Gradient Descent and Its Variants

Course:
INFO-6154 Data Science and Machine Learning

Developed by:
Mohammad Noorchenarboo

May 3, 2025

Current Section

1. Gradient Descent and Its Variants
   - Why We Need Gradient Descent
   - What Is a Gradient?
   - Basic Gradient Descent Algorithm
   - Types of Gradient Descent
   - The Role of the Learning Rate
   - Momentum
   - Practical Guidelines and Summary

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)
```

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**.

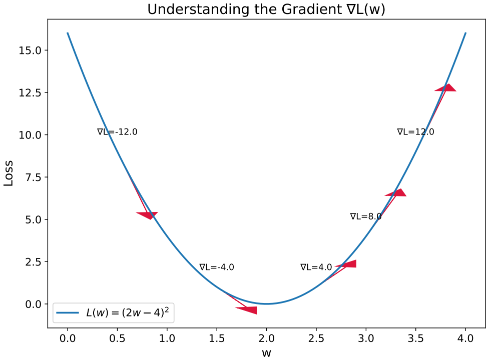
Real-World Analogy:

- 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**.

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)

```python
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}")
```

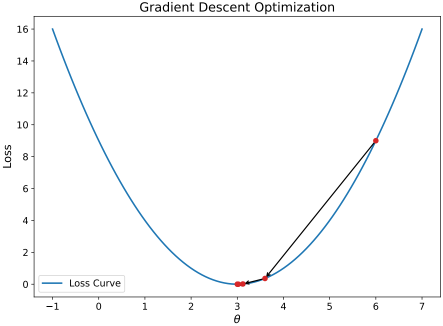
Expected Output:

- 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.



Types of Gradient Descent

Gradient Descent can be applied in different ways depending on how much data is used in each update step.

The Three Main Types:

- **Batch Gradient Descent** – Uses the entire dataset per update
- **Stochastic Gradient Descent (SGD)** – Uses a single sample per update
- **Mini-Batch Gradient Descent** – Uses a small batch of samples (e.g., 32–256)

Types of Gradient Descent

What Changes?

All three follow the same update rule:

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

The only difference is how the gradient is estimated: from the full data, one sample, or a small batch.

Batch Gradient Descent

Description:

- Computes the gradient using the entire training set.
- Very stable and smooth updates.
- Computationally expensive for large datasets.

Example Use Case:

- Small tabular datasets (e.g., diabetes dataset with 300 samples).

Pros

Very stable convergence – ideal for theoretical analysis.

Cons

Slow and memory-intensive for large datasets.

Stochastic Gradient Descent (SGD)

Description:

- Uses only one training example per update.
- Very noisy, but can escape local minima.
- Often needs careful learning rate tuning and regularization.

Example Use Case:

- Online learning or streaming data (e.g., real-time fraud detection).

Pros

Fast updates, good for large-scale or streaming data.

Cons

High variance in updates – unstable without proper tuning.

Mini-Batch Gradient Descent

Description:

- Uses a small batch of data (commonly 32, 64, or 128 samples).
- Combines benefits of both batch and stochastic approaches.
- Most widely used variant in practice.

Example Use Case:

- Deep learning training on GPUs (e.g., image classification with CIFAR-10).

Pros

Faster than batch; more stable than SGD; highly parallelizable.

Cons

Still depends on batch size and tuning learning rate.

Comparison Table

Method	Data Used per Step	Speed	Stability
Batch GD	All data	Slow	Very stable
SGD	One sample	Fast	Very noisy
Mini-Batch GD	Small batch	Medium	Medium-high

In Practice

Mini-Batch Gradient Descent is the standard approach in modern deep learning.

Note

Mini-batch training uses a `DataLoader` to iterate through the dataset in chunks.

Example: Batch vs. Mini-Batch Code

Training Loop Comparison

```python
# Batch Gradient Descent
for epoch in range(epochs):
    y_pred = model(X_train)
    loss = loss_fn(y_pred, y_train)
    loss.backward()
    optimizer.step()

# Mini-Batch Gradient Descent
for epoch in range(epochs):
    for batch_X, batch_y in dataloader:
        y_pred = model(batch_X)
        loss = loss_fn(y_pred, batch_y)
        loss.backward()
        optimizer.step()
```

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

```python
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}")
```

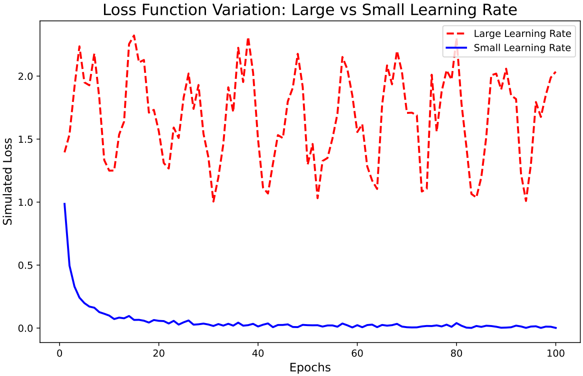
Observation:

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

Numerical Experiment: Different Learning Rates

Warning

Poor learning rate selection can waste computation or ruin optimization.



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.

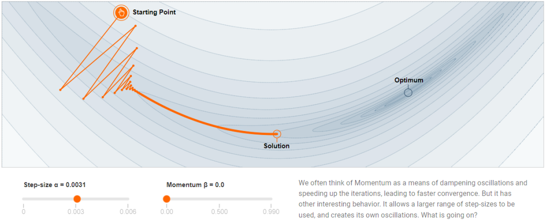
Solution: Momentum

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

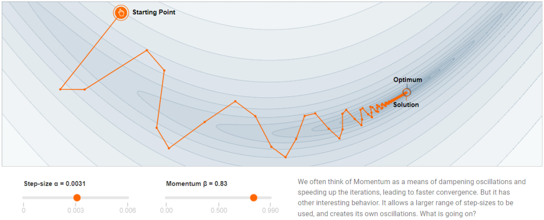
Effect:

- 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.

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 η = 0.01 to 0.1
- Mini-batch size = 32 to 128
- Momentum β = 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.

References

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