

# CS498: Algorithmic Engineering

## Lecture 21: The Learning Problem & Stochastic Gradient Descent

Elfarouk Harb

University of Illinois Urbana-Champaign

Week 12

# Outline

- 1 Where We Are
- 2 The Learning Problem
- 3 Stochastic Gradient Descent
- 4 Adam, AdamW & Learning Rate Schedules
- 5 Putting It All Together: Lasso Regression
- 6 Summary

- 1 Where We Are
- 2 The Learning Problem
- 3 Stochastic Gradient Descent
- 4 Adam, AdamW & Learning Rate Schedules
- 5 Putting It All Together: Lasso Regression
- 6 Summary

# Where We Are in the Course

## **Part I: Discrete Optimization** (L1–9)

LP, IP, branch-and-bound, ...

*Philosophy:* structured discrete problems → exact solvers.

# Where We Are in the Course

## **Part I: Discrete Optimization** (L1–9)

LP, IP, branch-and-bound, ...

*Philosophy:* structured discrete problems → exact solvers.

## **Part II: Continuous Opt & Metaheuristics** (L10–16)

GD, convex optimization, Lagrangian, KKT, GA.

*Philosophy:* convexity gives guarantees; metaheuristics when it doesn't.

# Where We Are in the Course

## **Part I: Discrete Optimization** (L1–9)

LP, IP, branch-and-bound, ...

*Philosophy:* structured discrete problems → exact solvers.

## **Part II: Continuous Opt & Metaheuristics** (L10–16)

GD, convex optimization, Lagrangian, KKT, GA.

*Philosophy:* convexity gives guarantees; metaheuristics when it doesn't.

## **Part III: SAT/SMT** (L17–20)

SAT, CDCL, SMT, Z3, formal verification.

*Philosophy:* encode constraints, let the solver search.

# Where We Are in the Course

## **Part I: Discrete Optimization** (L1–9)

LP, IP, branch-and-bound, ...

*Philosophy:* structured discrete problems → exact solvers.

## **Part II: Continuous Opt & Metaheuristics** (L10–16)

GD, convex optimization, Lagrangian, KKT, GA.

*Philosophy:* convexity gives guarantees; metaheuristics when it doesn't.

## **Part III: SAT/SMT** (L17–20)

SAT, CDCL, SMT, Z3, formal verification.

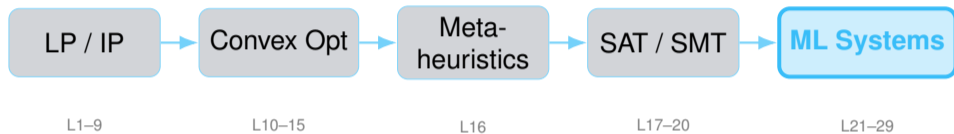
*Philosophy:* encode constraints, let the solver search.

## **Part IV: Machine Learning Systems** (L21–29)

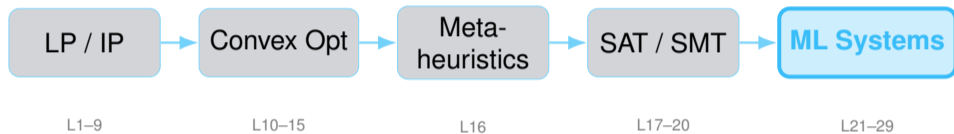
**Starts today.**

*Philosophy:* what happens when the objective itself is **unknown**?

# The Course Arc

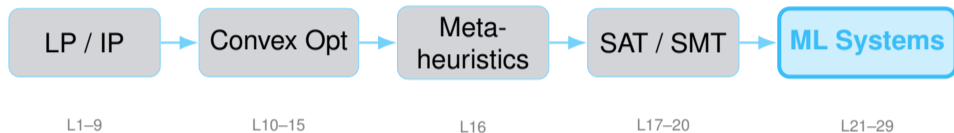


# The Course Arc



Every tool we've built solves a **known** objective.

# The Course Arc



Every tool we've built solves a **known** objective.  
In ML, the objective **comes from data**.

# What Changes Today?

**Before:** you are given  $f(\mathbf{x})$  explicitly. Minimize it or satisfy constraints.

# What Changes Today?

**Before:** you are given  $f(\mathbf{x})$  explicitly. Minimize it or satisfy constraints.

**Now:** you have **data**  $\{(x_i, y_i)\}_{i=1}^n$ , and we don't know the true relationship  $y = g(x)$ .

# What Changes Today?

**Before:** you are given  $f(\mathbf{x})$  explicitly. Minimize it or satisfy constraints.

**Now:** you have **data**  $\{(x_i, y_i)\}_{i=1}^n$ , and we don't know the true relationship  $y = g(x)$ .

You **choose** two things:

- A **model**  $f_\theta$ : a parameterized function that maps inputs to predictions. Examples: linear model  $f_\theta(\mathbf{x}) = \theta^\top \mathbf{x}$ , polynomial, or (later) a neural network.

# What Changes Today?

**Before:** you are given  $f(\mathbf{x})$  explicitly. Minimize it or satisfy constraints.

**Now:** you have **data**  $\{(x_i, y_i)\}_{i=1}^n$ , and we don't know the true relationship  $y = g(x)$ .

You **choose** two things:

- A **model**  $f_\theta$ : a parameterized function that maps inputs to predictions.  
Examples: linear model  $f_\theta(\mathbf{x}) = \theta^\top \mathbf{x}$ , polynomial, or (later) a neural network.
- A **loss function**  $L$ : measures how bad a prediction is for one  $\theta$ .  
Examples: squared error  $L(\hat{y}, y) = (\hat{y} - y)^2$ , absolute error  $|\hat{y} - y|$ .

# What Changes Today?

**Before:** you are given  $f(\mathbf{x})$  explicitly. Minimize it or satisfy constraints.

**Now:** you have **data**  $\{(x_i, y_i)\}_{i=1}^n$ , and we don't know the true relationship  $y = g(x)$ .

You **choose** two things:

- A **model**  $f_\theta$ : a parameterized function that maps inputs to predictions.  
Examples: linear model  $f_\theta(\mathbf{x}) = \theta^\top \mathbf{x}$ , polynomial, or (later) a neural network.
- A **loss function**  $L$ : measures how bad a prediction is for one  $\theta$ .  
Examples: squared error  $L(\hat{y}, y) = (\hat{y} - y)^2$ , absolute error  $|\hat{y} - y|$ .

Then optimize:

$$\min_{\theta} \frac{1}{n} \sum_{i=1}^n L(f_\theta(x_i), y_i)$$

# What Changes Today?

**Before:** you are given  $f(\mathbf{x})$  explicitly. Minimize it or satisfy constraints.

**Now:** you have **data**  $\{(x_i, y_i)\}_{i=1}^n$ , and we don't know the true relationship  $y = g(x)$ .

You **choose** two things:

- A **model**  $f_\theta$ : a parameterized function that maps inputs to predictions.  
Examples: linear model  $f_\theta(\mathbf{x}) = \theta^\top \mathbf{x}$ , polynomial, or (later) a neural network.
- A **loss function**  $L$ : measures how bad a prediction is for one  $\theta$ .  
Examples: squared error  $L(\hat{y}, y) = (\hat{y} - y)^2$ , absolute error  $|\hat{y} - y|$ .

Then optimize:

$$\min_{\theta} \frac{1}{n} \sum_{i=1}^n L(f_\theta(x_i), y_i)$$

**The question is...** how do you choose  $f_\theta$ ? How do you optimize when  $n$  is enormous? And why does any of it generalize?

- 1 Where We Are
- 2 The Learning Problem**
- 3 Stochastic Gradient Descent
- 4 Adam, AdamW & Learning Rate Schedules
- 5 Putting It All Together: Lasso Regression
- 6 Summary

# A Concrete Problem

You measure  $n = 100$  noisy data points from  $y = \sin(x) + \varepsilon$ .

# A Concrete Problem

You measure  $n = 100$  noisy data points from  $y = \sin(x) + \varepsilon$ .

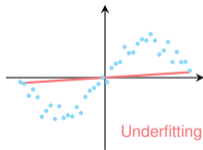
**Goal:** fit a polynomial  $\hat{y} = a_0 + a_1x + a_2x^2 + \dots + a_dx^d$ . Which degree  $d$ ?

# A Concrete Problem

You measure  $n = 100$  noisy data points from  $y = \sin(x) + \varepsilon$ .

**Goal:** fit a polynomial  $\hat{y} = a_0 + a_1x + a_2x^2 + \dots + a_dx^d$ . Which degree  $d$ ?

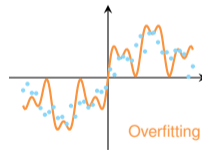
Degree 1



Degree 4



Degree 20



# Which Model is Better?

## Degree 1 (straight line):

- Low complexity, poor fit.
- **High bias**: the model can't capture the curve at all.

# Which Model is Better?

## Degree 1 (straight line):

- Low complexity, poor fit.
- **High bias**: the model can't capture the curve at all.

## Degree 20 (wiggly polynomial):

- Perfect fit on training data (near-zero training loss).
- **High variance**: terrible on new data. Memorized the noise.

# Which Model is Better?

## Degree 1 (straight line):

- Low complexity, poor fit.
- **High bias**: the model can't capture the curve at all.

## Degree 20 (wiggly polynomial):

- Perfect fit on training data (near-zero training loss).
- **High variance**: terrible on new data. Memorized the noise.

## Key insight

We care about performance on **unseen** data: this is called **generalization**. A model that memorizes training data is useless.

# The Bias–Variance Decomposition

For any model, the **expected test error** decomposes as:

$$\underbrace{\mathbb{E}[(y - \hat{y})^2]}_{\text{Expected test error}} = \underbrace{\text{Bias}^2}_{\text{error from wrong assumptions}} + \underbrace{\text{Variance}}_{\text{sensitivity to training set}} + \underbrace{\sigma^2}_{\text{irreducible noise}}$$

# The Bias–Variance Decomposition

For any model, the **expected test error** decomposes as:

$$\underbrace{\mathbb{E}[(y - \hat{y})^2]}_{\text{Expected test error}} = \underbrace{\text{Bias}^2}_{\text{error from wrong assumptions}} + \underbrace{\text{Variance}}_{\text{sensitivity to training set}} + \underbrace{\sigma^2}_{\text{irreducible noise}}$$

- **Bias:** a degree-1 polynomial can never capture  $\sin(x)$ , no matter how much data.

# The Bias–Variance Decomposition

For any model, the **expected test error** decomposes as:

$$\underbrace{\mathbb{E}[(y - \hat{y})^2]}_{\text{Expected test error}} = \underbrace{\text{Bias}^2}_{\text{error from wrong assumptions}} + \underbrace{\text{Variance}}_{\text{sensitivity to training set}} + \underbrace{\sigma^2}_{\text{irreducible noise}}$$

- **Bias**: a degree-1 polynomial can never capture  $\sin(x)$ , no matter how much data.
- **Variance**: a degree-20 polynomial changes wildly depending on which data points you sample.

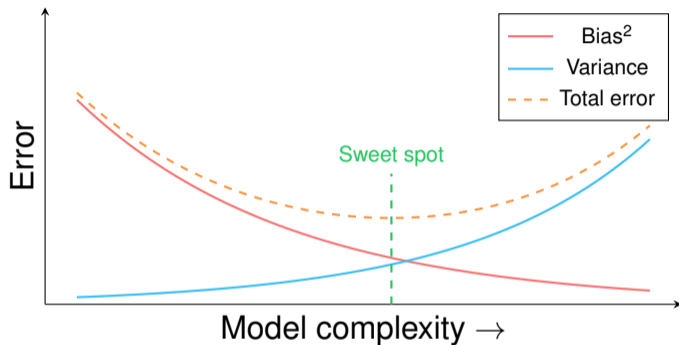
# The Bias–Variance Decomposition

For any model, the **expected test error** decomposes as:

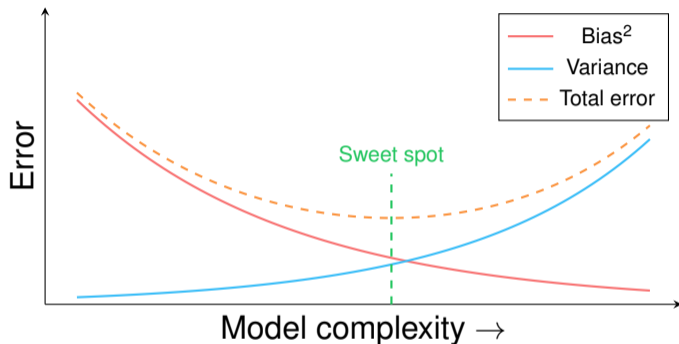
$$\underbrace{\mathbb{E}[(y - \hat{y})^2]}_{\text{Expected test error}} = \underbrace{\text{Bias}^2}_{\text{error from wrong assumptions}} + \underbrace{\text{Variance}}_{\text{sensitivity to training set}} + \underbrace{\sigma^2}_{\text{irreducible noise}}$$

- **Bias**: a degree-1 polynomial can never capture  $\sin(x)$ , no matter how much data.
- **Variance**: a degree-20 polynomial changes wildly depending on which data points you sample.
- **Noise**  $\sigma^2$ : no model can do better than this (the data is inherently noisy).

# The Bias–Variance Tradeoff



# The Bias–Variance Tradeoff



**The deal:** too simple → high bias; too complex → high variance. The sweet spot minimizes total error.

# Empirical Risk Minimization

How do we formalize “fit the data”?

# Empirical Risk Minimization

How do we formalize “fit the data”?

## Choose:

- A model family  $f_\theta$  (e.g., degree- $d$  polynomial, neural net).

# Empirical Risk Minimization

How do we formalize “fit the data”?

## Choose:

- A model family  $f_\theta$  (e.g., degree- $d$  polynomial, neural net).
- A loss function  $L$  (e.g., squared error  $L(\hat{y}, y) = (\hat{y} - y)^2$ ).

# Empirical Risk Minimization

How do we formalize “fit the data”?

## Choose:

- A model family  $f_\theta$  (e.g., degree- $d$  polynomial, neural net).
- A loss function  $L$  (e.g., squared error  $L(\hat{y}, y) = (\hat{y} - y)^2$ ).

## Solve:

$$\min_{\theta} \underbrace{\frac{1}{n} \sum_{i=1}^n L(f_\theta(x_i), y_i)}_{\text{Empirical Risk } \hat{R}(\theta)}$$

# Empirical Risk Minimization

How do we formalize “fit the data”?

## Choose:

- A model family  $f_\theta$  (e.g., degree- $d$  polynomial, neural net).
- A loss function  $L$  (e.g., squared error  $L(\hat{y}, y) = (\hat{y} - y)^2$ ).

## Solve:

$$\min_{\theta} \underbrace{\frac{1}{n} \sum_{i=1}^n L(f_\theta(x_i), y_i)}_{\text{Empirical Risk } \hat{R}(\theta)}$$

**In words:** pick the parameters  $\theta$  that minimize average loss on training data.

# Empirical Risk Minimization

How do we formalize “fit the data”?

## Choose:

- A model family  $f_\theta$  (e.g., degree- $d$  polynomial, neural net).
- A loss function  $L$  (e.g., squared error  $L(\hat{y}, y) = (\hat{y} - y)^2$ ).

## Solve:

$$\min_{\theta} \underbrace{\frac{1}{n} \sum_{i=1}^n L(f_\theta(x_i), y_i)}_{\text{Empirical Risk } \hat{R}(\theta)}$$

**In words:** pick the parameters  $\theta$  that minimize average loss on training data.

**But...**

Minimizing training loss perfectly gives us degree 20. We need something more.

# Train / Test Split

**The question is...** how do we measure generalization without deploying the model?

# Train / Test Split

**The question is...** how do we measure generalization without deploying the model?

**Simple answer:** split the data.

- **Training set (80%):** optimize  $\theta$  on this.

# Train / Test Split

**The question is...** how do we measure generalization without deploying the model?

**Simple answer:** split the data.

- **Training set** (80%): optimize  $\theta$  on this.
- **Test set** (20%): evaluate final performance here. **Never** train on it.

# Train / Test Split

**The question is...** how do we measure generalization without deploying the model?

**Simple answer:** split the data.

- **Training set** (80%): optimize  $\theta$  on this.
- **Test set** (20%): evaluate final performance here. **Never** train on it.

The test set is a **proxy** for real-world performance.

# Train / Test Split

**The question is...** how do we measure generalization without deploying the model?

**Simple answer:** split the data.

- **Training set** (80%): optimize  $\theta$  on this.
- **Test set** (20%): evaluate final performance here. **Never** train on it.

The test set is a **proxy** for real-world performance.

**But:** what about **hyperparameters**? These are choices you make *before* training:

- Learning rate  $\eta$ , regularization strength  $\lambda$ , model architecture (number of layers, etc.).

# Train / Test Split

**The question is...** how do we measure generalization without deploying the model?

**Simple answer:** split the data.

- **Training set** (80%): optimize  $\theta$  on this.
- **Test set** (20%): evaluate final performance here. **Never** train on it.

The test set is a **proxy** for real-world performance.

**But:** what about **hyperparameters**? These are choices you make *before* training:

- Learning rate  $\eta$ , regularization strength  $\lambda$ , model architecture (number of layers, etc.).
- If you tune hyperparameters on the test set, it's no longer a fair evaluation.

# Train / Test Split

**The question is...** how do we measure generalization without deploying the model?

**Simple answer:** split the data.

- **Training set** (80%): optimize  $\theta$  on this.
- **Test set** (20%): evaluate final performance here. **Never** train on it.

The test set is a **proxy** for real-world performance.

**But:** what about **hyperparameters**? These are choices you make *before* training:

- Learning rate  $\eta$ , regularization strength  $\lambda$ , model architecture (number of layers, etc.).
- If you tune hyperparameters on the test set, it's no longer a fair evaluation.
- Solution: **validation set**, a third split used for tuning.
- Train on train, tune on validation, report on test.

# Regularization = Constrained Optimization

Previously, we added constraints to optimization problems to penalize complexity:  
**Regularization.**

# Regularization = Constrained Optimization

Previously, we added constraints to optimization problems to penalize complexity:  
**Regularization.**

**L2 regularization** (ridge regression):

$$\min_{\theta} \frac{1}{n} \sum_{i=1}^n L(f_{\theta}(x_i), y_i) + \lambda \|\theta\|^2$$

# Regularization = Constrained Optimization

Previously, we added constraints to optimization problems to penalize complexity:  
**Regularization.**

**L2 regularization** (ridge regression):

$$\min_{\theta} \frac{1}{n} \sum_{i=1}^n L(f_{\theta}(x_i), y_i) + \lambda \|\theta\|^2$$

- $\lambda$  large  $\rightarrow$  strong penalty on big weights  $\rightarrow$  simpler model.

# Regularization = Constrained Optimization

Previously, we added constraints to optimization problems to penalize complexity:  
**Regularization.**

**L2 regularization** (ridge regression):

$$\min_{\theta} \frac{1}{n} \sum_{i=1}^n L(f_{\theta}(x_i), y_i) + \lambda \|\theta\|^2$$

- $\lambda$  large  $\rightarrow$  strong penalty on big weights  $\rightarrow$  simpler model.
- $\lambda = 0$   $\rightarrow$  no penalty  $\rightarrow$  back to plain ERM.

# Regularization = Constrained Optimization

Previously, we added constraints to optimization problems to penalize complexity:  
**Regularization.**

**L2 regularization** (ridge regression):

$$\min_{\theta} \frac{1}{n} \sum_{i=1}^n L(f_{\theta}(x_i), y_i) + \lambda \|\theta\|^2$$

- $\lambda$  large  $\rightarrow$  strong penalty on big weights  $\rightarrow$  simpler model.
- $\lambda = 0$   $\rightarrow$  no penalty  $\rightarrow$  back to plain ERM.

**The deal:** accept slightly worse training loss for **much better generalization.**

# Classification and Cross-Entropy

So far our running example is regression ( $y \in \mathbb{R}$ ). But what about **classification** ( $y \in \{0, 1\}$ )?

# Classification and Cross-Entropy

So far our running example is regression ( $y \in \mathbb{R}$ ). But what about **classification** ( $y \in \{0, 1\}$ )?

**Idea:** output a **probability**  $\hat{y} = p(y = 1 \mid x) \in [0, 1]$ .

# Classification and Cross-Entropy

So far our running example is regression ( $y \in \mathbb{R}$ ). But what about **classification** ( $y \in \{0, 1\}$ )?

**Idea:** output a **probability**  $\hat{y} = p(y = 1 \mid x) \in [0, 1]$ .

Squared error  $(\hat{y} - y)^2$  works, but does not penalize confident wrong predictions as sharply cross-entropy. We want a loss that **heavily penalizes confident wrong predictions**.

# Classification and Cross-Entropy

So far our running example is regression ( $y \in \mathbb{R}$ ). But what about **classification** ( $y \in \{0, 1\}$ )?

**Idea:** output a **probability**  $\hat{y} = p(y = 1 | x) \in [0, 1]$ .

Squared error  $(\hat{y} - y)^2$  works, but does not penalize confident wrong predictions as sharply cross-entropy. We want a loss that **heavily penalizes confident wrong predictions**.

**Cross-entropy loss** (binary):

$$L(\hat{y}, y) = -[y \log \hat{y} + (1 - y) \log(1 - \hat{y})]$$

# Classification and Cross-Entropy

So far our running example is regression ( $y \in \mathbb{R}$ ). But what about **classification** ( $y \in \{0, 1\}$ )?

**Idea:** output a **probability**  $\hat{y} = p(y = 1 | x) \in [0, 1]$ .

Squared error  $(\hat{y} - y)^2$  works, but does not penalize confident wrong predictions as sharply cross-entropy. We want a loss that **heavily penalizes confident wrong predictions**.

**Cross-entropy loss** (binary):

$$L(\hat{y}, y) = -[y \log \hat{y} + (1 - y) \log(1 - \hat{y})]$$

- If  $y = 1$  and  $\hat{y} \rightarrow 0$ :  $L \rightarrow +\infty$ . Confident and wrong  $\Rightarrow$  huge penalty.

# Classification and Cross-Entropy

So far our running example is regression ( $y \in \mathbb{R}$ ). But what about **classification** ( $y \in \{0, 1\}$ )?

**Idea:** output a **probability**  $\hat{y} = p(y = 1 \mid x) \in [0, 1]$ .

Squared error  $(\hat{y} - y)^2$  works, but does not penalize confident wrong predictions as sharply cross-entropy. We want a loss that **heavily penalizes confident wrong predictions**.

**Cross-entropy loss** (binary):

$$L(\hat{y}, y) = -[y \log \hat{y} + (1 - y) \log(1 - \hat{y})]$$

- If  $y = 1$  and  $\hat{y} \rightarrow 0$ :  $L \rightarrow +\infty$ . Confident and wrong  $\Rightarrow$  huge penalty.
- If  $y = 1$  and  $\hat{y} = 0.99$ :  $L \approx 0.01$ . Confident and right  $\Rightarrow$  tiny penalty.

# Classification and Cross-Entropy

So far our running example is regression ( $y \in \mathbb{R}$ ). But what about **classification** ( $y \in \{0, 1\}$ )?

**Idea:** output a **probability**  $\hat{y} = p(y = 1 | x) \in [0, 1]$ .

Squared error  $(\hat{y} - y)^2$  works, but does not penalize confident wrong predictions as sharply cross-entropy. We want a loss that **heavily penalizes confident wrong predictions**.

**Cross-entropy loss** (binary):

$$L(\hat{y}, y) = -[y \log \hat{y} + (1 - y) \log(1 - \hat{y})]$$

- If  $y = 1$  and  $\hat{y} \rightarrow 0$ :  $L \rightarrow +\infty$ . Confident and wrong  $\Rightarrow$  huge penalty.
- If  $y = 1$  and  $\hat{y} = 0.99$ :  $L \approx 0.01$ . Confident and right  $\Rightarrow$  tiny penalty.

**Key insight:** cross-entropy comes from information theory. It measures how “surprised” the model is by the true label.

# Common Loss Functions

<b>Task</b>	<b>Loss</b>	<b>Formula</b>
Regression	Squared error	$L = (\hat{y} - y)^2$

# Common Loss Functions

<b>Task</b>	<b>Loss</b>	<b>Formula</b>
Regression	Squared error	$L = (\hat{y} - y)^2$
Regression	Absolute error	$L =  \hat{y} - y $

# Common Loss Functions

Task	Loss	Formula
Regression	Squared error	$L = (\hat{y} - y)^2$
Regression	Absolute error	$L =  \hat{y} - y $
Classification	Cross-entropy	$L = -y \log \hat{y} - (1 - y) \log(1 - \hat{y})$

# Common Loss Functions

Task	Loss	Formula
Regression	Squared error	$L = (\hat{y} - y)^2$
Regression	Absolute error	$L =  \hat{y} - y $
Classification	Cross-entropy	$L = -y \log \hat{y} - (1 - y) \log(1 - \hat{y})$
Classification	Hinge (SVM)	$L = \max(0, 1 - y\hat{y})$

# Common Loss Functions

Task	Loss	Formula
Regression	Squared error	$L = (\hat{y} - y)^2$
Regression	Absolute error	$L =  \hat{y} - y $
Classification	Cross-entropy	$L = -y \log \hat{y} - (1-y) \log(1-\hat{y})$
Classification	Hinge (SVM)	$L = \max(0, 1 - y\hat{y})$

**Key insight:** the loss function encodes what “good” means. Squared error penalizes big mistakes quadratically; absolute error is more robust to outliers.

# The Learning Problem: Summary

## The Learning Problem

① **Given:** training data  $\{(x_i, y_i)\}_{i=1}^n$

# The Learning Problem: Summary

## The Learning Problem

- 1 **Given:** training data  $\{(x_i, y_i)\}_{i=1}^n$
- 2 **Choose:** model family  $f_\theta$ , loss function  $L$

# The Learning Problem: Summary

## The Learning Problem

- 1 **Given:** training data  $\{(x_i, y_i)\}_{i=1}^n$
- 2 **Choose:** model family  $f_\theta$ , loss function  $L$
- 3 **Solve:** Minimize  $\min_{\theta} \frac{1}{n} \sum_{i=1}^n L(f_\theta(x_i), y_i) + \lambda$  regularization on training data.

# The Learning Problem: Summary

## The Learning Problem

- 1 **Given:** training data  $\{(x_i, y_i)\}_{i=1}^n$
- 2 **Choose:** model family  $f_\theta$ , loss function  $L$
- 3 **Solve:** Minimize  $\min_{\theta} \frac{1}{n} \sum_{i=1}^n L(f_\theta(x_i), y_i) + \lambda \text{ regularization on training data.}$
- 4 **Evaluate:** on held-out test data, fine tune hyperparameters on validation data.

# The Learning Problem: Summary

## The Learning Problem

- 1 **Given:** training data  $\{(x_i, y_i)\}_{i=1}^n$
- 2 **Choose:** model family  $f_\theta$ , loss function  $L$
- 3 **Solve:** Minimize  $\min_{\theta} \frac{1}{n} \sum_{i=1}^n L(f_\theta(x_i), y_i) + \lambda$  regularization on training data.
- 4 **Evaluate:** on held-out test data, fine tune hyperparameters on validation data.

This is **just optimization**.

# The Learning Problem: Summary

## The Learning Problem

- 1 **Given:** training data  $\{(x_i, y_i)\}_{i=1}^n$
- 2 **Choose:** model family  $f_\theta$ , loss function  $L$
- 3 **Solve:** Minimize  $\min_{\theta} \frac{1}{n} \sum_{i=1}^n L(f_\theta(x_i), y_i) + \lambda$  regularization on training data.
- 4 **Evaluate:** on held-out test data, fine tune hyperparameters on validation data.

This is **just optimization**.

**The question is...** how do we optimize when  $n$  is enormous?

- 1 Where We Are
- 2 The Learning Problem
- 3 Stochastic Gradient Descent**
- 4 Adam, AdamW & Learning Rate Schedules
- 5 Putting It All Together: Lasso Regression
- 6 Summary

# The Scale Problem

## Modern datasets

- ImageNet: **1.2 million** images

# The Scale Problem

## Modern datasets

- ImageNet: **1.2 million** images
- Common Crawl (LLM training): **3 trillion** tokens

# The Scale Problem

## Modern datasets

- ImageNet: **1.2 million** images
- Common Crawl (LLM training): **3 trillion** tokens
- Spotify recommendations: **600 million+** users  $\times$  100 million tracks

# The Scale Problem

## Modern datasets

- ImageNet: **1.2 million** images
- Common Crawl (LLM training): **3 trillion** tokens
- Spotify recommendations: **600 million+** users  $\times$  100 million tracks

Recall: gradient descent computes

$$\nabla_{\theta} \hat{R}(\theta) = \frac{1}{n} \sum_{i=1}^n \nabla_{\theta} L(f_{\theta}(x_i), y_i)$$

# The Scale Problem

## Modern datasets

- ImageNet: **1.2 million** images
- Common Crawl (LLM training): **3 trillion** tokens
- Spotify recommendations: **600 million+** users  $\times$  100 million tracks

Recall: gradient descent computes

$$\nabla_{\theta} \hat{R}(\theta) = \frac{1}{n} \sum_{i=1}^n \nabla_{\theta} L(f_{\theta}(x_i), y_i)$$

For  $n = 10^9$ , **one gradient step** touches every data point. That's absurd.

# The Scale Problem

## Modern datasets

- ImageNet: **1.2 million** images
- Common Crawl (LLM training): **3 trillion** tokens
- Spotify recommendations: **600 million+** users  $\times$  100 million tracks

Recall: gradient descent computes

$$\nabla_{\theta} \hat{R}(\theta) = \frac{1}{n} \sum_{i=1}^n \nabla_{\theta} L(f_{\theta}(x_i), y_i)$$

For  $n = 10^9$ , **one gradient step** touches every data point. That's absurd.

**The question is...** can we get a useful gradient without touching all the data?

# SGD: The Key Idea

## Stochastic Gradient Descent:

# SGD: The Key Idea

## Stochastic Gradient Descent:

Sample a random **mini-batch**  $B \subset \{1, \dots, n\}$ ,  $|B| = b$ .

# SGD: The Key Idea

## Stochastic Gradient Descent:

Sample a random **mini-batch**  $B \subset \{1, \dots, n\}$ ,  $|B| = b$ .

Compute the **stochastic gradient**:

$$g_B = \frac{1}{b} \sum_{i \in B} \nabla_{\theta} L(f_{\theta}(x_i), y_i)$$

# SGD: The Key Idea

## Stochastic Gradient Descent:

Sample a random **mini-batch**  $B \subset \{1, \dots, n\}$ ,  $|B| = b$ .

Compute the **stochastic gradient**:

$$g_B = \frac{1}{b} \sum_{i \in B} \nabla_{\theta} L(f_{\theta}(x_i), y_i)$$

Update:  $\theta \leftarrow \theta - \eta g_B$

# SGD: The Key Idea

## Stochastic Gradient Descent:

Sample a random **mini-batch**  $B \subset \{1, \dots, n\}$ ,  $|B| = b$ .

Compute the **stochastic gradient**:

$$g_B = \frac{1}{b} \sum_{i \in B} \nabla_{\theta} L(f_{\theta}(x_i), y_i)$$

Update:  $\theta \leftarrow \theta - \eta g_B$

## Key insight

$\mathbb{E}[g_B] = \nabla_{\theta} \hat{R}(\theta)$ . The mini-batch gradient is an **unbiased estimate** of the full gradient.

# SGD: The Key Idea

## Stochastic Gradient Descent:

Sample a random **mini-batch**  $B \subset \{1, \dots, n\}$ ,  $|B| = b$ .

Compute the **stochastic gradient**:

$$g_B = \frac{1}{b} \sum_{i \in B} \nabla_{\theta} L(f_{\theta}(x_i), y_i)$$

Update:  $\theta \leftarrow \theta - \eta g_B$

## Key insight

$\mathbb{E}[g_B] = \nabla_{\theta} \hat{R}(\theta)$ . The mini-batch gradient is an **unbiased estimate** of the full gradient.

Cost per step:  $O(b)$  instead of  $O(n)$ .

If  $b = 64$  and  $n = 10^9$ , that's a **15-million-fold speedup** per step.

# Why Is the Estimate Unbiased?

Each  $i \in B$  is sampled uniformly from  $\{1, \dots, n\}$ .

# Why Is the Estimate Unbiased?

Each  $i \in B$  is sampled uniformly from  $\{1, \dots, n\}$ .

For a single sample  $i$ :

$$\mathbb{E}_i [\nabla_{\theta} L(f_{\theta}(x_i), y_i)] = \frac{1}{n} \sum_{j=1}^n \nabla_{\theta} L(f_{\theta}(x_j), y_j) = \nabla_{\theta} \hat{R}(\theta)$$

# Why Is the Estimate Unbiased?

Each  $i \in B$  is sampled uniformly from  $\{1, \dots, n\}$ .

For a single sample  $i$ :

$$\mathbb{E}_i [\nabla_{\theta} L(f_{\theta}(x_i), y_i)] = \frac{1}{n} \sum_{j=1}^n \nabla_{\theta} L(f_{\theta}(x_j), y_j) = \nabla_{\theta} \hat{R}(\theta)$$

Averaging  $b$  samples doesn't change the expectation:

$$\mathbb{E}[g_B] = \nabla_{\theta} \hat{R}(\theta)$$

# Why Is the Estimate Unbiased?

Each  $i \in B$  is sampled uniformly from  $\{1, \dots, n\}$ .

For a single sample  $i$ :

$$\mathbb{E}_i [\nabla_{\theta} L(f_{\theta}(x_i), y_i)] = \frac{1}{n} \sum_{j=1}^n \nabla_{\theta} L(f_{\theta}(x_j), y_j) = \nabla_{\theta} \hat{R}(\theta)$$

Averaging  $b$  samples doesn't change the expectation:

$$\mathbb{E}[g_B] = \nabla_{\theta} \hat{R}(\theta)$$

**But:** the estimate has **variance**. Larger  $b \rightarrow$  lower variance, higher cost.

# Full-Batch GD vs. SGD

	<b>Full-Batch GD</b>	<b>SGD (mini-batch)</b>
Gradient	Exact $\nabla \hat{R}(\theta)$	Estimate $g_B$

# Full-Batch GD vs. SGD

	<b>Full-Batch GD</b>	<b>SGD (mini-batch)</b>
Gradient	Exact $\nabla \hat{R}(\theta)$	Estimate $g_B$
Cost / step	$O(n)$	$O(b)$

# Full-Batch GD vs. SGD

	<b>Full-Batch GD</b>	<b>SGD (mini-batch)</b>
Gradient	Exact $\nabla \hat{R}(\theta)$	Estimate $g_B$
Cost / step	$O(n)$	$O(b)$
Path	Smooth, deterministic	Noisy, stochastic

# Full-Batch GD vs. SGD

	<b>Full-Batch GD</b>	<b>SGD (mini-batch)</b>
Gradient	Exact $\nabla \hat{R}(\theta)$	Estimate $g_B$
Cost / step	$O(n)$	$O(b)$
Path	Smooth, deterministic	Noisy, stochastic
Memory	All data in memory	Only batch in memory

# SGD Convergence

The mini-batch gradient  $g_B$  is unbiased, but it has **variance**:

$$\sigma^2 = \mathbb{E}[\|g_B - \nabla \hat{R}(\theta)\|^2]$$

# SGD Convergence

The mini-batch gradient  $g_B$  is unbiased, but it has **variance**:

$$\sigma^2 = \mathbb{E}[\|g_B - \nabla \hat{R}(\theta)\|^2]$$

**In words:**  $\sigma^2$  measures how much the mini-batch gradient *fluctuates* around the true gradient. Smaller batch  $\rightarrow$  larger  $\sigma^2$ .

# SGD Convergence

The mini-batch gradient  $g_B$  is unbiased, but it has **variance**:

$$\sigma^2 = \mathbb{E}[\|g_B - \nabla \hat{R}(\theta)\|^2]$$

**In words:**  $\sigma^2$  measures how much the mini-batch gradient *fluctuates* around the true gradient. Smaller batch  $\rightarrow$  larger  $\sigma^2$ .

For **convex**  $f$  with step size  $\eta_t = c/\sqrt{T}$ :

$$\mathbb{E}[f(\theta_T)] - f(\theta^*) \leq O\left(\frac{\|\theta_0 - \theta^*\|^2 + \sigma^2}{\sqrt{T}}\right)$$

# SGD Convergence

The mini-batch gradient  $g_B$  is unbiased, but it has **variance**:

$$\sigma^2 = \mathbb{E}[\|g_B - \nabla \hat{R}(\theta)\|^2]$$

**In words:**  $\sigma^2$  measures how much the mini-batch gradient *fluctuates* around the true gradient. Smaller batch  $\rightarrow$  larger  $\sigma^2$ .

For **convex**  $f$  with step size  $\eta_t = c/\sqrt{T}$ :

$$\mathbb{E}[f(\theta_T)] - f(\theta^*) \leq O\left(\frac{\|\theta_0 - \theta^*\|^2 + \sigma^2}{\sqrt{T}}\right)$$

SGD converges at rate  $O(1/\sqrt{T})$ , slower than GD's  $O(1/T)$ , but each step is **massively** cheaper.

# SGD Convergence

The mini-batch gradient  $g_B$  is unbiased, but it has **variance**:

$$\sigma^2 = \mathbb{E}[\|g_B - \nabla \hat{R}(\theta)\|^2]$$

**In words:**  $\sigma^2$  measures how much the mini-batch gradient *fluctuates* around the true gradient. Smaller batch  $\rightarrow$  larger  $\sigma^2$ .

For **convex**  $f$  with step size  $\eta_t = c/\sqrt{T}$ :

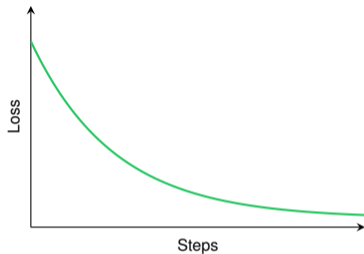
$$\mathbb{E}[f(\theta_T)] - f(\theta^*) \leq O\left(\frac{\|\theta_0 - \theta^*\|^2 + \sigma^2}{\sqrt{T}}\right)$$

SGD converges at rate  $O(1/\sqrt{T})$ , slower than GD's  $O(1/T)$ , but each step is **massively** cheaper.

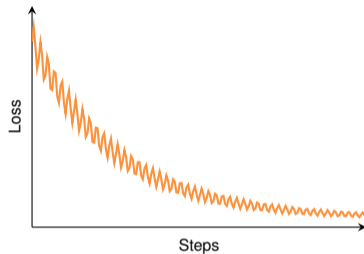
**The deal:** for the same wall-clock time, SGD typically makes far more progress because it takes many more (cheaper) steps.

# The Variance–Computation Tradeoff

Large batch ( $b = 512$ )

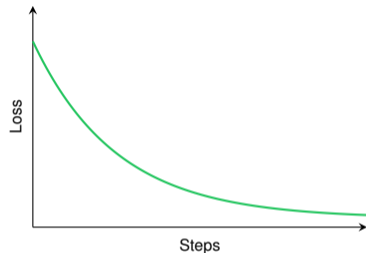


Small batch ( $b = 16$ )

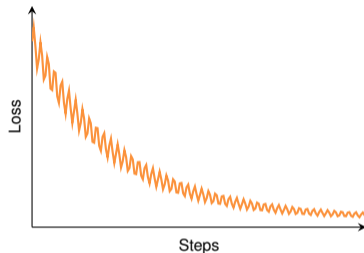


# The Variance–Computation Tradeoff

Large batch ( $b = 512$ )



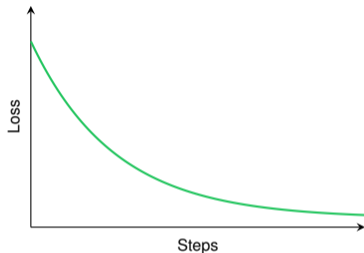
Small batch ( $b = 16$ )



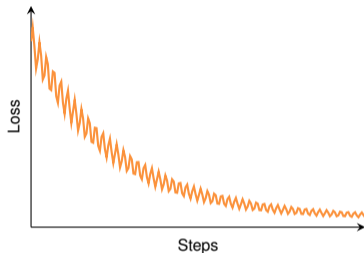
- **Larger batch** → lower variance → smoother curve, but more expensive per step.

# The Variance–Computation Tradeoff

Large batch ( $b = 512$ )



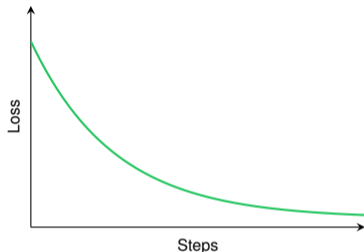
Small batch ( $b = 16$ )



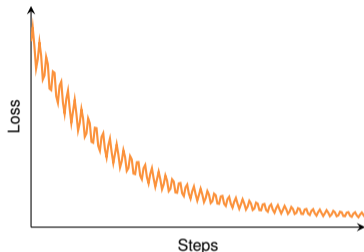
- **Larger batch** → lower variance → smoother curve, but more expensive per step.
- **Smaller batch** → higher variance → noisier, but cheaper and often converges to same final loss.

# The Variance–Computation Tradeoff

Large batch ( $b = 512$ )



Small batch ( $b = 16$ )



- **Larger batch** → lower variance → smoother curve, but more expensive per step.
- **Smaller batch** → higher variance → noisier, but cheaper and often converges to same final loss.
- **Rule of thumb:** batch sizes of 32–512 work well in practice.

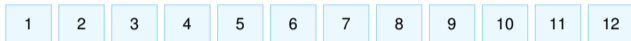
# An Epoch of SGD

Data: 

1	2	3	4	5	6	7	8	9	10	11	12
---	---	---	---	---	---	---	---	---	----	----	----

# An Epoch of SGD

Data:



Shuffle ↓



# An Epoch of SGD



# An Epoch of SGD



Each batch  $\rightarrow$  one gradient step  $\rightarrow$  one  $\theta$  update

**1 epoch** = 3 updates (if  $b = 4$ ,  $n = 12$ )

# SGD in Python

```
import numpy as np

def sgd(X, y, theta, lr=0.01, batch_size=32, epochs=100):
    n = len(y)
    for epoch in range(epochs):
        # Shuffle data each epoch
        perm = np.random.permutation(n)
        X, y = X[perm], y[perm]
        for i in range(0, n, batch_size):
            X_b = X[i:i+batch_size]
            y_b = y[i:i+batch_size]
            grad = compute_gradient(theta, X_b, y_b)
            theta = theta - lr * grad
    return theta
```

# SGD in Python

```
import numpy as np

def sgd(X, y, theta, lr=0.01, batch_size=32, epochs=100):
    n = len(y)
    for epoch in range(epochs):
        # Shuffle data each epoch
        perm = np.random.permutation(n)
        X, y = X[perm], y[perm]
        for i in range(0, n, batch_size):
            X_b = X[i:i+batch_size]
            y_b = y[i:i+batch_size]
            grad = compute_gradient(theta, X_b, y_b)
            theta = theta - lr * grad
    return theta
```

**Key points:** shuffle each epoch (avoid cycles), process in mini-batches, update after each batch.

# Why Does SGD Work for Neural Nets?

## The elephant in the room

Neural networks define **non-convex** loss landscapes.  
SGD convergence theory requires convexity.

**Why does this work at all?**

# Why Does SGD Work for Neural Nets?

## The elephant in the room

Neural networks define **non-convex** loss landscapes.  
SGD convergence theory requires convexity.

## Why does this work at all?

Three insights from the last decade of research:

# Why Does SGD Work for Neural Nets?

## The elephant in the room

Neural networks define **non-convex** loss landscapes.  
SGD convergence theory requires convexity.

## Why does this work at all?

Three insights from the last decade of research:

- 1 Benign loss landscapes

# Why Does SGD Work for Neural Nets?

## The elephant in the room

Neural networks define **non-convex** loss landscapes.  
SGD convergence theory requires convexity.

### Why does this work at all?

Three insights from the last decade of research:

- 1 Benign loss landscapes
- 2 SGD noise helps escape bad regions

# Why Does SGD Work for Neural Nets?

## The elephant in the room

Neural networks define **non-convex** loss landscapes.  
SGD convergence theory requires convexity.

### Why does this work at all?

Three insights from the last decade of research:

- 1 Benign loss landscapes
- 2 SGD noise helps escape bad regions
- 3 Implicit regularization

# Insight 1: Benign Loss Landscapes

**Overparameterized** networks: more parameters than data points.

# Insight 1: Benign Loss Landscapes

**Overparameterized** networks: more parameters than data points.

**Key conjecture:** in high dimensions, local minima have loss values **very close** to the global minimum.

# Insight 1: Benign Loss Landscapes

**Overparameterized** networks: more parameters than data points.

**Key conjecture:** in high dimensions, local minima have loss values **very close** to the global minimum.

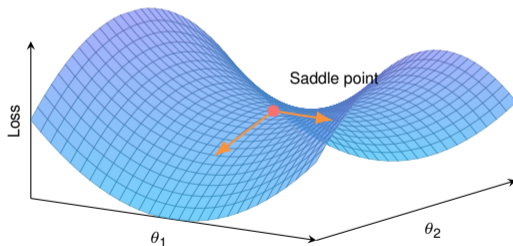
The real obstacles are **saddle points**, not local minima.

# Insight 1: Benign Loss Landscapes

**Overparameterized** networks: more parameters than data points.

**Key conjecture:** in high dimensions, local minima have loss values **very close** to the global minimum.

The real obstacles are **saddle points**, not local minima.



## Insight 2: SGD Noise Helps

At a saddle point, the true gradient is zero (or near zero).

## Insight 2: SGD Noise Helps

At a saddle point, the true gradient is zero (or near zero).

**Full-batch GD:** gets stuck. The gradient really is near zero.

## Insight 2: SGD Noise Helps

At a saddle point, the true gradient is zero (or near zero).

**Full-batch GD:** gets stuck. The gradient really is near zero.

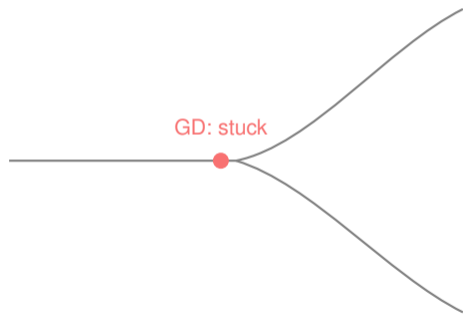
**SGD:** the stochastic noise “kicks” the iterate away from saddle points.

## Insight 2: SGD Noise Helps

At a saddle point, the true gradient is zero (or near zero).

**Full-batch GD:** gets stuck. The gradient really is near zero.

**SGD:** the stochastic noise “kicks” the iterate away from saddle points.



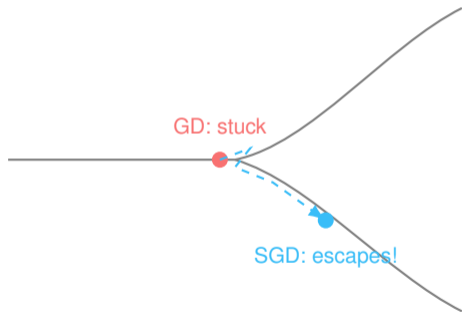
The noise in SGD is a **feature**, not a bug

## Insight 2: SGD Noise Helps

At a saddle point, the true gradient is zero (or near zero).

**Full-batch GD:** gets stuck. The gradient really is near zero.

**SGD:** the stochastic noise “kicks” the iterate away from saddle points.



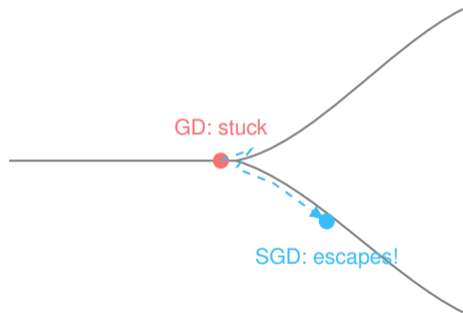
Noise breaks the symmetry at saddle points

## Insight 2: SGD Noise Helps

At a saddle point, the true gradient is zero (or near zero).

**Full-batch GD:** gets stuck. The gradient really is near zero.

**SGD:** the stochastic noise “kicks” the iterate away from saddle points.



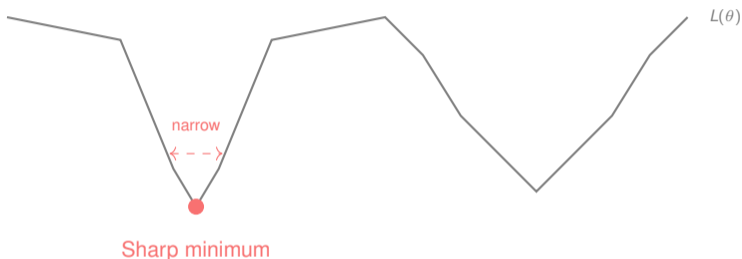
The noise in SGD is a **feature**, not a bug.

## Insight 3: Implicit Regularization

SGD with small batch sizes tends to converge to **flat** minima.

## Insight 3: Implicit Regularization

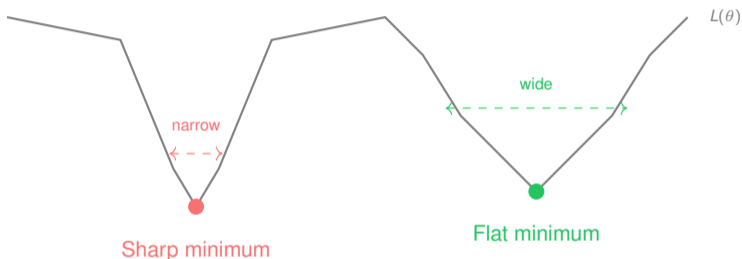
SGD with small batch sizes tends to converge to **flat** minima.



**Takeaway:** SGD's noise acts as a natural regularizer, steering toward solutions that generalize well.

## Insight 3: Implicit Regularization

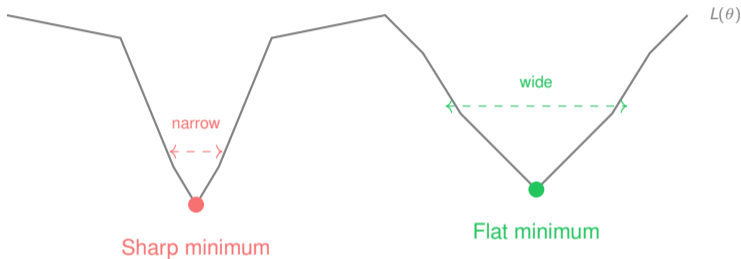
SGD with small batch sizes tends to converge to **flat** minima.



**Takeaway:** SGD's noise acts as a natural regularizer, steering toward solutions that generalize well.

# Insight 3: Implicit Regularization

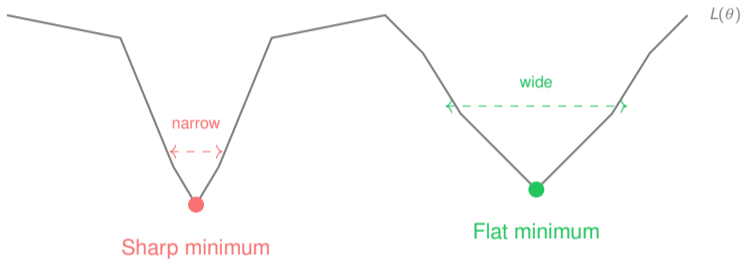
SGD with small batch sizes tends to converge to **flat** minima.



Both minima have similar training loss, but flat minima **generalize better**:  
small perturbations to  $\theta$  don't change the loss much.

# Insight 3: Implicit Regularization

SGD with small batch sizes tends to converge to **flat** minima.



Both minima have similar training loss, but flat minima **generalize better**:  
small perturbations to  $\theta$  don't change the loss much.

**Takeaway:** SGD's noise acts as a natural regularizer, steering toward solutions that generalize well.

# SGD for Non-Convex Problems: Summary

## Why SGD works in practice

- 1 **Benign landscapes:** in overparameterized models, most local minima are nearly as good as the global minimum.

# SGD for Non-Convex Problems: Summary

## Why SGD works in practice

- 1 **Benign landscapes:** in overparameterized models, most local minima are nearly as good as the global minimum.
- 2 **Noise helps:** stochastic gradients escape saddle points that trap full-batch GD.

# SGD for Non-Convex Problems: Summary

## Why SGD works in practice

- 1 **Benign landscapes:** in overparameterized models, most local minima are nearly as good as the global minimum.
- 2 **Noise helps:** stochastic gradients escape saddle points that trap full-batch GD.
- 3 **Implicit regularization:** SGD noise biases the solution toward flat minima that generalize.

# SGD for Non-Convex Problems: Summary

## Why SGD works in practice

- 1 **Benign landscapes:** in overparameterized models, most local minima are nearly as good as the global minimum.
- 2 **Noise helps:** stochastic gradients escape saddle points that trap full-batch GD.
- 3 **Implicit regularization:** SGD noise biases the solution toward flat minima that generalize.

**Key insight:** theory for convex problems gives us the foundation. Practice shows SGD works far beyond what theory can prove.

- 1 Where We Are
- 2 The Learning Problem
- 3 Stochastic Gradient Descent
- 4 Adam, AdamW & Learning Rate Schedules**
- 5 Putting It All Together: Lasso Regression
- 6 Summary

# Can We Do Better Than Vanilla SGD?

**Problem:** vanilla SGD uses the **same** learning rate  $\eta$  for all parameters.

# Can We Do Better Than Vanilla SGD?

**Problem:** vanilla SGD uses the **same** learning rate  $\eta$  for all parameters.

But parameters behave very differently:

- Some gradients are **large** and consistent, well-determined direction.

# Can We Do Better Than Vanilla SGD?

**Problem:** vanilla SGD uses the **same** learning rate  $\eta$  for all parameters.

But parameters behave very differently:

- Some gradients are **large** and consistent, well-determined direction.
- Some gradients are **small** and noisy, uncertain direction.

# Can We Do Better Than Vanilla SGD?

**Problem:** vanilla SGD uses the **same** learning rate  $\eta$  for all parameters.

But parameters behave very differently:

- Some gradients are **large** and consistent, well-determined direction.
- Some gradients are **small** and noisy, uncertain direction.
- Some dimensions have steep curvature, others are flat.

# Can We Do Better Than Vanilla SGD?

**Problem:** vanilla SGD uses the **same** learning rate  $\eta$  for all parameters.

But parameters behave very differently:

- Some gradients are **large** and consistent, well-determined direction.
- Some gradients are **small** and noisy, uncertain direction.
- Some dimensions have steep curvature, others are flat.

**The question is...** can we **adapt** the learning rate per parameter?

# Momentum: Smoothing the Gradient

Before Adam, a key idea: **momentum**.

# Momentum: Smoothing the Gradient

Before Adam, a key idea: **momentum**.

Instead of using  $g_t$  directly, maintain a running average:

$$m_t = \beta m_{t-1} + (1 - \beta) g_t$$

# Momentum: Smoothing the Gradient

Before Adam, a key idea: **momentum**.

Instead of using  $g_t$  directly, maintain a running average:

$$m_t = \beta m_{t-1} + (1 - \beta) g_t$$

Update:  $\theta \leftarrow \theta - \eta m_t$

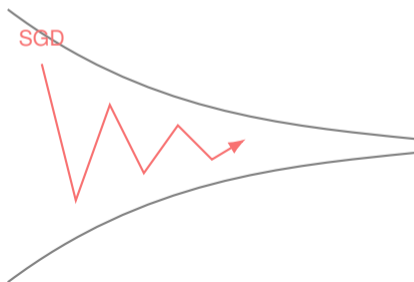
# Momentum: Smoothing the Gradient

Before Adam, a key idea: **momentum**.

Instead of using  $g_t$  directly, maintain a running average:

$$m_t = \beta m_{t-1} + (1 - \beta) g_t$$

Update:  $\theta \leftarrow \theta - \eta m_t$



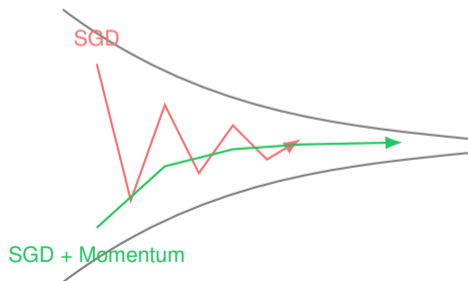
# Momentum: Smoothing the Gradient

Before Adam, a key idea: **momentum**.

Instead of using  $g_t$  directly, maintain a running average:

$$m_t = \beta m_{t-1} + (1 - \beta) g_t$$

Update:  $\theta \leftarrow \theta - \eta m_t$



# Adam: Adaptive Moment Estimation

Adam combines **momentum** and **adaptive learning rates**:

# Adam: Adaptive Moment Estimation

Adam combines **momentum** and **adaptive learning rates**:

**First moment** (smoothed gradient direction):

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

# Adam: Adaptive Moment Estimation

Adam combines **momentum** and **adaptive learning rates**:

**First moment** (smoothed gradient direction):

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

**Second moment** (smoothed gradient magnitude):

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

# Adam: Adaptive Moment Estimation

Adam combines **momentum** and **adaptive learning rates**:

**First moment** (smoothed gradient direction):

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

**Second moment** (smoothed gradient magnitude):

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

Bias correction:  $\hat{m}_t = m_t / (1 - \beta_1^t)$ ,  $\hat{v}_t = v_t / (1 - \beta_2^t)$

# Adam: Adaptive Moment Estimation

Adam combines **momentum** and **adaptive learning rates**:

**First moment** (smoothed gradient direction):

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

**Second moment** (smoothed gradient magnitude):

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

Bias correction:  $\hat{m}_t = m_t / (1 - \beta_1^t)$ ,  $\hat{v}_t = v_t / (1 - \beta_2^t)$

**Update:**

$$\theta \leftarrow \theta - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

# Adam: Adaptive Moment Estimation

Adam combines **momentum** and **adaptive learning rates**:

**First moment** (smoothed gradient direction):

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

**Second moment** (smoothed gradient magnitude):

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

Bias correction:  $\hat{m}_t = m_t / (1 - \beta_1^t)$ ,  $\hat{v}_t = v_t / (1 - \beta_2^t)$

**Update:**

$$\theta \leftarrow \theta - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}}$$

**In words:** move in the smoothed gradient direction, but **slow down** in dimensions with large gradients. Defaults:  $\beta_1 = 0.9$ ,  $\beta_2 = 0.999$ ,  $\epsilon = 10^{-8}$ .

# AdamW: The Weight Decay Fix

L2 regularization in Adam is NOT the same as weight decay

**Adam + L2:** the gradient of the penalty term  $\lambda\theta$  gets scaled by the adaptive learning rate  $1/\sqrt{\hat{v}_t}$ .

This means the regularization strength **varies per parameter**. Not what we intended.

# AdamW: The Weight Decay Fix

L2 regularization in Adam is NOT the same as weight decay

**Adam + L2:** the gradient of the penalty term  $\lambda\theta$  gets scaled by the adaptive learning rate  $1/\sqrt{\hat{v}_t}$ .

This means the regularization strength **varies per parameter**. Not what we intended.

**AdamW:** apply weight decay **directly** to parameters, *separate* from the gradient:

$$\theta \leftarrow \theta - \eta \left( \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \varepsilon}} + \lambda \theta \right)$$

## AdamW: The Weight Decay Fix

L2 regularization in Adam is NOT the same as weight decay

**Adam + L2:** the gradient of the penalty term  $\lambda\theta$  gets scaled by the adaptive learning rate  $1/\sqrt{\hat{v}_t}$ .

This means the regularization strength **varies per parameter**. Not what we intended.

**AdamW:** apply weight decay **directly** to parameters, *separate* from the gradient:

$$\theta \leftarrow \theta - \eta \left( \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \varepsilon}} + \lambda\theta \right)$$

**The deal:** AdamW **decouples** weight decay from the gradient update.

# AdamW: The Weight Decay Fix

L2 regularization in Adam is NOT the same as weight decay

**Adam + L2:** the gradient of the penalty term  $\lambda\theta$  gets scaled by the adaptive learning rate  $1/\sqrt{\hat{v}_t}$ .

This means the regularization strength **varies per parameter**. Not what we intended.

**AdamW:** apply weight decay **directly** to parameters, *separate* from the gradient:

$$\theta \leftarrow \theta - \eta \left( \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \varepsilon}} + \lambda\theta \right)$$

**The deal:** AdamW **decouples** weight decay from the gradient update.

This is the standard optimizer choice for many modern large-scale neural network training runs.

# Adam vs. AdamW: The Difference

## Adam + L2

$$g_t \leftarrow \nabla L(\theta) + \lambda \theta$$

$m_t, v_t$  from  $g_t$

$$\theta \leftarrow \theta - \eta \hat{m}_t / (\sqrt{\hat{v}_t} + \varepsilon)$$

Regularization gets scaled by  $1/\sqrt{\hat{v}_t}$

Coupling problem

## AdamW

$$g_t \leftarrow \nabla L(\theta)$$

$m_t, v_t$  from  $g_t$

$$\theta \leftarrow \theta - \eta (\hat{m}_t / (\sqrt{\hat{v}_t} + \varepsilon) + \lambda \theta)$$

Weight decay applied uniformly

Decoupled, correct

# Adam vs. AdamW: The Difference

## Adam + L2

$$g_t \leftarrow \nabla L(\theta) + \lambda \theta$$

$m_t, v_t$  from  $g_t$

$$\theta \leftarrow \theta - \eta \hat{m}_t / (\sqrt{\hat{v}_t} + \varepsilon)$$

Regularization gets scaled by  $1/\sqrt{\hat{v}_t}$

Coupling problem

## AdamW

$$g_t \leftarrow \nabla L(\theta)$$

$m_t, v_t$  from  $g_t$

$$\theta \leftarrow \theta - \eta (\hat{m}_t / (\sqrt{\hat{v}_t} + \varepsilon) + \lambda \theta)$$

Weight decay applied uniformly

Decoupled, correct

Loshchilov & Hutter (2019): this seemingly small fix **consistently improves generalization.**

# Learning Rate Schedules

**Problem:** a constant learning rate is suboptimal.

- Too high  $\rightarrow$  diverge or oscillate.

# Learning Rate Schedules

**Problem:** a constant learning rate is suboptimal.

- Too high  $\rightarrow$  diverge or oscillate.
- Too low  $\rightarrow$  converge painfully slowly.

# Learning Rate Schedules

**Problem:** a constant learning rate is suboptimal.

- Too high  $\rightarrow$  diverge or oscillate.
- Too low  $\rightarrow$  converge painfully slowly.

**Cosine decay:**

$$\eta_t = \eta_{\min} + \frac{1}{2}(\eta_{\max} - \eta_{\min}) \left( 1 + \cos\left(\frac{t - T_w}{T - T_w} \cdot \pi\right) \right)$$

# Learning Rate Schedules

**Problem:** a constant learning rate is suboptimal.

- Too high  $\rightarrow$  diverge or oscillate.
- Too low  $\rightarrow$  converge painfully slowly.

**Cosine decay:**

$$\eta_t = \eta_{\min} + \frac{1}{2}(\eta_{\max} - \eta_{\min}) \left( 1 + \cos\left(\frac{t - T_w}{T - T_w} \cdot \pi\right) \right)$$

**In words:** start carefully, take larger steps after warmup, then decay smoothly for fine-tuning.

# Learning Rate Schedules

**Problem:** a constant learning rate is suboptimal.

- Too high  $\rightarrow$  diverge or oscillate.
- Too low  $\rightarrow$  converge painfully slowly.

**Cosine decay:**

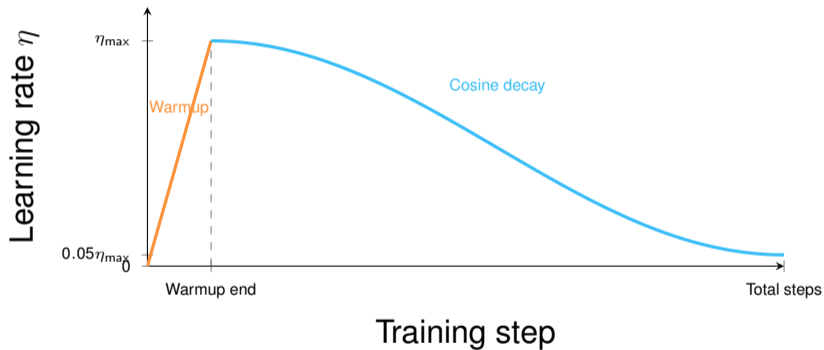
$$\eta_t = \eta_{\min} + \frac{1}{2}(\eta_{\max} - \eta_{\min}) \left( 1 + \cos\left(\frac{t - T_w}{T - T_w} \cdot \pi\right) \right)$$

**In words:** start carefully, take larger steps after warmup, then decay smoothly for fine-tuning.

**Why warmup?** Early gradients are unreliable; small initial steps let Adam's moments stabilize.

# Learning Rate Schedule: Visual

Linear warmup followed by cosine decay:



# Learning Rate Warmup: Intuition

At initialization, the model weights are **random**.

# Learning Rate Warmup: Intuition

At initialization, the model weights are **random**.

- Gradients in the first few steps point in essentially **random** directions.

# Learning Rate Warmup: Intuition

At initialization, the model weights are **random**.

- Gradients in the first few steps point in essentially **random** directions.
- Adam's running averages  $m_t$ ,  $v_t$  have no history yet.

# Learning Rate Warmup: Intuition

At initialization, the model weights are **random**.

- Gradients in the first few steps point in essentially **random** directions.
- Adam's running averages  $m_t$ ,  $v_t$  have no history yet.
- A large LR + unreliable gradients = **destructive updates**.

# Learning Rate Warmup: Intuition

At initialization, the model weights are **random**.

- Gradients in the first few steps point in essentially **random** directions.
- Adam's running averages  $m_t$ ,  $v_t$  have no history yet.
- A large LR + unreliable gradients = **destructive updates**.

**Warmup** (typically 1–5% of total training):

- Start with tiny LR, ramp up linearly.

# Learning Rate Warmup: Intuition

At initialization, the model weights are **random**.

- Gradients in the first few steps point in essentially **random** directions.
- Adam's running averages  $m_t$ ,  $v_t$  have no history yet.
- A large LR + unreliable gradients = **destructive updates**.

**Warmup** (typically 1–5% of total training):

- Start with tiny LR, ramp up linearly.
- By the time LR reaches its peak, Adam's moments are well-calibrated.

# Learning Rate Warmup: Intuition

At initialization, the model weights are **random**.

- Gradients in the first few steps point in essentially **random** directions.
- Adam's running averages  $m_t$ ,  $v_t$  have no history yet.
- A large LR + unreliable gradients = **destructive updates**.

**Warmup** (typically 1–5% of total training):

- Start with tiny LR, ramp up linearly.
- By the time LR reaches its peak, Adam's moments are well-calibrated.

**Key insight:** warmup is the “calibration phase” for adaptive optimizers.

# Optimizers: Comparison

	<b>SGD</b>	<b>Adam</b>	<b>AdamW</b>
Adaptive LR	×	✓	✓

# Optimizers: Comparison

	<b>SGD</b>	<b>Adam</b>	<b>AdamW</b>
Adaptive LR	×	✓	✓
Momentum	Optional	Built-in	Built-in

# Optimizers: Comparison

	<b>SGD</b>	<b>Adam</b>	<b>AdamW</b>
Adaptive LR	×	✓	✓
Momentum	Optional	Built-in	Built-in
Weight decay	Correct	Coupled	Decoupled

# Optimizers: Comparison

	<b>SGD</b>	<b>Adam</b>	<b>AdamW</b>
Adaptive LR	×	✓	✓
Momentum	Optional	Built-in	Built-in
Weight decay	Correct	Coupled	Decoupled
Hyperparams	$\eta$	$\eta, \beta_1, \beta_2$	$\eta, \beta_1, \beta_2, \lambda$

# Optimizers: Comparison

	<b>SGD</b>	<b>Adam</b>	<b>AdamW</b>
Adaptive LR	×	✓	✓
Momentum	Optional	Built-in	Built-in
Weight decay	Correct	Coupled	Decoupled
Hyperparams	$\eta$	$\eta, \beta_1, \beta_2$	$\eta, \beta_1, \beta_2, \lambda$
Used for LLMs	Rarely	Sometimes	<b>Almost always</b>

# Optimizers: Comparison

	<b>SGD</b>	<b>Adam</b>	<b>AdamW</b>
Adaptive LR	×	✓	✓
Momentum	Optional	Built-in	Built-in
Weight decay	Correct	Coupled	Decoupled
Hyperparams	$\eta$	$\eta, \beta_1, \beta_2$	$\eta, \beta_1, \beta_2, \lambda$
Used for LLMs	Rarely	Sometimes	<b>Almost always</b>

**Takeaway:** AdamW with warmup + cosine decay is the standard recipe for training large neural networks.

# Putting It All Together

```
import math
import torch
optimizer = torch.optim.AdamW(
    model.parameters(),
    lr=1e-3,
    weight_decay=0.01
)
def lr_lambda(step, warmup=2000, total=100000):
    if step < warmup:
        return step / warmup
    progress = (step - warmup) / (total - warmup)
    progress = min(progress, 1.0)
    return 0.5 * (1 + math.cos(math.pi * progress))

scheduler = torch.optim.lr_scheduler.LambdaLR(
    optimizer,
    lr_lambda=lr_lambda)

...
optimizer.step()    # update weights
scheduler.step()    # update learning rate for future steps
```

- 1 Where We Are
- 2 The Learning Problem
- 3 Stochastic Gradient Descent
- 4 Adam, AdamW & Learning Rate Schedules
- 5 Putting It All Together: Lasso Regression**
- 6 Summary

# Everything in Action

We now have all the pieces. Let's use them on a real problem.

# Everything in Action

We now have all the pieces. Let's use them on a real problem.

**Problem:** predict California housing prices from 8 features, with  $n = 20,640$  houses.

# Everything in Action

We now have all the pieces. Let's use them on a real problem.

**Problem:** predict California housing prices from 8 features, with  $n = 20,640$  houses.

**Model:** linear regression  $\hat{y} = \mathbf{w}^\top \mathbf{x} + b$  with **L1 regularization** (Lasso):

$$\min_{\mathbf{w}, b} \frac{1}{n} \sum_{i=1}^n (\mathbf{w}^\top \mathbf{x}_i + b - y_i)^2 + \lambda \|\mathbf{w}\|_1$$

# Everything in Action

We now have all the pieces. Let's use them on a real problem.

**Problem:** predict California housing prices from 8 features, with  $n = 20,640$  houses.

**Model:** linear regression  $\hat{y} = \mathbf{w}^\top \mathbf{x} + b$  with **L1 regularization** (Lasso):

$$\min_{\mathbf{w}, b} \frac{1}{n} \sum_{i=1}^n (\mathbf{w}^\top \mathbf{x}_i + b - y_i)^2 + \lambda \|\mathbf{w}\|_1$$

## Why Lasso?

- **ERM:** squared loss.

# Everything in Action

We now have all the pieces. Let's use them on a real problem.

**Problem:** predict California housing prices from 8 features, with  $n = 20,640$  houses.

**Model:** linear regression  $\hat{y} = \mathbf{w}^\top \mathbf{x} + b$  with **L1 regularization** (Lasso):

$$\min_{\mathbf{w}, b} \frac{1}{n} \sum_{i=1}^n (\mathbf{w}^\top \mathbf{x}_i + b - y_i)^2 + \lambda \|\mathbf{w}\|_1$$

## Why Lasso?

- **ERM:** squared loss.
- **Regularization:**  $\lambda \|\mathbf{w}\|_1$  encourages **sparse** solutions.

# Everything in Action

We now have all the pieces. Let's use them on a real problem.

**Problem:** predict California housing prices from 8 features, with  $n = 20,640$  houses.

**Model:** linear regression  $\hat{y} = \mathbf{w}^\top \mathbf{x} + b$  with **L1 regularization** (Lasso):

$$\min_{\mathbf{w}, b} \frac{1}{n} \sum_{i=1}^n (\mathbf{w}^\top \mathbf{x}_i + b - y_i)^2 + \lambda \|\mathbf{w}\|_1$$

## Why Lasso?

- **ERM:** squared loss.
- **Regularization:**  $\lambda \|\mathbf{w}\|_1$  encourages **sparse** solutions.
- **Feature selection:** Lasso tells you which features *matter*.

# Lasso with SGD: The Code

```
import torch
from sklearn.datasets import fetch_california_housing
from torch.utils.data import DataLoader, TensorDataset
# Load data
X, y = fetch_california_housing(return_X_y=True)
X = torch.tensor(X, dtype=torch.float32)
y = torch.tensor(y, dtype=torch.float32)

# Train/test split (80/20)
X_train, X_test = X[:16512], X[16512:]
y_train, y_test = y[:16512], y[16512:]

# Normalize using training mean/std only
mean = X_train.mean(dim=0)
std = X_train.std(dim=0)

X_train = (X_train - mean) / std
X_test = (X_test - mean) / std

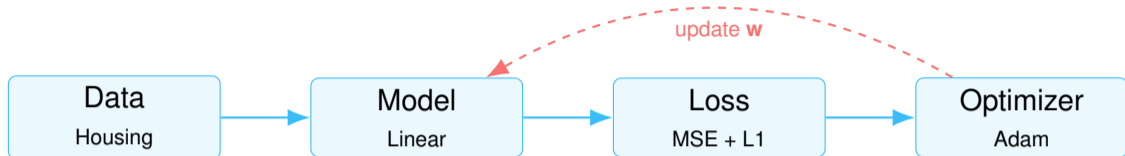
loader = DataLoader(TensorDataset(X_train, y_train), batch_size=256, shuffle=True)
```

# Lasso with SGD: Training

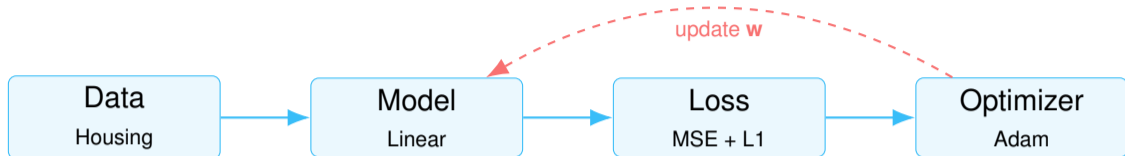
```
w = torch.randn(8, requires_grad=True)
b = torch.zeros(1, requires_grad=True)
optimizer = torch.optim.Adam([w, b], lr=0.01)
lam = 0.1 # L1 regularization strength

for epoch in range(100):
    for X_batch, y_batch in loader:
        y_hat = X_batch @ w + b
        mse = ((y_hat - y_batch)**2).mean() # ERM
        l1 = lam * w.abs().sum() # regularization
        loss = mse + l1
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

# What Just Happened?



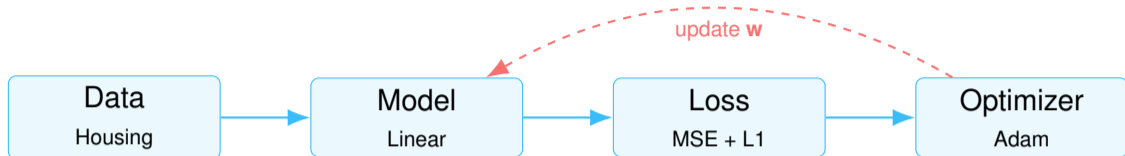
# What Just Happened?



Everything from this lecture:

- **ERM:** minimize MSE on training data.

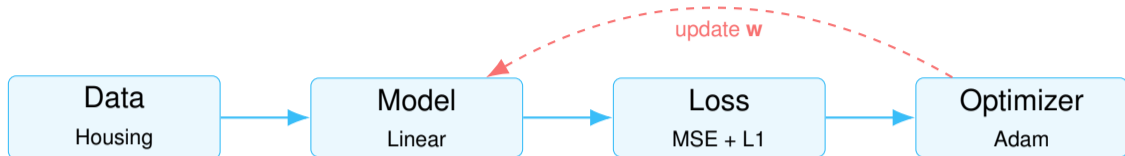
# What Just Happened?



Everything from this lecture:

- **ERM:** minimize MSE on training data.
- **Regularization:** L1 penalty drives unimportant weights to zero.

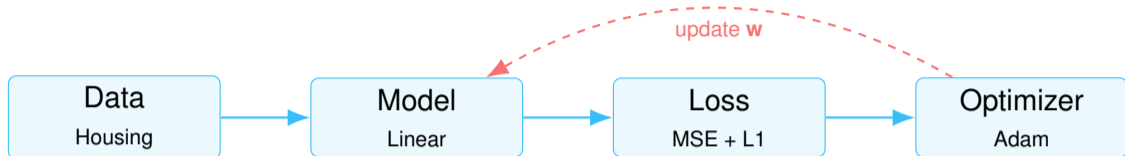
# What Just Happened?



Everything from this lecture:

- **ERM:** minimize MSE on training data.
- **Regularization:** L1 penalty drives unimportant weights to zero.
- **Mini-batch SGD + Adam:** scalable optimization on 20K data points.

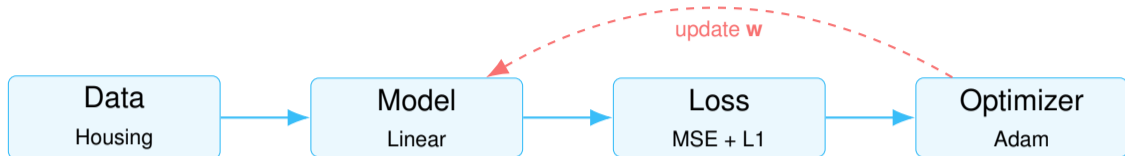
# What Just Happened?



Everything from this lecture:

- **ERM:** minimize MSE on training data.
- **Regularization:** L1 penalty drives unimportant weights to zero.
- **Mini-batch SGD + Adam:** scalable optimization on 20K data points.
- **Train/test split:** verify generalization on held-out data.

# What Just Happened?

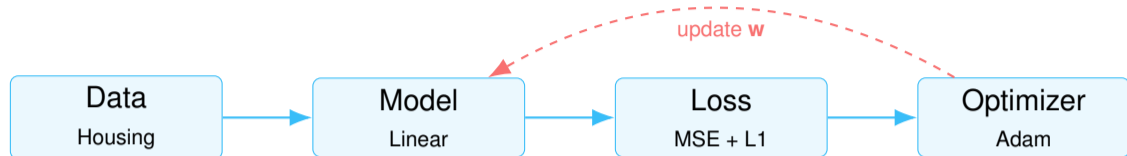


Everything from this lecture:

- **ERM:** minimize MSE on training data.
- **Regularization:** L1 penalty drives unimportant weights to zero.
- **Mini-batch SGD + Adam:** scalable optimization on 20K data points.
- **Train/test split:** verify generalization on held-out data.

**But:** this model is **linear**. It can only capture linear relationships.

# What Just Happened?



Everything from this lecture:

- **ERM:** minimize MSE on training data.
- **Regularization:** L1 penalty drives unimportant weights to zero.
- **Mini-batch SGD + Adam:** scalable optimization on 20K data points.
- **Train/test split:** verify generalization on held-out data.

**But:** this model is **linear**. It can only capture linear relationships.

*Can we do better?*  $\Rightarrow$  Next lecture: neural networks, nonlinear function approximators.

- 1 Where We Are
- 2 The Learning Problem
- 3 Stochastic Gradient Descent
- 4 Adam, AdamW & Learning Rate Schedules
- 5 Putting It All Together: Lasso Regression
- 6 Summary**

# Key Takeaways

- 1 **ML is optimization under uncertainty:** the objective comes from data, not a formula.

# Key Takeaways

- 1 **ML is optimization under uncertainty:** the objective comes from data, not a formula.
- 2 **Generalization is everything:** train/test split, regularization, and the bias-variance tradeoff.

# Key Takeaways

- 1 **ML is optimization under uncertainty:** the objective comes from data, not a formula.
- 2 **Generalization is everything:** train/test split, regularization, and the bias-variance tradeoff.
- 3 **SGD makes scale possible:** unbiased gradient estimates at a fraction of the cost.

# Key Takeaways

- 1 **ML is optimization under uncertainty:** the objective comes from data, not a formula.
- 2 **Generalization is everything:** train/test split, regularization, and the bias-variance tradeoff.
- 3 **SGD makes scale possible:** unbiased gradient estimates at a fraction of the cost.
- 4 **Adam/AdamW is the standard:** adaptive LR + decoupled weight decay.

# Key Takeaways

- 1 **ML is optimization under uncertainty:** the objective comes from data, not a formula.
- 2 **Generalization is everything:** train/test split, regularization, and the bias-variance tradeoff.
- 3 **SGD makes scale possible:** unbiased gradient estimates at a fraction of the cost.
- 4 **Adam/AdamW is the standard:** adaptive LR + decoupled weight decay.
- 5 **The noise in SGD is a feature:** it escapes saddle points and regularizes implicitly.