoring biases and counting one multiplication per kernel weight per output location, the first convolution uses $32^2 \cdot 432 = 442{,}368$ multiplications, while the second uses $16^2 \cdot 4608 = 1{,}179{,}648$. The layer with fewer spatial positions can still cost more because of its larger channel dimensions.

Using receptive-field jump $j_0 = 1$ and size $r_0 = 1$, the first layer gives $r_1 = 3, j_1 = 1$, and the second gives $r_2 = 5, j_2 = 2$. Each final local feature sees a 5×5 input region before global pooling. The global mean then aggregates these local features across the image; it does not retroactively make each intermediate feature globally contextual.

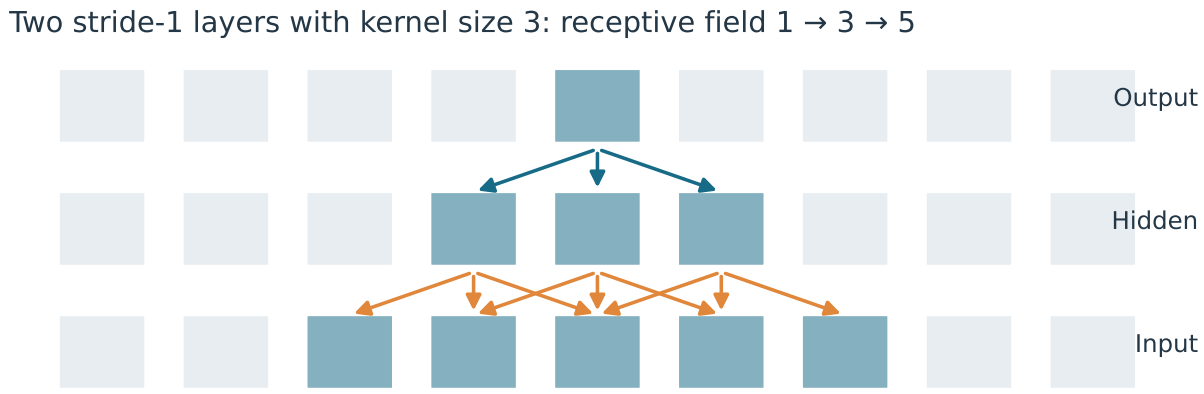


Figure F.4: Receptive fields grow through stacked local layers. Downsampling increases the spacing represented by each later kernel offset.

Supplement G

Normalization, residual paths, and spatial reconstruction

Source trail: lecture 10, lecture 11, the stochastic-depth addition in lecture 12; discussions 6 and 8.

G.1 When an identity skip still has a singular Jacobian

Return to lecture: Lecture 10, §10.3, Lecture 11, §11.2, Lecture 11, §11.1

The lecture gives $J_y = I + J_F$. This does not bound singular values away from zero. For a linear residual branch with

$$J_F = \begin{bmatrix} -1 & 0 \\ 0 & \varepsilon \end{bmatrix}, \qquad J_y = \begin{bmatrix} 0 & 0 \\ 0 & 1+\varepsilon \end{bmatrix},$$

the first input direction is completely erased despite the identity skip. A sufficient local condition is $\|J_F\|_2 \le \rho < 1$: the triangle inequality gives

$$(1-\rho)\|v\|_2 \le \|(I + J_F)v\|_2 \le (1+\rho)\|v\|_2.$$

Thus the block condition number is at most $(1+\rho)/(1-\rho)$. Across L blocks these bounds multiply, so a fixed positive ρ still permits exponential deterioration with depth. Residual scaling controls the size of each perturbation; the skip alone is not a conditioning theorem.

G.2 Coordinate dropout versus branch dropout: the noise covariance

Return to lecture: Lecture 12, §12.2

Fix a branch output $f = F(x)$ and a keep probability $p \in (0, 1]$. An inverted stochastic-depth gate gives $y = x + \xi f/p$, with $\xi \sim \text{Bernoulli}(p)$. Conditional on x,

$$\mathbb{E}[y \mid x] = x + f, \qquad \text{Cov}(y \mid x) = \frac{1-p}{p} f f^\top.$$

All branch coordinates share one random gate, so their fluctuations are correlated. Independent coordinate gates instead give covariance $(1-p)\,\text{diag}(f_1^2, \dots, f_d^2)/p$. For $f = (1, 2)^\top$ and $p = 1/2$, these are respectively

$$\begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 1 & 0 \\ 0 & 4 \end{bmatrix}.$$

The per-coordinate variances agree, but the noise direction is different: branch dropout varies only along f. A subsequent nonlinear layer need not preserve the equality of mean activations and deterministic evaluation output. Likewise, a dropped branch receives no parameter gradient on that example, while the identity route still carries a gradient. These facts explain why matching the keep probability does not make the two regularizers interchangeable.

G.3 Deriving the layer-normalization backward map

Return to lecture: Lecture 9, §9.6

Normalization has an axis. For row-token data $H \in \mathbb{R}^{T\times d}$, right multiplication by $C = I_d - \mathbf{1}\mathbf{1}^\top/d$ subtracts each token's feature mean: HC. Left multiplication by $I_T - \mathbf{1}\mathbf{1}^\top/T$ would instead subtract a mean over tokens, which is a different operation. LayerNorm then divides each row by its own data-dependent standard deviation and applies learned feature scales and offsets. Only the centering step is the fixed linear map pictured here; the full normalization is nonlinear.

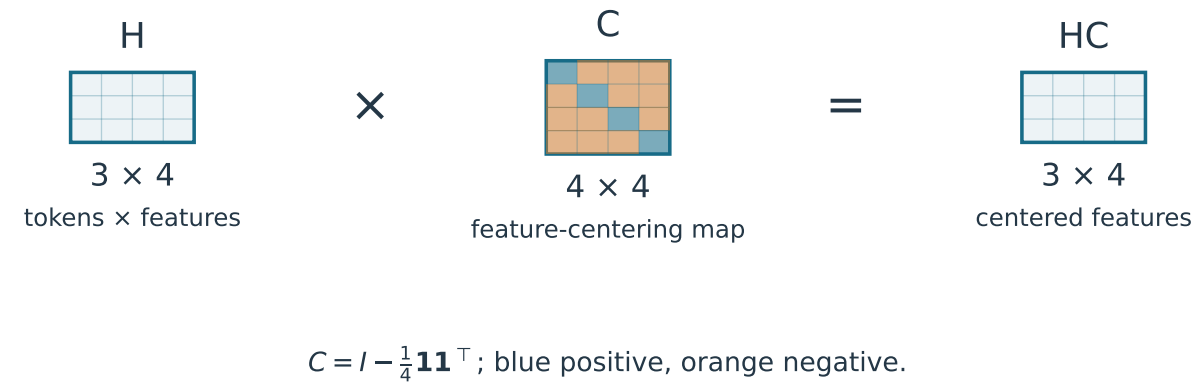


Figure G.1: LayerNorm's centering contracts the feature axis, leaving tokens separate. The subsequent variance normalization cannot be represented by one fixed matrix for every input.

For a d-vector x, define $\mu = d^{-1}\sum_i x_i$, $c = x - \mu\mathbf{1}$, $v = d^{-1}\sum_i c_i^2$, $r = \sqrt{v+\epsilon}$, and $\hat{x} = c/r$. The affine output is $y = \gamma \odot \hat{x} + \beta$. Let $g = \nabla_y L$ and $u = g \odot \gamma$. Then

$$\nabla_\beta L = g, \qquad \nabla_\gamma L = g \odot \hat{x},$$

with sums over any examples that share γ, β. For the input derivative,

$$\nabla_x L = \frac{1}{r}\left[u - \overline{u}\mathbf{1} - \hat{x}\,\overline{u \odot \hat{x}}\right],$$

where the bars denote coordinate means. This formula remains valid with the stated ϵ convention.

The first subtraction accounts for centering: adding a common constant to every input coordinate should not change the centered representation. The second accounts for the dependence of the denominator on the input. Treating the mean and variance as constants during training would omit both effects and compute a different gradient.

With $\epsilon = 0$, the derivative has no component along the all-ones direction and no component along the centered input direction. Layer normalization removes a common shift and a positive scale locally. With $\epsilon > 0$, exact scale invariance is softened. Learned affine parameters restore flexible output scales and offsets, but do not undo the example-dependent coupling introduced by normalization.

RMS normalization omits centering. With $r = \sqrt{d^{-1} \sum_i x_i^2 + \epsilon}$ and $\hat{x} = x/r$,

$$\nabla_x L = \frac{1}{r} \left[u - \hat{x}\, \overline{u \odot \hat{x}} \right].$$

The missing mean-subtraction term reflects a real functional difference. A constant shift in the input affects RMS normalization, whereas ideal layer normalization removes it.

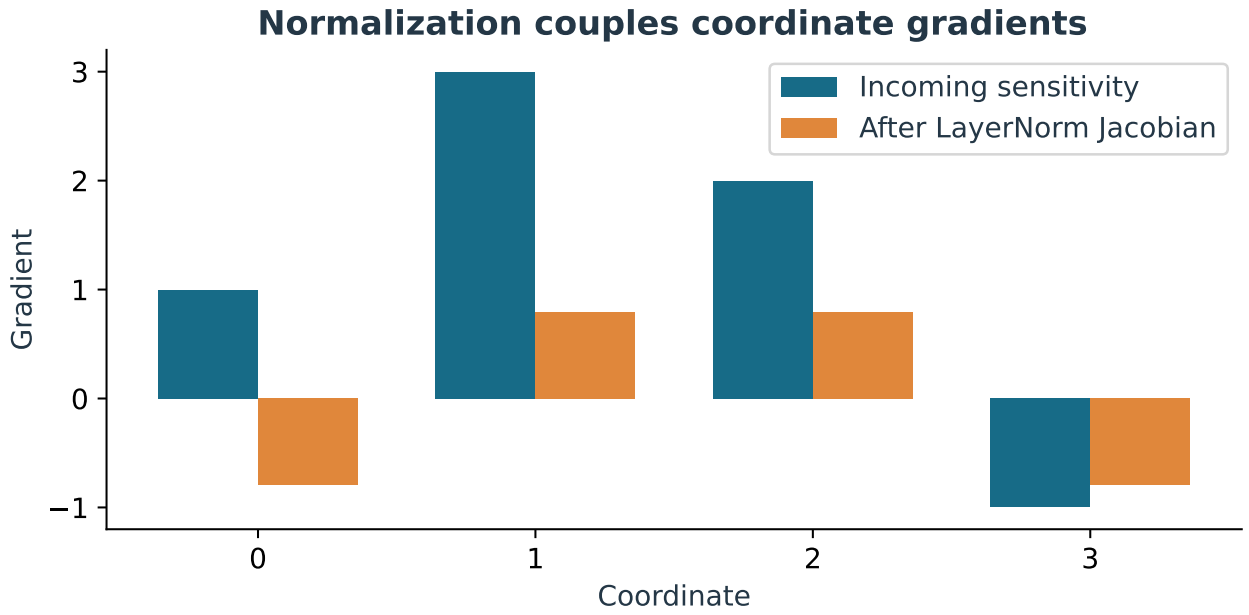


Figure G.2: The LayerNorm backward map removes sensitivity to a common shift and adjusts for the input-dependent scale.

G.4 Batch normalization has two different operating modes

Return to lecture: Lecture 10, §10.2

Change the axes before changing the formula. In the lecture's image cube, write x_{bcs} with s indexing HW positions. Training BatchNorm fixes c and reduces over (b, s); the illustrated LayerNorm fixes b and reduces over (c, s). Their set sizes are BHW and CHW.

RMSNorm can use a specified feature set too, but computes $\sqrt{\mathbb{E}[x^2] + \epsilon}$ without subtracting its mean. The cube alone specifies the set, not the formula.

For a transformer, write x_{btd}. The usual token-wise LN and RMSNorm both fix (b, t) and reduce only over d. Likewise a channel-only image LN fixes (b, h, w) and reduces over c; this differs from the full-example LN cube in the lecture. Always state the axes explicitly. Learned affine parameters do not change which entries share a statistic. A normalization named "LayerNorm" may have different affine-parameter shapes under different implementations.

View the corresponding lecture diagram (Figure 10.5, p. 131).

During training, BatchNorm computes statistics from a set of examples and, for a convolutional tensor, usually spatial positions within each channel. The normalized output of one example therefore depends on other examples in that normalization group. Its derivative couples those examples through the mean and variance.

During inference, a common implementation uses stored running estimates instead. These are moving estimates accumulated during training, not the exact statistics of the current test batch. A distribution change can make the stored statistics inappropriate. Tiny training batches can also make estimates noisy, although the spatial dimension can supply additional values for a convolutional channel.

Dropout and BatchNorm interact because dropout changes the activation distribution used to estimate statistics. The location of dropout relative to normalization is therefore consequential. Likewise, gradient accumulation with several small microbatches does not reproduce one large BatchNorm group, even if the final parameter gradient is averaged over the same examples.

LayerNorm instead normalizes within each example/token, so its statistics do not require a training batch. This makes its behavior convenient for variable batch sizes and autoregressive inference. That is a statement about the normalization axes, not a claim that one normalization method is universally superior.

G.5 Residual bottlenecks and ConvNeXt-style blocks

Return to lecture: Lecture 11, §11.3

Use depth to see where computation is spent. In the upper block, the 3×3 convolution has $9 \cdot 64^2 = 36{,}864$ weights, compared with $9 \cdot 256^2 = 589{,}824$ at the surrounding width. Both 1×1 maps together add $2 \cdot 256 \cdot 64 = 32{,}768$ weights. This shape comparison omits biases and normalization parameters. The lower branch instead keeps spatial filtering depthwise and spends capacity on an expanded channel map. Its LN reduces over channels at each spatial position, not over the entire image cube.

View the corresponding lecture diagram (Figure 11.3, p. 141).

Addition requires matching axes, not just compatible products. If a block changes width from three to five, a raw 4×3 input cannot be added to its 4×5 output. A skip projection $S = XW_s$ produces the required shape. For images, stride may also need to match

the spatial grid. Concatenating branches instead of adding them would enlarge an axis and define a different architecture.

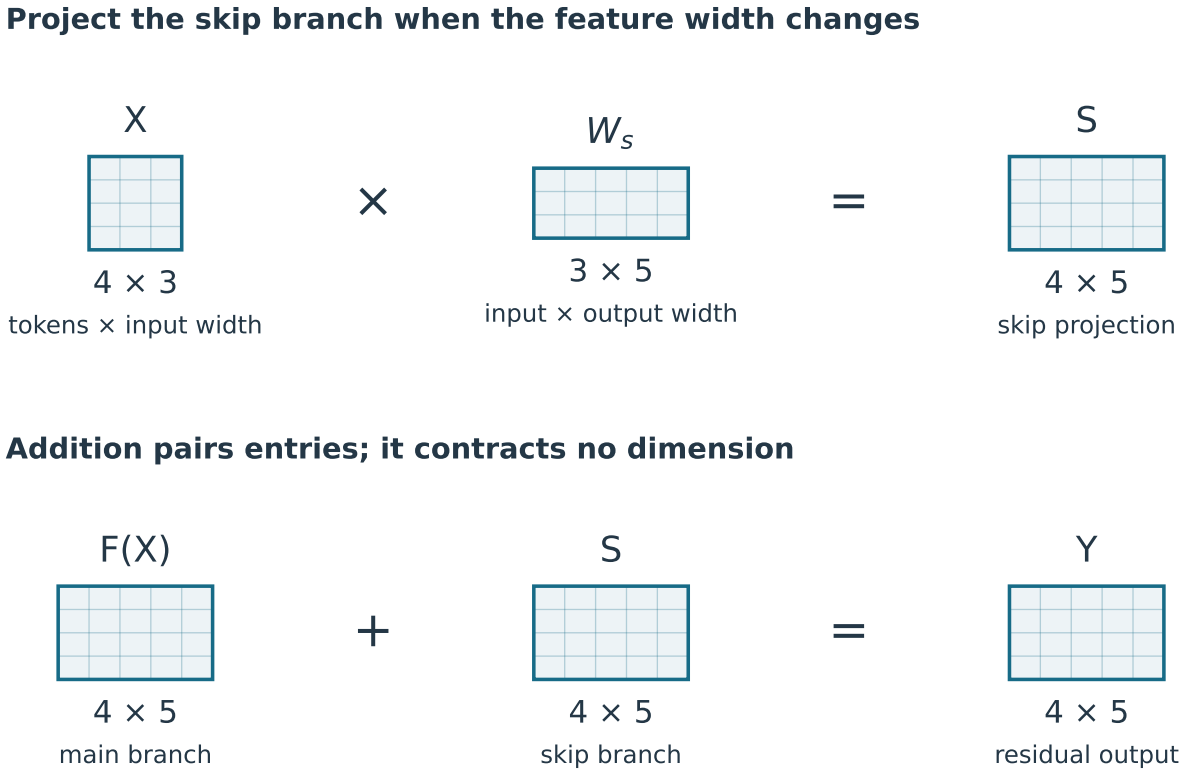


Figure G.3: A residual projection aligns the skip branch with the main branch. The final operation is elementwise addition, not matrix multiplication.

A conventional channel bottleneck compresses channels with a 1×1 map, performs spatial processing at the narrow width, then expands back. This reduces the expensive spatial computation. An inverted bottleneck expands an internal channel representation before projecting back. These arrangements have different computation and representation tradeoffs.

A ConvNeXt-style block combines a depthwise spatial convolution, normalization, channel expansion, a nonlinear activation, and channel projection inside a residual branch. A larger depthwise kernel broadens spatial context while remaining inexpensive relative to a full dense-channel kernel. Much of the channel mixing is performed by pointwise maps.

For a width-C block with expansion factor r and spatial kernel size k, a simplified parameter count is k^2C+2rC^2, ignoring normalization and biases. The C^2 pointwise terms can dominate. Changing kernel size and changing expansion width therefore affect cost differently.

The lecture uses these architectures to illustrate that small design details can matter: normalization placement, activation placement, channel ratios, and the location of resolution changes all influence training. A comparison between architecture names alone can conceal differences in training recipes and computational budget. The reader does not treat the old draft's broad performance claim as a universal ranking of ConvNets and transformers.

G.6 Uneven overlap in transposed convolution

Return to lecture: Lecture 11, §11.4

Transpose the operator, not its meaning. In the example $C : \mathbb{R}^5 \to \mathbb{R}^3$ is a valid width-three convolution. Backpropagation uses $C^\top : \mathbb{R}^3 \to \mathbb{R}^5$. Each output sensitivity contributes to every input position used by that output, and shared positions accumulate contributions. For stride or padding changes, build the corresponding forward operator first; its transpose determines the backward geometry. The transpose restores an input-shaped vector, but it does not reconstruct an arbitrary lost input.

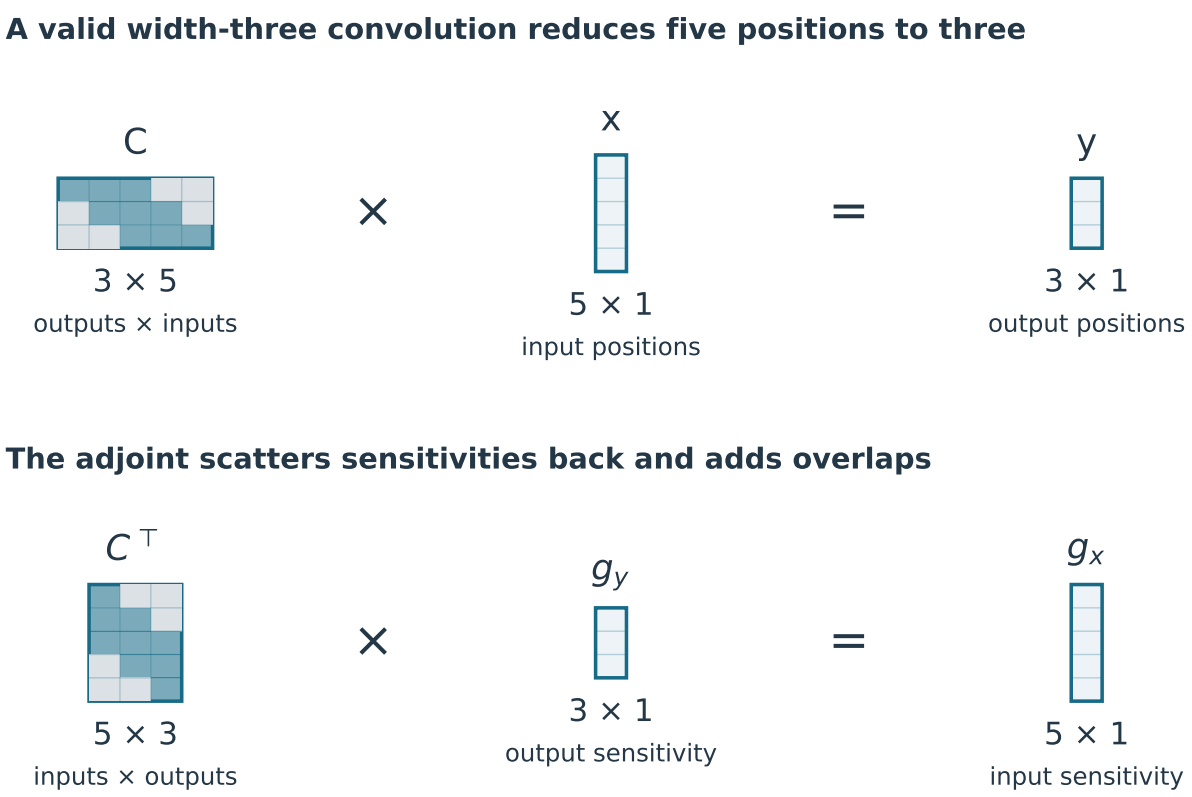


Figure G.4: The transposed convolution reverses the rectangular operator's direction. Support and overlap, rather than an inverse, determine its entries.

A transposed convolution spreads each low-resolution value through a shifted kernel and adds overlapping contributions. If stride and kernel size create an unequal number of overlaps at different output positions, even a simple input can produce a periodic pattern in output magnitudes.

For stride two and a length-three all-ones kernel, consecutive low-resolution values contribute to positions $(0, 1, 2)$ and $(2, 3, 4)$. With both values equal to one, the output is $(1, 1, 2, 1, 1)$. The central overlap receives twice the contribution. In two dimensions, analogous overlap patterns can create checkerboard-like structure.

Using a kernel compatible with the stride can reduce one source of uneven overlap, but learned weights can still produce artifacts. Upsampling by a fixed interpolation rule followed by an ordinary convolution is another design. The choice should be evaluated in the actual reconstruction task rather than equating all enlargement operations.

A U-Net shape check. If an encoder reduces a $64 \times 64 \times 32$ tensor to $32 \times 32 \times 64$, the decoder can upsample back to 64×64 and concatenate the earlier 32-channel feature map. Concatenation increases channel width; it is not residual addition. The following convolution must accept the sum of the two channel counts. Spatial cropping or padding may be needed if the encoder used valid convolutions or odd dimensions.

G.7 U-Net skip concatenation: memory and gradient accounting

Return to lecture: Lecture 11, §11.5

Using the lecture's same-padding U-Net, suppose retained encoder tensors have shapes $128 \times 128 \times 32$, $64 \times 64 \times 64$, and $32 \times 32 \times 128$. In float32 their storage is respectively 2, 1, and 0.5 MiB per example, since one MiB is 2^{20} bytes. These skips alone occupy 3.5 MiB; optimizer state, decoder activations, temporary workspaces, and saved backward tensors are additional costs.

At the finest level, concatenating two $128 \times 128 \times 32$ tensors produces $128 \times 128 \times 64$, or 4 MiB. A following 3×3 convolution with 32 output channels has $3^2 \cdot 64 \cdot 32 = 18{,}432$ weights, excluding biases. Adding the tensors instead leaves 32 input channels and halves that weight count, but requires aligned channels and cannot preserve the two inputs as separately addressable channel groups.

If $z = [e; d]$ is channel concatenation, the incoming gradient splits: $g_e = g_z[: C_e]$ and $g_d = g_z[C_e :]$. If $z = e + d$, both local derivatives are identity and both inputs receive g_z. Therefore concatenation preserves separate routes that the next learned layer can combine, whereas addition commits to a particular immediate combination. This comparison concerns these local operations; either architecture can learn further transformations.

Supplement H

Graph neural networks and differentiable pooling

Source trail: lecture 12, lecture 13; discussion 7. The central design constraint is that relabeling nodes should not change the underlying graph computation.

H.1 A numerical message-passing pass

Return to lecture: Lecture 12, §12.5

Take a three-node chain with scalar features $(1, 2, 4)$. Use a self-inclusive sum and no learned matrix or nonlinearity. The new features are $(1+2, 1+2+4, 2+4) = (3, 7, 6)$. A second layer gives $(10, 16, 13)$. The end nodes can now depend on each other through the middle node.

After k local message-passing layers, information can travel at most k graph hops unless other connections are introduced. This is a graph version of a receptive field. Increasing depth expands access, but repeated averaging can make nodes similar, and compressing a large neighborhood into a fixed vector can lose information. Access to a distant node does not imply that all its information survives.

Attention-weighted aggregation computes scores s_{ij} and normalized weights

$$\alpha_{ij} = \frac{\exp(s_{ij})}{\sum_{r \in \mathcal{N}(i)} \exp(s_{ir})}, \qquad m_i = \sum_j \alpha_{ij} v_j.$$

If scores and values are constructed with shared functions and consistent edge information, the operation respects relabeling. This creates a conceptual bridge from GNNs to transformers: attention can learn which interactions matter.

H.2 A concrete train/test leakage example on a graph

Return to lecture: Lecture 13, §13.3

Consider the undirected chain 1–2–3, with nodes 1 and 2 used for training and node 3 held out. A mean aggregator without self-loops has

$$P = \begin{bmatrix} 0 & 1 & 0 \\ 1/2 & 0 & 1/2 \\ 0 & 1 & 0 \end{bmatrix}.$$

If features X are permitted at all nodes in a transductive task, $(PX)_2 = (X_1 + X_3)/2$ is a legitimate training representation. In an inductive task that withholds node 3 and its edges, the training graph and normalization must instead exclude node 3.

Now suppose a preprocessing step inserts the true labels into a feature column y and caches Py. The representation at training node 2 contains $y_3/2$. Masking the supervised loss at node 3 afterwards does not remove that leaked label. The information already entered a training-node representation before the loss was computed. The remedy is to construct label-dependent inputs and preprocessing using only labels authorized for that split, and to rebuild caches for each split. The distinction is between permitted graph/features and unavailable labels; a transductive protocol permits the former without licensing the latter.

H.3 From a pixel grid to an arbitrary graph

Return to lecture: Lecture 12, §12.4, Lecture 12, §12.3

Graph propagation uses two distinct contractions. In AHW, A acts on node indices while W acts on feature indices. With the convention A_{ij} weights a message from sender j to receiver i, $(AH)_i = \sum_j A_{ij} H_j$. Then W maps the aggregated feature vector to the next width. Associativity gives $(AH)W = A(HW)$ for these linear maps; moving a nonlinearity between them generally breaks that equivalence.

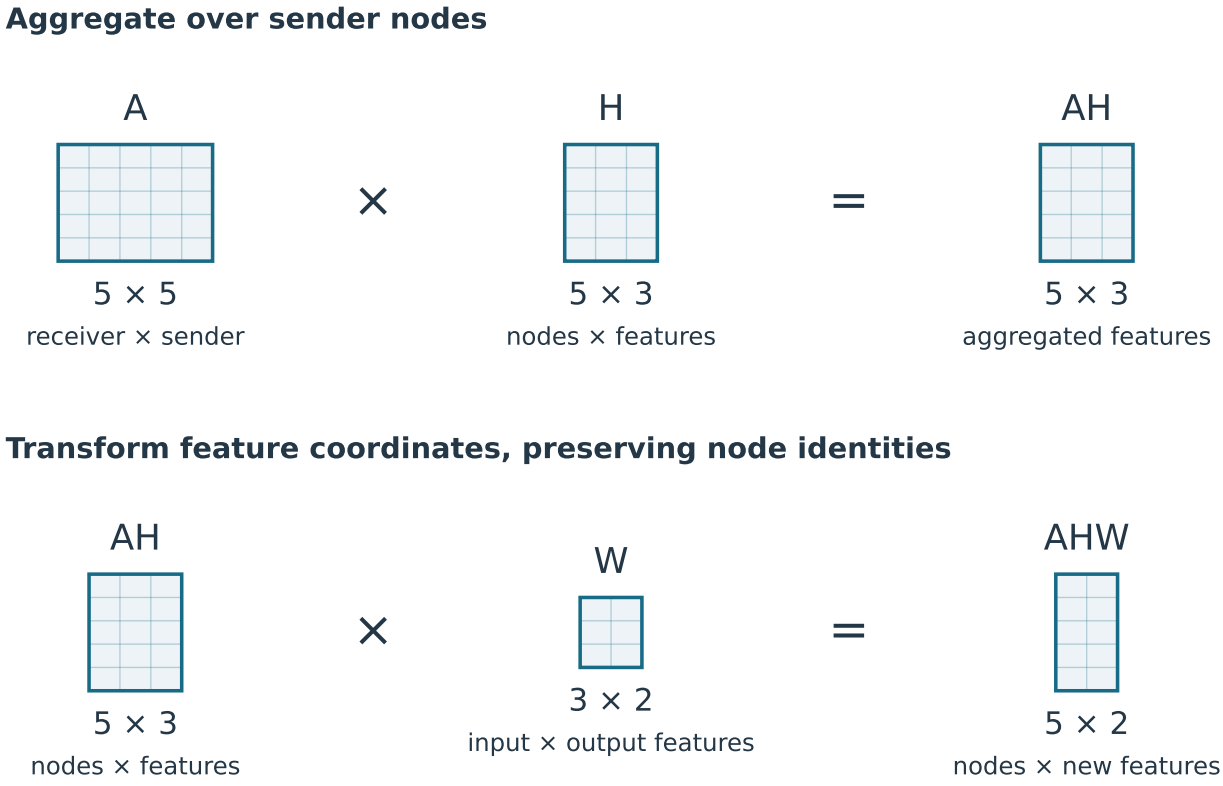


Figure H.1: Node mixing and feature mixing occur on different axes. A node permutation must relabel both axes of the adjacency and the row axis of the features.

A convolution can be viewed as message passing on a regular grid. Each pixel receives messages from a fixed pattern of neighboring pixels, and the weight depends on the neighbor's relative offset. On an arbitrary graph, there may be no canonical notion of left, right, or top, and node degrees can vary. A general graph layer must therefore define how edge information and neighbor sets replace grid coordinates.

Let $\mathcal{N}(i)$ be the neighbors of node i. A message can depend on both endpoint features and an edge feature:

$$m_{ij} = \phi(h_i, h_j, e_{ij}), \quad m_i = \sum_{j \in \mathcal{N}(i)} m_{ij}, \quad h'_i = \psi(h_i, m_i).$$

The sum is independent of neighbor ordering. Mean and maximum are also permutation-invariant aggregators, but they preserve different information. If all neighboring messages equal v, a sum reveals degree through $|\mathcal{N}(i)|v$, while a mean returns v regardless of degree. Neither choice is universally correct; the task determines whether multiplicity should matter.

For a graph-level prediction, a readout such as $\sum_i h_i$ is invariant to node relabeling. A node-level prediction should be equivariant: its outputs must follow the nodes when the nodes are

reordered. Confusing these two requirements can lead to an architecture that discards the very identity its output must preserve.

H.4 Degree normalization and repeated propagation

Return to lecture: Lecture 12, §12.5

A common linear propagation matrix is

$$S = \widetilde{D}^{-1/2}\widetilde{A}\widetilde{D}^{-1/2}, \qquad \widetilde{A} = A + I,$$

where $\widetilde{D}$ contains the row sums of $\widetilde{A}$. Self-loops retain a node's own representation in the aggregation. Degree normalization reduces the tendency of high-degree nodes to produce much larger sums.

The matrix $\widetilde{D}^{-1}\widetilde{A}$ has rows summing to one and describes a random-walk averaging operation on a nonnegative graph. The symmetric normalization S is related to that matrix by a similarity transformation. Its symmetry makes spectral analysis easier, but its rows do not generally sum to one.

Repeated linear propagation gives $S^k H$. Eigencomponents with magnitude below one decay with depth, so many distinct node features can become dominated by a small set of slowly decaying modes. This is a simplified explanation of oversmoothing. Nonlinearities, learned maps, residual connections, and graph structure change the full behavior; the linear calculation identifies a mechanism rather than proving an inevitable collapse for every GNN.

Depth also controls locality. After k message-passing layers, a node can depend only on nodes within k graph hops, absent global operations. A long-range task may require more layers, but simply adding depth can worsen optimization or homogenize features. Rewiring, global nodes, attention, and hierarchical pooling are ways to change the communication pattern.

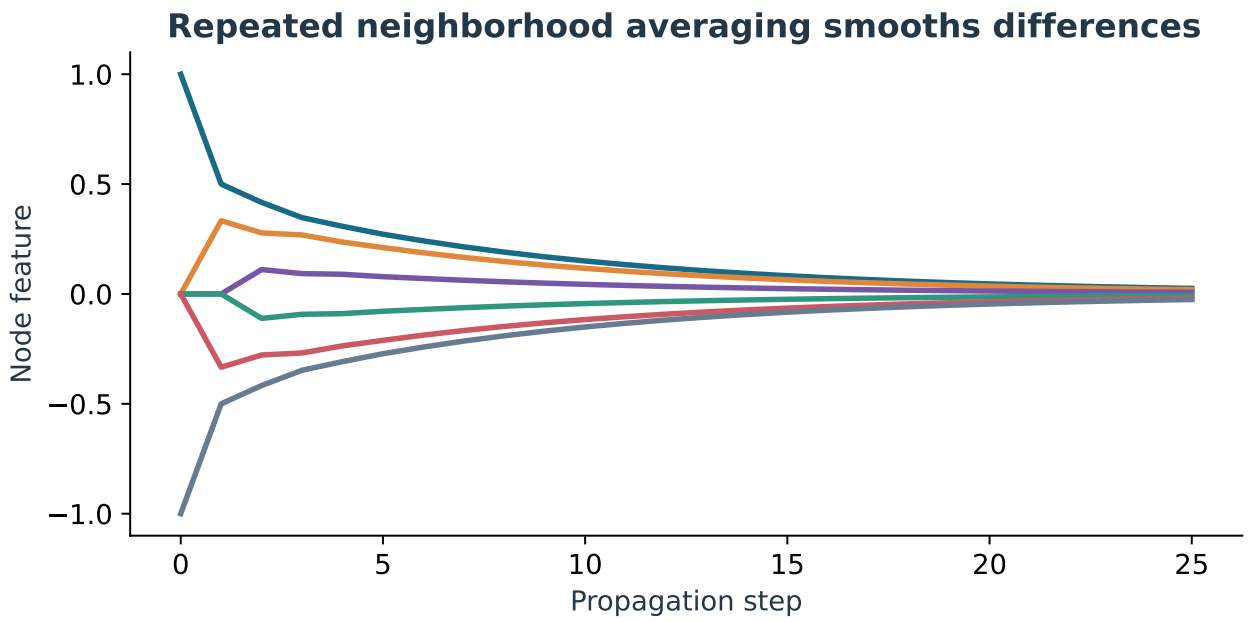


Figure H.2: Repeated random-walk averaging on a small chain reduces differences between node features. This linear example isolates an oversmoothing mechanism.

Receptive fields grow one hop per local layer. With synchronous updates and a self route, $h_v^{(k)}$ can depend only on initial features at graph distance at most k. In the picture, d, e, f can first influence v after two layers, while g needs three. This is a bound on which inputs can influence the output, not a guarantee that the learned weights retain all their information. Increasing depth expands the reachable neighborhood but can also mix away distinctions. Permutation-invariant aggregation ensures consistent behavior under relabeling; it does not by itself prevent oversmoothing. Global attention, graph-wide normalization, or global readouts introduce additional dependencies outside this strictly local model.

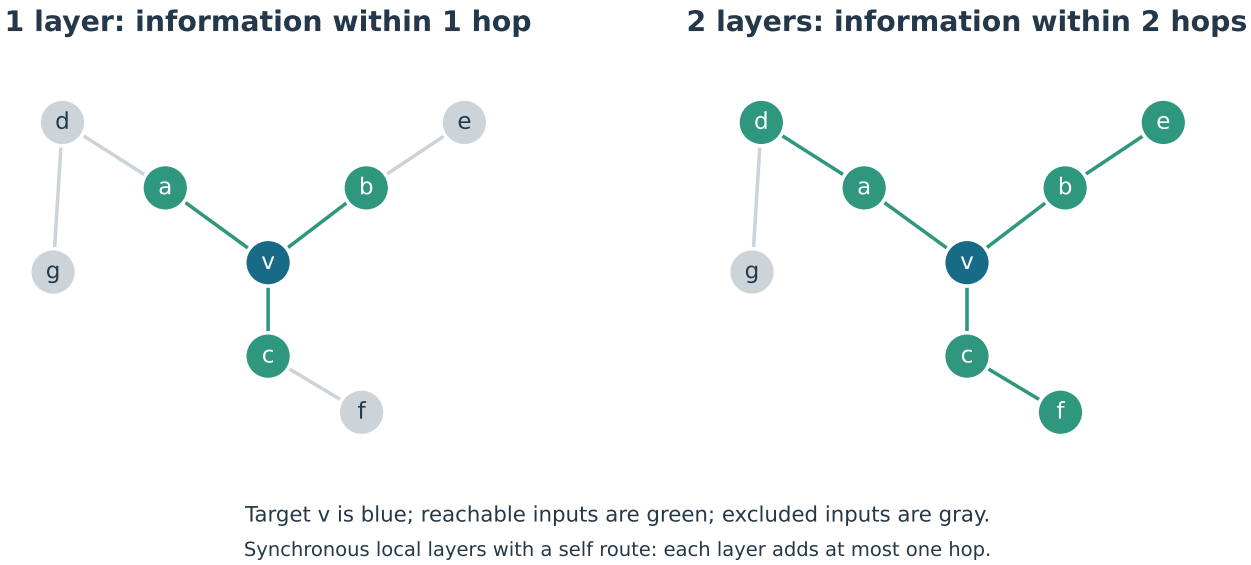


Figure H.3: The same graph after one and two local message-passing layers, viewed from target node v. Green marks potentially reachable input features; gray nodes remain outside the receptive field.

H.5 Graph attention is a normalized neighbor-weighting rule

Return to lecture: Lecture 12, §12.6

Suppose a node computes scores s_{ij} over its neighbors and normalizes them:

$$\alpha_{ij} = \frac{\exp(s_{ij})}{\sum_{k \in \mathcal{N}(i)} \exp(s_{ik})}, \qquad h'_i = \sum_{j \in \mathcal{N}(i)} \alpha_{ij} W h_j.$$

The scores may depend on learned projections of the receiving and sending features. The softmax denominator is specific to the receiving node's neighborhood. If two nodes have different neighbors, even an identical pairwise score can receive a different normalized weight.

Permutation equivariance requires that relabeling the graph also relabel the scores and neighborhoods consistently. A shared score function and a symmetric normalization over the neighbor set satisfy this property. Assigning a special weight to the first neighbor in an arbitrary input list generally does not.

Sparse graph attention evaluates only existing edges and costs roughly in proportion to the number of edges, excluding feature projections. Dense attention over all node pairs changes both the communication graph and the cost to quadratic in node count. A graph network and a transformer can use related attention formulas while imposing different connectivity assumptions.

Read attention as weighted edges. Take three allowed neighbors with scalar transformed values $1, 4, 2$ and illustrative scores $(\log 2, \log 3, \log 5)$. Their exponentials sum to ten, so the normalized weights are $(0.2, 0.3, 0.5)$ and the aggregate is 2.4. The picture excludes a self-loop; adding one adds another score to the same normalization denominator. A weight is

specific to the receiving node and the current features: an edge's reverse direction need not have the same weight. The weighted sum is the message before any subsequent activation or update map.

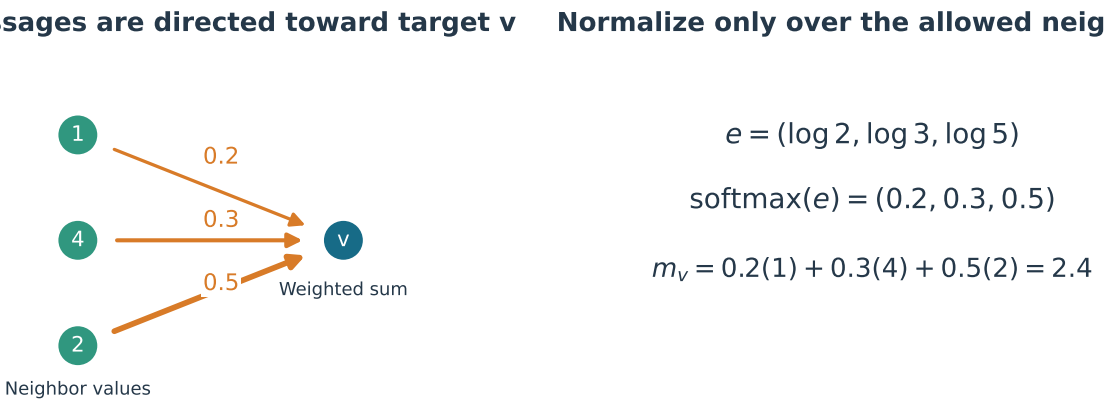


Figure H.4: Incoming arrows show the direction of information flow. Their weights sum to one over the allowed neighbors of this target, and their scalar values produce the displayed weighted sum.

H.6 DiffPool gradients and what a soft assignment means

Return to lecture: Lecture 13, §13.2

Pooling changes which entities index the rows. Let $S \in \mathbb{R}^{N \times K}$ assign original nodes to clusters. Feature pooling is $H_{\text{pool}} = S^\top H$, while adjacency pooling is $A_{\text{pool}} = S^\top AS$. The latter contracts both original edge endpoints. Even if each row of S sums to one, the rows of $S^\top$ generally do not: pooled features are weighted sums unless additional normalization is applied. $S^\top$ is not an inverse assignment map.

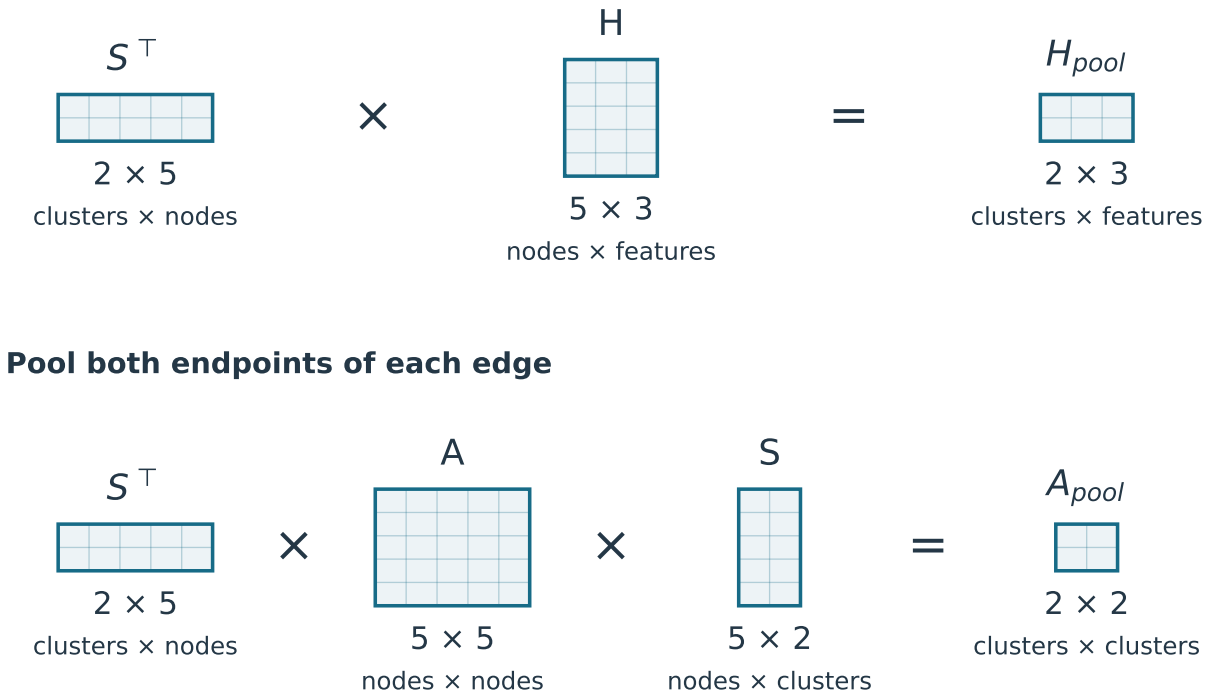


Figure H.5: DiffPool replaces the node axes with cluster axes. Features have one such axis; adjacency has two.

Let $S \in \mathbb{R}^{n \times k}$ contain soft cluster memberships, often with rowwise softmax so every row sums to one. Let Z be learned node embeddings. The pooled graph is

$$Z' = S^\top Z, \qquad A' = S^\top A S.$$

The first expression adds node features according to membership weights. It is a weighted sum, not a weighted mean unless separately normalized. Large clusters can therefore have larger feature magnitudes. The second expression accumulates edge mass between clusters, including diagonal entries for within-cluster connections.

For a downstream feature gradient $G = \nabla_{Z'} L$, the contribution from $Z' = S^\top Z$ is

$$\nabla_Z L = SG, \qquad \nabla_S L = ZG^\top.$$

The adjacency path supplies additional gradients to S. If $E = \nabla_{A'} L$, differentiating $S^\top A S$ gives

$$\nabla_S L\big|_{A'} = ASE^\top + A^\top SE.$$

For symmetric A and symmetric E, this becomes $2ASE$. These formulas show why pooling assignments can be trained end to end: both pooled features and pooled connectivity influence them.

Auxiliary connectivity or entropy losses bias the assignments. Sharp memberships need not be semantically correct; interpret them together with the task objective.

See the graph behind the pooled matrix. Assign nodes $1, 2, 3$ to one cluster and $4, 5, 6$ to the other. Each cluster contains a triangle, and two undirected edges cross between clusters. For the symmetric adjacency with unit edge weights, $S^\top AS = \left[\begin{smallmatrix} 6 & 2 \\ 2 & 6 \end{smallmatrix}\right]$: each internal edge contributes both ordered endpoint pairs to a diagonal entry. These diagonal values are aggregated internal connectivity, not six original self-loops. With scalar features $h_i = i$, feature pooling gives $(6, 15)^\top$. This hard-assignment example makes the bookkeeping visible; finite softmax assignments are generally soft and spread contributions across multiple cluster pairs.

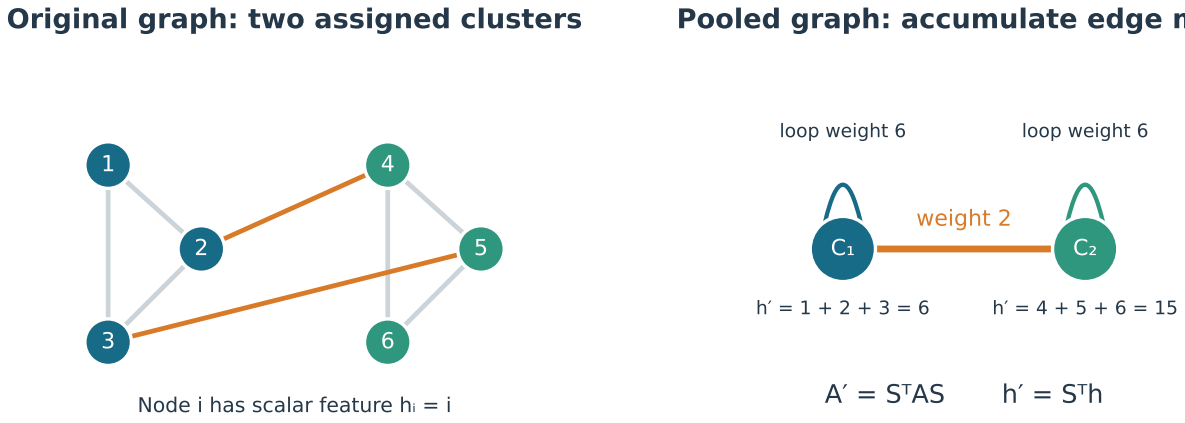


Figure H.6: A six-node graph becomes a two-node weighted graph. Node colors show assignments, orange edges cross clusters, and pooled loops represent accumulated internal edge mass.

Supplement I

Recurrent networks and self-supervised prediction

Source trail: lecture 14, lecture 15, the end of lecture 13; discussions 7–8. The lecture's signal-processing perspective distinguishes recurrence across time from depth across layers.

I.1 Nonunique latent coordinates and different reconstruction targets

Return to lecture: Lecture 14, §14.6

Four reconstruction tasks impose different constraints. A bottleneck limits the representation's dimension. A sparse autoencoder can instead use a wide code with loss $\|x - D(E(x))\|^2 + \lambda\|E(x)\|_1$, encouraging selective activation rather than a small number of available units. A denoising objective predicts clean x from corrupted $\tilde{x}$; the squared-error optimum is $\mathbb{E}[x \mid \tilde{x}]$, not necessarily the particular clean example that produced a given ambiguous observation. A masked objective hides selected coordinates and can score only those coordinates, preventing direct copying of the targets. Each task may preserve different information useful to a downstream predictor.

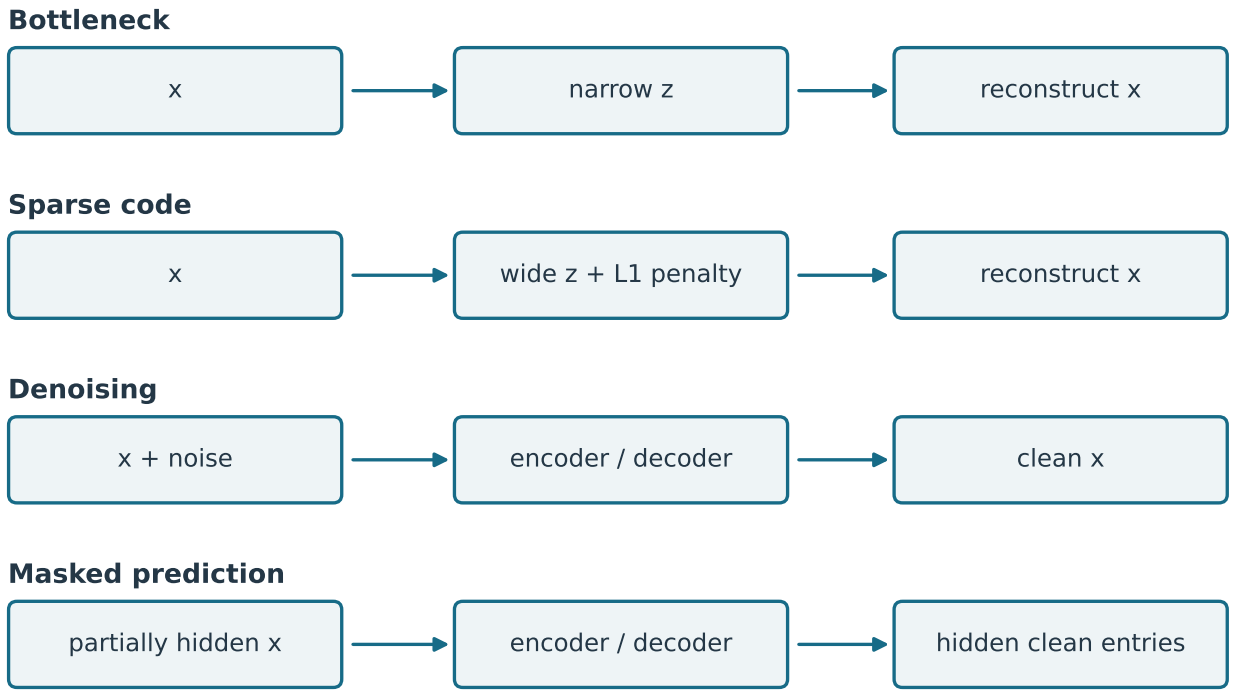


Figure I.1: Autoencoder variants differ in information available at the input, constraints on the code, and the scored target. A dimensional bottleneck, sparsity penalty, corruption, and masking are not interchangeable.

The bottleneck limits rank even when input and output shapes agree. With examples in columns, $E \in \mathbb{R}^{2\times 5}$ compresses $X \in \mathbb{R}^{5\times B}$ and $D \in \mathbb{R}^{5\times 2}$ reconstructs it. The resulting 5×5 map DE has rank at most two and therefore cannot be the identity on all of $\mathbb{R}^5$.

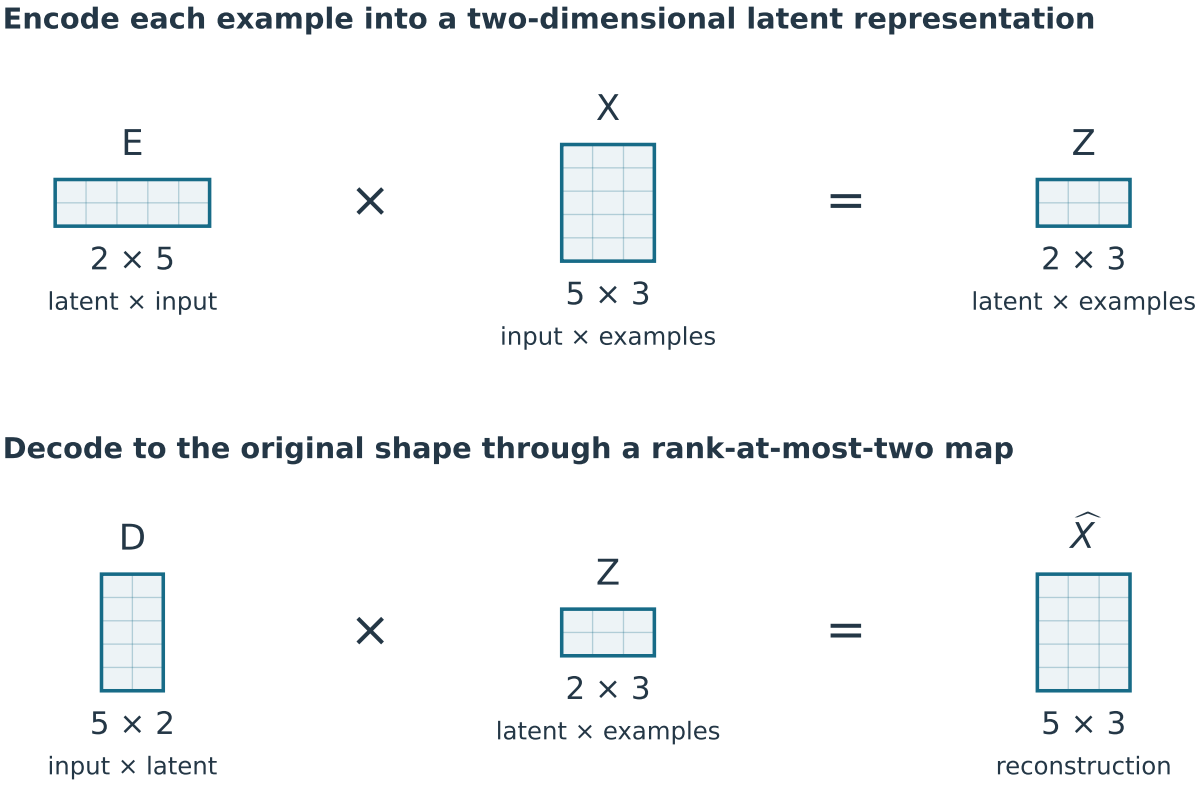


Figure I.2: A linear autoencoder returns to input shape while restricting the reconstruction subspace. Equal input/output dimensions do not imply an invertible network.

Reconstruction alone cannot identify latent coordinates. If R is invertible, replacing E by RE and D by DR^{-1} leaves DE unchanged. For $R = \mathrm{diag}(2, 1/2)$, the first code coordinate doubles and the second halves while every reconstruction stays fixed. Code norms, distances, and a sparsity penalty can change despite identical reconstruction error. Consequently a downstream procedure that depends on latent geometry needs additional conventions or regularization; an optimal reconstruction subspace does not uniquely determine a meaningful coordinate system.

I.2 Backpropagation through time with a loss at every step

Return to lecture: Lecture 13, §13.4, Lecture 14, §14.3

Keep time separate from the batch axis. For three independent sequences, put the states at a fixed time into $H_{t-1} \in \mathbb{R}^{4\times 3}$ and inputs into $X_t \in \mathbb{R}^{2\times 3}$. Both $W_h H_{t-1}$ and $W_x X_t$ have shape 4×3, so they can be added before the activation. Advancing time reuses the

same matrices; it does not mean multiplying across the example columns. Backpropagation accumulates parameter contributions over both time and examples.

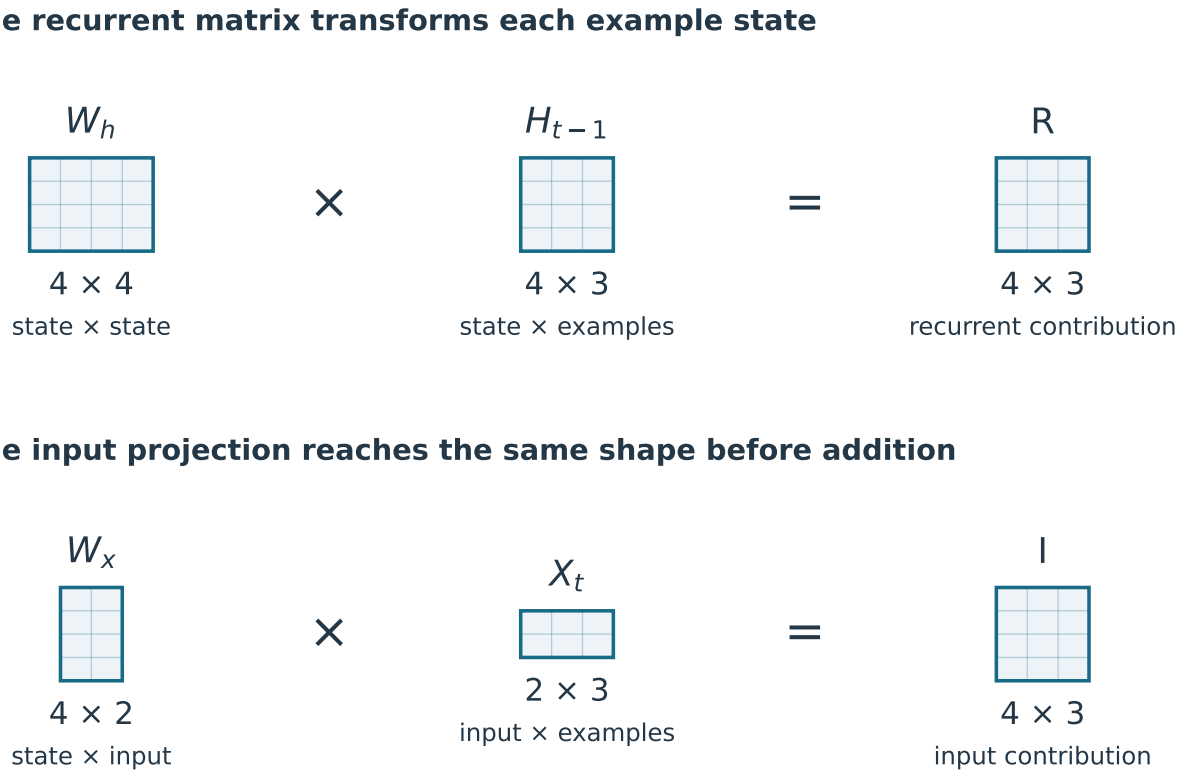


Figure I.3: Two recurrent-layer branches meet in state space. The example axis is preserved throughout each time step.

For $h_t = \phi(Wh_{t-1} + Ux_t + b)$ and $L = \sum_{t=1}^{T} \ell_t(h_t)$, define $a_t = Wh_{t-1} + Ux_t + b$ and the preactivation sensitivity $\delta_t = \nabla_{a_t} L$. Working backward,

$$\delta_t = \phi'(a_t) \odot \left(\nabla_{h_t} \ell_t + W^\top \delta_{t+1}\right), \quad \delta_{T+1} = 0.$$

The first term is the direct loss at time t; the second is the effect of h_t on all future losses. Shared-parameter gradients are

$$\nabla_W L = \sum_t \delta_t h_{t-1}^\top, \quad \nabla_U L = \sum_t \delta_t x_t^\top, \quad \nabla_b L = \sum_t \delta_t.$$

One recurrent matrix is reused many times, so every use contributes. Treating an unrolled diagram as if each copy had separate parameters computes a different model.

A product of Jacobians controls long-range sensitivity. A spectral-radius argument is suggestive for a fixed linear recurrence, but nonlinear, time-varying Jacobians need singular-value

or product analysis. Even matrices whose eigenvalues individually look harmless can have transient amplification when they are nonnormal. Gradient clipping bounds a realized update signal; it does not repair the loss of information from a gradient that has already vanished.

Truncated BPTT stops gradient propagation across selected state boundaries. The forward state can still carry information from earlier inputs, but the parameter update does not credit those earlier computations through the truncated boundary. This is a biased approximation to full-sequence differentiation that trades memory and computation for a shorter learning horizon.

I.3 Gates create controllable memory paths

Return to lecture: Lecture 14, §14.2, Lecture 14, §14.1

A gated memory cell separates retaining old information from writing new information. An LSTM-style cell uses

$$f_t = \sigma(W_f[h_{t-1}; x_t] + b_f), \qquad i_t = \sigma(W_i[h_{t-1}; x_t] + b_i), \tag{I.1}$$

$$o_t = \sigma(W_o[h_{t-1}; x_t] + b_o), \qquad \tilde{c}_t = \tanh(W_c[h_{t-1}; x_t] + b_c), \tag{I.2}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t, \qquad h_t = o_t \odot \tanh(c_t). \tag{I.3}$$

Along the direct cell-state path, the derivative is multiplied by f_t, which can be near one when the model should retain information. There are also indirect paths through gate dependencies. The full Jacobian is therefore more than just $\mathrm{diag}(f_t)$.

A GRU-style recurrence combines a retained hidden state and a candidate update, with update and reset gates. Different references reverse the symbol used for retention versus writing; write the actual equation before interpreting a gate's value. If $h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t$, then $z_t \approx 1$ means retention under that convention.

These gates are a useful bridge to selective state-space models. Both allow input-dependent control over what to keep and what to overwrite. The computational details differ: a general nonlinear recurrent transition does not become a fixed convolution merely because it has a memory interpretation.

I.4 The Kalman filter as an exact recurrent inference program

Return to lecture: Lecture 14, §14.4

Consider a linear Gaussian system

$$s_t = As_{t-1} + \omega_t, \qquad x_t = Cs_t + \nu_t,$$

with independent Gaussian process noise of covariance Q and observation noise of covariance R. If the previous posterior is Gaussian with mean m_{t-1} and covariance P_{t-1}, the prediction step is

$$m_t^- = Am_{t-1}, \qquad P_t^- = AP_{t-1}A^\top + Q.$$

The innovation is $e_t = x_t - Cm_t^-$ with covariance $S_t = CP_t^-C^\top + R$. The update is

$$K_t = P_t^-C^\top S_t^{-1}, \qquad m_t = m_t^- + K_t e_t, \qquad P_t = (I - K_t C)P_t^-.$$

The gain weighs trust in the new observation against uncertainty in the prediction. In one dimension, if the prior variance is 4 and observation-noise variance is1 with $C = 1$, the gain is 4/5. If the prior mean is0 and the observation is 5, the updated mean is 4 and variance is0.8.

This is a recurrent computation with an interpretable hidden state: posterior statistics. A learned recurrent model trained to predict observations may discover a representation sufficient for prediction without recovering these exact coordinates. Any invertible reparameterization of a sufficient state can support the same external behavior.

Self-supervised prediction therefore offers an observable training signal for a hidden-state problem, but it does not identify the physical latent state uniquely. The lecture's Kalman example illustrates learning useful state from observations rather than requiring a label for every hidden variable.

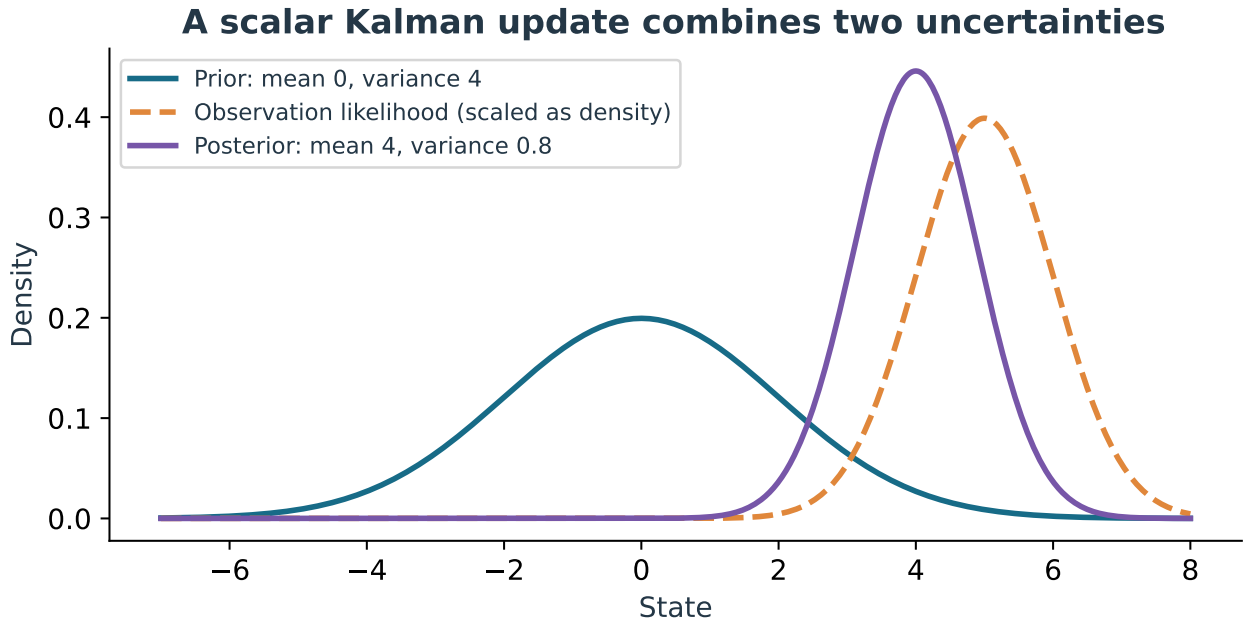


Figure I.4: A Kalman update combines prior uncertainty with an observation likelihood. The posterior mean lies between the prior mean and the observation.

Observability is not an instantaneous inverse

Discussion 8's system-identification discussion informally uses C^{-1} to recover a state from its observation. This requires a square invertible observation matrix, which a general state-space model need not have. For example, let

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \qquad C = \begin{pmatrix} 1 & 0 \end{pmatrix}.$$

A single noiseless observation $y_t = Ch_t$ reveals only the first state coordinate. With zero input, the next observation is $y_{t+1} = CAh_t$ and reveals the second. The stacked observability matrix $[C^\top, (CA)^\top]^\top = I_2$ has full column rank, even though C has no two-sided inverse. State inference can therefore require a history of observations. With noise, the Kalman filter additionally tracks uncertainty; an inverse of a single observation map does not replace that inference problem.

I.5 Contrastive prediction and the role of negative examples

Return to lecture: Lecture 14, §14.5, Lecture 15, §15.1

Contrastive learning constructs a batch-pair matrix. For two aligned views of four examples, stack their embeddings as rows of $Z, Z' \in \mathbb{R}^{4\times 3}$. The score matrix $ZZ'^\top/\tau$ is 4×4: each anchor row compares against all candidate examples. Under this alignment the positive candidate is on the diagonal, and a row-softmax loss assigns it the target class. Unlike ordinary feature maps, this matrix has example identity on both axes. A same-example pair and a semantically related different example need not receive the same target.

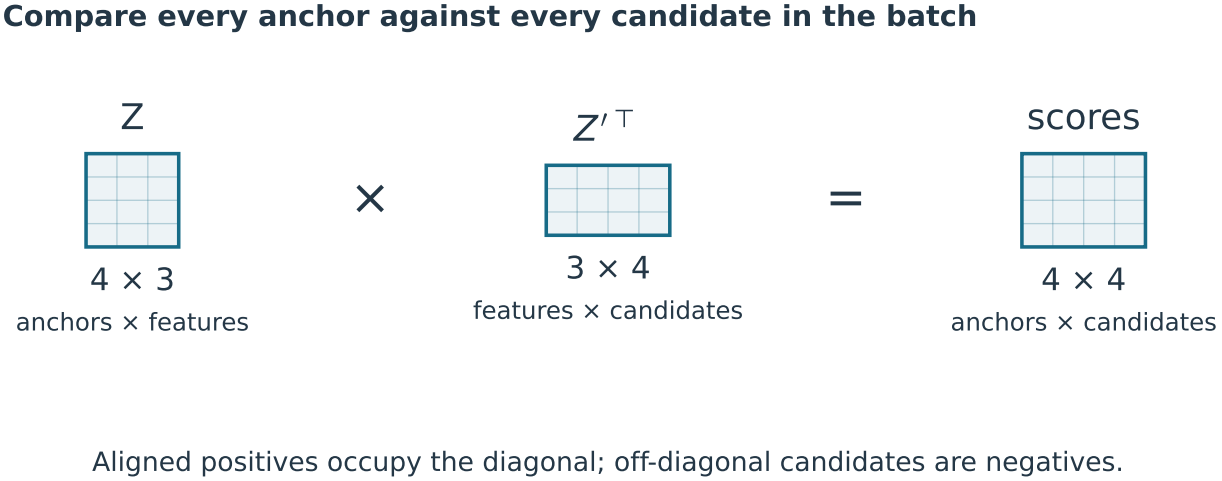


Figure I.5: Contrastive similarities contract embedding features to create example-pair scores. The diagonal-positive convention depends on keeping the two views aligned.

For an anchor representation z, a positive representation z^+, and candidate negatives z_j^-, a contrastive loss can take the form

$$-\log \frac{\exp(s(z, z^+)/\tau)}{\exp(s(z, z^+)/\tau) + \sum_j \exp(s(z, z_j^-)/\tau)}.$$

It is a classification loss asking which candidate matches the anchor under the chosen positive-pair construction. Augmentations define what should be preserved. Negative sampling defines what should be distinguished. If two semantically similar examples are treated as negatives, the objective can push them apart.

A constant representation cannot usually distinguish the positive from the negatives and therefore incurs a nonminimal contrastive loss. Reconstruction and prediction methods prevent collapse through different mechanisms. The phrase "self-supervised" describes where targets come from, not one universal objective or one guarantee of useful representation quality.

Supplement J

State-space models: recurrence, convolution, and selective memory

Source trail: lecture 16, lecture 17; discussion 9. The lecture 16 PDF also supports the state-space material begun alongside lecture 15. The notes and recordings have overlapping boundaries.

J.1 The full convolution derivation, including initial state

Return to lecture: Lecture 15, §15.2

Use the convention $h_t = Ah_{t-1} + Bx_t$ and $y_t = Ch_t + Dx_t$. Repeated substitution gives

$$h_t = A^t h_0 + \sum_{j=1}^{t} A^{t-j} B x_j,$$

so

$$y_t = CA^t h_0 + \sum_{j=1}^{t} CA^{t-j} B x_j + Dx_t.$$

The convolution matrix acts on time. For a scalar-input, scalar-output LTI system with zero initial state, stack a length-T sequence into a column. The matrix K_{time} is lower-triangular Toeplitz: each diagonal repeats a kernel coefficient. Its main diagonal is $CB + D$ and its ℓth lower diagonal is $CA^{\ell}B$. Two columns can hold two independent examples. With multiple input/output channels, each scalar entry becomes a block. Input-dependent selective parameters generally destroy the fixed Toeplitz pattern, even though the recurrence remains causal.

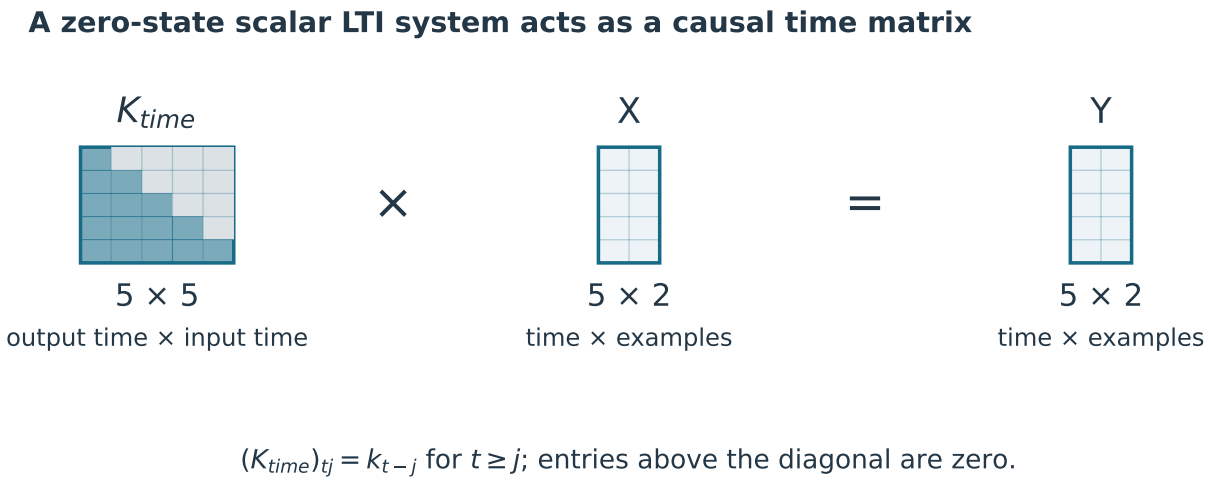


Figure J.1: A recurrent SSM becomes one time-axis matrix product. Triangular support enforces causality; constant diagonals express time invariance.

The initial-state response is separate from the input convolution. Setting $h_0 = 0$ removes it, but a streaming system continuing from a previous segment may have a nonzero state. Resetting the state at arbitrary segment boundaries changes the model's behavior.

The fixed convolution kernel has instantaneous term $CB + D$ and later terms CA^kB. A kernel with long support can be applied by FFT convolution after sufficient zero padding to avoid circular wraparound. The FFT changes the computation strategy; it does not make a time-varying system time invariant.

For a finite sequence of length T, a direct convolution can require quadratic work in T if the kernel is also length T. An FFT approach can reduce the convolution part to order $T \log T$. Kernel construction and feature projections have their own costs. Recurrent generation instead updates a state whose size does not grow with the number of previously processed tokens.

J.2 Diagonalization, poles, and memory

Return to lecture: Lecture 16, §16.1

If $A = V\Lambda V^{-1}$, then

$$CA^kB = CV\Lambda^kV^{-1}B = \sum_i \widetilde{C}_i \lambda_i^k \widetilde{B}_i.$$

Each mode contributes an exponential sequence. A real positive eigenvalue below one gives decaying memory; a negative eigenvalue alternates signs; a complex-conjugate pair gives damped oscillation in a real-valued system. Eigenvalues near the unit circle provide long time scales but can also make optimization and numerical stability more delicate.

The eigenvalues alone do not determine every observed behavior. A mode matters to the output only if it is excited through B and observed through C. A nonnormal eigenvector

matrix can cause transient amplification. A Jordan block can introduce polynomial factors multiplying the exponential. Consequently, "all eigenvalues have modulus at most one" is weaker than a full bounded-input bounded-output stability analysis.

For $0 < \lambda < 1$, define a decay time $\tau = -1/\log\lambda$. Then $\lambda^k = e^{-k/\tau}$. When λ is close to one, $\tau \approx 1/(1-\lambda)$. A bank of modes with different λ values can represent multiple memory scales. Structured SSM parameterizations make such long kernels tractable without storing an unrestricted dense transition matrix and repeatedly multiplying it naively.

J.3 Exact discretization and the meaning of a learned step size

Return to lecture: Lecture 16, §16.3

For a continuous-time system $\dot{h} = Ah + Bx$ with input held constant over an interval Δ,

$$h(t+\Delta) = e^{A\Delta}h(t) + \left(\int_0^{\Delta} e^{A(\Delta-s)}B\,ds\right)x.$$

If A is invertible, the input coefficient is $A^{-1}(e^{A\Delta} - I)B$. The integral form remains valid when A is singular and avoids pretending an inverse always exists.

In the scalar case $A = -a$ with $a > 0$, the update is

$$h_t = e^{-a\Delta_t}h_{t-1} + \frac{1 - e^{-a\Delta_t}}{a}Bx_t.$$

For small Δ_t, the state mostly retains its past. For large Δ_t, the old-state coefficient shrinks and the current input has greater influence. If Δ_t depends on the input, it acts as a selection mechanism. This is a concrete interpretation of selective time scales, not just a metaphor about memory.

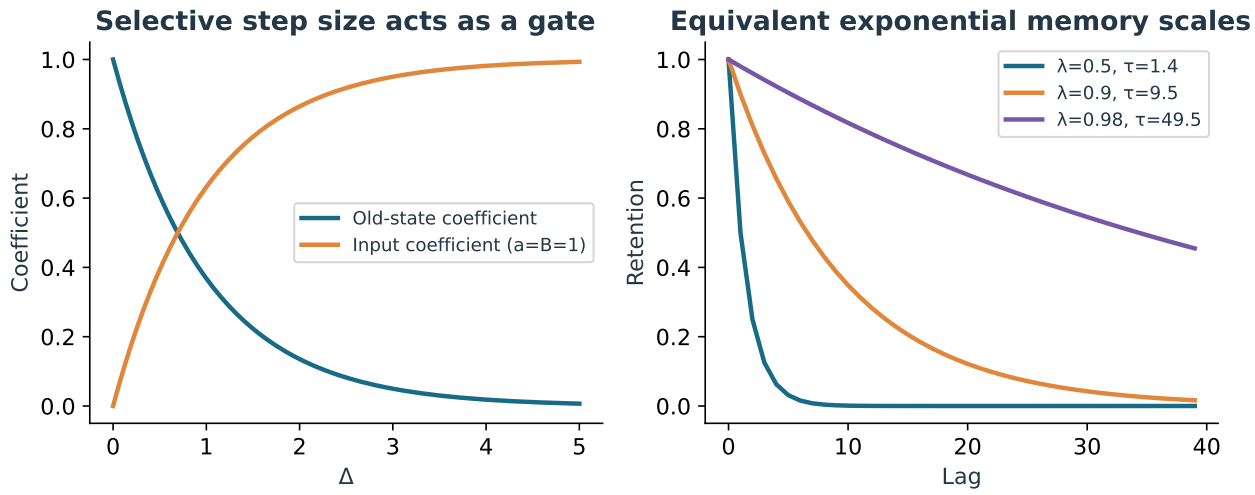


Figure J.2: An input-dependent discretization interval controls retention and writing. Fixed eigenvalues correspond to exponential memory time scales.

J.4 Associative scans preserve a different kind of parallelism

Return to lecture: Lecture 16, §16.2

A time-varying affine recurrence $h_t = A_t h_{t-1} + b_t$ corresponds to the affine map $f_t(h) = A_t h + b_t$. Composing two updates gives

$$(A_2, b_2) \circ (A_1, b_1) = (A_2 A_1, A_2 b_1 + b_2).$$

Function composition is associative, so these pairs can be combined using a parallel prefix scan. Associativity does not require commutativity: changing the order of time steps generally changes the result.

A shortcut in Discussion 9 writes a product over a subinterval as a ratio of two global prefix products. That identity requires a nonzero denominator. It fails when an earlier coefficient is zero and can be unstable when long prefix products underflow. Composing affine pairs directly avoids division and remains algebraically valid for zero or singular transitions. Parallel evaluation can still change floating-point rounding; associativity here is an exact-arithmetic statement.

For scalar updates $(a_1, b_1) = (2, 1)$ and $(a_2, b_2) = (3, 4)$, the composition is $(6, 7)$, because $3(2h + 1) + 4 = 6h + 7$. Reversing the order gives $(6, 9)$. A correct scan preserves sequence order while changing the grouping of computations.

Selective SSMs exploit structure such as diagonal transitions to make these compositions efficient. An unrestricted dense matrix scan may have substantial matrix-multiplication cost. The opportunity is therefore not merely that affine maps associate, but that the chosen parameterization keeps the scan's elementary operations cheap.

J.5 Compare state memory with retrieval memory

Return to lecture: Lecture 16, §16.3

An SSM compresses history into a fixed-dimensional state; attention retains a growing collection of keys and values. Ask whether the task needs a smooth summary or exact retrieval of a particular earlier event. Hybrid models can combine these mechanisms, but their actual state size, connectivity, and computation determine the tradeoff.

Supplement K

Attention as differentiable retrieval

Source trail: lecture 18; discussion 9 and discussion 10 question 1. The lecture builds attention from the failure of a single encoder state to preserve everything a decoder may need.

K.1 Gradients through the retrieval weights

Return to lecture: Lecture 17, §17.6

For $\alpha = \mathrm{softmax}(s)$,

$$\frac{\partial \alpha_i}{\partial s_j} = \alpha_i(\delta_{ij} - \alpha_j).$$

If $o = \sum_i \alpha_i v_i$ and downstream sensitivity is g_o, then

$$\frac{\partial L}{\partial s_j} = \alpha_j g_o^\top (v_j - o).$$

The score of a value increases when that value would improve the output relative to the current weighted average, with the sign determined by the loss gradient. This equation gives a more precise interpretation than saying attention simply learns to "look at important words." Importance is task- and layer-dependent.

Attention weights are not automatically a causal explanation of the final prediction. Values, residual paths, later layers, and alternative parameterizations also contribute. They are internal mixture coefficients that can be inspected, but their interpretation needs care.

K.2 Track attention with rectangular matrices

Return to lecture: Lecture 17, §17.6

Attention here uses *rows as tokens*, unlike the column-example convention in Supplement A. For T_q queries, T_k keys, key width d_k, and value width d_v,

$$Q : T_q \times d_k, \quad K^\top : d_k \times T_k, \quad P : T_q \times T_k, \quad V : T_k \times d_v, \quad O : T_q \times d_v.$$

The first product contracts feature coordinates to produce query–key scores. Row-wise softmax preserves the score matrix's shape. The second product contracts key positions to mix value vectors. In self-attention $T_q = T_k$; in cross-attention the two sequence lengths can differ.

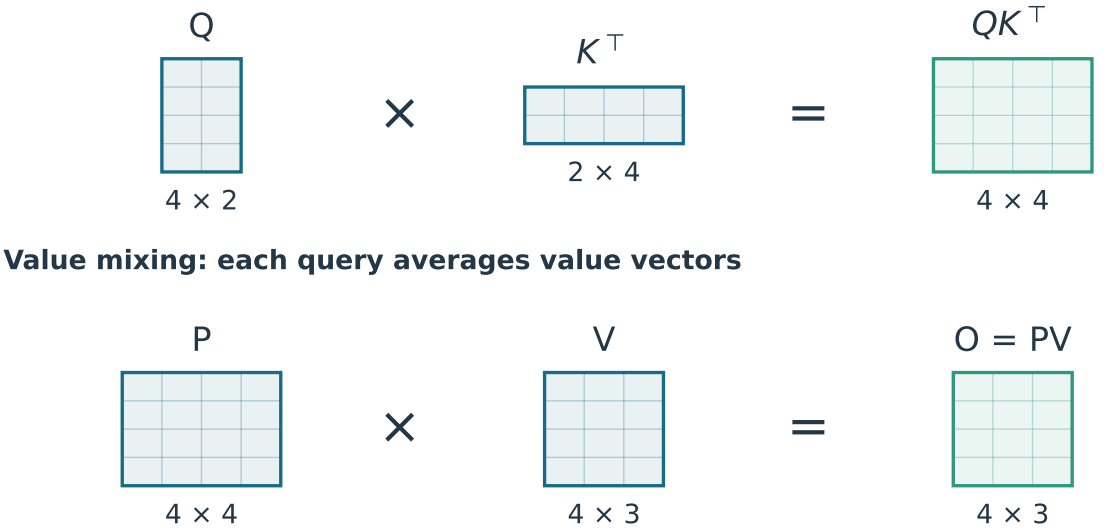


Figure K.1: Two different contractions inside attention. In this example there are four query and key positions, key width two, and value width three. Set $S = QK^\top/\sqrt{2}$ before applying the mask and row-wise softmax.

A causal mask changes which entries of P may be nonzero, not the output dimensions. The $T_q \times T_k$ score matrix also explains the quadratic sequence-length cost of ordinary self-attention. Its axes are token positions; it is not a $d_k \times d_k$ feature covariance matrix.

K.3 Derive all projection gradients in one place

Return to lecture: Lecture 17, §17.6

For row-token matrices, let $S = QK^\top/\sqrt{d} + M$, $P = \mathrm{softmax}_{\mathrm{row}}(S)$, and $O = PV$. Given an incoming gradient G_O, first differentiate the value mixing:

$$G_V = P^\top G_O, \qquad G_P = G_O V^\top.$$

For each row i, the softmax derivative gives

$$(G_S)_{ij} = P_{ij}\left[(G_P)_{ij} - \sum_k P_{ik}(G_P)_{ik}\right].$$

Then

$$G_Q = G_S K/\sqrt{d}, \qquad G_K = G_S^\top Q/\sqrt{d}.$$

If $Q = XW_Q$, then $\nabla_{W_Q} L = X^\top G_Q$ and the input receives $G_Q W_Q^\top$. Add the analogous key and value contributions because X feeds all three branches. Omitting one of those contributions is the attention analogue of forgetting a branch in a simple computation graph.

Masked positions have probability zero and should contribute zero score gradient. Numerically, the mask is often implemented using a sufficiently negative value or a masked softmax. A row with no permitted keys cannot be normalized as a probability distribution without a special convention; preventing that situation is usually cleaner than letting undefined values propagate.

K.4 A two-token numerical attention example

Return to lecture: Lecture 17, §17.3, Lecture 18, §18.2

Take scalar keys and queries with $Q = (1, 2)^\top$, $K = (1, 0)^\top$, and values $V = (3, 9)^\top$. Without a mask, the first query has scores $(1, 0)$ and output

$$\frac{3e + 9}{e + 1} \approx 4.614.$$

The second has scores $(2, 0)$ and output $(3e^2 + 9)/(e^2 + 1) \approx 3.715$. The stronger query concentrates more weight on the first key.

With a causal mask, the first query can read only the first token, so its output becomes exactly 3. The second output is unchanged because both keys are in its prefix. The projection matrices did not change; the information constraint did. A mask is therefore part of the model, not a cosmetic adjustment to a visualization.

Now suppose a loss wants the second output to increase. Its score derivative is proportional to $P_{2j}(v_j - o_2)$. The first value lies below the current average and receives a derivative with the opposite sign from the second value. Gradient descent on a suitable increasing-target loss adjusts relevance toward values that move the output in the desired direction.

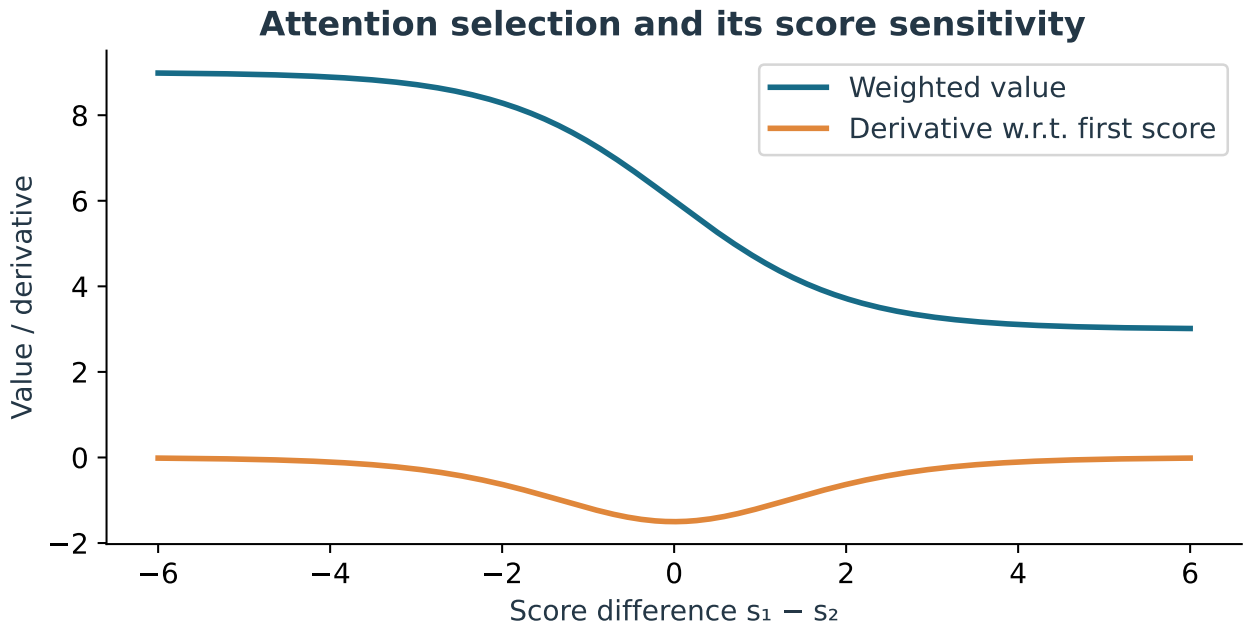


Figure K.2: The output of a two-value attention mixture changes smoothly with score difference. Its derivative shrinks as selection saturates.

K.5 Softmax temperature controls sensitivity as well as selection

Return to lecture: Lecture 17, §17.5, Lecture 18, §18.3

For $P_j = \exp(s_j/\tau)/\sum_k \exp(s_k/\tau)$,

$$\frac{\partial P_i}{\partial s_j} = \frac{1}{\tau} P_i(\delta_{ij} - P_j).$$

The factor $1/\tau$ increases as temperature decreases, but probabilities may simultaneously saturate near zero or one. Thus lowering temperature does not monotonically increase every useful gradient. A nearly deterministic choice can have tiny derivatives for most alternatives.

Subtracting the maximum score before exponentiating preserves probabilities and reduces overflow risk:

$$P_j = \frac{\exp(s_j - m)}{\sum_k \exp(s_k - m)}, \qquad m = \max_k s_k.$$

This is an algebraic identity, not an approximation. Scaling by $1/\sqrt{d}$ addresses the distribution of dot products as head width changes; maximum subtraction addresses numerical range. They solve different problems and are commonly used together.

K.6 Attention as a learned kernel, with important limits

Return to lecture: Lecture 17, §17.2, Lecture 17, §17.1

For fixed projected queries and keys, attention resembles normalized kernel regression: weights depend on a similarity function, and the output averages values. The similarity itself is learned through the projections. Unlike a fixed nearest-neighbor method, the network can change what counts as a match and what information each matched item returns.

A single head outputs a convex combination of its value vectors because the weights are nonnegative and sum to one. This restricts that head's output to the convex hull of the values at that stage. The following output projection, multiple heads, residual addition, and nonlinear layers greatly expand the overall model's function class. The convex-hull statement should not be generalized to the final network output.

An attention map also omits the content of the values. A large weight on a near-zero value can have little effect. Different heads may cancel or combine after output projection. To reason about a token's influence, inspect the full computation or a targeted perturbation, rather than treating attention weight alone as a complete causal attribution.

K.7 Where quadratic cost enters

Return to lecture: Lecture 17, §17.7, Lecture 18, §18.4

For T tokens and model width d, forming all query-key scores and multiplying the weights by values each cost roughly order T^2d across heads. The score/weight matrices also occupy order T^2 entries per head if materialized. Projection and MLP costs are closer to order Td^2 for fixed width ratios.

At small T, projection and MLP work can dominate. At large T, pairwise interactions and memory traffic can dominate. Algorithms that avoid materializing the entire attention matrix can reduce memory use while computing the same mathematical attention operation. Sparse or local attention changes the allowed interaction pattern and therefore the model itself. These two forms of efficiency should be distinguished.

Supplement L

Transformers, position, and efficient inference

Source trail: lecture 19, lecture 20, and discussion 10. This chapter puts attention inside a complete trainable sequence model.

L.1 The KV cache and the cost of generation

Return to lecture: Lecture 19, §19.4

During causal generation, old token representations at a given layer do not depend on future tokens. Their keys and values can be cached. The new query attends to those stored keys and values; only the new token's projections need to be appended. This saves recomputing the whole prefix, although reading an increasingly long cache still costs work.

If there are L layers, batch size B, context length T, h_{kv} key/value heads, head dimension d_h, and b bytes per element, the cache requires approximately

$$2LBTh_{kv}d_hb \quad \text{bytes.}$$

The factor two counts keys and values. With $L = 24$, $B = 1$, $T = 4096$, $h_{kv} = 8$, $d_h = 64$, and two-byte elements, this is 201,326,592 bytes, or 192 MiB. Reducing to one KV head reduces it to 24 MiB. These are illustrative architecture choices, not specifications of a particular model. Weight storage, activations, allocator overhead, and temporary buffers are additional costs.

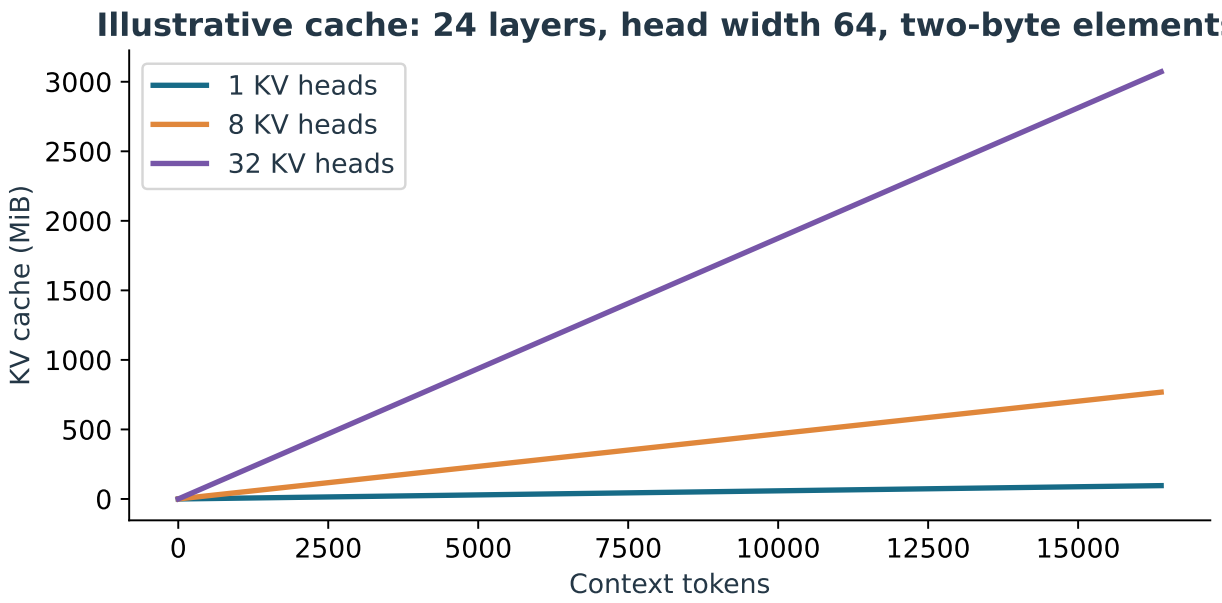


Figure L.1: Cache storage grows linearly with context length. Sharing key/value heads lowers the slope while retaining multiple query heads.

A mixture-of-experts MLP routes tokens to a subset of expert networks, sometimes alongside a shared expert. This can increase total parameter capacity without activating every parameter for each token. Routing and load balancing introduce additional computation and optimization issues. It is not correct to equate the number of stored parameters with the work performed on every token.

Checkpoint. A transformer is a repeated sequence of content-based communication, positionwise nonlinear processing, normalization, and residual addition. Its behavior depends as much on masks and training targets as on the attention equation itself.

L.2 A complete decoder training example

Return to lecture: Lecture 17, §17.4, Lecture 18, §18.5, Lecture 18, §18.1, Lecture 20, §20.1

Track the unchanged axes through a whole block. Attention can mix positions t inside one example, subject to the causal mask. The channel MLP applies the same feature transformation at every position and does not directly mix different tokens. Each branch must return (B, T, D) before its residual addition. Two normalizations and two additions appear in this pre-normalized example; moving normalization after an addition changes the computation. A gated MLP replaces the simple expansion/nonlinearity with parallel feature projections and an elementwise gate, while preserving the required output shape.

View the corresponding lecture diagram (Figure 18.2, p. 234).

Heads split features, not the sequence. In this illustrative two-head block, X has four tokens and model width six. Combined query and key projections each produce width four,

reshaped into two heads of width two; value projections can use the same example widths. Every head still sees all four tokens and forms its own 4×4 score matrix. Concatenating the two 4×2 head outputs gives 4×4, and $W_O \in \mathbb{R}^{4\times 6}$ returns to model width six. The head-feature total need not equal model width, although implementations often choose equality.

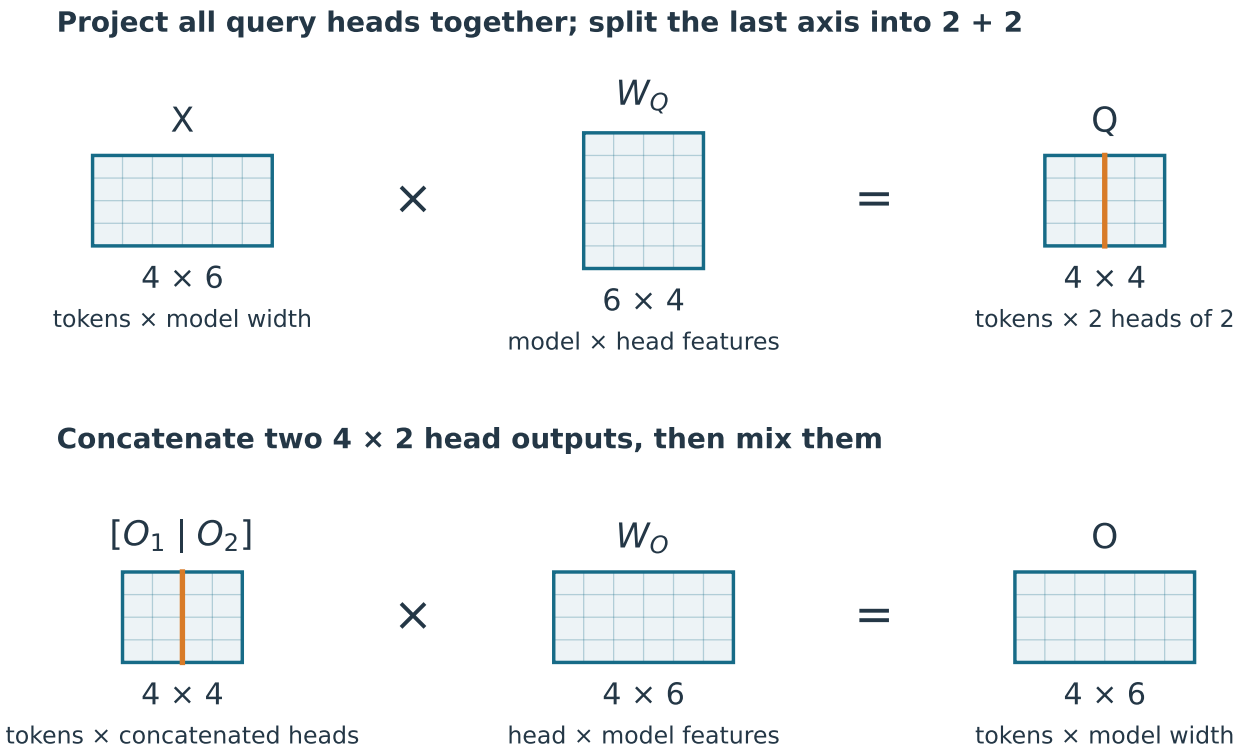


Figure L.2: Projection, head splitting, concatenation, and output mixing preserve the token axis. Concatenation joins columns; it does not add heads together.

Suppose the observed sequence is `the cat sat EOS`. With a beginning token, the input sequence is `BOS the cat sat`, and the targets are `the cat sat EOS`. Position 1 predicts `the` from `BOS`; position 2 predicts `cat` from the prefix `BOS the`; and so on. A causal mask allows these four predictions to be trained in parallel without revealing a future target.

The token loss is evaluated from logits, preferably through a numerically stable log-softmax. If the correct class has logit z_y, then

$$\ell = -z_y + m + \log \sum_j \exp(z_j - m), \qquad m = \max_j z_j.$$

For logits $(2, 1, 0)$ and target index2, the loss is $\log(e^2 + e + 1) - 1 \approx 1.408$. The gradient is the predicted probability vector minus the one-hot target. It does not require differentiating through a sampled token during teacher-forced training.

For instruction tuning, prepend the instruction and mask its loss terms while keeping its tokens available to response positions. For padded batches, remove padding positions from both the intended loss average and the attention keys as appropriate. Dividing by padded length rather than the number of valid target tokens changes example weighting.

L.3 Position aliasing and information supplied by a causal boundary

Return to lecture: Lecture 18, §18.6, Lecture 19, §19.2

The vocabulary is a separate axis. A one-hot token matrix $E \in \mathbb{R}^{T \times V}$ multiplied by an embedding table $W \in \mathbb{R}^{V \times d}$ selects T rows of that table. Software normally performs an index lookup rather than constructing E. If the output weights are tied, hidden states produce vocabulary logits through $HW^\top$, optionally with a bias. The transpose maps features to vocabulary scores; it is not an inverse lookup. Softmax acts over vocabulary entries independently at each predicted position.

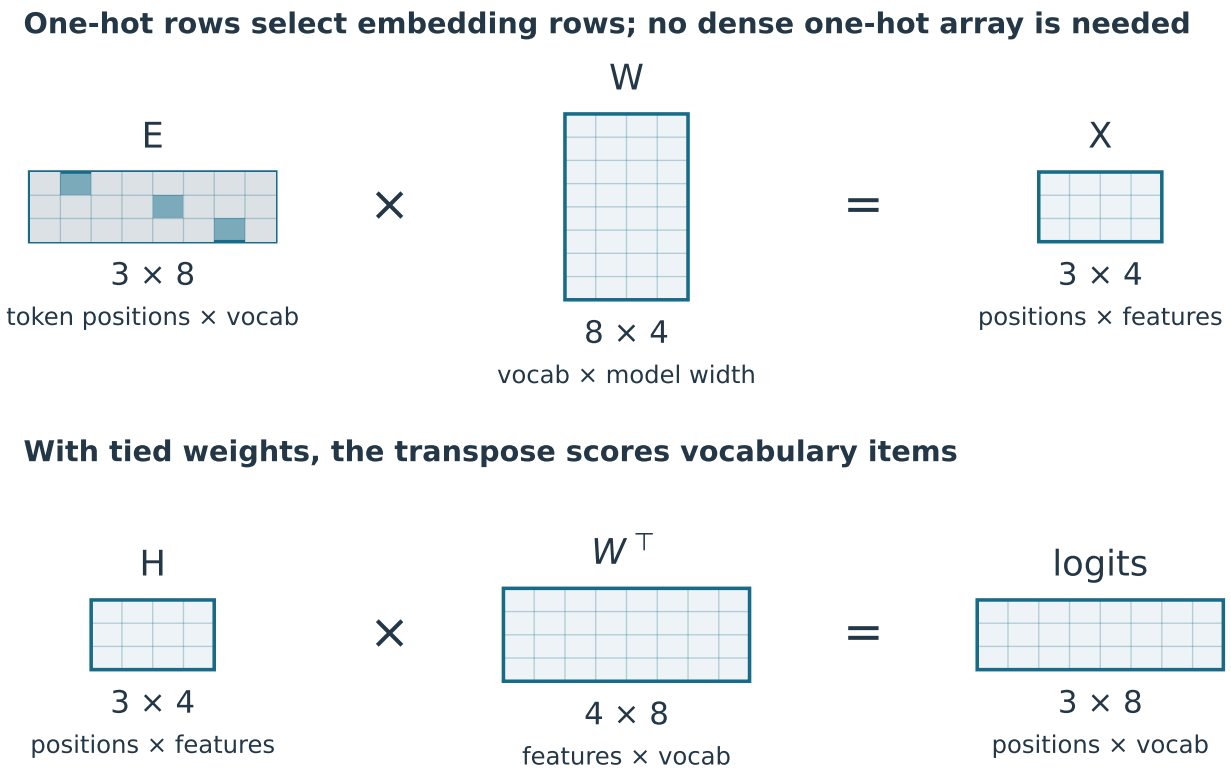


Figure L.3: Vocabulary-to-feature lookup and feature-to-vocabulary scoring run in opposite directions through a rectangular table.

The lecture establishes RoPE's relative-position identity. That identity does not imply unique position recovery. A single rotation frequency ω repeats after a displacement Δ whenever $\omega\Delta$ is an integer multiple of 2π. Multiple frequencies reduce exact collisions, but approximate simultaneous recurrences can still make distant positions look similar. Behavior beyond the training range therefore requires more than the algebraic relative-position property.

A causal mask can supply position-related information without an explicit positional vector, but only through available content. Suppose the first token has a distinguished feature 1 and all later tokens have feature 0. Uniform attention over the first t positions yields $1/t$, from which a later computation could infer t. If all input values are identical, the same uniform average is constant and this particular mechanism cannot infer prefix length. Thus claims about position information must specify content, boundary markers, and the computation available to exploit them.

L.4 Encoder, decoder, and encoder-decoder information sets

Return to lecture: Lecture 19, §19.3

A bidirectional encoder lets every input position read the other unmasked positions. It is suited to representation tasks where the whole input is available. A causal decoder predicts a continuation from a prefix. An encoder-decoder model first encodes a source sequence and then uses causal target-side self-attention plus cross-attention to the source.

In cross-attention, the query length and key length can differ. The encoder's representations can be cached because they do not change while the decoder emits successive tokens. The decoder's self-attention cache grows with its generated prefix. These are two different caches serving two different retrieval operations.

Masked language modeling must hide or corrupt some target information to prevent trivial copying. Causal modeling hides the future through its mask. Both are self-supervised, but their conditional distributions differ. A model trained to fill an arbitrary missing token does not automatically define the same left-to-right factorization as a causal language model.

L.5 Parameter count versus active computation

Return to lecture: Lecture 19, §19.5, Lecture 19, §19.4

A gated MLP has three learned projections. For row-token inputs, $U = XW_u$ and $G = \phi(XW_g)$ share the expanded shape $T \times d_{\text{ff}}$. Their elementwise product $U \odot G$ is then multiplied by W_d to return to $T \times d_{\text{model}}$. No token axis is contracted inside this MLP: different tokens use shared weights but are processed separately. Attention supplies token mixing elsewhere in the block. An ungated MLP removes the separate gate projection, changing both parameter and operation counts.

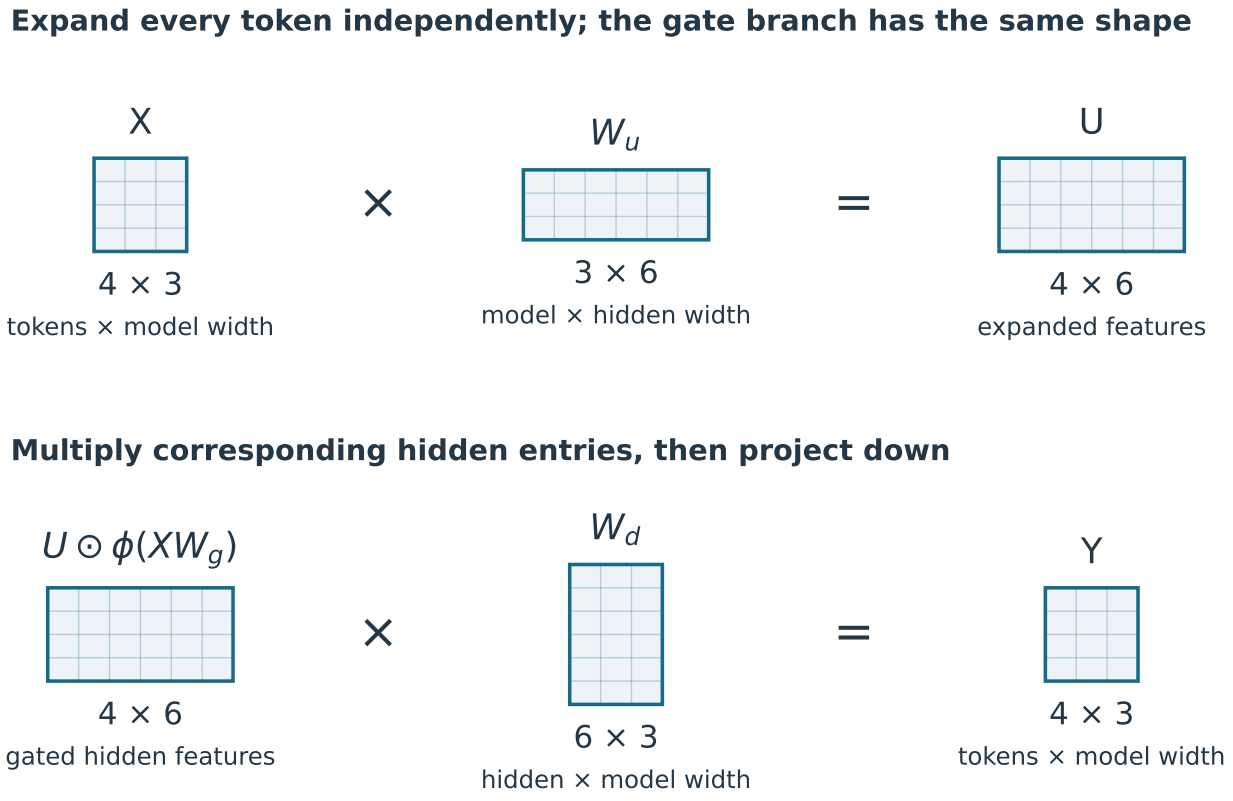


Figure L.4: The two wide hidden branches must align entry by entry. The down projection contracts hidden features, not tokens.

For model width d and a conventional attention block whose combined head width equals d, the query, key, value, and output projections contain about $4d^2$ weights. An MLP with intermediate width rd contains about $2rd^2$ weights in the ungated case. Thus the MLP can contain more weights than attention even though attention is the component with quadratic sequence interactions.

A gated MLP has three principal matrices, giving about $3rd^2$ weights. To compare it with an ungated block at equal parameter count, the intermediate widths may need adjustment. Comparing only the expansion ratio while ignoring the number of matrices can misstate the budget.

For mixture-of-experts, distinguish total expert parameters from the subset activated for a token. A router can select a few experts, but routing decisions, communication, capacity limits, and load imbalance add costs. A model with more stored parameters can have similar per-token arithmetic to a smaller dense model while requiring more total memory.

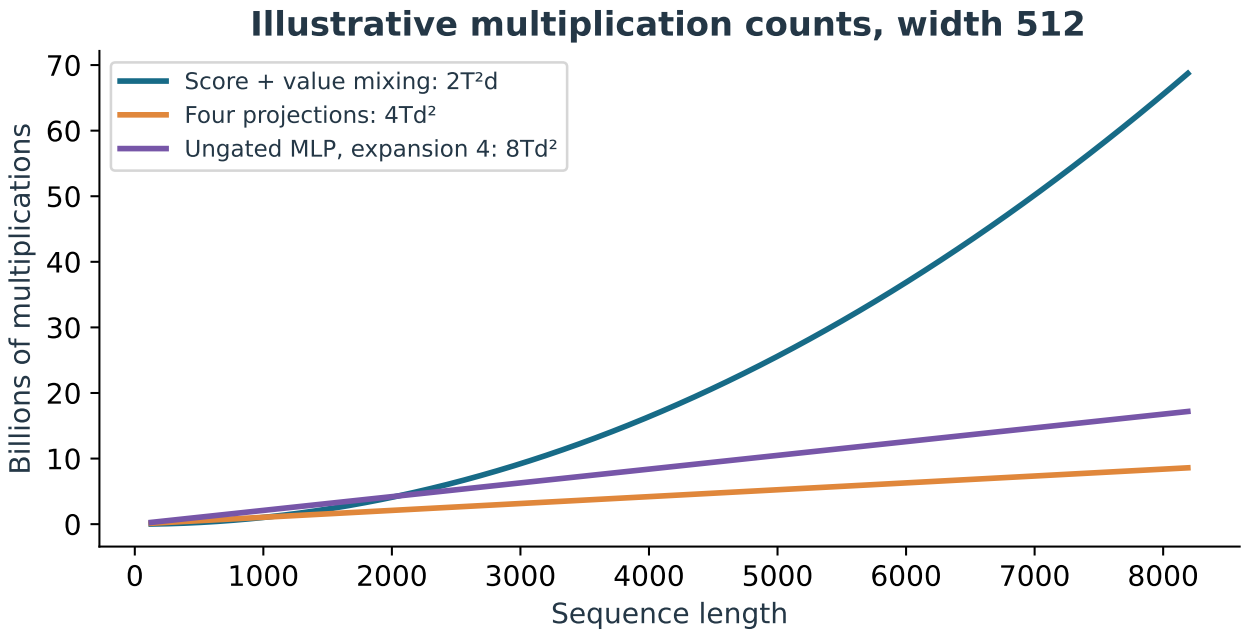


Figure L.5: Different transformer operations dominate at different sequence lengths. These curves count illustrative multiplications rather than measured runtime.

L.6 Cache correctness and generation boundaries

Return to lecture: Lecture 19, §19.1

Caching changes the query axis during generation. At a step with five available keys, a single-head query $q_{\text{new}} \in \mathbb{R}^{1\times d_k}$ creates one score row, rather than recomputing all historical query rows. The key cache has shape $5 \times d_k$ and the value cache $5 \times d_v$. After softmax, the result is one $1 \times d_v$ value mixture. The available keys include the current position when causal self-attention permits attending to self. Caching saves repeated prefix projections and activations, but each new query still reads an increasingly long prefix.

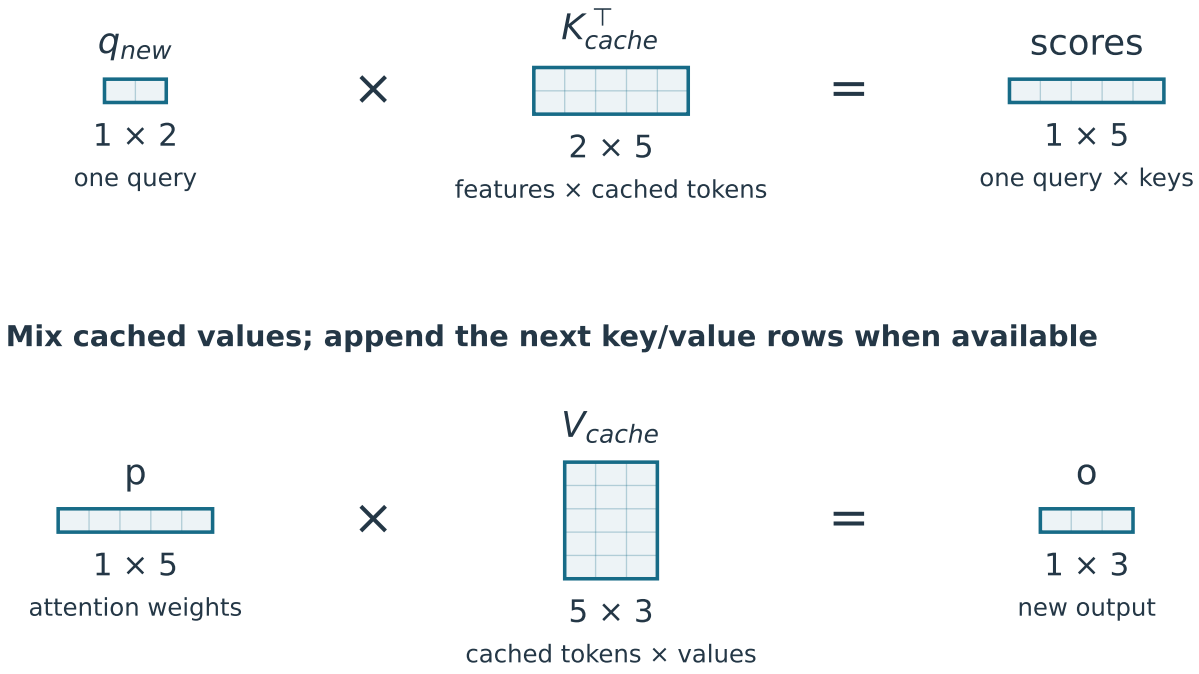


Figure L.6: A decoding step uses a thin query rectangle against growing key/value caches. Token count and key/value feature widths are separate dimensions.

A KV cache is valid when the old hidden representations would be unchanged by appending future tokens. Causal masking provides this property. If an operation recomputes representations using future context, simply retaining old keys and values can become incorrect.

Cache position indices must advance consistently with rotary or other positional transformations. Reusing a cache from a different prompt without the intended shared prefix changes the conditioning context. Beam search also requires reordering cache entries when hypotheses are reordered or pruned. These are consequences of what the cache represents: layer-specific summaries of particular prefix tokens, not a generic pool of reusable vectors.

Worked memory check. Doubling batch size doubles KV storage; doubling context length also doubles it. Doubling the number of query heads while keeping the number and width of KV heads fixed does not necessarily double the cache. This distinction explains the memory benefit of grouped-query and multi-query designs without equating them to a smaller hidden state everywhere in the network.

Supplement M

Adapting a pretrained model

Source trail: lecture 21, lecture 22, and discussion 11. The central question is what changes when a general model is specialized to a task.

M.1 Checking the first LoRA gradient numerically

Return to lecture: Lecture 21, §21.3

Worked gradient check. Let $s = 1$, $A = (1, -1)$, $B = (0, 0)^\top$, $x = (2, 1)^\top$, and $g = (3, -2)^\top$. Then $Ax = 1$, so $\nabla_B L = (3, -2)^\top$, while $\nabla_A L = 0$. A gradient step changes B immediately without initially changing A. This is deliberate symmetry breaking, not an optimizer bug.

M.2 Differentiate a masked response loss through the prompt

Return to lecture: Lecture 20, §20.3, Lecture 21, §21.1

One scalar example separates three masks. Let $h = 2p$ and put loss only on the response: $L = \frac{1}{2}(h - 1)^2$. At $p = 1$, $\partial L/\partial p = 2$ even though the prompt has no separate loss term. Replacing p in the response computation by $\mathrm{sg}(p)$ preserves the forward value and loss but makes that derivative zero. Removing the response's only read of the prompt with an attention mask instead changes the forward dependency itself. In a multilayer network, alternate paths can still carry information unless they too are removed.

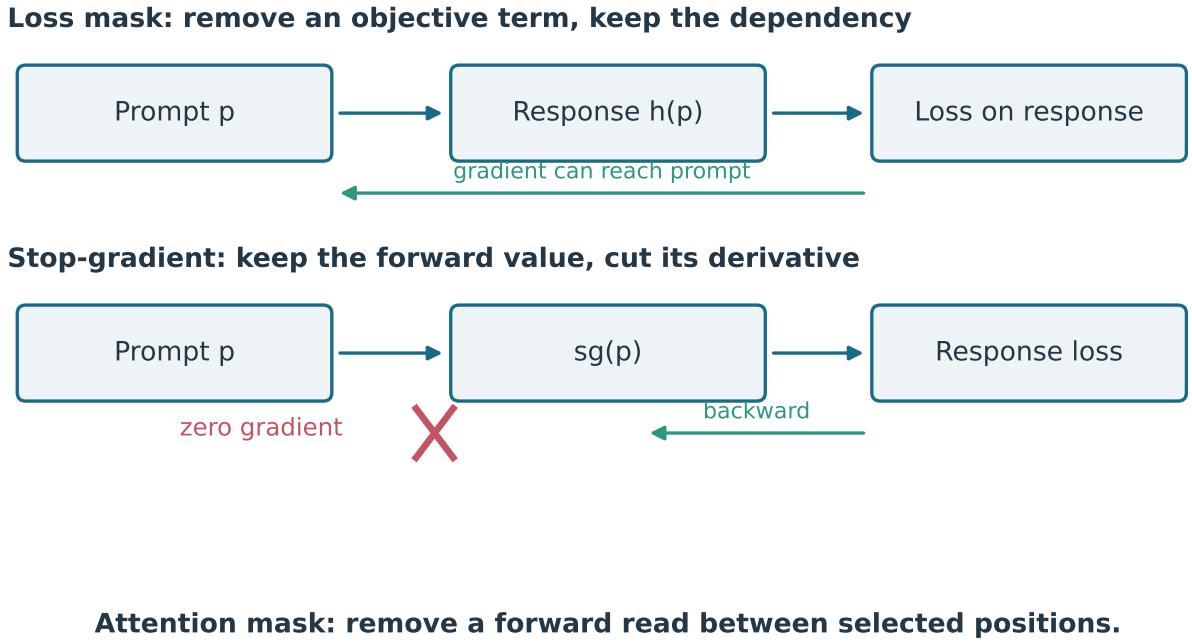


Figure M.1: Loss masking, stopping a derivative, and masking attention are different interventions. Frozen weights also do not imply that gradients through the frozen computation vanish.

Let an input prompt embedding be p and the response hidden state be $h_t = F_\theta(p, v_{<t})$. The response loss is $\ell_t(h_t)$. Even if no loss is attached to the prompt position,

$$\nabla_p L = \sum_{t \in \text{response}} \left(\frac{\partial h_t}{\partial p}\right)^\top \nabla_{h_t} \ell_t.$$

The prompt influences the response computation, so its gradient can be nonzero. A loss mask removes selected terms from the sum of losses. A stop-gradient would instead remove a dependency from the differentiation graph. An attention mask removes a dependency from the forward information flow. These are three separate interventions.

For a soft prompt, p itself is trainable while θ is fixed. The gradient still needs to propagate through the frozen model to reach p. This often saves optimizer-state memory relative to updating all weights, but it does not eliminate activation storage or backward computation through every relevant layer.

Prefix key/value parameters can enter at several layers. Their effect is local to the attention read at that layer, but later layers propagate the consequences. The number of prompt tokens, layers receiving prefixes, head dimensions, and sharing conventions determine the actual trainable parameter count.

M.3 LoRA as a bilinear parameterization

Return to lecture: Lecture 21, §21.3

Take $W_0 \in \mathbb{R}^{6\times 5}$, $A \in \mathbb{R}^{2\times 5}$, and $B \in \mathbb{R}^{6\times 2}$. The two factors have $2 \cdot 5 + 6 \cdot 2 = 22$ parameters, compared with 30 in a full update. Larger widths and small rank make the fractional saving greater. The rank bound follows because every column of BA lies in the span of the two columns of B.

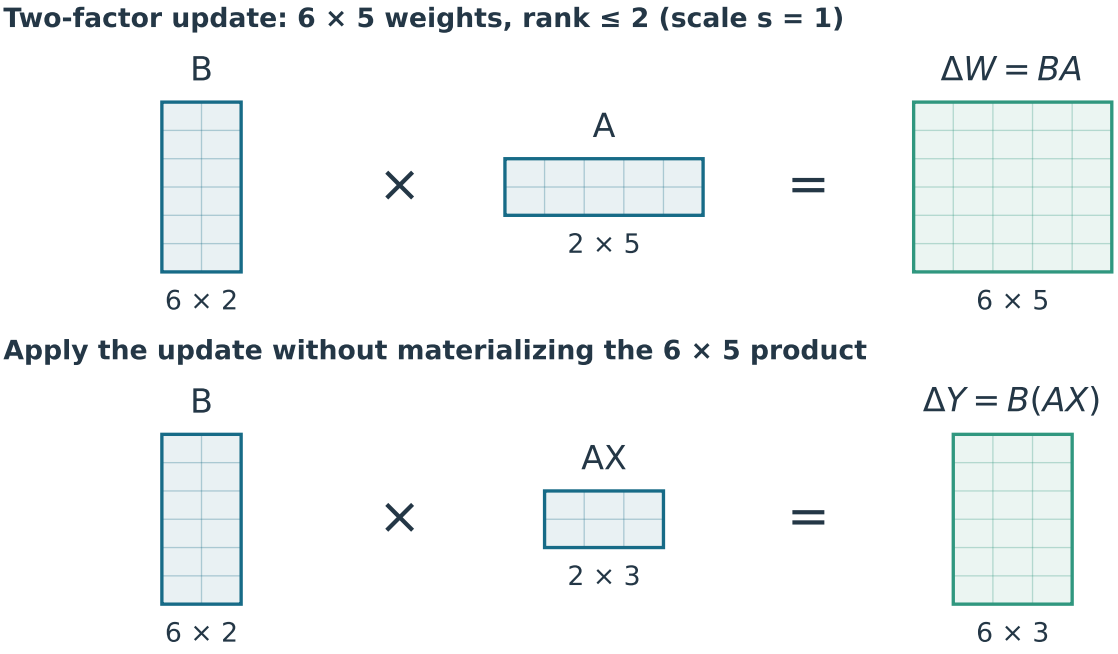


Figure M.2: LoRA factors form a narrow intermediate representation. For a batch X, associativity gives $(BA)X = B(AX)$, so the update can be applied without forming the full matrix. The frozen branch W_0X is added separately.

For $\Delta W = sBA$, the map from the two factors to the update matrix is bilinear. The loss can be convex in the product ΔW under a linear regression model while being nonconvex in (A, B). Scaling B upward and A downward preserves the product but changes gradient magnitudes.

For a batch X and output sensitivities G, the gradients are

$$\nabla_B L = sGX^\top A^\top, \qquad \nabla_A L = sB^\top GX^\top.$$

The base weight remains fixed if the method is implemented as intended. A full-gradient update $-\eta GX^\top$ is not generally the same as the matrix update induced by separate steps on A and B.

To see the induced change, write

$$(B + \Delta B)(A + \Delta A) - BA = \Delta B\, A + B\, \Delta A + \Delta B\, \Delta A.$$

Even the first-order terms depend on the current factors. At initialization with $B = 0$, only the $\Delta B\, A$ term contributes to first order. The initially random row space of A constrains the first update directions, while later factor learning can change that subspace.

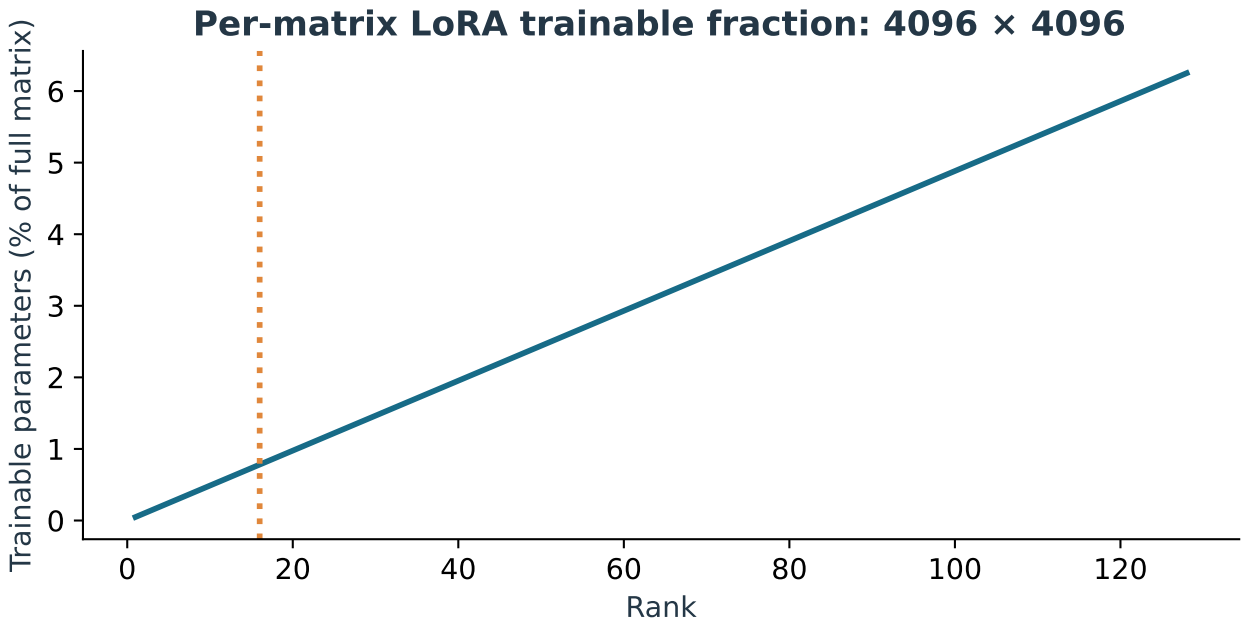


Figure M.3: LoRA parameter count grows linearly with rank for a fixed matrix shape. This fraction excludes the stored frozen base weights.

M.4 Rank, placement, and merging are separate choices

Return to lecture: Lecture 20, §20.4

Rank r limits the update to each selected matrix, not the rank of the entire nonlinear network's function change. Adapting query and value projections affects attention differently from adapting an MLP or output head. Multiple low-rank updates composed through nonlinear layers can have rich effects even though each matrix change is low rank.

Increasing r raises parameter count linearly in r but may alter the scale of the initial update if the scaling convention changes. The factor $s = \alpha/r$ is a convention, not a mathematical necessity. If comparing ranks, record whether α, s, or an update-norm statistic is held fixed.

After training, a fixed adapter can be merged into $W_0 + sBA$. Merging is an algebraic operation for that linear map. It does not automatically apply to methods that modify the forward computation nonlinearly or switch adapters dynamically. Quantized storage can introduce an additional approximation when the merged weight is requantized.

M.5 A controlled adaptation experiment

Return to lecture: Lecture 21, §21.2, Lecture 22, §22.1

Choose a base model and a fixed train/validation/test split. Compare a frozen head, a small adapter, and a full fine-tune with the same target definition. For each, report the number of trainable parameters, peak memory, processed tokens or examples, and validation-selected stopping point. Evaluate both the new task and representative old capabilities when retention matters.

If the full fine-tune receives more optimization steps or a different input format, an observed gain cannot be attributed solely to its larger trainable parameter set. If an adapter uses a much smaller learning rate without tuning, a failure may reflect optimization rather than insufficient rank. Controlled comparisons isolate one question at a time.

The old reader's broad descriptions of what is "best" are replaced here by this more explicit experiment. The course's main lesson is to reason from the actual computation and learning signal, then test the resulting hypothesis.

M.6 Choose the initial LoRA subspace deliberately

Return to lecture: Lecture 21, §21.3

The ordinary initialization $B_0 = 0$ and nonzero random A_0 preserves the pretrained function because $B_0 A_0 = 0$. It also determines the first update's subspace. Let $W \in \mathbb{R}^{m \times n}$ and write $G_W = \nabla_W \mathcal{L}$ for the full weight gradient on a representative batch at the pretrained model. For $W = W_0 + sBA$,

$$\nabla_B \mathcal{L} = s G_W A^\top, \qquad \nabla_A \mathcal{L} = s B^\top G_W.$$

At $B_0 = 0$, the gradient of A vanishes, but the gradient of B generally does not. With a simultaneous plain gradient step of size η_B on B, the first matrix update is exactly

$$\Delta W = -\eta_B s^2 G_W A_0^\top A_0.$$

Thus the initialization is a choice of which input directions the first update can use. The statement above is about plain gradient descent; an adaptive optimizer changes this formula.

The November 13 recording, approximately 51:58–55:40, discusses using singular vectors of either the initial full gradient or the pretrained weight. Here is a precise derivation of the gradient-based option. Compute $G_W = U \Sigma V^\top$ once, without updating the base weights, and set $A_0 = V_r^\top$, where V_r contains r leading right singular vectors. Then

$$A_0^\top A_0 = V_r V_r^\top, \qquad \Delta W = -\eta_B s^2 U_r \Sigma_r V_r^\top.$$

The initial update is a scaled best rank-r Frobenius approximation to the full gradient. This follows from the same truncated-SVD result used for PCA. It is a one-step statement for the batch used to compute G_W, not a guarantee of better generalization or faster training thereafter. Collecting full gradients and computing an SVD also costs memory and computation.

Taking the right singular vectors of W_0 instead prioritizes directions prominent in the pretrained operator. These need not be the directions the new task wants to change. For

$$W_0 = \begin{bmatrix} 10 & 0 \\ 0 & 1 \end{bmatrix}, \qquad G_W = \begin{bmatrix} 0 & 0 \\ 0 & 3 \end{bmatrix},$$

a rank-one weight-based initialization $A_0 = (1, 0)$ gives $G_W A_0^\top A_0 = 0$. A gradient-based initialization $A_0 = (0, 1)$ retains the entire nonzero gradient. This counterexample distinguishes the hypotheses being tested; it does not prove the second method always wins on real tasks.

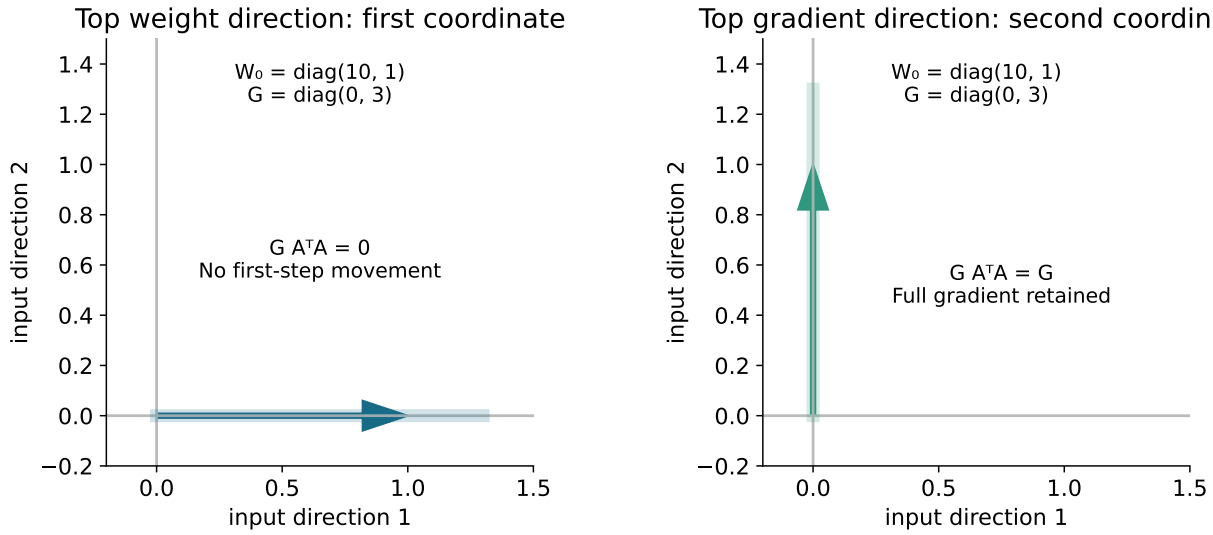


Figure M.4: The dominant pretrained direction and the dominant task-gradient direction can differ. Initializing a LoRA row space projects the first plain-gradient update onto that space.

The lecture also discusses different learning rates for the two factors. The fact that B starts at zero does not itself imply a particular optimal ratio η_B/η_A, especially with Adam's normalization. Treat such ratios as empirical hyperparameters. If both factors start nonzero, preserving the original model requires compensation, for example choosing the frozen branch as $W_0 - sB_0A_0$. Without compensation, the initialization changes the model before any learning.

M.7 Adapter composition and embedding matrices

Return to lecture: Lecture 21, §21.3

Two adapters acting on the same base matrix combine algebraically as

$$W = W_0 + s_1 B_1 A_1 + s_2 B_2 A_2.$$

The update has rank at most $r_1 + r_2$. It can be represented by concatenating the column factors and stacking the row factors:

$$\Delta W = \begin{bmatrix} s_1 B_1 & s_2 B_2 \end{bmatrix} \begin{bmatrix} A_1 \\ A_2 \end{bmatrix}.$$

Multiplying $(B_1 + B_2)(A_1 + A_2)$ instead introduces unwanted cross terms when the shapes permit that expression. Algebraic merging does not ensure that two task behaviors remain compatible; evaluate each task after composition.

Low-rank updates can also be defined for an embedding matrix or an output vocabulary projection. An embedding lookup is multiplication by a one-hot vector, so it is still a linear map in the embedding weights. Practical choices include which vocabulary rows are used, how the factors are accessed, and whether input/output weights are tied. The old reader's categorical exclusion of these matrices is too strong. Parameter savings, training memory, and inference speed should be computed for the actual implementation.

Supplement N

Learning to adapt and remembering earlier tasks

Source trail: lecture 23 and discussion 12. Meta-learning treats adaptation itself as part of a larger learning problem.

N.1 A differentiable closed-form learner

Return to lecture: Lecture 22, §22.2

Suppose a learned representation produces support features $Z \in \mathbb{R}^{n\times d}$ and targets Y. A ridge-regression head solves

$$W^* = (Z^\top Z + \lambda I)^{-1} Z^\top Y, \qquad \lambda > 0.$$

Query predictions use the learned representation and W^*. The outer loss can differentiate through this solve, teaching the representation to make simple adaptation effective. In numerical code, solve the linear system instead of explicitly constructing the inverse.

Let $M = Z^\top Z + \lambda I$ and $C = Z^\top Y$. Differentiating $MW^* = C$ yields

$$dW^* = M^{-1}(dC - dM\, W^*).$$

This identity is the clean route to the derivative. The regularizer both defines the fitted head and helps make the solve well posed. If $n \ll d$, a dual formulation using an $n \times n$ system may be cheaper.

N.2 Several inner steps require a product of Jacobians

Return to lecture: Lecture 21, §21.4

Let the inner learner perform K steps,

$$\theta^{k+1} = \theta^k - \alpha_k \nabla L_S(\theta^k), \qquad \theta^0 = \theta.$$

Define $J_k = I - \alpha_k H_S(\theta^k)$. Then the adaptation derivative is $J_{K-1} \cdots J_0$. The outer gradient is

$$\nabla_\theta L_Q(\theta^K) = J_0^\top J_1^\top \cdots J_{K-1}^\top \nabla_{\theta^K} L_Q.$$

Equal shapes can hide a reversed chain rule. With three parameters and two inner steps, the forward perturbation is $\delta\theta^2 = J_1 J_0 \delta\theta$, but the meta-gradient is $g_0 = J_0^\top J_1^\top g_2$. Both orders are shape-compatible, so dimension checking alone cannot catch an incorrect ordering. In practice, automatic differentiation can apply these Jacobian-vector or transposed-Jacobian-vector products without materializing dense parameter-by-parameter matrices.

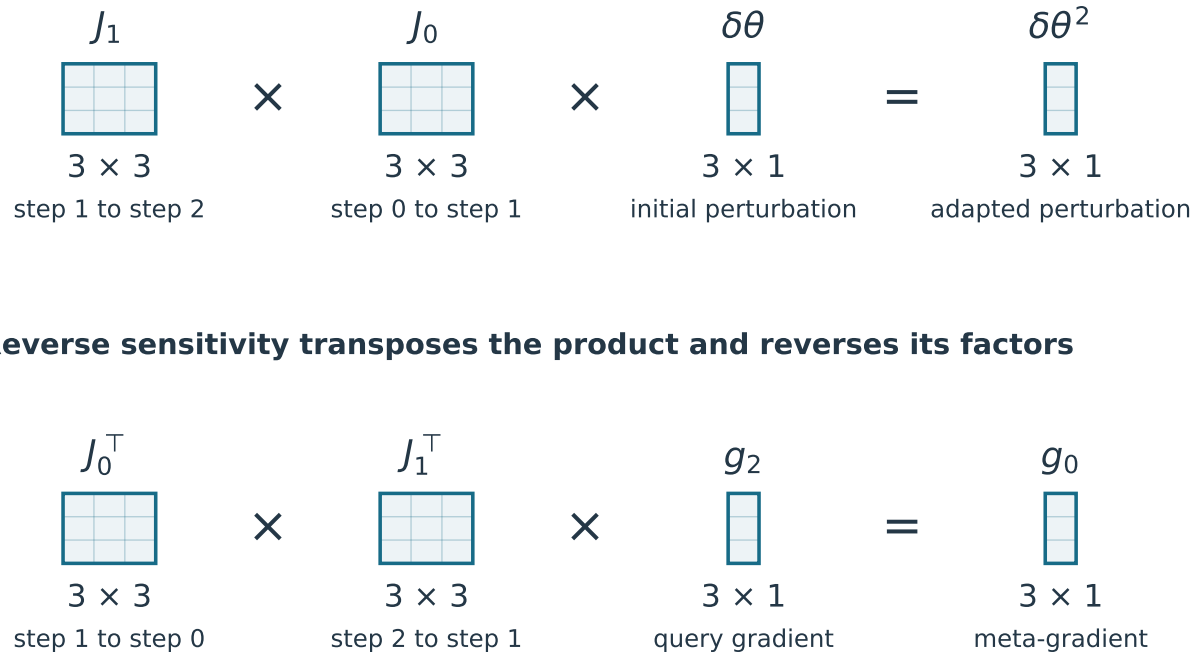


Figure N.1: The same parameter dimension appears at each inner step, but the Jacobian factors carry different time indices. Read both product actions from right to left.

The order reverses under transposition. In a nonlinear problem, the Hessian is evaluated at a different parameter vector at each step; replacing the product by one Hessian power is generally incorrect.

If learning rates are also meta-parameters, the outer loss can differentiate through them. For one step, $\partial\theta'/\partial\alpha = -\nabla L_S(\theta)$, so

$$\frac{\partial L_Q}{\partial \alpha} = -\nabla L_S(\theta)^\top \nabla L_Q(\theta').$$

A negative derivative indicates that increasing the inner step size locally improves the query loss. This ties adaptation-rate learning to alignment between support and post-adaptation query gradients.

N.3 Exact, first-order, and implicit differentiation

Return to lecture: Lecture 22, §22.2

Exact unrolled differentiation follows every inner operation. First-order MAML omits Hessian terms in the initialization derivative. It saves work but changes the gradient. The approximation is most plausible when the omitted update Jacobians remain near identity, although this is a condition to check rather than a universal guarantee.

Implicit differentiation applies when an inner solution is defined by a stationarity equation. Suppose $w^*(\phi)$ minimizes a smooth inner objective $L_S(w, \phi)$ and the relevant Hessian in w is invertible. Differentiating $\nabla_w L_S(w^*, \phi) = 0$ gives

$$\frac{dw^*}{d\phi} = -\left(\nabla^2_{ww} L_S\right)^{-1} \nabla^2_{w\phi} L_S.$$

This describes the derivative of the optimized solution, not necessarily the derivative of a finite number of optimizer steps. If the inner learner is deliberately stopped early, unrolling and implicit differentiation answer different questions.

For ridge adaptation, the stationarity equation is a linear system, making the implicit derivative especially transparent. Matrix-vector solves avoid explicitly storing a giant inverse. Conditioning still matters: a nearly singular inner problem can make the outer derivative large and unstable.

N.4 Meta-learning an initialization on a family of quadratics

Return to lecture: Lecture 22, §22.2

Let each task have a scalar optimum a_τ and loss $L_\tau(\theta) = \frac{1}{2} c_\tau (\theta - a_\tau)^2$. One inner step gives

$$\theta'_\tau - a_\tau = (1 - \alpha c_\tau)(\theta - a_\tau).$$

If query loss has the same task curvature and optimum, the meta-objective is

$$F(\theta) = \frac{1}{2} \mathbb{E}_\tau \left[c_\tau (1 - \alpha c_\tau)^2 (\theta - a_\tau)^2 \right].$$

Its optimum is a weighted average of task optima, with weights $c_\tau (1 - \alpha c_\tau)^2$, provided their expectation is positive. Tasks solved almost perfectly by the inner step receive little remaining outer loss. The best initialization is therefore shaped by the adaptation algorithm, not merely by the average task optimum.

If $\alpha c_\tau = 1$ for every task, one step solves every task independently of initialization, and the outer objective supplies no information about which initialization to choose. This solvable example shows why an adaptation procedure can make some parameters unidentifiable from the meta-objective.

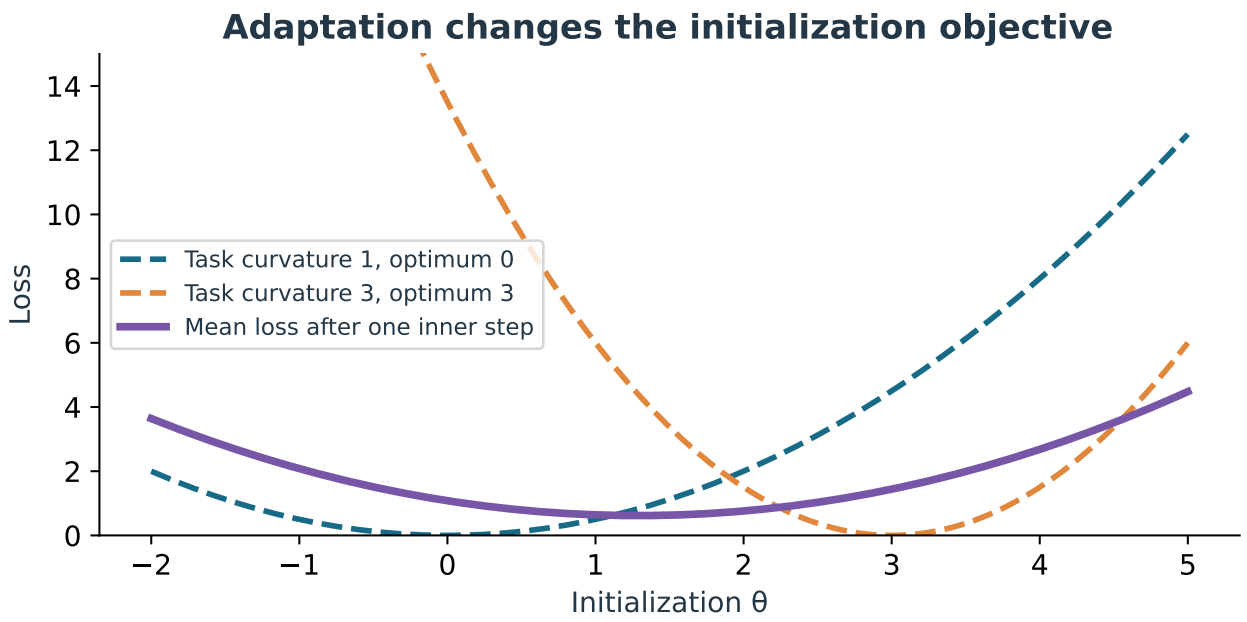


Figure N.2: The loss after adaptation weights task optima differently from the original losses. The inner learning rate is 0.2 in this illustration.

N.5 Forgetting as a multi-objective tradeoff

Return to lecture: Lecture 22, §22.3

For two tasks, a weighted joint objective is $L_A + \lambda L_B$. Its gradient combines both tasks at the current parameters. Replay approximates such a mixture using stored examples. Parameter anchoring approximates old-task protection through a penalty around earlier parameters. Distillation protects selected old predictions on selected inputs.

These procedures preserve different things. A small parameter displacement may still cause a large prediction change in a sensitive direction. Matching outputs on a small replay set may not preserve behavior elsewhere. An importance-weighted penalty can be more targeted than an isotropic penalty, but its usefulness depends on the quality of the importance approximation.

For scalar losses $L_A = \frac{1}{2}(\theta - a)^2$ and $L_B = \frac{1}{2}(\theta - b)^2$, minimizing $L_A + \lambda L_B$ gives $\theta = (a + \lambda b)/(1 + \lambda)$. Unless $a = b$, improving one task near its optimum requires accepting error on the other under this shared scalar representation. A larger model may create separate parameter directions and reduce the tradeoff, but optimization still has to find them.

Evaluation should measure both adaptation and retention across time. Reporting only the final new-task score can conceal severe forgetting; reporting only retention rewards a learner that refuses to change. The relevant object is a learning trajectory across tasks, not one isolated accuracy number.

Supplement O

Probability, information, and gradients through randomness

Source trail: discussion 12, lecture 24, and lecture 26. These tools connect supervised prediction, latent-variable models, and reward-based learning.

O.1 Joint distributions, conditioning, and expectation

Return to lecture: Lecture 23: VAE key ideas

The tools in this section supply prerequisites for entropy, variational inference, and diffusion. A joint distribution describes quantities together; a marginal forgets one quantity by summing or integrating it out; a conditional distribution fixes an observed value and renormalizes what remains. Consider this complete discrete model:

	$Y=0$	$Y=1$	$Y=2$	$P(X=x)$
$X=0$	0.30	0.15	0.05	0.50
$X=1$	0.05	0.15	0.30	0.50
$P(Y=y)$	0.35	0.30	0.35	1

For example, $P(Y=2 \mid X=1) = 0.30/0.50 = 0.60$ while $P(Y=2) = 0.35$. Conditioning changes the weights because observing X conveys information about Y. Bayes' rule follows from the two factorizations

$$p(x,y) = p(x)p(y \mid x) = p(y)p(x \mid y), \qquad p(x \mid y) = \frac{p(y \mid x)p(x)}{p(y)},$$

when the denominator is positive. For continuous variables these expressions use densities and integrals; a density may exceed one, because event probability is its integral over a region. A uniform variable on an interval of width 0.1 has density 10 and total probability one.

A likelihood holds observed data fixed and treats this density or mass as a function of model parameters. A posterior additionally combines likelihood with a prior and normalizes. For class priors $(0.2, 0.8)$ and observation likelihoods $(0.6, 0.1)$, the posterior for class 1 is $0.12/(0.12+0.08) = 0.6$.

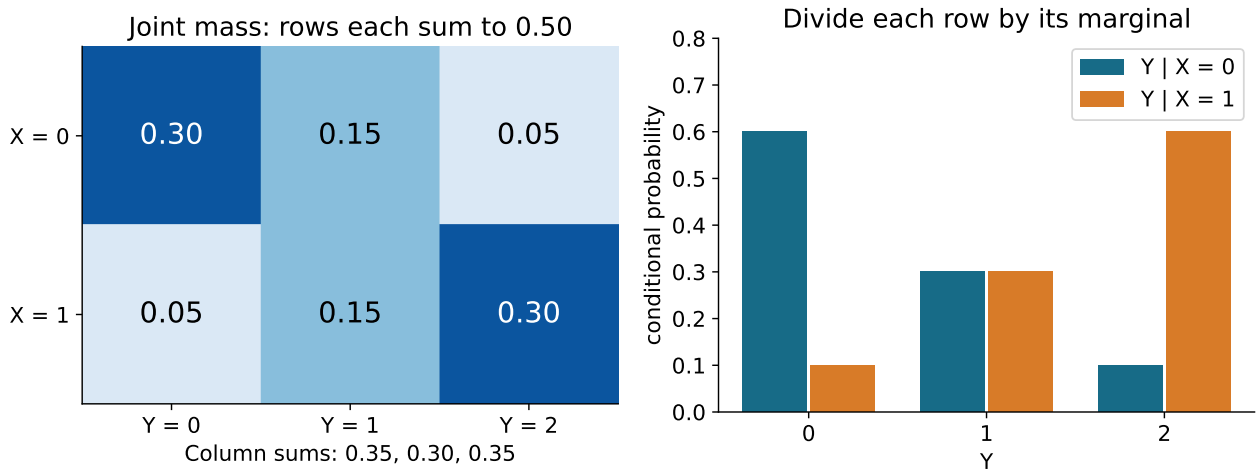


Figure O.1: Marginalization adds joint masses. Conditioning divides a selected row by its total. Both operations must preserve the appropriate normalization.

Expectation is a weighted average. Thus $\mathbb{E}[Y] = 0(0.35) + 1(0.30) + 2(0.35) = 1$. Linearity gives $\mathbb{E}[aU + bV] = a\mathbb{E}[U] + b\mathbb{E}[V]$ for integrable variables, without requiring independence. Nonlinear operations do not generally commute with expectation: $\mathbb{E}[Y^2] = 1.7$ here, whereas $(\mathbb{E}[Y])^2 = 1$.

Independence requires the whole joint distribution to factor, not merely zero covariance. If U is uniform on $\{-1, 0, 1\}$ and $V = U^2$, then $\mathbb{E}[U] = \mathbb{E}[UV] = 0$, so $\mathrm{Cov}(U, V) = 0$. Yet V is determined by U, and the variables are dependent. Replacing independence assumptions by uncorrelatedness in a nonlinear calculation is therefore unsafe.

O.2 Conditional expectation and the tower property

Return to lecture: Lecture 23: VAE key ideas

For each x, let $m(x) = \mathbb{E}[Y \mid X = x]$. In the table, $m(0) = 0.5$ and $m(1) = 1.5$. The expression $\mathbb{E}[Y \mid X]$ is the random variable $m(X)$; it becomes a number after observing $X = x$. Averaging these conditional means recovers the original mean:

$$\mathbb{E}[\mathbb{E}[Y \mid X]] = \sum_x p(x) \sum_y y p(y \mid x) = \sum_{x,y} y p(x, y) = \mathbb{E}[Y].$$

Here the answer is $0.5(0.5) + 0.5(1.5) = 1$. This is the law of total expectation, also called the tower property. More generally, conditioning successively on nested information sets leaves the coarser conditional expectation. All expectations must exist; finite tables make this automatic.

If $a(X)$ is known once X is observed, it can be taken outside the inner conditional expectation. Consequently the residual $R = Y - m(X)$ satisfies

$$\mathbb{E}[R \mid X] = 0, \qquad \mathbb{E}[R\, a(X)] = 0$$

whenever the products are integrable. This orthogonality makes conditional expectation the optimal squared-error predictor. For any square-integrable prediction $f(X)$, expand $Y - f(X) = R + m(X) - f(X)$ to obtain

$$\mathbb{E}[(Y - f(X))^2] = \mathbb{E}[R^2] + \mathbb{E}[(m(X) - f(X))^2].$$

The cross term vanishes by conditioning. The minimum occurs at $f = m$ almost surely. This is an unrestricted statistical optimum; a neural network with limited capacity, data, or optimization time need not attain it. A squared-loss denoiser therefore learns a conditional mean, which can average across multiple plausible clean outcomes rather than produce a typical sample.

O.3 Total variance and total covariance

Return to lecture: Lecture 23: VAE key ideas

Set $f(X) = \mathbb{E}[Y]$ in the previous identity:

$$\mathrm{Var}(Y) = \mathbb{E}[\mathrm{Var}(Y \mid X)] + \mathrm{Var}(\mathbb{E}[Y \mid X]).$$

The first term measures uncertainty remaining within groups defined by X; the second measures variation between their means. In the table, both conditional variances are 0.45, while the conditional means are 0.5 and 1.5. Hence total variance is $0.45 + 0.25 = 0.70$, also equal to $1.7 - 1^2$. Knowing X reduces the best achievable MSE from 0.70 to 0.45 on average. It need not reduce variance for every particular observation in every distribution.

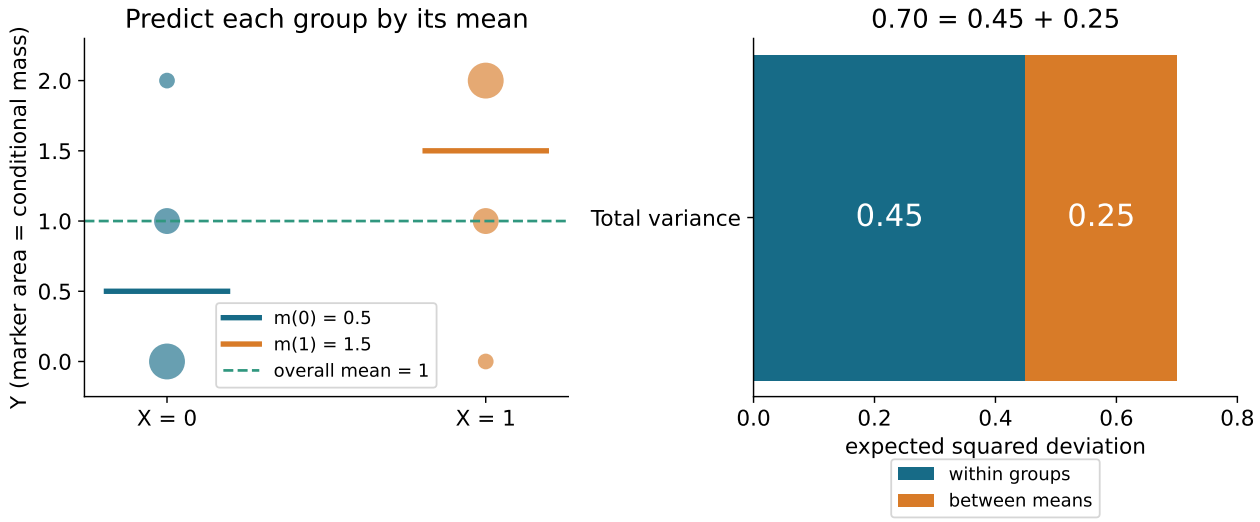


Figure O.2: Within-group uncertainty and between-group variation add to total variance. The conditional mean is an orthogonal projection in the space of square-integrable random variables.

For a vector Y, covariance uses the outer product rather than an elementwise square:

$$\mathrm{Cov}(Y) = \mathbb{E}[(Y - \mathbb{E}Y)(Y - \mathbb{E}Y)^\top].$$

The same residual decomposition gives

$$\mathrm{Cov}(Y) = \mathbb{E}[\mathrm{Cov}(Y \mid X)] + \mathrm{Cov}(\mathbb{E}[Y \mid X]).$$

Both terms are positive semidefinite. The identity also holds conditionally, for example with every expectation and covariance additionally conditioned on a noisy observation Z. In diffusion, replacing the uncertain clean image by its posterior mean drops the second covariance term; this explains why a reverse mean alone does not describe the full reverse distribution.

O.4 Jensen's inequality connects curvature to probability

Return to lecture: Lecture 23: VAE architecture and loss

If ϕ is convex and the relevant expectations are finite, then

$$\phi(\mathbb{E}[U]) \leq \mathbb{E}[\phi(U)].$$

For a differentiable convex function, the supporting-line inequality is $\phi(u) \geq \phi(a) + \phi'(a)(u - a)$. Put $a = \mathbb{E}[U]$ and average: the linear remainder has mean zero. A supporting hyperplane gives the vector version. Concavity reverses the inequality; in particular,

$$\mathbb{E}[\log U] \leq \log \mathbb{E}[U] \quad (U > 0).$$

For U equally likely to be 1 or 4, the left side is $\log 2 \approx 0.6931$, while the right side is $\log 2.5 \approx 0.9163$. The gap is not sampling error. It remains even with exact expectations. Strict convexity or concavity makes equality require an almost-surely constant variable on the relevant domain.

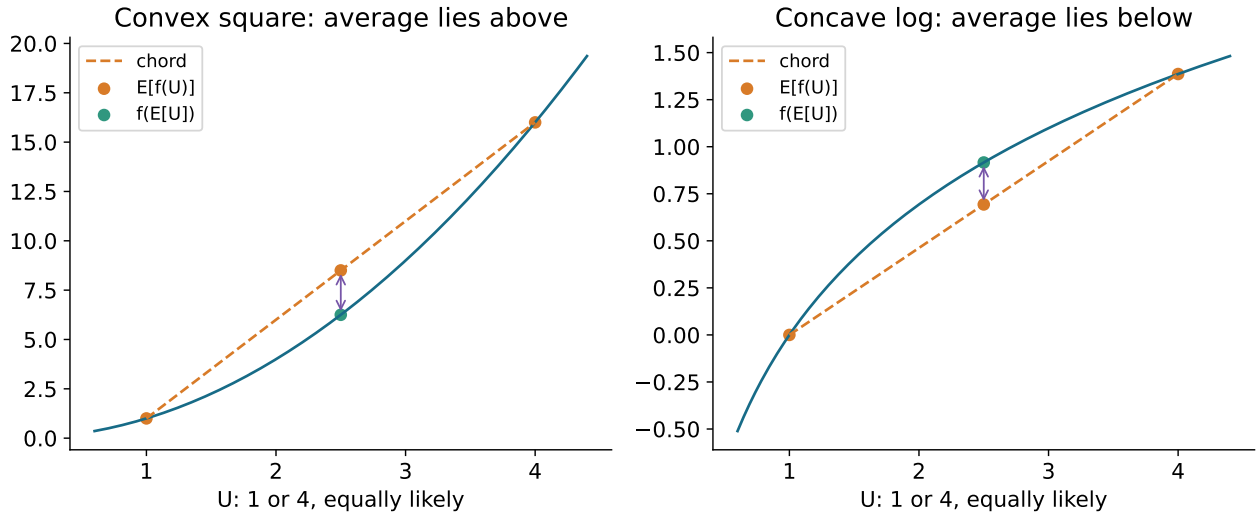


Figure O.3: Average the function values or evaluate the function at the average: curvature determines which is larger. Logarithm's concavity is the source of the evidence lower bound.

For a latent model, fix x and sample $Z \sim q(z \mid x)$. If q covers the support of $p(x, z)$, then $\mathbb{E}_q[p(x, Z)/q(Z \mid x)] = p(x)$. Jensen gives

$$\mathbb{E}_q[\log p(x, Z) - \log q(Z \mid x)] \leq \log p(x).$$

If q does not cover the entire joint support, the expectation of the ratio accounts only for the covered mass and is at most $p(x)$, so the lower bound still follows under suitable integrability. Conversely, assigning q positive mass where $p(x, z) = 0$ can make the bound $-\infty$. The later variational chapter identifies the gap exactly as a posterior KL when these expressions are well-defined.

O.5 Sample averages and differentiation under expectations

Return to lecture: Lecture 27: diffusion training

For independent, identically distributed variables U_i with finite variance σ^2, the sample mean is unbiased and has variance σ^2/N. Its standard deviation decreases as $1/\sqrt{N}$, not $1/N$. Correlated Monte Carlo samples include covariance terms:

$$\operatorname{Var}\left(\frac{1}{N}\sum_i U_i\right) = \frac{1}{N^2}\sum_{i,j}\operatorname{Cov}(U_i, U_j).$$

This is why many consecutive MCMC states need not carry as much information as equally many independent draws.

Differentiating an integral requires a justification. A useful sufficient condition is differentiability of the integrand in a neighborhood of the parameter and an integrable bound on its derivative there, with a fixed integration domain. Parameter-dependent support needs extra care. For $Z \sim \text{Uniform}(0, \theta)$, $\mathbb{E}[Z] = \theta/2$ has derivative $1/2$, but naively using only the interior score $\partial_\theta \log p_\theta(z) = -1/\theta$ gives $-1/2$: it misses the moving boundary. Reparameterizing $Z = \theta U$, $U \sim \text{Uniform}(0, 1)$, correctly gives $\mathbb{E}[U] = 1/2$. Likewise, Cauchy symmetry does not supply a finite mean or variance; check moment assumptions before using an expectation identity.

O.6 Entropy and cross-entropy have different arguments

Return to lecture: Lecture 23: KL divergence

For discrete distributions p, q,

$$H(p) = -\sum_x p(x)\log p(x), \qquad H(p, q) = -\sum_x p(x)\log q(x).$$

Entropy measures uncertainty under p. Cross-entropy evaluates how well q assigns probability to outcomes drawn from p. Their difference is

$$D_{\mathrm{KL}}(p\|q) = \sum_x p(x) \log \frac{p(x)}{q(x)} = H(p,q) - H(p).$$

If p is fixed, minimizing cross-entropy over q is equivalent to minimizing this KL divergence. In supervised learning, an empirical target distribution often places all its mass on the observed class, reducing cross-entropy to negative log probability of that class.

The convention $0 \log 0 = 0$ is understood by continuity. If $p(x) > 0$ but $q(x) = 0$, the KL divergence is infinite. This support condition matters in importance sampling and policy optimization as well as information theory.

O.7 Nonnegativity and asymmetry

Return to lecture: Lecture 23: KL divergence

Using concavity of logarithm,

$$\mathbb{E}_p \log \frac{q(X)}{p(X)} \le \log \mathbb{E}_p \frac{q(X)}{p(X)} \le 0.$$

Therefore $D_{\mathrm{KL}}(p\|q) \ge 0$, with equality when the distributions agree almost everywhere on the relevant support. KL is not a distance: it is asymmetric and does not satisfy the triangle inequality.

For $p = (0.8, 0.2)$ and $q = (0.5, 0.5)$, $D_{\mathrm{KL}}(p\|q) \approx 0.193$ nats, whereas $D_{\mathrm{KL}}(q\|p) \approx 0.223$ nats. Natural logarithms give nats; base-two logarithms give bits.

When fitting a restricted family to a multimodal target, $D_{\mathrm{KL}}(p_{\mathrm{target}}\|q)$ heavily penalizes failing to cover target mass. The reverse direction, $D_{\mathrm{KL}}(q\|p_{\mathrm{target}})$, penalizes placing model mass where the target has little density. This can encourage different compromises. "Mode covering" and "mode seeking" are useful tendencies in particular approximation problems, not guarantees for every parameterization and optimizer.

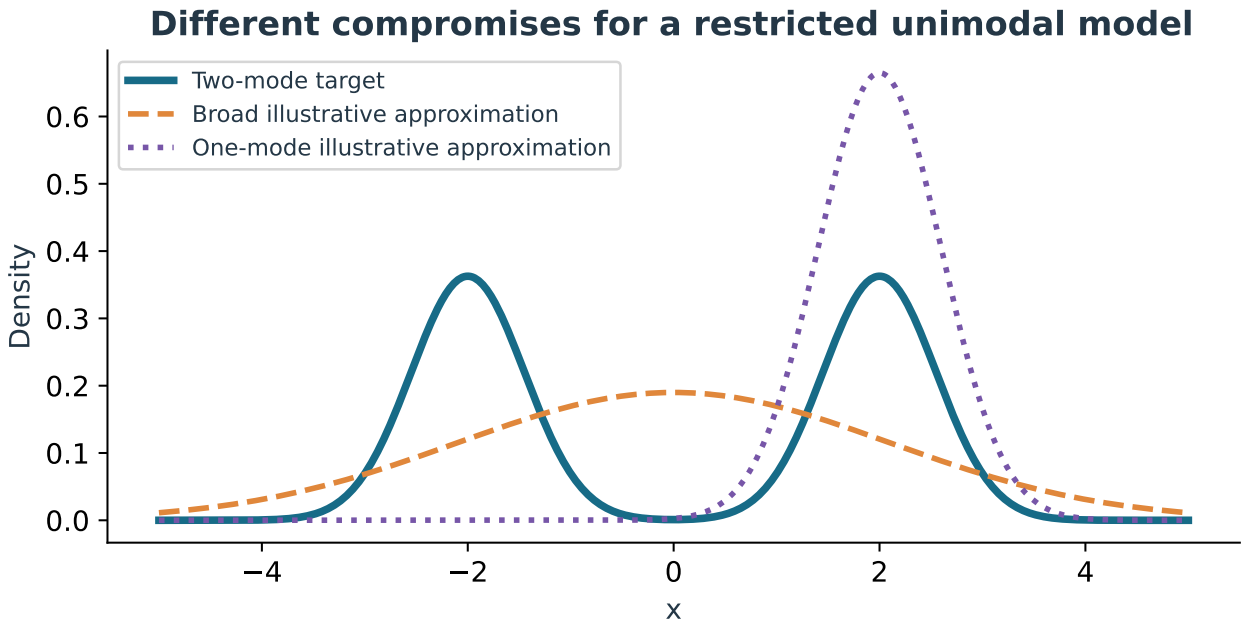


Figure O.4: Two KL directions weight errors differently. A restricted approximation must balance covering separated target modes against assigning density between them.

O.8 The score-function identity

Return to lecture: Lecture 26: RLVR recap

Consider $J(\theta) = \mathbb{E}_{z\sim p_\theta}[f_\theta(z)]$. Under conditions permitting differentiation and integration to interchange,

$$\nabla_\theta J = \mathbb{E}_{p_\theta}\left[\nabla_\theta f_\theta(z) + f_\theta(z)\nabla_\theta \log p_\theta(z)\right].$$

The second term accounts for the change in the sampling distribution. Treating a discrete sampled token as if it had an ordinary derivative with respect to its logits misses this effect.

Because $\mathbb{E}_{p_\theta}\nabla \log p_\theta(z) = 0$, subtracting a baseline independent of z from the coefficient of the score does not change the expectation. A good baseline can reduce variance. If the baseline depends on the sampled action in an arbitrary way, this cancellation generally fails.

Bernoulli example. Let $p_\theta(z = 1) = \sigma(\theta)$ and reward $r(z) = z$. The exact derivative of expected reward is $p(1-p)$. The score is $z-p$, so $\mathbb{E}[z(z-p)] = p(1-p)$. A sampled estimator can be noisy even though its expectation is correct.

O.9 Reparameterization moves randomness outside the parameters

Return to lecture: Lecture 23: reparameterization

Two estimators can target the same derivative. Let $z = \mu + \epsilon$ with $\epsilon \sim \mathcal{N}(0, 1)$ and $J(\mu) = \mathbb{E}[z^2] = \mu^2 + 1$. The exact derivative is 2μ. A pathwise sample gives $2z$; a score-function sample gives $z^2(z - \mu)$. Both average to 2μ, but their sample variances differ. At $\mu = 0$, the variances are 4 and $\mathbb{E}[z^6] = 15$, respectively. This example illustrates a variance difference, not a universal ranking. The pathwise estimator requires an appropriate differentiable sampling map and loss; score-function estimation can also handle discrete samples under its regularity assumptions.

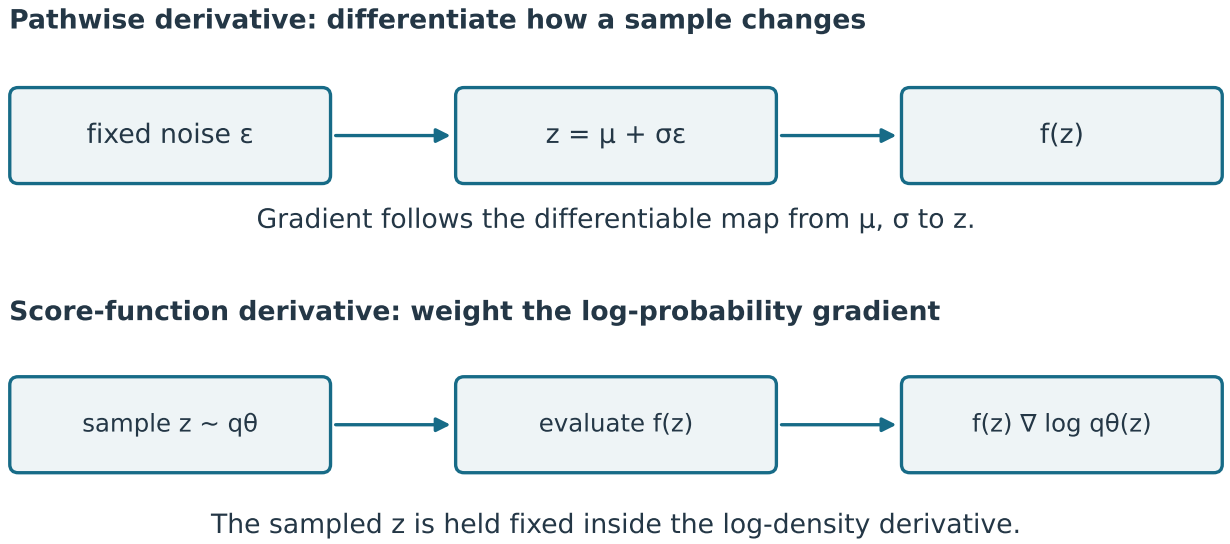


Figure O.5: Reparameterization differentiates a sample-producing computation. Score-function estimation differentiates a log probability with the sampled outcome held fixed. They follow different derivative paths.

If a sample can be written $z = g_\theta(\epsilon)$ with ϵ drawn from a parameter-independent distribution, then

$$\nabla_\theta \mathbb{E}_\epsilon f_\theta(g_\theta(\epsilon)) = \mathbb{E}_\epsilon \nabla_\theta f_\theta(g_\theta(\epsilon)).$$

For a diagonal Gaussian,

$$z = \mu_\theta + \sigma_\theta \odot \epsilon, \qquad \epsilon \sim \mathcal{N}(0, I).$$

Hold the random draw ϵ fixed while backpropagating through μ and σ. This provides a pathwise derivative. It often has lower variance than a score estimator for smooth objectives, but it relies on an appropriate differentiable sampling representation. A discrete categorical draw does not admit this simple Gaussian construction.

O.10 Importance sampling and the meaning of stop-gradient

Return to lecture: Lecture 26: RLVR recap

If q covers the support of p,

$$\mathbb{E}_p[f(z)] = \mathbb{E}_q\left[\frac{p(z)}{q(z)} f(z)\right].$$

The ratio corrects the sampling distribution. Very large ratios can create high variance; clipping reduces extreme weights but generally introduces bias. Renormalizing sampled weights also changes the finite-sample estimator.

A stop-gradient operator, written $\mathrm{sg}(a)$, has forward value a and derivative zero. The surrogate

$$\mathbb{E}_{z\sim q}\left[\mathrm{sg}\left(\frac{p_\theta(z)}{q(z)} r(z)\right) \log p_\theta(z)\right]$$

has gradient $\mathbb{E}_q[(p_\theta/q) r \nabla \log p_\theta]$ when the sample distribution is held fixed. Without stop-gradient, differentiating the ratio contributes an additional term. Two expressions with identical numerical values at the current parameters can therefore implement different update rules if their differentiation graphs differ. This distinction will be essential when reading the lecture's clipped policy update.

O.11 Cross-entropy from maximum likelihood

Return to lecture: Lecture 23: VAE architecture and loss

For independent observations, likelihood multiplies and log-likelihood adds. A categorical model with probabilities $q_\theta(k \mid x)$ gives empirical negative log-likelihood

$$-\frac{1}{N}\sum_i \log q_\theta(y_i \mid x_i).$$

For a soft target distribution p_i, the corresponding expected loss is $-\sum_k p_i(k) \log q_\theta(k \mid x_i)$. Its derivative with respect to logits is $q_\theta - p_i$ when p_i sums to one.

For Gaussian regression with fixed variance σ^2, negative log-likelihood is squared error divided by $2\sigma^2$ plus a constant. If the variance is also learned, the log-variance term is no longer constant. Omitting it would allow a model to drive the normalized residual term down by increasing variance without penalty.

O.12 Differential entropy and KL under coordinate changes

Return to lecture: Lecture 23: KL divergence

For a continuous density, differential entropy is $h(p) = -\int p(x) \log p(x)\, dx$. It can be negative and depends on coordinate units. If $Y = aX$ with $a \neq 0$ in one dimension, then $h(Y) = h(X) + \log |a|$.

KL divergence is different. When both distributions are transformed by the same invertible smooth coordinate map, the Jacobian factors cancel in the density ratio, so KL is unchanged. This makes KL a comparison between distributions that does not depend on merely renaming coordinates in the same way for both.

For Gaussians $p = \mathcal{N}(\mu_p, \sigma_p^2)$ and $q = \mathcal{N}(\mu_q, \sigma_q^2)$,

$$D_{\mathrm{KL}}(p\|q) = \log \frac{\sigma_q}{\sigma_p} + \frac{\sigma_p^2 + (\mu_p - \mu_q)^2}{2\sigma_q^2} - \frac{1}{2}.$$

This formula follows by taking expectations of the two quadratic log densities under p. Swapping p and q changes both the variance denominator and the logarithm, making asymmetry explicit.

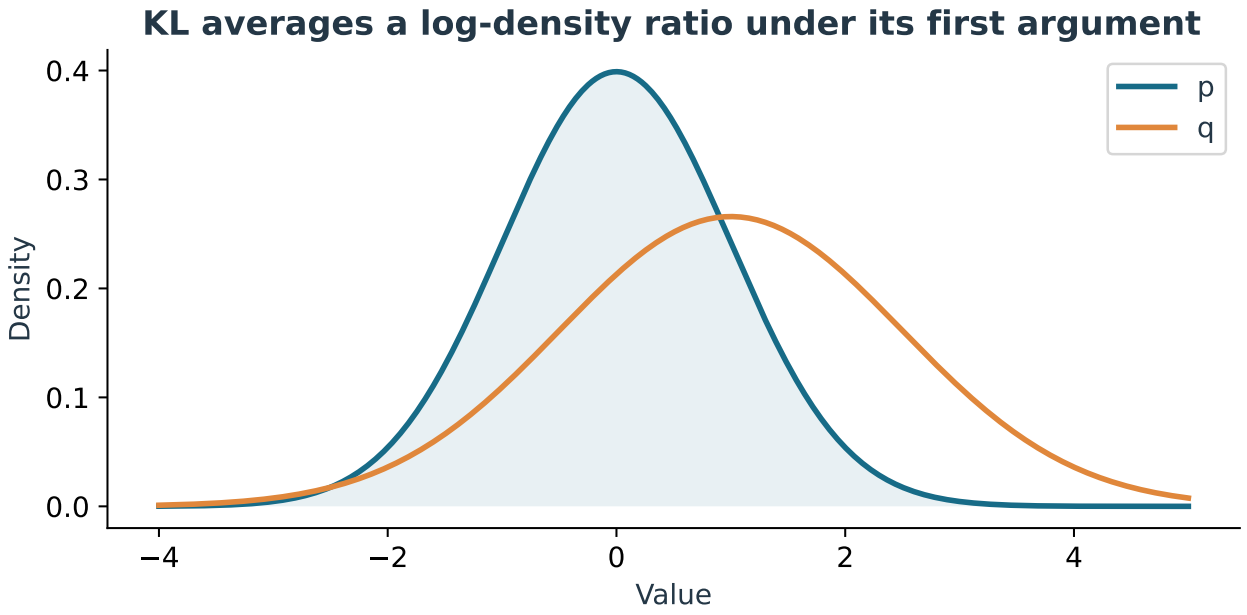


Figure O.6: A Gaussian KL compares log densities, averaging under the first distribution. Swapping the distributions changes the weighting.

O.13 Variance reduction for score-function estimators

Return to lecture: Lecture 26: RLVR recap

For a scalar parameter and score $s(z) = \partial_\theta \log p_\theta(z)$, the estimator $(r(z)-b)s(z)$ has the same mean for any constant baseline b. Minimizing its second moment over b gives

$$b^* = \frac{\mathbb{E}[r(z)s(z)^2]}{\mathbb{E}[s(z)^2]},$$

assuming the denominator is positive. The variance-minimizing baseline is not necessarily the mean reward: outcomes with large score magnitude receive more weight in this criterion.

For vector parameters and squared Euclidean estimator norm, the analogous baseline uses $\|s(z)\|^2$. A learned value baseline is a practical approximation conditioned on the state or prompt. It can reduce variance while preserving the intended expectation when its action dependence is handled correctly.

Reparameterization and score-function estimators can be compared on $z \sim \mathcal{N}(\mu, 1)$ and $f(z) = z^2$. The true derivative is 2μ. The pathwise estimator is $2(\mu + \epsilon)$, while a score estimator is $z^2(z - \mu)$. Both have the correct expectation under suitable integrability, but their variances differ. This illustrates why an unbiased estimator can still be impractical when its noise is large.

O.14 Importance weighting, support, and effective sample size

Return to lecture: Lecture 26: RLVR recap

If samples come from q but the desired expectation is under p, use weights $w = p/q$ on the support of p. If q assigns zero probability where p is positive, no finite reweighting of samples from q can recover those missing outcomes. Support is a structural condition, not a numerical nuisance.

Even with full support, a few enormous weights can dominate an estimate. A commonly used descriptive quantity is

$$N_{\text{eff}} = \frac{(\sum_i w_i)^2}{\sum_i w_i^2}.$$

Equal positive weights give $N_{\text{eff}} = N$; one dominant weight makes it near one. This is a diagnostic of weight concentration, not a guarantee that an estimator has the same behavior as that many independent target samples.

Clipping weights, mixing proposals, or drawing data from a policy closer to the target can improve numerical behavior. Each choice changes either the estimator, the sampling cost, or both. In policy optimization, this is why rollout freshness, ratio definitions, and clipping rules are integral parts of the algorithm.

Supplement P

Variational autoencoders and latent-variable models

Source trail: lecture 23, lecture 24, and discussion 13 question 2. A VAE makes an encoder-decoder construction into a probabilistic model that can generate new observations.

P.1 Derive the evidence lower bound

Return to lecture: Lecture 23: VAE architecture and loss

For any suitable $q_\phi(z \mid x)$,

$$\begin{aligned}\log p_\theta(x) &= \log \mathbb{E}_{q_\phi}\left[\frac{p_\theta(x,z)}{q_\phi(z \mid x)}\right] \\ &\geq \mathbb{E}_{q_\phi}\left[\log p_\theta(x,z) - \log q_\phi(z \mid x)\right] \\ &= \mathbb{E}_{q_\phi} \log p_\theta(x \mid z) - D_{\mathrm{KL}}(q_\phi(z \mid x) \| p(z)).\end{aligned}$$

The inequality is Jensen's inequality. The final expression is the evidence lower bound, or ELBO. More precisely,

$$\log p_\theta(x) - \mathrm{ELBO}(x) = D_{\mathrm{KL}}(q_\phi(z \mid x) \| p_\theta(z \mid x)).$$

The bound is tight exactly when the approximate posterior equals the true posterior. Notice the two different KL terms: the training objective contains a KL to the **prior**, while the gap in the bound is a KL to the **true posterior**. Substituting one for the other obscures the derivation.

P.2 A linear-Gaussian model as a solvable reference

Return to lecture: Lecture 22, §22.4, Lecture 23, §23.1

Let $z \sim \mathcal{N}(0, I)$ and $x = Az + \epsilon$, with $\epsilon \sim \mathcal{N}(0, \sigma^2 I)$. Then $x \sim \mathcal{N}(0, AA^\top + \sigma^2 I)$. The posterior is Gaussian with

$$\Sigma_{z|x} = (I + A^\top A/\sigma^2)^{-1}, \qquad \mu_{z|x} = \Sigma_{z|x} A^\top x/\sigma^2.$$

This model provides a useful check on the general machinery: the posterior mean is a regularized linear inverse, and posterior covariance quantifies uncertainty rather than returning only one code. A nonlinear VAE replaces the linear maps with learned networks while preserving the distinction between a generative model and approximate inference.

P.3 Derive the Gaussian posterior by completing the square

Return to lecture: Lecture 23: VAE key ideas

For $z \sim \mathcal{N}(0, I)$ and $x \mid z \sim \mathcal{N}(Az, \sigma^2 I)$, the posterior log density, ignoring terms independent of z, is

$$-\frac{1}{2} z^\top z - \frac{1}{2\sigma^2}(x - Az)^\top (x - Az).$$

Expanding the second quadratic and collecting terms gives

$$-\frac{1}{2} z^\top \left(I + \frac{A^\top A}{\sigma^2} \right) z + z^\top \frac{A^\top x}{\sigma^2} + \text{constant}.$$

A Gaussian log density has the form $-\frac{1}{2}(z-\mu)^\top \Sigma^{-1}(z-\mu) + C$. Matching coefficients yields

$$\Sigma^{-1} = I + A^\top A/\sigma^2, \qquad \Sigma^{-1}\mu = A^\top x/\sigma^2.$$

Covariance and precision have named spaces. For a five-dimensional observation generated from a two-dimensional latent variable, the decoder A is 5×2. With observation-noise covariance $R \succ 0$, $A^\top R^{-1} A$ is a 2×2 contribution to latent precision. A standard-normal prior adds I_2, giving $\Sigma_{\text{post}}^{-1} = I_2 + A^\top R^{-1} A$ and $\mu_{\text{post}} = \Sigma_{\text{post}} A^\top R^{-1} x$. The isotropic case above sets $R = \sigma^2 I_5$. A numerical implementation should solve the relevant linear systems rather than explicitly invert every matrix.

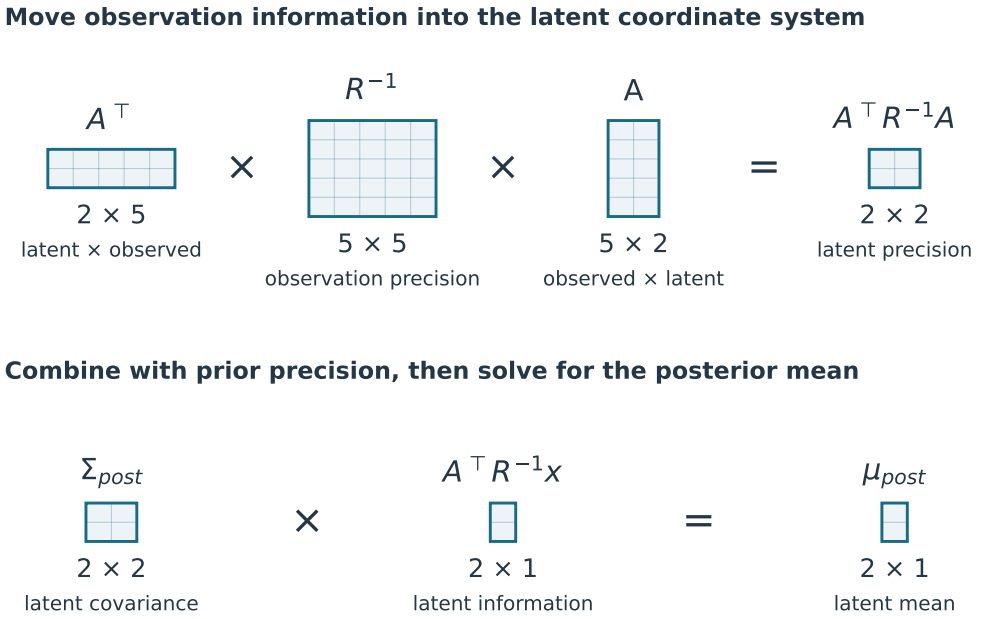


Figure P.1: A rectangular decoder links two covariance spaces. Posterior precision lives in latent space, not observation space.

This recovers the posterior formulas in the main chapter. The prior contributes identity precision; the likelihood contributes $A^\top A/\sigma^2$. Information adds naturally in precision coordinates.

For a scalar decoder $x = 2z + \epsilon$ with noise variance 1 and observed $x = 3$, posterior variance is $1/5$ and posterior mean is $6/5$. The observation alone suggests $z = 1.5$, while the standard-normal prior pulls the posterior mean toward zero. The result is a distribution, not just a regularized point estimate.

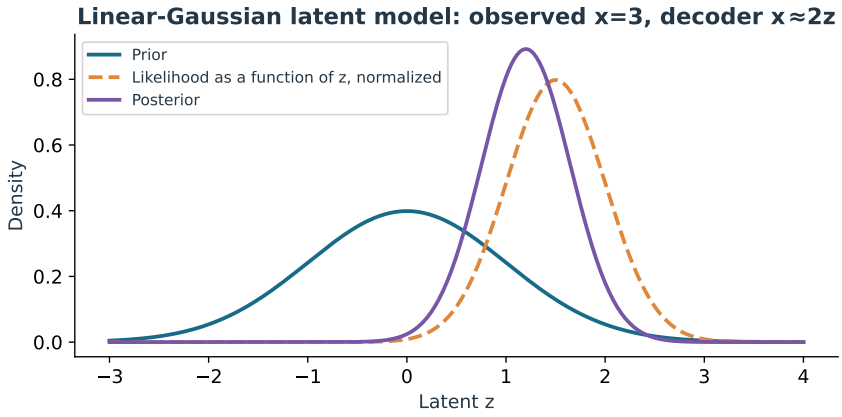


Figure P.2: In a solvable latent model, the posterior combines prior precision with likelihood precision. This is the calculation a nonlinear encoder approximates.

P.4 The ELBO is exact likelihood minus an inference gap

Return to lecture: Lecture 23: VAE architecture and loss

Starting from Bayes' rule,

$$\log p_\theta(x, z) = \log p_\theta(z \mid x) + \log p_\theta(x).$$

Subtract $\log q_\phi(z \mid x)$ and take expectation under q_ϕ:

$$\mathrm{ELBO}(x) = \log p_\theta(x) - D_{\mathrm{KL}}(q_\phi(z \mid x) \| p_\theta(z \mid x)).$$

This derivation complements Jensen's inequality by displaying the exact gap. Improving the ELBO can improve the generative likelihood, improve approximate inference, or both. A restricted variational family may leave a nonzero gap even after optimization.

A diagonal Gaussian encoder cannot represent every correlated or multimodal posterior. The decoder can partly adapt so that the posterior is easier for the encoder, because both are trained together. This interaction is useful but means the variational approximation can influence the learned generative model, not merely provide a neutral estimate of a fixed model's posterior.

P.5 Amortized inference shares computation across examples

Return to lecture: Lecture 23: VAE architecture and loss

Without an encoder network, one could optimize a separate approximate posterior for each observation. Amortized inference instead uses $q_\phi(z \mid x)$ to predict posterior parameters in one forward pass. The shared encoder makes inference fast on new examples but constrains how independently each example's posterior can be fitted.

The total discrepancy from the best possible posterior approximation can be separated conceptually into a family limitation and an amortization limitation. The first comes from restricting q to a chosen family; the second comes from using one trained mapping to produce all those distributions. This distinction helps explain why optimizing local variational parameters can sometimes improve a bound even when the decoder is fixed.

Sampling one latent per example gives an unbiased Monte Carlo estimate of the reconstruction expectation and its pathwise gradient under suitable conditions. More latent samples reduce some Monte Carlo noise but increase computation. The analytic Gaussian KL term need not be sampled when its closed form is available.

P.6 Gradients with a log-variance parameterization

Return to lecture: Lecture 23: reparameterization

Let $\ell = \log \sigma^2$ and $z = \mu + e^{\ell/2} \odot \epsilon$. If the decoder loss sends gradient g_z, the pathwise contributions are

$$\nabla_\mu L_{\text{recon}} = g_z, \qquad \nabla_\ell L_{\text{recon}} = \frac{1}{2} g_z \odot e^{\ell/2} \odot \epsilon.$$

For KL to a standard normal,

$$\nabla_\mu D_{\text{KL}} = \mu, \qquad \nabla_\ell D_{\text{KL}} = \frac{1}{2}(e^\ell - 1).$$

The full encoder gradient combines these terms and then propagates through the networks producing μ and ℓ. Parameterizing variance directly without ensuring positivity can produce invalid distributions. Very large positive or negative log variances can also create numerical difficulties; implementation choices should preserve the intended objective as closely as possible.

P.7 Reconstruction, interpolation, and unconditional sampling

Return to lecture: Lecture 23: VAE architecture and loss

Matching the aggregate prior does not identify an informative code. Define $q(x, z) = p_{\text{data}}(x) q(z \mid x)$ and $q(z) = \mathbb{E}_{p_{\text{data}}} q(z \mid x)$. Adding and subtracting $\log q(z)$ gives

$$\mathbb{E}_x D_{\text{KL}}(q(z \mid x) \| p(z)) = \mathbb{E}_{q(x,z)} \log \frac{q(z \mid x)}{q(z)} + \mathbb{E}_{q(z)} \log \frac{q(z)}{p(z)} \tag{P.1}$$

$$= I_q(X; Z) + D_{\text{KL}}(q(z) \| p(z)). \tag{P.2}$$

For the figure, let x be equally likely to be $-$ or $+$. In the informative construction, its two posteriors are a standard normal restricted to the negative and positive half-lines, each renormalized. Their aggregate is exactly standard normal, but $I_q(X; Z) = \log 2$ because the sign determines x. In the collapsed construction both posteriors equal the standard-normal prior, so $I_q(X; Z) = 0$. Thus identical aggregate latent distributions can hide very different information content. These are illustrative distributions, not necessarily members of a diagonal-Gaussian encoder family. Reconstruction quality and the decoder's reliance on the latent must also be examined.

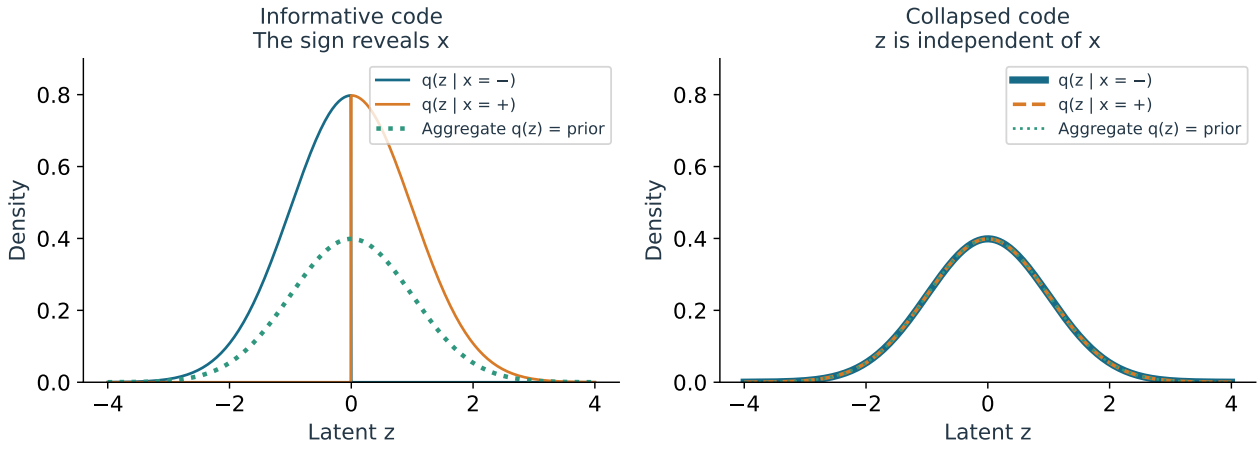


Figure P.3: An aggregate latent distribution can match the prior whether the code carries information or carries none. The conditional distributions, not just a histogram of all latent samples, reveal the difference.

A reconstruction encodes an observed x, samples or selects a latent from $q(z \mid x)$, and decodes it. Unconditional generation samples z from the prior. Interpolation moves between latent representations and decodes intermediate points. These procedures probe different parts of the model.

A smooth latent interpolation does not prove that every prior sample is realistic. Good reconstructions do not prove that the learned aggregate posterior equals the prior. A low KL can indicate useful regularity, but can also accompany a decoder that ignores its latent. Evaluate generation, reconstruction, and latent use separately.

The linear-Gaussian reference model shows what an exact latent-variable calculation looks like. The nonlinear VAE retains the same probabilistic roles while replacing tractable inference with a learned approximation. This is the essential conceptual step: an encoder is an inference model, and a decoder plus prior is a generative model.

Supplement Q

Decoding and computation at inference time

Source trail: discussion 11, lecture 24, and lecture 25. A trained probability distribution does not specify a unique procedure for choosing an answer.

Q.1 The exact next-token marginal of a powered sequence model

Return to lecture: Lecture 24, §24.5

For a prefix u and a candidate next token a, define the powered continuation sum

$$Z_\alpha(ua) = \sum_v p(v \mid ua)^\alpha,$$

where v ranges over complete remaining suffixes under the specified stopping convention. Then the next-token distribution of the globally powered sequence model is

$$q_\alpha(a \mid u) = \frac{p(a \mid u)^\alpha Z_\alpha(ua)}{\sum_b p(b \mid u)^\alpha Z_\alpha(ub)}.$$

Tokenwise temperature drops the continuation factor $Z_\alpha(ua)$. The missing factor depends on how concentrated the future distribution is after choosing a. It is generally expensive to compute because it sums over many possible continuations.

For $\alpha = 1$, every normalized continuation distribution has $Z_1(ua) = 1$, so the formula reduces to the original model. For $\alpha > 1$, a concentrated continuation has a larger power sum than a diffuse one. Global powering can therefore prefer a branch whose future is highly predictable, even when its first token was less likely.

In the two-branch example, the powered leaf sum for branch A is $0.306^2 + 0.294^2 = 0.180072$, while branch B gives $0.396^2 + 0.004^2 = 0.156832$. The normalized A mass is $0.180072/0.336904 \approx 0.53449$. Tokenwise powering gives about 0.69231. Writing the four leaves explicitly prevents the normalization error that is easy to make when reasoning only from first-token probabilities.

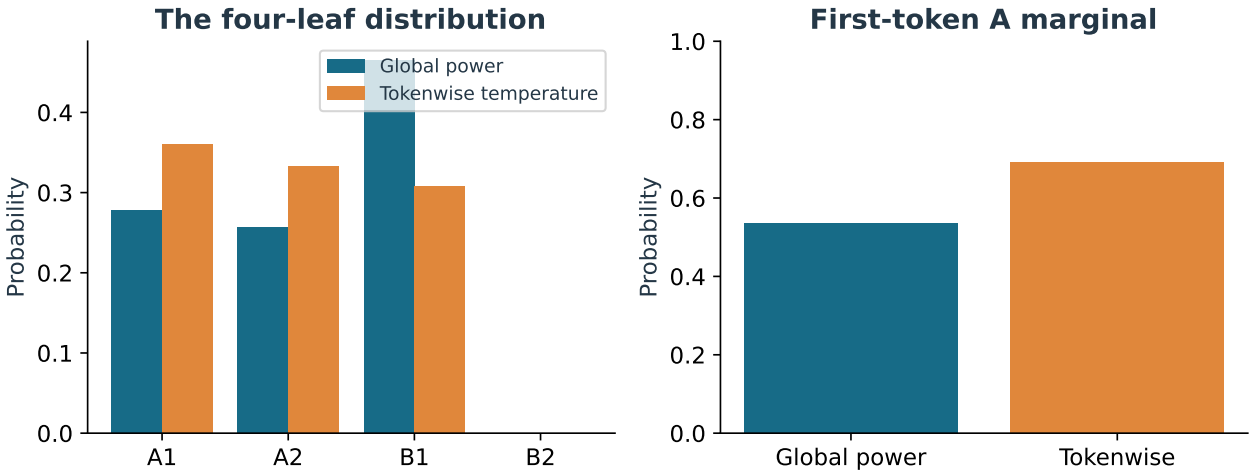


Figure Q.1: Global sequence powering and tokenwise temperature assign different probabilities even in a four-leaf tree. The continuation normalizers cause the difference.

Q.2 Detailed balance explains Metropolis-Hastings

Return to lecture: Lecture 24, §24.5

A complete two-state chain. Suppose the target probabilities are $q(A) = 0.8$ and $q(B) = 0.2$, and every proposal switches to the other state. Metropolis–Hastings accepts $A \to B$ with probability $0.2/0.8 = 0.25$ and $B \to A$ with probability one. The transition matrix, with rows denoting the current state, is

$$P = \begin{bmatrix} 0.75 & 0.25 \\ 1 & 0 \end{bmatrix}.$$

At stationarity the two directional flows both equal 0.2: $0.8 \cdot 0.25 = 0.2 \cdot 1$. The rejected proposal supplies the $A \to A$ self-transition. Starting from a fixed state does not instantly produce the target distribution, and successive samples remain dependent.

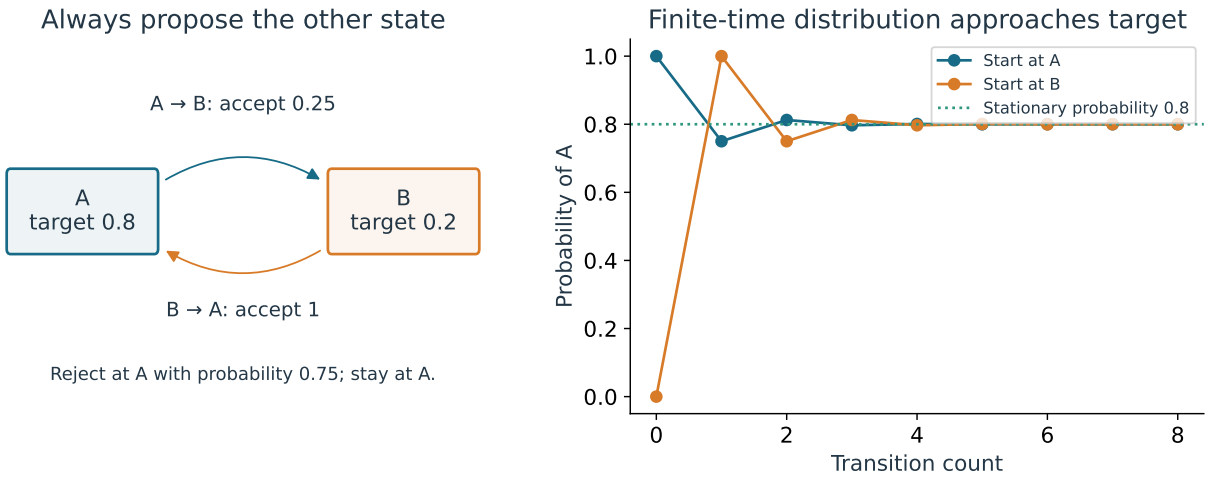


Figure Q.2: Acceptance corrects a proposal distribution toward the target. Detailed balance concerns probability flow, while finite-time convergence is a separate question.

For target mass $q(x)$ and proposal $r(x' \mid x)$, define acceptance probability

$$a(x, x') = \min\left(1, \frac{q(x')r(x \mid x')}{q(x)r(x' \mid x)}\right).$$

The probability flow between distinct states is

$$q(x)r(x' \mid x)a(x, x') = \min\{q(x)r(x' \mid x), q(x')r(x \mid x')\},$$

which is symmetric in x, x'. This is detailed balance, and it establishes stationarity when the transition kernel is properly defined. Rejections retain the current state and supply the remaining probability mass on the diagonal.

The normalizing constant of q cancels from the ratio. That is why an unnormalized target such as $p(x)^\alpha$ can be used. However, the proposal must connect the relevant state space for the chain to explore it, and finite-time samples can be strongly correlated. A stationary-distribution proof is not a mixing-time guarantee.

If a proposal redraws only a suffix, the prefix is held fixed and proposal probabilities are conditional suffix probabilities. The reverse move must be evaluated under the same proposal rule. Ignoring this asymmetry can produce a chain with the wrong stationary distribution even if its samples look plausible.

Q.3 Beam search with explicit termination

Return to lecture: Lecture 24, §24.4

A beam maintains partial hypotheses and their cumulative log probabilities. At each step, expand candidates, add the next-token log probability, and retain the highest-scoring allowed

candidates. Completed hypotheses ending in EOS should be handled according to a specified stopping rule rather than repeatedly extended as if EOS were an ordinary token.

With raw log probability, every added token contributes a nonpositive quantity. This creates a length preference that depends on the model's EOS probabilities. Dividing by length or adding a length penalty changes the scoring objective. Such adjustments may improve task metrics, but the resulting search is no longer simply maximizing the original sequence probability.

A finite beam can discard a prefix that later would have led to the best full sequence. Increasing beam width reduces some search approximation but can also emphasize a model's miscalibrated preference for undesired outputs. Search quality and model quality are distinct. Better optimization of a flawed score can worsen the external task metric.

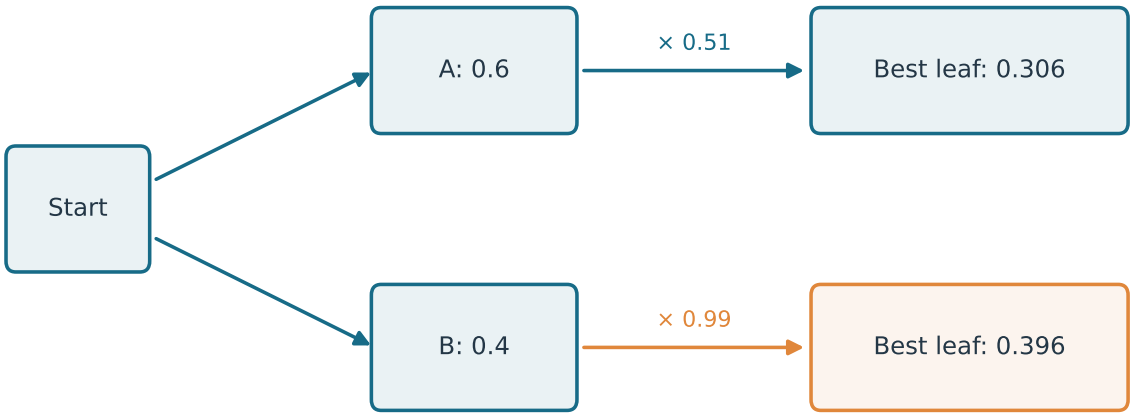


Figure Q.3: A locally preferred first token can lead to a less probable complete sequence. Beam search preserves multiple partial candidates, subject to its beam width.

Q.4 Best-of-many requires a selection mechanism

Return to lecture: Lecture 24, §24.3

If k independent samples each succeed with probability p, at least one succeeds with probability $1-(1-p)^k$. A real system still needs to identify a successful sample. A perfect verifier turns availability into success; an imperfect verifier can select a high-scoring wrong answer.

For a binary task with independent answers and correctness probability above one-half, majority vote can improve accuracy as the number of votes grows. But answers may be correlated, may have multiple distinct wrong alternatives, and may not be easily normalized into equivalence classes. The independence and aggregation assumptions should be made explicit.

A verifier trained on the same distribution as the generator can share its errors. A reward model can prefer superficial features associated with good answers rather than actual correctness. Testing selection on adversarial or shifted examples helps determine whether more sampling is improving reasoning or merely exploiting a ranking signal.

Q.5 An evaluation table that separates the mechanisms

Return to lecture: Lecture 24, §24.2

A useful comparison reports the base model, decoding rule, number of sampled tokens, number of candidates, verifier computation, and final task score. Equal candidate counts are not equal compute if one method generates much longer responses. Equal generated-token counts are not equal latency when methods have different parallelism.

Compare proposed sampling methods with simpler baselines at a comparable budget. High model probability need not mean correctness; include ordinary sampling, prompt changes, and a specified ranking rule when relevant.

Q.6 Why cut-specific power sampling preserves its target

Return to lecture: Lecture 24, §24.5

Source trail. The algorithm displayed in lecture 25, page 3 grows a sequence in blocks and applies suffix-resampling Metropolis–Hastings steps. The associated paper is Karan and Du, *Reasoning with Sampling: Your Base Model Is Smarter Than You Think* (2025), section 4.3. The derivation below spells out the acceptance rule and its limits using the course algorithm.

Fix a prompt c, a finite vocabulary, and a final generated length T. Use a specified fixed-length state space; an EOS convention must be included consistently if termination is modeled. At current horizon n, define

$$q_n(x_{1:n} \mid c) = \frac{p(x_{1:n} \mid c)^\alpha}{Z_n(c)}, \qquad \alpha > 0.$$

Use the lecture's cut-and-resample algorithm. For cut m, write the retained prefix as u and the old/new suffixes as v, v'. A general autoregressive proposal r has acceptance probability

$$\min\left(1, \frac{p(x' \mid c)^\alpha r(v \mid c, u)}{p(x \mid c)^\alpha r(v' \mid c, u)}\right).$$

For a fixed cut, the unchanged prefix partitions the state space. Metropolis–Hastings preserves the conditional target in each partition and hence preserves q_n overall. A state-independent uniform mixture of these valid cut-specific kernels also preserves q_n. This formulation avoids confusing the conditional proposal for a selected cut with the marginal proposal summed over all possible cuts.

A proposal mismatch that breaks the simplified rule

If the proposal uses temperature, a different model, or another distribution, retain the full proposal ratio. In a two-state example, let base masses be $(0.8, 0.2)$ and $\alpha = 2$, so the target is $(16/17, 1/17)$. Suppose the independent proposal has masses $(0.6, 0.4)$. Acceptance from state A to state B is

$$\frac{0.2^2\, 0.6}{0.8^2\, 0.4} = \frac{3}{32},$$

while the reverse is accepted with probability one. The stationary flows agree:

$$\frac{16}{17}\frac{2}{5}\frac{3}{32} = \frac{1}{17}\frac{3}{5} = \frac{3}{85}.$$

Using the simplified base-proposal formula would accept $A \to B$ with probability $1/4$ and break this equality. A truncated proposal can additionally destroy reachability; an acceptance correction does not restore states that can never be proposed.

For numerical stability use $\ell_p(x) = \log p(x \mid c)$ and compute

$$\log R = \alpha(\ell_p(x') - \ell_p(x)) + \log r(v \mid c, u) - \log r(v' \mid c, u).$$

Draw $U \sim \mathrm{Uniform}(0, 1)$ and accept if $\log U < \min(0, \log R)$. Retain enough information to evaluate both forward and reverse suffix likelihoods under the same rule.

Stationarity is not finite-time exactness

At each fixed horizon the corrected kernel has stationary distribution q_n. Convergence from a chosen initialization additionally needs reachability and aperiodicity; a finite number of steps gives only an approximation, potentially a poor one when mixing is slow. Growing a sequence changes the target, so it does not preserve stationarity automatically. Moreover, q_n is generally not the marginal of q_T: summing powered full-sequence probabilities introduces the continuation normalizer derived earlier. Intermediate horizons are warm starts for the final chain.

Choose block size, number of refinement steps, and power jointly with a computation budget. Small blocks mean more stages; early cuts can require regenerating many tokens. Likelihood evaluations, rejected proposals, and regenerated tokens all count toward cost. More probable language-model output is not necessarily more correct, so evaluate task performance separately from acceptance rate and target likelihood.

Supplement R

Reward optimization, RLHF, and preference learning

Source trail: lecture 25, lecture 26, and the instructor's correction in Ed thread #328. The written objective maximizes reward; the clipping operation caps large importance weights. The oral explanation contained a sign/interpretation slip corrected on Ed.

R.1 Standard PPO is a useful comparison, with a different formula

Return to lecture: Lecture 24, §24.6

A common PPO clipped surrogate maximizes

$$J_{\text{PPO}} = \mathbb{E}\left[\min\left(\rho\widehat{A}, \text{clip}(\rho, 1-\epsilon, 1+\epsilon)\widehat{A}\right)\right].$$

For positive advantage, the objective stops rewarding increases beyond $1+\epsilon$. For negative advantage, it stops rewarding decreases below $1-\epsilon$. For example, with $\widehat{A} = 2$ and $\epsilon = 0.2$, ratios 1.1 and 1.4 give contributions 2.2 and 2.4. With $\widehat{A} = -2$, ratios 0.6 and 0.9 give -1.6 and -1.8.

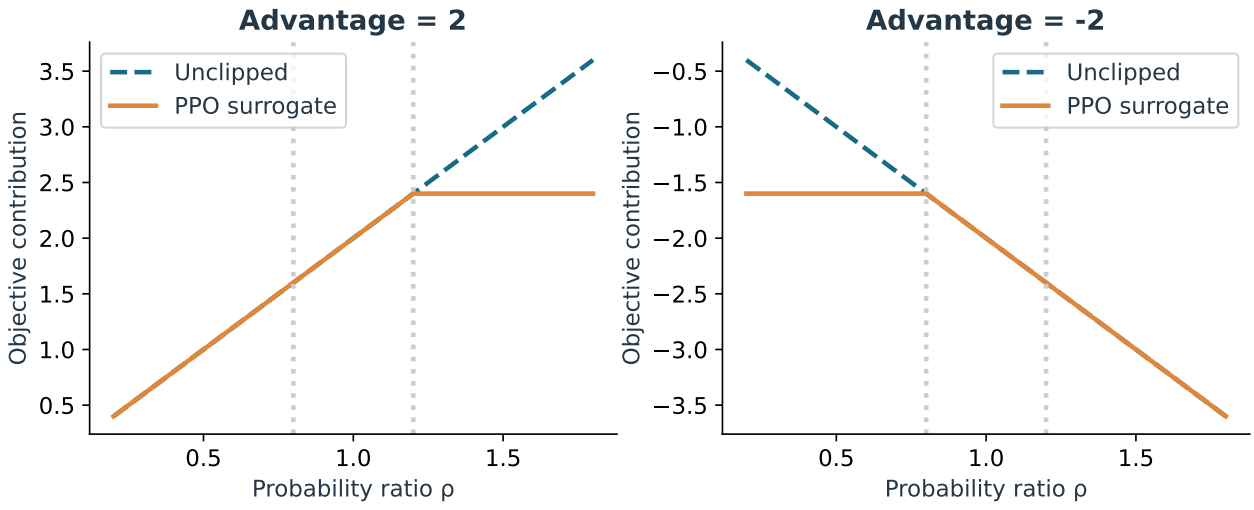


Figure R.1: Standard PPO clips the favorable direction differently for positive and negative advantages. This diagram describes PPO's two-sided surrogate, not the course's stopped-weight ScaleRL expression.

The minimum is not a generic instruction to minimize reward. It selects a conservative contribution inside an objective that is maximized. PPO clipping also does not impose a hard bound on every policy ratio after a shared-parameter optimizer step.

R.2 Deriving the trajectory policy gradient

Return to lecture: Lecture 23, §23.3

For a response trajectory $y_{1:T}$, the joint log probability is a sum of conditional log probabilities. Differentiating expected terminal reward gives

$$\nabla_\theta J = \sum_y r(y) \nabla_\theta \pi_\theta(y) \tag{R.1}$$

$$= \sum_y \pi_\theta(y) r(y) \nabla_\theta \log \pi_\theta(y) \tag{R.2}$$

$$= \mathbb{E}_{\pi_\theta} \left[r(y) \sum_t \nabla_\theta \log \pi_\theta(y_t \mid y_{<t}) \right]. \tag{R.3}$$

The first equality assumes reward has no direct parameter dependence and that exchanging differentiation and summation is valid. If a reward term itself depends on the policy parameters, its direct derivative must also be included or deliberately stopped according to the intended objective.

For rewards arriving at multiple times, actions cannot influence rewards that occurred before them. The expectation of a past reward times the current action's score is zero after conditioning on the current state. This allows replacing total trajectory reward with a return-to-go, reducing unnecessary variance while preserving the expected gradient.

A state-dependent value baseline subtracts expected future reward before the action is sampled. An advantage measures how the selected action compares with that baseline. Estimating advantages accurately is a learning problem of its own; the policy-gradient identity does not make a learned value estimate exact.

R.3 Why a same-group mean is not an independent baseline

Return to lecture: Lecture 26: RLVR recap

Let G independent responses have rewards r_i and score vectors s_i, with $\mathbb{E}s_i = 0$. For $\bar{r} = G^{-1} \sum_j r_j$,

$$\mathbb{E}[(r_i - \bar{r}) s_i] = \left(1 - \frac{1}{G}\right) \mathbb{E}[r_i s_i].$$

The terms with $j \neq i$ vanish by independence, but the self term remains. A leave-one-out baseline $\bar{r}_{-i}$ is independent of response i conditional on the prompt and avoids this multiplicative factor.

Dividing by an empirical group standard deviation introduces further dependence. Filtering prompts by their observed group success rate changes the sampled training distribution. These can be useful algorithmic choices, but they should not be justified by the independent-baseline identity without additional analysis.

For $G = 1$, subtracting the same group's mean reward makes every centered reward zero. That limiting case is an immediate check on a group-relative update. For larger groups, all-identical rewards also supply no relative preference signal after centering, even if the prompt is uniformly easy or uniformly impossible.

R.4 PPO clipping and stopped weighting have different derivatives

Return to lecture: Lecture 24, §24.6

Consider a positive advantage A and ratio $\rho > 1+\epsilon$. The standard PPO clipped contribution is $(1+\epsilon)A$, constant in ρ along that scalar branch. Its derivative through the ratio is zero. Other samples and shared parameters can still change the corresponding probability after an optimizer step.

For the course's schematic stopped-weight contribution,

$$\mathrm{sg}(\min(\rho, c)A) \log \pi_\theta(y \mid x),$$

the derivative remains $\min(\rho, c)A\nabla \log \pi_\theta$. Capping bounds the coefficient rather than making the whole contribution locally flat in the log probability. The two expressions implement different computation graphs.

A reliable way to read a displayed objective is to mark every quantity that depends on parameters, then mark every dependency stopped during differentiation. Only after that should the expression be interpreted as a gradient update. Forward numerical equality does not imply derivative equality when stop-gradient operators are present.

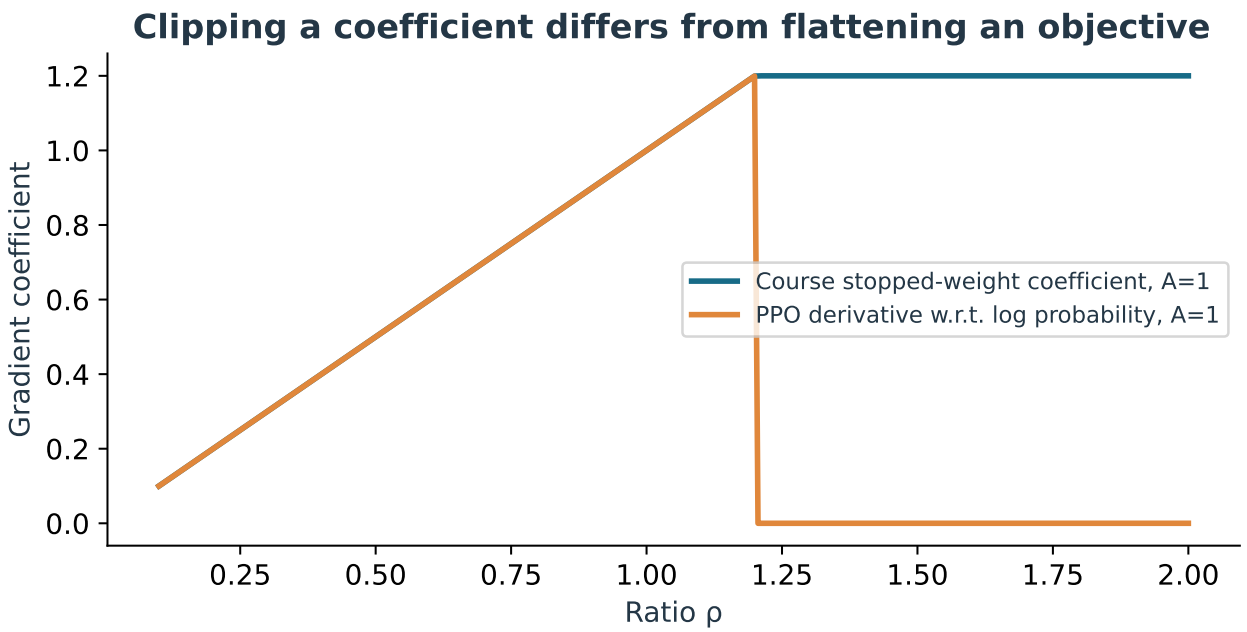


Figure R.2: For positive advantage, PPO becomes flat beyond its upper threshold, while the course stopped-weight expression retains a bounded nonzero gradient coefficient. The threshold kink uses a chosen plotting convention.

R.5 Reward learning from paired comparisons

Return to lecture: Lecture 26, §26.2

Compare the learning signals before comparing algorithms. Supervised fine-tuning increases the likelihood of supplied target tokens. Reward-based policy training samples answers and uses their evaluated reward to weight log-probability gradients, usually with further variance-control and regularization choices. DPO uses preferred/rejected pairs and a frozen reference to construct a logistic objective directly; it need not train a separate reward model as an intermediate step. These distinctions do not guarantee which approach performs better. They identify what data is required and which quantities the displayed objective differentiates.

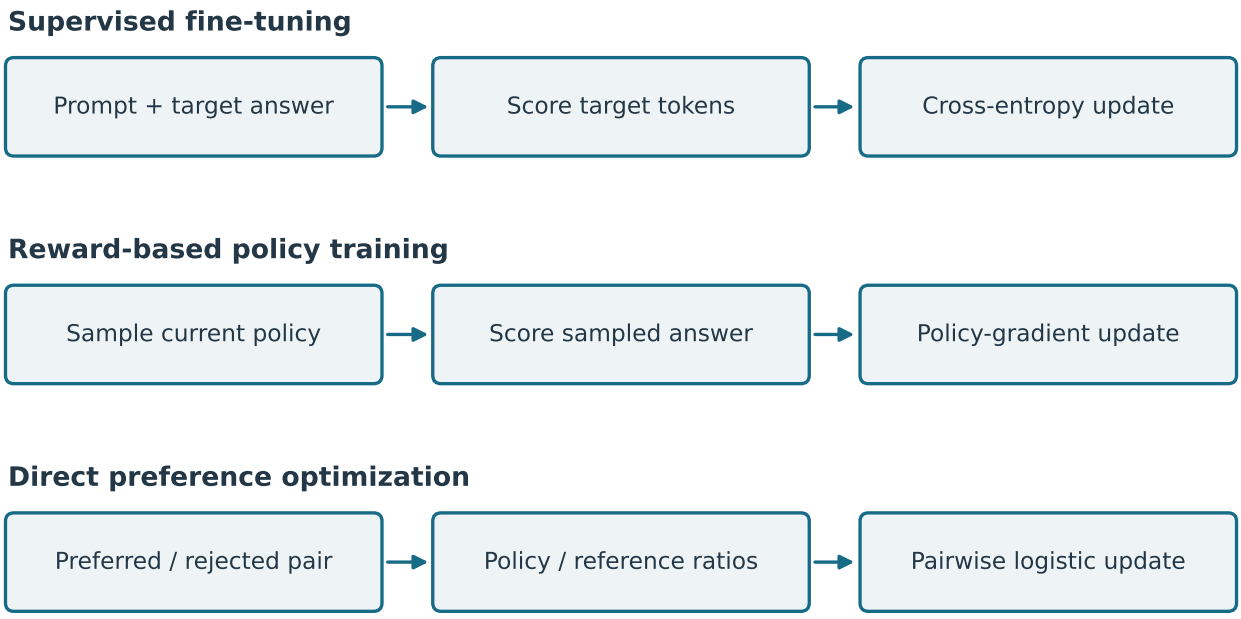


Figure R.3: Three post-training routes use different targets and gradient weights. The diagram omits implementation-specific rollout, clipping, and reference-update choices; those are specified in the surrounding equations.

A reward model can be trained on preferred and rejected responses using

$$L_r = -\log \sigma(r_\psi(x, y_w) - r_\psi(x, y_l)).$$

Only reward differences are identified by this pairwise model. Adding a prompt-dependent constant to all responses leaves the comparison probabilities unchanged. Absolute reward zero has no intrinsic meaning in this setup.

The data can contain inconsistent preferences, ties, annotator differences, or comparisons that depend on omitted context. A scalar reward model compresses these into one score. Optimizing the policy strongly against that score can expose weaknesses that were not obvious on the original comparison dataset.

KL regularization to a reference policy limits one form of distributional departure. It does not guarantee that all relevant behaviors are preserved. The direction of KL matters: $D_{\mathrm{KL}}(\pi \| \pi_{\mathrm{ref}})$ weights outcomes under the updated policy, while the reverse weights them under the reference. Sampling-based implementations should match the intended expectation and support assumptions.

R.6 A numerical DPO example

Return to lecture: Lecture 26: DPO

Suppose the policy-to-reference ratios for a preferred and rejected response are 2 and 0.5. Their log-ratio difference is $\Delta = \log 4 \approx 1.386$. With $\beta = 1$, the modeled preference probability is $\sigma(\Delta) = 0.8$ and the loss is $-\log 0.8 \approx 0.223$.

The derivative with respect to Δ is -0.2. If the ratios are reversed, $\Delta = -\log 4$, the preference probability is 0.2, loss is about 1.609, and derivative is -0.8. The misordered pair receives a stronger push. The reference ratios matter even though the reference weights are frozen.

Sequence log probabilities sum token log probabilities, so longer responses can accumulate larger-magnitude values. Replacing a sum with a length average changes the objective. Whether such a modification is desirable depends on the training method; it should not be made silently while claiming to use the displayed DPO loss unchanged.

R.7 What post-training can be evaluated to improve

Return to lecture: Lecture 26, §26.2

A reward increase on sampled training prompts is evidence about that reward and distribution. To assess broader improvement, evaluate held-out prompts, actual correctness when available, response length, failure modes, and retention of useful prior behavior. If the reward is verifiable, specify what the verifier checks and what it leaves untested.

The course's emphasis on compute-aware comparisons applies here too. More rollout computation, longer responses, additional verifier calls, and more policy updates are separate resources. A better score under a larger budget can be valuable, but it does not isolate a better optimization rule.

Supplement S

Diffusion: learning to reverse a known corruption process

Source trail: lecture 26, lecture 27, and discussion 13. The lecture develops additive Gaussian noising and the distinction between stochastic reversal and deterministic generation. We derive that convention first, then connect it to variance-preserving DDPM notation.

S.1 Binary diffusion makes the same ideas visible

Return to lecture: Lecture 27: diffusion training

Discussion 13 uses pixels in $\{-1, +1\}$, flipped independently with probability δ. For the fair-coin replacement interpretation assume $0 < \delta \leq 1/2$: replace a pixel by a fresh fair coin with probability 2δ, otherwise retain it. After t steps the probability of never being replaced is $(1-2\delta)^t$.

Thus one can sample Y_t directly from Y_0 by retaining each original pixel with that probability, otherwise drawing a fair coin. The probability that at least one of m pixels has not been replaced is at most $m(1-2\delta)^t$ by the union bound. This supplies a joint-distribution argument, stronger than checking only that every marginal becomes fair.

For an original $+1$ pixel, let $\rho = [1 + (1-2\delta)^{t-1}]/2$. Bayes' rule gives

$$P(Y_{t-1} = +1 \mid Y_t = +1, Y_0 = +1) = \frac{(1-\delta)\rho}{(1-\delta)\rho + \delta(1-\rho)}.$$

A sigmoid output predicts a Bernoulli probability, and cross-entropy/KL trains it against the known conditional target. Conditional on Y_0, independent forward pixels permit factorized targets; after marginalizing the unknown original image, the true reverse pixels can be dependent. A product of predicted Bernoulli distributions is therefore a modeling approximation to the full reverse joint, even if each marginal prediction is accurate.

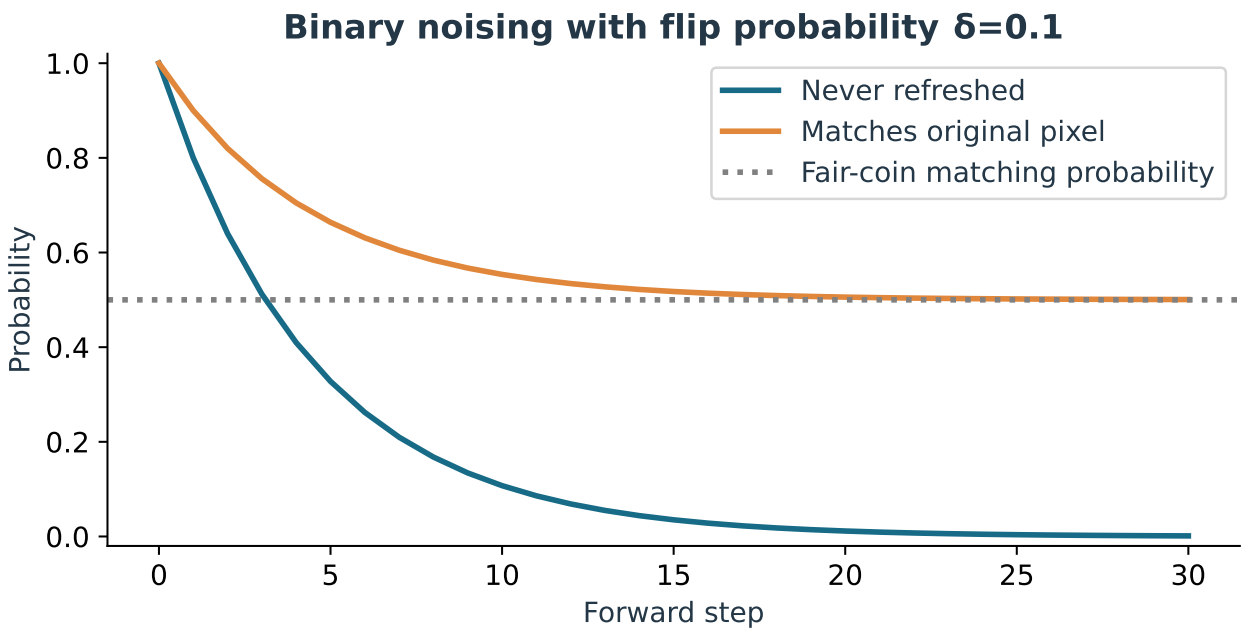


Figure S.1: Binary noising progressively destroys memory of the original pixel. The probability of retaining the original value differs from the probability of never having been refreshed.

The common pattern across binary and Gaussian diffusion is now clear: choose a tractable corruption process, sample noisy training examples directly, learn a conditional reverse statistic, and use it in an explicit generation procedure. Each stage has its own assumptions, and the reverse sampler must match the learned parameterization.

S.2 The Brownian bridge gives an exact conditional mean

Return to lecture: Lecture 27: Gaussian reverse-process argument

Why covariance transforms on both sides. If $y = Ax + \epsilon$ with independent centered noise of covariance R, then $\mathrm{Cov}(y) = A\,\mathrm{Cov}(x)A^\top + R$. Each covariance index refers to one coordinate of the random vector, so both indices must be transformed. The off-diagonal entries describe correlations, not extra examples. This rule underlies Gaussian forward corruption, conditioning, and the Kalman calculations; independence is what removes cross terms.

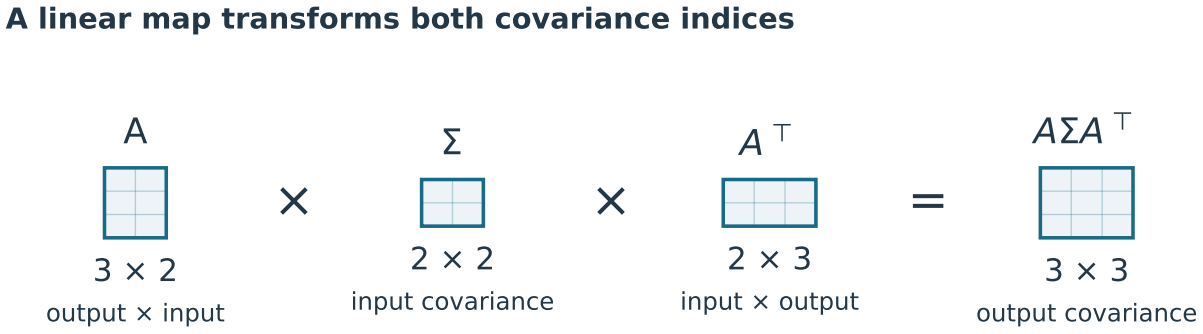


Figure S.2: Covariance propagation changes both coordinate axes. A rectangular linear map produces an output covariance whose size matches the output vector.

In the additive process, write $x_t = x_{t-h} + \eta$ with $\eta \sim \mathcal{N}(0, \sigma^2 h I)$ independent of the earlier forward noise. Conditional on x_0, the pair (x_{t-h}, x_t) is jointly Gaussian. Its covariance blocks are $\sigma^2(t-h)I$, $\sigma^2 t I$, and cross covariance $\sigma^2(t-h)I$.

Gaussian conditioning gives

$$\mathbb{E}[x_{t-h} \mid x_t, x_0] = x_0 + \frac{t-h}{t}(x_t - x_0) = x_t + \frac{h}{t}(x_0 - x_t),$$

with conditional covariance

$$\mathrm{Cov}(x_{t-h} \mid x_t, x_0) = \sigma^2 h \frac{t-h}{t} I.$$

Taking conditional expectation over $x_0 \mid x_t$ yields the denoiser-based reverse mean. The law of total covariance additionally gives

$$\mathrm{Cov}(x_{t-h} \mid x_t) = \sigma^2 h \frac{t-h}{t} I + \frac{h^2}{t^2}\,\mathrm{Cov}(x_0 \mid x_t).$$

The second term reflects uncertainty about the original data. It is order h^2 when the posterior covariance is controlled, while the leading Brownian term is order h. This explains why a small reverse step is approximately Gaussian with variance $\sigma^2 h I$, while an arbitrary finite reverse step need not have that simple form.

For a multimodal data distribution, $p(x_0 \mid x_t)$ can be multimodal. Integrating over it can make the finite-step reverse conditional a mixture. A network predicting only a conditional mean does not describe that entire mixture; the small-step sampling construction is what makes the mean/score sufficient in the continuous-time idealization.

S.3 Deriving the half drift from density evolution

Return to lecture: Lecture 27, §27.4

The forward Brownian density satisfies the heat equation

$$\partial_t p_t = \frac{\sigma^2}{2}\Delta p_t.$$

A deterministic flow with velocity $v_t(x)$ transports density according to the continuity equation

$$\partial_t p_t = -\nabla \cdot (p_t v_t).$$

Choose $v_t = -(\sigma^2/2)\nabla \log p_t$. Since $p_t \nabla \log p_t = \nabla p_t$, the right side becomes $(\sigma^2/2)\Delta p_t$. Thus this deterministic forward flow has the same density evolution as Brownian noising under suitable regularity.

Integrating that ordinary differential equation backward in time changes the sign of the time increment, producing the reverse half-score drift. The stochastic reverse process has a different drift coefficient because it also injects diffusion. The two processes can share marginals while coupling samples across time differently.

This derivation also clarifies the point-mass example. A deterministic map must transport the whole spread of the noisy marginal to the correct earlier spread. Returning a conditional mean averages over hidden earlier states and reduces variance. The missing randomness is not repaired by simply omitting a noise term from the stochastic sampler.

S.4 A Gaussian-data denoiser can be solved exactly

Return to lecture: Lecture 26, §26.3, Lecture 27, §27.2

Let clean data be $x_0 \sim \mathcal{N}(0, \tau^2 I)$ and additive noise have variance $\sigma^2 t I$. Then

$$x_t \sim \mathcal{N}(0, (\tau^2 + \sigma^2 t)I), \qquad D^*(x_t, t) = \frac{\tau^2}{\tau^2 + \sigma^2 t} x_t.$$

The denoiser shrinks the noisy observation toward the prior mean, with stronger shrinkage at larger noise levels. Its score is

$$s_t(x) = -\frac{x}{\tau^2 + \sigma^2 t}.$$

Substituting the denoiser into $(D^* - x)/(\sigma^2 t)$ yields the same expression, providing a direct check of the denoiser-score identity.

For $\tau^2 = 1$, $\sigma^2 t = 3$, and observation $x_t = 4$, the conditional mean is 1. The posterior variance is $\tau^2\sigma^2 t/(\tau^2 + \sigma^2 t) = 0.75$. Predicting 1 does not assert that the original was exactly 1; it is the squared-error-optimal average under that uncertainty.

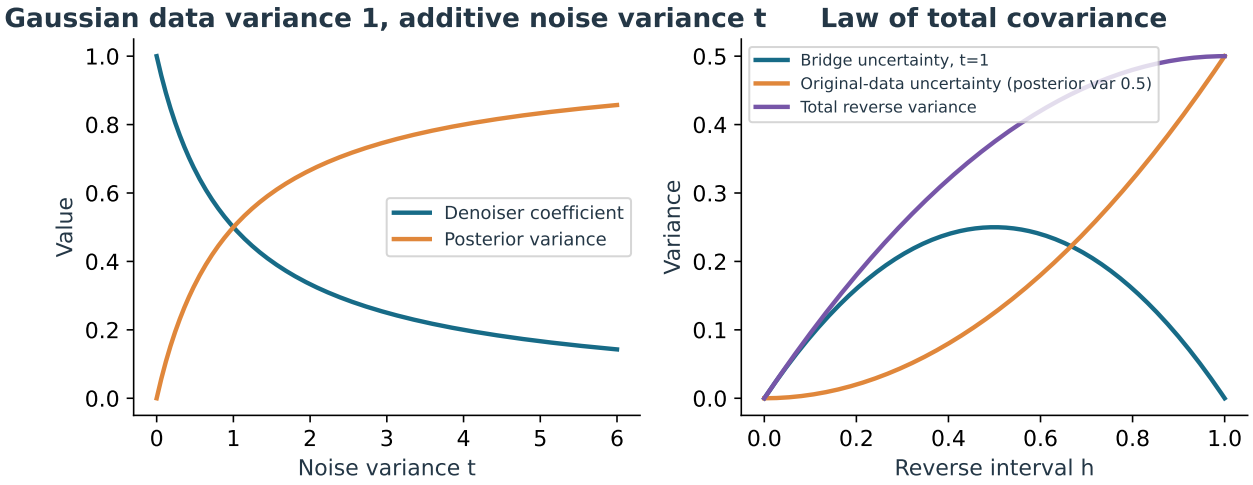


Figure S.3: The Gaussian-data example has a closed-form denoiser. Reverse uncertainty separates into Brownian-bridge uncertainty and uncertainty about the original data.

S.5 Complete the square for the variance-preserving posterior

Return to lecture: Lecture 27: Gaussian reverse-process argument

Use $\alpha_t = 1 - \beta_t$, $\bar{\alpha}_t = \prod_{s=1}^{t} \alpha_s$, and $0 < \beta_s < 1$. For $t > 1$, the two factors involving x_{t-1} are

$$q(x_t \mid x_{t-1}) = \mathcal{N}(\sqrt{\alpha_t} x_{t-1}, \beta_t I)$$

and

$$q(x_{t-1} \mid x_0) = \mathcal{N}(\sqrt{\bar{\alpha}_{t-1}} x_0, (1 - \bar{\alpha}_{t-1}) I).$$

Multiplying them adds precisions. The posterior precision is

$$\frac{\alpha_t}{\beta_t} + \frac{1}{1 - \bar{\alpha}_{t-1}} = \frac{1 - \bar{\alpha}_t}{\beta_t (1 - \bar{\alpha}_{t-1})}.$$

Its inverse is $\tilde{\beta}_t = \beta_t (1 - \bar{\alpha}_{t-1}) / (1 - \bar{\alpha}_t)$. The precision-weighted mean is

$$\frac{\sqrt{\alpha_t}}{\beta_t} x_t + \frac{\sqrt{\bar{\alpha}_{t-1}}}{1 - \bar{\alpha}_{t-1}} x_0.$$

Multiplying by $\tilde{\beta}_t$ gives the posterior mean $\tilde{\mu}_t = \frac{\sqrt{\alpha_t}(1 - \bar{\alpha}_{t-1})}{1 - \bar{\alpha}_t} x_t + \frac{\beta_t \sqrt{\bar{\alpha}_{t-1}}}{1 - \bar{\alpha}_t} x_0$. This derivation exposes every factor and makes off-by-one errors in $\bar{\alpha}$ easier to detect.

At $t = 1$, conditioning on x_0 makes the posterior of $x_{t-1} = x_0$ degenerate. The zero posterior variance in that limit should not be inserted blindly into a Gaussian-density formula requiring positive variance. The first reconstruction term is handled separately in a full variational derivation.

S.6 Training targets and reverse samplers are distinct objects

Return to lecture: Lecture 27: improved diffusion objective

Noise prediction learns a conditional mean of injected noise under the training distribution. Clean-data prediction learns a conditional mean of x_0. Algebraic conversion relates their optimal predictions for a fixed noise level, but the squared losses weight errors differently across levels. Uniform sampling over discrete timesteps is also a choice of training emphasis.

The reverse sampler combines a learned prediction with a numerical update and a schedule. Changing the number of steps changes the integration error and the distribution of states visited by the network. A denoiser trained at certain noise levels may interpolate imperfectly at others. The generation procedure is therefore part of the model evaluation, even when the network weights are fixed.

Conditioning a denoiser on labels, text, or another observation changes the target score to a conditional score. The same Bayes and denoising identities can be applied with that conditioning held fixed. A U-Net or transformer supplies the parameterized function; diffusion specifies the probabilistic training and sampling construction around it.

S.7 Binary reverse targets average to the desired marginals

Return to lecture: Lecture 27: diffusion training

Let the training target for one pixel be the tractable posterior probability $a(Y_t, Y_0) = P(Y_{t-1} = +1 \mid Y_t, Y_0)$. A network predicts $q(Y_t)$. Conditional on Y_t, expected Bernoulli cross-entropy is

$$-\mathbb{E}[a \mid Y_t] \log q - (1 - \mathbb{E}[a \mid Y_t]) \log(1 - q).$$

It is minimized at $q = \mathbb{E}[a \mid Y_t]$. By the law of total probability, that expectation equals $P(Y_{t-1} = +1 \mid Y_t)$. Thus training against the original-conditioned proxy target can recover the desired reverse *marginal* probability in the population limit.

This does not make different pixels independent after conditioning only on Y_t. A product decoder may fit all marginal probabilities while missing correlations. The distinction between correct marginals and a correct joint distribution is central to understanding what the binary exercise proves and what remains a modeling approximation.

S.8 Late binary reverse targets are not generally one-half

Return to lecture: Lecture 27: diffusion training

There are two distinct conditionals: $P(Y_t[j] = +1 \mid Y_0)$ describes the forward noisy pixel, while $P(Y_{t-1}[j] = +1 \mid Y_t, Y_0)$ describes a reverse step. For fixed $0 < \delta < 1/2$, the first tends to $1/2$ as t grows. The second still observes the current pixel and retains strong evidence about the immediately previous pixel.

For $Y_0[j] = +1$, let $\rho_t = [1 + (1 - 2\delta)^{t-1}]/2$. The already-derived reverse targets are

$$a_t^+ = \frac{(1-\delta)\rho_t}{(1-\delta)\rho_t + \delta(1-\rho_t)}, \qquad a_t^- = \frac{\delta\rho_t}{\delta\rho_t + (1-\delta)(1-\rho_t)}.$$

As $t \to \infty$, $\rho_t \to 1/2$, so $a_t^+ \to 1 - \delta$ and $a_t^- \to \delta$. With $\delta = 0.1$, the limits are 0.9 and 0.1, not 0.5. At $t = 1$ both targets equal 1 for the stipulated original positive pixel. Thus a time-dependent soft target is a useful analogy, but the label-smoothing explanation in Discussion 13, page 3 incorrectly identifies its late limit. At stationarity a reverse step remains correlated with the observed current pixel, even though the marginal pixel distribution is fair.

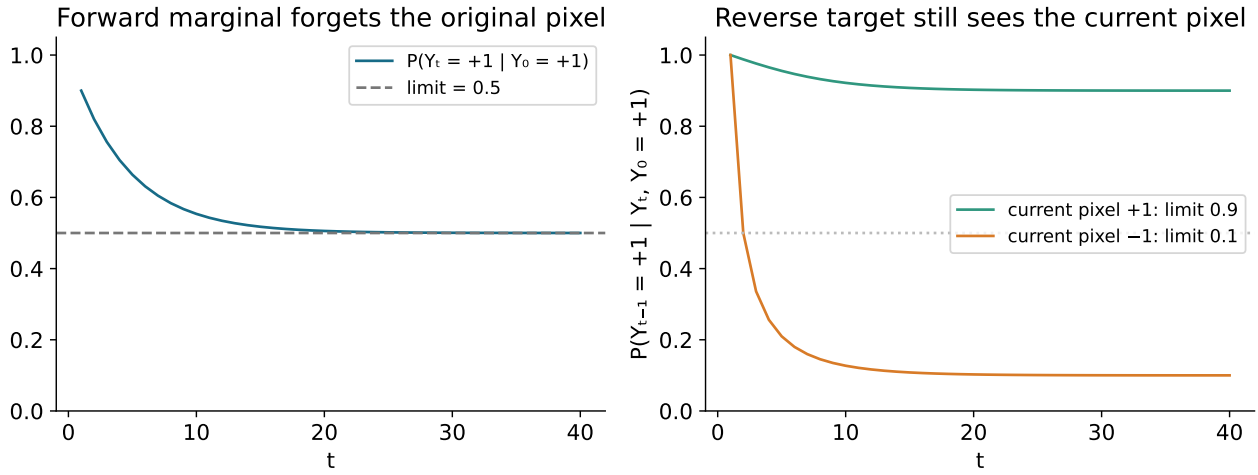


Figure S.4: A fair forward marginal does not make consecutive states independent. The two reverse targets approach different limits because they condition on different observed current pixels.

Supplement T

Discussion companion: thirteen sets of practice

This companion gives a route through every collected discussion, with additional derivations and small examples. The linked PDFs contain the complete original questions and official solutions. Attempt the questions before opening the solutions; then use the explanations below to diagnose the reasoning step that failed. The examples here are newly written and are not a replacement for the original problem statements.

T.1 Discussion 1 — gradient descent and ReLU mechanics

Return to lecture: Lecture 1, §1.4

Original: questions embedded in the solution PDF. A separate question-only PDF was not found in the inspected course posts.

The progression from a line to a nonlinear network is about differentiating through more structure. For a scalar linear predictor $\hat{y} = wx + b$ and loss $\frac{1}{2}(\hat{y} - y)^2$, the gradients are $(\hat{y} - y)x$ and $\hat{y} - y$. The residual is the error signal; the input determines how much changing the weight changes the prediction.

For $\hat{y} = a\,\mathrm{ReLU}(wx + b) + c$, let $r = \hat{y} - y$ and $z = wx + b$. Away from $z = 0$,

$$\partial_a L = r\,\mathrm{ReLU}(z), \quad \partial_w L = ra\mathbf{1}_{z>0}x, \quad \partial_b L = ra\mathbf{1}_{z>0}, \quad \partial_c L = r.$$

A negative preactivation blocks the gradient into w, b for this example, while the output bias can still change. A unit that is inactive for every training example may have no route to recover through these gradients alone. Whether it is permanently inactive depends on updates to other parameters, data, and the architecture.

Try it. With $a = 2, w = 1, b = -1, c = 0, x = 3, y = 1$, the prediction is 4 and residual is 3. The gradients in (a, w, b, c) order are $(6, 18, 6, 3)$. A step of 0.01 gives $(1.94, 0.82, -1.06, -0.03)$. Recompute the prediction after updating all parameters simultaneously; using one updated value while computing another parameter's old gradient changes the algorithm.

T.2 Discussion 2 — derivatives as changes in a function

Return to lecture: Lecture 3, §3.3

Originals: questions · solutions.

Draw $f(x)$ and the function $\partial f(x)/\partial\theta_j$ on the same input interval. For $f(x) = a\,\mathrm{ReLU}(wx+b)$, the derivative with respect to a is a hinge, while the derivative with respect to b is $a\mathbf{1}_{wx+b>0}$. A small bias change shifts the breakpoint; its first-order effect is a step-shaped function away from the kink.

Computation graphs make gradient addition unavoidable. If $u = wx$ and $f = u^2 + 3u$, then $df/du = 2u+3$ because both downstream branches depend on u. Hence $df/dw = (2wx+3)x$. Omitting either branch is a common reverse-mode error.

When comparing SGD, momentum, and adaptive updates on the same ReLU network, keep the function change visible. Equal numerical learning rates do not mean equal steps in function space. An optimizer that rescales coordinates changes the combination of derivative functions used to update the prediction.

Check: perturb one parameter positively and negatively, plot the centered difference, and compare with the analytic derivative except near breakpoints. This turns an abstract gradient into a directly observable change in the represented curve.

T.3 Discussion 3 — optimizers as geometric choices

Return to lecture: Lecture 5, §5.4

Originals: questions · solutions.

The local model $L(\theta+\Delta) \approx L(\theta) + g^\top\Delta$ has no finite minimizer without a size constraint or penalty. Different notions of size produce different update directions. A Euclidean penalty yields ordinary gradient descent; a maximum-coordinate constraint yields a sign direction; a matrix spectral constraint yields a polar factor.

For the penalized problem

$$\min_{\Delta}\ g^\top\Delta + \frac{1}{2\eta}\|\Delta\|^2,$$

the minimum value is $-\eta\|g\|_*^2/2$, where $\|\cdot\|_*$ is the dual norm. To see the magnitude, write $\Delta = ru$ with $\|u\| = 1$. The best direction attains $g^\top u = -\|g\|_*$. Minimize $-r\|g\|_* + r^2/(2\eta)$ to obtain $r = \eta\|g\|_*$. A constrained step fixes its radius; a penalized step chooses its radius using the gradient's dual norm. Do not silently interchange them.

For RMS units, $\|x\|_{\mathrm{RMS}} = \|x\|_2/\sqrt{d}$, so the dual norm under the ordinary dot product is $\sqrt{d}\|g\|_2$, not $\|g\|_2/\sqrt{d}$. Substituting into the penalty directly gives $\Delta = -\eta d g$.

For convergence, inspect the multiplier of each eigenmode rather than asserting that a decreasing loss must reach zero. A decreasing nonnegative sequence can converge to a positive constant. A uniform contraction $\mathbb{E}[V_{t+1} \mid V_t] \le CV_t$ with $C < 1$ gives a much stronger conclusion.

T.4 Discussion 4 — maximal updates and Newton–Schulz

Return to lecture: Lecture 7, §7.2

Originals: questions · solutions.

Start from the exact single-example gradient $G = gx^\top$. Then ask how the proposed update acts on the actual input and how large that action is in RMS units. For SignGD with all nonzero coordinates, the outer-product sign matrix has spectral norm $\sqrt{d_{\text{out}} d_{\text{in}}}$. Its RMS-to-RMS norm is d_{in}, giving the inverse-fan-in coordinate step derived earlier.

For Newton–Schulz, test one singular value before trusting a matrix implementation. Starting at $s_0 = 0.2$, the cubic map $s_{k+1} = 1.5s_k - 0.5s_k^3$ gives 0.296, then approximately 0.431, then 0.607. A small singular value is lifted gradually. Starting at zero leaves it zero; no finite polynomial step creates a missing rank direction.

For rectangular matrices, verify that $XX^\top X$ has the same shape as X. An expression such as X^3 is often undefined. In a numerical experiment, compare the singular values after the matrix iteration with scalar iterations of the original singular values. This checks the actual algebra without relying on a visual impression of "orthogonality."

T.5 Discussion 5 — convolution as shared linear algebra

Return to lecture: Lecture 9, §9.2

Originals: questions · solutions.

Implement convolution first as an explicit loop or matrix multiplication on a tiny input. For a one-dimensional valid cross-correlation with kernel (a, b) and input (x_1, x_2, x_3),

$$y = \begin{bmatrix} a & b & 0 \\ 0 & a & b \end{bmatrix} x.$$

The same parameter occurs in multiple matrix entries. If the output sensitivities are (g_1, g_2), then $\partial L/\partial a = g_1 x_1 + g_2 x_2$ and $\partial L/\partial b = g_1 x_2 + g_2 x_3$. Sharing changes how gradients accumulate.

For a 3×3 kernel with 16 input channels and 32 output channels, a standard convolution has $3 \cdot 3 \cdot 16 \cdot 32 = 4608$ weights, plus 32 biases if present. The count is independent of spatial image size. A locally connected layer with separate kernels at every location does not have this property.

Debugging sequence: verify channel order, compute one output by hand, check padding and stride, check output dimensions, then compare backward gradients. A library call can run successfully while using the wrong semantic axis order.

T.6 Discussion 6 — residual paths, upsampling, and dropout

Return to lecture: Lecture 11, §11.4

Originals: questions · solutions.

For $y = x + F(x)$, the Jacobian is $I + J_F$. The identity contribution gives a direct gradient path, but cancellation can still occur if J_F has an eigenvalue near -1. "Residual connections help gradients" is an architectural intuition, not a guarantee that every singular value stays close to one.

If spatial size or channel count changes, both branches must end with compatible shapes. A projection skip is a learned or fixed map P giving $y = Px + F(x)$; its direct Jacobian is P, not I. The lecture's updated downsampling discussion emphasizes which branch actually performs the resolution change.

With inverted dropout, $\tilde{h} = mh/q$, where $m \sim \text{Bernoulli}(q)$, the mean is h and variance is $h^2(1-q)/q$. For $h = 3, q = 0.75$, the output is 4 with probability 0.75 and 0 otherwise, with mean 3 and variance 3. Preserving a layer's expected activation does not imply that a nonlinear downstream network's expected output is unchanged.

Compare nearest-neighbor upsampling followed by convolution with transposed convolution. Both can enlarge a spatial representation, but they impose different weight patterns and overlap behavior. Neither reconstructs information discarded earlier without additional assumptions or learned priors.

T.7 Discussion 7 — graphs and recurrent backpropagation

Return to lecture: Lecture 14, §14.3

Originals: questions · solutions.

A GNN layer should respect relabeling of nodes. If P is a permutation matrix, transform features as PH and adjacency as $PAP^\top$. Verify that the output becomes PH' under a permutation-equivariant layer. An aggregator depending on an arbitrary neighbor listing can break the required symmetry.

For a three-node chain with features $(1, 2, 4)$ and a sum of neighbors including self, one layer gives $(3, 7, 6)$. A mean aggregator instead gives $(1.5, 7/3, 3)$. These are different computations, not merely different normalizations of a final global answer. Repeated message passing increases the receptive field but can also make node representations less distinguishable.

A scalar linear RNN $h_t = wh_{t-1} + ux_t$ illustrates shared-parameter derivatives. With $h_0 = 0, x_1 = 1, x_2 = 2, w = 0.5, u = 1$, we have $h_1 = 1, h_2 = 2.5$. For $L = \frac{1}{2}(h_2 - 1)^2$, $\partial L/\partial w = 1.5$ and $\partial L/\partial u = 3.75$. The latter includes both the direct use of u at time 2 and its earlier use through h_1.

T.8 Discussion 8 — transposes, sequence prediction, and autoencoders

Return to lecture: Lecture 14, §14.6

Originals: questions · solutions.

For the convolution matrix C in discussion 5, the transposed operation is $C^\top v$. Its defining property is the adjoint identity $\langle Cx, v\rangle = \langle x, C^\top v\rangle$. This is the backward map from output sensitivity to input sensitivity. It is not an inverse unless additional exceptional conditions hold.

An autoregressive RNN predicts $p(x_t \mid x_{<t})$ from a state that summarizes the prefix. During training, the true preceding tokens can be supplied; during free generation, the model receives its own outputs. Keep this change in input distribution visible when interpreting generated sequences.

For a linear autoencoder with a k-dimensional bottleneck, centered data, squared reconstruction loss, and suitable optimization, the optimal reconstruction subspace is a leading principal subspace. The encoder/decoder factors themselves are not unique: inserting an invertible latent transformation and its inverse preserves the reconstruction. A good reconstruction loss therefore does not establish unique semantic meaning for each latent coordinate.

T.9 Discussion 9 — state-space kernels and attention

Return to lecture: Lecture 16, §16.2

Originals: questions · solutions.

State the timing convention before computing a kernel. With $h_t = Ah_{t-1} + Bx_t$, $y_t = Ch_t + Dx_t$, and zero initial state, the kernel is $K_0 = CB + D$ and $K_k = CA^kB$ for $k \geq 1$. Under the alternative convention where output uses the previous state, the instantaneous term and lag indexing differ.

For $A = 0.5, B = C = 1, D = 0$, input $(1, 2, 0)$ gives outputs $(1, 2.5, 1.25)$. Convolution with kernel $(1, 0.5, 0.25)$ gives the same result. If a direct recurrence and your convolution disagree by one position, suspect timing before suspecting floating-point error.

Attention uses the current query to select a mixture of stored values. In the discussion's sequence task, early instructions affect later character outputs. Separate the feature that determines what to retrieve from the feature that should be returned. A key can mark relevance while a value carries the character or transformation information.

Shape check: with five queries, seven keys, and value width three, attention weights have shape 5×7 and outputs 5×3. Softmax runs across the seven keys for each query.

T.10 Discussion 10 — position and transformer architecture

Return to lecture: Lecture 19, §19.2

Originals: questions · solutions. The collected question PDF is the updated version linked by course staff.

Derive the RoPE relative-position identity using $R_i^\top R_j = R_{j-i}$. Rotating query and key by the same global positional offset cancels in their dot product within a frequency pair. This is a concrete algebraic property, not a claim that all other parts of a transformer are shift invariant.

For the NoPE construction, examine what the causal mask reveals. The first token has one accessible position; later tokens have larger prefixes. An attention layer can exploit prefix-dependent aggregation even without an explicit position vector. The updated arbitrary third embedding coordinate should be treated as a degree of freedom, not as a required constant.

When reading a transformer diagram, label every tensor's token and feature axes. Layer normalization acts within a token; attention mixes token positions; the MLP transforms features within each position. Residual additions require equal dimensions. This axis-level description is more reliable than memorizing the order of boxes.

T.11 Discussion 11 — inference computation and fine-tuning

Return to lecture: Lecture 21, §21.3

Originals: questions · solutions.

Separate the underlying model probability from the decoding score. Greedy search is local; beam search approximates a sequence objective; sampling explores a distribution; a verifier ranks candidates using another signal. The best method depends on the objective and compute budget, not merely on which algorithm searches more branches.

For adaptation, distinguish frozen-feature classification, full fine-tuning, and parameter-efficient methods. In a frozen-feature classifier, gradients train the new head while the feature extractor stays fixed. In LoRA, selected internal linear maps change through low-rank factors. A soft prompt changes input vectors while holding the base weights fixed.

Exercise: a matrix $W \in \mathbb{R}^{512\times1024}$ has 524,288 entries. Rank 8 LoRA adds $8(512+1024) = 12,288$ trainable entries, about 2.34% of that matrix's count. If the model has other unadapted matrices, compute the whole-model percentage separately.

T.12 Discussion 12 — information and forgetting

Return to lecture: Lecture 22, §22.3

Originals: questions · solutions.

For the handout's $p = (0.5, 0.5)$ and $q = (0.1, 0.9)$, use base-two logarithms to obtain $D_{\mathrm{KL}}(p\|q) \approx 0.737$ bits and $D_{\mathrm{KL}}(q\|p) \approx 0.531$ bits. Using natural logs gives different numerical values but the same asymmetry. Always report units.

For forgetting, draw low-loss regions of two tasks. Freezing the whole model protects old parameters but may prevent new learning; unrestricted fine-tuning can leave the old region; joint training adds pressure to remain good on both. Whether the regions overlap is an empirical and representational question. There may be a genuine tradeoff rather than one ideal midpoint.

The paper-reading component is a method: identify the problem and assumptions, reconstruct the claimed mechanism, inspect the evaluation protocol, and ask what comparison would distinguish that mechanism from a simpler explanation. For a scaling plot, inspect both axes, which variables were controlled, and whether the comparison uses equal data or equal computation. A curve alone is not a derivation.

T.13 Discussion 13 — binary diffusion and latent models

Return to lecture: Lecture 27: diffusion training

Originals: questions · solutions.

The binary diffusion exercise replaces continuous Gaussian algebra with coin flips. For $\delta = 0.1$ and $t = 5$, the probability of never having been refreshed is $0.8^5 = 0.32768$. The probability a pixel equals its original value is larger: $[1 + 0.8^5]/2 = 0.66384$, because a refreshed fair coin can match the original by chance.

At reverse step $t = 1$, conditioning on the original image makes the target a point mass on that image. A Bernoulli cross-entropy then becomes ordinary supervised binary reconstruction. At later steps, the tractable target is a soft conditional probability, computed using Bayes' rule.

The latent-variable exercise asks how a prior, likelihood, and approximate posterior interact. Derive the ELBO by inserting the approximate posterior into the marginal likelihood and applying Jensen, then identify the exact posterior KL gap. A diagram with arrows alone is not enough: write which distribution supplies each expectation.

Final synthesis exercise. Explain a VAE and a diffusion model using the same four headings: how training pairs are obtained, what conditional object the network predicts, what objective is optimized, and how a new sample is generated. Both use latent randomness, but their inference and generation procedures differ substantially.

T.14 Worked case: build and train a piecewise-linear function

Return to lecture: Lecture 1, §1.1

Problem. Construct a scalar ReLU network whose slope is -1 before $x = 0$, 2 between 0 and2, and0 after 2, with $f(0) = 1$. Then derive the gradient of a squared-error observation at $x = 1$ with target 4.

Step 1: encode slope changes. The slope changes by 3 at 0 and by -2 at 2. Therefore

$$f(x) = 1 - x + 3\,\mathrm{ReLU}(x) - 2\,\mathrm{ReLU}(x - 2).$$

For $x < 0$, this is $1 - x$. For $0 \le x < 2$, it is $1 + 2x$. For $x \ge 2$, it is 5. The values agree at both knots, so the representation is continuous.

Step 2: identify trainable parameters. Write $f(x) = b + mx + a_1\,\mathrm{ReLU}(x - t_1) + a_2\,\mathrm{ReLU}(x - t_2)$. The parameter values are $(b, m, a_1, t_1, a_2, t_2) = (1, -1, 3, 0, -2, 2)$. At $x = 1$, only the first hinge is active and $f(1) = 3$.

Step 3: differentiate the loss. For $L = \frac{1}{2}(f(1) - 4)^2$, the residual is -1. The prediction derivatives are $\partial_b f = 1$, $\partial_m f = 1$, $\partial_{a_1} f = 1$, and $\partial_{t_1} f = -a_1 = -3$. The second hinge contributes zero derivatives at this input. Thus

$$\nabla L = (-1, -1, -1, 3, 0, 0).$$

A positive gradient in t_1 means gradient descent moves the first knot left, increasing its active contribution at $x = 1$. This explains the sign geometrically rather than relying only on symbolic differentiation.

What this case teaches. Parameters can affect height, slope, and knot location differently. One observation does not constrain inactive hinges. A network may interpolate observed points while leaving substantial freedom elsewhere. The shape of parameter derivatives reveals what each training example can and cannot teach.

T.15 Worked case: solve a rank-deficient regression three ways

Return to lecture: Lecture 2, §2.1

Problem. Fit the single equation $w_1 + 2w_2 = 3$. Compare minimum-norm least squares, ridge with strength $\gamma = 1$, and gradient descent from zero with step $\eta = 0.1$.

Minimum norm. The minimum-norm solution lies along the row direction $(1, 2)$. Setting $w = c(1, 2)$ gives $5c = 3$, hence $w^+ = (0.6, 1.2)$. The norm squared is 1.8. Every vector $w^+ + d(-2, 1)$ fits equally well but has norm squared $1.8 + 5d^2$.

Ridge. The dual form gives

$$w_\gamma = X^\top (XX^\top + \gamma)^{-1} y = \frac{3}{5 + \gamma}(1, 2)^\top.$$

At $\gamma = 1$, the result is $(0.5, 1)$ and the prediction is 2.5. The training residual is nonzero because the objective also penalizes parameter magnitude. Ridge did not fail to solve its objective; it solved a different objective.

Gradient descent. Let $p_t = w_{1,t} + 2w_{2,t}$ be the prediction. The gradient step yields

$$p_{t+1} = p_t - 5\eta(p_t - 3).$$

With $\eta = 0.1$ and $p_0 = 0$, the error halves each step. Thus $p_t = 3(1 - 2^{-t})$ and

$$w_t = \frac{3}{5}(1 - 2^{-t})(1, 2)^\top.$$

After three steps, $w_3 = (0.525, 1.05)$ and $p_3 = 2.625$. As $t \to \infty$, the minimum-norm solution is recovered.

Comparison. Early stopping and ridge both shrink the row-space coefficient, but their dependence on singular values differs in a multi-direction problem. In this one-dimensional observed subspace, one can choose a ridge strength matching a particular stopping time. That special match does not establish global equivalence on a general spectrum.

T.16 Worked case: compare optimizer geometries on the same gradient

Return to lecture: Lecture 5, §5.3

Problem. For $g = (3, 1)$, compute a radius-one step under L_2, L_∞, and L_1 constraints. Then solve the corresponding squared-norm penalized problems with $\eta = 0.1$.

Constrained directions. The Euclidean step is $-(3, 1)/\sqrt{10}$, with improvement $-\sqrt{10}$. The infinity-norm step is $(-1, -1)$, with improvement -4. The one-norm step is $(-1, 0)$, with improvement -3. These improvements use different unit balls; comparing their numerical values as if the radius meant the same physical change would be misleading.

Penalized magnitudes. The optimal radius is $\eta\|g\|_*$ under each chosen primal norm. The dual norms are $\sqrt{10}$,4, and3 respectively. Hence the penalized steps are

$$u_2 = (-0.3, -0.1), \quad u_\infty = (-0.4, -0.4), \quad u_1 = (-0.3, 0).$$

The sign direction alone does not determine the infinity-norm penalized step; its magnitude contains the gradient's L_1 norm.

Verify one objective. For $u_\infty = (-0.4, -0.4)$, the linear term is -1.6 and the penalty is $0.4^2/(2 \cdot 0.1) = 0.8$, giving total -0.8. The formula $-\eta\|g\|_*^2/2$ gives $-0.1 \cdot 16/2 = -0.8$ as well.

Matrix extension. For $G = \mathrm{diag}(3, 1)$ and a spectral radius-one constraint, the polar step is $-I$, giving Frobenius inner product -4. Flattened Euclidean descent instead uses a direction proportional to $-G$. The distinction is whether size is measured by entrywise energy or worst-case input amplification.

T.17 Worked case: a convolution and its exact backward pass

Return to lecture: Lecture 9, §9.2

Problem. Use valid cross-correlation with kernel $(a, b) = (2, -1)$ on input $x = (1, 3, 2)$. Let targets be $(0, 1)$ and loss $L = \frac{1}{2}\|y - y^*\|^2$. Compute all gradients.

Forward map. The convolution matrix is

$$C = \begin{bmatrix} 2 & -1 & 0 \\ 0 & 2 & -1 \end{bmatrix}, \qquad y = Cx = (-1, 4)^\top.$$

The output sensitivity is $g_y = y - y^* = (-1, 3)^\top$.

Input gradient. Apply the adjoint:

$$g_x = C^\top g_y = \begin{bmatrix} 2 & 0 \\ -1 & 2 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} -1 \\ 3 \end{bmatrix} = \begin{bmatrix} -2 \\ 7 \\ -3 \end{bmatrix}.$$

The middle input appears in both outputs, so its gradient adds two contributions: $(-1)(-1) + 2(3) = 7$.

Kernel gradient. Shared weights accumulate across positions:

$$\partial_a L = (-1)x_1 + 3x_2 = 8, \qquad \partial_b L = (-1)x_2 + 3x_3 = 3.$$

A separate bias added to both outputs would have gradient $-1 + 3 = 2$. No averaging over output positions should be introduced unless it is part of the loss definition.

Check by perturbation. Increasing a by a small ϵ changes the output by $\epsilon(x_1, x_2)$. The first-order loss change is $\epsilon g_y^\top(x_1, x_2) = 8\epsilon$, matching the derivative. This directional check tests shared-parameter accounting without deriving every entry of a large Jacobian.

T.18 Worked case: distinguish normalization invariances

Return to lecture: Lecture 9, §9.6

Problem. Compare ideal LayerNorm and RMSNorm on $x = (1, 3)$, then on $x + 10\mathbf{1}$ and $2x$. Ignore learned affine parameters and take $\epsilon = 0$ for the algebra.

LayerNorm. The mean is 2, centered vector is $(-1, 1)$, and coordinate variance is 1. The normalized vector is $(-1, 1)$. Adding10 leaves the centered vector unchanged, so the output is unchanged. Multiplying by positive2 scales both the centered vector and its standard deviation by 2, again leaving the output unchanged.

RMSNorm. The RMS is $\sqrt{5}$, so the output is $(1, 3)/\sqrt{5}$. Multiplying by 2 preserves it because numerator and denominator both double. Adding10 produces $(11, 13)/\sqrt{145}$, which is different. RMSNorm preserves the common offset information that LayerNorm removes.

Derivative interpretation. The LayerNorm Jacobian annihilates the constant-shift direction. In two dimensions with zero epsilon and nonidentical coordinates, the normalized output is locally fixed up to the ordering of the coordinates, so its derivative is zero away

from the equality boundary. This special low-dimensional case should not be used to infer that LayerNorm gradients vanish in a high-dimensional network.

Implementation qualification. Positive epsilon softens exact scale invariance and prevents division by zero. Learned γ, β change the output coordinates after normalization. Numerical checks must include the same epsilon, reduction axes, and affine parameters as the implementation under test.

T.19 Worked case: compute an SSM in two representations

Return to lecture: Lecture 16, §16.2

Problem. Let $A = \mathrm{diag}(1/2, 1/4)$, $B = (1,1)^\top$, $C = (1,-1)$, $D = 1$, and $h_0 = 0$. Use $h_t = Ah_{t-1} + Bx_t$, $y_t = Ch_t + Dx_t$ on input $(1,2,0)$.

Recurrence. At the first step, $h_1 = (1,1)$ and $y_1 = 1$. Next, $h_2 = (2.5, 2.25)$ and $y_2 = 0.25 + 2 = 2.25$. Finally, $h_3 = (1.25, 0.5625)$ and $y_3 = 0.6875$.

Kernel. The instantaneous coefficient is $K_0 = CB + D = 1$. The next two coefficients are

$$K_1 = CAB = 0.5 - 0.25 = 0.25, \qquad K_2 = CA^2B = 0.25 - 0.0625 = 0.1875.$$

Convolution gives $y_1 = K_0$, $y_2 = 2K_0 + K_1 = 2.25$, and $y_3 = 2K_1 + K_2 = 0.6875$. The two computations agree exactly.

Interpretation. The hidden state contains two decaying modes, and the output subtracts one from the other. A large hidden response can cancel at the output. Looking only at eigenvalues or only at hidden-state magnitude does not reveal the input-output kernel; B and C matter.

Extension. If A changes with the current input, the recurrence remains valid but these fixed kernel coefficients no longer describe every sequence. If the initial state is nonzero, add the term $CA^t h_0$. These two modifications explain common failures when converting a recurrent implementation into a convolutional one.

T.20 Worked case: attention with a causal mask

Return to lecture: Lecture 18, §18.2

Problem. At the third position, scores for three values are $(\log 2, 0, \log 3)$ and values are $(1, 4, 7)$. Compute the output and the derivative with respect to each score. Then mask the third key.

Unmasked weights. Exponentiating gives $(2, 1, 3)$, so weights are $(1/3, 1/6, 1/2)$. The output is

$$o = \frac{1}{3} + \frac{4}{6} + \frac{7}{2} = \frac{9}{2} = 4.5.$$

For a scalar output derivative of one, score derivatives are $p_j(v_j - o)$:

$$\left(-\frac{7}{6}, -\frac{1}{12}, \frac{5}{4}\right).$$

They sum to zero, as they must because adding the same constant to all scores leaves softmax unchanged.

Masked weights. Removing the third key before softmax gives $(2/3, 1/3, 0)$ and output2. The derivatives are $(-2/3, 2/3, 0)$. The masked value contributes neither forward mass nor a score gradient.

Loss gradient. If the target output is 5 and the loss is $\frac{1}{2}(o-5)^2$, multiply these score derivatives by the residual. In the masked case the residual is -3, so the loss score gradient is $(2, -2, 0)$. Gradient descent lowers the first score and raises the second, moving weight toward the larger available value4. It cannot reach target 5 with these fixed unmasked values because their convex hull ends at 4.

What can change next. Training can also change value projections, output projections, and other layers. The convex-hull restriction applies to one head with its current values, not to the full trainable network. This distinction explains why a local limitation does not establish an overall expressivity limit.

T.21 Worked case: one exact meta-gradient and one approximation

Return to lecture: Lecture 22, §22.2

Problem. Let support loss be $L_S(\theta) = \frac{1}{2}(2\theta - 4)^2$, query loss be $L_Q(\theta') = \frac{1}{2}(\theta' - 3)^2$, and the inner step size be $\alpha = 0.1$. Compute the adapted parameter and outer gradient at $\theta = 0$.

Inner gradient. $\nabla L_S = 2(2\theta - 4) = 4\theta - 8$. Thus $\theta' = \theta - 0.1(4\theta - 8) = 0.6\theta + 0.8$. At zero, $\theta' = 0.8$.

Outer chain rule. The query derivative at the adapted parameter is $0.8 - 3 = -2.2$. The derivative of adaptation is 0.6. Therefore the exact meta-gradient is $-2.2 \cdot 0.6 = -1.32$.

First-order approximation. Ignoring the adaptation Jacobian returns -2.2. The approximation changes the update magnitude here. For a matrix Hessian, it can change direction as well. A small inner step is relevant because it makes $I - \alpha H$ closer to identity, but the Hessian scale must be considered along with the raw step size.

Learning the inner rate. At $\theta = 0$, $\partial\theta'/\partial\alpha = 8$, so $\partial L_Q/\partial\alpha = (-2.2)8 = -17.6$. Increasing the inner rate locally reduces query loss in this example. That conclusion is local; a sufficiently large rate can overshoot.

T.22 Worked case: a VAE loss with all constants identified

Return to lecture: Lecture 23: VAE architecture and loss

Problem. For one latent coordinate, take $q(z \mid x) = \mathcal{N}(1, 0.25)$, prior $p(z) = \mathcal{N}(0, 1)$, and decoder $p(x \mid z) = \mathcal{N}(2z, 1)$. Let observed $x = 3$. Compute the negative ELBO exactly.

Expected reconstruction term. The negative log likelihood is $\frac{1}{2}(3 - 2z)^2 + \frac{1}{2}\log(2\pi)$. Under q, the residual has mean1 and variance $4 \cdot 0.25 = 1$. Therefore $\mathbb{E}(3-2z)^2 = 1^2 + 1 = 2$, and the expected negative log likelihood is $1 + \frac{1}{2}\log(2\pi) \approx 1.919$.

Prior KL. The Gaussian formula gives

$$D_{\mathrm{KL}}(q\|p) = \frac{1}{2}(1 + 0.25 - 1 - \log 0.25) \approx 0.818.$$

The total negative ELBO is about 2.737.

Check the bound. The generative marginal is $x \sim \mathcal{N}(0, 5)$, so its negative log likelihood at 3 is $\frac{1}{2}\log(10\pi) + 9/10 \approx 2.624$. The negative ELBO is larger by about 0.113, equal to the KL from the chosen approximate posterior to the exact posterior $\mathcal{N}(1.2, 0.2)$.

Why constants matter. Constants can be omitted when optimizing fixed-variance decoder parameters, but must be retained when comparing the numerical bound to a true likelihood. The example separates reconstruction expectation, prior KL, marginal likelihood, and posterior approximation gap rather than calling all of them "the VAE loss."

T.23 Worked case: reverse a Gaussian corruption step

Return to lecture: Lecture 27: Gaussian reverse-process argument

Problem. Let $x_0 \sim \mathcal{N}(0, 1)$ and $x_t = x_0 + \sqrt{t}\epsilon$. At $t = 3$, observe $x_t = 4$. Compute the clean conditional mean, score, and the reverse conditional mean and variance for an interval $h = 1$.

Clean posterior. Gaussian conditioning gives $D^* = 4/(1 + 3) = 1$ and posterior variance $3/4$. The score is $(1 - 4)/3 = -1$, also obtained directly from the noisy marginal $\mathcal{N}(0, 4)$.

Reverse mean. The Brownian-bridge formula gives

$$\mathbb{E}[x_2 \mid x_3 = 4] = 4 + \frac{1}{3}(1 - 4) = 3.$$

The reverse conditional variance is

$$h\frac{t-h}{t} + \frac{h^2}{t^2}\operatorname{Var}(x_0 \mid x_t) = \frac{2}{3} + \frac{1}{9}\frac{3}{4} = \frac{3}{4}.$$

In this Gaussian-data case the full reverse conditional is Gaussian, so these two moments characterize it exactly.

Deterministic marginal transport. The marginal variance at time3 is 4 and at time2 is 3. A deterministic map preserving the Gaussian marginal is $x_2 = \sqrt{3/4}\, x_3$, which maps4 to approximately3.464. This differs from the reverse conditional mean3. Both constructions can be correct because they specify different couplings between times.

Interpretation. A denoiser estimates a conditional average. A stochastic reverse sampler adds conditional uncertainty. A deterministic probability-flow sampler transports an entire marginal distribution. Confusing these three objects is the source of many apparent contradictions in diffusion derivations.

Supplement U

Reference, corrections, and a study route

U.1 A compact notation guide

Symbol	Meaning in this reader
θ, W, b	Trainable parameters, a weight matrix, and a bias
$x, y, \hat{y}$	Input, observed target, and prediction
L, ℓ	Aggregate loss and an individual-example loss
g, H, J	Gradient, Hessian, and Jacobian
η, α	Step-size symbols; α_t has a separate diffusion meaning
λ, γ	Eigenvalues or regularization constants, defined locally
$U\Sigma V^\top$	Singular value decomposition
$\|W\|_2$	Matrix spectral norm; for a vector, Euclidean norm
$\|W\|_F$	Frobenius norm
$\|x\|_{\mathrm{RMS}}$	$\|x\|_2/\sqrt{d}$ for a d-coordinate vector
A, B, C, D	State-space matrices; LoRA reuses A, B locally
Q, K, V	Query, key, and value matrices in attention
p, q, π	Probability distributions; arguments identify their roles
$D_{\mathrm{KL}}(p\|q)$	Expectation under the first distribution of $\log(p/q)$
f, G, L, μ, κ	In optimization bounds: objective, function-Lipschitz constant, gradient-Lipschitz constant, strong-convexity constant, and L/μ (when $\mu > 0$)
G_W	Full gradient with respect to a weight matrix; distinct from the scalar Lipschitz bound G
μ, Σ	In Gaussian formulas: mean and covariance; μ is not a convexity constant there

Symbol	Meaning in this reader
$\mathbb{E}[Y \mid X], \mathrm{Cov}$	Conditional expectation and covariance; conditioning is part of the quantity
α, s, r	Locally: sequence power, LoRA scale, and LoRA rank or proposal distribution; inspect the defining equation
sg	Stop-gradient: identity value, zero derivative
ϵ	Noise, a tolerance, or PPO clip width, always defined locally

The reader uses column vectors for ordinary linear layers and token rows for attention matrices, explicitly declaring the convention at the transition. A squared loss includes a factor of one-half unless stated otherwise. Probability calculations use natural logarithms except where a discussion explicitly asks for bits. Constants from another convention must be converted before comparison. In particular, L denotes loss in some chapters and a smoothness constant in optimization bounds; f names the objective in those bounds. A symbol is local to its explicitly stated role, not globally interchangeable.

U.2 Corrections and qualifications applied in the exposition

The original PDFs are preserved unchanged. The following distinctions are applied in the newly written explanations. The first group comes from faculty/staff updates on Ed; the second consists of mathematical checks performed while preparing this reader.

Recorded course updates. Lecture 3's Ed post explains that the expected Lyapunov contraction needs a uniform factor strictly below one; monotonic decrease alone is insufficient. Lecture 11's posted PDF includes an office-hours clarification about downsampling paths that is not in the original recording. Lecture 13's posted PDF includes later annotations about graph pooling. Lecture 25's Ed correction says the reward objective is maximized and the minimum caps large importance weights; the oral explanation had a slip. Discussion 10's NoPE problem was updated so its third embedding coordinate is arbitrary.

Quadratic loss normalization. In the optimization draft's unhalved least-squares convention, $H = 2X^\top X$. The optimal constant step for worst-case spectral contraction is $1/(\lambda_{\min}(X^\top X) + \lambda_{\max}(X^\top X))$. Using $2/(\lambda_{\min} + \lambda_{\max})$ with that unhalved loss doubles the intended step. The reader derives both conventions rather than silently copying the draft's equation 24.

Normalized descent and state recovery. The optimization draft's equation 56 has a positive gradient direction where minimizing the linearized loss requires the negative direction. A norm ball used to attain the minimizing step should be closed; an open ball generally has only an infimum. Discussion 8's use of C^{-1} additionally requires invertibility, whereas general state recovery uses observability across time. Discussion 9's prefix-product quotient requires nonzero denominators; the affine scan in this reader avoids that division.

Error recurrence. The quadratic recurrence applies the matrix $(I - \eta H)$ to the current error vector. The error factor is missing or misplaced in parts of the draft's equations 20–21; the reader writes $e_{t+1} = (I - \eta H)e_t$ and diagonalizes that expression directly.

Ridge and learning-rate bounds. Ridge improves the condition number by lifting small eigenvalues, but it also raises the largest eigenvalue. Hence the largest stable absolute learning rate decreases. Better conditioning does not imply a larger permissible absolute step.

Constant-step SGD. Exact contraction to an interpolating solution relies on conditions that remove gradient noise at that solution, together with sufficient identifiability on the analyzed subspace. General noisy objectives instead exhibit a constant-step error floor. A decreasing expected distance should not be claimed without checking the noise term.

Rectangular matrix polynomials. An ordinary odd polynomial in an arbitrary non-symmetric matrix does not generally act independently on its singular values. The correct Newton–Schulz expressions use alternating transpose products such as $XX^\top X$. Zero singular values stay zero.

Maximal-update reasoning. An upper bound on feature-update norms controls growth. It does not by itself prove nonvanishing useful learning; alignment and nondegeneracy assumptions are needed for a lower bound. The reader separates these claims.

State-space stability. An eigenvalue of modulus one lies on a discrete-time stability boundary. Calling it "critical damping" is generally inappropriate: critical damping is a different second-order dynamical notion. Jordan blocks, forcing, and observable/controllable directions can matter beyond eigenvalue magnitudes alone.

Binary noising. The fair-coin replacement interpretation uses a replacement probability 2δ, so it requires $0 \le \delta \le 1/2$. The discussion assumes a small positive flip probability; the reader makes the range explicit. Pixelwise marginal convergence alone does not establish independence, so the reader also gives a joint coupling/union-bound argument.

Reverse diffusion. A reverse conditional mean is not the same as a deterministic map transporting the full marginal distribution. In the additive Brownian convention the probability-flow drift has half the score coefficient of the stochastic reverse drift. This is demonstrated with a solvable point-mass example.

Single-example contribution versus unbiased SGD. Discussion 1, pages 2–3 writes a single summand of a mean-loss gradient with its $1/N$ factor. Under uniform sampling, an unbiased stochastic gradient is N times that summand. For $N^{-1}\sum_i(\theta - y_i)^2$, use $2(\theta - y_I)$ with I uniform. Keeping the extra $1/N$ instead reduces the expected step by N; it is a different learning-rate convention.

Residual Jacobians. Discussion 6, page 2 overstates the identity shortcut as a lower bound of one on gradient size. The reader's counterexample $J_F = -I$ gives $I + J_F = 0$. If $\|J_F\|_2 \le \varepsilon < 1$, a valid bound is $\|(I + J_F)v\|_2 \ge (1 - \varepsilon)\|v\|_2$, by the reverse triangle inequality. Also, the dropout penalty coefficient $\sqrt{(1-p)/p}$ on page 4 is exactly 3 at $p = 0.1$, not 3.16.

Signed updates and singular values. Discussion 4, page 1 prints a plus sign before the loss-gradient sign; minimizing the displayed loss requires a minus sign. With independent

standard-normal inputs and nonzero downstream coordinates, the exact squared SignGD feature-update RMS expectation is $\eta^2[d+d(d-1)2/\pi]$, which is asymptotically proportional to d^2 at fixed η. Ordinary singular values cannot be negative: a negative scalar coefficient after an unnormalized Newton–Schulz iteration represents a sign change absorbed into singular vectors. The normalized nonnegative regime avoids that ambiguity.

Statistical summaries on log axes. Discussion 2's notebook guidance describes medians and quartiles. These are useful for skewed positive errors, but a logarithmic axis does not make an arithmetic mean invalid. State whether the center is a mean, median, or geometric mean and what the interval represents. A mean-minus-standard-deviation endpoint can be nonpositive and hence unplottable on a log axis; that plotting issue is different from the statistic's definition.

Scan work and shortest paths. Discussion 7's direct scalar recurrence uses two arithmetic operations per step. A work-efficient parallel scan has $O(T)$ work and $O(\log T)$ depth under a suitable elementary-operation model, but need not use exactly $2T$ operations. For synchronous shortest-path relaxation, count edges on a minimum-weight path, not the numerical weighted distance or the unweighted diameter. A minimum-weight path may use more hops than a minimum-hop path. Dijkstra's priority-queue schedule is a different computation from synchronous message passing.

Attention counts and position. Discussion 10's hard-attention receptive-field count $(h+1)^n$ is an upper bound on distinct selected value-path ancestors, capped by sequence length; paths can overlap. Selection scores themselves may depend on other keys. Its output projection acts on every token, so the example count includes Mhd^2, not just hd^2. The stated big-O order is unaffected. The NoPE construction is an existence result; it does not prove that optimization discovers it or that finite precision supports unlimited lengths.

Newly restored examples. Sections 21.3–M.7 correct prompt arithmetic and distinguish LoRA initialization hypotheses. Section Q.6 retains proposal ratios and allows later cuts to revise earlier generated blocks. Section E.5 distinguishes Cauchy symmetry from a finite mean. Section S.8 corrects the late binary reverse-target limit. The VAE quantity written as reconstruction log likelihood minus KL is an ELBO to maximize; its negative is the loss to minimize. KL regularization alone does not mathematically select only low spatial frequencies.

These notes are mathematical clarifications, not modifications to the archived course files. If comparing a lecture recording with the reader, consult the original PDF and the linked Ed post to distinguish a faculty update, a notation change, and an added explanatory derivation.

U.3 A dependency-based study route

Use prerequisites to diagnose confusion. If LoRA gradients are unclear, return to the dense-layer chain rule and low-rank products. If diffusion's reverse mean is unclear, return to conditional expectation and Gaussian conditioning. If a policy-gradient expression is unclear, return to the score-function identity and the distribution under which the expectation is taken.

The map highlights useful routes through the material; it is not a requirement to finish every topic in one branch before opening another.

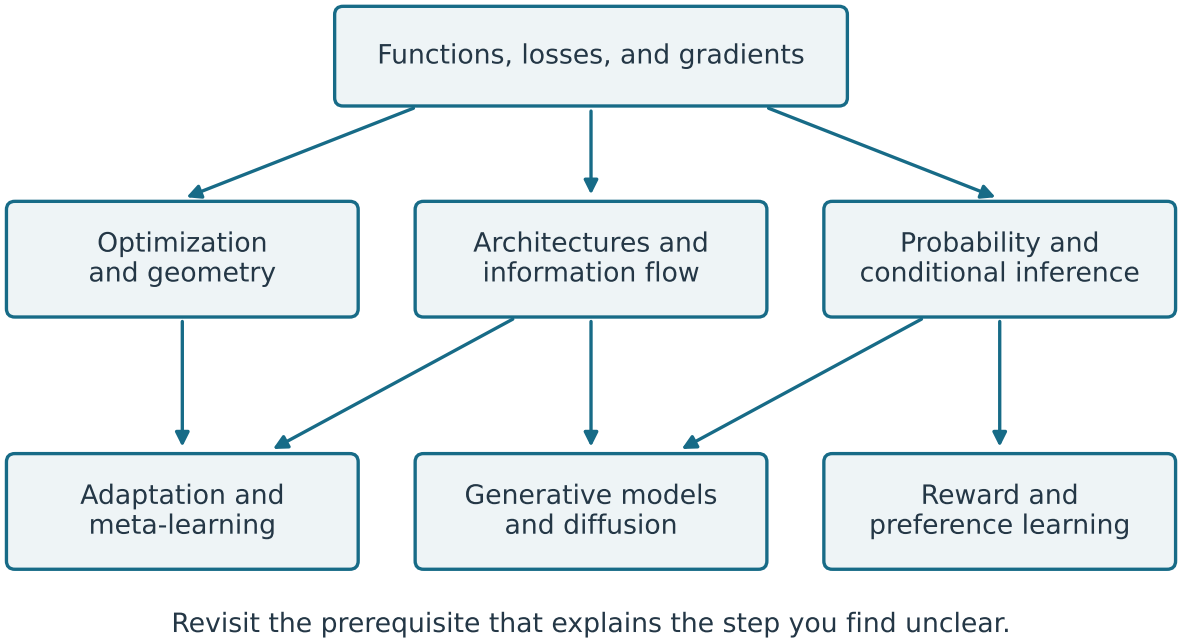


Figure U.1: A conceptual route through the course. Arrows indicate useful dependencies, not a strict lecture calendar or an exhaustive prerequisite graph.

First pass: optimize a function. Read Supplements A–E and complete discussions 1–4. Derive one ReLU gradient, one quadratic convergence bound, one dual-norm update, and one width-scaling rule without notes. This establishes the mathematical vocabulary for the remainder.

Second pass: choose information flow. Read Supplements F–L and complete discussions 5–10. Trace a single input through a convolution, a graph layer, a recurrence, an SSM kernel, and a causal attention layer. In each case identify the receptive field, shared parameters, tensor shapes, and computational bottleneck.

Third pass: adapt representations. Read Supplements M–N, then the adaptation and forgetting portions of discussions 11–12. Contrast changes in context with changes in weights. Differentiate one LoRA step and one MAML inner update explicitly.

Fourth pass: model and change distributions. Read Supplements O–S and complete discussion 13. Derive the ELBO, the policy-gradient identity, the KL-regularized optimal policy, and the denoiser-score relation. These derivations share the habit of naming the sampling distribution before manipulating an expectation.

Review pass: compare procedures rather than names. For every method, write the forward computation, loss, gradient estimator, update rule, and inference procedure. Two

methods may share a forward expression while differentiating it differently, or share a trained model while sampling from it differently.

U.4 Integrative worked questions

Why can normalization and residual connections help an optimizer without changing the task loss? They change the parameter-to-function map and its Jacobians. The same prediction loss can therefore have different conditioning and gradient paths. Loss design and parameterization are independent levers.

Why is an SSM sometimes a convolution and sometimes not? A fixed linear time-invariant recurrence has an impulse response independent of the input. Input-dependent selective parameters generally destroy that fixed-kernel property, although the affine recurrence may still admit an associative scan.

Why does low reconstruction error not guarantee good unconditional generation? Reconstruction evaluates latents produced from observed examples. Generation samples from a prior. If those latent distributions do not align, a decoder can reconstruct well but behave poorly on prior samples. The VAE explicitly couples these distributions through its objective.

Why can clipping stabilize an update while changing its target? Importance weights correct a sampling mismatch exactly only when retained as specified. Capping them reduces the influence of rare large ratios, changing the estimator's expectation. The resulting bias may be a deliberate variance tradeoff, but it should be acknowledged.

Why can a deterministic diffusion sampler still generate diverse outputs? The initial noise is random. Conditional on it, the trajectory may be deterministic; across initial draws, the outputs vary. Randomness at every step is not necessary for a random final sample.

U.5 How to use the archived transcripts

Transcript files are indexed by playlist position, date, and video ID. Timestamps allow returning to the recording. The supplied captions contain recognition errors, especially for names, symbols, and technical terms; equations should be checked against the faculty PDF. Repeated rolling caption text was removed when converting the SRT exports, while the original exports were retained for comparison.

The recordings and handouts do not form a strict one-to-one sequence around the self-supervision/state-space transition. Playlist indices therefore identify videos, not asserted lecture numbers. The source register below keeps handout numbering and recording order separate.

U.6 Alphabetical concept index

Entries link to an explanation and its printed page number. Related derivations and worked cases are cross-referenced within those sections.

Concept	Section; page
A numerical DPO example	R.6; p. 467
Adam	3.2; p. 42
AdamW	3.2; p. 42
Adapter composition	M.7; p. 434
Associative scan	J.4; p. 415
Associative scans preserve a different kind of parallelism	J.4; p. 415
Attention, gradients	K.1; p. 416
Attention, matrix shapes	17.6; p. 221
Autoregressive training	20.1; p. 256
Backpropagation	A.2; p. 342
Backpropagation through time	14.3; p. 177
BatchNorm	9.6; p. 117
Bayes rule	O.1; p. 440
Beam search	24.5; p. 303
Beam search with explicit termination	Q.3; p. 459
Binary cross-entropy, stable evaluation	A.7; p. 349
Binary diffusion, late targets	S.8; p. 474
Blockwise power sampling	Q.6; p. 461
Cauchy distribution	E.5; p. 380
Clipping, gradients	A.7; p. 351
Conditional expectation	O.2; p. 441
Convexity	B.4; p. 355
Convexity: chords, tangents, and global optimality	B.4; p. 355
Convolution, kernel sharing	F.2; p. 383
Convolution, locality	8.4; p. 99
Covariance, total	O.3; p. 442
Cross-entropy	O.6; p. 444
Data splitting, graphs	H.2; p. 395
Decoding, power distributions	Q.6; p. 461
Denoising, conditional mean	O.2; p. 441
Derive the Gaussian posterior by completing the square	P.3; p. 452
Deriving the trajectory policy gradient	R.2; p. 464
DiffPool	13.2; p. 165
Discussion 12 — information and forgetting	T.12; p. 481
DPO	26.2; p. 318
Dropout	G.2; p. 388
Early stopping	2.1; p. 24

Concept	Section; page
Eigenvalues, gradient descent	1.4; p. 20
ELBO	P.4; p. 454
Embedding adaptation	M.7; p. 434
Entropy	O.6; p. 444
Equivariance versus invariance	F.1; p. 382
Expectation, interchanging derivatives	O.5; p. 444
Feature learning	6.4; p. 78
Floating-point arithmetic	A.7; p. 348
Forgetting	T.12; p. 481
Forgetting as a multi-objective tradeoff	N.5; p. 439
From pairwise preferences to DPO	26.2; p. 318
Gradient accumulation	A.7; p. 350
Graph message passing	12.5; p. 156
Greedy decoding, sampling, and beam search	24.5; p. 303
He initialization	4.4; p. 57
Heavy tails	E.5; p. 380
Implicit bias	2.1; p. 24
Importance ratios and the course's clipped surrogate	24.6; p. 308
Importance sampling	O.10; p. 448
Importance sampling and the meaning of stop-gradient	O.10; p. 448
Importance weighting, support, and effective sample size	O.14; p. 450
Independence and covariance	O.1; p. 440
Jensen inequality	O.4; p. 443
Kalman filter	14.4; p. 182
KL divergence	O.7; p. 445
KV cache	L.1; p. 421
L-smoothness	B.6; p. 357
L-smoothness bounds changes in the gradient	B.6; p. 357
LayerNorm	9.6; p. 117
Least squares	1.4; p. 19
Lipschitz continuity	B.5; p. 357
Lipschitz continuity bounds changes in function value	B.5; p. 357
Local linearization	3.3; p. 47
Log-sum-exp	A.7; p. 348
LoRA, initialization	M.6; p. 433
Loss reduction, sum versus mean	A.7; p. 350
Loss scaling	A.7; p. 351
Masked rows, numerical handling	A.7; p. 349
Maximal updates	6.4; p. 78
Metropolis–Hastings, proposal ratio	Q.6; p. 461
Momentum	2.3; p. 32
Monte Carlo variance	O.5; p. 444

Concept	Section; page
Multi-query attention	18.4; p. 230
Muon	7.2; p. 85
Newton–Schulz	7.2; p. 85
Normalization axes	9.6; p. 117
Numerical stability	A.7; p. 348
Optimization geometry	4.2; p. 52
PCA and linear autoencoders	I.1; p. 404
Position information	19.2; p. 238
PPO	O.5; p. 444
PPO clipping and stopped weighting have different derivatives	R.4; p. 465
Prefix parameter counts	21.3; p. 269
Probability, joint and conditional	O.1; p. 440
Receptive fields	8.5; p. 103
ReLU representation	A.1; p. 342
Reparameterization	O.9; p. 447
Reparameterization moves randomness outside the parameters	O.9; p. 447
Residual Jacobians	G.1; p. 388
Ridge regression	1.5; p. 22
RMS norm	6.1; p. 73
RMSNorm	9.6; p. 117
Selectivity changes the algorithmic opportunity	16.3; p. 207
Self-supervision	14.5; p. 185
Shampoo	D.1; p. 373
SignGD width scaling	6.3; p. 75
Smooth convex gradient descent has a 1/k gap bound	B.10; p. 362
Soft prompts	21.3; p. 269
Softplus, stable evaluation	A.7; p. 349
Standard PPO is a useful comparison, with a different formula	R.1; p. 463
State-space models	16.1; p. 202
State-space stability	16.1; p. 202
Stationarity	B.13; p. 364
Stochastic gradient descent	2.2; p. 29
Strong convexity	B.7; p. 359
Strong convexity supplies a positive lower curvature	B.7; p. 359
Strong convexity turns descent into geometric convergence	B.11; p. 363
Subgradients	B.12; p. 364
Support, moving boundaries	O.5; p. 444
The Brownian bridge gives an exact conditional mean	S.2; p. 470
The ELBO is exact likelihood minus an inference gap	P.4; p. 454
The score-function identity	O.8; p. 446
The support/query split defines the task	21.4; p. 273
Total variance	O.3; p. 442

Concept	Section; page
Tower property	O.2; p. 441
Transposed convolution	11.4; p. 142
U-Net	11.5; p. 145
Variance reduction for score-function estimators	O.13; p. 449
Without convexity: descent can certify stationarity only	B.13; p. 364
Without smoothness: a subgradient guarantee	B.12; p. 364
Worked case: a VAE loss with all constants identified	T.22; p. 488
Worked case: one exact meta-gradient and one approximation	T.21; p. 487
Xavier initialization	4.4; p. 57

Supplement V

Source register

This register separates handout numbering from playlist order. JSON manifests in `sources/` retain the original attachment URLs, posting identities, Ed links, and SHA-256 hashes. The public Fall 2025 course schedule supplies the course context; the actual archived notes came from the signed-in course Ed discussion.

The lecture chapters follow the original reader's sequence 0–24, 26, 27 (27 chapters). Its lecture 24 contains faculty handout 25's sampling and reward material; the skipped heading is not a missing archived handout. Around self-supervision and state-space models, recording and handout boundaries also overlap. Each lecture therefore links its dated transcript and the relevant faculty PDFs separately. Lecture 0 has a recording but no separate faculty PDF in the inspected Ed post.

The lecture narrative retains the original reader's topics and historical context, with targeted corrections to mathematical statements and attribution. The original reader is not treated as an authority. Supplements extend specific lecture sections with further derivations, worked examples, counterexamples, and practice. They assume the linked lecture setup; only notation and assumptions needed for the additional calculation are restated. Captions are not a manually verified transcription; the faculty PDFs remain the reference when caption terminology is unclear. Original files and attribution are preserved unchanged.

V.1 Faculty lecture PDFs

Lecture	Faculty PDF	Posted by	Ed post
01 sahai	Open PDF	Anant Sahai	#28
01 ranade	Open PDF	Gireeja Ranade	#28
02	Open PDF	Gireeja Ranade	#58
03	Open PDF	Gireeja Ranade	#58
04	Open PDF	Gireeja Ranade	#79
05	Open PDF	Gireeja Ranade	#79
06	Open PDF	Gireeja Ranade	#90
07	Open PDF	Gireeja Ranade	#90
08	Open PDF	Anant Sahai	#109
09	Open PDF	Anant Sahai	#109
10	Open PDF	Anant Sahai	#108
11	Open PDF	Anant Sahai	#132
12	Open PDF	Anant Sahai	#136
13	Open PDF	Anant Sahai	#157
14	Open PDF	Anant Sahai	#161
15 sahai	Open PDF	Anant Sahai	#174

Lecture	Faculty PDF	Posted by	Ed post
16	Open PDF	Gireeja Ranade	#237
17	Open PDF	Gireeja Ranade	#237
18	Open PDF	Gireeja Ranade	#237
19	Open PDF	Gireeja Ranade	#237
20	Open PDF	Gireeja Ranade	#237
21	Open PDF	Anant Sahai	#297
22	Open PDF	Anant Sahai	#297
23	Open PDF	Anant Sahai	#298
24	Open PDF	Anant Sahai	#307
26	Open PDF	Anant Sahai	#445
25	Open PDF	Anant Sahai	#328
27	Open PDF	Anant Sahai	#498

The optimization draft is available locally, posted by Gireeja Ranade in Ed #949. The reader's reference chapter explains corrections and convention changes rather than altering this draft.

V.2 Official discussion materials

Set	Questions	Solutions	Posting identity
1	Embedded in solutions	Solutions	Anant Sahai
2	Questions	Solutions	Sultan Daniels
3	Questions	Solutions	Hong Joey
4	Questions	Solutions	Hong Joey
5	Questions	Solutions	Lance Mathias
6	Questions	Solutions	Hong Joey
7	Questions	Solutions	Sultan Daniels
8	Questions	Solutions	Hong Joey
9	Questions	Solutions	Lance Mathias
10	Questions	Solutions	Lance Mathias
11	Questions	Solutions	Lance Mathias
12	Questions	Solutions	Sultan Daniels
13	Questions	Solutions	Lance Mathias

V.3 Recording transcripts in playlist order

The CS182 Fall 2025 playlist contains the 27 recordings below. The caption exports were collected from the Ed central index. Filenames encode playlist position, recording date, and YouTube video ID; they are not inferred lecture numbers.

Position	Recording date	Video	Local transcript
01	2025-08-28	Watch	Transcript
02	2025-09-02	Watch	Transcript
03	2025-09-04	Watch	Transcript
04	2025-09-09	Watch	Transcript
05	2025-09-11	Watch	Transcript
06	2025-09-16	Watch	Transcript
07	2025-09-18	Watch	Transcript
08	2025-09-23	Watch	Transcript
09	2025-09-25	Watch	Transcript
10	2025-09-30	Watch	Transcript
11	2025-10-02	Watch	Transcript
12	2025-10-07	Watch	Transcript
13	2025-10-09	Watch	Transcript
14	2025-10-14	Watch	Transcript
15	2025-10-16	Watch	Transcript
16	2025-10-21	Watch	Transcript
17	2025-10-23	Watch	Transcript
18	2025-10-28	Watch	Transcript
19	2025-10-30	Watch	Transcript
20	2025-11-04	Watch	Transcript
21	2025-11-06	Watch	Transcript
22	2025-11-13	Watch	Transcript
23	2025-11-18	Watch	Transcript
24	2025-11-20	Watch	Transcript
25	2025-11-25	Watch	Transcript
26	2025-12-02	Watch	Transcript
27	2025-12-04	Watch	Transcript

The transcript manifest records cue counts, cleaned word counts, and the first and last caption timestamps. The first caption may begin after the recording starts, and silence or uncaptioned portions are not reconstructed. The original subtitle exports are retained for auditing.