



I have uploaded lecture number four. In lecture number four we covered the topics of Learning a generative model Setting goal of learning What is best? Learning as density estimation KL divergence expected log-likelihood Maximum likelihood. Monte Carlo estimation. Monte Carlo estimation properties Example MLE scoring for coin example. MLE principle for autoregressive model MLE learning gradient descent MLA learning sophisticated gradient descent Empirical risk of overfitting Bias-variance tradeoff How to avoid overfitting? Conditional generative models And finally the recap and summary. These are the topics we have learned. Ask question relevant to the notes structure, notes formatting and notes detail. Before Creating The notes for me.

Perfect! I've reviewed your Lecture 4 material on Maximum Likelihood Learning. Before I create detailed notes for you, let me ask you some important questions about structure, formatting, and detail level:

Questions About Note Structure:

1. Organization & Flow

- Would you prefer **topic-by-topic sections** (e.g., KL-Divergence → MLE → Gradient Descent) or a **concept-flow structure** (e.g., Problem → Solution → Implementation)?
- Should I include a "**Big Picture**" section at the beginning that connects all topics before diving into details?

2. Mathematical Detail Level

Looking at topics like KL-divergence, Expected Log-Likelihood, and Gradient Descent:

- Do you want **step-by-step mathematical derivations** with explanations between steps?
- Should I include **intuitive explanations first**, then formal math?
- How much emphasis on **proofs** vs **understanding when/why to use**?

3. Examples & Applications

The lecture has the coin-flipping example:

- Should I **expand existing examples** with more calculations?
- Do you want **additional real-world examples** (e.g., image generation, text modeling)?
- Would you like **practice problems** or **self-check questions** included?

4. Visual Elements

- Should I describe the **graphs/figures** from slides in detail (e.g., bias-variance curves)?
- Do you want **flowcharts** for algorithms (Gradient Descent, SGD)?
- Should I include **comparison tables** (e.g., KL-divergence properties, overfitting solutions)?

5. Connection & Context

- Should I link back to **Lecture 3 (Autoregressive Models)** concepts?
- Do you want a section showing "**Why This Matters for Generative Models**"?

6. Depth Focus

Which topics need **extra detail vs brief summary**?

- Core concepts needing deep understanding?
- Topics you find confusing that need multiple explanations?

7. Format Preferences

- Bullet points, paragraphs, or **mixed format**?
- Should key equations be **highlighted/boxed**?
- Do you want **summary boxes** after each major section?

Please answer these questions so I can create the most effective notes for your learning style! ☺

One topic at a time, give a bigger picture and then Move towards the topics. Mathematical implementation and define the formula. in detail before we using in an example Example real world example should be Mathematical example should be Entertain Like practice problems or There is no self-question required. Explain the graph and figures in detail Draw flowcharts and also if required. Raw comparison table as well. Yes also link back lecture number 3, 2 and 1 if it is required. No, I don't want why this matter. Alternative models Don't add this section. Just a summary. It is a new topic then Hi. Discuss the detail. If it is require not much detail Then brief summary will be good. Eh? Also use bullet points and paragraphs and numbering ordered and ordered lists. Highlighted Well manner well-organized And keep the flow. So I can understand it very well. Go ahead.

Then we will move towards the next topic. Okay, don't go all at once. One topic at a time, step-by-step in detail, and keep the flow going in that manner so that all the knowledge would connect the previous

Perfect! I understand completely. Let me start creating your notes **one topic at a time, step-by-step**. We'll build connected knowledge progressively.

□ LECTURE 4: MAXIMUM LIKELIHOOD LEARNING

Course: CS 844 - Generative Deep Models

□ BIG PICTURE OVERVIEW

Before diving into individual topics, let's understand **where we are** and **where we're going**:

The Journey So Far (Connecting Previous Lectures)

From Lecture 1-3, we learned:

- **Lecture 1:** What generative models are - models that can generate new data samples
- **Lecture 2:** Background on probability distributions, Bayesian networks, neural models
- **Lecture 3: Autoregressive (AR) Models** - how to represent $p(x)$ by breaking it into conditional distributions: $p(x) = \prod_{i=1}^n p(x_i|x_{<i})$

The Missing Piece: How Do We LEARN These Models?

In Lecture 3, we learned **how to represent $p(x)$** using autoregressive factorization. But we didn't answer:

- **Given training data** (e.g., 10,000 dog images), how do we find the **parameters θ** of our model?
- How do we make sure our model $P_\theta(x)$ **matches the real data distribution $P_{data}(x)$** ?

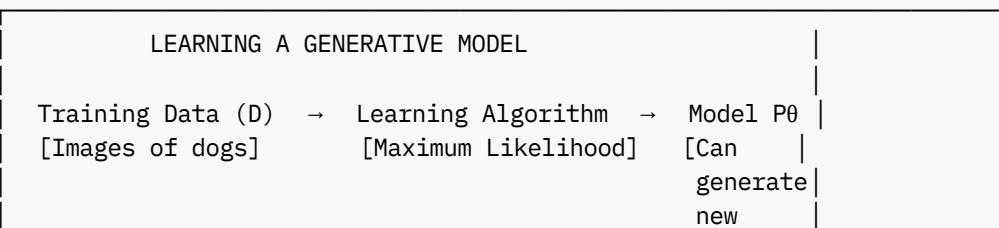
What Lecture 4 Solves

This lecture answers the fundamental question: "**How to LEARN generative models from data?**"

The Path We'll Take:

1. PROBLEM SETUP
↓
2. DEFINING "GOOD" (What makes a model good?)
↓
3. MEASURING CLOSENESS (KL-Divergence)
↓
4. THE SOLUTION (Maximum Likelihood Estimation)
↓
5. PRACTICAL IMPLEMENTATION (Gradient Descent)
↓
6. CHALLENGES (Overfitting, Bias-Variance)
↓
7. SOLUTIONS (Regularization, Validation)

VISUAL ROADMAP



Key Question: What learning algorithm should we use?

Now let me start with **TOPIC 1** in complete detail.

□ **TOPIC 1: LEARNING A GENERATIVE MODEL**

(Introduction & Core Objectives)

1.1 What Are We Trying to Achieve?

The Goal:

Given a **training dataset D** containing examples (e.g., images of dogs), we want to learn a **probability distribution $p(x)$** over the data space.

Why Learn a Probability Distribution?

Once we have $p(x)$, we can perform **three powerful tasks**:

□ Task 1: GENERATION (Sampling)

Definition:

- Sample new data points: $x_{new} \sim p(x)$
- The generated x_{new} should **look like it came from the training data**

Real-World Example:

- Training data: 10,000 images of dogs
- After learning $p(x)$: Generate new, realistic dog images that don't exist in the training set
- **Application:** Image synthesis, data augmentation, creative AI art

Mathematical Formulation:

$$x_{new} \sim P(x) \implies P(x_{new}) \text{ should be high}$$

Connection to Lecture 3:

- Remember **autoregressive sampling**? We sample x_1 , then $x_2|x_1$, then $x_3|x_1, x_2$, etc.
- Now we're learning the **parameters** of those conditional distributions!

II Task 2: DENSITY ESTIMATION

Definition:

- Given any data point x , compute $p(x)$
- High $p(x) \rightarrow x$ looks like training data
- Low $p(x) \rightarrow x$ is unusual/anomalous

Real-World Example:

- **Anomaly Detection in Manufacturing:**

- Training data: Images of non-defective products
- New image arrives: $p(x) = 0.92 \rightarrow$ Normal product ✓
- Another image: $p(x) = 0.03 \rightarrow$ Defective! Flag for inspection ✗

Mathematical Formulation:

$$p(x) = \begin{cases} \text{High} & \text{if } x \text{ is similar to training data} \\ \text{Low} & \text{if } x \text{ is anomalous} \end{cases}$$

Real Application Example:

- **Medical Imaging:** Train on healthy brain scans \rightarrow New scan with $p(x)$ very low \rightarrow Potential tumor detected

II Task 3: UNSUPERVISED REPRESENTATION LEARNING

Definition:

- Learn **features** that the data has in common
- Extract meaningful patterns without labels

Real-World Example:

- Training data: Dog images
- The model learns internal representations:
 - "Dogs have ears" (feature 1)
 - "Dogs have tails" (feature 2)
 - "Dogs have four legs" (feature 3)
 - Texture patterns for fur (feature 4)

Why This Matters:

- These learned features can be **transferred** to other tasks
- Example: Use the learned "ear detector" for a dog breed classifier

Mathematical Context:

- The hidden layers of neural networks in $p_{neural}(x_i|x_{<i}; \theta)$ learn these features
- **Recall Lecture 3:** NADE, PixelCNN use neural networks - their hidden layers capture features!

1.2 The Two Fundamental Questions

Our journey has **two stages**:

Question 1: How to REPRESENT $p(x)$?

✓ **ANSWERED IN LECTURE 3**

- Use **Autoregressive Models**
- Factorize:

$$p(x) = \prod_{i=1}^n p(x_i|x_{<i}; \theta_i)$$

- Implement with: FVSBN, NADE, MADE, PixelRNN, PixelCNN, Transformers

Question 2: How to LEARN $p(x)$ from data?

□ **THIS LECTURE'S FOCUS**

Given:

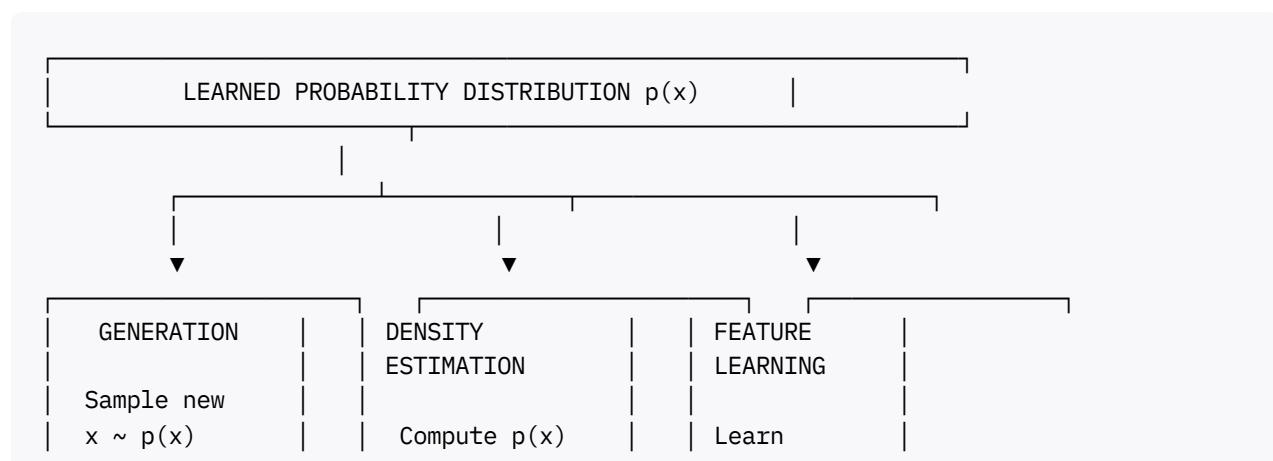
- Training data $D = \{x^{(1)}, x^{(2)}, \dots, x^{(m)}\}$
- Model family with parameters θ

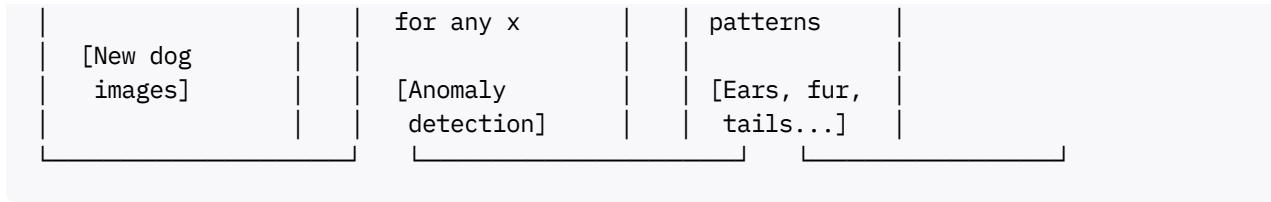
Find:

- The **best parameters** θ^* that make our model fit the data

This is what Maximum Likelihood Learning solves!

1.3 Visual Summary: Three Capabilities





1.4 Practical Example: Dog Image Generator

Let's make this concrete with a **complete example**:

Setup:

- **Data:** $D = \{ 50,000 \text{ dog images} \}$
- **Image size:** $64 \times 64 \text{ pixels, RGB} \rightarrow n = 64 \times 64 \times 3 = 12,288 \text{ variables}$
- **Model:** Autoregressive with neural networks (like PixelCNN from Lecture 3)

After Learning (What We Can Do):

1. Generation:

```
Sample x_new ~ p(x)
Result: New dog image that never existed before
```

2. Density Estimation:

Input: Image of a dog	$\rightarrow p(x) = 0.89$ (high - looks like a dog)	✓
Input: Image of a cat	$\rightarrow p(x) = 0.12$ (low - doesn't fit)	✗
Input: Random noise	$\rightarrow p(x) = 0.001$ (very low - nonsense)	✗

3. Feature Learning:

```
Hidden layer neurons learn:
- Neuron 47: Activates for "floppy ears"
- Neuron 103: Activates for "wet nose"
- Neuron 256: Activates for "fur texture"
```

1.5 Summary Table: Three Tasks

Task	Input	Output	Use Case
Generation	None (sample from model)	New data x_{new}	Image synthesis, data augmentation
Density Estimation	Any data point x	Probability $p(x)$	Anomaly detection, quality control

Task	Input	Output	Use Case
Feature Learning	Training data	Learned representations	Transfer learning, interpretability

✓ KEY TAKEAWAYS - TOPIC 1

1. **Goal:** Learn a probability distribution $p(x)$ from training data
2. **Three capabilities:** Generation, Density Estimation, Feature Learning
3. **Two questions:**
 - How to represent $p(x)$? → **Autoregressive models** (Lecture 3)
 - How to learn $p(x)$? → **Maximum Likelihood** (this lecture)
4. **Connection:** We're building on AR models from Lecture 3, now learning their parameters

□ Connection to Next Topic

Now that we understand **WHAT** we want to achieve, the next question is:

"What exactly is the SETTING for learning? What do we have, and what assumptions do we make?"

This leads us to **TOPIC 2: SETTING** □

Would you like me to proceed to TOPIC 2: SETTING? (This covers the formal problem setup, IID assumption, model family M, etc.)

Just say "next" and I'll continue! □

□ **TOPIC 2: SETTING (The Learning Problem Setup)**

(Formal Problem Definition)

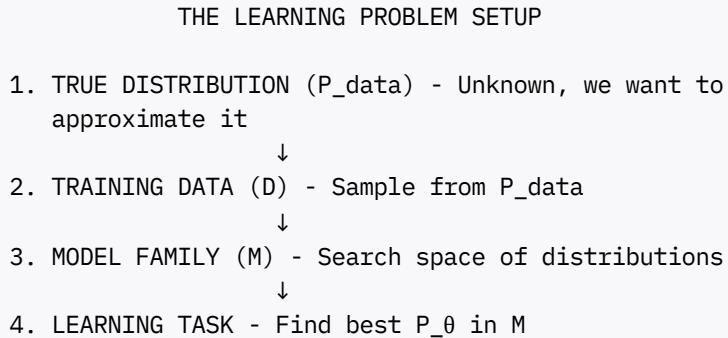
2.0 Big Picture: What is "Setting"?

Before we can solve a problem, we need to **define it precisely**. This section answers:

- What **data** do we have?
- What **assumptions** do we make?
- What are we **searching for**?
- What **constraints** exist?

Think of this as "**Rules of the Game**" - we're establishing the playing field before we start learning.

2.1 The Four Core Components of the Setting



Let's explore each component in detail!

2.2 Component 1: The True Data Distribution P_{data}

Definition:

P_{data} = The unknown, underlying probability distribution that governs our domain

Key Properties:

1. **Unknown:** We **never** have direct access to P_{data}
2. **Real:** It exists in the real world (e.g., distribution of real dog images)
3. **Target:** Our goal is to approximate it

Real-World Examples:

Domain	What is P_{data} ?
Dog Images	The distribution over ALL possible dog images in nature
Text	The distribution of meaningful English sentences
Medical Scans	Distribution of healthy vs. diseased tissue patterns
Speech	Distribution of natural human speech audio patterns

Important Insight:

P_{data} exists "out there" in the real world
↓
We cannot see it directly
↓
We only get SAMPLES from it (training data)

↓

Like trying to understand an ocean by looking at a bucket of water!

2.3 Component 2: Training Dataset D

Definition:

$$D = \{x^{(1)}, x^{(2)}, x^{(3)}, \dots, x^{(m)}\}$$

Where:

- m = number of training examples (dataset size)
- Each $x^{(i)}$ is a **sample** drawn from P_{data}

Characteristics of Each Sample $x^{(i)}$:

Each sample is an **assignment of values** to variables. Let's see examples:

Example 1: Discrete Variables (Classification)

```
x^{(1)} = (X_bank = 1, X_dollar = 0, X_money = 1, Y = spam)
```

↑

This is one email, represented as features

Example 2: Continuous Variables (Images)

```
x^{(1)} = [Pixel1=0.23, Pixel2=0.89, ..., Pixel784=0.45]
```

↑

A 28×28 grayscale image (784 pixels)

Example 3: Mixed (Medical Data)

```
x^{(1)} = (Age=45, Weight=70.5kg, BloodPressure=120/80, Disease=No)
```

↑

Patient record with continuous and categorical variables

Mathematical Notation Breakdown:

$$D = \{x^{(1)}, x^{(2)}, \dots, x^{(m)}\}$$

- **Superscript (i):** Index of the training example (NOT exponentiation!)
- $x^{(1)}$ = first training example
- $x^{(2)}$ = second training example
- m = total number of examples

⚠ Common Confusion Alert:

- $x^{(i)}$ means "i-th training example" (not x to the power i)
- Each $x^{(i)}$ itself can be a vector: $x^{(i)} = (x_1^{(i)}, x_2^{(i)}, \dots, x_n^{(i)})$

2.4 Component 3: The IID Assumption

Full Name: Independent and Identically Distributed

This is **one of the most important assumptions** in machine learning!

What Does IID Mean?

Part 1: Identically Distributed (ID)

Meaning: All samples come from the **same distribution** P_{data}

$$x^{(1)} \sim P_{data}, \quad x^{(2)} \sim P_{data}, \quad \dots, \quad x^{(m)} \sim P_{data}$$

Real-World Analogy:

- Like drawing marbles from the same jar
- Each draw follows the same probability rules

Example - Dog Images:

All images are from "real dogs in nature"
NOT: Some from dogs, some from cats, some from cartoons

Part 2: Independent (I)

Meaning: Knowing one sample gives **no information** about another

$$P(x^{(1)}, x^{(2)}, \dots, x^{(m)}) = \prod_{i=1}^m P(x^{(i)})$$

Real-World Analogy:

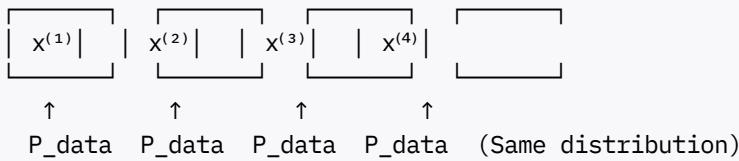
- Drawing marbles **with replacement**
- Flipping a coin - each flip doesn't affect the next

Example - Medical Records:

- ✓ GOOD (Independent): Records from different, unrelated patients
- ✗ BAD (Not Independent): Multiple records from the same patient
(records are correlated!)

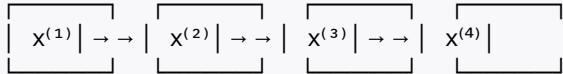
Visual Representation: IID vs Non-IID

IID DATA (GOOD):



No arrows between samples (Independent)

NON-IID DATA (PROBLEMATIC):



Arrows show dependence (e.g., time series, correlated data)

Why is IID Important?

Mathematical Reason:

If samples are IID, the joint probability **factorizes**:

$$P(D) = P(x^{(1)}, x^{(2)}, \dots, x^{(m)}) = \prod_{i=1}^m P(x^{(i)})$$

This makes learning **mathematically tractable**!

Practical Reason:

With IID	Without IID
Simple algorithms work	Need complex sequential models
Can shuffle data	Order matters
Easy to parallelize	Must process sequentially
Clear statistical guarantees	Guarantees break down

Real-World Example: When IID is Violated

Scenario: Predicting stock prices

- ✗ Stock price today depends on yesterday's price
- ✗ Samples are NOT independent
- ✗ Must use time-series models (RNNs, LSTMs)
- ✗ Standard IID learning fails!

Scenario: Random patient records

- ✓ Patient A's health independent of Patient B
- ✓ Can assume IID
- ✓ Standard learning works!

2.5 Component 4: Model Family M

Definition:

$$M = \{\text{Set of all probability distributions we're considering}\}$$

The model family is our **search space** - we're looking for the best distribution **within M** .

Parameterization:

Each model in M is specified by **parameters θ** :

$$M = \{P_\theta \mid \theta \in \Theta\}$$

Where:

- P_θ = A specific probability distribution with parameters θ
- Θ = The parameter space (all possible values of θ)

Real-World Examples:

Example 1: Bayesian Network with Fixed Structure

Model Family:

M = All Bayes nets with this graph structure:

$$A \rightarrow C \leftarrow B$$

For all possible CPD (Conditional Probability Distribution) tables

Parameters θ :

- θ = All the probability values in the CPD tables
- Example: $P(C = 1 | A = 1, B = 0) = 0.7$ is part of θ

Example 2: Fully Visible Sigmoid Belief Network (FVSBN)

Recall from Lecture 3:

Autoregressive model where each conditional is logistic regression:

$$P(x_i = 1 | x_{<i}) = \sigma(w_i^T x_{<i} + b_i)$$

Parameters θ :

$$\theta = (w_1, b_1, w_2, b_2, \dots, w_n, b_n)$$

- All weights and biases concatenated
- If each x_i is binary and we have n variables: $\theta \in \mathbb{R}^{O(n^2)}$

Example 3: PixelCNN (From Lecture 3)

Model Family:

M = All autoregressive models of form:

$$p(x) = \prod p_{\text{neural}}(x_i | x_{\{<i\}}; \theta)$$

where p_{neural} is a convolutional neural network

Parameters θ :

- All CNN weights and biases
- Could be millions or billions of parameters!
- Example: PixelCNN with 10 layers, 256 filters: ~10M parameters

2.6 The Learning Task (Putting It All Together)

Formal Problem Statement:

GIVEN:

- Unknown distribution P_{data}
- Training data $D = \{x^{(1)}, \dots, x^{(m)}\}$ sampled IID from P_{data}
- Model family $M = \{P_\theta | \theta \in \Theta\}$

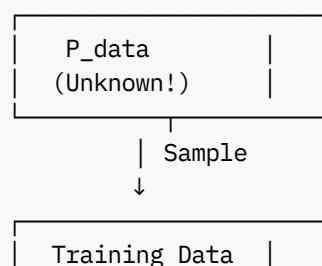
FIND:

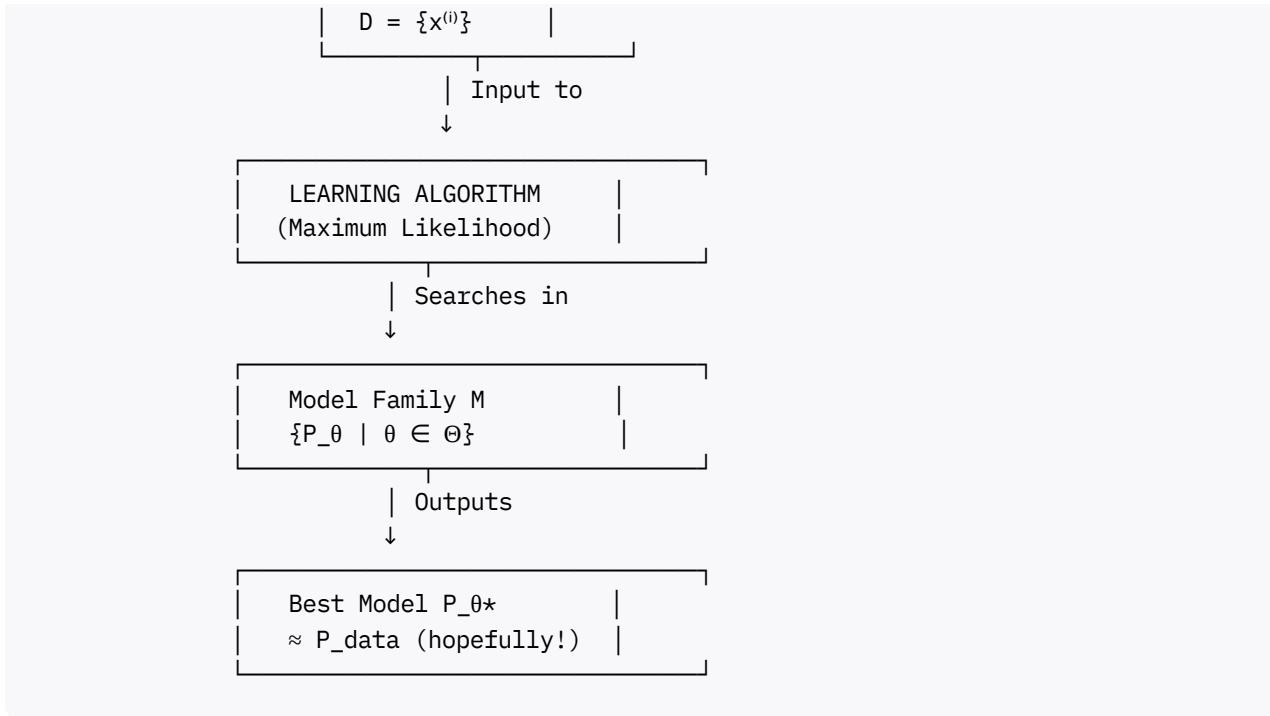
- Parameters θ^* that give the "best" model $P_{\theta^*} \in M$

GOAL:

- Make P_{θ^*} as close as possible to P_{data}

Visual Flowchart:





2.7 Detailed Example: Image Classification Setup

Let's make this concrete with a **complete example**.

Domain: MNIST Handwritten Digits

Component 1: P_{data}

The true distribution of handwritten digits

- How humans naturally write 0-9
- Variations in style, size, rotation
- We don't know this distribution explicitly!

Component 2: Dataset D

```

D = {60,000 training images}

Each image: 28×28 grayscale pixels

x(1) = [0.0, 0.0, 0.1, ..., 0.9, 0.8] ← First image (digit "3")
          └────────── 784 values ──────────

x(2) = [0.0, 0.1, 0.0, ..., 0.7, 0.6] ← Second image (digit "7")

...
x(60000) = [0.2, 0.0, 0.1, ..., 0.5, 0.4] ← Last image

```

Component 3: IID Assumption

- ✓ Each image written by different people
- ✓ No temporal correlation
- ✓ Reasonable to assume IID

Component 4: Model Family M

Choice: Autoregressive model with neural networks

$$p(x) = \prod_{i=1}^{784} p_{neural}(x_i | x_{<i}; \theta)$$

Parameters θ :

Neural network with:

- 3 hidden layers
- 128 neurons per layer
- Total parameters: $\theta \in \mathbb{R}^{150000}$ (150,000 parameters!)

The Task:

Find θ^* such that $P_{\theta^*}(x) \approx P_{data}(x)$

After learning:

- $P_{\theta^*}(\text{image of "3"})$ should be HIGH
- $P_{\theta^*}(\text{random noise})$ should be LOW
- Can sample $x_{\text{new}} \sim P_{\theta^*}$ to generate new digit images!

2.8 Why These Components Matter

Table: Impact of Each Component

Component	If Changed...	Impact on Learning
P_{data}	Different domain	Entirely different problem
Dataset D	More data	Better approximation, less overfitting
	Less data	Risk of overfitting
IID Assumption	Violated	Standard algorithms may fail
Model Family M	Too simple	High bias (underfitting)
	Too complex	High variance (overfitting)
	Wrong choice	Cannot represent P_{data} well

2.9 Comparison Table: Different Model Families

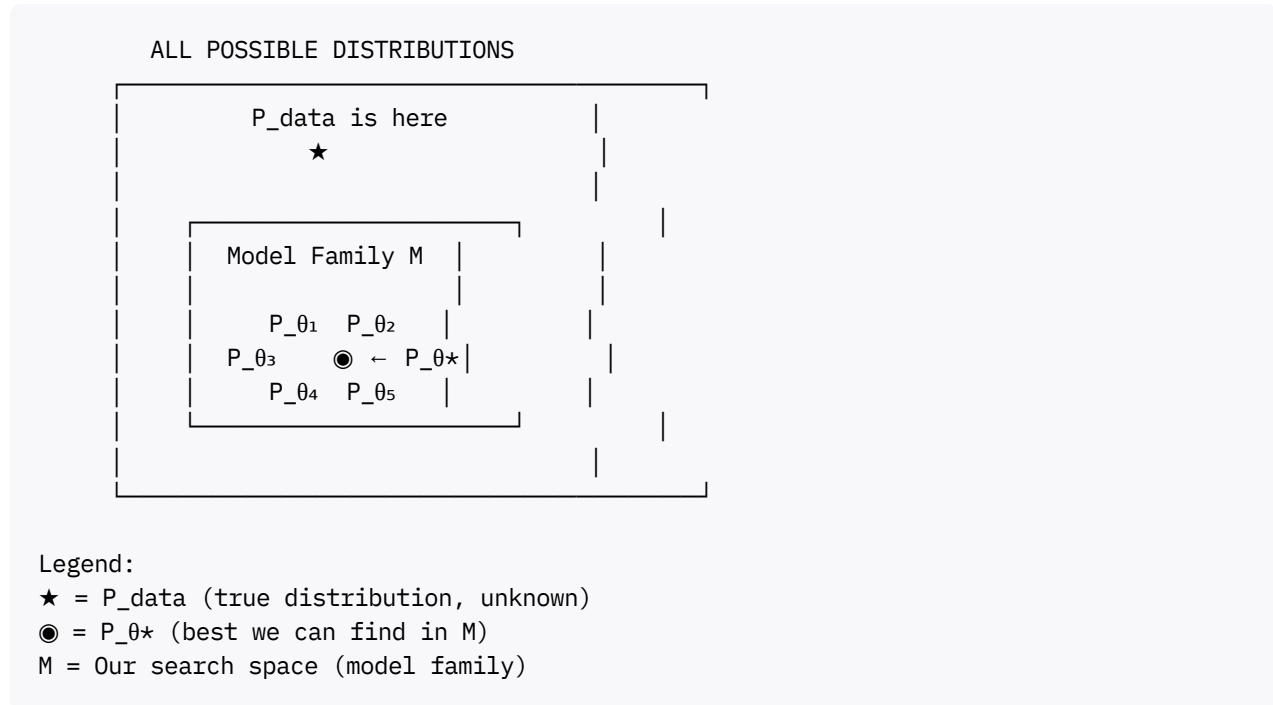
Let's compare different choices for M :

Model Family	Complexity	Parameters	Best For	Limitations
Fixed CPD Tables	Low	$O(2^n)$	Small discrete domains	Doesn't scale to images
FVSBN	Medium	$O(n^2)$	Binary variables	Limited expressiveness
NADE	Medium-High	$O(n \cdot h)$	Tabular data	Better than FVSBN
PixelCNN	High	Millions	Images	Slow sampling
Transformer	Very High	Billions	Text, sequences	Requires huge data

Connection to Lecture 3: These are all autoregressive models we studied!

2.10 Critical Insight: The Search Space

Visual Representation:



Key Observations:

1. P_{data} might not be in M !
 - Our model family might not include the true distribution
 - This introduces **approximation error**
2. Learning = Searching in M
 - We move through M trying different θ
 - Looking for the P_θ closest to P_{data}

3. Trade-off:

- **Larger M :** Better chance to include P_{data} , but harder to search
- **Smaller M :** Easier to search, but might miss P_{data}

✓ KEY TAKEAWAYS - TOPIC 2

The Four Pillars of the Learning Setup:

1. P_{data} : Unknown true distribution we want to approximate
2. **Dataset D :** m samples drawn IID from P_{data}
3. **IID Assumption:** Samples are independent and identically distributed
4. **Model Family M :** Search space of distributions parameterized by θ

The Learning Task:

Find θ^* such that P_{θ^*} in M best approximates P_{data}

Important Constraints:

- We **never see** P_{data} directly
- We only have **finite samples** D
- We search in a **restricted space** M
- Our samples are assumed **IID**

Connection to Next Topic:

We've defined the problem. Now the **crucial question**:

"**What does 'best approximation' mean? How do we measure if P_θ is good?**"

This leads us to **TOPIC 3: GOAL OF LEARNING & WHAT IS "BEST"?** □

Ready to continue to TOPIC 3? Just say "next"! □

□ **TOPIC 3: GOAL OF LEARNING & WHAT IS "BEST"?**

(Defining Success in Learning)

3.0 Big Picture: The Central Question

From Topic 2, we have:

- Training data D from P_{data}
- Model family $M = \{P_\theta\}$

The Burning Question:

We want to find the "BEST" model P_{θ^*} in M

↓

But what is "BEST"?

↓

How do we measure quality?

This topic explores **different ways to define "best"** depending on what we want to achieve.

3.1 Why "Best" Depends on the Goal

Key Insight:

There is **no universal definition** of "best model" - it depends on:

- What **task** you want to perform
- What you care about **measuring**
- Your **application domain**

Analogy:

"What is the best car?"

- For racing? → Sports car (fast, low weight)
- For family? → Minivan (space, safety)
- For off-road? → Jeep (durability, clearance)
- For efficiency? → Electric car (low cost per mile)

Different goals → Different "best" answers!

Similarly for models:

"What is the best probability distribution P_{θ} ?"

- Depends on your goal!

3.2 Three Major Goals in Learning

Overview Table:

Goal	Focus	What "Best" Means	Example Applications
1. Density Estimation	Full distribution $p(x)$	Matches P_{data} globally	Generative modeling, anomaly detection
2. Specific Prediction	Particular queries	High accuracy on target task	Spam detection, classification
3. Structure Discovery	Model structure itself	Interpretable, causal	Scientific discovery, understanding

Let's explore each in detail!

3.3 GOAL 1: Density Estimation (Our Main Focus)

Definition:

We want to learn the FULL probability distribution $p(x)$ accurately.

Why Learn the Full Distribution?

Flexibility: Once we have $p(x)$, we can answer ANY probabilistic query later!

After learning $p(x)$, we can compute:

- $p(x_1 = 1)$ (marginal probability)
- $p(x_3 \mid x_1 = 0, x_2 = 1)$ (conditional probability)
- $\text{argmax}_x p(x)$ (most likely state)
- $E[x_5]$ (expected value)
- Sample new data $x \sim p(x)$ (generation)

What "Best" Means Here:

P_θ is "best" if it is CLOSE to P_{data}

We want $P_\theta(x) \approx P_{data}(x)$ for all possible x .

Real-World Examples:

Example 1: Image Generation

Goal: Learn $p(x)$ over natural images

After learning:

- ✓ Generate new realistic images: $x_{\text{new}} \sim p(x)$
- ✓ Compute likelihood: "Does this image look natural?"
- ✓ In-paint: Fill missing parts by maximizing $p(x)$
- ✓ Anomaly detection: Is this image corrupted?

Applications:

- Photo editing software (Photoshop generative fill)
- Medical imaging (detect abnormal scans)
- Art generation (DALL-E, Stable Diffusion foundations)

Example 2: Anomaly Detection in Manufacturing

Goal: Learn $p(x)$ over images of non-defective products

Setup:

Training data D: 10,000 images of good products
 Learn: $p(x)$ over "normal" appearance

At Test Time:

New product arrives with image x_{test}
 Compute: $p(x_{\text{test}})$
 If $p(x_{\text{test}}) > \text{threshold}$: → Normal product ✓
 If $p(x_{\text{test}}) < \text{threshold}$: → Defective! Flag for inspection ✗

Mathematical Example:

x_1 (normal product):	$p(x_1) = 0.89$	→ PASS ✓
x_2 (scratched):	$p(x_2) = 0.34$	→ PASS (minor issue) ✓
x_3 (broken):	$p(x_3) = 0.02$	→ FAIL ✗
x_4 (wrong part):	$p(x_4) = 0.001$	→ FAIL ✗

Example 3: Medical Diagnosis Support

Goal: Learn $p(x)$ over healthy brain MRI scans

Training: 50,000 scans from healthy patients

Application:

New patient scan arrives: x_{new}
 Compute: $p(x_{\text{new}})$
 High $p(x_{\text{new}})$ → Looks like healthy brain ✓
 Low $p(x_{\text{new}})$ → Unusual pattern, investigate further ✗

Could indicate:

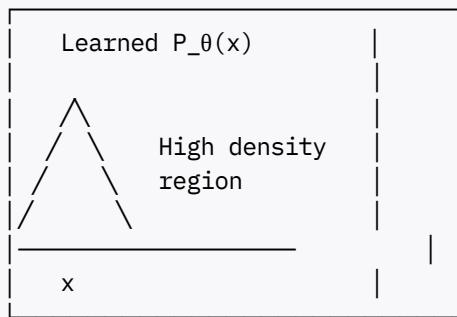
- Tumor
- Stroke damage
- Atrophy
- Acquisition error (technical issue)

Visual Representation: Density Estimation Goal

P_{data} (Unknown True Distribution)



We want $P_\theta \approx P_{data}$



For ALL x : $P_\theta(x)$ should match $P_{data}(x)$

Mathematical Formulation:

Ideal Scenario (Impossible):

$$P_\theta(x) = P_{data}(x) \quad \forall x$$

Realistic Goal:

Minimize "distance" between P_θ and P_{data}

Question: How to measure "distance" between distributions?

Answer (Next Topic): KL-Divergence! ☺

3.4 GOAL 2: Specific Prediction Tasks

Definition:

We use the distribution to make specific predictions, not interested in full $p(x)$.

What "Best" Means Here:

P_θ is "best" if it gives HIGH ACCURACY on the target prediction task

We **only care** about specific queries, not the entire distribution.

Comparison with Density Estimation:

Density Estimation:	Learn ALL of $p(x)$
	↓
	Can answer ANY query later
Specific Prediction:	Learn $p(x)$ to answer SPECIFIC query
	↓
	Only care about task performance

Example 2.1: Binary Classification (Spam Detection)

Task:

Given email x , predict: Spam or Not Spam?

Setup:

Variables: $x = (x_1, x_2, \dots, x_n, y)$
• x_1, \dots, x_n : email features (words, sender, etc.)
• y : label (spam=1, not spam=0)

Goal: Predict y given x_1, \dots, x_n

What We Need:

We only need: $p(y | x_1, \dots, x_n)$
We DON'T need: Full joint $p(x_1, \dots, x_n, y)$

Prediction Rule:

$$\hat{y} = \begin{cases} 1 (\text{spam}) & \text{if } p(y = 1 | x_1, \dots, x_n) > 0.5 \\ 0 (\text{not spam}) & \text{otherwise} \end{cases}$$

What "Best" Means:

P_θ is "best" if it maximizes:

- Classification accuracy
- F1-score
- Area under ROC curve

NOT necessarily if it models $p(x_1, \dots, x_n)$ accurately!

Mathematical Example:

Email 1: "Buy cheap watches now!"

Features: $x_1=1$ (contains "buy"), $x_2=1$ (contains "\$\$\$"), ...

Model computes: $p(y=1 | x_1=1, x_2=1, \dots) = 0.92$

Prediction: SPAM ✓

Email 2: "Meeting tomorrow at 3pm"

Features: $x_1=0$, $x_2=0$, ...

Model computes: $p(y=1 | x_1=0, x_2=0, \dots) = 0.08$

Prediction: NOT SPAM ✓

Example 2.2: Structured Prediction

Task 1: Video Frame Prediction

Input: Video frames x_1, x_2, \dots, x_t (first t frames)

Output: Predict $x_{\{t+1\}}$ (next frame)

Need: $p(x_{\{t+1\}} | x_1, \dots, x_t)$

Application:

- Self-driving cars (predict pedestrian movement)
- Video compression
- Action anticipation

Task 2: Image Captioning

Input: Image x

Output: Text caption $y = (y_1, y_2, \dots, y_T)$

"A brown dog playing in grass"

Need: $p(y | x) = p(y_1 | x) \cdot p(y_2 | x, y_1) \cdot \dots \cdot p(y_T | x, y_1, \dots, y_{\{T-1\}})$

Application:

- Accessibility (describe images for visually impaired)
- Image search
- Social media auto-tagging

Visual: Specific Prediction vs Density Estimation

DENSITY ESTIMATION:

Learn ENTIRE $p(x_1, x_2, \dots, x_n)$

Can answer:

- $p(x_1)$
- $p(x_2 | x_1)$
- $p(x_3 | x_1, x_5)$
- ANY query!

SPECIFIC PREDICTION:

Learn ONLY $p(y | x)$

Focus on:

- ONE specific query
- Task accuracy
- Don't need full $p(x)$

3.5 GOAL 3: Structure or Knowledge Discovery

Definition:

We are interested in the MODEL STRUCTURE itself, not just predictions.

What "Best" Means Here:

P_θ is "best" if the MODEL STRUCTURE reveals TRUE relationships

We want to discover:

- **Causal relationships:** What causes what?
- **Dependencies:** Which variables interact?
- **Interpretable patterns:** How does the system work?

Key Difference:

Previous Goals: Model is a BLACK BOX
↓
We only care about OUTPUT

Structure Discovery: Model is INTERPRETABLE
↓
We care about INTERNAL STRUCTURE

Example 3.1: Gene Interaction Networks

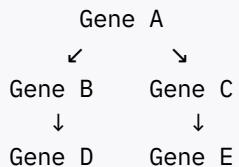
Scientific Question:

"Which genes interact with each other? What are the regulatory pathways?"

Setup:

Variables: x_1, x_2, \dots, x_n (expression levels of n genes)

Learn: Bayesian Network structure



This structure tells us:

- Gene A regulates B and C
- Gene B affects D
- Gene C affects E

What "Best" Means:

The learned graph structure should reflect:

- TRUE biological regulatory relationships
- Causal mechanisms
- Interpretable by biologists

NOT just prediction accuracy!

Real Application:

Discovered structure → Suggests experiments
"Does knocking out Gene A affect Gene B?"

If model structure is correct:

- Experiment confirms it
- Leads to drug targets!

Example 3.2: Disease Causation

Medical Research Question:

"What causes cancer? What are the risk factors?"

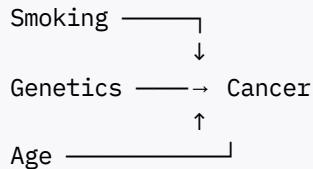
Variables:

```
x1: Smoking (yes/no)  
x2: Age  
x3: Genetic markers  
x4: Diet  
x5: Exercise  
x6: Pollution exposure  
...  
y: Cancer (yes/no)
```

Goal:

Learn a model where the **structure reveals causal relationships**

Learned Causal Graph:



Interpretation:

- Smoking CAUSES cancer (can intervene!)
- Genetics CAUSE cancer (cannot intervene easily)
- Age is a risk factor

Why Structure Matters:

Prediction-Only Model	Structure Discovery Model
"Predict who gets cancer"	"Understand WHY cancer happens"
Black box	Interpretable
Cannot guide interventions	Suggests preventive measures
90% accuracy	85% accuracy BUT reveals: "Stop smoking reduces risk by 40%"

Medical Impact:

Structure reveals: Smoking → Cancer (causal)

Policy Implication: Anti-smoking campaigns

Research Direction: Study molecular pathway

Treatment: Drugs targeting that pathway

Example 3.3: Climate Science

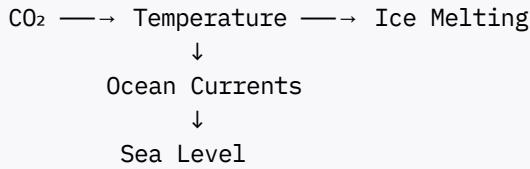
Question:

"How do different climate factors interact?"

Variables:

- CO₂ levels
- Temperature
- Ice coverage
- Sea level
- Ocean currents

Learned Structure Reveals:



This tells us:

- Feedback loops
- Intervention points
- Prediction of cascade effects

3.6 Detailed Comparison: The Three Goals

Comparison Table:

Aspect	Density Estimation	Specific Prediction	Structure Discovery
What to learn	Full $p(x)$	Specific $p(y x)$	Model structure + $p(x)$
Measure of "best"	Close to P_{data}	Task accuracy	Interpretability + causality
Flexibility	Can answer any query later	Limited to trained task	Reveals mechanisms
Typical metric	Log-likelihood, KL-div	Accuracy, F1, MSE	Domain expert validation
Example	Image generation	Spam classification	Gene networks
Model complexity	High (full distribution)	Medium (conditional only)	Medium (+ interpretability)
Data needs	Large	Medium	Large + expert knowledge

Visual: Different Optimization Objectives

DENSITY ESTIMATION:

Minimize: $D(P_{\text{data}} || P_{\theta})$ [KL-divergence - coming next!]
↓
Learn full $p(x)$

SPECIFIC PREDICTION:

Minimize: Classification error on y
↓
Learn only $p(y|x)$

STRUCTURE DISCOVERY:

Minimize: Prediction error + Complexity penalty
↓
Learn sparse, interpretable structure

3.7 Focus of This Lecture: Density Estimation

Why We Focus on Density Estimation:

1. Most General:

Learn $p(x)$ → Can answer any query
→ Includes prediction as special case
→ Most flexible

2. Natural for Generative Models:

Recall Lecture 1 Goals:

- Generate new samples (need $p(x)$)
- Density estimation (need $p(x)$)
- Feature learning (need $p(x)$)

All require full distribution!

3. Connects to Autoregressive Models (Lecture 3):

AR models give us $p(x) = \prod p(x_i | x_{\{<i\}})$
↓
Perfect for density estimation!

The Path Forward:

```
We've decided: Our goal is DENSITY ESTIMATION  
↓  
We want  $P_{\theta} \approx P_{\text{data}}$   
↓  
Question: How to measure "closeness"?  
↓  
Answer: KL-Divergence! (Next topic)
```

3.8 Mathematical Formalization: Density Estimation Goal

The Optimization Problem:

Informal:

```
Find  $\theta^*$  such that  $P_{\theta^*}$  is as close as possible to  $P_{\text{data}}$ 
```

Formal (What we'll derive):

$$\theta^* = \arg \min_{\theta} \text{distance}(P_{\text{data}}, P_{\theta})$$

Where "distance" could be:

- KL-divergence (most common - next topic!)
- Wasserstein distance
- Total variation distance
- etc.

Ideal vs Realistic:

```
IDEAL (Impossible):  
 $P_{\theta^*}(x) = P_{\text{data}}(x)$  for all  $x$   
↓  
Perfect match!
```

```
REALISTIC (What we can do):  
 $P_{\theta^*}(x) \approx P_{\text{data}}(x)$  for "most"  $x$   
↓  
Minimize average distance over data distribution
```

✓ KEY TAKEAWAYS - TOPIC 3

Main Insights:

1. "Best" depends on goal - no universal definition
2. Three major goals:
 - Density Estimation: Learn full $p(x)$ → Most flexible
 - Specific Prediction: Learn $p(y|x)$ → Task-focused
 - Structure Discovery: Learn interpretable model → Scientific insight
3. This lecture focuses on Density Estimation because:
 - Most general (includes others as special cases)
 - Natural for generative models
 - Matches autoregressive models from Lecture 3
4. For Density Estimation, "best" means:
 P_θ is close to P_{data}
5. Next question: How to measure "closeness" between distributions?

Connection to Previous Topics:

Topic 1: What can we do with $p(x)$?
→ Three capabilities (generation, density, features)

Topic 2: What is the learning setup?
→ P_{data} , D, IID, Model family M

Topic 3: What is our goal?
→ Density estimation: $P_\theta \approx P_{data}$

Next Topic: How to measure closeness?
→ KL-Divergence! □

□ Connection to Next Topic

We've established our goal:

Find P_θ that is "close" to P_{data}

Critical question remains:

"What mathematical measure should we use to quantify 'closeness' between two probability distributions?"

This leads us to **TOPIC 4: KL-DIVERGENCE** □

Where we'll learn:

- How to measure distance between distributions
- Properties of KL-divergence
- Why it's the right choice for our problem
- Intuitive meaning (compression, information theory)

Ready to continue to **TOPIC 4: KL-DIVERGENCE?** Just say "next"! ☺

□ **TOPIC 4: KL-DIVERGENCE**

(Kullback-Leibler Divergence: Measuring Distance Between Distributions)

4.0 Big Picture: Why Do We Need KL-Divergence?

Recap: Where We Are

From Topic 3, we decided:

```
GOAL: Density Estimation
      ↓
Find P_θ that is CLOSE to P_data
      ↓
But how do we measure "CLOSE"?
```

The Challenge:

We cannot use simple distance measures like:

- ✗ Euclidean distance (doesn't work for distributions)
- ✗ Subtraction $P_\theta(x) - P_{data}(x)$ (need to aggregate over all x)
- ✗ Maximum absolute difference (too sensitive to outliers)

We need a measure specifically designed for probability distributions!

The Solution: KL-Divergence

KL-Divergence (Kullback-Leibler divergence) is:

- A **measure of difference** between two probability distributions
- Based on **information theory**
- The **standard tool** in machine learning for comparing distributions
- Has beautiful **mathematical properties**

4.1 Mathematical Definition

Formal Definition:

For two probability distributions p and q over the same space, the **KL-divergence from q to p** is:

$$D(p \parallel q) = \sum_x p(x) \log \frac{p(x)}{q(x)}$$

Alternative form (equivalent):

$$D(p \parallel q) = \sum_x p(x) \log p(x) - \sum_x p(x) \log q(x)$$

Expectation notation:

$$D(p \parallel q) = \mathbb{E}_{x \sim p} \left[\log \frac{p(x)}{q(x)} \right]$$

Breaking Down the Formula:

Let's understand each component:

$$D(p \parallel q) = \sum p(x) \log[p(x)/q(x)]$$

↑ ↑ ↑ ↑ ↑
| | | | |
p(x): weight by true distribution
q(x): "approximate" distribution
log ratio (how different?)
Sum over all possible x

Notation Clarification:

⚠ **Important:** The notation $D(p \parallel q)$ means:

- p is the "reference" or "true" distribution
- q is the "approximating" distribution
- Read as: "KL-divergence from q to p " or "divergence of q from p "

For our learning problem:

$$D(P_{data} \parallel P_{\theta})$$

- P_{data} = true distribution (reference)
- P_{θ} = our model (approximation)

4.2 Continuous Version (For Continuous Variables)

Definition for Continuous Distributions:

If x is continuous with probability density functions $p(x)$ and $q(x)$:

$$D(p \parallel q) = \int p(x) \log \frac{p(x)}{q(x)} dx$$

Replace:

- Sum $\sum \rightarrow$ Integral \int
- PMF $p(x) \rightarrow$ PDF $p(x)$

Everything else stays the same!

Mixed Variables:

For models with both discrete and continuous variables:

$$D(p \parallel q) = \sum_{\text{discrete}} \int_{\text{continuous}} p(x) \log \frac{p(x)}{q(x)}$$

4.3 Intuitive Meaning: Information Theory Perspective

Interpretation 1: Extra Bits for Compression

KL-divergence measures the expected number of EXTRA BITS needed to encode samples from $p(x)$ using a code optimized for $q(x)$ instead of $p(x)$.

Let's unpack this step-by-step:

Background: Data Compression and Probability

Key Principle from Information Theory:

The optimal number of bits to encode a symbol x is:

$$\text{bits}(x) = -\log_2 p(x)$$

Why?

- Frequent events (high $p(x)$) \rightarrow Use few bits
- Rare events (low $p(x)$) \rightarrow Use many bits

Compression Example: Coin Flips

Setup: Unbiased Coin

True distribution p : $P[\text{Heads}] = 0.5$, $P[\text{Tails}] = 0.5$

Optimal encoding:

Heads → 0 (1 bit)
Tails → 1 (1 bit)

Average bits per flip = $0.5 \times 1 + 0.5 \times 1 = 1$ bit ✓

Setup: Biased Coin

True distribution p : $P[\text{Heads}] = 0.9$, $P[\text{Tails}] = 0.1$

Optimal encoding (Huffman-like):

Heads → 0 (1 bit) - more common, shorter code
Tails → 10 (2 bits) - less common, longer code

Average bits per flip = $0.9 \times 1 + 0.1 \times 2 = 1.1$ bits

Key insight: Biased coin needs **more than 1 bit on average** because we need to distinguish rare events!

The Compression Mismatch Problem

Scenario: Your data comes from p , but you design your code for q

Reality: Data sampled from $p(x)$
Your assumption: Data follows $q(x)$
↓
Mismatch! You use WRONG code!

Consequence:

- Code optimized for q uses: $-\log q(x)$ bits for symbol x
- But data comes from p
- **Extra bits needed:** $-\log q(x) - (-\log p(x)) = \log \frac{p(x)}{q(x)}$

Average over all x (weighted by p):

$$\mathbb{E}_{x \sim p} \left[\log \frac{p(x)}{q(x)} \right] = D(p \parallel q)$$

This is exactly KL-divergence! □

4.4 Detailed Mathematical Example: Compression

Let's make this concrete with numbers.

Example Setup: Coin Flipping

True distribution p :

$$\begin{aligned} P[H] &= 0.8 \\ P[T] &= 0.2 \end{aligned}$$

Approximating distribution q (you incorrectly think it's unbiased):

$$\begin{aligned} Q[H] &= 0.5 \\ Q[T] &= 0.5 \end{aligned}$$

Step 1: Optimal Bits for Each Distribution

Using TRUE distribution p :

Optimal bits for H: $-\log_2(0.8) = 0.322$ bits

Optimal bits for T: $-\log_2(0.2) = 2.322$ bits

Average (Entropy of p):

$$H(p) = 0.8 \times 0.322 + 0.2 \times 2.322 = 0.722 \text{ bits}$$

Using WRONG distribution q :

Bits for H (using q): $-\log_2(0.5) = 1$ bit

Bits for T (using q): $-\log_2(0.5) = 1$ bit

Average when data comes from p but using code for q :

$$\text{Cross-entropy} = 0.8 \times 1 + 0.2 \times 1 = 1.0 \text{ bits}$$

Step 2: Calculate KL-Divergence

$$D(p \parallel q) = \sum_x p(x) \log \frac{p(x)}{q(x)}$$

For Heads:

$$p(H) \log \frac{p(H)}{q(H)} = 0.8 \times \log_2 \frac{0.8}{0.5} = 0.8 \times \log_2(1.6) = 0.8 \times 0.678 = 0.542$$

For Tails:

$$p(T) \log \frac{p(T)}{q(T)} = 0.2 \times \log_2 \frac{0.2}{0.5} = 0.2 \times \log_2(0.4) = 0.2 \times (-1.322) = -0.264$$

Total:

$$D(p \parallel q) = 0.542 + (-0.264) = 0.278 \text{ bits}$$

Step 3: Interpretation

Optimal encoding (using p): 0.722 bits per flip

Your encoding (using q): 1.000 bits per flip

Extra bits wasted: 0.278 bits per flip

↑

This is $D(p \parallel q)$!

Verification:

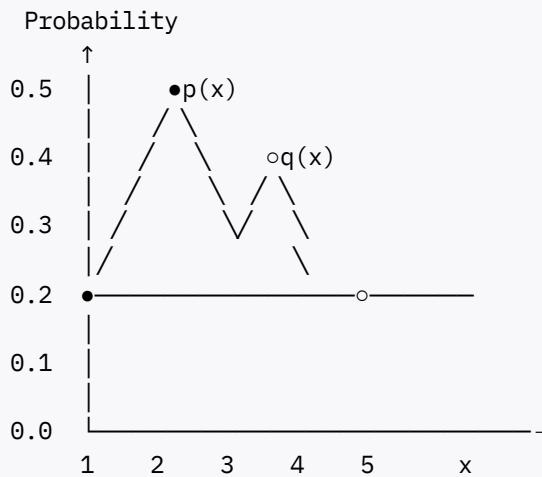
$$\text{Cross-entropy} - \text{Entropy} = 1.0 - 0.722 = 0.278 = D(p \parallel q) \checkmark$$

Meaning: By assuming the wrong distribution q instead of the true p , you waste **0.278 extra bits per coin flip** on average!

4.5 Visual Representation of KL-Divergence

Graph: Two Distributions

$p(x)$ and $q(x)$ over discrete space $x \in \{1, 2, 3, 4, 5\}$



Where p and q differ most → Contributes most to $D(p \parallel q)$

Key Observations:

1. At $x = 2$: $p(x)$ high, $q(x)$ low → Large contribution to $D(p \parallel q)$
2. At $x = 4$: $q(x)$ high, $p(x)$ low → Small contribution (weighted by $p(x)$!)

3. Where they match: Zero contribution

4.6 Key Properties of KL-Divergence

Property 1: Non-Negativity

$$D(p \parallel q) \geq 0 \quad \forall p, q$$

Meaning: KL-divergence is **always non-negative** (or zero).

Property 2: Zero if and Only if Distributions Match

$$D(p \parallel q) = 0 \iff p(x) = q(x) \text{ for all } x$$

Meaning:

- $D(p \parallel q) = 0 \implies$ Distributions are **identical**
- $D(p \parallel q) > 0 \implies$ Distributions **differ**

This makes it perfect for measuring closeness! ☺

Property 3: Asymmetry

$$D(p \parallel q) \neq D(q \parallel p) \text{ (in general)}$$

Meaning: KL-divergence is **NOT a true distance metric** (not symmetric).

Example:

p: $P[H]=0.9, P[T]=0.1$
q: $P[H]=0.5, P[T]=0.5$

$$D(p \parallel q) \neq D(q \parallel p)$$

This is okay! Order matters for compression.

Asymmetry Visualization:

$D(p \parallel q)$: "Cost of using q when data comes from p"
↓
Different from
↓
 $D(q \parallel p)$: "Cost of using p when data comes from q"

4.7 Mathematical Proof: Non-Negativity (Property 1)

Let's prove $D(p \parallel q) \geq 0$.

Proof Using Jensen's Inequality:

Step 1: Starting point

$$D(p \parallel q) = \sum_x p(x) \log \frac{p(x)}{q(x)}$$

Step 2: Rewrite

$$D(p \parallel q) = - \sum_x p(x) \log \frac{q(x)}{p(x)}$$

Step 3: Apply Jensen's Inequality

Since \log is **concave** and $\sum_x p(x) = 1$:

$$-\sum_x p(x) \log \frac{q(x)}{p(x)} \geq -\log \left(\sum_x p(x) \frac{q(x)}{p(x)} \right)$$

Step 4: Simplify

$$-\log \left(\sum_x p(x) \frac{q(x)}{p(x)} \right) = -\log \left(\sum_x q(x) \right) = -\log(1) = 0$$

Step 5: Conclusion

$$D(p \parallel q) \geq 0 \quad \checkmark$$

Equality holds when:

$$\frac{q(x)}{p(x)} = \text{constant} \quad \forall x$$

Since both are probability distributions ($\sum q(x) = \sum p(x) = 1$), this means:

$$q(x) = p(x) \quad \forall x$$

Therefore: $D(p \parallel q) = 0 \iff p = q \checkmark$

Visual Proof Intuition:

log function (concave)





Jensen's Inequality:

$$E[\log(X)] \leq \log(E[X])$$



$$-E[\log(X)] \geq -\log(E[X])$$



$$D(p || q) \geq 0$$

4.8 Detailed Calculation Example

Let's compute KL-divergence **step-by-step** for a concrete example.

Example: Three-Outcome Distribution

Setup:

Variable $x \in \{A, B, C\}$

True distribution p :

$$p(A) = 0.5$$

$$p(B) = 0.3$$

$$p(C) = 0.2$$

Approximate distribution q :

$$q(A) = 0.4$$

$$q(B) = 0.4$$

$$q(C) = 0.2$$

Goal: Compute $D(p || q)$

Step-by-Step Calculation:

$$D(p || q) = \sum_{x \in \{A, B, C\}} p(x) \log \frac{p(x)}{q(x)}$$

For $x = A$:

$$p(A) \log \frac{p(A)}{q(A)} = 0.5 \times \log \frac{0.5}{0.4}$$

Calculate ratio: $\frac{0.5}{0.4} = 1.25$

Calculate log (using natural log): $\log(1.25) = 0.223$

Result: $0.5 \times 0.223 = 0.112$

For $x = B$:

$$p(B) \log \frac{p(B)}{q(B)} = 0.3 \times \log \frac{0.3}{0.4}$$

Calculate ratio: $\frac{0.3}{0.4} = 0.75$

Calculate log: $\log(0.75) = -0.288$

Result: $0.3 \times (-0.288) = -0.086$

For $x = C$:

$$p(C) \log \frac{p(C)}{q(C)} = 0.2 \times \log \frac{0.2}{0.2}$$

Calculate ratio: $\frac{0.2}{0.2} = 1.0$

Calculate log: $\log(1.0) = 0$

Result: $0.2 \times 0 = 0$

Total:

$$D(p \parallel q) = 0.112 + (-0.086) + 0 = 0.026 \text{ nats}$$

(nats = natural log units; divide by $\log 2$ to get bits)

Calculation Summary Table:

x	$p(x)$	$q(x)$	$\frac{p(x)}{q(x)}$	$\log \frac{p(x)}{q(x)}$	$p(x) \log \frac{p(x)}{q(x)}$
A	0.5	0.4	1.25	0.223	0.112
B	0.3	0.4	0.75	-0.288	-0.086
C	0.2	0.2	1.00	0.000	0.000
Total	1.0	1.0	—	—	0.026

Interpretation:

- $D(p \parallel q) = 0.026$ nats
- Distributions are **close but not identical**
- Main difference: p puts more mass on A (0.5) than q (0.4)
- q puts too much mass on B relative to p

4.9 KL-Divergence for Our Learning Problem

Connecting to Density Estimation:

Recall our goal from Topic 3:

Find P_θ that is CLOSE to P_{data}

Now we can formalize this:

$$\boxed{\text{Minimize } D(P_{data} \parallel P_\theta)}$$

In words: Find parameters θ that minimize the KL-divergence from P_θ to P_{data} .

Why This Makes Sense:

1. $D(P_{data} \parallel P_\theta) = 0 \implies$ Perfect match! (impossible in practice)
2. Smaller $D \implies$ Better approximation
3. Non-negative \implies Always a valid optimization objective
4. Information-theoretic meaning \implies Minimizes coding inefficiency

Mathematical Formulation:

$$\theta^* = \arg \min_{\theta} D(P_{data} \parallel P_\theta)$$

Where:

$$D(P_{data} \parallel P_\theta) = \sum_x P_{data}(x) \log \frac{P_{data}(x)}{P_\theta(x)}$$

Or in expectation form:

$$D(P_{data} \parallel P_\theta) = \mathbb{E}_{x \sim P_{data}} \left[\log \frac{P_{data}(x)}{P_\theta(x)} \right]$$

4.10 The Problem: We Don't Know P_{data} !

The Challenge:

To compute:

$$D(P_{data} \parallel P_\theta) = \sum_x P_{data}(x) \log \frac{P_{data}(x)}{P_\theta(x)}$$

We need to know $P_{data}(x)$ for all x , but:

- ✗ P_{data} is UNKNOWN!
- ✗ We only have samples $D = \{x^{(1)}, \dots, x^{(m)}\}$
- ✗ Cannot compute $D(P_{\text{data}} \parallel P_{\theta})$ directly

The Solution Preview:

We'll expand the KL-divergence and show that **some terms don't depend on θ** :

$$D(P_{\text{data}} \parallel P_{\theta}) = \underbrace{-\mathbb{E}_{x \sim P_{\text{data}}}[\log P_{\theta}(x)]}_{\text{Depends on } \theta} + \underbrace{H(P_{\text{data}})}_{\text{Constant (doesn't depend on } \theta)}$$

Therefore:

$$\begin{aligned} \arg \min_{\theta} D(P_{\text{data}} \parallel P_{\theta}) &= \arg \min_{\theta} (-\mathbb{E}_{x \sim P_{\text{data}}}[\log P_{\theta}(x)]) \\ &= \arg \max_{\theta} \mathbb{E}_{x \sim P_{\text{data}}}[\log P_{\theta}(x)] \end{aligned}$$

This is Expected Log-Likelihood! (Topic 5) ☺

4.11 Real-World Example: Image Generation

Let's see how KL-divergence applies to a **practical scenario**.

Setup:

Task: Generate realistic dog images

Distributions:

- P_{data} : Distribution of real dog images in nature
- P_{θ} : Our learned generative model

Scenario Analysis:

Case 1: Perfect Model

$P_{\theta}(x) = P_{\text{data}}(x)$ for all images x

$D(P_{\text{data}} \parallel P_{\theta}) = 0$

Result: Generated images indistinguishable from real dogs ✓

Case 2: Good Model

$P_{\theta}(x) \approx P_{\text{data}}(x)$ for most typical dog images

$D(P_{\text{data}} \parallel P_{\theta}) = 0.05$ (small)

Result: Generated images look realistic, minor artifacts

Case 3: Poor Model

$P_{\theta}(x)$ differs significantly from $P_{\text{data}}(x)$

$D(P_{\text{data}} \parallel P_{\theta}) = 2.5$ (large)

Result: Generated images look unrealistic, obvious artifacts

What KL-Divergence Measures:

High $P_{\text{data}}(x)$, Low $P_{\theta}(x)$:

- Real dogs that model can't generate well
- Model "misses" important dog types
- HIGH PENALTY in KL-divergence

Low $P_{\text{data}}(x)$, High $P_{\theta}(x)$:

- Model generates unrealistic images
- But weighted by P_{data} (low weight!)
- Lower penalty in KL-divergence

Key Insight: KL-divergence penalizes **missing real data** more than **generating fake data**!

4.12 Comparison Table: Properties

Property	KL-Divergence	Notes
Range	$[0, \infty)$	Never negative
Zero value	$D(p \parallel q) = 0 \iff p = q$	Equals zero only when distributions match
Symmetry	Asymmetric: $D(p \parallel q) \neq D(q \parallel p)$	NOT a distance metric
Triangle inequality	Does NOT satisfy	Not a true metric
Additivity	For independent variables: $D(p_{XY} \parallel q_{XY}) = D(p_X \parallel q_X) + D(p_Y \parallel q_Y)$	Useful property
Convexity	Convex in both arguments	Good for optimization
Units	Nats (natural log) or bits (\log_2)	Depends on log base

4.13 Connection to Entropy and Cross-Entropy

Definitions:

Entropy of p :

$$H(p) = - \sum_x p(x) \log p(x) = -\mathbb{E}_{x \sim p} [\log p(x)]$$

Cross-Entropy of p and q :

$$H(p, q) = - \sum_x p(x) \log q(x) = -\mathbb{E}_{x \sim p} [\log q(x)]$$

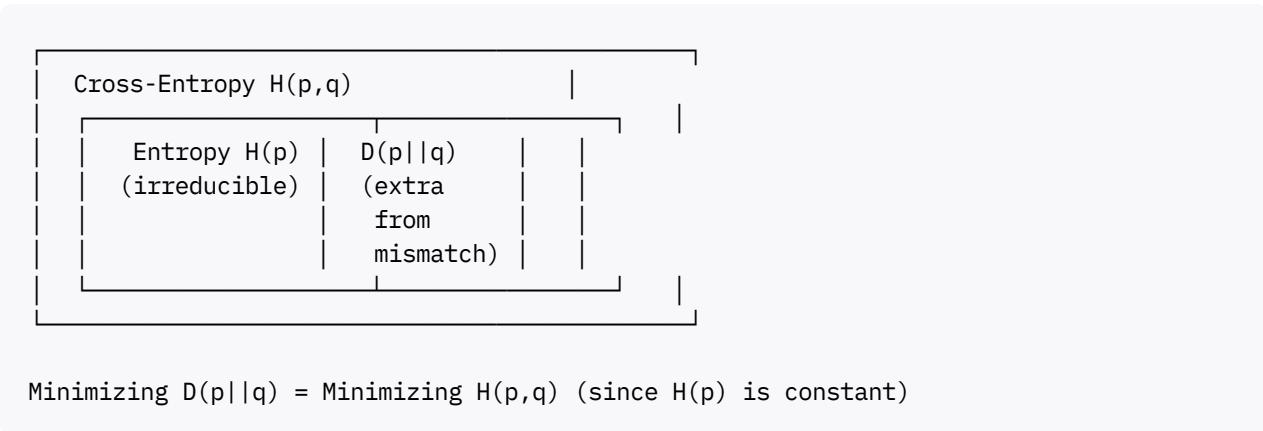
Relationship:

$$D(p \parallel q) = H(p, q) - H(p)$$

Proof:

$$\begin{aligned} D(p \parallel q) &= \sum_x p(x) \log \frac{p(x)}{q(x)} \\ &= \sum_x p(x) \log p(x) - \sum_x p(x) \log q(x) \\ &= -H(p) + H(p, q) = H(p, q) - H(p) \quad \checkmark \end{aligned}$$

Visual Representation:



✓ KEY TAKEAWAYS - TOPIC 4

Main Concepts:

1. Definition:

$$D(p \parallel q) = \sum_x p(x) \log \frac{p(x)}{q(x)} = \mathbb{E}_{x \sim p} \left[\log \frac{p(x)}{q(x)} \right]$$

2. Intuitive Meaning:

- Extra bits needed for compression when using wrong distribution
- Information-theoretic measure of distribution mismatch

3. Key Properties:

- ✓ Always non-negative: $D(p \parallel q) \geq 0$
- ✓ Zero iff distributions match: $D(p \parallel q) = 0 \iff p = q$
- ✗ Asymmetric: $D(p \parallel q) \neq D(q \parallel p)$

4. For Learning:

Minimize $D(P_{data} \parallel P_\theta) \implies$ Find best θ

5. Problem: Cannot compute directly because P_{data} is unknown

Connection to Previous Topics:

Topic 1: Want to learn $p(x)$

Topic 2: Have data D from P_{data} , model family M with parameters θ

Topic 3: Goal is density estimation - make P_θ close to P_{data}

Topic 4: Use KL-divergence to measure "close"

↓

Next: How to optimize this in practice?

□ Connection to Next Topic

We've established:

Goal: Minimize $D(P_{data} \parallel P_\theta)$

↓

Problem: Don't know P_{data} explicitly!

↓

Solution: ??? (Coming next!)

The key insight we'll derive:

$$D(P_{data} \parallel P_\theta) = -\mathbb{E}_{x \sim P_{data}} [\log P_\theta(x)] + \text{constant}$$

Therefore, minimizing KL-divergence = **maximizing Expected Log-Likelihood!**

This leads us to **TOPIC 5: EXPECTED LOG-LIKELIHOOD** □

Ready to continue to **TOPIC 5: EXPECTED LOG-LIKELIHOOD? Just say "next"!** □

TOPIC 5: EXPECTED LOG-LIKELIHOOD

(Connecting KL-Divergence to a Practical Objective)

5.0 Big Picture: Solving the P_{data} Problem

Recap: The Challenge from Topic 4

We want to minimize:

$$D(P_{data} \parallel P_\theta) = \sum_x P_{data}(x) \log \frac{P_{data}(x)}{P_\theta(x)}$$

But there's a problem:

- ✗ We don't know $P_{data}(x)$!
- ✗ Can't compute $D(P_{data} \parallel P_\theta)$ directly
- ✗ Can't minimize it!

The Breakthrough:

We'll show that:

```
Minimizing D(P_data || P_θ)
  ↓
  Is EQUIVALENT to
  ↓
  Maximizing Expected Log-Likelihood: E_{x~P_data} [log P_θ(x)]
  ↓
  Which we CAN work with! ✓
```

This is the **key mathematical insight** that makes learning tractable!

5.1 The Mathematical Derivation (Step-by-Step)

Let's derive this carefully, showing every step.

Starting Point:

$$D(P_{data} \parallel P_\theta) = \sum_x P_{data}(x) \log \frac{P_{data}(x)}{P_\theta(x)}$$

Step 1: Expand the Logarithm

Using the property: $\log \frac{a}{b} = \log a - \log b$

$$D(P_{data} \parallel P_\theta) = \sum_x P_{data}(x) [\log P_{data}(x) - \log P_\theta(x)]$$

Step 2: Distribute the Sum

$$D(P_{data} \parallel P_\theta) = \sum_x P_{data}(x) \log P_{data}(x) - \sum_x P_{data}(x) \log P_\theta(x)$$

Step 3: Identify the Components

Let's name these two terms:

First term (doesn't depend on θ):

$$\sum_x P_{data}(x) \log P_{data}(x) = -H(P_{data})$$

This is the **negative entropy** of P_{data} .

Second term (depends on θ):

$$-\sum_x P_{data}(x) \log P_\theta(x)$$

Step 4: Rewrite

$$D(P_{data} \parallel P_\theta) = \underbrace{-H(P_{data})}_{\text{No } \theta \text{ here!}} - \sum_x P_{data}(x) \log P_\theta(x)$$

Or equivalently:

$$D(P_{data} \parallel P_\theta) = -\mathbb{E}_{x \sim P_{data}} [\log P_\theta(x)] - H(P_{data})$$

Step 5: The Key Insight

Critical Observation:

$$H(P_{data}) = -\sum_x P_{data}(x) \log P_{data}(x)$$

This is the **entropy of the true data distribution**:

- It's a **property of the data alone**
- It does **NOT depend on our model parameters θ !**
- It's a **constant** with respect to optimization over θ

Step 6: Optimization Equivalence

When minimizing over θ :

$$\arg \min_{\theta} D(P_{data} \parallel P_{\theta}) = \arg \min_{\theta} [-\mathbb{E}_{x \sim P_{data}} [\log P_{\theta}(x)] - H(P_{data})]$$

Since $H(P_{data})$ is constant:

$$= \arg \min_{\theta} [-\mathbb{E}_{x \sim P_{data}} [\log P_{\theta}(x)]]$$

Minimizing the negative = Maximizing the positive:

$$= \boxed{\arg \max_{\theta} \mathbb{E}_{x \sim P_{data}} [\log P_{\theta}(x)]}$$

This is the Expected Log-Likelihood! ☐

5.2 Visual Derivation Flowchart

$$\begin{aligned} D(P_{data} \parallel P_{\theta}) &= \sum P_{data}(x) \log [P_{data}(x)/P_{\theta}(x)] \\ &\downarrow \text{[Expand log]} \\ &= \sum P_{data}(x) \log P_{data}(x) - \sum P_{data}(x) \log P_{\theta}(x) \\ &\downarrow \text{[Identify terms]} \\ &= -H(P_{data}) - \sum P_{data}(x) \log P_{\theta}(x) \\ &\downarrow \text{[}H(P_{data})\text{ is constant in } \theta\text{]} \\ \\ \arg \min_{\theta} D(P_{data} \parallel P_{\theta}) & \\ &\downarrow \\ &= \arg \min_{\theta} [-\sum P_{data}(x) \log P_{\theta}(x)] \\ &\downarrow \text{[Flip min/max]} \\ &= \arg \max_{\theta} [\sum P_{data}(x) \log P_{\theta}(x)] \\ &\downarrow \\ &= \arg \max_{\theta} \mathbb{E}_{x \sim P_{data}} [\log P_{\theta}(x)] \\ \\ &\checkmark \text{ EXPECTED LOG-LIKELIHOOD!} \end{aligned}$$

5.3 Definition: Expected Log-Likelihood

Formal Definition:

The **Expected Log-Likelihood** is:

$$\boxed{\ell(\theta) = \mathbb{E}_{x \sim P_{data}} [\log P_{\theta}(x)] = \sum_x P_{data}(x) \log P_{\theta}(x)}$$

Or in expectation notation:

$$\ell(\theta) = \mathbb{E}_{x \sim P_{data}} [\log P_\theta(x)]$$

In Words:

"The expected value of the log-probability assigned by our model P_θ to data points drawn from the true distribution P_{data} ."

Why "Expected"?

- We're taking an **expectation** (average) over the data distribution P_{data}
- Each data point x is weighted by $P_{data}(x)$

Why "Log-Likelihood"?

- $P_\theta(x)$ is the **likelihood** of x under our model
- We take the **log** for mathematical convenience
- "Log-likelihood" is standard terminology in statistics

5.4 Why This Solves Our Problem

The Beautiful Result:

$$\boxed{\min_{\theta} D(P_{data} \parallel P_\theta) \equiv \max_{\theta} \mathbb{E}_{x \sim P_{data}} [\log P_\theta(x)]}$$

Why This is Progress:

Before (Topic 4):

```
Minimize: D(P_data || P_θ) = Σ P_data(x) log[P_data(x)/P_θ(x)]
          ↑
          Need P_data(x) explicitly ✗
```

After (Topic 5):

```
Maximize: E_x~P_data[log P_θ(x)] = Σ P_data(x) log P_θ(x)
          ↑
          Still need P_data(x)...
          But wait! We can approximate with samples! ✓
```

The Key Insight:

While we still have P_{data} in the formula, **expectations can be approximated using samples!**

We have: Training data $D = \{x^{(1)}, x^{(2)}, \dots, x^{(m)}\} \sim P_{data}$
↓
Can approximate: $E_{x \sim P_{data}}[\log P_\theta(x)] \approx (1/m) \sum_i \log P_\theta(x^{(i)})$
↓
This we CAN compute! ✓

This will lead to Maximum Likelihood Estimation (Topic 6)! ☺

5.5 Intuitive Interpretation

What Does Expected Log-Likelihood Measure?

Interpretation 1: Average Model Quality

$$\mathbb{E}_{x \sim P_{data}}[\log P_\theta(x)]$$

Measures: "On average, how much log-probability does our model P_θ assign to real data points?"

High Expected Log-Likelihood:

- Model assigns HIGH probability to real data
- Model is GOOD! ✓

Low Expected Log-Likelihood:

- Model assigns LOW probability to real data
- Model is BAD! ✗

Interpretation 2: Model Confidence

$\log P_\theta(x)$ for $x \sim P_{data}$:

$\log P_\theta(x) \approx 0$	$\rightarrow P_\theta(x) \approx 1$	\rightarrow Very confident (good!)
$\log P_\theta(x) \approx -1$	$\rightarrow P_\theta(x) \approx 0.37$	\rightarrow Moderate confidence
$\log P_\theta(x) \approx -5$	$\rightarrow P_\theta(x) \approx 0.007$	\rightarrow Low confidence (bad!)
$\log P_\theta(x) \approx -\infty$	$\rightarrow P_\theta(x) \approx 0$	\rightarrow Zero confidence (disaster!)

Expected Log-Likelihood averages this confidence over all real data.

Interpretation 3: Compression Efficiency (Revisited)

From information theory (Topic 4):

- $-\log P_\theta(x)$ = bits needed to encode x using code based on P_θ

Therefore:

$$-\mathbb{E}_{x \sim P_{data}} [\log P_\theta(x)] = \text{Average bits needed to encode real data}$$

Maximizing expected log-likelihood = Minimizing average code length \square

5.6 Detailed Mathematical Example

Let's compute expected log-likelihood for a **concrete example**.

Example Setup: Three-Outcome Distribution

True distribution P_{data} :

$$\begin{aligned}P_{data}(A) &= 0.5 \\P_{data}(B) &= 0.3 \\P_{data}(C) &= 0.2\end{aligned}$$

Model distribution P_θ (parametrized):

$$\begin{aligned}P_\theta(A) &= \theta_1 = 0.4 \\P_\theta(B) &= \theta_2 = 0.4 \\P_\theta(C) &= \theta_3 = 0.2\end{aligned}$$

(where $\theta_1 + \theta_2 + \theta_3 = 1$)

Step 1: Compute Log-Probabilities

x	$P_\theta(x)$	$\log P_\theta(x)$
A	0.4	$\log(0.4) = -0.916$
B	0.4	$\log(0.4) = -0.916$
C	0.2	$\log(0.2) = -1.386$

Step 2: Weight by P_{data}

$$\mathbb{E}_{x \sim P_{data}} [\log P_\theta(x)] = \sum_x P_{data}(x) \log P_\theta(x)$$

For $x = A$:

$$P_{data}(A) \times \log P_\theta(A) = 0.5 \times (-0.916) = -0.458$$

For $x = B$:

$$P_{data}(B) \times \log P_{\theta}(B) = 0.3 \times (-0.916) = -0.275$$

For $x = C$:

$$P_{data}(C) \times \log P_{\theta}(C) = 0.2 \times (-1.609) = -0.322$$

Step 3: Sum Up

$$\ell(\theta) = -0.458 + (-0.275) + (-0.322) = -1.055$$

Expected Log-Likelihood: $\ell(\theta) = -1.055$ nats

Interpretation:

- Negative value is normal (\log of probabilities < 1)
- Higher (less negative) is better
- This model assigns reasonable probability to real data

Comparison with Different Model:

Alternative model P'_{θ} :

```
P'_θ(A) = 0.1 (BAD: P_data(A) = 0.5, but model gives only 0.1!)
```

```
P'_θ(B) = 0.5
```

```
P'_θ(C) = 0.4
```

Expected Log-Likelihood:

$$\begin{aligned}\ell'(\theta) &= 0.5 \times \log(0.1) + 0.3 \times \log(0.5) + 0.2 \times \log(0.4) \\ &= 0.5 \times (-2.303) + 0.3 \times (-0.693) + 0.2 \times (-0.916) \\ &= -1.152 + (-0.208) + (-0.183) = -1.543\end{aligned}$$

Comparison:

```
Original model: ℓ(θ) = -1.055 ✓ Better!
Alternative:   ℓ'(θ) = -1.543 ✗ Worse!
```

Why? Alternative model puts low probability (0.1) on outcome A, which is actually common in P_{data} (0.5)!

5.7 Properties of Expected Log-Likelihood

Property 1: Upper Bound

$$\mathbb{E}_{x \sim P_{data}} [\log P_\theta(x)] \leq -H(P_{data})$$

Equality when: $P_\theta = P_{data}$

Meaning:

- The best possible expected log-likelihood is the negative entropy of P_{data}
- We can never exceed this (unless we perfectly match P_{data})

Property 2: Penalizes Low Probability on Real Data

Because of the **log function**:

If $P_\theta(x) \approx 0$ for some x where $P_{data}(x) > 0$:
→ $\log P_\theta(x) \rightarrow -\infty$
→ Huge penalty!

This is important: Models that assign near-zero probability to any real data are heavily penalized.

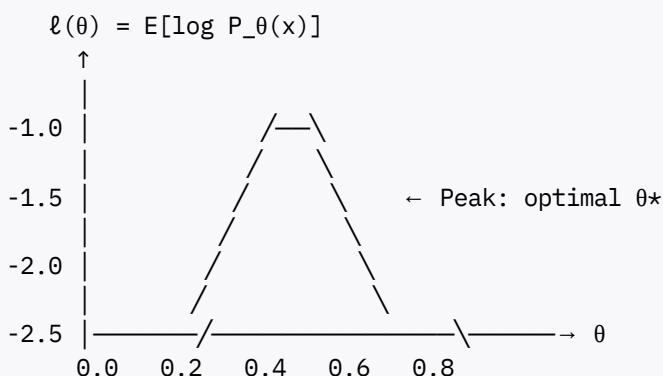
Property 3: Concave in θ

For many model families (e.g., exponential families), $\log P_\theta(x)$ is **concave** in θ .

Implication: Optimization is often easier (local optima may be global).

5.8 Visual Representation

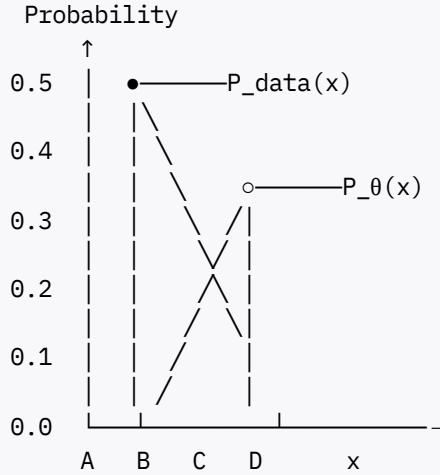
Graph: Expected Log-Likelihood as Function of θ



Goal: Find θ^* that maximizes $\ell(\theta)$

Visual: Effect of Mismatch

P_{data} vs P_{θ} :



Where P_{data} is high but P_{θ} is low:

- Large negative contribution to $\ell(\theta)$
- Model is bad at those x !

5.9 Connection to Autoregressive Models (Lecture 3)

Recall from Lecture 3:

For autoregressive models:

$$P_{\theta}(x) = \prod_{i=1}^n p_{\text{neural}}(x_i | x_{<i}; \theta_i)$$

Log-Likelihood Decomposes:

$$\log P_{\theta}(x) = \log \left[\prod_{i=1}^n p_{\text{neural}}(x_i | x_{<i}; \theta_i) \right]$$

Using $\log(\prod) = \sum \log$:

$$\log P_{\theta}(x) = \sum_{i=1}^n \log p_{\text{neural}}(x_i | x_{<i}; \theta_i)$$

Expected Log-Likelihood:

$$\mathbb{E}_{x \sim P_{\text{data}}} [\log P_{\theta}(x)] = \mathbb{E}_{x \sim P_{\text{data}}} \left[\sum_{i=1}^n \log p_{\text{neural}}(x_i | x_{<i}; \theta_i) \right]$$

By linearity of expectation:

$$= \sum_{i=1}^n \mathbb{E}_{x \sim P_{data}} [\log p_{neural}(x_i | x_{<i}; \theta_i)]$$

Key Insight: We maximize the sum of expected log-probabilities for each conditional!

Practical Implication:

For autoregressive models:

Maximizing expected log-likelihood =
Maximizing each conditional's expected log-probability

This is why AR models are natural for maximum likelihood! ✓

5.10 Detailed Example: Image Model

Setup: Binary Image Generation

Domain: 28×28 binary images (MNIST-like)

- $n = 784$ pixels
- Each pixel: $x_i \in \{0, 1\}$

Model: Autoregressive (PixelCNN-style)

$$P_\theta(x) = \prod_{i=1}^{784} p_{neural}(x_i | x_{<i}; \theta)$$

Sample Calculation:

For one image $x^{(1)}$:

Suppose the model predicts:

$p_{neural}(x_1=0 \emptyset) = 0.9$	$\rightarrow \log p = -0.105$
$p_{neural}(x_2=1 x_1) = 0.8$	$\rightarrow \log p = -0.223$
$p_{neural}(x_3=1 x_1, x_2) = 0.95$	$\rightarrow \log p = -0.051$
...	
$p_{neural}(x_{784}=0 x_1, \dots, x_{783}) = 0.85$	$\rightarrow \log p = -0.163$

Log-likelihood for this image:

$$\begin{aligned} \log P_\theta(x^{(1)}) &= \sum_{i=1}^{784} \log p_{neural}(x_i | x_{<i}) \\ &= -0.105 + (-0.223) + (-0.051) + \dots + (-0.163) \end{aligned}$$

$$\approx -156.3 \text{ (example value)}$$

Expected Log-Likelihood:

Average over all possible images weighted by P_{data} :

$$\ell(\theta) = \mathbb{E}_{x \sim P_{data}} [\log P_\theta(x)]$$

Cannot compute exactly, but can approximate:

$$\ell(\theta) \approx \frac{1}{m} \sum_{j=1}^m \log P_\theta(x^{(j)})$$

where $x^{(j)} \in D$ are training images.

This is Maximum Likelihood Estimation! (Next topic) [\[link\]](#)

5.11 Why "Expected" Matters

Comparison: Expected vs Empirical

Expected Log-Likelihood	Empirical Log-Likelihood
$\mathbb{E}_{x \sim P_{data}} [\log P_\theta(x)]$	$\frac{1}{m} \sum_{i=1}^m \log P_\theta(x^{(i)})$
Theoretical (uses true P_{data})	Practical (uses training data)
Infinite data	Finite data m samples
Cannot compute directly	Can compute!
Target we want to maximize	Approximation we actually maximize

Why the Distinction Matters:

Expected (Theoretical):

Measures: How well model performs on TRUE distribution P_{data}
Including unseen data!

Empirical (Practical):

Measures: How well model performs on TRAINING data only
Might overfit if not careful!

The Gap:

$$\left| \mathbb{E}_{x \sim P_{data}} [\log P_\theta(x)] - \frac{1}{m} \sum_{i=1}^m \log P_\theta(x^{(i)}) \right|$$

- Decreases as $m \rightarrow \infty$ (law of large numbers)
- For large m , empirical \approx expected ✓

5.12 Mathematical Properties Summary

Key Properties:

1. Equivalence to KL Minimization:

$$\arg \max_{\theta} \mathbb{E}_{x \sim P_{data}} [\log P_{\theta}(x)] = \arg \min_{\theta} D(P_{data} \parallel P_{\theta})$$

2. Decomposition for AR Models:

$$\mathbb{E}[\log P_{\theta}(x)] = \sum_{i=1}^n \mathbb{E}[\log p(x_i | x_{<i}; \theta)]$$

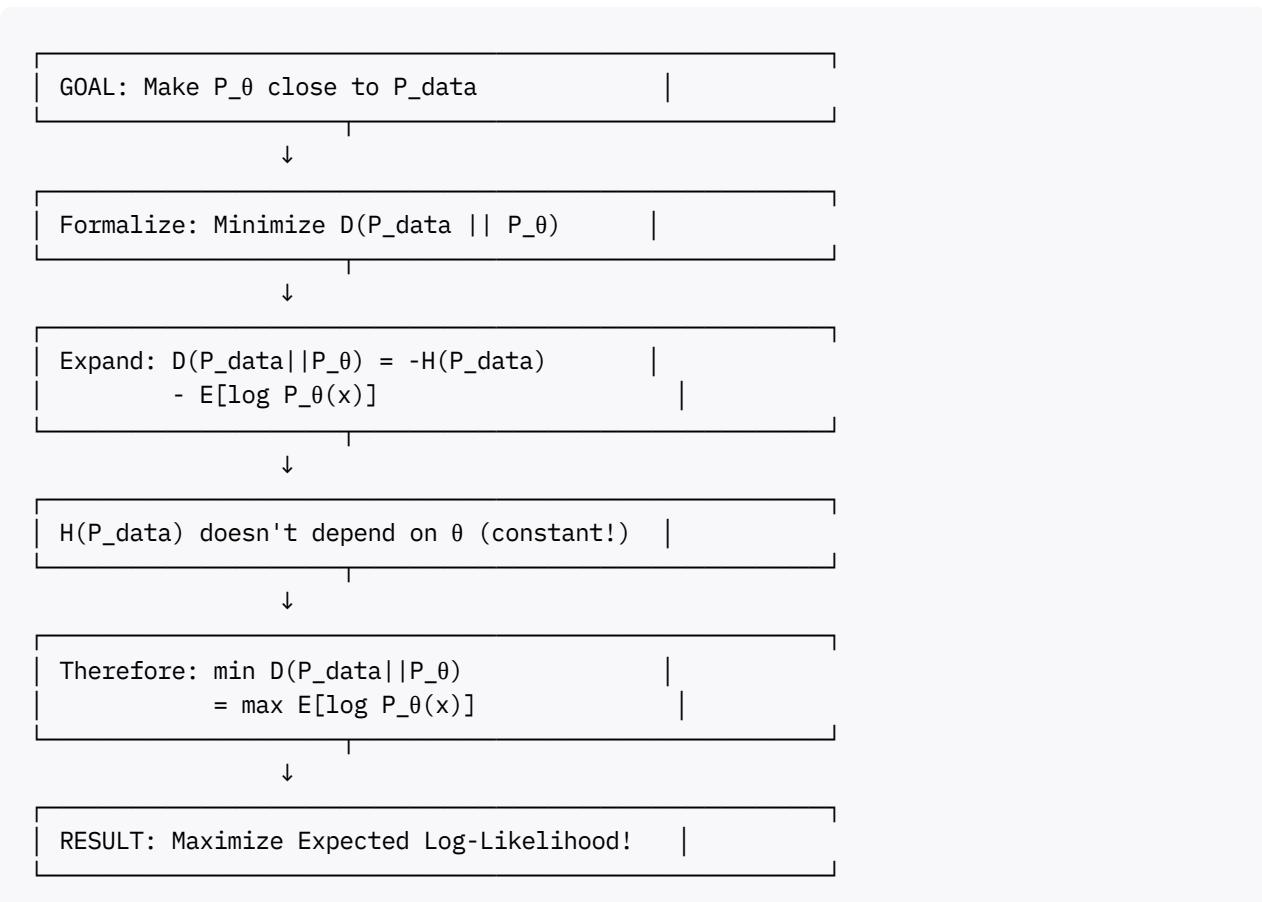
3. Upper Bound:

$$\mathbb{E}_{x \sim P_{data}} [\log P_{\theta}(x)] \leq -H(P_{data})$$

4. Sensitivity to Zero Probability:

If $P_{\theta}(x) = 0$ for any x where $P_{data}(x) > 0$: $\mathbb{E}[\log P_{\theta}(x)] = -\infty$

5.13 Flowchart: Complete Derivation Chain



✓ KEY TAKEAWAYS - TOPIC 5

Main Results:

1. Equivalence Theorem:

$$\arg \min_{\theta} D(P_{data} \parallel P_{\theta}) \equiv \arg \max_{\theta} \mathbb{E}_{x \sim P_{data}} [\log P_{\theta}(x)]$$

2. Expected Log-Likelihood Definition:

$$\ell(\theta) = \mathbb{E}_{x \sim P_{data}} [\log P_{\theta}(x)] = \sum_x P_{data}(x) \log P_{\theta}(x)$$

3. Why This Helps:

- Removed the troublesome $P_{data}(x)/P_{\theta}(x)$ ratio
- Only need $\log P_{\theta}(x)$ (which our model provides!)
- Can approximate expectation with samples

4. Intuitive Meaning:

- "Average log-probability our model assigns to real data"
- Higher is better (less negative = more confident)
- Penalizes low probability on real data

The Derivation Chain:

```
Minimize KL-divergence
↓
Expand: D(p || q) = -H(p) - E[log q(x)]
↓
H(P_data) is constant in θ
↓
Equivalent to maximize E[log P_θ(x)]
↓
Expected Log-Likelihood! ✓
```

Connection to Previous Topics:

```
Topic 1: Want to learn p(x) from data
Topic 2: Have setup: P_data, D, IID, M
Topic 3: Goal is density estimation
Topic 4: Use KL-divergence to measure closeness
Topic 5: KL-minimization = Expected log-likelihood maximization
↓
Next: How to compute this in practice?
```

□ Connection to Next Topic

We've made great progress:

- ✓ Defined learning goal (minimize KL-divergence)
- ✓ Showed it's equivalent to maximizing expected log-likelihood
- ✓ Understand what expected log-likelihood means

But still a problem:

- ✗ Expected log-likelihood still involves $P_{\text{data}}(x)$!
$$E[\log P_{\theta}(x)] = \sum P_{\text{data}}(x) \log P_{\theta}(x)$$

↑
Still need P_{data} !

The final step:

Approximate the expectation using **training data samples**:

$$\mathbb{E}_{x \sim P_{\text{data}}} [\log P_{\theta}(x)] \approx \frac{1}{m} \sum_{j=1}^m \log P_{\theta}(x^{(j)})$$

This is **Maximum Likelihood Estimation!**

This leads us to **TOPIC 6: MAXIMUM LIKELIHOOD** □

Where we'll learn:

- How to use training data to approximate expected log-likelihood
- The Maximum Likelihood Estimation (MLE) principle
- Practical optimization objectives
- Connection to empirical risk

Ready to continue to **TOPIC 6: MAXIMUM LIKELIHOOD?** Just say "next"! □

□ **TOPIC 6: MAXIMUM LIKELIHOOD**

(From Theory to Practice: Using Training Data)

6.0 Big Picture: The Final Piece of the Puzzle

The Journey So Far:

- ```
Topic 1: Want to learn p(x) from data
 ↓
Topic 2: Setup with P_data, training data D, model family M
 ↓
```

```

Topic 3: Goal = Density estimation (make $P_\theta \approx P_{\text{data}}$)
↓
Topic 4: Measure closeness with KL-divergence
↓
Topic 5: $\min D(P_{\text{data}} || P_\theta) \equiv \max E_{x \sim P_{\text{data}}}[\log P_\theta(x)]$
↓
Topic 6: ??? How to compute this ???

```

## The Remaining Challenge:

From Topic 5, we want to maximize:

$$\mathbb{E}_{x \sim P_{\text{data}}}[\log P_\theta(x)] = \sum_x P_{\text{data}}(x) \log P_\theta(x)$$

### Problem:

- ✗ Still involves  $P_{\text{data}}(x)$ !
- ✗ We don't know  $P_{\text{data}}(x)$  for all  $x$
- ✗ Cannot compute expectation directly

## The Breakthrough Solution:

**Key Insight:** We have **samples** from  $P_{\text{data}}$ !

```

Training data: $D = \{x^{(1)}, x^{(2)}, \dots, x^{(m)}\} \sim P_{\text{data}}$
↓
Use Monte Carlo approximation!
↓
 $E[\log P_\theta(x)] \approx (1/m) \sum \log P_\theta(x^{(i)})$
↓
This we CAN compute! ✓

```

This is Maximum Likelihood Estimation (MLE)! ☐

## 6.1 Monte Carlo Approximation (Quick Recap from Lecture)

### Monte Carlo Principle:

To estimate an expectation  $\mathbb{E}_{x \sim P}[f(x)]$ :

#### Three Steps:

1. **Express** the quantity as an expected value:  $\mathbb{E}_{x \sim P}[f(x)]$
2. **Sample**  $T$  independent samples from  $P$ :  $x^1, x^2, \dots, x^T \sim P$
3. **Estimate** using sample average:

$$\hat{f} = \frac{1}{T} \sum_{t=1}^T f(x^t) \approx \mathbb{E}_{x \sim P}[f(x)]$$

**Properties (from Lecture Slide 13):**

**1. Unbiased:**

$$\mathbb{E}_P[\hat{f}] = \mathbb{E}_P\left[\frac{1}{T} \sum_{t=1}^T f(x^t)\right] = \frac{1}{T} \sum_{t=1}^T \mathbb{E}_P[f(x^t)] = \mathbb{E}_P[f(x)]$$

**2. Convergence (Law of Large Numbers):**

$$\hat{f} \rightarrow \mathbb{E}_P[f(x)] \quad \text{as } T \rightarrow \infty$$

**3. Variance:**

$$\text{Var}[\hat{f}] = \frac{\text{Var}_P[f(x)]}{T}$$

**Interpretation:** Variance decreases as  $1/T \rightarrow$  More samples = Better estimate!

## 6.2 Applying Monte Carlo to Expected Log-Likelihood

### Step 1: Identify the Components

**Expectation to estimate:**

$$\mathbb{E}_{x \sim P_{\text{data}}}[\log P_\theta(x)]$$

**Map to Monte Carlo framework:**

- **Distribution  $P$ :**  $P_{\text{data}}$  (the true data distribution)
- **Function  $f(x)$ :**  $\log P_\theta(x)$  (log-probability from our model)
- **Samples:**  $D = \{x^{(1)}, \dots, x^{(m)}\}$  (our training data!)

### Step 2: Apply Monte Carlo Approximation

$$\mathbb{E}_{x \sim P_{\text{data}}}[\log P_\theta(x)] \approx \frac{1}{m} \sum_{j=1}^m \log P_\theta(x^{(j)})$$

Where:

- $m = |D|$  = number of training examples
- $x^{(j)} \in D$  are the training samples

**This approximation is called the EMPIRICAL LOG-LIKELIHOOD!**

## 6.3 Definition: Empirical Log-Likelihood

### Formal Definition:

The **Empirical Log-Likelihood** (also called **Log-Likelihood of the Data**) is:

$$\ell_{\text{empirical}}(\theta; D) = \frac{1}{m} \sum_{j=1}^m \log P_\theta(x^{(j)})$$

**Alternative notation (without averaging):**

$$\log L(\theta; D) = \sum_{j=1}^m \log P_\theta(x^{(j)})$$

### Terminology:

- **Likelihood:**  $L(\theta; D) = \prod_{j=1}^m P_\theta(x^{(j)})$
- **Log-Likelihood:**  $\log L(\theta; D) = \sum_{j=1}^m \log P_\theta(x^{(j)})$
- **Empirical/Average Log-Likelihood:**  $\frac{1}{m} \sum_{j=1}^m \log P_\theta(x^{(j)})$

### In Words:

"The empirical log-likelihood is the average log-probability that our model  $P_\theta$  assigns to the training examples in  $D$ ."

## 6.4 The Maximum Likelihood Principle

### Mathematical Statement:

**Maximum Likelihood Estimation (MLE)** seeks parameters  $\theta^*$  that maximize the empirical log-likelihood:

$$\theta^* = \arg \max_{\theta} \frac{1}{m} \sum_{j=1}^m \log P_\theta(x^{(j)})$$

**Equivalent forms** (since  $\frac{1}{m}$  is constant):

$$\theta^* = \arg \max_{\theta} \sum_{j=1}^m \log P_\theta(x^{(j)}) = \arg \max_{\theta} \log L(\theta; D)$$

Or equivalently (maximize likelihood directly):

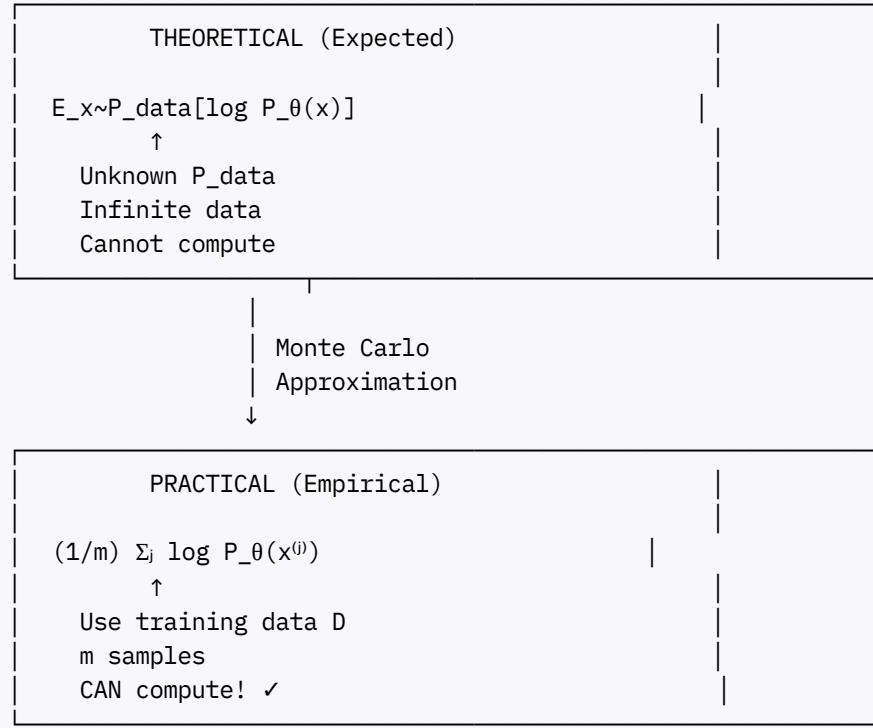
$$\theta^* = \arg \max_{\theta} L(\theta; D) = \arg \max_{\theta} \prod_{j=1}^m P_\theta(x^{(j)})$$

## The Complete Chain of Reasoning:

```
Goal: Make P_θ close to P_data
↓
Minimize D(P_data || P_θ)
↓
Equivalent to: Maximize Ex~P_data[log P_θ(x)]
↓
Approximate with samples: Maximize (1/m)Σ log P_θ(x(j))
↓
MAXIMUM LIKELIHOOD ESTIMATION! ✓
```

## 6.5 Visual Representation

### Diagram: From Theory to Practice



### Visualization: Sample Approximation

True Expectation (Unknown):  
 $E[\log P_\theta(x)] = \int P_{\text{data}}(x) \log P_\theta(x) dx$

Continuous integral  
over entire space

Empirical Approximation (Computable):  
 $(1/m) \sum \log P_\theta(x^{(j)})$

Discrete sum  
over  $m$  samples

As  $m \rightarrow \infty$ : Empirical  $\rightarrow$  True (Law of Large Numbers)

## 6.6 Detailed Mathematical Example: Coin Flipping

Let's work through the **coin example from Lecture Slide 14-15** in complete detail.

### Setup:

**Domain:** Coin flips with two outcomes

- Heads (H)
- Tails (T)

### Training Data:

$$D = \{H, H, T, H, T\}$$

So  $m = 5$  samples:

- Number of heads: 3
- Number of tails: 2

### Model Family:

#### Parametrization:

$$P_\theta(x = H) = \theta$$

$$P_\theta(x = T) = 1 - \theta$$

Where  $\theta \in [0, 1]$  is the single parameter (probability of heads).

### Goal:

Find  $\theta^*$  that maximizes the likelihood of the observed data  $D$ .

## 6.7 MLE for Coin Example: Step-by-Step Solution

### Step 1: Write the Likelihood Function

The likelihood of observing data  $D$  is:

$$L(\theta; D) = P_\theta(H) \cdot P_\theta(H) \cdot P_\theta(T) \cdot P_\theta(H) \cdot P_\theta(T)$$

$$L(\theta; D) = \theta \cdot \theta \cdot (1 - \theta) \cdot \theta \cdot (1 - \theta)$$

$$L(\theta; D) = \theta^3 \cdot (1 - \theta)^2$$

**General form:**

$$L(\theta; D) = \theta^{n_H} \cdot (1 - \theta)^{n_T}$$

Where:

- $n_H = 3$  (number of heads)
- $n_T = 2$  (number of tails)

### Step 2: Take the Log

$$\log L(\theta; D) = \log[\theta^{n_H} \cdot (1 - \theta)^{n_T}]$$

Using  $\log(ab) = \log a + \log b$  and  $\log(a^n) = n \log a$ :

$$\log L(\theta; D) = n_H \log \theta + n_T \log(1 - \theta)$$

$$\log L(\theta; D) = 3 \log \theta + 2 \log(1 - \theta)$$

### Step 3: Take the Derivative

To find the maximum, take derivative with respect to  $\theta$  and set to zero:

$$\begin{aligned} \frac{d}{d\theta} \log L(\theta; D) &= \frac{d}{d\theta} [3 \log \theta + 2 \log(1 - \theta)] \\ &= 3 \cdot \frac{1}{\theta} + 2 \cdot \frac{1}{1 - \theta} \cdot (-1) \\ &= \frac{3}{\theta} - \frac{2}{1 - \theta} \end{aligned}$$

### Step 4: Set Derivative to Zero

$$\frac{3}{\theta} - \frac{2}{1 - \theta} = 0$$

$$\frac{3}{\theta} = \frac{2}{1 - \theta}$$

Cross-multiply:

$$3(1 - \theta) = 2\theta$$

$$3 - 3\theta = 2\theta$$

$$3 = 5\theta$$

$$\theta^* = \frac{3}{5} = 0.6$$

### Step 5: Verify it's a Maximum

**Second derivative test:**

$$\frac{d^2}{d\theta^2} \log L(\theta; D) = -\frac{3}{\theta^2} - \frac{2}{(1-\theta)^2} < 0$$

Since second derivative is negative,  $\theta^* = 0.6$  is indeed a **maximum** ✓

### Step 6: Interpretation

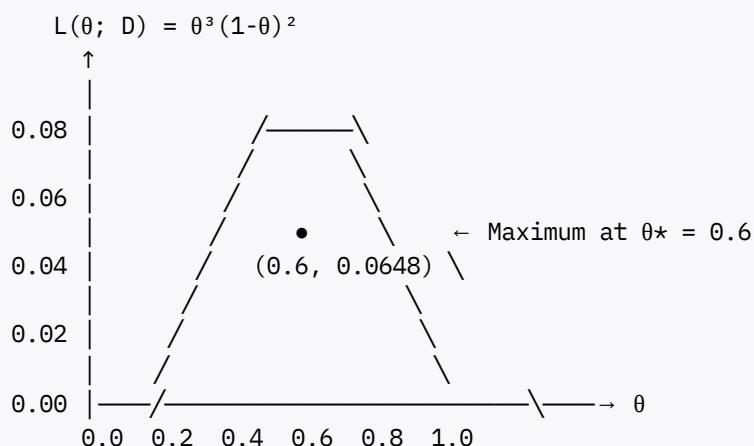
$$\theta^* = \frac{3}{5} = \frac{\text{Number of heads}}{\text{Total flips}}$$

**Intuitive Result:** The MLE for coin flip probability is simply the **empirical frequency**!

**Makes sense:** Best estimate of probability = observed proportion in data.

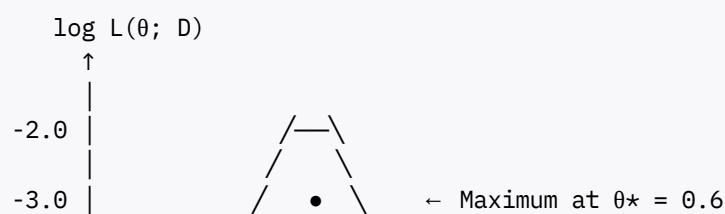
## 6.8 Graphical Visualization: Coin Example

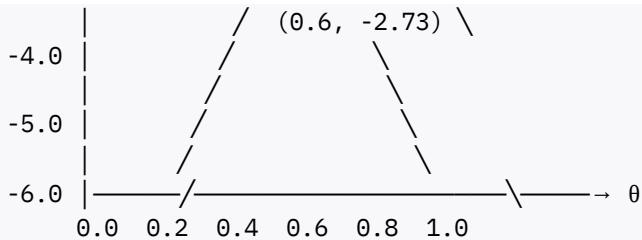
### Graph: Likelihood Function



Peak at  $\theta = 0.6$  (exactly 3/5, the proportion of heads!)

### Graph: Log-Likelihood Function





Concave function (easier to optimize!)

## 6.9 Numerical Verification: Coin Example

Let's verify by computing log-likelihood at different  $\theta$  values.

### Log-Likelihood Formula:

$$\log L(\theta; D) = 3 \log \theta + 2 \log(1 - \theta)$$

### Calculations:

| $\theta$   | $\log \theta$ | $3 \log \theta$ | $\log(1 - \theta)$ | $2 \log(1 - \theta)$ | Total $\log L(\theta; D)$ |
|------------|---------------|-----------------|--------------------|----------------------|---------------------------|
| 0.1        | -2.303        | -6.908          | -0.105             | -0.211               | <b>-7.119</b>             |
| 0.3        | -1.204        | -3.611          | -0.357             | -0.714               | <b>-4.325</b>             |
| 0.5        | -0.693        | -2.079          | -0.693             | -1.386               | <b>-3.466</b>             |
| <b>0.6</b> | <b>-0.511</b> | <b>-1.533</b>   | <b>-0.916</b>      | <b>-1.833</b>        | <b>-3.365 ✓ Best!</b>     |
| 0.7        | -0.357        | -1.071          | -1.204             | -2.408               | <b>-3.478</b>             |
| 0.9        | -0.105        | -0.316          | -2.303             | -4.605               | <b>-4.921</b>             |

**Observation:**  $\theta = 0.6$  gives the **highest** (least negative) log-likelihood! ✓

## 6.10 General Insight from Coin Example

### The Beautiful Result:

For **simple distributions**, MLE often gives **closed-form solutions** that match our intuition.

### Coin flip:

$$\theta_{\text{MLE}}^* = \frac{\text{count of heads}}{\text{total flips}}$$

### Gaussian mean:

$$\mu_{\text{MLE}}^* = \frac{1}{m} \sum_{j=1}^m x^{(j)} \quad (\text{sample mean})$$

**Gaussian variance:**

$$\sigma_{\text{MLE}}^{2*} = \frac{1}{m} \sum_{j=1}^m (x^{(j)} - \mu)^2 \quad (\text{sample variance})$$

**General Pattern:** MLE gives **statistics** that match observed data!

## 6.11 MLE for Autoregressive Models (No Closed Form)

**Recall from Lecture Slide 16:**

For **autoregressive models** with neural networks:

$$P_\theta(x) = \prod_{i=1}^n p_{\text{neural}}(x_i | x_{<i}; \theta_i)$$

Where  $\theta = (\theta_1, \dots, \theta_n)$  are all parameters.

**Likelihood Function:**

$$L(\theta; D) = \prod_{j=1}^m P_\theta(x^{(j)}) = \prod_{j=1}^m \prod_{i=1}^n p_{\text{neural}}(x_i^{(j)} | x_{<i}^{(j)}; \theta_i)$$

**Log-Likelihood:**

$$\log L(\theta; D) = \sum_{j=1}^m \sum_{i=1}^n \log p_{\text{neural}}(x_i^{(j)} | x_{<i}^{(j)}; \theta_i)$$

**The Problem:**

**No closed-form solution!**

- ✗ Cannot solve  $d/d\theta [\log L(\theta; D)] = 0$  analytically
- ✗ Neural networks have millions of parameters
- ✗ Non-convex optimization problem

**The Solution:**

**Gradient-based optimization!** (Next Topic 7)

$$\theta^{(t+1)} = \theta^{(t)} + \alpha_t \nabla_\theta \log L(\theta; D)$$

Use **backpropagation** to compute gradients efficiently.

## 6.12 Maximum Likelihood as an Optimization Problem

**Formal Statement:**

$$\theta^* = \arg \max_{\theta \in \Theta} \frac{1}{m} \sum_{j=1}^m \log P_\theta(x^{(j)})$$

**Components:**

**Objective Function:**

$$\ell(\theta) = \frac{1}{m} \sum_{j=1}^m \log P_\theta(x^{(j)})$$

**Optimization:**

- **Type:** Maximization
- **Decision variables:**  $\theta \in \Theta$
- **Constraint set:**  $\Theta$  (parameter space, e.g., probabilities sum to 1)

**Alternative: Minimization Form**

Equivalently, **minimize negative log-likelihood (NLL):**

$$\theta^* = \arg \min_{\theta} -\frac{1}{m} \sum_{j=1}^m \log P_\theta(x^{(j)})$$

**This is the form commonly used in machine learning frameworks!**

**Why Minimization?**

- Most optimization libraries are designed for **minimization**
- Gradient descent naturally **descends** (minimizes)
- Loss function terminology: "loss" should be minimized

## 6.13 Detailed Example: Three-Variable Discrete Model

Let's work through a **more complex example** step-by-step.

**Setup:**

**Variables:**  $X_1, X_2, X_3$ , each binary  $\{0, 1\}$

**Model:** Autoregressive factorization

$$P_\theta(x_1, x_2, x_3) = P_\theta(x_1) \cdot P_\theta(x_2|x_1) \cdot P_\theta(x_3|x_1, x_2)$$

### Parameters:

- $\theta_1 = P(X_1 = 1)$
- $\theta_2(x_1) = P(X_2 = 1 | X_1 = x_1)$  for each  $x_1 \in \{0, 1\}$
- $\theta_3(x_1, x_2) = P(X_3 = 1 | X_1 = x_1, X_2 = x_2)$  for each combination

**Training Data:**  $D = \{x^{(1)}, x^{(2)}, x^{(3)}\}$

$$\begin{aligned} x^{(1)} &= (1, 0, 1) \rightarrow X_1=1, X_2=0, X_3=1 \\ x^{(2)} &= (1, 1, 1) \rightarrow X_1=1, X_2=1, X_3=1 \\ x^{(3)} &= (0, 1, 0) \rightarrow X_1=0, X_2=1, X_3=0 \end{aligned}$$

### Step 1: Likelihood for Each Sample

For  $x^{(1)} = (1, 0, 1)$ :

$$\begin{aligned} P_\theta(x^{(1)}) &= P(X_1 = 1) \cdot P(X_2 = 0 | X_1 = 1) \cdot P(X_3 = 1 | X_1 = 1, X_2 = 0) \\ &= \theta_1 \cdot (1 - \theta_2(1)) \cdot \theta_3(1, 0) \end{aligned}$$

For  $x^{(2)} = (1, 1, 1)$ :

$$P_\theta(x^{(2)}) = \theta_1 \cdot \theta_2(1) \cdot \theta_3(1, 1)$$

For  $x^{(3)} = (0, 1, 0)$ :

$$P_\theta(x^{(3)}) = (1 - \theta_1) \cdot \theta_2(0) \cdot (1 - \theta_3(0, 1))$$

### Step 2: Total Log-Likelihood

$$\begin{aligned} \log L(\theta; D) &= \sum_{j=1}^3 \log P_\theta(x^{(j)}) \\ &= \log[\theta_1(1 - \theta_2(1))\theta_3(1, 0)] \\ &\quad + \log[\theta_1\theta_2(1)\theta_3(1, 1)] \\ &\quad + \log[(1 - \theta_1)\theta_2(0)(1 - \theta_3(0, 1))] \end{aligned}$$

### Step 3: Decompose by Parameters

$$\begin{aligned} \log L(\theta; D) &= 2 \log \theta_1 + \log(1 - \theta_1) \\ &\quad + \log(1 - \theta_2(1)) + \log \theta_2(1) + \log \theta_2(0) \\ &\quad + \log \theta_3(1, 0) + \log \theta_3(1, 1) + \log(1 - \theta_3(0, 1)) \end{aligned}$$

## Step 4: MLE Solution (Closed Form!)

For autoregressive models with **no parameter sharing**, each conditional can be optimized **independently!**

For  $\theta_1$ :

- Appears in: 2 samples with  $X_1 = 1$ , 1 sample with  $X_1 = 0$
- MLE:  $\theta_1^* = 2/3 \checkmark$

For  $\theta_2(1)$ : (conditional on  $X_1 = 1$ )

- Appears in: 1 sample with  $X_2 = 0$ , 1 sample with  $X_2 = 1$
- MLE:  $\theta_2^*(1) = 1/2 \checkmark$

For  $\theta_2(0)$ : (conditional on  $X_1 = 0$ )

- Appears in: 1 sample with  $X_2 = 1$
- MLE:  $\theta_2^*(0) = 1 \checkmark$

And so on for  $\theta_3\dots$

**General Pattern:** Count frequencies in the data!

## 6.14 Why MLE Works: Statistical Justification

### Property 1: Consistency

As  $m \rightarrow \infty$  (infinite data):

$$\theta_{\text{MLE}}^* \rightarrow \theta_{\text{true}} \quad (\text{if } P_{\text{data}} \in M)$$

**Meaning:** With enough data, MLE converges to the true parameters!

### Property 2: Asymptotic Efficiency

MLE achieves the **Cramér-Rao lower bound**:

- Among all unbiased estimators, MLE has the **lowest variance** asymptotically
- **Most efficient** use of data

### Property 3: Invariance

If  $\theta^*$  is MLE for  $\theta$ , then for any function  $g$ :

$$g(\theta^*) \text{ is MLE for } g(\theta)$$

**Example:** If  $\mu^*$  is MLE for mean, then  $(\mu^*)^2$  is MLE for  $\mu^2$ .

## 6.15 Connection: Empirical Risk Minimization

**Viewing MLE as Risk Minimization:**

**Define loss function:**  $\mathcal{L}(x; \theta) = -\log P_\theta(x)$

**Empirical risk:**

$$R_{\text{emp}}(\theta) = \frac{1}{m} \sum_{j=1}^m \mathcal{L}(x^{(j)}; \theta) = -\frac{1}{m} \sum_{j=1}^m \log P_\theta(x^{(j)})$$

**MLE = Empirical Risk Minimization:**

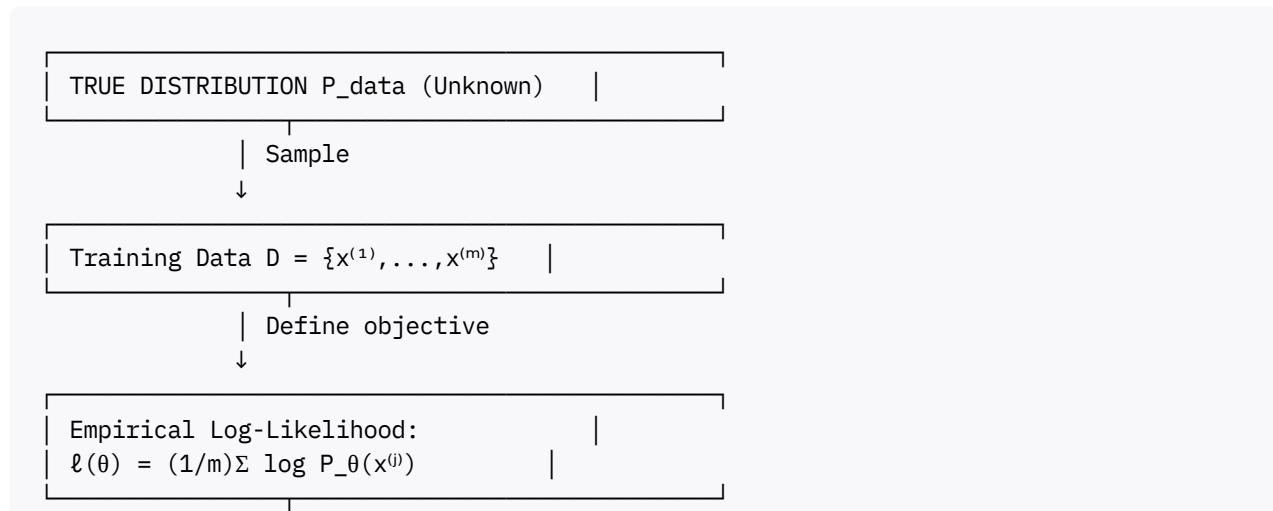
$$\arg \max_{\theta} \frac{1}{m} \sum_{j=1}^m \log P_\theta(x^{(j)}) = \arg \min_{\theta} R_{\text{emp}}(\theta)$$

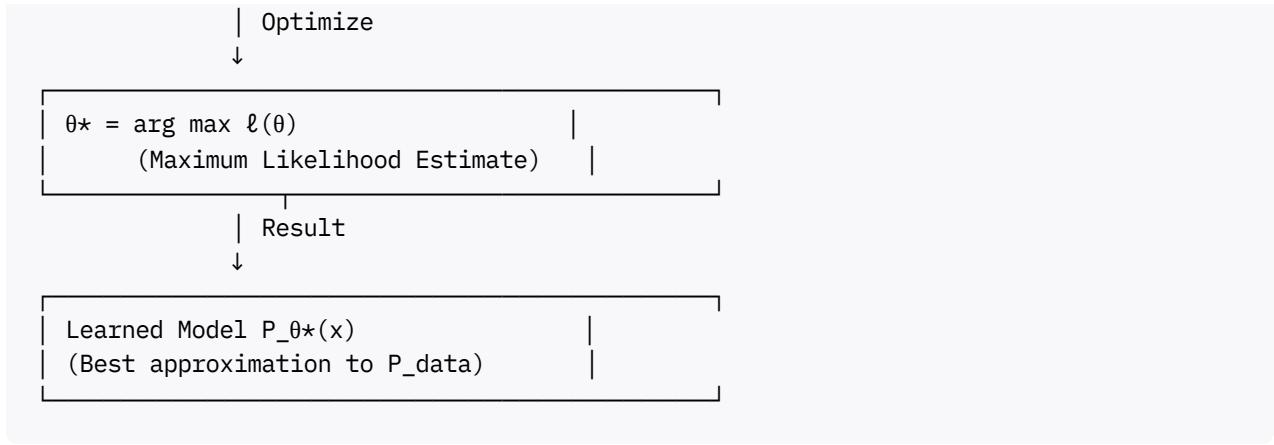
**Interpretation:** MLE minimizes average "surprise" (negative log-probability) on training data.

## 6.16 Comparison Table: Expected vs Empirical

| Aspect                  | Expected Log-Likelihood                                  | Empirical Log-Likelihood                          |
|-------------------------|----------------------------------------------------------|---------------------------------------------------|
| <b>Definition</b>       | $\mathbb{E}_{x \sim P_{\text{data}}} [\log P_\theta(x)]$ | $\frac{1}{m} \sum_{j=1}^m \log P_\theta(x^{(j)})$ |
| <b>Data requirement</b> | True distribution $P_{\text{data}}$                      | Samples $D$ from $P_{\text{data}}$                |
| <b>Computability</b>    | Cannot compute directly ✗                                | Can compute ✓                                     |
| <b>Sample size</b>      | Infinite (expectation)                                   | Finite ( $m$ samples)                             |
| <b>Relation</b>         | Target we want to maximize                               | Approximation we actually maximize                |
| <b>Convergence</b>      | —                                                        | → Expected as $m \rightarrow \infty$              |
| <b>Also called</b>      | Population log-likelihood                                | Sample log-likelihood, Training log-likelihood    |

## 6.17 Visual Flowchart: Complete Learning Pipeline





## ✓ KEY TAKEAWAYS - TOPIC 6

### Main Concepts:

1. **The Problem:** Expected log-likelihood involves unknown  $P_{data}$

2. **The Solution:** Approximate using training data (Monte Carlo)

$$\mathbb{E}_{x \sim P_{data}} [\log P_\theta(x)] \approx \frac{1}{m} \sum_{j=1}^m \log P_\theta(x^{(j)})$$

3. **Maximum Likelihood Estimation:**

$$\theta^* = \arg \max_{\theta} \frac{1}{m} \sum_{j=1}^m \log P_\theta(x^{(j)})$$

4. **Interpretation:** Find parameters that make training data **most likely**

5. **Coin Example:** For simple models, MLE gives closed-form solutions

- $\theta^* = 3/5 = 0.6$  (proportion of heads)

6. **Complex Models:** For neural networks, need gradient-based optimization (next topic!)

### The Complete Chain:

```

Minimize KL-divergence D(P_data || P_θ)
↓
Maximize Expected log-likelihood E[log P_θ(x)]
↓
Approximate with samples: (1/m)Σ log P_θ(x⁽⁰⁾)
↓
MAXIMUM LIKELIHOOD ESTIMATION ✓

```

## Why MLE?

- ✓ Theoretically justified (minimizes KL-divergence)
- ✓ Computable (uses training data)
- ✓ Statistically efficient (asymptotically optimal)
- ✓ Simple and elegant principle

## Connection to Next Topic

We now have our optimization objective:

$$\max_{\theta} \frac{1}{m} \sum_{j=1}^m \log P_{\theta}(x^{(j)})$$

But how do we actually SOLVE this optimization problem?

For complex models (neural networks with millions of parameters):

- ✗ No closed-form solution
- ✗ Non-convex optimization
- ✗ High-dimensional parameter space

The answer: Gradient-based optimization!

This leads us to **TOPIC 7: GRADIENT DESCENT & STOCHASTIC GRADIENT DESCENT** ☰

Where we'll learn:

- How to optimize using gradients
- Gradient descent algorithm
- Stochastic gradient descent (SGD) for large datasets
- Backpropagation for computing gradients
- Practical considerations

Ready to continue to **TOPIC 7: GRADIENT DESCENT?** Just say "next"! ☰

## ☐ **TOPIC 7: GRADIENT DESCENT & STOCHASTIC GRADIENT DESCENT**

## (Practical Optimization for Maximum Likelihood)

### 7.0 Big Picture: From Objective to Algorithm

#### Where We Are:

From Topic 6, we have our optimization objective:

$$\theta^* = \arg \max_{\theta} \frac{1}{m} \sum_{j=1}^m \log P_{\theta}(x^{(j)})$$

#### The Challenge:

- ✓ We know WHAT to optimize (MLE objective)
- ✓ We can evaluate the objective for any  $\theta$
- ✓ For simple models (coin flip), we can solve analytically

But for COMPLEX models:

- ✗ No closed-form solution
- ✗ Millions of parameters (neural networks)
- ✗ Non-convex optimization landscape

### The Solution: Gradient-Based Optimization

Idea: Move in the direction that INCREASES the objective

↓

Compute GRADIENT (direction of steepest ascent)

↓

Update parameters iteratively

↓

GRADIENT DESCENT (or ASCENT for maximization)!

#### Two Key Algorithms We'll Learn:

##### 1. Gradient Descent (GD)

- Use **all** training data in each iteration
- Accurate gradient
- Slow for large datasets

##### 2. Stochastic Gradient Descent (SGD)

- Use **one** or **mini-batch** of data in each iteration
- Approximate gradient
- Fast, scalable to huge datasets
- **Standard in deep learning!** ☺

## 7.1 The Optimization Problem (Formal Setup)

### Objective Function:

Define the **log-likelihood function**:

$$\ell(\theta) = \log L(\theta; D) = \sum_{j=1}^m \log P_\theta(x^{(j)})$$

Or with averaging (equivalent for optimization):

$$\ell(\theta) = \frac{1}{m} \sum_{j=1}^m \log P_\theta(x^{(j)})$$

### Optimization Goal:

$$\theta^* = \arg \max_{\theta \in \Theta} \ell(\theta)$$

### Where:

- $\theta \in \mathbb{R}^d$  (parameter vector with  $d$  dimensions)
- $\Theta$  is the feasible parameter space
- $\ell(\theta)$  is typically **non-convex** for neural networks

### Alternative: Minimization Form

In practice, we often **minimize negative log-likelihood** (NLL):

$$\mathcal{L}(\theta) = -\ell(\theta) = -\sum_{j=1}^m \log P_\theta(x^{(j)})$$

$$\theta^* = \arg \min_{\theta} \mathcal{L}(\theta)$$

This is the "loss function" commonly used in ML frameworks!

## 7.2 Gradient: The Key Mathematical Tool

### Definition: Gradient

The **gradient** of  $\ell(\theta)$  with respect to  $\theta$  is:

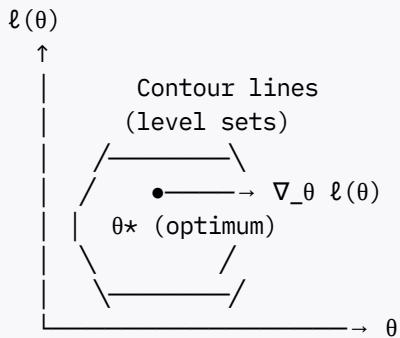
$$\nabla_{\theta} \ell(\theta) = \begin{bmatrix} \frac{\partial \ell}{\partial \theta_1} \\ \frac{\partial \ell}{\partial \theta_2} \\ \vdots \\ \frac{\partial \ell}{\partial \theta_d} \end{bmatrix}$$

**In words:** Vector of partial derivatives with respect to each parameter.

### Geometric Interpretation:

$\nabla_{\theta} \ell(\theta)$  points in the direction of STEEPEST ASCENT  
 ↓  
 Moving in this direction INCREASES  $\ell(\theta)$  most rapidly  
 ↓  
 For MAXIMIZATION: Move along  $+\nabla_{\theta} \ell(\theta)$   
 For MINIMIZATION: Move along  $-\nabla_{\theta} \ell(\theta)$  (negative gradient)

### Visual Representation: Gradient in 2D



The gradient points TOWARD the maximum (perpendicular to contour lines)

### Properties of Gradient:

1. **Direction:** Points toward steepest increase
2. **Magnitude:** Rate of increase
  - Large  $\|\nabla_{\theta} \ell(\theta)\| \rightarrow$  Steep
  - Small  $\|\nabla_{\theta} \ell(\theta)\| \rightarrow$  Flat
3. **At optimum:**  $\nabla_{\theta} \ell(\theta^*) = 0$  (critical point)

## 7.3 GRADIENT DESCENT ALGORITHM

### Basic Idea:

Start with initial parameters and iteratively move in the direction of the gradient.

### Algorithm (for MAXIMIZATION):

$$\boxed{\theta^{(t+1)} = \theta^{(t)} + \alpha_t \nabla_{\theta} \ell(\theta^{(t)})}$$

For MINIMIZATION of  $\mathcal{L}(\theta) = -\ell(\theta)$ :

$$\boxed{\theta^{(t+1)} = \theta^{(t)} - \alpha_t \nabla_{\theta} \mathcal{L}(\theta^{(t)})}$$

### Components:

- $\theta^{(t)}$ : Parameters at iteration  $t$
- $\alpha_t$ : Learning rate (step size)
- $\nabla_{\theta} \ell(\theta^{(t)})$ : Gradient at current parameters

### Complete Algorithm: Gradient Descent

Algorithm: Gradient Descent for Maximum Likelihood

Input:

- Training data  $D = \{x^{(1)}, \dots, x^{(m)}\}$
- Learning rate schedule  $\{\alpha_t\}$
- Convergence tolerance  $\epsilon$

Output:

- Optimal parameters  $\theta^*$

Steps:

1. Initialize  $\theta^{(0)}$  randomly (e.g., from standard normal)

2. For  $t = 0, 1, 2, \dots$  until convergence:

a) Compute log-likelihood:

$$\ell(\theta^{(t)}) = \sum_{j=1}^m \log P_{\theta}(x^{(j)})$$

b) Compute gradient:

$$\nabla_{\theta} \ell(\theta^{(t)}) = \sum_{j=1}^m \nabla_{\theta} \log P_{\theta}(x^{(j)}) \Big|_{\{\theta=\theta^{(t)}\}}$$

(use backpropagation for neural networks)

c) Update parameters:

$$\theta^{(t+1)} = \theta^{(t)} + \alpha_t \nabla_{\theta} \ell(\theta^{(t)})$$

d) Check convergence:

If  $\|\nabla_{\theta} \ell(\theta^{(t)})\| < \epsilon$ :  
STOP (converged)

3. Return  $\theta^* = \theta^{(t)}$

## 7.4 Gradient Computation for MLE

### The Gradient Formula:

For maximum likelihood:

$$\ell(\theta) = \sum_{j=1}^m \log P_\theta(x^{(j)})$$

The gradient is:

$$\nabla_\theta \ell(\theta) = \sum_{j=1}^m \nabla_\theta \log P_\theta(x^{(j)})$$

**Key insight:** Gradient of sum = Sum of gradients (linearity!)

### For Autoregressive Models:

Recall (Lecture 3 + Slide 16):

$$P_\theta(x) = \prod_{i=1}^n p_{neural}(x_i | x_{<i}; \theta_i)$$

Taking log:

$$\log P_\theta(x) = \sum_{i=1}^n \log p_{neural}(x_i | x_{<i}; \theta_i)$$

Gradient:

$$\nabla_\theta \log P_\theta(x) = \sum_{i=1}^n \nabla_\theta \log p_{neural}(x_i | x_{<i}; \theta_i)$$

### Total Gradient (over all data):

$$\nabla_\theta \ell(\theta) = \sum_{j=1}^m \sum_{i=1}^n \nabla_\theta \log p_{neural}(x_i^{(j)} | x_{<i}^{(j)}; \theta_i)$$

This is computed via BACKPROPAGATION! □

## 7.5 Detailed Mathematical Example: Simple Case

Let's work through gradient descent for a **simple 2D example**.

**Setup:**

**Model:** Two parameters  $\theta = (\theta_1, \theta_2)$

**Objective:**

$$\ell(\theta_1, \theta_2) = -(\theta_1 - 2)^2 - (\theta_2 + 1)^2$$

(Negative quadratic, maximum at  $(2, -1)$ )

**Step 1: Compute Gradient**

$$\frac{\partial \ell}{\partial \theta_1} = -2(\theta_1 - 2)$$

$$\frac{\partial \ell}{\partial \theta_2} = -2(\theta_2 + 1)$$

$$\nabla_{\theta} \ell(\theta) = \begin{bmatrix} -2(\theta_1 - 2) \\ -2(\theta_2 + 1) \end{bmatrix}$$

**Step 2: Initialize**

$$\theta^{(0)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad \alpha = 0.1$$

**Step 3: Iteration 1**

**Compute gradient at  $\theta^{(0)}$ :**

$$\nabla_{\theta} \ell(\theta^{(0)}) = \begin{bmatrix} -2(0 - 2) \\ -2(0 + 1) \end{bmatrix} = \begin{bmatrix} 4 \\ -2 \end{bmatrix}$$

**Update:**

$$\begin{aligned} \theta^{(1)} &= \theta^{(0)} + \alpha \nabla_{\theta} \ell(\theta^{(0)}) \\ &= \begin{bmatrix} 0 \\ 0 \end{bmatrix} + 0.1 \begin{bmatrix} 4 \\ -2 \end{bmatrix} = \begin{bmatrix} 0.4 \\ -0.2 \end{bmatrix} \end{aligned}$$

**Step 4: Iteration 2**

**Gradient at  $\theta^{(1)}$ :**

$$\nabla_{\theta} \ell(\theta^{(1)}) = \begin{bmatrix} -2(0.4 - 2) \\ -2(-0.2 + 1) \end{bmatrix} = \begin{bmatrix} 3.2 \\ -1.6 \end{bmatrix}$$

**Update:**

$$\theta^{(2)} = \begin{bmatrix} 0.4 \\ -0.2 \end{bmatrix} + 0.1 \begin{bmatrix} 3.2 \\ -1.6 \end{bmatrix} = \begin{bmatrix} 0.72 \\ -0.36 \end{bmatrix}$$

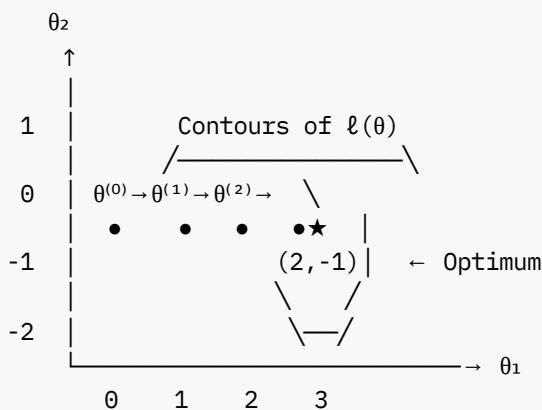
**Continuing...**

| Iteration | $\theta_1$ | $\theta_2$ | $\ell(\theta)$ |
|-----------|------------|------------|----------------|
| 0         | 0.00       | 0.00       | -5.00          |
| 1         | 0.40       | -0.20      | -3.20          |
| 2         | 0.72       | -0.36      | -2.05          |
| 5         | 1.34       | -0.67      | -0.55          |
| 10        | 1.79       | -0.89      | -0.09          |
| 20        | 1.98       | -0.99      | -0.001         |
| $\infty$  | 2.00 ✓     | -1.00 ✓    | 0.00 ✓         |

**Observation:** Converges to true optimum  $(2, -1)$ ! ✓

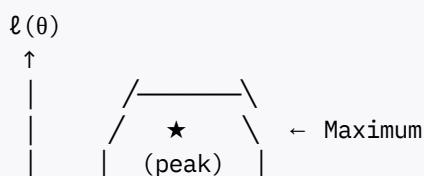
## 7.6 Visual Trajectory: Gradient Descent

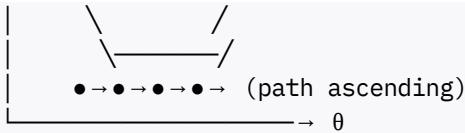
**2D Visualization:**



Path:  $\theta^{(0)} \rightarrow \theta^{(1)} \rightarrow \theta^{(2)} \rightarrow \dots \rightarrow \theta^*$   
 Following gradient direction at each step

**3D Surface View:**





Gradient descent climbs UP the surface toward the maximum

## 7.7 Learning Rate: Critical Hyperparameter

### What is Learning Rate $\alpha_t$ ?

The **step size** in each iteration:

$$\theta^{(t+1)} = \theta^{(t)} + \alpha_t \nabla_{\theta} \ell(\theta^{(t)})$$

#### Trade-off:

$\alpha$  too small:

- ✓ Stable, smooth convergence
- ✗ VERY SLOW (many iterations needed)

$\alpha$  too large:

- ✓ Fast initial progress
- ✗ UNSTABLE, may oscillate or diverge
- ✗ Can overshoot the optimum

$\alpha$  just right:

- ✓ Fast convergence
- ✓ Stable
- ✓ Reaches good optimum

## Visual: Effect of Learning Rate

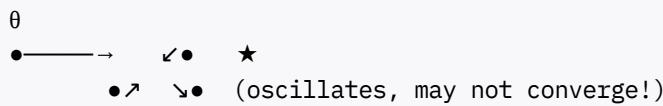
Learning Rate Too Small ( $\alpha = 0.01$ ):



Learning Rate Good ( $\alpha = 0.1$ ):



Learning Rate Too Large ( $\alpha = 1.0$ ):



## Common Learning Rate Schedules:

### 1. Constant:

$$\alpha_t = \alpha \quad \forall t$$

Simple, but may not converge to exact optimum.

### 2. Step Decay:

$$\alpha_t = \alpha_0 \cdot \gamma^{\lfloor t/k \rfloor}$$

Reduce by factor  $\gamma$  every  $k$  iterations.

### 3. Exponential Decay:

$$\alpha_t = \alpha_0 e^{-\lambda t}$$

### 4. 1/t Schedule:

$$\alpha_t = \frac{\alpha_0}{1 + \beta t}$$

Common in theory (guarantees convergence).

### 5. Adaptive (Adam, RMSprop):

Different learning rate for each parameter, adapted during training.

## 7.8 Properties of Gradient Descent

### Convergence Guarantees:

#### For CONVEX objectives:

- Gradient descent **converges to global optimum**
- Rate:  $O(1/\epsilon)$  iterations for  $\epsilon$ -accuracy

#### For NON-CONVEX objectives (our case!):

- Gradient descent converges to **local optimum** or **saddle point**
- No guarantee of global optimum
- **But works well in practice for neural networks!** ☺

### Why Non-Convex is OK for Deep Learning:

Neural networks are non-convex, BUT:

- ✓ Many local optima are "good enough"
- ✓ High-dimensional spaces have fewer bad local minima

- ✓ SGD with noise helps escape poor local optima
- ✓ Over-parameterization helps (more parameters than data)

## 7.9 The Computational Bottleneck

### Cost of Gradient Descent:

Per iteration:

$$\nabla_{\theta} \ell(\theta) = \sum_{j=1}^m \nabla_{\theta} \log P_{\theta}(x^{(j)})$$

Computational cost:

- Must compute gradient for **ALL  $m$  training examples**
- For large  $m$  (e.g.,  $m = 10^6$ ), this is **very expensive!**

### Example: ImageNet

Dataset:  $m = 1,200,000$  images  
 Model: ResNet (25 million parameters)

One gradient computation:

- Forward pass through ALL 1.2M images
- Backward pass through ALL 1.2M images
- Time: ~1 hour on GPU!

For 100 iterations: 100 hours = 4+ days!

Too slow! ✗

### The Problem:

```

Gradient Descent requires FULL dataset pass per iteration
 ↓
 TOO SLOW for large datasets!
 ↓
 Need a faster alternative!
 ↓
 STOCHASTIC GRADIENT DESCENT! ✓

```

## 7.10 STOCHASTIC GRADIENT DESCENT (SGD)

## The Key Idea:

Instead of using ALL data, use ONE (or a few) samples per iteration!

## Motivation: Monte Carlo Again!

Recall the gradient is an **expectation**:

$$\nabla_{\theta} \ell(\theta) = m \cdot \mathbb{E}_{x \sim P_{data}} [\nabla_{\theta} \log P_{\theta}(x)]$$

### Monte Carlo approximation:

Use **one random sample**  $x^{(j)}$  from  $D$ :

$$\nabla_{\theta} \ell(\theta) \approx m \cdot \nabla_{\theta} \log P_{\theta}(x^{(j)})$$

Or equivalently (without the  $m$  factor):

$$\nabla_{\theta} \ell(\theta) \approx \nabla_{\theta} \log P_{\theta}(x^{(j)})$$

This is a **NOISY** estimate of the gradient, but it's unbiased!

## 7.11 SGD Algorithm (Pure Stochastic Version)

### Algorithm: Stochastic Gradient Descent

Algorithm: SGD for Maximum Likelihood

Input:

- Training data  $D = \{x^{(1)}, \dots, x^{(m)}\}$
- Learning rate schedule  $\{\alpha_t\}$
- Number of epochs  $T$

Output:

- Optimized parameters  $\theta^*$

Steps:

1. Initialize  $\theta^{(0)}$  randomly
2. For epoch  $t = 1, 2, \dots, T$ :
  - a) Shuffle training data  $D$  (randomize order)
  - b) For each sample  $x^{(j)}$  in  $D$ :
    - i) Compute stochastic gradient:  
$$g = \nabla_{\theta} \log P_{\theta}(x^{(j)}) \mid_{\{\theta=\theta^{(t)}\}}$$
    - ii) Update parameters:  
$$\theta^{(t+1)} = \theta^{(t)} + \alpha_t \cdot g$$
    - iii)  $t = t + 1$

3. Return  $\theta^* = \theta^{(T)}$

### Mathematical Update Rule:

$$\theta^{(t+1)} = \theta^{(t)} + \alpha_t \nabla_{\theta} \log P_{\theta}(x^{(j)})$$

Where  $x^{(j)}$  is a **single randomly sampled** training example.

### 7.12 Mini-Batch SGD (Most Common in Practice)

#### The Best of Both Worlds:

Use a **small batch** of samples (e.g., 32, 64, 256) instead of just one.

#### Algorithm: Mini-Batch SGD

Algorithm: Mini-Batch SGD

Input:

- Training data  $D = \{x^{(1)}, \dots, x^{(m)}\}$
- Batch size  $B$  (e.g., 32, 64, 128, 256)
- Learning rate  $\alpha_t$
- Number of epochs  $T$

Steps:

1. Initialize  $\theta^{(0)}$  randomly
2. For epoch  $t = 1, 2, \dots, T$ :
  - a) Shuffle  $D$  randomly
  - b) Partition  $D$  into mini-batches:  
 $B_1, B_2, \dots, B_{\lfloor m/B \rfloor}$   
where  $|B_k| = B$
  - c) For each mini-batch  $B_k$ :
    - i) Compute mini-batch gradient:  
$$g = (1/B) \sum_{x^{(j)} \in B_k} \nabla_{\theta} \log P_{\theta}(x^{(j)})$$
    - ii) Update:  
$$\theta = \theta + \alpha_t \cdot g$$
3. Return  $\theta$

## Update Formula:

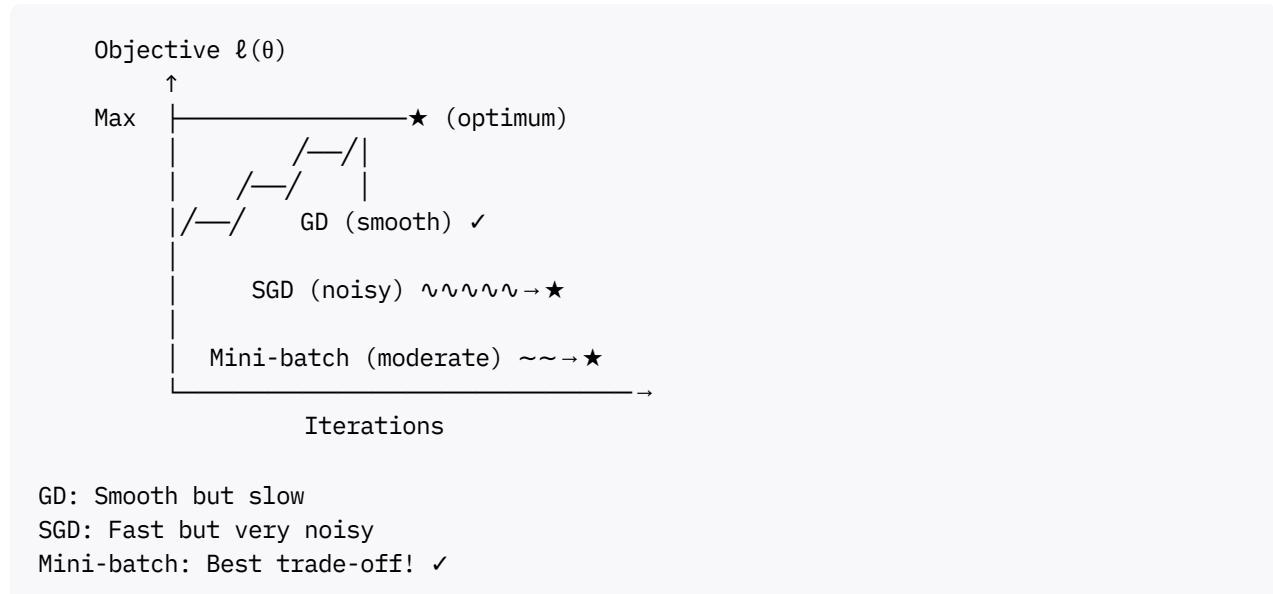
$$\theta^{(t+1)} = \theta^{(t)} + \alpha_t \cdot \frac{1}{B} \sum_{x^{(j)} \in \mathcal{B}_k} \nabla_{\theta} \log P_{\theta}(x^{(j)})$$

## 7.13 Comparison: GD vs SGD vs Mini-Batch SGD

### Detailed Comparison Table:

| Aspect                 | Gradient Descent (GD)                 | SGD (Single Sample)                               | Mini-Batch SGD               |
|------------------------|---------------------------------------|---------------------------------------------------|------------------------------|
| Samples per update     | All $m$                               | 1                                                 | $B$ (e.g., 32-256)           |
| Gradient               | Exact: $\nabla_{\theta} \ell(\theta)$ | Noisy: $\nabla_{\theta} \log P_{\theta}(x^{(j)})$ | Less noisy: Average over $B$ |
| Convergence            | Smooth, deterministic                 | Noisy, stochastic                                 | Moderate noise               |
| Speed per iteration    | SLOW (all data)                       | FAST (one sample)                                 | FAST (small batch)           |
| Iterations to converge | Fewer                                 | Many more                                         | Moderate                     |
| Memory usage           | High                                  | Low                                               | Moderate                     |
| Parallelization        | Full batch                            | None                                              | Yes (GPU efficient!)         |
| Typical use            | Small datasets                        | Rare                                              | <b>STANDARD ✓</b>            |

### Visual: Convergence Paths



## 7.14 Why SGD Works: Statistical Perspective

### Key Properties:

#### 1. Unbiased Gradient Estimate:

$$\mathbb{E}_{x^{(j)} \sim \text{uniform}(D)} [\nabla_{\theta} \log P_{\theta}(x^{(j)})] = \frac{1}{m} \sum_{j=1}^m \nabla_{\theta} \log P_{\theta}(x^{(j)})$$

**Meaning:** On average, the stochastic gradient equals the true gradient!

#### 2. Variance:

Single sample has high variance, but:

- Mini-batches reduce variance by factor of  $\sqrt{B}$
- Averaging helps stabilize

#### 3. Convergence:

With proper learning rate schedule ( $\sum_t \alpha_t = \infty$ ,  $\sum_t \alpha_t^2 < \infty$ ):

- SGD converges to stationary point
- Even for non-convex objectives!

## 7.15 Practical Example: Image Classification

Let's see how SGD works for a **realistic scenario**.

### Setup: MNIST Digit Classification

#### Dataset:

- $m = 60,000$  training images
- Each image:  $28 \times 28 = 784$  pixels
- 10 classes (digits 0-9)

**Model:** Autoregressive model (or could be classifier)

**Parameters:**  $d = 500,000$  (500K parameters)

### Gradient Descent (Full Batch):

Per iteration:

- Process all 60,000 images
- Compute 60,000 forward passes
- Compute 60,000 backward passes
- Average gradients
- Update parameters

Time per iteration: ~60 seconds  
Memory required:  $60,000 \times 784 \times 4$  bytes  $\approx 180$  MB  
Iterations needed: ~100

Total time:  $100 \times 60\text{s} = 6000\text{s} \approx 1.7$  hours

### Mini-Batch SGD (Batch Size = 64):

Per iteration:  
- Process 64 images  
- Compute 64 forward passes  
- Compute 64 backward passes  
- Average gradients  
- Update parameters

Time per iteration: ~0.05 seconds  
Memory required:  $64 \times 784 \times 4$  bytes  $\approx 200$  KB  
Iterations per epoch:  $60,000/64 \approx 937$   
Epochs needed: ~10

Total iterations:  $937 \times 10 = 9,370$   
Total time:  $9,370 \times 0.05\text{s} \approx 470\text{s} \approx 8$  minutes

Speed-up: 1.7 hours  $\rightarrow$  8 minutes! ✓  
12× FASTER!

### Why SGD Wins:

- Many more iterations, but **each iteration is WAY faster**
- Updates happen more frequently  $\rightarrow$  faster learning
- Can use GPUs efficiently with mini-batches

## 7.16 SGD for Autoregressive Models (From Lecture Slide 18)

### Recall from Slide 16:

For autoregressive models:

$$\ell(\theta) = \log L(\theta; D) = \sum_{j=1}^m \sum_{i=1}^n \log p_{neural}(x_i^{(j)} | x_{<i}^{(j)}; \theta_i)$$

### Gradient (Slide 18):

$$\nabla_{\theta_i} \ell(\theta) = \sum_{j=1}^m \nabla_{\theta_i} \sum_{i=1}^n \log p_{neural}(x_i^{(j)} | x_{<i}^{(j)}; \theta_i)$$

$$= \sum_{j=1}^m \nabla_{\theta_i} \log p_{neural}(x_i^{(j)} | x_{<i}^{(j)}; \theta_i)$$

### Key Observation (Slide 18):

If NO parameter sharing:

Each conditional  $p_{neural}(x_i | x_{<i}; \theta_i)$  can be optimized SEPARATELY!  
 ↓  
 Parallel optimization of each conditional

### In practice (e.g., NADE, PixelCNN):

Parameters ARE shared across conditionals  
 ↓  
 Must optimize jointly  
 ↓  
 Backpropagation computes gradients efficiently!

## 7.17 What if Dataset is HUGE? (Slide 19)

### The Challenge:

Dataset size:  $m = 100$  million images  
 ↓  
 Even ONE pass through data takes hours!  
 ↓  
 Cannot compute full gradient

### Solution: Monte Carlo Sampling (Slide 19)

Recall the gradient:

$$\nabla_{\theta} \ell(\theta) = \sum_{j=1}^m \nabla_{\theta} \log P_{\theta}(x^{(j)})$$

Can be written as:

$$\begin{aligned} \nabla_{\theta} \ell(\theta) &= m \cdot \frac{1}{m} \sum_{j=1}^m \nabla_{\theta} \log P_{\theta}(x^{(j)}) \\ &= m \cdot \mathbb{E}_{x^{(j)} \sim \text{uniform}(D)} [\nabla_{\theta} \log P_{\theta}(x^{(j)})] \end{aligned}$$

## Monte Carlo Approximation (Slide 19):

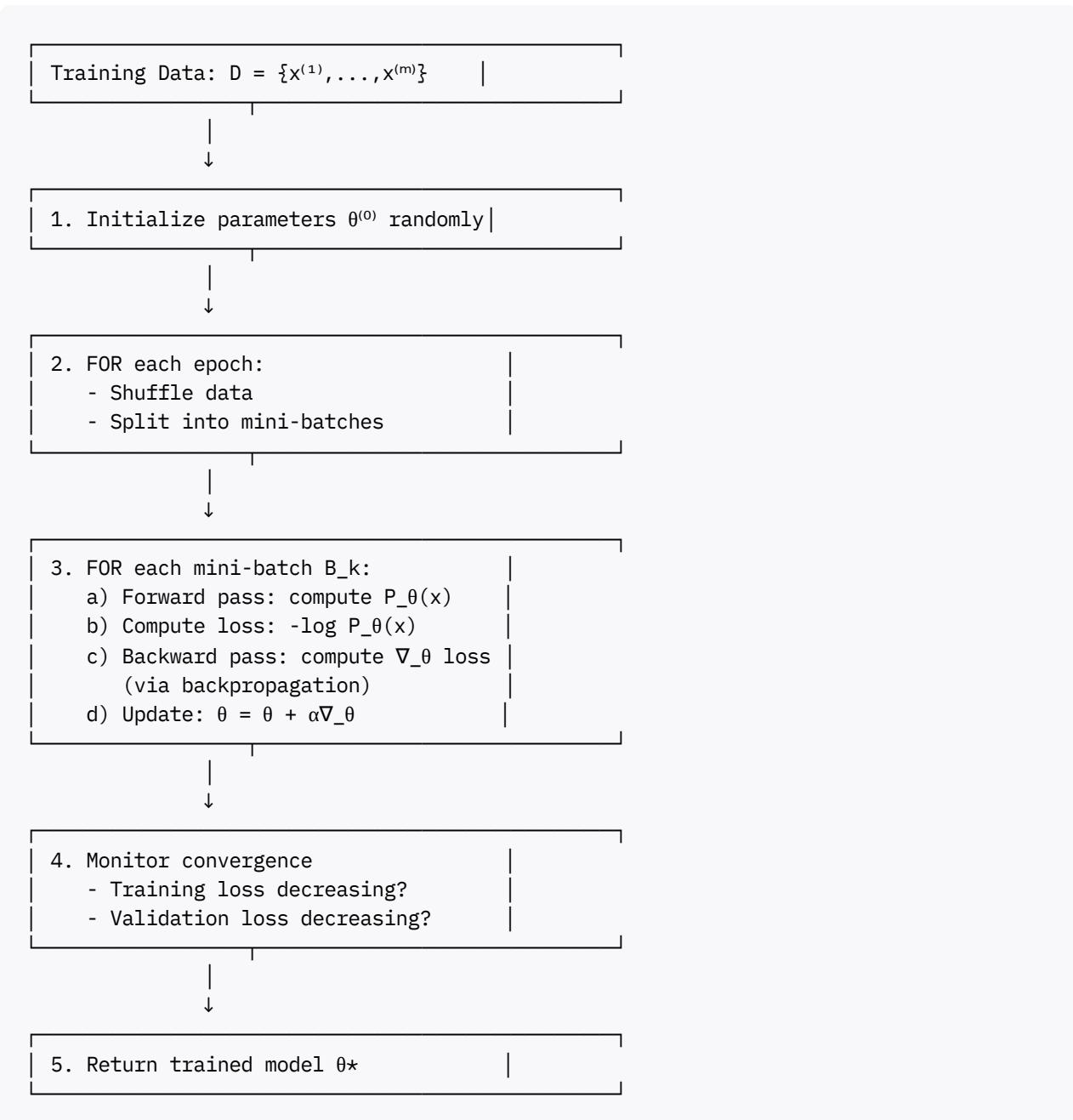
Sample  $x^{(j)} \sim D$  (one random training example):

$$\nabla_{\theta} \ell(\theta) \approx m \sum_{i=1}^n \nabla_{\theta} \log p_{neural}(x_i^{(j)} | x_{<i}^{(j)}; \theta_i)$$

This is the SGD UPDATE!

## 7.18 Complete Training Pipeline: Putting It All Together

### Flowchart: From Data to Trained Model



## 7.19 Backpropagation: Computing Gradients Efficiently

### The Challenge:

For neural networks:

```
Thousands to billions of parameters!
↓
Need gradient with respect to EACH parameter
↓
Naïve approach: Compute $\partial \ell / \partial \theta_i$ individually → Too slow!
```

### The Solution: Backpropagation

**Key Idea:** Use **chain rule** to compute all gradients in **one backward pass**.

#### From Slide 17:

```
"Compute $\nabla_{\theta} \ell(\theta)$ (by back propagation)"
```

### Backpropagation Algorithm (High-Level):

1. **Forward pass:** Compute outputs and intermediate activations
2. **Compute loss:**  $\mathcal{L} = -\log P_{\theta}(x)$
3. **Backward pass:** Propagate gradient backward through network using chain rule
4. **Result:** Gradients  $\nabla_{\theta} \mathcal{L}$  for all parameters

### Efficiency:

- Time:  $O(E)$  where  $E$  = number of edges in computation graph
- **Only ~2× cost of forward pass!**
- Modern frameworks (PyTorch, TensorFlow) do this automatically!

## 7.20 Practical Considerations & Tips

### 1. Initialization:

```
 $\theta^{(0)}$ should be random (break symmetry)
```

Good initializations:

- ✓ Xavier/Glorot:  $\theta \sim N(0, 1/\sqrt{n_{in}})$
- ✓ He initialization:  $\theta \sim N(0, 2/\sqrt{n_{in}})$

Bad initialization:

- ✗ All zeros → No learning!
- ✗ Too large → Exploding gradients

## 2. Learning Rate Schedule:

Start large, decay over time

Common schedule:

$$\alpha_t = \alpha_0 / (1 + \text{decay\_rate} \times \text{epoch})$$

Or step decay:

$$\alpha_t = \alpha_0 \times 0.5^{(\text{epoch} // 10)}$$

## 3. Batch Size:

Typical values: 32, 64, 128, 256

Trade-offs:

Small batch (32):

- ✓ More noise → better generalization
- ✓ More frequent updates
- ✗ Less GPU efficiency

Large batch (256):

- ✓ More stable gradients
- ✓ Better GPU utilization
- ✗ May overfit
- ✗ Fewer updates per epoch

## 4. Monitoring Training:

Track metrics:

- Training loss (should decrease)
- Validation loss (should decrease, then plateau)
- Gradient norms (should not explode or vanish)
- Learning rate (may need adjustment)

Stop when:

- Validation loss stops improving (early stopping)
- Gradient norms become very small
- Maximum epochs reached

## 7.21 Non-Convex Optimization (From Slide 17)

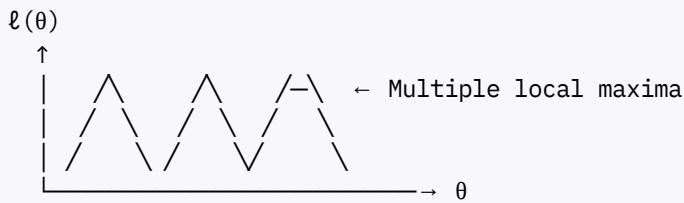
## The Reality:

From Slide 17:

"Non-convex optimization problem, but often works well in practice"

## Why Non-Convex is Challenging:

NON-CONVEX LANDSCAPE:



GD may get stuck in local maximum!  
No guarantee of finding global maximum!

## Why It Works in Practice:

### 1. High Dimensionality:

- In high dimensions, local minima are rare
- Most critical points are **saddle points** (can escape!)

### 2. Over-Parameterization:

- More parameters than data points
- Many global optima exist
- Many local optima are "equally good"

### 3. SGD Noise:

- Stochastic noise helps escape poor local optima
- Acts like implicit regularization

### 4. Careful Initialization:

- Good starting point helps

**Bottom Line:** Despite non-convexity, SGD finds good solutions for deep learning! ☐

## ✓ KEY TAKEAWAYS - TOPIC 7

### Main Concepts:

1. **Gradient Descent:** Iterative optimization using gradients

$$\theta^{(t+1)} = \theta^{(t)} + \alpha_t \nabla_{\theta} \ell(\theta^{(t)})$$

2. **Stochastic Gradient Descent:** Use mini-batches instead of full data

$$\theta^{(t+1)} = \theta^{(t)} + \alpha_t \cdot \frac{1}{B} \sum_{x \in \mathcal{B}} \nabla_{\theta} \log P_{\theta}(x)$$

3. **Why SGD?**

- **Much faster** per iteration
- **Scalable** to huge datasets
- **Works well** despite noise
- **Standard in deep learning** ✓

4. **Key Components:**

- Learning rate  $\alpha_t$ : Controls step size
- Batch size  $B$ : Trade-off between speed and stability
- Backpropagation: Efficient gradient computation

5. **Challenges:**

- Non-convex optimization (no global optimum guarantee)
- Hyperparameter tuning (learning rate, batch size)
- Works well in practice! ☺

### Comparison:

| Method         | Speed     | Accuracy | Use Case       |
|----------------|-----------|----------|----------------|
| GD             | Slow      | High     | Small datasets |
| SGD            | Very fast | Good     | Large datasets |
| Mini-batch SGD | Fast      | Good     | Standard ✓     |

### Complete Learning Pipeline:

```
Data D → Initialize θ(0) → Shuffle & batch
↓
Forward pass (compute Pθ(x))
↓
Compute loss (-log Pθ(x))
↓
Backpropagation (∇θ loss)
↓
Update θ (SGD step)
```

```
↓
Repeat until convergence
↓
Trained model θ^* ✓
```

## 🔗 Connection to Next Topic

We now have a complete learning algorithm:

- ✓ Objective: Maximum likelihood (Topic 6)
- ✓ Optimization: SGD with backpropagation (Topic 7)
- ✓ Can train complex models on large datasets!

But there's a critical problem we haven't addressed:

- ✗ What if our model OVERFITS the training data?
- ✗ What if it memorizes instead of generalizing?
- ✗ How do we ensure good performance on NEW data?

This leads us to **TOPIC 8: EMPIRICAL RISK AND OVERFITTING** ↗

Where we'll learn:

- What is overfitting and why it happens
- Empirical risk vs. true risk
- Generalization to unseen data
- How to detect overfitting

Ready to continue to TOPIC 8: EMPIRICAL RISK AND OVERFITTING? Just say "next"! ↗

# 🔗 **TOPIC 8: EMPIRICAL RISK AND OVERFITTING**

(The Generalization Challenge)

## 8.0 Big Picture: The Hidden Danger in Learning

Where We Are:

From Topics 1-7, we have a complete learning pipeline:

- ✓ Goal: Learn  $p(x)$  from data (Topic 1-3)
- ✓ Objective: Maximize log-likelihood (Topics 4-6)

- ✓ Algorithm: SGD with backpropagation (Topic 7)
- ✓ Can train models successfully!

**But we've been measuring success ONLY on training data...**

### The Critical Question:

```

Our model performs GREAT on training data D!
 ↓
 But what about NEW data?
 ↓
 Will it generalize to unseen examples?
 ↓
 Or did it just MEMORIZE the training set?

```

### The Central Problem: Overfitting

#### From Slide 20:

```

"Empirical risk minimization can easily OVERFIT the data"
"Extreme example: The data is the model (memorize all training data)"

```

This topic addresses:

1. What is overfitting?
2. Why does it happen?
3. How to detect it?
4. What can we do about it? (Next topic!)

### 8.1 Empirical Risk vs True Risk

#### Two Different Objectives:

##### 1. EMPIRICAL RISK (What we actually minimize):

$$R_{\text{emp}}(\theta) = -\frac{1}{m} \sum_{j=1}^m \log P_\theta(x^{(j)})$$

**In words:** Average negative log-likelihood on **training data  $D$** .

#### Alternative names:

- Training loss
- Training error
- Empirical loss

## 2. TRUE RISK (What we actually care about):

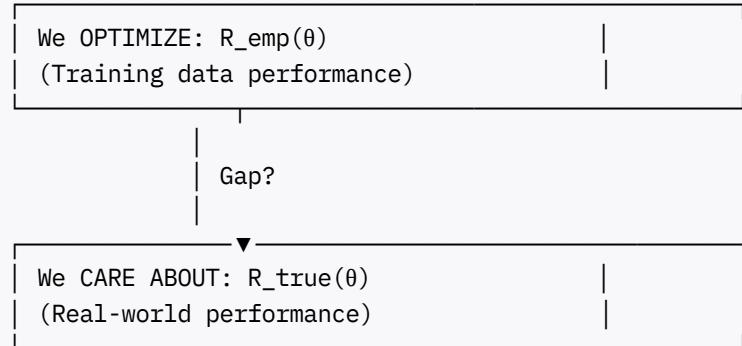
$$R_{\text{true}}(\theta) = -\mathbb{E}_{x \sim P_{\text{data}}} [\log P_{\theta}(x)]$$

In words: Expected negative log-likelihood over **entire data distribution**  $P_{\text{data}}$ .

### Alternative names:

- Test loss
- Generalization error
- Population risk

## The Fundamental Gap:



The GAP between these is the generalization gap!

## Mathematical Relationship:

$$\text{Generalization Gap} = R_{\text{true}}(\theta) - R_{\text{emp}}(\theta)$$

### Goal of Learning:

- Minimize  $R_{\text{true}}(\theta)$  (true risk)
- We only have access to  $R_{\text{emp}}(\theta)$  (empirical risk)
- Hope that minimizing empirical risk also minimizes true risk!

## 8.2 Definition: Overfitting

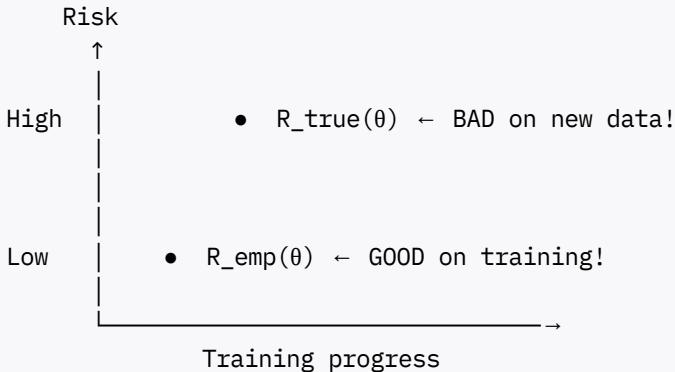
### Formal Definition:

**Overfitting occurs when:**

$$R_{\text{emp}}(\theta) \text{ is LOW, but } R_{\text{true}}(\theta) \text{ is HIGH}$$

In words: Model performs well on **training data** but poorly on **new data**.

## Visual Definition:



Large gap = OVERFITTING!

## Three Scenarios:

### SCENARIO 1: GOOD FIT (Desired!)

$R_{emp}(\theta) = 0.5$  (low)

$R_{true}(\theta) = 0.6$  (low)

Gap = 0.1 (small)

✓ Good generalization!

### SCENARIO 2: UNDERFITTING

$R_{emp}(\theta) = 2.5$  (high)

$R_{true}(\theta) = 2.7$  (high)

Gap = 0.2 (small)

✗ Model too simple, can't fit data

### SCENARIO 3: OVERFITTING

$R_{emp}(\theta) = 0.1$  (very low!)

$R_{true}(\theta) = 3.5$  (very high!)

Gap = 3.4 (HUGE!)

✗ Memorized training, fails on new data!

## 8.3 Extreme Example: Memorization (From Slide 20)

### The Most Extreme Overfitting:

#### From Slide 20:

"Extreme example: The data is the model (memorize all training data)"

## How It Works:

### Setup:

- Training data:  $D = \{x^{(1)}, x^{(2)}, \dots, x^{(m)}\}$

### "Model":

$$P_\theta(x) = \begin{cases} \frac{1}{m} & \text{if } x \in D \text{ (seen in training)} \\ 0 & \text{otherwise (never seen)} \end{cases}$$

### Empirical Risk (Training Performance):

$$R_{\text{emp}}(\theta) = -\frac{1}{m} \sum_{j=1}^m \log P_\theta(x^{(j)})$$

For each  $x^{(j)} \in D$ :

$$P_\theta(x^{(j)}) = \frac{1}{m}$$

$$R_{\text{emp}}(\theta) = -\frac{1}{m} \cdot m \cdot \log\left(\frac{1}{m}\right) = -\log\left(\frac{1}{m}\right) = \log(m)$$

**Training performance:** Constant (neither great nor terrible)

### True Risk (New Data Performance):

For most new data points  $x \notin D$ :

$$P_\theta(x) = 0$$

$$\log P_\theta(x) = -\infty$$

$$R_{\text{true}}(\theta) = +\infty$$

**Test performance:** INFINITE LOSS! X

### Why This is Bad:

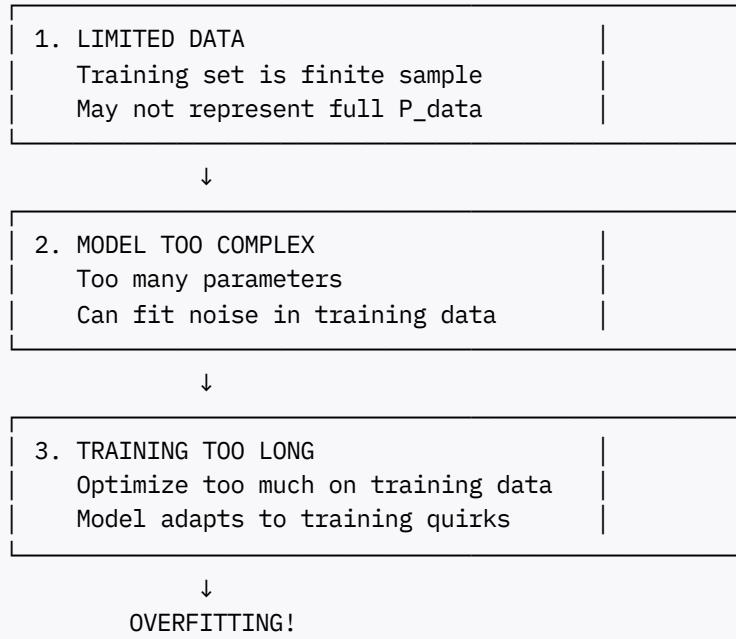
Training: Assigns uniform probability to training examples  
→ Reasonable training loss

Testing: Assigns ZERO probability to new examples  
→ INFINITE test loss!  
→ Model is USELESS for anything new!

This is the WORST possible overfitting!

## 8.4 Why Overfitting Happens

### Three Main Causes:



Let's explore each in detail:

## 8.5 Cause 1: Limited Data

### The Problem:

#### From Slide 4 (earlier in lecture):

"Example: Images represented with a vector  $X$  of 784 binary variables  $2^{784}$  possible images/states. Even 10 million examples is extremely sparse coverage."

### The Math:

For MNIST ( $28 \times 28$  binary images):

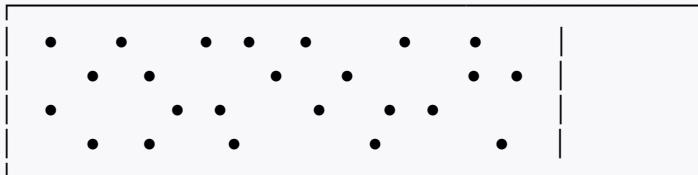
- **Possible states:**  $2^{784} \approx 10^{236}$
- **Training examples:**  $m = 60,000$
- **Coverage:**  $\frac{60,000}{10^{236}} \approx 0$  (essentially zero!)

## Consequence:

Training data D is a TINY sample of all possible data  
↓  
Model learns patterns specific to D  
↓  
These patterns may not hold for new data  
↓  
Overfitting!

## Visual Analogy:

TRUE DISTRIBUTION  $P_{\text{data}}$  (All possible images):



Billions of possible images

TRAINING DATA D (What we see):



Only 60,000 samples!

Model trained on D might miss important patterns that exist in  $P_{\text{data}}$  but weren't in D!

## Mathematical Example:

### True Distribution:

$$P_{\text{data}}(\text{dogs with spots}) = 0.3 \quad (\text{30\% of dogs have spots})$$

### Training Data ( $m=100$ ):

By chance, only 10 spotted dogs in training set  
Empirical:  $10/100 = 0.1 \quad (10\%)$

### Model learns:

$$P_{\theta}(\text{spots}) = 0.1 \quad (\text{matches training})$$

**On new data:**

Encounters spotted dogs 30% of time  
But assigns low probability!  
→ Poor generalization!

## 8.6 Cause 2: Model Too Complex (High Capacity)

**The Problem:**

Models with **too many parameters** can fit **any pattern**, including noise!

**Parameter Count vs Data Size:**

UNDERFITTING (Model too simple):  
Parameters:  $d = 10$   
Training examples:  $m = 10,000$   
Ratio:  $d/m = 0.001$   
→ Model can't capture complexity

GOOD FIT:  
Parameters:  $d = 1,000$   
Training examples:  $m = 10,000$   
Ratio:  $d/m = 0.1$   
→ Balanced

OVERFITTING (Model too complex):  
Parameters:  $d = 100,000$   
Training examples:  $m = 10,000$   
Ratio:  $d/m = 10$   
→ Model can memorize!

## Detailed Example: Polynomial Fitting

Let's see overfitting in a **simple regression setting**.

**Setup:**

**True function:**  $y = \sin(2\pi x) + \epsilon$  where  $\epsilon \sim \mathcal{N}(0, 0.1)$

**Training data:** 10 points sampled from true function

**Model family:** Polynomials of degree  $d$

$$P_{\theta}(y|x) = \mathcal{N}(y; \sum_{i=0}^d \theta_i x^i, \sigma^2)$$

## Three Models:

### Model 1: Linear (d=1)

$$\hat{y} = \theta_0 + \theta_1 x$$

- Parameters: 2
- Training error: 0.45
- Test error: 0.47
- **Diagnosis: UNDERFITTING** (model too simple)

### Model 2: Polynomial (d=4)

$$\hat{y} = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$$

- Parameters: 5
- Training error: 0.08
- Test error: 0.10
- **Diagnosis: GOOD FIT ✓**

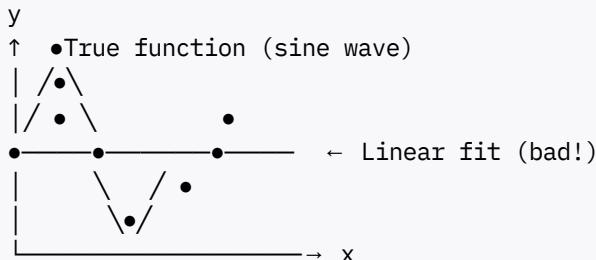
### Model 3: High-degree Polynomial (d=15)

$$\hat{y} = \sum_{i=0}^{15} \theta_i x^i$$

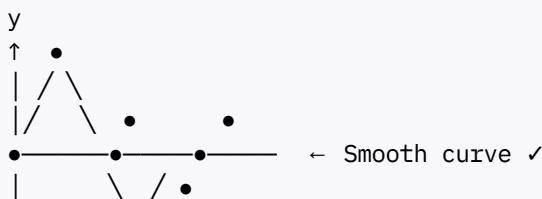
- Parameters: 16 (more than training points!)
- Training error: 0.001 (almost perfect!)
- Test error: 2.35 (terrible!)
- **Diagnosis: OVERFITTING ✗**

## Visual: Polynomial Fitting

Model 1 (Linear): UNDERFITTING

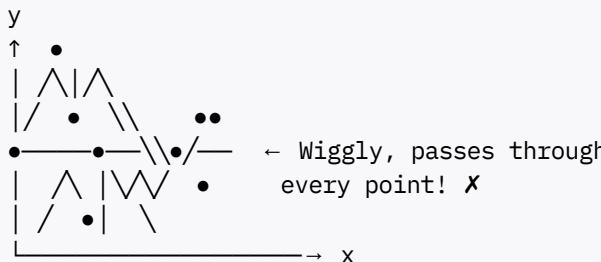


Model 2 (d=4): GOOD FIT





Model 3 ( $d=15$ ): OVERFITTING



Fits training perfectly but crazy between points!

## Why High-Degree Polynomial Overfits:

```

15 parameters for 10 data points
↓
Can fit training data EXACTLY
↓
Learns NOISE in training data
↓
Wildly oscillates between points
↓
Poor on new data!

```

## 8.7 Cause 3: Training Too Long

### The Problem:

Even with reasonable model capacity, **optimizing too long** can cause overfitting.

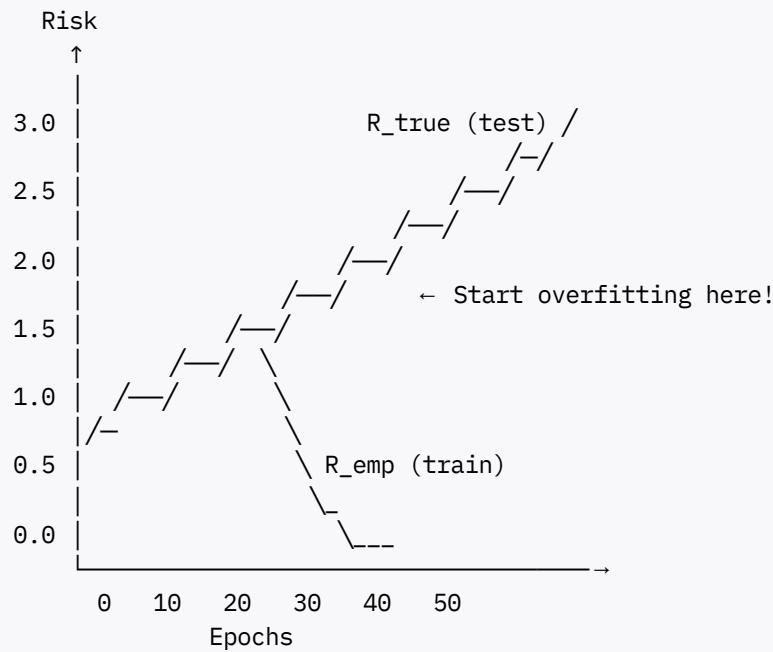
### Training Dynamics:

Early in training:  
 Model learns GENERAL patterns  
 Both  $R_{\text{emp}}$  and  $R_{\text{true}}$  decrease  
 ✓ Good generalization

Later in training:  
 Model starts fitting TRAINING-SPECIFIC quirks  
 $R_{\text{emp}}$  continues decreasing  
 $R_{\text{true}}$  starts INCREASING!  
 ✗ Overfitting begins

Very late in training:  
 Model memorizes training data  
 $R_{\text{emp}} \approx 0$  (perfect training)  
 $R_{\text{true}} \gg R_{\text{emp}}$  (terrible generalization)  
 ✗ Severe overfitting

## Visual: Training Curves



PHASES:

- Epochs 0-15: Good learning (both decrease)
- Epochs 15-25: Start overfitting (test increases)
- Epochs 25+: Severe overfitting (large gap)

Best model: Around epoch 15! (before overfitting starts)

## 8.8 Generalization: The Ultimate Goal

### Definition: Generalization

From Slide 20:

"Generalization: the data is a sample, usually there is vast amount of samples that you have never seen. Your model should generalize well to these 'never-seen' samples."

### Formal Definition:

A model **generalizes well** if:

$$[R_{\text{true}}(\theta) \approx R_{\text{emp}}(\theta)]$$

**In words:** Performance on **new data** is similar to performance on **training data**.

## What Makes a Model Generalize?

Good Generalization requires:

- ✓ Sufficient training data (large  $m$ )
- ✓ Model complexity matches problem (not too simple/complex)
- ✓ Training stopped at right time (not too early/late)
- ✓ Regularization (next topic!)
- ✓ Data is representative of true distribution

## The Generalization-Memorization Spectrum:

PERFECT GENERALIZATION



Learns TRUE  
patterns in P\_data

$R_{true} \approx R_{emp}$

Works on new data ✓

Simple model

Example:  
Linear regression  
with 2 parameters

PERFECT MEMORIZATION



Memorizes TRAINING  
data specifics

$R_{true} >> R_{emp}$

Fails on new data ✗

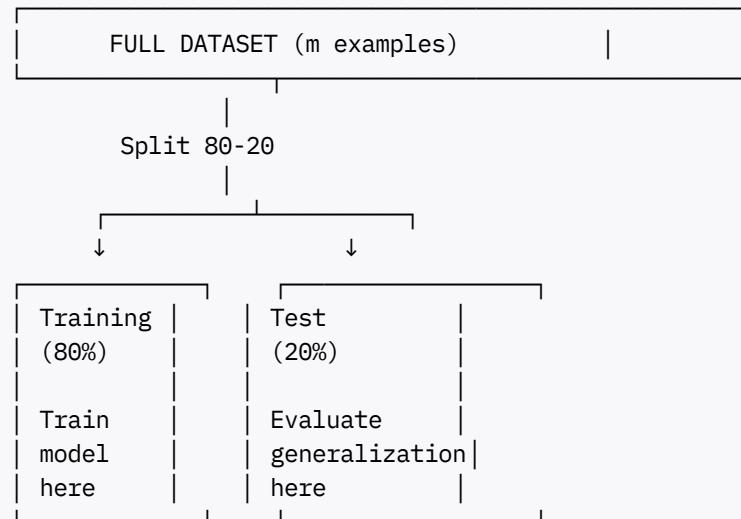
Complex model

Example:  
Lookup table  
(memorize dataset)

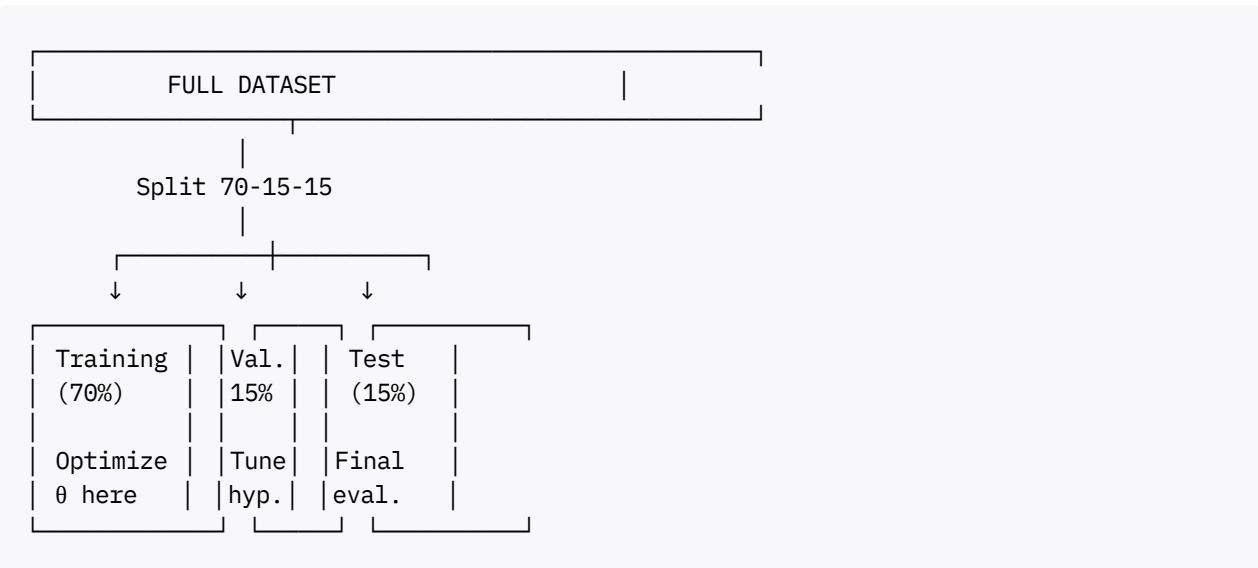
## 8.9 Detecting Overfitting: Train-Test Split

### The Standard Approach:

Split data into subsets:



## Three-Way Split (More Common):



### Roles:

#### Training Set:

- Used to **optimize parameters  $\theta$**
- Compute  $R_{\text{emp}}(\theta)$
- SGD updates use this data

#### Validation Set:

- Used to **tune hyperparameters** (learning rate, model size, etc.)
- Detect overfitting during training
- **Early stopping:** Stop when validation loss increases

#### Test Set:

- Used **ONLY ONCE** for **final evaluation**
- Estimates  $R_{\text{true}}(\theta)$
- Should NEVER be used during training!
- Gives unbiased estimate of generalization

## Why Three Sets?

Two sets only (Train + Test):

Problem: Might overfit to test set during hyperparameter tuning!

Three sets (Train + Val + Test):

- ✓ Train on training set
- ✓ Tune on validation set (can look many times)
- ✓ Final evaluation on test set (look only once)
- ✓ Test set gives unbiased generalization estimate!

## 8.10 Detailed Example: MNIST Overfitting

Let's see overfitting on a **real dataset**.

### Setup:

**Dataset:** MNIST handwritten digits

- Training: 60,000 images
- Test: 10,000 images

**Models:** Autoregressive with varying complexity

### Model A: Simple (100K parameters)

```
Training after 50 epochs:
R_emp = 0.15 (training NLL)
R_true = 0.18 (test NLL)
Gap = 0.03
```

Diagnosis: SLIGHT UNDERFITTING  
- Model too simple  
- Could learn more  
- Gap is small (good generalization)

### Model B: Medium (1M parameters)

```
Training after 50 epochs:
R_emp = 0.05 (training NLL)
R_true = 0.08 (test NLL)
Gap = 0.03
```

Diagnosis: GOOD FIT ✓  
- Low training error  
- Low test error  
- Small gap (good generalization)  
- This is the sweet spot!

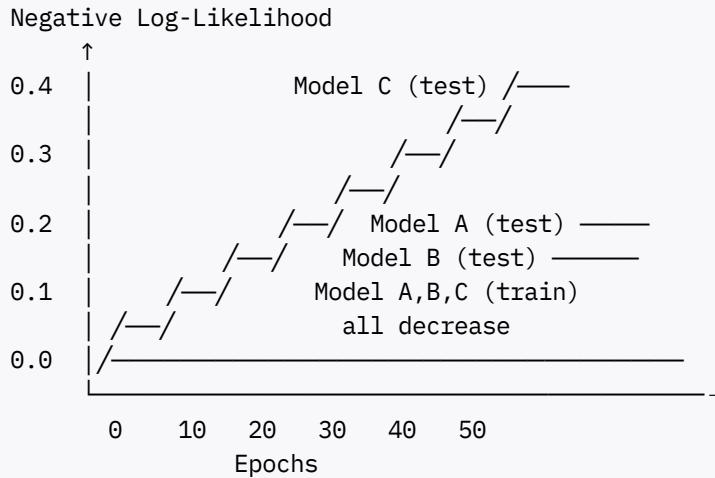
### Model C: Large (10M parameters)

```
Training after 50 epochs:
R_emp = 0.001 (training NLL - almost perfect!)
R_true = 0.35 (test NLL - poor!)
Gap = 0.349
```

Diagnosis: SEVERE OVERFITTING ✗  
- Extremely low training error  
- High test error

- HUGE gap (poor generalization)
- Model memorized training data!

## Training Curves:



### Observation:

- All models fit training well
- Only Model B generalizes well
- Model C overfits severely (test diverges)

## 8.11 Mathematical Analysis: Generalization Bound

### Statistical Learning Theory:

Under certain conditions, we can **bound** the generalization gap:

$$R_{\text{true}}(\theta) \leq R_{\text{emp}}(\theta) + \sqrt{\frac{d \log(m) + \log(1/\delta)}{m}}$$

(Simplified version of VC-dimension bound)

### Where:

- $d$ : Model complexity (e.g., number of parameters)
- $m$ : Number of training examples
- $\delta$ : Confidence level

### Implications:

#### 1. More Data Helps:

$$\text{Gap} \propto \frac{1}{\sqrt{m}}$$

Larger  $m \rightarrow$  Smaller gap  $\rightarrow$  Better generalization ✓

## 2. Simpler Models Help:

$$\text{Gap} \propto \sqrt{d}$$

Larger  $d \rightarrow$  Larger gap  $\rightarrow$  Worse generalization ✗

## 3. Trade-off:

Small  $m$ , Large  $d \rightarrow$  Large gap  $\rightarrow$  OVERFITTING

Large  $m$ , Small  $d \rightarrow$  Small gap  $\rightarrow$  UNDERFITTING (but good generalization)

Large  $m$ , Medium  $d \rightarrow$  Small gap + good fit  $\rightarrow$  BEST! ✓

## 8.12 Real-World Example: Image Classification

### Scenario: Cat vs Dog Classifier

#### Training Data:

- 1,000 cat images
- 1,000 dog images
- Total: 2,000 images

**Model:** Deep neural network with 50 million parameters

#### What Happens:

EPOCH 1:

Training accuracy: 55% (random guess level)

Test accuracy: 52%

Comment: Model learning basics

EPOCH 10:

Training accuracy: 85%

Test accuracy: 82%

Comment: Good learning, small gap

EPOCH 50:

Training accuracy: 98%

Test accuracy: 88%

Comment: Starting to overfit (gap growing)

EPOCH 200:

Training accuracy: 99.9%

Test accuracy: 75%

Comment: SEVERE OVERFITTING!

Model has MEMORIZED training data:

- Recognizes specific training images

- But fails on new images
- Learned "this exact pixel pattern = cat"  
instead of "cats have whiskers, ears, etc."

## What the Model Learned (Overfit Version):

GOOD GENERALIZATION (Desired):

- "Cats have pointy ears"
- "Cats have whiskers"
- "Cats have vertical pupils"
- Works on ANY cat image ✓

BAD OVERTFITTING (What happened):

- "Training image #47 has pixel (23,156) = 0.78 → cat"
- "Training image #193 has blue tint in corner → dog"
- Only works on training images ✗

## 8.13 Why We Care: Real-World Consequences

### Scenario 1: Medical Diagnosis

Model trained on 10,000 X-rays from Hospital A

Training accuracy: 99%

Deployed at Hospital B:

Test accuracy: 65%

Problem: Overfit to Hospital A's imaging equipment!

- Different scanners at Hospital B
- Model memorized Hospital A artifacts
- DANGEROUS for patients! ✗

### Scenario 2: Self-Driving Cars

Model trained on California roads

Training: Drives perfectly in CA

Deployed in snowy Michigan:

Test: Fails catastrophically

Problem: Never saw snow in training!

- Overfit to California conditions
- Can't generalize to weather/road variations
- SAFETY HAZARD! ✗

### Scenario 3: Fraud Detection

Model trained on 2023 fraud patterns  
Training: Catches 95% of 2023 fraud

Used in 2024:  
Test: Catches only 60% of fraud

Problem: Fraudsters adapted!  
- Model memorized 2023-specific patterns  
- New fraud techniques in 2024  
- Loses company millions! X

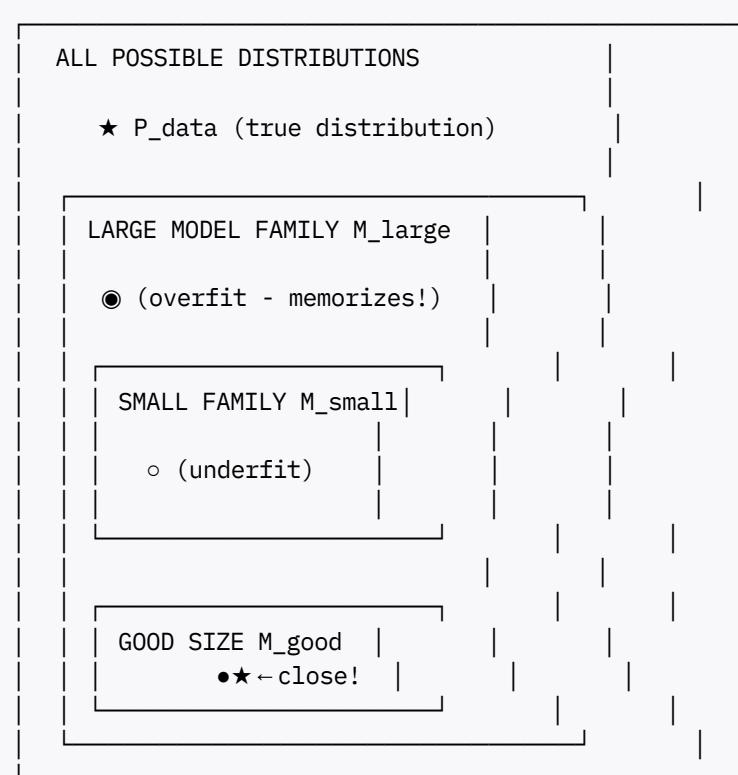
**The lesson:** Overfitting is not just a theoretical problem - it has **real, costly consequences!**

### 8.14 Hypothesis Space and Overfitting

Recall from Slide 20:

"Thus, we typically restrict the hypothesis space of distributions that we search over"

**The Trade-off:**



SMALL M: Can't reach  $P_{\text{data}}$  → Underfitting

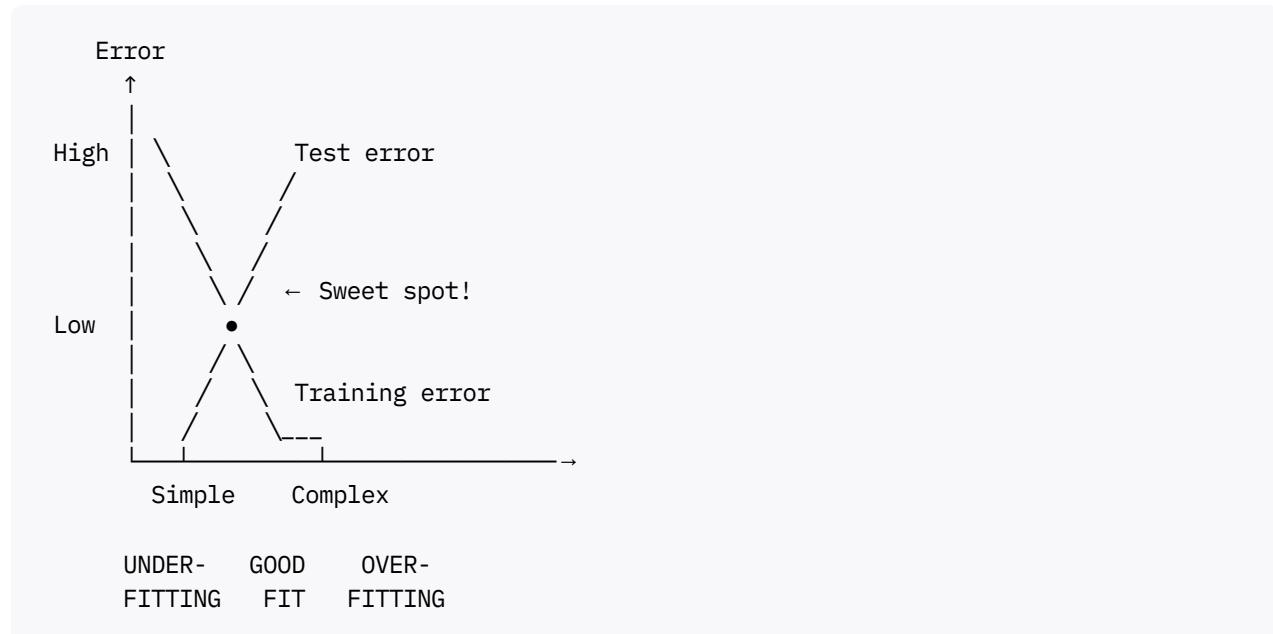
LARGE M: Can memorize → Overfitting  
 MEDIUM M: Just right! → Good fit ✓

## 8.15 Summary Table: Underfitting vs Good Fit vs Overfitting

| Aspect                    | Underfitting           | Good Fit         | Overfitting             |
|---------------------------|------------------------|------------------|-------------------------|
| <b>Training Error</b>     | High                   | Low              | Very Low                |
| <b>Test Error</b>         | High                   | Low              | High                    |
| <b>Gap</b>                | Small                  | Small            | <b>LARGE</b>            |
| <b>Model Complexity</b>   | Too simple             | Just right       | Too complex             |
| <b>Parameters vs Data</b> | $d \ll m$              | $d \approx 0.1m$ | $d \gg m$               |
| <b>Problem</b>            | Can't capture patterns | —                | Memorizes noise         |
| <b>Training Time</b>      | Any                    | Optimal          | Too long                |
| <b>Symptom</b>            | Both errors high       | Both errors low  | Training good, test bad |
| <b>Solution</b>           | Increase capacity      | —                | Regularize (Topic 9!)   |
| <b>Example</b>            | Linear for sine wave   | 4th degree poly  | 15th degree poly        |

## 8.16 Visual Summary: The Three Regimes

Diagram: Model Complexity vs Error



## Explanation:

### Left side (Simple models):

- High training error
- High test error
- Model too simple to fit data
- **UNDERFITTING**

### Middle (Optimal complexity):

- Low training error
- Low test error
- Errors are close
- **GOOD FIT ✓**

### Right side (Complex models):

- Very low training error
- High test error
- Large gap
- **OVERFITTING**

## ✓ KEY TAKEAWAYS - TOPIC 8

### Main Concepts:

#### 1. Two Types of Risk:

- **Empirical Risk:** Performance on training data (what we optimize)
- **True Risk:** Performance on all data (what we care about)
- **Gap:** Generalization gap =  $R_{\text{true}} - R_{\text{emp}}$

#### 2. Overfitting Definition:

$R_{\text{emp}}(\theta)$  LOW, but  $R_{\text{true}}(\theta)$  HIGH

Model performs well on training but poorly on new data.

#### 3. Three Causes of Overfitting:

- Limited training data (sparse coverage of  $P_{\text{data}}$ )
- Model too complex (too many parameters)
- Training too long (memorizes training specifics)

#### 4. Extreme Case: Memorization

- Model = Lookup table of training data
- Perfect on training, useless on new data

- $R_{\text{true}} = \infty$

## 5. Generalization Goal:

Model should perform well on **unseen data**

$$R_{\text{true}}(\theta) \approx R_{\text{emp}}(\theta)$$

## 6. Detection: Train-test split

- Monitor both training and test loss
- If test loss increases while training decreases → Overfitting!

## 7. Trade-off:

Simple model → Underfitting (high bias)  
 Complex model → Overfitting (high variance)  
 Need balance! (Bias-variance trade-off - next topic)

## Key Quote (Slide 20):

"Empirical risk minimization can easily overfit the data"  
 "Generalization: the data is a sample, usually there is vast amount of samples that you have never seen. Your model should generalize well to these 'never-seen' samples."

## The Problem-Solution Chain:

```

PROBLEM: Overfitting
↓
Models memorize training data instead of learning patterns
↓
Poor performance on new data
↓
SOLUTION: ??? (Coming next!)

```

## □ Connection to Next Topic

We've identified the problem:

- ✗ Overfitting occurs when model is too complex
- ✗ Memorizes training data
- ✗ Fails to generalize

## Critical Questions Remain:

- ? How to prevent overfitting?
- ? How to control model complexity?
- ? What if we need a complex model?
- ? Is there a way to have the best of both worlds?

This leads us to **TOPIC 9: BIAS-VARIANCE TRADE-OFF** ☺

Where we'll learn:

- The fundamental trade-off in learning
- Bias: Error from underfitting
- Variance: Error from overfitting
- Why we can't eliminate both simultaneously
- How to find the right balance
- Connection to model selection

Ready to continue to **TOPIC 9: BIAS-VARIANCE TRADE-OFF? Just say "next"!** ☺

## ☺ **TOPIC 9: BIAS-VARIANCE TRADE-OFF**

**(The Fundamental Dilemma in Machine Learning)**

### **9.0 Big Picture: The Unavoidable Trade-Off**

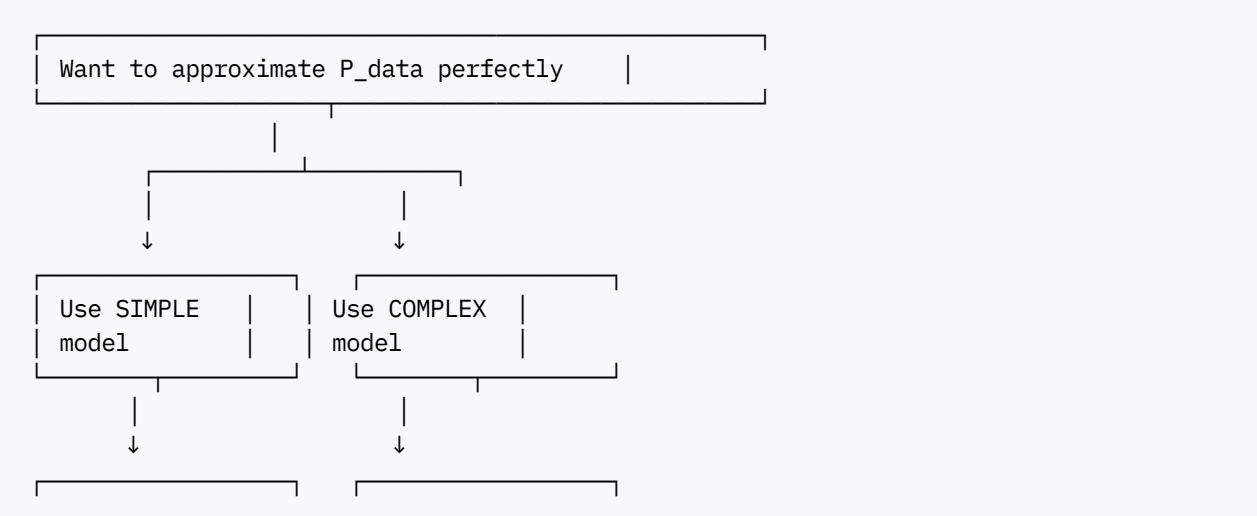
#### **Where We Are:**

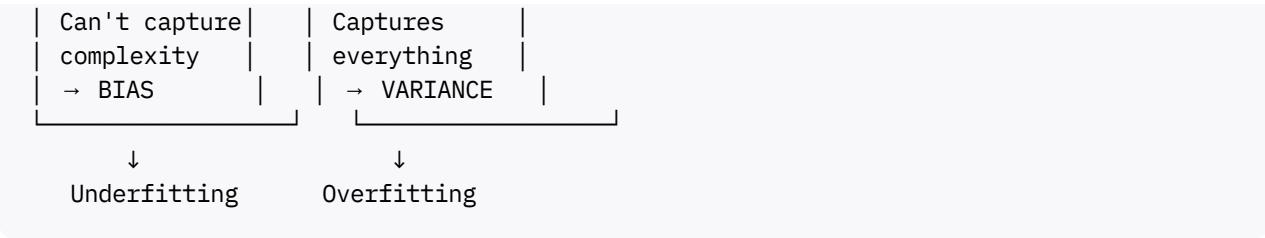
From Topic 8, we learned about overfitting:

- ✗ Too simple model → Underfitting (bad on training AND test)
- ✗ Too complex model → Overfitting (good on training, bad on test)
- ✓ Need to find the "right" complexity!

**But there's a deeper, fundamental problem...**

#### **The Central Dilemma:**





**The cruel reality:** We **cannot eliminate both bias AND variance simultaneously!**

### From Lecture Slide 21:

"There is an inherent bias-variance trade off when selecting the hypothesis class. Error in learning due to both things: bias and variance."

This topic explores:

1. What is bias? (Error from simplicity)
2. What is variance? (Error from complexity)
3. Why can't we eliminate both?
4. How to find the optimal trade-off?

## 9.1 Mathematical Framework: Sources of Error

### Total Error Decomposition:

For any learned model  $P_\theta$ , the **total error** can be decomposed:

$$\text{Total Error} = \text{Bias}^2 + \text{Variance} + \text{Irreducible Error}$$

Let's understand each component:

### Component 1: Irreducible Error

**Definition:** Error inherent in the problem itself

$$\text{Irreducible Error} = \text{Var}[\epsilon]$$

#### Sources:

- **Measurement noise:** Sensors aren't perfect
- **Inherent randomness:** Quantum effects, chaotic systems
- **Missing information:** Hidden variables we don't observe

#### Example:

Predicting stock prices:

- Influenced by unpredictable human emotions
- Affected by unknown future events
- Inherently stochastic
- Cannot be predicted perfectly, no matter how good the model!

**Important:** This is **NOT** about our model - it's about the problem itself!

## Component 2: Bias

**Definition:** Error from **wrong assumptions** in the model

$$\text{Bias} = \mathbb{E}_D[P_\theta(x)] - P_{\text{data}}(x)$$

(Expected prediction minus true value, averaged over all possible training sets)

**In words:** How far off is our **best possible model** in the hypothesis space from the truth?

## Component 3: Variance

**Definition:** Error from **sensitivity to training data**

$$\text{Variance} = \mathbb{E}_D[(P_\theta(x) - \mathbb{E}_D[P_\theta(x)])^2]$$

**In words:** How much does the learned model **vary** when trained on different training sets?

## 9.2 BIAS: Error from Model Limitations

### Formal Definition:

From Slide 21:

"If the hypothesis space is very limited, it might not be able to represent  $P_{\text{data}}$ , even with unlimited data"

This type of limitation is called bias, as the learning is limited on how close it can approximate the target distribution"

### Mathematical Definition:

$$\boxed{\text{Bias}[\theta] = \mathbb{E}_{D \sim P_{\text{data}}^m}[P_\theta(x)] - P_{\text{data}}(x)}$$

Where:

- $\mathbb{E}_{D \sim P_{\text{data}}^m}$ : Expectation over all possible training sets of size  $m$
- $P_\theta(x)$ : Model trained on training set  $D$
- $P_{\text{data}}(x)$ : True distribution

## Intuitive Meaning:

BIAS asks: "Even with INFINITE data, can our model family represent the true distribution?"

If NO → HIGH BIAS (model family too limited)

If YES → LOW BIAS (model family expressive enough)

## Why Bias Occurs:

### 1. Model Family Too Restrictive:

TRUE DISTRIBUTION P\_data:  
Complex, non-linear pattern

MODEL FAMILY M:  
Only contains linear models  
↓  
CANNOT represent P\_data  
↓  
HIGH BIAS!

### 2. Wrong Inductive Assumptions:

Assume: Data is linear  
Reality: Data is quadratic  
↓  
No amount of data helps!  
↓  
Model systematically wrong

## 9.3 Detailed Example: Bias in Polynomial Fitting

### Setup:

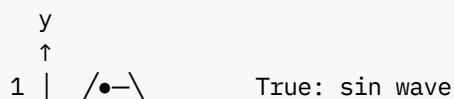
**True function:**  $y = \sin(2\pi x) + \epsilon$  where  $\epsilon \sim \mathcal{N}(0, 0.1)$

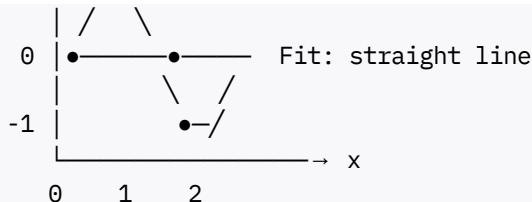
**Training data:** Infinite samples (eliminate variance!)

### Model 1: Linear (High Bias)

**Model family:**  $M = \{y = \theta_0 + \theta_1 x\}$

**Best fit:**  $y = 0.1 + 0.05x$





Even with infinite data, can't fit sine wave with line!  
Systematic error at all points!

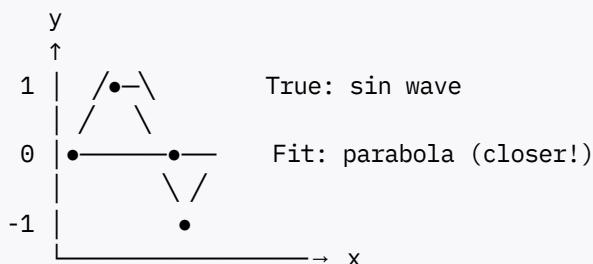
### Analysis:

- **True function:** Oscillates
- **Best linear fit:** Flat line
- **Bias:** Large (can't capture oscillations)
- **Variance:** Zero (same line regardless of data)
- **Total Error:** High (dominated by bias)

### Model 2: Quadratic (Medium Bias)

**Model family:**  $M = \{y = \theta_0 + \theta_1x + \theta_2x^2\}$

**Best fit:**  $y = 0.2 + 0.1x - 0.3x^2$



Better approximation, but still systematic error

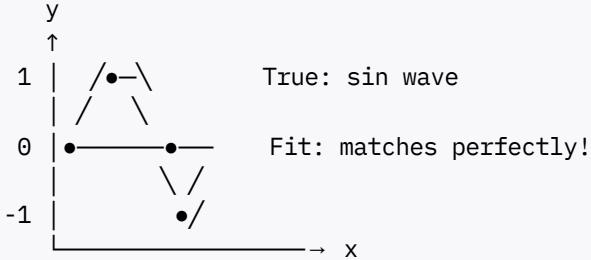
### Analysis:

- **True function:** Sine wave (periodic)
- **Best quadratic:** Single parabola
- **Bias:** Medium (captures general shape, misses details)
- **Variance:** Zero
- **Total Error:** Medium

### Model 3: Fourier Series (Low Bias)

**Model family:**  $M = \{y = \sum_{k=0}^{10} a_k \sin(kx) + b_k \cos(kx)\}$

**Best fit:** Can represent sine wave perfectly!



Model can represent true function!

#### Analysis:

- **True function:** Sine wave
- **Best Fourier:** Exact match
- **Bias:** Near zero (model family includes true function)
- **Variance:** Will be non-zero with finite data
- **Total Error:** Low bias, but might have high variance!

## 9.4 VARIANCE: Error from Data Sensitivity

#### Formal Definition:

From Slide 21:

"If we select a highly expressive hypothesis class, we might represent better the data

When we have small amount of data, multiple models can fit well, or even better than the true model. Moreover, small perturbations on D will result in very different estimates

This limitation is call the variance."

#### Mathematical Definition:

$$\text{Variance}[\theta] = \mathbb{E}_{D \sim P_{\text{data}}^m} [(P_\theta(x) - \mathbb{E}_D[P_\theta(x)])^2]$$

**In words:** How much does the learned model **change** when trained on different samples?

## Intuitive Meaning:

VARIANCE asks: "How much does the model change if I use a different training set?"

If models are very different → HIGH VARIANCE

If models are similar → LOW VARIANCE

## Why Variance Occurs:

### 1. Limited Training Data:

TRUE DISTRIBUTION P\_data:

Many possible data patterns

TRAINING SET D:

Only see SMALL sample

↓

Model fits THIS specific sample

↓

Different sample → Different model!

↓

HIGH VARIANCE!

### 2. Model Too Complex:

Model has 1 million parameters

Training data has 1,000 points

↓

Many parameter settings fit training well

↓

Small data changes → Large model changes

↓

HIGH VARIANCE!

## 9.5 Detailed Example: Variance in Polynomial Fitting

### Setup:

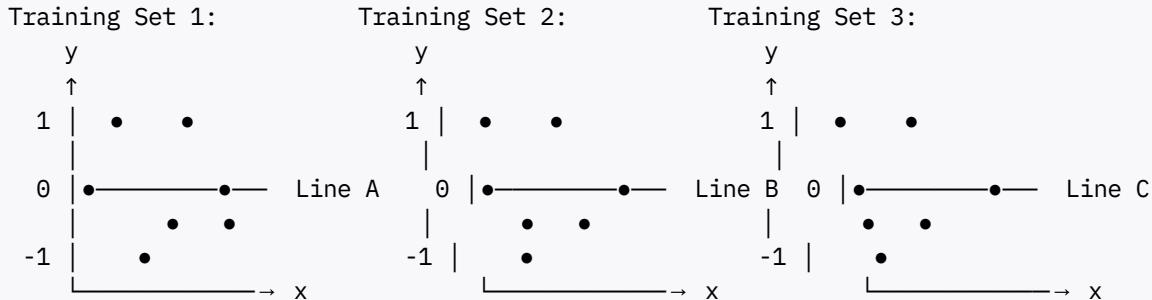
**True function:**  $y = \sin(2\pi x) + \epsilon$

**Training data:** Only 10 points (limited data!)

## Experiment:

Train on **three different training sets** (each with 10 points sampled from same distribution)

### Model 1: Linear (Low Variance)



$$\text{Line A: } y = 0.10 + 0.05x$$

$$\text{Line B: } y = 0.12 + 0.04x \leftarrow \text{Very similar!}$$

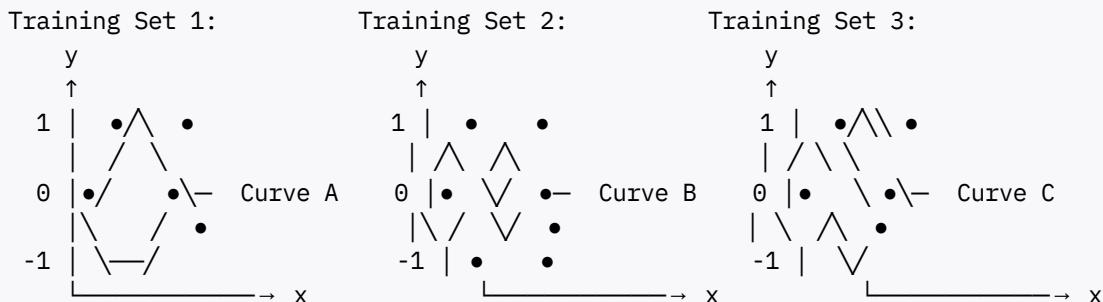
$$\text{Line C: } y = 0.09 + 0.06x$$

Variance: **LOW** (lines are almost identical)

### Analysis:

- Different training sets → Very similar models
- Model is **stable** (insensitive to data)
- **Variance:** Low ✓
- **But:** High bias (can't fit sine wave) ✗

### Model 2: 15th Degree Polynomial (High Variance)



Curve A: Wiggly, goes through every point

Curve B: COMPLETELY DIFFERENT shape!

Curve C: Yet another totally different curve!

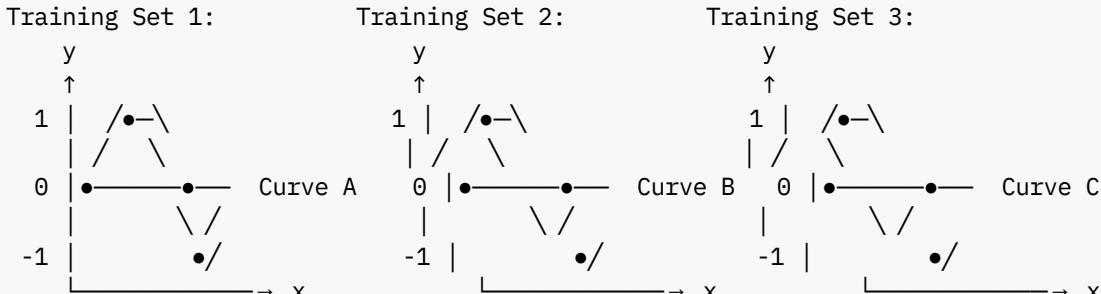
Variance: **VERY HIGH** (curves are wildly different)

### Analysis:

- Different training sets → Completely different models

- Model is **unstable** (very sensitive to data)
- **Variance:** High ✗
- **But:** Low bias (can fit sine wave) ✓

### Model 3: 4th Degree Polynomial (Good Balance)



Curve A, B, C: Similar smooth curves

Variance: MODERATE (curves similar but not identical)

### Analysis:

- Different training sets → Similar models
- Model is **reasonably stable**
- **Variance:** Low-medium ✓
- **Bias:** Low-medium ✓
- **Total Error:** Best! ✓✓

## 9.6 The Trade-Off: You Can't Have Both

### The Fundamental Dilemma:

$$\text{Total Error} = \text{Bias}^2 + \text{Variance} + \text{Irreducible Error}$$

### The cruel truth:

Decrease Bias → Increase Variance  
 Decrease Variance → Increase Bias

Can't minimize both simultaneously!

## Why The Trade-Off Exists:

### To reduce BIAS:

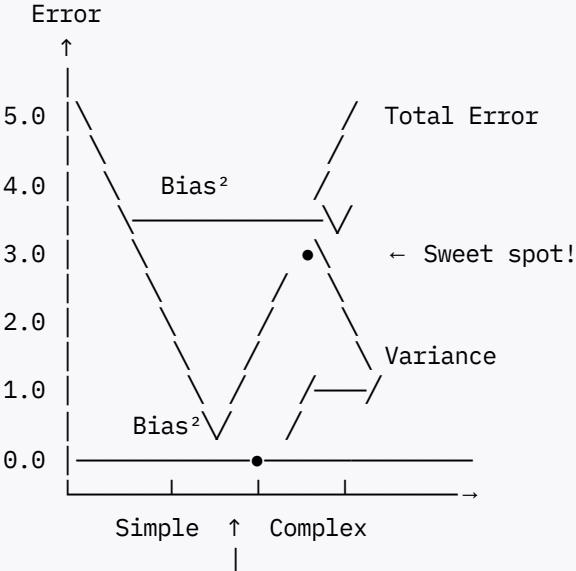
```
Need more complex model
↓
More parameters
↓
Model more flexible
↓
Can fit data better
↓
But: More sensitive to training data
↓
VARIANCE INCREASES! ✗
```

### To reduce VARIANCE:

```
Need simpler model
↓
Fewer parameters
↓
Model more rigid
↓
Less affected by training data
↓
But: Can't capture complexity
↓
BIAS INCREASES! ✗
```

## 9.7 Visual Representation: The U-Shaped Curve

### Classic Bias-Variance Plot:



### Optimal complexity

LEFT (Simple): High bias, low variance  
CENTER (Optimal): Balanced  
RIGHT (Complex): Low bias, high variance

## Three Regions:

### 1. Underfit Region (Left):

```
Model too simple
↓
High Bias2 (dominates)
Low Variance (small contribution)
↓
Total Error HIGH
```

### 2. Sweet Spot (Center):

```
Model complexity just right
↓
Medium Bias2
Medium Variance
↓
Total Error MINIMIZED ✓
```

### 3. Overfit Region (Right):

```
Model too complex
↓
Low Bias2 (small contribution)
High Variance (dominates)
↓
Total Error HIGH
```

## 9.8 Detailed Mathematical Example

Let's compute bias and variance for a **concrete scenario**.

### Setup:

**True model:**  $y = 2x + 3 + \epsilon$  where  $\epsilon \sim \mathcal{N}(0, 1)$

**Training:** 5 data points per training set

### Models:

1. Constant:  $y = \theta_0$

2. Linear:  $y = \theta_0 + \theta_1 x$

3. High-degree polynomial:  $y = \sum_{i=0}^{10} \theta_i x^i$

## Experiment:

Generate 1000 different training sets, train each model, compute:

$$\text{Bias}^2 = (\mathbb{E}_D[\hat{y}] - y_{true})^2$$

$$\text{Variance} = \mathbb{E}_D[(\hat{y} - \mathbb{E}_D[\hat{y}])^2]$$

## Results (at test point $x = 1.5$ ):

**True value:**  $y_{true} = 2(1.5) + 3 = 6$

### Model 1: Constant

1000 training sets produce:

$\theta_0$  estimates: 5.8, 6.1, 5.9, 6.2, ... (all around 6)

Average prediction:  $E[\hat{y}] = 6.0$

$\text{Bias}^2 = (6.0 - 6.0)^2 = 0$  (perfect on average!)

Wait... but model is  $y = \text{constant}$ , true is  $y = 2x + 3$

At  $x=1.5$ : true is 6, average prediction is 6

But at  $x=0$ : true is 3, prediction is 6!

→ Bias changes with  $x$ !

More careful analysis shows:

$\text{Bias}^2 = 0$  at this specific  $x$

But non-zero at most  $x$  values

$\text{Variance} = E[(\hat{y} - 6.0)^2] = 0.2$  (low - predictions consistent)

Total Error =  $0 + 0.2 + 1.0 = 1.2$

### Model 2: Linear (BEST)

1000 training sets produce:

Fits:  $y = 1.8x + 3.2$ ,  $y = 2.1x + 2.9$ ,  $y = 2.0x + 3.0$ , ...

Average prediction at  $x=1.5$ :

$E[\hat{y}] = E[\theta_0 + \theta_1(1.5)] = 3.0 + 2.0(1.5) = 6.0$

$\text{Bias}^2 = (6.0 - 6.0)^2 = 0$  ✓ (model family includes true model!)

$\text{Variance} = E[(\hat{y} - 6.0)^2] = 0.4$  (predictions vary somewhat)

Total Error =  $0 + 0.4 + 1.0 = 1.4$

This is BEST! (only slightly worse than constant because true model is linear, so bias is zero for linear model)

### Model 3: 10th Degree Polynomial (WORST)

1000 training sets produce wildly different fits:

$$\text{Fit 1: } y = 6 + 0.1x + 0.05x^2 - 0.3x^3 + 2x^4 - 5x^5 + \dots$$

$$\text{Fit 2: } y = 5.5 - 0.2x + 0.5x^2 + 0.1x^3 - 1x^4 + 3x^5 + \dots$$

(Very different coefficients!)

Average prediction at  $x=1.5$ :

$$E[\hat{y}] \approx 6.1 \quad (\text{close to true, averaged over many fits})$$

$$\text{Bias}^2 = (6.1 - 6.0)^2 = 0.01 \quad \checkmark \quad (\text{low - model can fit anything!})$$

$$\text{Variance} = E[(\hat{y} - 6.1)^2] = 8.5 \quad \times \quad (\text{HUGE - predictions all over the place!})$$

$$\text{Total Error} = 0.01 + 8.5 + 1.0 = 9.51 \quad (\text{WORST!})$$

### Summary Table:

| Model     | Parameters | Bias <sup>2</sup> | Variance | Irreducible | Total Error   |
|-----------|------------|-------------------|----------|-------------|---------------|
| Constant  | 1          | 0.00              | 0.20     | 1.0         | <b>1.20</b>   |
| Linear    | 2          | 0.00              | 0.40     | 1.0         | <b>1.40 ✓</b> |
| 10th Poly | 11         | 0.01              | 8.50     | 1.0         | <b>9.51 ✗</b> |

**Winner:** Linear model (matches true model complexity)!

### 9.9 Real-World Example: From Lecture Slide 22

#### The Polynomial Example (Slide 22):

##### From the lecture:

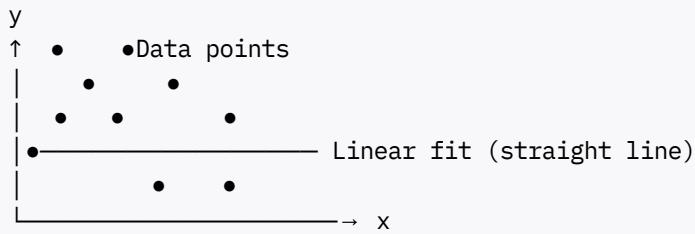
"There is an inherent bias-variance trade off when selecting the hypothesis class."

Three scenarios shown:

- Hypothesis space: linear relationship → Underfits
- Hypothesis space: high degree polynomial → Overfits
- Hypothesis space: low degree polynomial → Right tradeoff

## Visual from Slide:

LINEAR (Underfit):

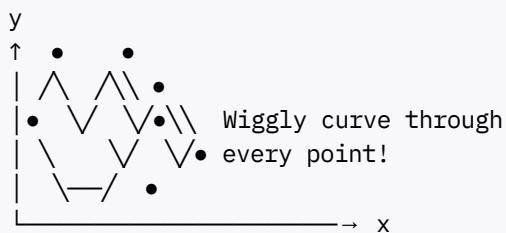


Bias: HIGH (can't capture curvature)

Variance: LOW (same line regardless of data)

Does it fit well? NO - Underfits

HIGH DEGREE POLYNOMIAL (Overfit):

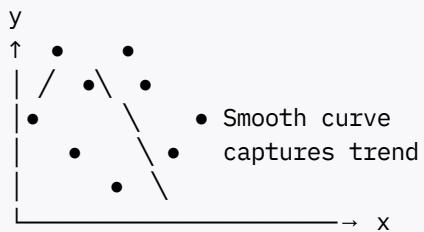


Bias: LOW (can fit any shape)

Variance: HIGH (different data → different crazy curves)

Does it fit well? NO - Overfits

LOW DEGREE POLYNOMIAL (Good):



Bias: MEDIUM (slight approximation error)

Variance: MEDIUM (stable across datasets)

Does it fit well? YES - Right tradeoff ✓

## 9.10 How to Find the Sweet Spot

### Strategy 1: Cross-Validation

**Method:** Try different model complexities, evaluate on validation set

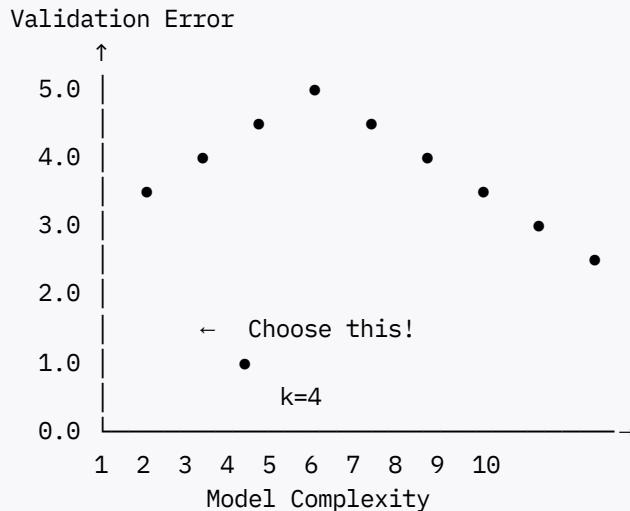
For each model complexity  $k = 1, 2, \dots, 20$ :

1. Train model with  $k$  parameters on training set
2. Evaluate on validation set

### 3. Record validation error

Choose  $k^*$  with lowest validation error!

#### Visual:



Best:  $k=4$  (lowest validation error)

## Strategy 2: Regularization (Next Topic!)

Idea: Use complex model but **penalize** complexity

$$\text{Objective} = \text{Log-likelihood} - \lambda \times \text{Complexity Penalty}$$

#### Benefits:

- Can use expressive model (low bias)
- Penalty prevents overfitting (low variance)
- **Best of both worlds!** ✓

## Strategy 3: More Data

**Key insight:** More data reduces variance!

Training size  $m$ :

|               |               |                 |            |
|---------------|---------------|-----------------|------------|
| $m = 100$     | $\rightarrow$ | Variance = 10.0 | (high)     |
| $m = 1,000$   | $\rightarrow$ | Variance = 3.2  | (medium)   |
| $m = 10,000$  | $\rightarrow$ | Variance = 1.0  | (low)      |
| $m = 100,000$ | $\rightarrow$ | Variance = 0.3  | (very low) |

More data  $\rightarrow$  More stable model  $\rightarrow$  Lower variance!

## The statistical learning bound revisited:

$$\text{Variance} \propto \frac{d}{m}$$

Where:

- $d$  = number of parameters
- $m$  = training set size

**Implication:** Can use more complex model if you have more data!

## 9.11 Bias-Variance in Deep Learning

### Modern Deep Learning Paradox:

Deep neural networks:

- Millions of parameters ( $d \gg m$ )
- Should have HUGE variance
- Should overfit terribly

But in practice:

- Often generalize well!
- Why? ☺

### Explanations:

#### 1. Over-parameterization:

When  $d \gg m$ :

- Multiple global minima exist
- SGD finds "flat" minima (good generalization)
- Implicit regularization from SGD noise

#### 2. Inductive Biases:

Architecture design:

- Convolutions (spatial locality)
- Attention (relevance)
- Built-in biases help generalization

#### 3. Data Augmentation:

Artificially increase  $m$ :

- Rotate images
- Add noise
- Effectively larger training set
- Reduces variance

#### 4. Early Stopping:

Don't train to zero loss:

- Stop before fully fitting training data
- Prevents excessive variance
- Acts as implicit regularization

## 9.12 Practical Guidelines

### How to Diagnose:

#### Check Training vs Validation Error:

SCENARIO 1:

Training error: HIGH (e.g., 5.0)  
Validation error: HIGH (e.g., 5.2)  
Gap: SMALL (0.2)

Diagnosis: HIGH BIAS (underfitting)  
Solution: Increase model complexity

SCENARIO 2:

Training error: LOW (e.g., 0.1)  
Validation error: HIGH (e.g., 4.5)  
Gap: LARGE (4.4)

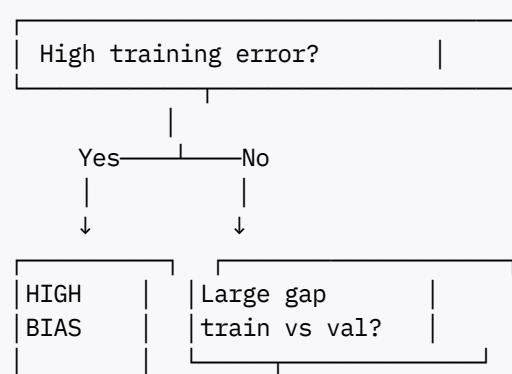
Diagnosis: HIGH VARIANCE (overfitting)  
Solution: Decrease complexity or regularize

SCENARIO 3:

Training error: MEDIUM (e.g., 1.2)  
Validation error: MEDIUM (e.g., 1.5)  
Gap: SMALL (0.3)

Diagnosis: GOOD BALANCE ✓  
Solution: This is good! Maybe try slight tuning

### Decision Tree:





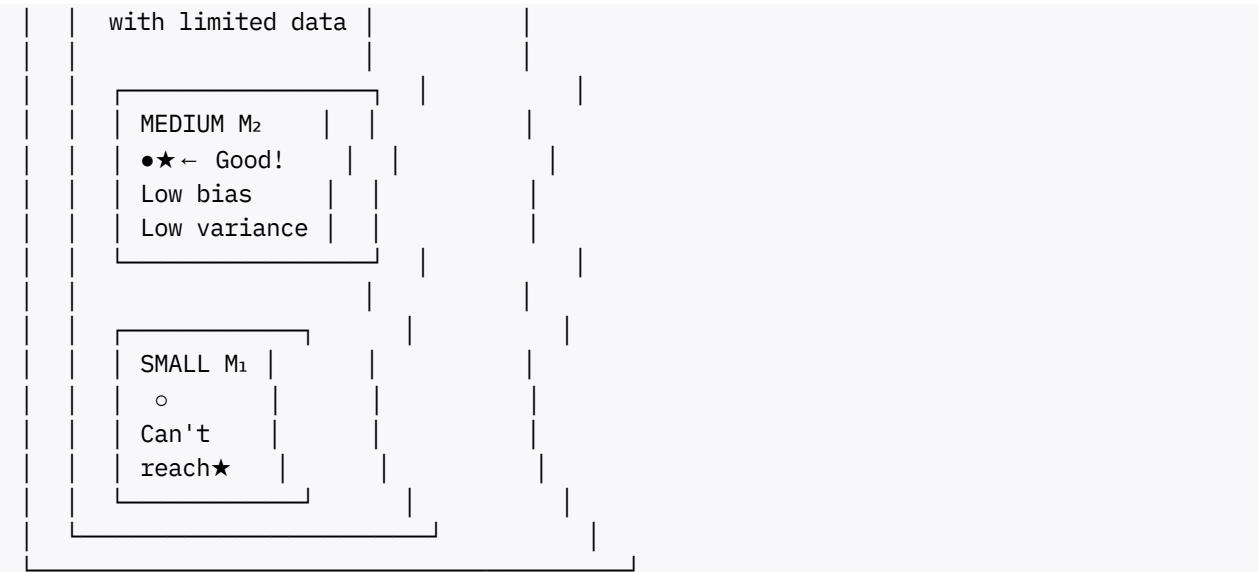
## 9.13 Comprehensive Comparison Table

| Aspect                  | High Bias            | Balanced            | High Variance               |
|-------------------------|----------------------|---------------------|-----------------------------|
| <b>Model Complexity</b> | Too simple           | Just right          | Too complex                 |
| <b>Parameters</b>       | $d \ll m$            | $d \approx 0.1m$    | $d \gg m$                   |
| <b>Training Error</b>   | High                 | Medium-Low          | Very Low                    |
| <b>Test Error</b>       | High                 | Medium-Low          | High                        |
| <b>Error Gap</b>        | Small                | Small               | <b>Large</b>                |
| <b>Problem</b>          | Can't fit            | —                   | Memorizes                   |
| <b>Symptom</b>          | Both errors high     | Both low, gap small | Training low, test high     |
| <b>Cause</b>            | Underfit             | —                   | Overfit                     |
| <b>Example</b>          | Linear for sine      | 4th deg poly        | 15th deg poly               |
| <b>Fix</b>              | More capacity        | —                   | Less capacity or regularize |
| <b>What dominates</b>   | $\text{Bias}^2$ term | Balanced            | Variance term               |

## 9.14 Connection: Hypothesis Space and Bias

### Visual: Hypothesis Space and Bias





SMALL  $M_1$ : High bias (can't represent  $P_{\text{data}}$ )

MEDIUM  $M_2$ : Balanced (can approximate  $P_{\text{data}}$  well)

LARGE  $M_3$ : Low bias BUT high variance with small data

## 9.15 Summary: The Fundamental Trade-Off

### The Equation:

$$\text{Total Error} = \underbrace{\text{Bias}^2}_{\text{Underfit}} + \underbrace{\text{Variance}}_{\text{Overfit}} + \underbrace{\text{Irreducible}}_{\text{Can't fix}}$$

### The Trade-Off:

#### SIMPLE MODEL:

- ✓ Low variance (stable)
- ✗ High bias (can't fit)
- Underfitting

#### COMPLEX MODEL:

- ✓ Low bias (flexible)
- ✗ High variance (unstable)
- Overfitting

#### OPTIMAL MODEL:

- ✓ Medium bias
- ✓ Medium variance
- ✓ Minimizes total error!

## Key Insights:

1. **Cannot eliminate both** bias and variance
2. Must find **balance** (sweet spot)
3. **More data** helps reduce variance
4. **Regularization** helps control variance (next topic!)
5. Choice of hypothesis space determines the trade-off

## ✓ KEY TAKEAWAYS - TOPIC 9

### Main Concepts:

#### 1. Total Error Decomposition:

$$\text{Error} = \text{Bias}^2 + \text{Variance} + \text{Irreducible}$$

#### 2. Bias (from Slide 21):

- Error from model being **too simple**
- Can't represent  $P_{data}$  even with infinite data
- Caused by limited hypothesis space
- **High bias** → **Underfitting**

#### 3. Variance (from Slide 21):

- Error from model being **too sensitive** to training data
- Small data changes → Large model changes
- Caused by high model complexity + limited data
- **High variance** → **Overfitting**

#### 4. The Trade-Off:

- Decrease bias → Increase variance
- Decrease variance → Increase bias
- **Must find balance!**

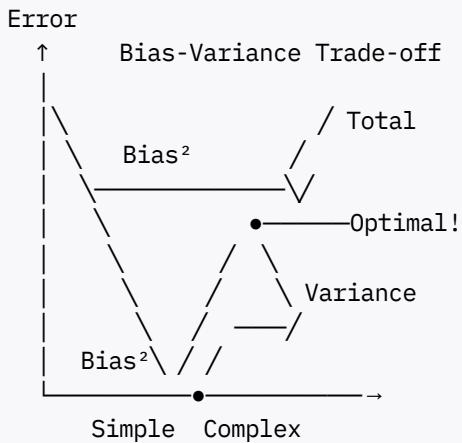
#### 5. From Slide 22: Three cases

- Linear: Underfits (high bias)
- High degree polynomial: Overfits (high variance)
- Low degree polynomial: Right tradeoff ✓

#### 6. Solutions:

- Cross-validation (find optimal complexity)
- More training data (reduces variance)
- Regularization (next topic!)

## The Visual Summary:



Cannot minimize both simultaneously!

## Connection to Next Topic

We understand the problem:

- ✓ Bias-variance trade-off exists
- ✓ Can't eliminate both
- ✓ Need balance

But: What if we NEED a complex model?

**Question:** Can we use a complex model (low bias) while **controlling variance**?

**Answer:** YES! Through **REGULARIZATION!**

This leads us to **TOPIC 10: HOW TO AVOID OVERFITTING? (REGULARIZATION TECHNIQUES)**

Where we'll learn:

- Regularization techniques
- L1, L2 penalties
- Early stopping
- Dropout
- Weight sharing
- Validation sets for model selection
- Practical strategies

Ready to continue to **TOPIC 10: HOW TO AVOID OVERFITTING? Just say "next"!**

# □ TOPIC 10: HOW TO AVOID OVERFITTING?

## (Regularization and Practical Techniques)

### 10.0 Big Picture: Controlling the Trade-Off

#### Where We Are:

From Topics 8-9, we understand the problem:

- ✗ Overfitting occurs when model too complex
- ✗ Bias-variance trade-off: can't eliminate both
- ✗ Complex models have low bias BUT high variance

#### The Challenge:

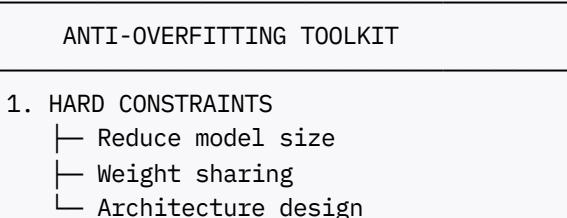
We often NEED complex models (e.g., deep neural networks)  
↓  
But complex models overfit!  
↓  
How to solve this? □

### The Solution: Multiple Strategies

#### From Lecture Slide 23:

- "How to avoid overfitting?"
- Hard constraints (e.g., by selecting a less expressive model family)
    - Smaller neural networks with less parameters
    - Weight sharing
  - Soft preference for "simpler" models: Occam Razor
    - Augment the objective function with regularization
  - Evaluate generalization performance on a held-out validation set

#### Overview of Techniques:



- 2. SOFT REGULARIZATION
  - |— L2 penalty (Ridge)
  - |— L1 penalty (Lasso)
  - |— Other penalties
- 3. TRAINING STRATEGIES
  - |— Early stopping
  - |— Dropout
  - |— Data augmentation
- 4. VALIDATION & SELECTION
  - |— Held-out validation set

## 10.1 STRATEGY 1: Hard Constraints (Model Selection)

**Philosophy:** Restrict the Search Space

From Slide 23:

"Hard constraints, e.g. by selecting a less expressive model family"

**Core Idea:**

Instead of searching over ALL possible models:  
 → Restrict to SMALLER family M  
 → Physically limit model capacity  
 → Can't overfit as much (less flexibility)

**Mathematical Formulation:**

**Before (Unconstrained):**

$$\theta^* = \arg \max_{\theta \in \Theta_{\text{large}}} \ell(\theta; D)$$

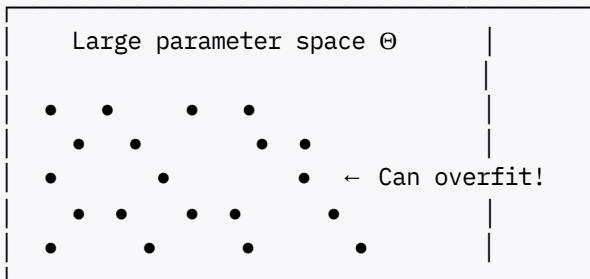
**After (Constrained):**

$$\theta^* = \arg \max_{\theta \in \Theta_{\text{small}} \subset \Theta_{\text{large}}} \ell(\theta; D)$$

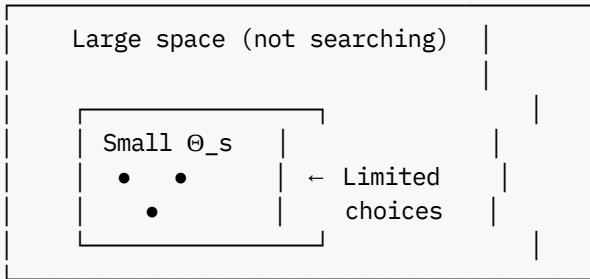
Where  $\Theta_{\text{small}}$  is a restricted parameter space.

## Visual Representation:

UNCONSTRAINED SEARCH:



CONSTRAINED SEARCH:



## 10.2 TECHNIQUE 1A: Smaller Neural Networks

From Slide 23:

"Smaller neural networks with less parameters"

The Approach:

Reduce the number of parameters by:

1. **Fewer layers** (shallow network)
2. **Fewer neurons per layer** (narrow network)
3. **Both**

Mathematical Example:

Large Network (Prone to Overfit):

Architecture:

Input (784)  $\rightarrow$  Hidden<sub>1</sub> (1024)  $\rightarrow$  Hidden<sub>2</sub> (1024)  $\rightarrow$  Hidden<sub>3</sub> (512)  $\rightarrow$  Output (10)

Parameter count:

Layer 1:  $784 \times 1024 = 802,816$

Layer 2:  $1024 \times 1024 = 1,048,576$

Layer 3:  $1024 \times 512 = 524,288$

Layer 4:  $512 \times 10 = 5,120$

TOTAL: 2,380,800 parameters! (2.4 million)

With  $m=60,000$  training examples:

$d/m = 2,380,800 / 60,000 \approx 40$

→ 40 parameters per data point!

→ HIGH risk of overfitting ✗

## Small Network (Less Overfitting):

Architecture:

Input (784) → Hidden<sub>1</sub> (128) → Hidden<sub>2</sub> (64) → Output (10)

Parameter count:

Layer 1:  $784 \times 128 = 100,352$

Layer 2:  $128 \times 64 = 8,192$

Layer 3:  $64 \times 10 = 640$

TOTAL: 109,184 parameters (109K)

With  $m=60,000$  training examples:

$d/m = 109,184 / 60,000 \approx 1.8$

→ 1.8 parameters per data point

→ Much safer! ✓

## Trade-off:

LARGE NETWORK:

- ✓ Can learn complex patterns (low bias)
- ✗ High risk of overfitting (high variance)
- ✗ Slower to train
- ✗ More memory

SMALL NETWORK:

- ✗ May miss complex patterns (higher bias)
- ✓ Less overfitting (low variance)
- ✓ Faster to train
- ✓ Less memory

## Practical Guidelines:

### Start small, grow if needed:

1. Start with small network
2. Train to convergence
3. If underfitting (high training error):
  - Increase capacity (add layers/neurons)

4. If overfitting (large train-val gap):
  - Keep current size or reduce
  - Apply other regularization (next sections)

## Example: MNIST Classification

EXPERIMENT:

Network 1: [784 → 32 → 10]  
 Parameters: 25,450  
 Training acc: 92%  
 Test acc: 91%  
 Gap: 1%  
 Diagnosis: Slight underfit, but good generalization ✓

Network 2: [784 → 128 → 64 → 10]  
 Parameters: 109,184  
 Training acc: 99%  
 Test acc: 97%  
 Gap: 2%  
 Diagnosis: Best balance! ✓✓

Network 3: [784 → 512 → 512 → 256 → 10]  
 Parameters: 933,386  
 Training acc: 99.9%  
 Test acc: 93%  
 Gap: 6.9%  
 Diagnosis: Overfitting! ✗

WINNER: Network 2 (medium size)

## 10.3 TECHNIQUE 1B: Weight Sharing

### From Slide 23:

"Weight sharing"

### The Concept:

Instead of having **independent parameters** for each connection, **reuse** the same parameters across multiple positions.

### Why Weight Sharing Helps:

WITHOUT weight sharing:  
 Each connection has its own parameter
 

- Many parameters
- High variance

WITH weight sharing:

- Same parameters used in multiple places
- Fewer unique parameters
- Lower variance
- Built-in inductive bias

## Example 1: Convolutional Neural Networks (CNNs)

### The Problem:

Fully connected for 28×28 image:

First layer:  $784 \times 1000 = 784,000$  parameters!

### The Solution (CNN with weight sharing):

Convolutional layer with 3×3 filters:

Filter size:  $3 \times 3 = 9$  weights

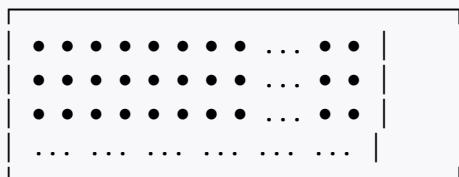
Number of filters: 32

TOTAL:  $9 \times 32 = 288$  parameters!

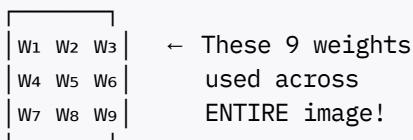
Reduction:  $784,000 \rightarrow 288$  (2700× fewer!) ✓

## Visual: Weight Sharing in Convolutions

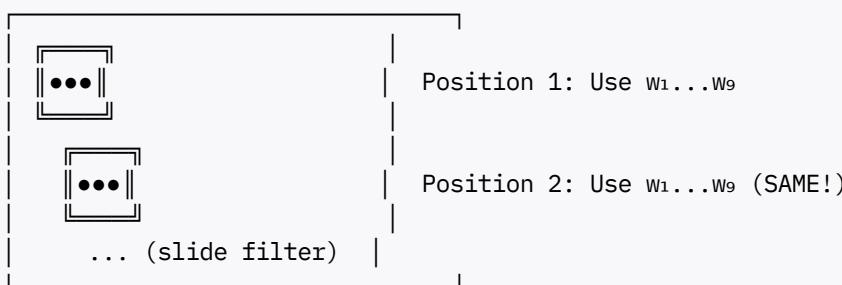
INPUT IMAGE (28×28):



FILTER (3×3) - SAME WEIGHTS EVERYWHERE:



APPLY TO EVERY POSITION:



Result: 9 parameters instead of 784,000!

## Benefits of Weight Sharing:

### 1. Massive parameter reduction

Parameters:  $O(k^2)$  instead of  $O(n^2)$

where  $k$  = filter size,  $n$  = image size

### 2. Translation invariance

Cat in top-left corner: Detected ✓  
Same cat in bottom-right: Also detected ✓  
(Same weights applied everywhere)

### 3. Better generalization

Fewer parameters → Lower variance → Less overfitting ✓

## Example 2: Autoregressive Models (From Lecture 3)

Recall: NADE, PixelRNN, etc.

### WITHOUT weight sharing:

Each conditional  $p(x_i | x_{<i}; \theta_i)$  has its own parameters  $\theta_i$   
→  $n$  sets of parameters (one per variable)  
→ Many parameters if  $n$  is large

### WITH weight sharing:

All conditionals share some parameters:  
 $p(x_i | x_{<i}; \theta_{\text{shared}})$   
→ Single neural network processes all conditionals  
→ Massively fewer parameters  
→ Examples: PixelCNN, PixelRNN

## Mathematical Formulation:

### No sharing:

$$P_\theta(x) = \prod_{i=1}^n p(x_i | x_{<i}; \theta_i)$$

where  $\theta = (\theta_1, \theta_2, \dots, \theta_n)$  are all independent.

### With sharing:

$$P_{\theta}(x) = \prod_{i=1}^n p(x_i|x_{<i}; \theta_{\text{shared}})$$

where the same  $\theta_{\text{shared}}$  is used for all  $i$ .

#### Parameter count:

- No sharing:  $O(n \times d)$  where  $d$  = parameters per conditional
- With sharing:  $O(d)$  (independent of  $n!$ )

## 10.4 STRATEGY 2: Soft Regularization (Occam's Razor)

#### Philosophy: Prefer Simpler Models

##### From Slide 23:

"Soft preference for 'simpler' models: Occam Razor  
Augment the objective function with regularization"

#### Occam's Razor:

**Principle:** "Among competing hypotheses, the simplest is most likely to be correct."

#### In Machine Learning:

If two models fit data equally well:  
 → Choose the SIMPLER one  
 → Simpler = better generalization (usually)

## How to Implement: Regularization

**Idea:** Add a **penalty term** to the objective that favors simple models.

#### Modified Objective:

$$\text{Objective} = \underbrace{\text{Loss}(\theta)}_{\text{Fit data}} + \underbrace{\lambda \cdot R(\theta)}_{\text{Complexity penalty}}$$

Where:

- $\text{Loss}(\theta)$ : Negative log-likelihood (MLE)
- $R(\theta)$ : Regularization term (measures complexity)
- $\lambda \geq 0$ : Regularization strength (hyperparameter)

## For Maximum Likelihood:

Original (unregularized):

$$\theta^* = \arg \max_{\theta} \sum_{j=1}^m \log P_{\theta}(x^{(j)})$$

Regularized:

$$\theta^* = \arg \max_{\theta} \left[ \sum_{j=1}^m \log P_{\theta}(x^{(j)}) - \lambda \cdot R(\theta) \right]$$

Or in minimization form (negative log-likelihood):

$$\theta^* = \arg \min_{\theta} \left[ - \sum_{j=1}^m \log P_{\theta}(x^{(j)}) + \lambda \cdot R(\theta) \right]$$

## 10.5 TECHNIQUE 2A: L2 Regularization (Ridge / Weight Decay)

Definition:

$$R(\theta) = \|\theta\|_2^2 = \sum_{i=1}^d \theta_i^2$$

Regularized objective:

$$\mathcal{L}_{\text{reg}}(\theta) = - \sum_{j=1}^m \log P_{\theta}(x^{(j)}) + \frac{\lambda}{2} \|\theta\|_2^2$$

What It Does:

Penalizes large parameter values:

$$\begin{aligned} \theta &= [10, -8, 12, 5] \rightarrow \|\theta\|^2 = 100+64+144+25 = 333 \text{ (HIGH penalty)} \\ \theta &= [2, -1, 3, 1] \rightarrow \|\theta\|^2 = 4+1+9+1 = 15 \text{ (LOW penalty)} \end{aligned}$$

Encourages parameters to stay SMALL!

Effect on Learning:

Gradient with L2 regularization:

$$\frac{\partial \mathcal{L}_{\text{reg}}}{\partial \theta_i} = - \sum_{j=1}^m \frac{\partial \log P_{\theta}(x^{(j)})}{\partial \theta_i} + \lambda \theta_i$$

**SGD update becomes:**

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \left[ -\nabla_{\theta} \ell(\theta) + \lambda \theta^{(t)} \right]$$

Rearranging:

$$\theta^{(t+1)} = (1 - \alpha \lambda) \theta^{(t)} + \alpha \nabla_{\theta} \ell(\theta)$$

**Interpretation:**

- Factor  $(1 - \alpha \lambda)$  **shrinks** parameters toward zero each step
- Also called "**weight decay**" in deep learning

**Why It Prevents Overfitting:**

Without L2:

- Model can use arbitrarily large weights
  - Can fit training data exactly
  - Overfits

With L2:

- Large weights penalized
  - Parameters stay smaller
  - Model more "smooth"
  - Better generalization ✓

**Detailed Example:**

**Setup:** Polynomial regression

**True function:**  $y = 2x - 1$

**Model:** 10th degree polynomial  $y = \sum_{i=0}^{10} \theta_i x^i$

**Training data:** 10 noisy points

**Without Regularization ( $\lambda = 0$ ):**

Learned parameters:

$\theta = [-1.2, 2.3, 15.6, -42.1, 38.7, -15.2, 3.8, -0.5, 0.03, -0.001, 0.00001]$

$||\theta||^2 = 1.44 + 5.29 + 243.36 + \dots \approx 3,847$  (HUGE!)

**Fit:** Wiggly curve through every training point

**Training error:** 0.01 (perfect!)

**Test error:** 8.5 (terrible!)

**Result:** OVERFIT ✗

## With L2 Regularization ( $\lambda = 0.1$ ):

Learned parameters:

$$\theta = [-0.98, 2.05, 0.12, -0.05, 0.02, 0.0, 0.0, 0.0, 0.0, 0.0]$$

$$\|\theta\|^2 = 0.96 + 4.20 + 0.014 + 0.0025 + 0.0004 \approx 5.2 \text{ (small!)}$$

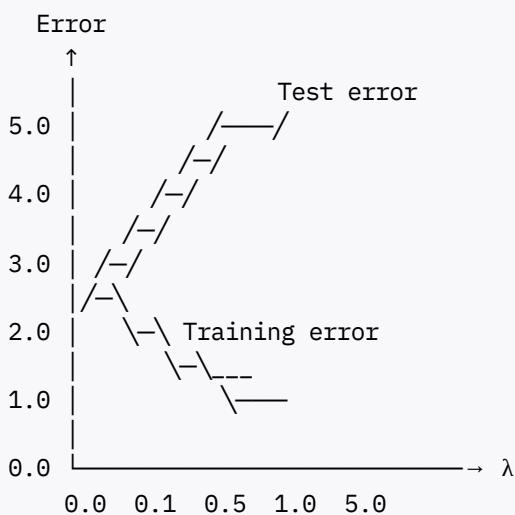
Fit: Smooth curve, approximately linear (like true function!)

Training error: 0.25 (slightly worse than unregularized)

Test error: 0.30 (MUCH better!)

Result: GOOD GENERALIZATION ✓

## Tuning $\lambda$ :



$\lambda$  too small (0): Overfits (high test error)

$\lambda$  too large (5): Underfits (high train & test error)

$\lambda$  just right (0.1-0.5): Best test error! ✓

## 10.6 TECHNIQUE 2B: L1 Regularization (Lasso)

### Definition:

$$R(\theta) = \|\theta\|_1 = \sum_{i=1}^d |\theta_i|$$

### Regularized objective:

$$\mathcal{L}_{\text{reg}}(\theta) = -\sum_{j=1}^m \log P_\theta(x^{(j)}) + \lambda \|\theta\|_1$$

## What It Does:

Encourages SPARSITY (many parameters become exactly zero):

With L2:  $\theta = [0.1, 0.2, 0.05, 0.15, 0.08, \dots]$  (all small but non-zero)  
With L1:  $\theta = [0.0, 0.5, 0.0, 0.0, 0.3, \dots]$  (many exact zeros!)

## Why Sparsity Helps:

### 1. Automatic Feature Selection:

$\theta_i = 0 \rightarrow$  Feature  $i$  not used  
→ Effectively removes irrelevant features  
→ Simpler model!

### 2. Interpretability:

Only non-zero weights matter  
→ Easy to see which features are important  
→ Model is more interpretable

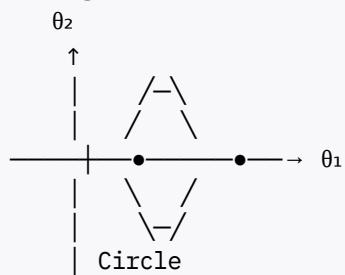
### 3. Computational Efficiency:

Sparse model → Can skip zero-weight computations  
→ Faster inference

## Comparison: L1 vs L2

VISUAL: CONSTRAINT REGIONS

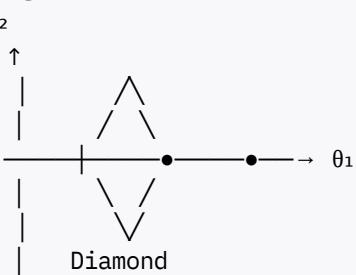
L2 Regularization:



Optimum often NOT  
on axes (non-sparse)

Example solution:  
 $\theta = [0.5, 0.3]$   
(both non-zero)

L1 Regularization:



Optimum often ON  
axes (sparse!) ✓

Example solution:  
 $\theta = [0.7, 0.0]$   
(sparse!)

## Mathematical Insight:

### Gradient:

$$\frac{\partial \|\theta\|_1}{\partial \theta_i} = \text{sign}(\theta_i) = \begin{cases} +1 & \text{if } \theta_i > 0 \\ -1 & \text{if } \theta_i < 0 \\ \text{undefined} & \text{if } \theta_i = 0 \end{cases}$$

### Effect:

- Constant push toward zero (independent of  $\theta_i$  magnitude)
- Can push small parameters all the way to exactly zero!

### L2 gradient:

$$\frac{\partial \|\theta\|_2^2}{\partial \theta_i} = 2\theta_i$$

### Effect:

- Push proportional to current value
- Asymptotically approaches zero but never reaches it exactly

## 10.7 TECHNIQUE 2C: Elastic Net (L1 + L2 Combined)

### Definition:

$$R(\theta) = \alpha \|\theta\|_1 + (1 - \alpha) \|\theta\|_2^2$$

### Combines benefits of both:

- L1 → Sparsity (feature selection)
- L2 → Stability (small weights)

### When to Use Each:

| Regularization | Best For                           | Properties                   |
|----------------|------------------------------------|------------------------------|
| L2 (Ridge)     | Correlated features, all relevant  | Shrinks all weights, smooth  |
| L1 (Lasso)     | Sparse features, feature selection | Zeros out irrelevant weights |
| Elastic Net    | Best of both worlds                | Sparse + stable              |

## 10.8 TECHNIQUE 3: Early Stopping

### The Concept:

Monitor validation error during training and STOP when it starts increasing!

### Algorithm:

```
Algorithm: Early Stopping

1. Split data: Train (80%) + Validation (20%)

2. Initialize:
 - best_val_error = ∞
 - patience_counter = 0
 - patience = 10 (epochs to wait)

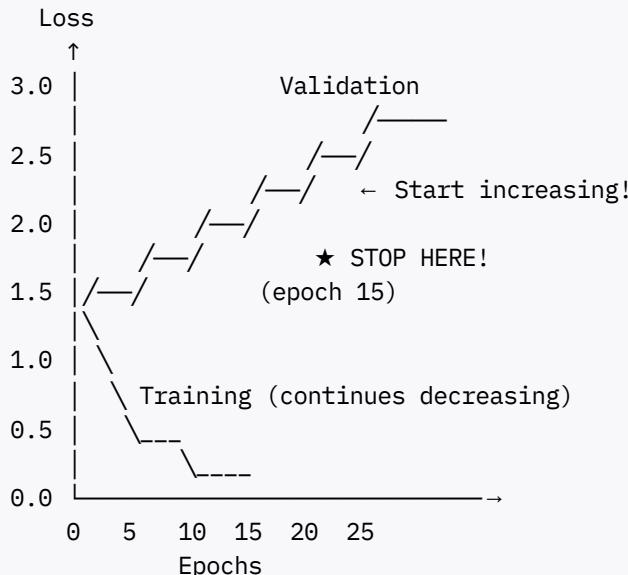
3. FOR each epoch:
 a) Train on training set for one epoch
 b) Evaluate on validation set → val_error

 c) IF val_error < best_val_error:
 - best_val_error = val_error
 - Save current model ✓
 - patience_counter = 0
 ELSE:
 - patience_counter += 1

 d) IF patience_counter ≥ patience:
 STOP TRAINING!
 Return saved model

4. Return best model (from best validation epoch)
```

### Visual: Training with Early Stopping



Without early stopping: Train until epoch 25 → Overfit!  
With early stopping: Stop at epoch 15 → Good generalization ✓

## Why It Works:

Early in training:

- Model learns GENERAL patterns
- Both train & val error decrease
- Good!

Later in training:

- Model starts fitting TRAINING-SPECIFIC noise
- Train error decreases, val error INCREASES
- Overfitting begins!

Early stopping:

- STOP before severe overfitting
- Use model from best validation performance
- Acts as implicit regularization ✓

## Practical Example:

| Epoch | Train Loss | Val Loss | Status            |
|-------|------------|----------|-------------------|
| 1     | 2.50       | 2.55     | Improving         |
| 5     | 1.20       | 1.30     | Improving         |
| 10    | 0.50       | 0.58     | Improving ✓       |
| 15    | 0.20       | 0.55     | Best val! ★       |
| 20    | 0.10       | 0.60     | Val increasing... |
| 25    | 0.05       | 0.75     | Overfitting!      |
| 30    | 0.02       | 0.95     | Severe overfit!   |

Early stopping triggers at epoch 25 (patience=10)  
Returns model from epoch 15 (best validation)

## Advantages:

- ✓ Simple to implement
- ✓ No hyperparameters to tune (except patience)
- ✓ Computationally efficient (stop early!)
- ✓ Works with any model/optimization

## Disadvantages:

- Requires held-out validation set
- May stop too early if validation is noisy
- Need to tune "patience" parameter

## 10.9 TECHNIQUE 4: Dropout (For Neural Networks)

### The Concept:

During training, randomly "drop" (zero out) neurons with probability  $p$ .

### Algorithm:

```
TRAINING (each mini-batch):
FOR each layer:
 1. Generate random mask: $m \sim \text{Bernoulli}(1-p)$
 Example: [1, 0, 1, 1, 0, 1, ...] (0 = drop, 1 = keep)

 2. Multiply activations by mask:
 $h_{\text{dropped}} = h * m$

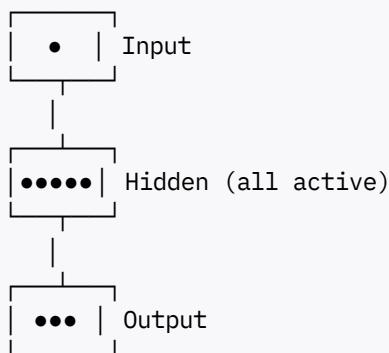
 3. Scale: $h_{\text{dropped}} = h_{\text{dropped}} / (1-p)$
 (keeps expected value same)

 4. Continue forward pass with h_{dropped}
```

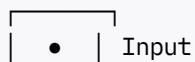
```
TESTING (inference):
Use ALL neurons (no dropout)
Predictions are averaged over all possible masks
```

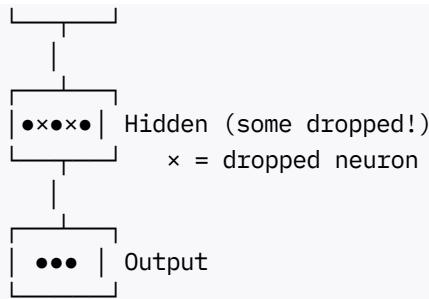
### Visual Representation:

#### NORMAL NETWORK:



#### WITH DROPOUT ( $p=0.5$ ):





Different mask each mini-batch!

## Why It Prevents Overfitting:

### 1. Ensemble Effect:

Each mini-batch trains a different "sub-network"  
Final model = average of many sub-networks  
Like training an ensemble!  
→ Better generalization

### 2. Reduces Co-Adaptation:

Without dropout:  
Neurons can rely on specific other neurons  
→ Creates fragile dependencies  
→ Overfits to training data

With dropout:  
Neurons can't rely on others (might be dropped!)  
→ Must learn robust features  
→ Better generalization ✓

### 3. Acts as Noise Injection:

Dropout adds noise during training  
→ Prevents exact memorization  
→ Forces model to learn robust patterns

## Mathematical Formulation:

### Forward pass during training:

$$\tilde{h}_i = \begin{cases} 0 & \text{with probability } p \\ \frac{h_i}{1-p} & \text{with probability } 1-p \end{cases}$$

### Expected value:

$$\mathbb{E}[\tilde{h}_i] = 0 \cdot p + \frac{h_i}{1-p} \cdot (1-p) = h_i$$

(Scaling ensures expected value matches original!)

### Typical Hyperparameters:

```
p = 0.5 for hidden layers (most common)
p = 0.2 for input layer (if used)
p = 0.0 for output layer (no dropout)
```

### Example Results:

MNIST Classification:

Without Dropout:

Train accuracy: 99.8%  
Test accuracy: 97.2%  
Gap: 2.6%

With Dropout (p=0.5):

Train accuracy: 98.5%  
Test accuracy: 98.1%  
Gap: 0.4% ✓

Slightly worse training, but BETTER test!  
Much smaller gap → Better generalization ✓

## 10.10 TECHNIQUE 5: Data Augmentation

### The Concept:

Artificially increase training data size by creating modified versions of existing data.

### Why It Helps:

More data → Better statistics → Lower variance → Less overfitting!

### From bias-variance perspective:

$$\text{Variance} \propto \frac{1}{m}$$

Larger  $m$  → Lower variance ✓

## Examples for Images:

### Original Image:

[Dog photo]

### Augmented Versions:

1. Horizontal flip: [Flipped dog]
2. Rotation ( $\pm 15^\circ$ ): [Rotated dog]
3. Crop & resize: [Cropped dog]
4. Color jitter: [Different brightness]
5. Add noise: [Noisy dog]
6. Zoom in/out: [Zoomed dog]

**Result:** 1 image  $\rightarrow$  10+ augmented versions!

### Mathematical View:

**Original dataset:**  $D = \{x^{(1)}, \dots, x^{(m)}\}$

**Augmented dataset:**  $D_{\text{aug}} = \{x^{(1)}, T_1(x^{(1)}), T_2(x^{(1)}), \dots\}$

Where  $T_k$  are augmentation transformations.

**Effective size:**  $m_{\text{eff}} = m \times k$  where  $k$  = augmentations per sample

### Example Impact:

CIFAR-10 Image Classification:

Original data only ( $m=50,000$ ):  
Test accuracy: 75%

With augmentation ( $m_{\text{eff}} \approx 250,000$ ):  
Test accuracy: 85% ✓

10% improvement from augmentation alone!

## Important Considerations:

### Valid augmentations:

- ✓ Horizontal flip for dogs (dogs can face either way)
- ✗ Vertical flip for dogs (dogs don't walk upside-down!)
- ✓ Rotation  $\pm 15^\circ$  for digits (handwriting varies)
- ✗ Rotation  $180^\circ$  for digits (would change meaning!)

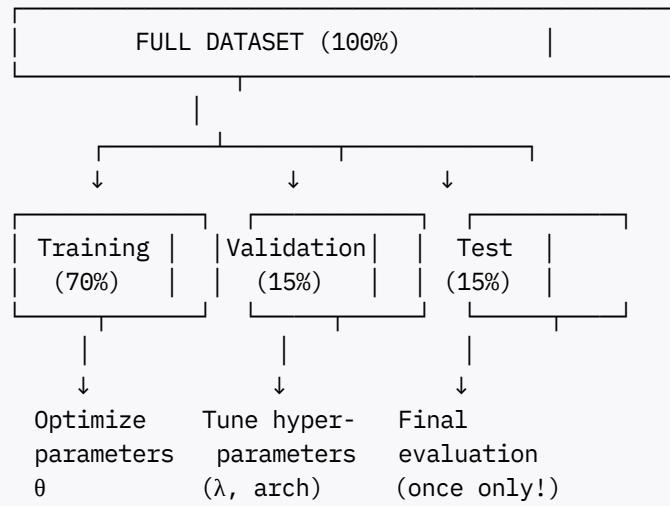
**Domain knowledge is crucial!**

## 10.11 TECHNIQUE 6: Validation Set for Model Selection

From Slide 23:

"Evaluate generalization performance on a held-out validation set"

**Three-Way Split (Standard):**



**Workflow:**

1. TRAINING PHASE:  
FOR each hyperparameter setting:
  - a) Train model on TRAINING set
  - b) Evaluate on VALIDATION set
  - c) Record validation performance

Select hyperparameters with best validation performance

2. FINAL EVALUATION:
  - a) Retrain with best hyperparameters on Train+Val
  - b) Evaluate ONCE on TEST set
  - c) Report test performance

3. NEVER:
  - Use test set during hyperparameter tuning
  - Peek at test set during development
  - Report best of multiple test evaluations

## Example: Hyperparameter Tuning

Trying different  $\lambda$  values for L2 regularization:

$\lambda = 0.001$ :  
Val accuracy: 92.3%

$\lambda = 0.01$ :  
Val accuracy: 94.5% ← Best!

$\lambda = 0.1$ :  
Val accuracy: 91.8%

$\lambda = 1.0$ :  
Val accuracy: 88.2%

SELECTED:  $\lambda = 0.01$

Final test evaluation: 94.2% ✓  
(Close to validation, confirms good generalization)

## Why Three Sets?

TWO SETS (Train + Test):

- Problem: Tune hyperparameters on test set
  - "Overfits" to test set through hyperparameter selection!
  - Test performance optimistically biased

THREE SETS (Train + Val + Test):

- ✓ Train on training set
- ✓ Tune on validation set (can look many times)
- ✓ Final eval on test set (look ONCE)
- ✓ Test set gives unbiased estimate ✓

## 10.12 Cross-Validation (When Data is Limited)

### The Problem:

Small dataset (e.g.,  $m=1000$ )

- Training set only 700 samples
- Not enough to train well!
- Validation set only 150 samples
- Not enough to estimate reliably!

## Solution: K-Fold Cross-Validation

### Algorithm:

K-FOLD CROSS-VALIDATION (K=5 example):

Split data into 5 equal folds:

|                |  |                |  |                |  |                |  |                |  |  |
|----------------|--|----------------|--|----------------|--|----------------|--|----------------|--|--|
| F <sub>1</sub> |  | F <sub>2</sub> |  | F <sub>3</sub> |  | F <sub>4</sub> |  | F <sub>5</sub> |  |  |
|----------------|--|----------------|--|----------------|--|----------------|--|----------------|--|--|

Iteration 1: Val=F<sub>1</sub>, Train=F<sub>2</sub>+F<sub>3</sub>+F<sub>4</sub>+F<sub>5</sub> → Val error<sub>1</sub>

Iteration 2: Val=F<sub>2</sub>, Train=F<sub>1</sub>+F<sub>3</sub>+F<sub>4</sub>+F<sub>5</sub> → Val error<sub>2</sub>

Iteration 3: Val=F<sub>3</sub>, Train=F<sub>1</sub>+F<sub>2</sub>+F<sub>4</sub>+F<sub>5</sub> → Val error<sub>3</sub>

Iteration 4: Val=F<sub>4</sub>, Train=F<sub>1</sub>+F<sub>2</sub>+F<sub>3</sub>+F<sub>5</sub> → Val error<sub>4</sub>

Iteration 5: Val=F<sub>5</sub>, Train=F<sub>1</sub>+F<sub>2</sub>+F<sub>3</sub>+F<sub>4</sub> → Val error<sub>5</sub>

Average: Val error = (error<sub>1</sub> + ... + error<sub>5</sub>) / 5

Every sample used for validation exactly once!

### Benefits:

- ✓ Uses all data for both training and validation
- ✓ More reliable performance estimate
- ✓ Reduces variance in estimate

### Cost:

- ✗ Expensive (train K models)
- ✗ Only practical for small-medium datasets

## 10.13 Complete Anti-Overfitting Strategy

### Comprehensive Approach:

#### STEP 1: START SIMPLE

- Small network architecture
- No regularization initially
- Establish baseline

#### STEP 2: INCREASE CAPACITY IF NEEDED

- If high training error → Add capacity
- Larger network
- More layers/neurons

#### STEP 3: ADD REGULARIZATION

- L2 regularization ( $\lambda = 0.01$  start)
- Dropout ( $p = 0.5$ )

- └ Data augmentation

STEP 4: TUNE WITH VALIDATION

- ├ Try different  $\lambda$  values
- ├ Try different dropout rates
- ├ Try different architectures
- └ Select best based on validation

STEP 5: EARLY STOPPING

- ├ Monitor validation loss
- ├ Stop when starts increasing
- └ Use best checkpoint

STEP 6: FINAL EVALUATION

- └ Test set (once only!)

### Practical Guidelines:

| Symptom               | Diagnosis           | Solution           |
|-----------------------|---------------------|--------------------|
| High train, high val  | Underfitting        | Increase capacity  |
| Low train, high val   | Overfitting         | Add regularization |
| Gap growing over time | Starting to overfit | Early stopping     |
| Good train, good val  | Good fit            | Done! ✓            |

### 10.14 Summary Table: All Techniques

| Technique                | Type              | How It Works            | Pros                 | Cons                   |
|--------------------------|-------------------|-------------------------|----------------------|------------------------|
| <b>Smaller Network</b>   | Hard constraint   | Fewer parameters        | Simple, fast         | May underfit           |
| <b>Weight Sharing</b>    | Hard constraint   | Reuse parameters        | Huge reduction       | Architecture-specific  |
| <b>L2 Regularization</b> | Soft penalty      | Penalize large weights  | Universal, effective | Need to tune $\lambda$ |
| <b>L1 Regularization</b> | Soft penalty      | Sparse weights          | Feature selection    | Less common for NNs    |
| <b>Early Stopping</b>    | Training strategy | Stop when val increases | Simple, no tuning    | Needs validation set   |
| <b>Dropout</b>           | Training strategy | Random neuron dropping  | Very effective       | Slows training         |
| <b>Data Augmentation</b> | Data strategy     | More training samples   | Often huge gains     | Domain-specific        |
| <b>Validation Set</b>    | Evaluation        | Tune on separate data   | Reliable selection   | Reduces training data  |

## 10.15 Practical Example: Complete Pipeline

### Scenario: Image Classification (CIFAR-10)

**Task:** Classify  $32 \times 32$  color images into 10 classes

**Data:** 50,000 train, 10,000 test

### Iteration 1: Baseline (No Regularization)

Model: CNN [Conv32 → Conv64 → FC512 → FC10]

Parameters: 1.2M

Regularization: None

Results after 100 epochs:

Train accuracy: 99.5%

Val accuracy: 70.2%

Gap: 29.3%

Diagnosis: SEVERE OVERFITTING ✗

### Iteration 2: Add L2 Regularization

Model: Same architecture

Parameters: 1.2M

Regularization: L2 with  $\lambda=0.001$

Results:

Train accuracy: 95.2%

Val accuracy: 78.5%

Gap: 16.7%

Improvement: +8.3% on validation ✓

Still some overfitting...

### Iteration 3: Add Dropout

Model: Same + Dropout( $p=0.5$ ) after each FC layer

Regularization: L2 ( $\lambda=0.001$ ) + Dropout ( $p=0.5$ )

Results:

Train accuracy: 92.8%

Val accuracy: 82.1%

Gap: 10.7%

Improvement: +3.6% more ✓

Getting better!

## Iteration 4: Add Data Augmentation

Model: Same  
Regularization: L2 + Dropout  
Data: Random crop, horizontal flip, color jitter

Results:  
Train accuracy: 90.5%  
Val accuracy: 85.3%  
Gap: 5.2%

Improvement: +3.2% more ✓✓  
Much better!

## Iteration 5: Early Stopping

Model: Same  
Regularization: L2 + Dropout + Data Aug  
Training: Stop at epoch 75 (best val)

Results:  
Train accuracy: 89.2%  
Val accuracy: 86.0%  
Gap: 3.2%

Improvement: +0.7% more ✓✓✓  
Excellent generalization!

## Final Test Evaluation:

Test accuracy: 85.8%  
Close to validation → Confirms good generalization! ✓✓✓  
Summary:  
Baseline: 70.2%  
Final: 85.8%  
Gain: +15.6% through regularization techniques!

## ✓ KEY TAKEAWAYS - TOPIC 10

### Main Concepts:

1. **From Slide 23:** Three strategies to avoid overfitting
  - Hard constraints (smaller models, weight sharing)
  - Soft regularization (Occam's Razor, penalty terms)
  - Validation-based selection

## 2. Hard Constraints:

- **Smaller networks:** Fewer parameters → Less overfitting
- **Weight sharing:** Reuse parameters → Massive reduction (CNNs!)

## 3. Soft Regularization:

- **L2 (Ridge):** Penalize large weights, smooth solutions
- **L1 (Lasso):** Sparse solutions, feature selection
- Modified objective: Loss +  $\lambda \times$  Regularization

## 4. Training Strategies:

- **Early stopping:** Stop when validation error increases
- **Dropout:** Randomly drop neurons during training
- **Data augmentation:** Artificially increase training data

## 5. Model Selection:

- Use validation set for hyperparameter tuning
- Test set for final evaluation only (once!)
- Cross-validation when data is limited

## 6. Combined Approach:

Best results come from COMBINING multiple techniques:

- ✓ L2 regularization
- ✓ Dropout
- ✓ Data augmentation
- ✓ Early stopping
- ✓ Validation-based selection

## Practical Workflow:

1. Start simple → Baseline
2. Increase capacity if underfitting
3. Add regularization if overfitting
4. Tune with validation set
5. Apply early stopping
6. Final evaluation on test set

## Key Equation:

$$\text{Regularized Objective} = -\sum_{j=1}^m \log P_\theta(x^{(j)}) + \lambda \cdot R(\theta)$$

## LECTURE 4 COMPLETE!

### What We've Learned:

#### TOPIC 1-2: Problem Setup

- Learning generative models from data

#### TOPIC 3-5: Theoretical Foundation

- KL-divergence
- Expected log-likelihood
- Maximum likelihood principle

#### TOPIC 6-7: Practical Algorithms

- Maximum likelihood estimation
- Gradient descent & SGD

#### TOPIC 8-9: Understanding Challenges

- Overfitting
- Bias-variance trade-off

#### TOPIC 10: Solutions

- Regularization techniques ✓

### The Complete Learning Pipeline:

DATA → MLE OBJECTIVE → SGD OPTIMIZATION → REGULARIZATION → VALIDATION → TRAINED MODEL

Congratulations! You now have comprehensive notes on Maximum Likelihood Learning! ☺

That completes Lecture 4! These notes cover all topics in detail with mathematical rigor, practical examples, and visual explanations. Ready for the next lecture? ☺

1. ML\_Learning.pdf
2. ML\_Learning.pdf
3. Introduction.pdf
4. Background.pdf
5. Reference Material Probability Review.pdf
6. Reference Material Linear Algebra Review.pdf
7. AR\_Models.pdf