Machine Learning (CE 40717)

Fall 2024

Ali Sharifi-Zarchi

CE Department
Sharif University of Technology

October 9, 2024

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

$$\mathscr{L} = \frac{1}{N}\sum_{i=1}^{N} \ell(y_i, \hat{y}_i)$$

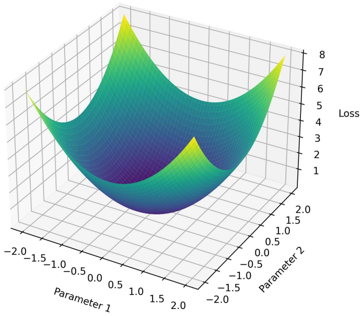
$$\mathbf{W}^* = \arg\min_{\mathbf{W}} \frac{1}{N}\sum_{i=1}^{n} \ell(y_i, \hat{y}_i)$$

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?**

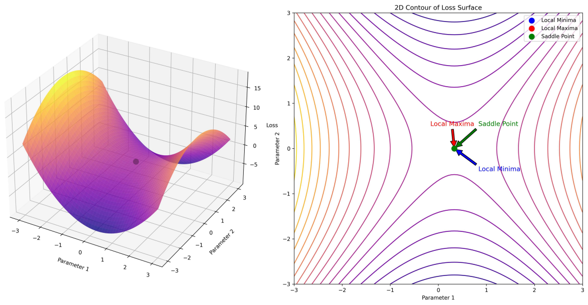
Convexity and Optimization

- **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



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

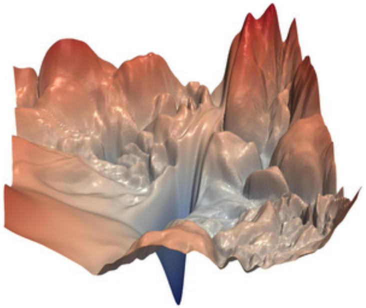


Figure 1: ResNet56 from `https://github.com/tomgoldstein/loss-landscape`

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

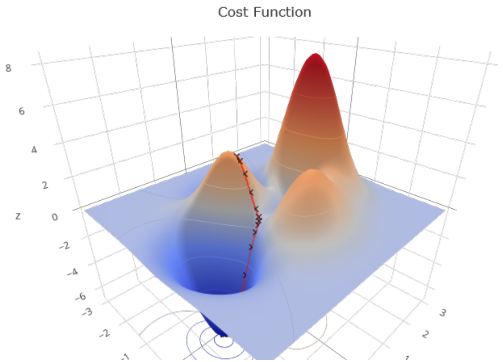


Figure 2: Gradient Descent visualization from `https://blog.skz.dev/gradient-descent`

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)

$$\theta_t = \theta_{t-1} - \eta \nabla_\theta J(\theta)$$

  where:
  - η 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
Batch	Stable convergence Accurate gradient estimate	Computationally expensive Slow for large datasets
Stochastic (SGD)	Fast updates Can escape local minima	Noisy updates May not converge smoothly
Mini-Batch	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

Problem 1 with SGD

- **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

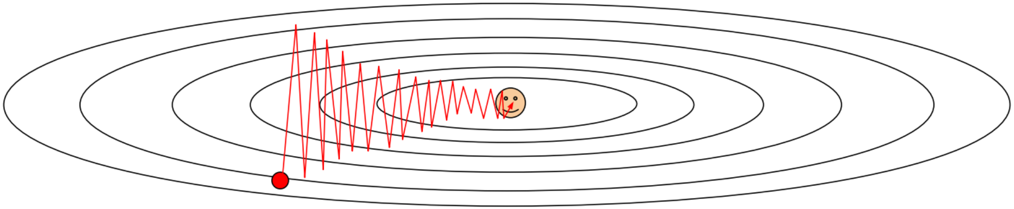


Figure 3: SGD visualization from CS231n: Convolutional Neural Networks for Visual Recognition

Problem 2 with SGD

- 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

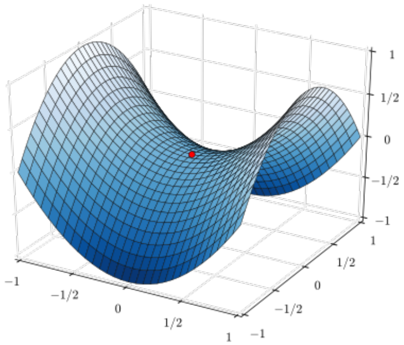


Figure 4: Saddle point from `https://en.wikipedia.org/wiki/Saddle_point`

Problem 3 with SGD

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

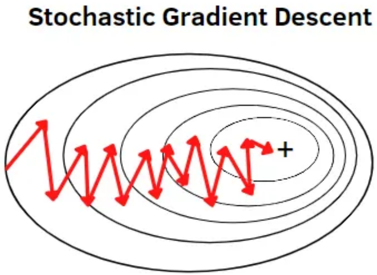


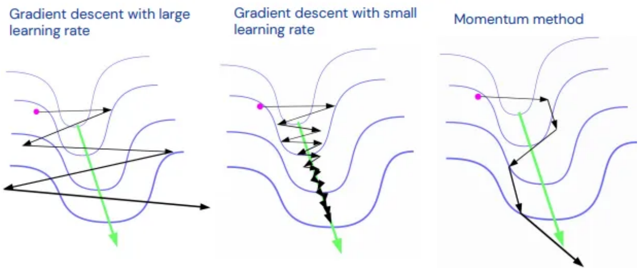
Figure 5: SGD visualization from `https://laptrinhx.com/understanding-optimization-algorithms-3818430905/`

Problem 4 with SGD

- Using the same learning rate for all dimensions can lead to:
  - Smooth convergence in some directions
  - Oscillations or divergence in other directions
- **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**


Gradient descent with large learning rate
Gradient descent with small learning rate
Momentum method

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

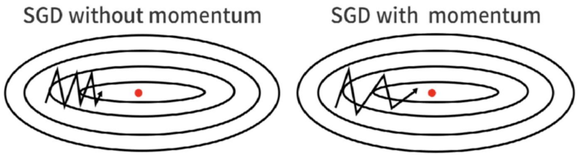


Figure 6: SGD comparison from `https://paperswithcode.com/method/sgd-with-momentum`

First Momentum

- The first momentum, denoted as m_t, is essentially **the moving average of the gradients**
- It's inspired by physics: like a ball rolling in a frictionless bowl, it keeps moving due to accumulated momentum
- In gradient descent with momentum, the update rule is:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla_\theta J(\theta_{t-1})$$

$$\theta_t = \theta_{t-1} - \eta m_t$$

where:

- m_t is the first moment
- β_1 (momentum term) controls how much of the past gradients to include (typically 0.9 or 0.99)
- η is the learning rate
- $\nabla_\theta J(\theta_t)$ is the gradient of the cost function

Second Momentum

- The second momentum, denoted as v_t, is **the moving average of the squared gradients** It tracks the magnitude of the gradients over time
- This is updated similarly to the first momentum:

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\nabla_\theta J(\theta_t) \odot \nabla J(\theta_t))$$

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

where:

- v_t is the second moment
- β_2 is the decay rate
- η is the learning rate
- $\nabla_\theta J(\theta_t)$ is the gradient of the cost function
- $\odot$ denotes the element-wise (Hadamard) product

Moment Bias Correction

- **Problem**: When we start training, both m_t and v_t are initialized to zero. This means that the estimates for the first and second moments will be biased toward zero, especially during the initial steps when the gradients are still small
- **Solution**: To counteract this initial bias, we use the bias-corrected versions of m_t and v_t:

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

Momentum update

- Momentum update steps involve two stages:
  1. Take a step in the opposite direction of the gradient at the current position
  2. Add a scaled version of the previous step to this move
- Reversing these steps leads to a more optimal approach, known as *Nesterov's Accelerated Gradient (NAG)*.

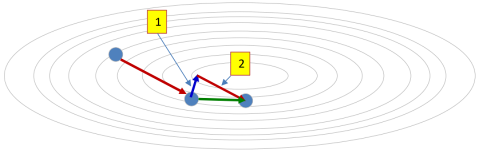


Figure 7: Momentum update from `https://deeplearning.cs.cmu.edu/F24/document/slides/lec6.pdf`

AdaGrad

- The Adaptive Gradient algorithm, or **AdaGrad** for short, adjusts the learning rate for each feature based on the cumulative sum of squared gradients
- Features that are updated frequently receive a progressively smaller learning rate, allowing sparse features that update slowly to catch up

$$G_t = G_{t-1} + \nabla J(\theta_t) \odot \nabla J(\theta_t)$$

$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{G_{t-1}} + \epsilon} \nabla J(\theta_{t-1})$$

Where:

- G_t is the cumulative sum of squared gradients at time step t
- ϵ is a small constant added for numerical stability

RMSProp

- The problem with AdaGrad is that it becomes incredibly slow. This is because the sum of squared gradients only grows and never shrinks
- **RMSProp** (Root Mean Square Propagation) addresses AdaGrad's slow convergence by introducing a **decay factor**, allowing only the most recent squared gradients to contribute to the update, which prevents the accumulation of excessively large values

$$v_t = \beta v_{t-1} + (1-\beta)\nabla J(\theta_t) \odot \nabla J(\theta_t)$$
$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{v_{t-1}} + \epsilon}\nabla J(\theta_{t-1})$$

Where:

- v_t is the **weighted** cumulative sum of squared gradients at time step t and is equal to the second moment
- ϵ is a small constant added for numerical stability

Adam

- Adaptive Moment Estimation, or **Adam**, combines the concepts of momentum and adaptive learning rates 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 Adam optimizer at step t is denoted by θ_t:

$$\theta_t = \theta_{t-1} - \eta \frac{\hat{m}_{t-1}}{\sqrt{\hat{v}_{t-1}} + \epsilon}$$

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla_\theta J(\theta_t)$$

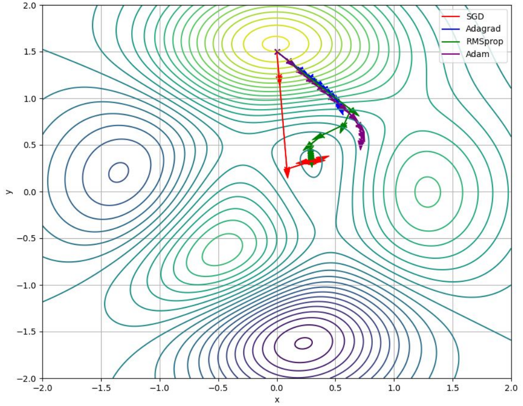
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\nabla_\theta J(\theta_t) \odot \nabla J(\theta_t)$$

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

where:

- m_t and v_t are the first and second moments
- β_1 and β_2 are decay rates for the first and second moments
- $\hat{m}_t$ and $\hat{v}_t$ are bias-corrected estimates of the first and second moments
- ϵ is a small constant to prevent division by zero

Comparison of Momentum Methods



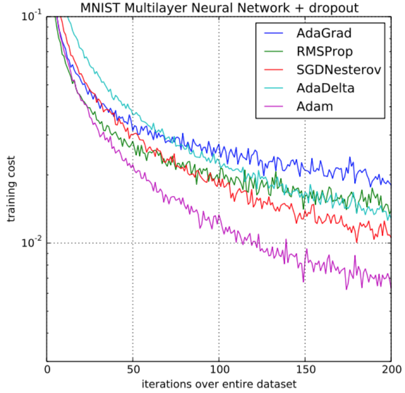


Figure 8: GD comparison on MNIST from `kingma2014adam`