Gradient Descent and Its Variants

Course:
INFO-6154 Data Science and Machine Learning

Developed by:
Mohammad Noorchenarboo

May 12, 2025

Current Section

Why Do We Need Gradient Descent?

In the previous session, you optimized a model by manually guessing parameters. But in real applications, this becomes impossible:

- 1,000+ features instead of 1
- Millions of data points
- Non-linear models with millions of parameters

Limitation

Manual tuning doesn't scale. We need a **mathematical and automated** solution to improve the model iteratively.

Real-World Example: Insurance Premium Prediction

- Predict costs based on features: age, BMI, smoking, region, etc.
- Model: linear regression or shallow neural network.
- Goal: minimize error across thousands of historical records.

Why Do We Need Gradient Descent?

What Do We Need?

An algorithm that can:

- Evaluate how far off the predictions are
- Compute how to change the parameters to improve
- Repeat this process efficiently over data

This is the Role of Gradient Descent

Gradient descent uses the slope (gradient) of the loss function to adjust model parameters and minimize error.

Why Do We Need Gradient Descent?

How Optimization Appears in Practice

```
# PyTorch
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
loss.backward()
optimizer.step()

# TensorFlow (Keras)
model.compile(optimizer='sgd', loss='mse')
model.fit(X_train, y_train, epochs=10)
```

Summary: Why We Need Gradient Descent

Challenge	Gradient Descent Role
High-dimensional parameter space	Gradient descent scales better than manual tuning
Need for automation	Enables iterative improvement through loss gradients
Real-world ML training	Used in every major ML library (TensorFlow, PyTorch)

What Is a Gradient? (Conceptual View)

The **gradient** is a mathematical tool that tells us which direction to move in order to reduce a function's value the fastest.

In 1D: The gradient is the slope of the function.

$$\text{If } L(w) = (2w - 4)^2, \text{ then } \frac{dL}{dw} = 4(2w - 4)$$

- If the gradient is **positive**, the slope is going up – move **left**.
- If the gradient is **negative**, the slope is going down – move **right**.

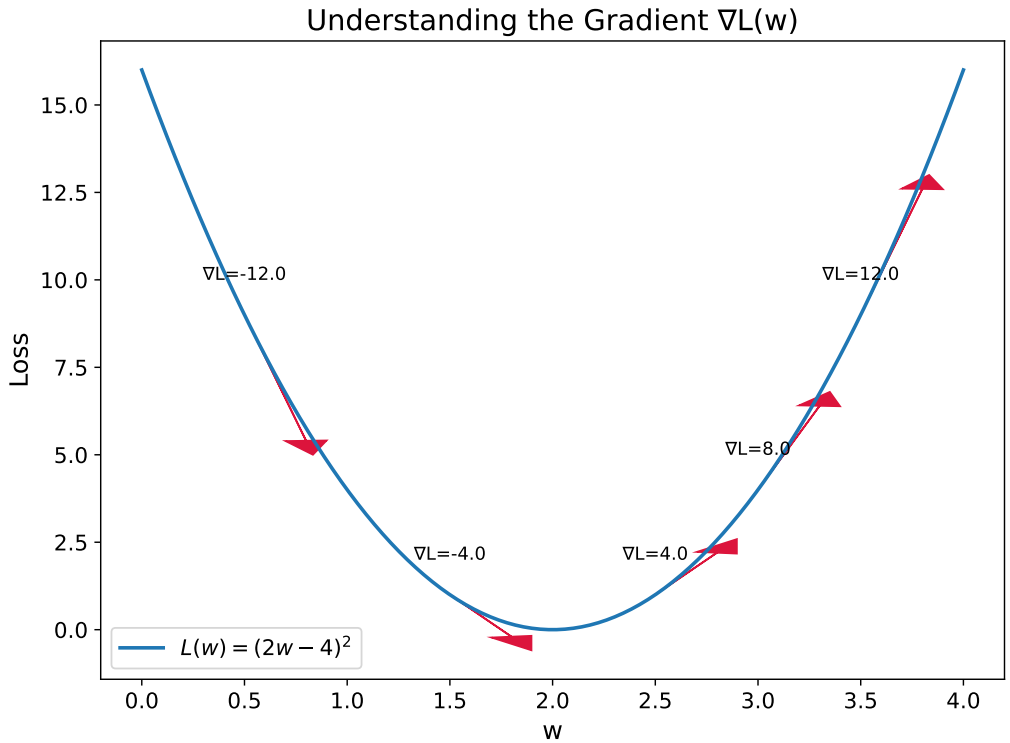
Real-World Analogy:

- Imagine you're on a hill with fog and no map.
- The gradient tells you the direction of steepest uphill.
- To go downhill and reach the minimum, you move in the opposite direction.

What Is a Gradient? (Conceptual View)

Takeaway

In optimization, we use the **negative gradient** to update model parameters and reduce the loss.



What Is a Gradient? (In Code)

Let's see how gradients are computed automatically using PyTorch's auto-differentiation.

Computing Gradients in PyTorch

```
import torch

w = torch.tensor(2.0, requires_grad=True)
loss = (2 * w - 4)**2

loss.backward()
print(w.grad)  # Output: 8.0
```

What Is a Gradient? (In Code)

Explanation:

- The function is: `(2 * w - 4)^2`
- When `w = 2`, then `(2 * 2 - 4)^2 = 0`
- The gradient is: `dL/dw = 4(2w - 4)`, so when `w = 2`, it's 8

The gradient tells us how the loss would change if we slightly change the parameter *w*. Since the gradient is positive, decreasing *w* would reduce the loss–this is the core idea behind gradient descent.

Auto-Differentiation

Modern libraries like PyTorch and TensorFlow compute gradients automatically using a technique called **auto-diff**.

Summary: What Is a Gradient?

Concept	Explanation
Gradient	Slope that indicates direction of steepest increase
Negative Gradient	Used to move toward lower loss (descent)
Auto-Differentiation	Libraries compute gradients automatically

Current Section

Gradient Descent: The Core Idea

Once we know the gradient, we can update the model's parameters to reduce the loss.

Gradient Descent Update Rule

$$\theta \leftarrow \theta - \eta \cdot \nabla_\theta L(\theta)$$

What each term means:

- θ – model parameters (e.g., weights and biases)
- η – learning rate (step size)
- $\nabla_\theta L(\theta)$ – gradient of the loss with respect to the parameters

Key idea: Move in the direction that decreases the loss.

Reminder

We use the **negative gradient** because we want to minimize the loss.

Numerical Example: One Step of Gradient Descent

Suppose we have:

$$L(w) = (2w - 4)^2$$

Then:

$$\frac{dL}{dw} = 4(2w - 4)$$

Assume:

- Initial weight: $w = 1$
- Learning rate: $\eta = 0.1$

Step-by-step:

- Compute gradient: $4(2 \cdot 1 - 4) = -8$
- Update: $w \leftarrow 1 - 0.1 \cdot (-8) = 1 + 0.8 = 1.8$

After One Step

The parameter moved from 1 to 1.8 – we are heading toward the minimum.

Real-World Analogy

Imagine pushing a shopping cart down a slope.

- The slope tells you which direction the cart will roll.
- A steeper slope means faster movement (larger gradient).
- You adjust the force (step size) with your foot – that's the learning rate.

Link to Machine Learning

- Cart = model parameters
- Slope = gradient
- Speed of adjustment = learning rate

Manual Gradient Descent in Python

Let's simulate gradient descent manually for a few steps.

Manual Gradient Descent (1D)

```
w = 1.0                # Initial value
lr = 0.1               # Learning rate

for step in range(3):
    grad = 4 * (2 * w - 4)
    w = w - lr * grad
    print(f"Step {step + 1}: w = {w}")
```

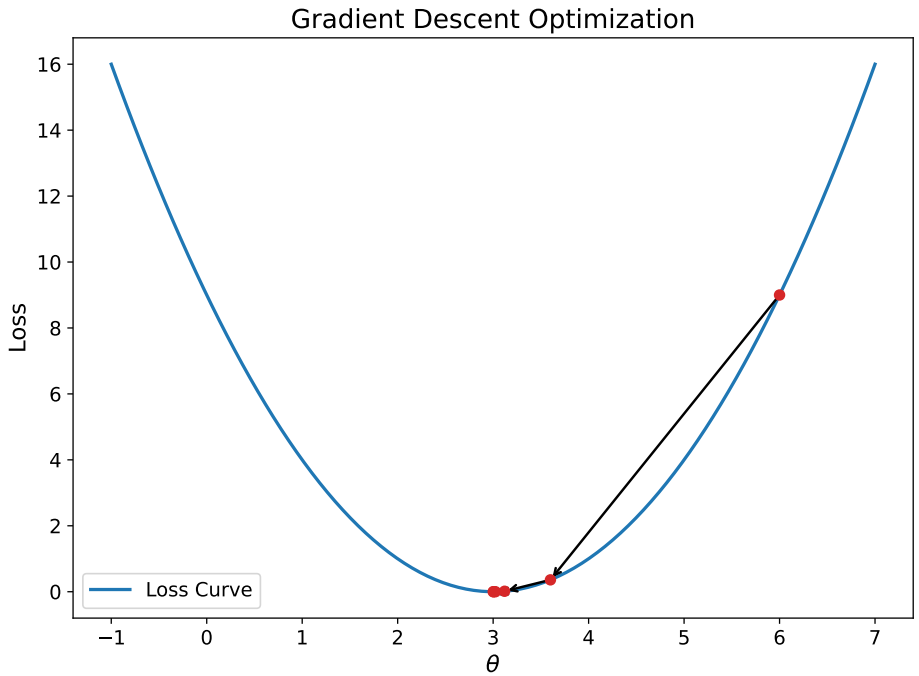
Expected Output:

- Step 1: $w = 1.8$
- Step 2: $w = 2.28$
- Step 3: $w = 2.568$

Manual Gradient Descent in Python

Observation

Each step moves the parameter closer to the value that minimizes the loss.



Summary: Basic Gradient Descent Algorithm

Term	Meaning
θ	Model parameters (weights, biases)
η (Learning Rate)	Step size for parameter update
$\nabla_\theta L(\theta)$	Gradient of the loss with respect to θ
Update Rule	$\theta \leftarrow \theta - \eta \cdot \nabla_\theta L(\theta)$

Why Stochastic Gradient Descent?

How do we train a model when computing the full gradient is too costly or memory-intensive?

Definition

Stochastic Gradient Descent (SGD) is an optimization technique that updates model parameters incrementally using a single or mini-batch sample from the dataset rather than the full dataset.

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla_\theta \mathscr{L}(\theta_t; x^{(i)}, y^{(i)})$$

Best Practice

Use mini-batches (e.g., size 32–256) to balance noisy updates and computational efficiency.

Why Stochastic Gradient Descent?

Pitfall

Pure SGD (without momentum) can oscillate or converge slowly, especially on complex loss surfaces.

SGD Update Rules & Parameters Explained

Standard SGD:

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla_\theta \mathscr{L}(\theta_t)$$

SGD with Momentum:

$$v_{t+1} = \mu v_t - \eta \cdot \nabla_\theta \mathscr{L}(\theta_t), \quad \theta_{t+1} = \theta_t + v_{t+1}$$

Parameter Breakdown

- θ_t – Model parameters at time t
- η – Learning rate: controls step size (e.g., 0.01)
- $\nabla_\theta \mathscr{L}$ – Gradient of the loss function
- v_t – Velocity term (momentum)
- μ – Momentum coefficient (commonly 0.9)

Warning

Too high a learning rate (η) may cause divergence; too low slows convergence.

TensorFlow Implementation of SGD

Plain SGD (No Momentum)

```
from tensorflow.keras.optimizers import SGD

optimizer = SGD(learning_rate=0.01)
```

SGD with Momentum

```
optimizer = SGD(learning_rate=0.01, momentum=0.9)
```

SGD with Nesterov Acceleration

```
optimizer = SGD(learning_rate=0.01, momentum=0.9, nesterov=True)
```

Output Notes

All versions will be passed to the ‘compile‘ method:

```
model.compile(optimizer=optimizer, loss='mse')
```

SGD: Summary Table

Element	Description
Core Formula	$\theta_{t+1} = \theta_t - \eta \cdot \nabla_\theta \mathscr{L}$
Key Hyperparameters	`learning_rate`, `momentum`, `nesterov`
Momentum Rule	$v_{t+1} = \mu v_t - \eta \nabla \mathscr{L}$; $\theta_{t+1} = \theta_t + v_{t+1}$
When to Use	Suitable for large datasets, online learning, and when faster methods (e.g., Adam) are too unstable
Common Learning Rate	0.01 to 0.1 for vanilla SGD; may be lower with momentum or large batches
Strengths	Simple, memory-efficient, online-capable
Weaknesses	Can be slow and noisy without tuning or momentum

Why Adam?

How can we combine the benefits of SGD with momentum and adaptive learning rates in one method?

Definition

Adam (Adaptive Moment Estimation) is an optimization algorithm that computes adaptive learning rates for each parameter by maintaining exponentially decaying averages of past gradients (first moment) and squared gradients (second moment).

$$\theta_{t+1} = \theta_t - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon}$$

Use Case

Adam works well in most practical deep learning scenarios and is generally the first-choice optimizer.

Why Adam?

Pitfall

Despite fast convergence, Adam may result in slightly poorer generalization than SGD with momentum on some tasks.

Adam Update Rule & Parameter Breakdown

Adam maintains two moving averages:

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

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

$$\theta_{t+1} = \theta_t - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon}$$

Term Breakdown

- g_t – Current gradient of the loss function at time step t
- m_t – Exponentially weighted average of past gradients (1st moment)
- v_t – Exponentially weighted average of past squared gradients (2nd moment)
- β_1 – Decay rate for m_t (usually 0.9)
- β_2 – Decay rate for v_t (usually 0.999)
- η – Learning rate (step size), typically 0.001
- ε – Small constant to prevent division by zero (e.g., 10^{-7})

TensorFlow Implementation of Adam

Default Adam Optimizer

```
from tensorflow.keras.optimizers import Adam

optimizer = Adam()  # Uses default: lr=0.001, beta_1=0.9,
    beta_2=0.999
```

Adam with Custom Settings

```
optimizer = Adam(
    learning_rate=0.0003,
    beta_1=0.85,
    beta_2=0.995,
    epsilon=1e-8
)
```

Adam: Summary Table

Element	Description
Update Formula	$\theta_{t+1} = \theta_t - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon}$
Gradient Estimate (m_t)	Mean of past gradients, weighted by β_1
Variance Estimate (v_t)	Mean of squared gradients, weighted by β_2
Bias Correction	$\hat{m}_t = m_t/(1-\beta_1^t)$ and $\hat{v}_t = v_t/(1-\beta_2^t)$
Key Parameters	`learning_rate`, `beta_1`, `beta_2`, `epsilon`
When to Use	Most deep learning models (CV, NLP, RL, etc.)
Strengths	Fast convergence, adaptive per-parameter updates
Weaknesses	May generalize worse than SGD in some cases

Why RMSProp?

How do we handle noisy, non-stationary gradients in deep models like RNNs or reinforcement learning?

Definition

RMSProp is an adaptive learning rate optimization algorithm that maintains a moving average of the squared gradients and divides the current gradient by the root of this average.

Use Case

RMSProp is especially effective for recurrent neural networks and non-stationary objectives.

Why RMSProp?

Pitfall

RMSProp uses only the second moment (squared gradients), unlike Adam which uses both first and second. RMSProp doesn't use momentum by default; combining both (as in some TensorFlow versions) must be explicit.

RMSProp Update Rule & Parameter Breakdown

Exponential moving average of squared gradients:

$$E[g^2]_t = \rho \cdot E[g^2]_{t-1} + (1-\rho) \cdot g_t^2$$

Update rule:

$$\theta_{t+1} = \theta_t - \eta \cdot \frac{g_t}{\sqrt{E[g^2]_t + \varepsilon}}$$

Term Breakdown

- θ_t – Current parameter value at step t
- η – Learning rate (typically 0.001 to 0.01)
- g_t – Gradient of the loss with respect to θ at time t
- $E[g^2]_t$ – Running average of squared gradients
- ρ – Decay rate for moving average (default: 0.9)
- ε – Small constant to ensure numerical stability (e.g., 10^{-7})

TensorFlow Implementation of RMSProp

Default RMSProp Optimizer

```
from tensorflow.keras.optimizers import RMSprop

optimizer = RMSprop()  # Defaults: lr=0.001, rho=0.9, epsilon=1e-7
```

Custom RMSProp Configuration

```
optimizer = RMSprop(
    learning_rate=0.0005,
    rho=0.95,
    momentum=0.9,
    epsilon=1e-8,
    centered=True
)
```

RMSProp: Summary Table

Element	Description
Update Formula	$\theta_{t+1} = \theta_t - \eta \cdot \frac{g_t}{\sqrt{E[g^2]_t + \varepsilon}}$
Squared Gradient Average	$E[g^2]_t = \rho \cdot E[g^2]_{t-1} + (1-\rho) \cdot g_t^2$
Key Parameters	`learning_rate`, `rho`, `epsilon`, `momentum`, `centered`
When to Use	Best for non-stationary problems and RNN training
Common Learning Rate	0.001 (default); lower if gradients are unstable
Strengths	Adaptive step sizes, handles noisy gradients well
Weaknesses	Ignores gradient direction (no first moment)

Current Section

Why Batch Gradient Descent?

How do we make parameter updates using the *entire dataset* at once?
This is useful when:

- Your dataset fits entirely in memory.
- You want smooth, stable convergence.

Optimization Objective

Given loss function $L(\theta)$ over dataset $\mathscr{D} = \{(x_i, y_i)\}_{i=1}^{N}$, we want to minimize:

$$L(\theta) = \frac{1}{N} \sum_{i=1}^{N} \ell(x_i, y_i; \theta)$$

Gradient Formula for Batch GD

Full Gradient Computation

$$\nabla_\theta L(\theta) = \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta \ell(x_i, y_i; \theta)$$

Update Rule

$$\theta \leftarrow \theta - \eta \cdot \nabla_\theta L(\theta)$$

Where:

- η is the learning rate
- $\nabla_\theta L(\theta)$ is the average gradient over all samples

Numerical Example: Batch Gradient Descent

Suppose we want to fit a simple linear model: $y = \theta x$
Dataset:

$$\mathscr{D} = \{(1,2),(2,4)\}$$

Loss function: Mean squared error:

$$\ell(x,y;\theta) = \frac{1}{2}(y-\theta x)^2$$

Step-by-step:
1. Compute gradients for each sample:

$$\nabla_\theta \ell(1,2;\theta) = -x(y-\theta x) = -(1)(2-\theta)$$

$$\nabla_\theta \ell(2,4;\theta) = -(2)(4-2\theta)$$

2. Average gradient:

$$\frac{1}{2}[-(2-\theta)-2(4-2\theta)] = -\frac{1}{2}[(2-\theta)+2(4-2\theta)]$$

Numerical Example: Batch Gradient Descent

3. Plug in $\theta = 1$:

$$\text{Gradient} = -\frac{1}{2}[(2-1)+2(4-2)] = -\frac{1}{2}[1+4] = -2.5$$

4. Update with $\eta = 0.1$:

$$\theta \leftarrow 1 - 0.1 \cdot (-2.5) = 1.25$$

Python Implementation: Batch Gradient Descent

Simple Batch GD for Linear Regression

```
# Data: X = [1, 2], Y = [2, 4]
X = [1, 2], Y = [2, 4], theta = 1.0, eta = 0.1  # Learning rate
# Compute gradients over the full batch
grads = []
for x, y in zip(X, Y):
    pred = theta * x
    grad = -x * (y - pred)
    grads.append(grad)
# Average the gradients
avg_grad = sum(grads) / len(grads)
# Update rule
theta = theta - eta * avg_grad
print(theta)  # Expected: 1.25
```

Explanation

This code mimics the manual gradient derivation:

- Computes gradients for each sample.
- Averages them (batch update).
- Applies the learning rule: $\theta \leftarrow \theta - \eta \cdot \nabla_\theta L$

Pros and Cons of Batch GD

Pros

- Deterministic and stable updates
- Converges smoothly to a minimum

Cons

- Slow on large datasets
- Requires entire dataset in memory

Why Stochastic Gradient Descent (SGD)?

How can we update parameters when we canât wait for the whole dataset?
SGD is useful when:

- The dataset is too large to load all at once.
- You want faster, frequent updates.
- You're working in an online or streaming context.

Optimization Objective

We still aim to minimize:

$$L(\theta) = \frac{1}{N} \sum_{i=1}^{N} \ell(x_i, y_i; \theta)$$

But instead of averaging over all samples, we use a single random sample.

SGD Gradient Formula

Single-Sample Estimate

At each iteration, select (x_i, y_i) randomly:

$$\nabla_\theta L(\theta) \approx \nabla_\theta \ell(x_i, y_i; \theta)$$

SGD Update Rule

$$\theta \leftarrow \theta - \eta \cdot \nabla_\theta \ell(x_i, y_i; \theta)$$

This leads to high variance updates but faster iterations.

Numerical Example: Stochastic GD

We again fit a model: $y = \theta x$

Dataset:

$$\mathscr{D} = \{(1,2),(2,4)\}$$

Loss:

$$\ell(x,y;\theta) = \frac{1}{2}(y-\theta x)^2$$

Step-by-step:

1. Pick a sample at random, say $(2,4)$
2. Compute gradient:
$$\nabla_\theta \ell = -x(y-\theta x) = -2(4-2\theta)$$
3. Plug in $\theta = 1$:
$$\nabla_\theta \ell = -2(4-2) = -4$$
4. Update:
$$\theta \leftarrow 1 - 0.1 \cdot (-4) = 1.4$$

Python Implementation: Stochastic Gradient Descent

SGD on One Random Sample

```
import random
# Dataset
X = [1, 2], Y = [2, 4], theta = 1.0, eta = 0.1
# Randomly pick one sample
i = random.choice([0, 1])
x, y = X[i], Y[i]
# Compute gradient
grad = -x * (y - theta * x)
# Update
theta = theta - eta * grad
print(theta)  # Could be 1.2 or 1.4 depending on sample
```

Explanation

Only one sample is used to compute the gradient. This speeds up training but may lead to erratic convergence unless averaged over time.

Pros and Cons of SGD

Pros

- Fast and efficient on large datasets
- Can escape shallow local minima

Cons

- High variance in parameter updates
- May oscillate and not converge without decay or momentum

Why Mini-Batch Gradient Descent?

How can we balance stability and speed when optimizing deep models?
Mini-Batch Gradient Descent is effective when:

- Your data doesn't fit in memory all at once.
- You want faster convergence than full batch.
- You still want some gradient averaging for stability.

Key Idea

Instead of using one sample (SGD) or all samples (batch), we use a small subset:

$$\mathscr{B} = \{(x_1, y_1), \ldots, (x_m, y_m)\} \quad \text{with } m \ll N$$

Mini-Batch Gradient Formula

Mini-Batch Gradient Estimate

$$\nabla_\theta L(\theta) \approx \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta \ell(x_i, y_i; \theta)$$

where m is the mini-batch size.

Update Rule

$$\theta \leftarrow \theta - \eta \cdot \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta \ell(x_i, y_i; \theta)$$

This method approximates batch gradient descent but is much faster.

Numerical Example: Mini-Batch GD

Model: $y = \theta x$

Dataset:

$$\mathscr{D} = \{(1,2),(2,4),(3,6)\}$$

Use mini-batch of size $m = 2$: select batch $\{(1,2),(3,6)\}$

Loss:

$$\ell(x,y;\theta) = \frac{1}{2}(y - \theta x)^2$$

1. Compute gradients:

$$\nabla_\theta \ell(1,2) = -(1)(2-\theta)$$

$$\nabla_\theta \ell(3,6) = -3(6-3\theta)$$

2. Plug in $\theta = 1$:

$$\text{Grad } 1 = -1(2-1) = -1, \quad \text{Grad } 2 = -3(6-3) = -9$$

3. Average gradient:

$$\frac{-1+(-9)}{2} = -5$$

4. Update:

$$\theta \leftarrow 1 - 0.1 \cdot (-5) = 1.5$$

Python Implementation: Mini-Batch Gradient Descent

Mini-Batch GD with Manual Batch Selection

```
# Dataset
X = [1, 2, 3], Y = [2, 4, 6], theta = 1.0, eta = 0.1
# Select mini-batch (e.g. indices [0, 2])
batch_indices = [0, 2]
grads = []
for i in batch_indices:
    x, y = X[i], Y[i]
    pred = theta * x
    grad = -x * (y - pred)
    grads.append(grad)
# Average gradient
avg_grad = sum(grads) / len(grads)
# Update
theta = theta - eta * avg_grad
print(theta)  # Expected: 1.5
```

Explanation

This code selects a small batch, computes gradients for each example, averages them, and updates θ. It is both memory-efficient and faster than full batch updates.

Pros and Cons of Mini-Batch GD

Pros

- Balances speed and convergence stability
- Works well with vectorized GPU operations
- Most commonly used in practice

Cons

- Sensitive to batch size choice
- May require tuning learning rate and scheduling

Unified Summary: All Types of Gradient Descent

Aspect	**Batch Gradient Descent**	**Stochastic Gradient Descent (SGD)**
Gradient Computation	$\frac{1}{N}\sum_{i=1}^{N}\nabla_\theta \ell(x_i, y_i; \theta)$	$\nabla_\theta \ell(x_i, y_i; \theta)$ for random i
Stability	Very stable; smooth convergence	High variance; unstable without tuning
Performance	Slow and memory-intensive	Very fast; scalable to large data
Aspect	**Mini-Batch Gradient Descent**	
Gradient Computation	$\frac{1}{m}\sum_{i=1}^{m}\nabla_\theta \ell(x_i, y_i; \theta)$ for $m \ll N$	
Stability	Balanced; more stable than SGD, faster than Batch GD	
Performance	Highly parallelizable; ideal for deep learning	

Current Section

What Is the Learning Rate?

The **learning rate** (η) controls how big a step we take in the direction of the negative gradient.

$$\theta \leftarrow \theta - \eta \cdot \nabla_\theta L(\theta)$$

Why it matters:

- If η is too **small** – learning is very slow.
- If η is too **large** – updates may overshoot or diverge.
- If η is just right – smooth convergence to minimum.

Key Insight

The learning rate is one of the most important hyperparameters in optimization.

Visualizing the Effect of Learning Rate

Imagine a bowl-shaped loss surface:

- A small η means taking tiny steps downhill.
- A large η might jump across the bowl and oscillate.
- A well-chosen η follows a smooth curved path to the minimum.

Real-world analogy:

- You're trying to descend a mountain trail in fog.
- If your steps are too short, you'll take forever to get down.
- If they're too long, you might fall or miss the trail.

What You Want

A step size that adapts to the slope and reaches the goal efficiently.

Numerical Experiment: Different Learning Rates

Try three different learning rates to see how they behave.

Simulating Learning Rate Behavior

```
w = 1.0
for step in range(10):
    grad = 4 * (2 * w - 4)
    w = w - 0.01 * grad  # Try 0.01, 0.1, 1.0
    print(f"Step {step+1}: w = {w}")
```

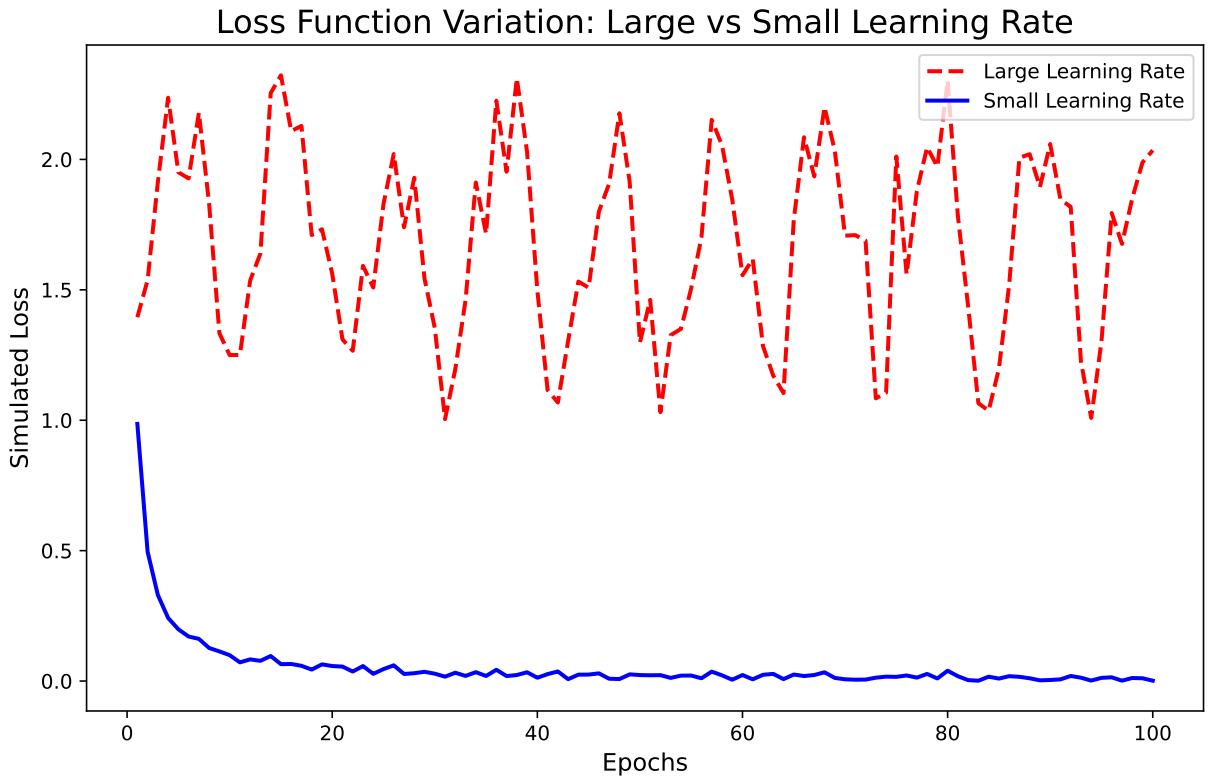
Observation:

- $\eta = 0.01$: very slow movement
- $\eta = 0.1$: fast and smooth convergence
- $\eta = 1.0$: diverges or oscillates wildly

Warning

Poor learning rate selection can waste computation or ruin optimization.

Numerical Experiment: Different Learning Rates



What is Learning Rate Decay?

Problem: A fixed learning rate can overshoot near minima or get stuck in noisy regions.

Solution: Learning Rate Decay

Gradually reduce the learning rate over time to allow fine-tuned updates as training progresses.

Common Decay Methods

- **Step Decay:** Reduce every k epochs
- **Exponential Decay:** $\eta_t = \eta_0 \cdot \gamma^t$
- **Inverse Time Decay:** $\eta_t = \frac{\eta_0}{1+kt}$

Why It Matters

Large learning rates help escape flat regions early. Smaller rates improve convergence near minima.

Numerical Example: Learning Rate Decay

Goal: Show how decayed learning rates change parameter updates.
Decay Type: Inverse Time Decay

$$\eta_t = \frac{\eta_0}{1 + kt}, \quad \eta_0 = 0.1,\ k = 0.1$$

Step t	**Learning Rate** η_t	**Update:** $\theta_{t+1} = \theta_t - \eta_t \cdot g$
0	$\frac{0.1}{1+0.1\cdot 0} = 0.1$	Large step
1	$\frac{0.1}{1+0.1\cdot 1} = 0.0909$	Slightly smaller step
5	$\frac{0.1}{1+0.1\cdot 5} = 0.0666$	Smaller step again
10	$\frac{0.1}{1+0.1\cdot 10} = 0.05$	Half of original step size

Effect

Early training: Large learning rates explore the space.
Later training: Small learning rates refine the minima.

Summary: The Role of the Learning Rate

Setting	Outcome
Too Small	Converges slowly; inefficient training
Too Large	Overshoots minimum; may diverge
Well-Tuned	Efficient descent toward optimum

Current Section

Why Use Momentum?

Even with the right learning rate, gradient descent can struggle in narrow or curved regions of the loss surface.

Problem: Zig-Zagging

- In steep but narrow valleys, gradients keep changing direction.
- Updates may oscillate back and forth instead of moving forward.

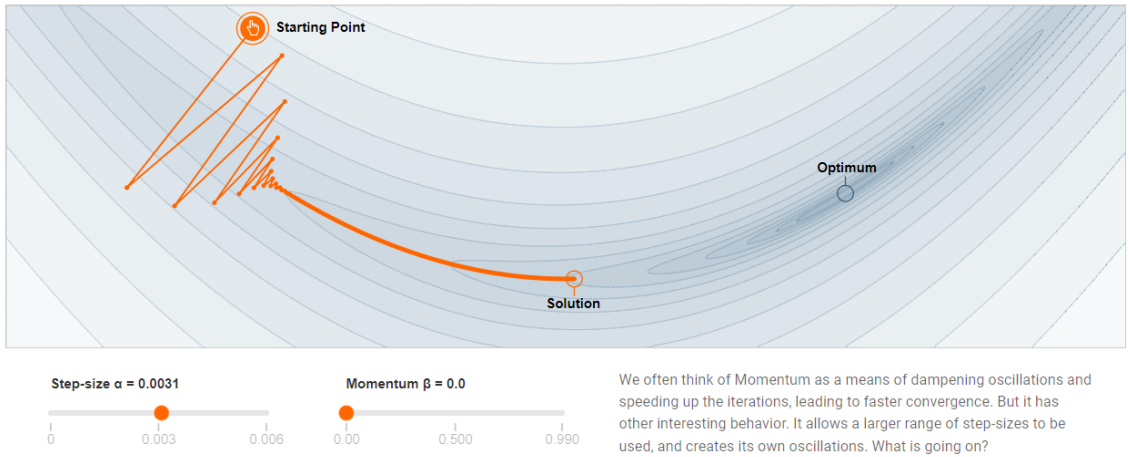
Solution: Momentum

Instead of updating based only on the current gradient, we **accumulate past gradients** to build velocity.

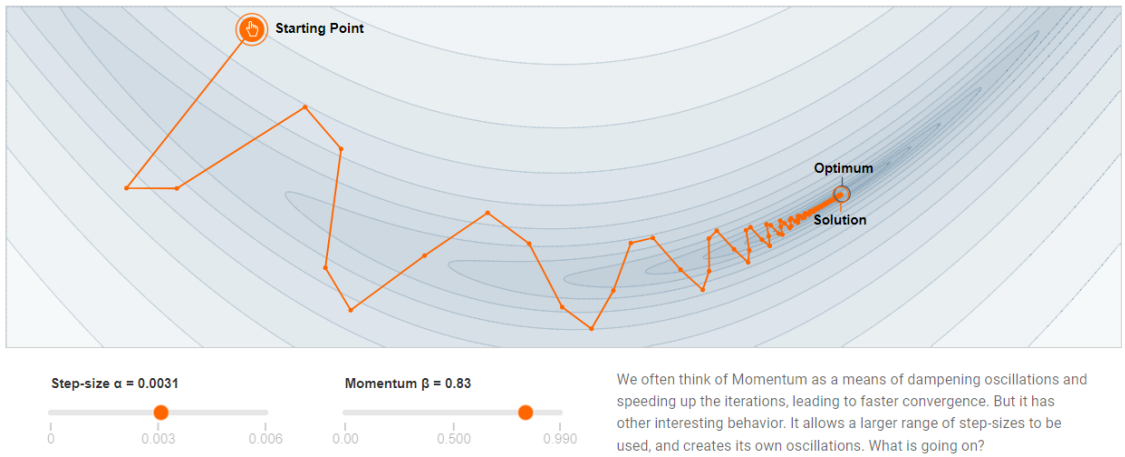
Effect:

- Speeds up movement along long slopes
- Dampens oscillations in noisy or curved regions

Why Use Momentum?



Adapted from [1] (CC-BY 2.0).



Adapted from [1] (CC-BY 2.0).

How Momentum Works

Momentum introduces a velocity term v_t that accumulates a moving average of gradients.

$$v_t = \beta v_{t-1} + \eta \cdot \nabla_\theta L(\theta)$$

$$\theta \leftarrow \theta - v_t$$

Where:

- β is the momentum coefficient (e.g., 0.9)
- v_t acts like inertia, carrying updates forward

When to Use It

Momentum helps when:

- You see slow progress in flat regions
- You observe zig-zag behavior in loss curves

Real-World Analogy: Rolling Ball

Imagine:

- You're rolling a ball down a hill.
- On flat terrain, it moves slowly.
- On a steep slope, it picks up speed.
- If the path curves, it doesn't change direction immediately – it has momentum.

Optimization Analogy

- The ball = parameter values
- The hill = loss surface
- Gravity + inertia = gradient + momentum

Benefit: Momentum accelerates learning in consistent directions and reduces noise in updates.

Gradient Descent With and Without Momentum

Manual Momentum Simulation

```
w = 1.0, v = 0.0, eta = 0.1, beta = 0.9
for step in range(5):
    grad = 4 * (2 * w - 4)
    v = beta * v + eta * grad
    w = w - v
    print(f"Step {step+1}: w = {w}")
```

Observation:

- Early steps may be small, but velocity builds up.
- Fewer oscillations than regular gradient descent.

Tuning Tip

Start with $\beta = 0.9$ and $\eta = 0.01$ to 0.1 depending on the task.

Numerical Example: With vs Without Momentum

Function: $f(w) = (2w-4)^2, \quad \nabla f(w) = 4(2w-4)$
Hyperparameters: $\eta = 0.1$, $\beta = 0.9$, initial $w_0 = 1.0$

Gradient Descent Without Momentum

Step	Gradient	Update (w)
0	$4(2 \cdot 1.0 - 4) = -8$	$1.0 + 0.8 = 1.8$
1	$4(2 \cdot 1.8 - 4) = 7.2$	$1.8 - 0.72 = 1.08$
2	$4(2 \cdot 1.08 - 4) = -7.36$	$1.08 + 0.736 = 1.816$

Wobbles back and forth due to steep curvature.

Gradient Descent With Momentum ($\beta = 0.9$)

Step	Gradient	Velocity (v)	Update (w)
0	-8.0	$v = 0.1 \cdot (-8) = -0.8$	$1.0 + 0.8 = 1.8$
1	7.2	$v = 0.9 \cdot (-0.8) + 0.1 \cdot 7.2 = -0.72 + 0.72 = 0$	$1.8 - 0 = 1.8$
2	7.2	$v = 0.9 \cdot 0 + 0.1 \cdot 7.2 = 0.72$	$1.8 - 0.72 = 1.08$

Momentum damps reversal and gradually builds directional speed.

Summary: Momentum

Concept	Description
Momentum Term (v_t)	Stores past gradients to build velocity
Benefit	Reduces oscillations; speeds convergence on consistent gradients
Analogy	Like a ball rolling downhill with inertia

What is Nesterov Momentum?

Problem: In classical momentum, the gradient is computed at the current position θ_t.
Improvement: Nesterov Accelerated Gradient (NAG) computes the gradient *after moving* partway in the direction of momentum.

Nesterov Formula

$$v_t = \beta v_{t-1} + \eta \cdot \nabla_\theta L(\theta_t - \beta v_{t-1})$$

$$\theta_{t+1} = \theta_t - v_t$$

Key Insight

Instead of blindly following the momentum, we "look ahead" before calculating the gradient. This improves stability and convergence near minima.

Numerical Example: Nesterov Momentum vs Classical Momentum

Let: $\eta = 0.1$, $\beta = 0.9$, $w_0 = 1.0$, $v_0 = 0$
Loss: $f(w) = (2w - 4)^2$, gradient: $\nabla f(w) = 4(2w - 4)$

Classical Momentum (Step 1)

$$\begin{aligned} \text{Gradient at } w_0 &= 4(2 \cdot 1 - 4) = -8 \\ v_1 &= 0.9 \cdot 0 + 0.1 \cdot (-8) = -0.8 \\ w_1 &= 1.0 - (-0.8) = 1.8 \end{aligned}$$

Numerical Example: Nesterov Momentum vs Classical Momentum

Nesterov Momentum (Step 1)

$$\text{Lookahead point: } w_0 - \beta v_0 = 1.0$$
$$\text{Gradient at lookahead: } \nabla f(1.0) = -8$$
$$v_1 = 0.9 \cdot 0 + 0.1 \cdot (-8) = -0.8$$
$$w_1 = 1.0 - (-0.8) = 1.8$$

Same result here, but the gradient came from the lookahead.

In later steps: Nesterov avoids overshooting and stabilizes better as v_t grows.

Current Section

When to Use Which Gradient Descent Variant?

- **Batch Gradient Descent** – Use for very small datasets where memory is not a concern.
- **SGD (Stochastic)** – Use for online learning or streaming data.
- **Mini-Batch Gradient Descent** – Use as the **default choice** in almost all deep learning tasks.

Default Starting Points (Recommended)

- Learning rate $\eta = 0.01$ to 0.1
- Mini-batch size = 32 to 128
- Momentum $\beta = 0.9$

Common Mistakes to Avoid

Watch Out For

- Using too high a learning rate without testing convergence
- Not shuffling data during mini-batch training
- Confusing gradient noise with model instability
- Forgetting to zero out gradients between steps (in PyTorch: `optimizer.zero_grad()`)

Good Habits

- Always visualize your loss curve
- Try learning rate scheduling if stuck
- Start simple, then experiment with optimizers

Session Summary: Gradient Descent and Variants

In this session, you learned:

- Why manual tuning doesn't scale to real-world ML
- What a gradient is, and how it points toward error reduction
- How basic gradient descent works mathematically and numerically
- The difference between batch, stochastic, and mini-batch updates
- How learning rate and momentum impact convergence

Next Steps

In the next session, we will explore popular **adaptive optimizers** like Adam, RMSProp, and learning rate schedulers that help training deep models converge faster and more reliably.

Summary: Practical Gradient Descent Guidelines

Recommendation	Context or Reason
Use Mini-Batch GD	Best balance for deep learning efficiency and stability
Tune Learning Rate	Critical for convergence; try scheduling if needed
Monitor Loss Curves	Helps detect divergence, plateaus, or overfitting

References

[1] Gabriel Goh.
Why momentum really works.
Distill, 2017.