

# Machine Learning (CE 40717)

## Fall 2024

Ali Sharifi-Zarchi

CE Department  
Sharif University of Technology

October 18, 2024



- 1 Optimization
- 2 The Loss Surface
- 3 Gradient Descent
- 4 Momentum

# 1 Optimization

Problem Definition

Convexity and Optimization

## 2 The Loss Surface

## 3 Gradient Descent

## 4 Momentum

# 1 Optimization

## Problem Definition

### Convexity and Optimization

## 2 The Loss Surface

## 3 Gradient Descent

## 4 Momentum

# Problem Definition

- **Goal:** Given a function  $f(x)$  of some variable  $x$ , find the value of  $x$  where  $f(x)$  is **minimum** or **maximum**
- In neural networks, the goal is to make the prediction error as small as possible
- We want to find the network weights  $W^*$  that result in the **lowest loss**:

$$w^* = \arg \min_w J(w)$$

# Problem Definition

- Simply put, we want to find the direction to step in that will **reduce our loss** as quickly as possible
- In other words, **which way is downhill?**

# 1 Optimization

Problem Definition

Convexity and Optimization

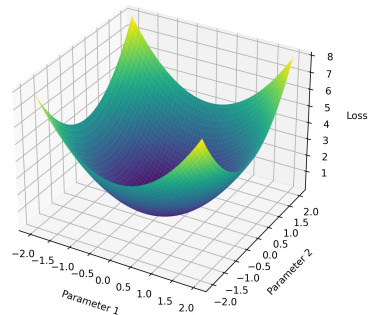
## 2 The Loss Surface

## 3 Gradient Descent

## 4 Momentum

# Convexity and Optimization

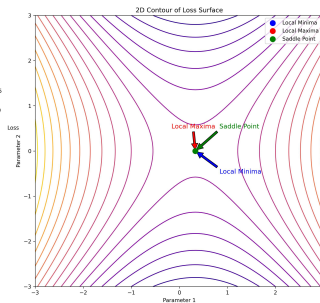
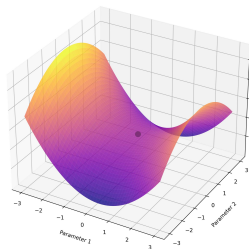
- **Definition:** A function is **convex** if, for any two points on the curve, the line segment connecting them lies above or on the curve
  - **Example:** Bowl-shaped curves
- **Convex functions** are easier to optimize because they have only **one global minimum** (the lowest point)
  - Analytical solutions ( $\nabla f(x) = 0$ ) and second-order methods ( $\nabla^2 f(x) > 0$ ) can be used for faster and more accurate convergence
  - **Gradient descent** is guaranteed to converge to the global minimum in convex functions





# Convexity and Optimization

- **Definition:** a function is **non-convex** if it has multiple local minima and maxima
  - **Global Minimum:** The very lowest point across the whole curve
  - **Local Minimum:** A point that's lower than nearby points, but not the lowest overall
  - **Saddle Points:** A flat region where the slope is almost zero. It can go up in some directions and down in others
- Non-convex functions make finding the **global minimum** complicated



- 1 Optimization
- 2 The Loss Surface**
- 3 Gradient Descent
- 4 Momentum

# Definition

- The loss surface represents how the error changes based on the network's weights
- The loss surface of a neural network is typically **non-convex** due to multiple layers, nonlinear activation functions, and complex interactions between parameters, resulting in multiple local minima and saddle points
- In large networks, most local minima give similar error values and are close to the **global minimum**. This isn't true for smaller networks

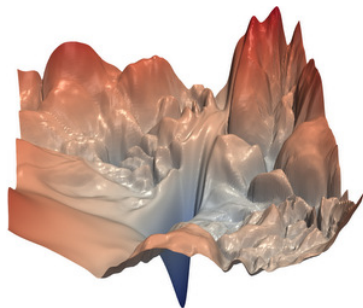
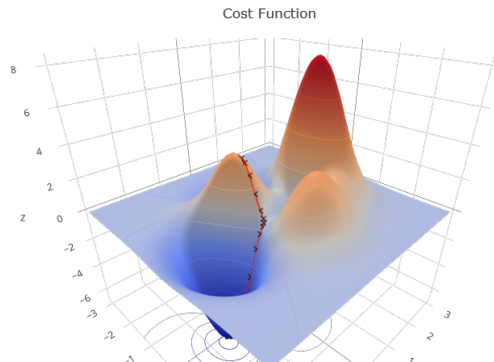


Figure 1: ResNet56 from <https://github.com/tomgoldstein/loss-landscape>

# Loss Optimization

- How can we optimize a non-convex loss function?
- **Strategy:** Instead of randomly searching for a good direction, we calculate the **best direction** to reduce the loss
  - This is mathematically guaranteed to be the direction of the steepest descent



**Figure 2:** Gradient Descent visualization from <https://blog.skz.dev/gradient-descent>

## 1 Optimization

## 2 The Loss Surface

## 3 Gradient Descent

Recap of Gradient Descent

Problems with Gradient Descent

## 4 Momentum

#### ④ Momentum

# Gradient Descent

- As introduced earlier, Gradient Descent is an iterative method for minimizing the error by updating network weights in the direction of the steepest descent (negative gradient)

$$w_{t+1} = w_t - \eta \nabla J(w_t)$$

where:

- $\eta$  is called the 'learning rate' or 'step size'
- The goal is to keep adjusting the weights until we reach a point where the error is as low as possible (a local or global minimum)

# Types of Gradient Descent

- **Batch Gradient Descent:** Uses the entire dataset to calculate the gradient. This gives smooth updates but can be slow
- **Stochastic Gradient Descent (SGD):** Uses one data point at a time, leading to faster but noisier updates
- **Mini-batch Gradient Descent:** Uses small groups (batches) of data points. This is a balance between batch and stochastic, combining speed with more stable updates



# Types of Gradient Descent

Type	Advantages	Disadvantages
<b>Batch</b>	Stable convergence Accurate gradient estimate	Computationally expensive Slow for large datasets
<b>Stochastic (SGD)</b>	Fast updates Can escape local minima	Noisy updates May not converge smoothly
<b>Mini-Batch</b>	Faster than batch, more stable than SGD Efficient for larger datasets	Requires tuning batch size Some noise remains

Table 1: Comparison of Gradient Descent Types

## ① Optimization

## ② The Loss Surface

## ③ Gradient Descent

Recap of Gradient Descent

Problems with Gradient Descent

## ④ Momentum

# Problem 1

- **SGD** is fast and can escape local minima, but it faces some issues
- What if the loss changes quickly in one direction but slowly in another?
  - **Slow** progress along the shallow dimension, **jitter** along the steep direction

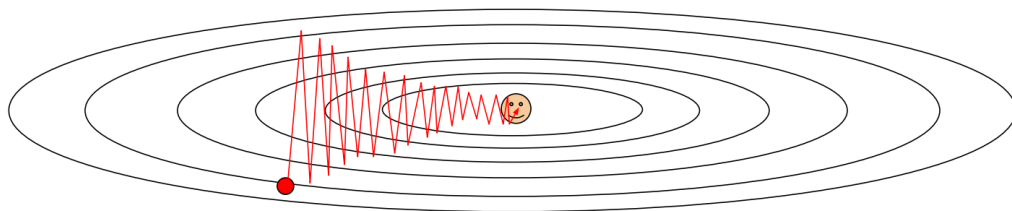


Figure 3: SGD visualization from CS231n: Convolutional Neural Networks for Visual Recognition

## Problem 2

- What if the loss function has a local minima or saddle point?
  - The algorithm may settle for **sub-optimal solutions** or take a **long time** to make significant progress

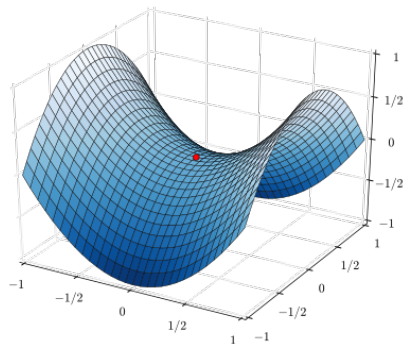


Figure 4: Saddle point from  
[https://en.wikipedia.org/wiki/Saddle\\_point](https://en.wikipedia.org/wiki/Saddle_point)

## Problem 3

- Gradients that come from single data points or mini-batches can be **noisy**

### Stochastic Gradient Descent

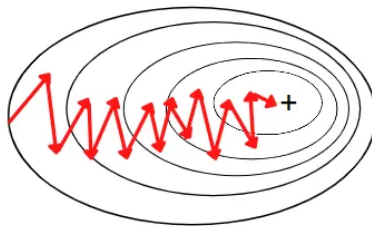


Figure 5: SGD visualization from <https://laptrinhx.com/understanding-optimization-algorithms-3818430905/>

## Problem 4

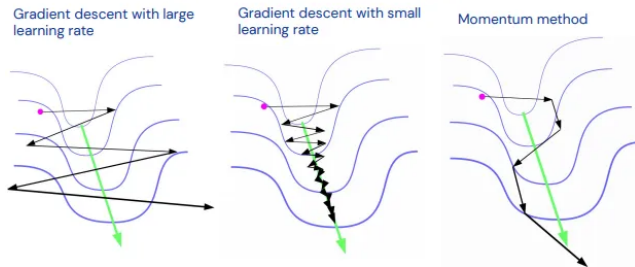
- Why not just use mini-batches?
  - Even though mini-batch gradient descent helps **reduce the noise**, It can still have a **slow convergence** and might **get stuck** in regions where the gradients are small (like plateaus or valleys), which makes learning inefficient
- In addition, using the same learning rate for all dimensions can lead to:
  - Smooth convergence in some directions
  - Oscillations or divergence in other directions

# Problem Definition

- So, how can we improve the vanilla Gradient Descent algorithm?
- **Proposal:**
  - Track oscillations in each direction
  - Increase steps in stable directions
  - Decrease steps in oscillating directions

# Problem Definition

- **Goal:** Choose an appropriate learning rate to avoid slow convergence and getting stuck in local minima while not overshooting or becoming unstable
- **Naive Approach:** Try lots of different learning rates and see what works "just right"
  - This can be inefficient and time-consuming
- **Smarter Approach:** Design an adaptive learning rate that adapts to the loss surface
  - We can achieve this by incorporating **Momentum**





- 1 Optimization
- 2 The Loss Surface
- 3 Gradient Descent

#### ④ Momentum

## Momentum Definition and Types

- 1 Optimization
- 2 The Loss Surface
- 3 Gradient Descent

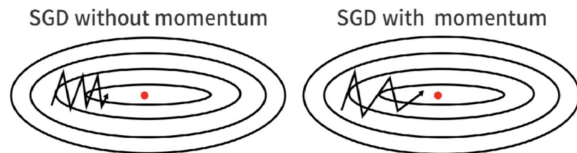
#### ④ Momentum

## Momentum Definition and Types

Adam

# Momentum

- **Goal:** Speed up convergence by using past gradients to smooth out oscillations and avoid getting stuck
  - Keep moving in the same general direction as previous steps
- **Benefit:** Accelerates learning in important directions while reducing oscillations in irrelevant ones



**Figure 6:** SGD comparison from <https://paperswithcode.com/method/sgd-with-momentum>

# First Momentum

- The first momentum, denoted as  $m_t$ , is essentially **the moving average of the gradients**. The update rule is:

$$m_{t+1} = \beta_1 m_t + (1 - \beta_1) \nabla_w J(w_t)$$

$$w_{t+1} = w_t - \eta m_{t+1}$$

where:

- $m_{t+1}$  is the first moment at time step  $t + 1$
- $\beta_1$  is the decay rate, which controls how much of the past gradients to include (typically 0.9 or 0.99)
- $\nabla_w J(w_t)$  is the gradient at time step  $t$

## Why use the first moment?

- Inspired by physics, it keeps us moving due to accumulated momentum, like a ball rolling in a frictionless bowl

## Second Momentum

- The second momentum, denoted as  $v_t$ , is a **moving average of the squared gradients**.
- It helps us track how large the gradients have been over time. The update rule is:

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

$$w_{t+1} = w_t - \eta v_{t+1}$$

where:

- $v_{t+1}$  is the second moment at time step  $t + 1$
- $\beta_2$  is the decay rate
- $\nabla_w J(w_t)$  is the gradient at time step  $t$

### Why use the second moment?

- The second moment helps **control the size of updates** by adjusting for consistently large or small gradients, preventing overshooting or slow learning

# Moment Bias Correction

- **Problem:** When we start training, both  $m_t$  and  $v_t$  are initialized to zero, causing their estimates to be biased toward zero in the early steps, especially when gradients are small.
- **Solution:** We use bias-corrected versions of  $m_t$  and  $v_t$  to fix this:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

- These corrections compensate for the bias by scaling  $m_t$  and  $v_t$  upward, especially in the early steps when  $t$  is small, ensuring more accurate estimates of the moments.

- 1 Optimization
- 2 The Loss Surface
- 3 Gradient Descent

#### ④ Momentum

## Momentum Definition and Types

# Adam

- Adaptive Moment Estimation, or **Adam**, combines the concepts of **momentum** and **adaptive learning rate** by maintaining an exponentially decaying average of both past gradients and squared gradients
- Adam adapts the learning rate for each parameter based on the gradient history
  - Larger gradients lead to smaller update steps, and vice versa



## Adam

- The update rule for the Adam optimizer at step  $t + 1$  is denoted by  $w_{t+1}$ :

$$m_{t+1} = \beta_1 m_t + (1 - \beta_1) \nabla_w J(w_t)$$

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

$$\hat{m}_{t+1} = \frac{m_{t+1}}{1 - \beta_1^{t+1}}, \quad \hat{v}_{t+1} = \frac{v_{t+1}}{1 - \beta_2^{t+1}}$$

$$w_{t+1} = w_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}}$$

where:

- $m_{t+1}$  and  $v_{t+1}$  are the first and second moments at step  $t + 1$
- $\beta_1$  and  $\beta_2$  are decay rates for the first and second moments
- $\hat{m}_{t+1}$  and  $\hat{v}_{t+1}$  are bias-corrected estimates of the first and second moments
- $\epsilon$  is a small constant to prevent division by zero

# Comparison of Momentum Methods

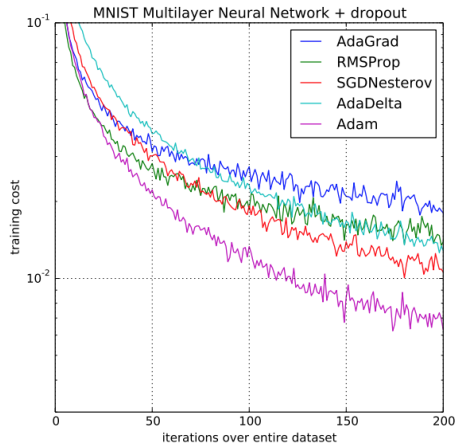
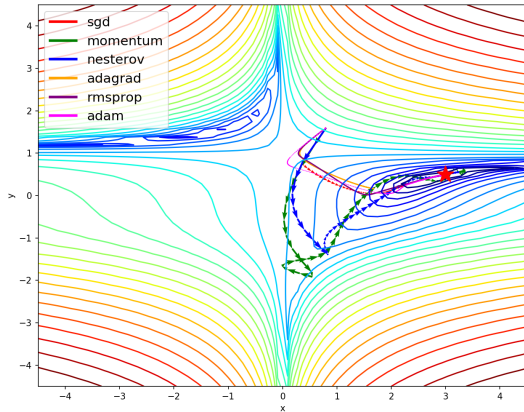


Figure 7: GD comparison from  
<https://github.com/ilguyi/optimizers.numpy>

Figure 8: GD comparison on MNIST from  
 kingma2014adam