Lecture 5
Gradient Descent

Contents

1 How Backpropagation and Gradient Descent are Connected

Backpropagation and Gradient Descent are two sides of the same learning process in neural networks.

Backpropagation figures out how much each weight contributed to the total error by calculating gradients (partial derivatives) of the loss function with respect to every weight.

Gradient Descent then uses those gradients to adjust the weights so that the error gets smaller after every iteration.

Backpropagation = Calculates gradients.
Gradient Descent = Uses those gradients to update weights.

Together they form the basic learning loop of a neural network:

1. **Forward Pass:** Input data flows through the network to produce an output.
2. **Loss Calculation:** The prediction is compared with the target to measure error.
3. **Backward Pass (Backprop):** Gradients $\left(\frac{\partial L}{\partial w}\right)$ are computed layer by layer using the chain rule.
4. **Weight Update (Gradient Descent):** Apply $w \leftarrow w - \eta \cdot \frac{\partial L}{\partial w}$ to reduce loss.
5. **Repeat:** Iterate until the loss stops improving—this is convergence.

1.1 What is Gradient Descent?

Imagine you're standing on a hill and want to reach the valley's lowest point. The fastest way down is to take small steps downhill, always moving in the direction that decreases your height the most.

That's what gradient descent does—except instead of height, we're minimizing the **loss function**.

It's an optimization algorithm that updates a model's parameters (weights) to find the lowest possible loss.

$$w \leftarrow w - \eta \cdot \frac{\partial L}{\partial w}$$

where:

- w — weight or parameter,
- η — learning rate (step size),
- $\frac{\partial L}{\partial w}$ — gradient of the loss with respect to w.

Intuition: Finding the Line of Best Fit

Imagine you're trying to draw a straight line that best fits a set of points. The goal is to minimize the distance between the line's predicted values ($\hat{y}$) and the actual data points (y).

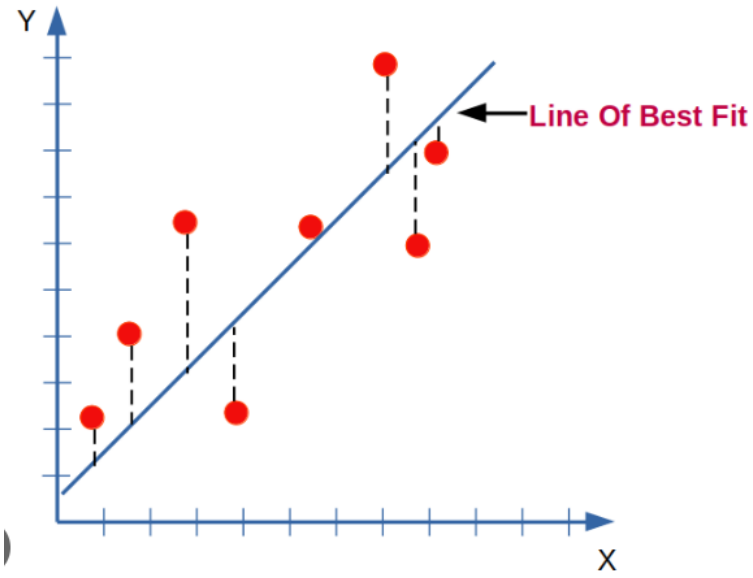

Figure 1: The line of best fit minimizes the vertical distance between actual points (red) and predicted points on the line.

The measure of how "off" our predictions are is captured by the **loss function**. For simple linear regression, we can define the loss as:

$$L = (y - \hat{y})^2$$

This represents the squared difference between actual and predicted outputs. If our line doesn't fit well, L will be large. We want to find the parameters (slope m and bias c) that make L as small as possible.

Geometric View

The loss function, when plotted against parameters (like the slope or bias), often forms a **parabola-shaped curve**. The lowest point of this curve is where the model achieves the **minimum loss** — meaning it fits the data best.

How Gradient Descent Works

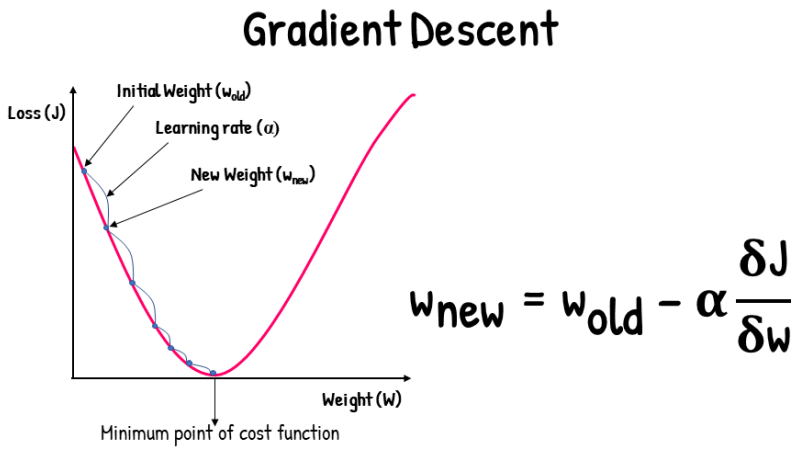

Figure 2: Gradient Descent — moving step by step toward the minimum of the loss function.

We can think of gradient descent as sliding down the loss curve step by step until we reach the bottom (the minimum). Here's how it works:

1. **Pick a random starting point:** Start with a random value for the parameter, say b.
2. **Calculate the slope (gradient):** Compute the derivative of the loss function with respect to b, i.e. $\frac{\partial L}{\partial b}$.
3. **Decide the direction:**
   - If the slope is negative $(-)$, increase b — you need to move right.
   - If the slope is positive $(+)$, decrease b — you need to move left.
4. **Update the parameter:**
$$b_{\text{new}} = b_{\text{old}} - \eta \cdot \frac{\partial L}{\partial b}$$
   Here, η is the **learning rate**, which controls how big a step you take.
5. **Repeat until convergence:** Keep updating b until the loss stops decreasing significantly or a set number of epochs (iterations) is reached.

The Role of Learning Rate

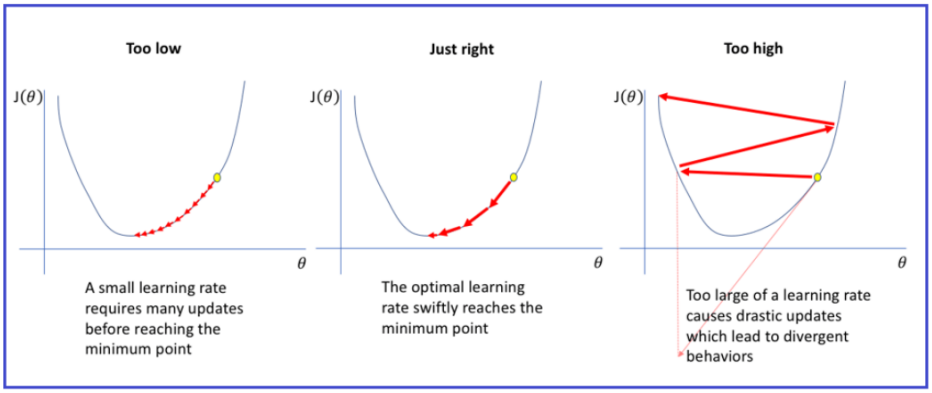

Figure 3: Effect of learning rate on convergence speed and stability. Too high η overshoots the minimum; too low η converges slowly.

The **learning rate** η is a small constant that determines how large each update step is:

- If η is **too high**, you might overshoot the minimum and never converge.
- If η is **too low**, learning becomes very slow and may get stuck before reaching the minimum.

In Summary

Gradient Descent is:

- An **optimization algorithm** used to minimize a loss function.
- It works by **iteratively updating** model parameters (weights and biases).
- The **learning rate** controls the size of each step toward the minimum.
- It continues until the model reaches a state of **convergence**.

Mathematically, for any parameter (weight or bias), the update rule is:

$$w_{\text{new}} = w_{\text{old}} - \eta \frac{\partial J}{\partial w}$$

where:

- J — cost or loss function,
- $\frac{\partial J}{\partial w}$ — gradient (slope) of J with respect to w,
- η — learning rate (controls step size).

1.2 Types of Gradient Descent

Gradient Descent comes in different variants, depending on how much data is used to compute the gradient in each iteration. The choice of variant directly impacts **speed**, **stability**, and **accuracy** of convergence.

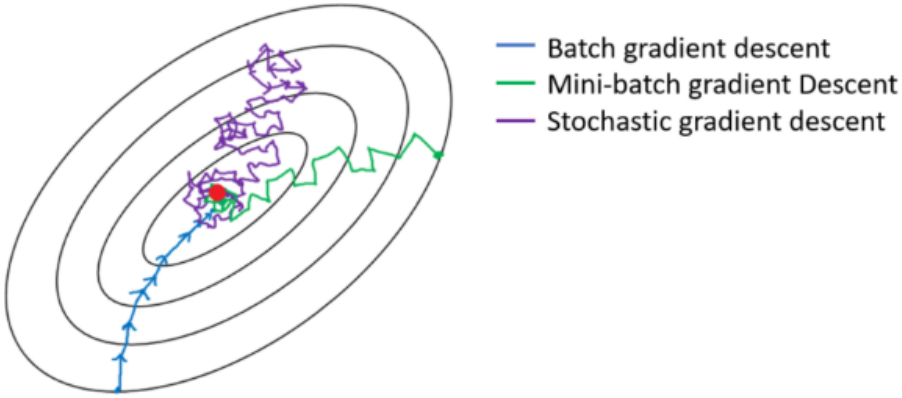

Figure 4: Comparison of different Gradient Descent variants — Batch, Mini-batch, and Stochastic. The contour lines represent the loss surface, and the paths show how each method approaches the minimum.

- **Batch Gradient Descent (BGD):**
  - Uses the *entire training dataset* to compute the gradient for every update.
  - Guarantees a stable and accurate direction toward the global minimum but is computationally expensive.
  - Works well for small datasets where full-batch computation is feasible.
  - Convergence is smooth, but it can be very slow on large-scale data or deep models.

  Update rule:

$$w_{\text{new}} = w_{\text{old}} - \eta \cdot \frac{1}{N} \sum_{i=1}^{N} \frac{\partial L_i}{\partial w}$$

  where N is the total number of samples.

- **Stochastic Gradient Descent (SGD):**
  - Updates parameters after *each individual training sample*.
  - Extremely fast per update and allows real-time or online learning.
  - However, the path toward the minimum is noisy and may fluctuate around the optimal point.
  - The randomness sometimes helps escape local minima, which can be beneficial for deep networks.

  Update rule:

$$w_{\text{new}} = w_{\text{old}} - \eta \cdot \frac{\partial L_i}{\partial w}$$

  where the gradient is computed for one sample i at a time.

- **Mini-batch Gradient Descent:**
  - A **hybrid approach** between Batch and SGD.

- Divides the dataset into small groups called *mini-batches* (e.g., 32, 64, or 128 samples).
- Updates the parameters after computing the gradient over each mini-batch.
- Strikes a balance between computational efficiency and stability — faster than Batch and smoother than SGD.
- It also leverages GPU parallelization effectively, which makes it the **most widely used** variant in deep learning.

Update rule:

$$w_{\text{new}} = w_{\text{old}} - \eta \cdot \frac{1}{m} \sum_{i=1}^{m} \frac{\partial L_i}{\partial w}$$

where m is the mini-batch size.

Key Differences

Feature	**Batch GD**	**Stochastic GD (SGD)**	**Mini-batch GD**
Data used per update	Entire dataset	One sample	Subset (mini-batch)
Convergence	Stable but slow	Fast but noisy	Balanced and efficient
Memory requirement	High	Low	Moderate
Suitability	Small datasets	Large/streaming data	Deep learning (standard choice)
Noise in updates	Minimal	High	Medium
Parallelization	Difficult	Easy	Excellent (GPU-optimized)
Typical use case	Simple regression models	Online learning, reinforcement learning	CNNs, RNNs, large datasets

Summary

- **Batch GD** — Accurate but slow; computes exact gradients on full data.
- **SGD** — Fast and noisy; updates after each sample, good for large or streaming data.
- **Mini-batch GD** — The best of both worlds; efficient, stable, and widely used in deep learning.

1.3 Convex vs Non-Convex Functions

To truly understand how a model converges during training, it's important to grasp the concepts of **convex** and **non-convex** functions. These determine whether gradient descent will smoothly reach an optimal solution or get stuck along the way.

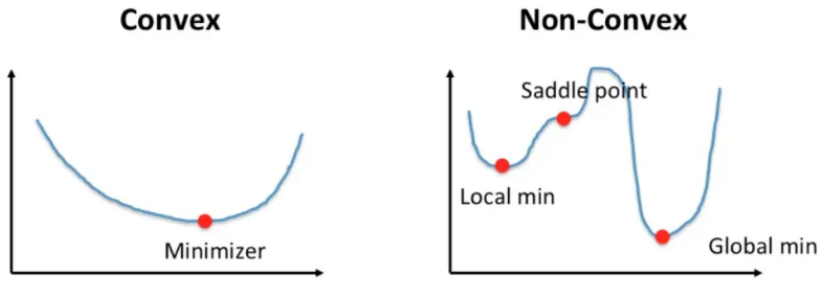

Figure 5: Comparison of Convex and Non-Convex Functions — Convex functions have a single global minimum, while non-convex functions contain multiple minima and saddle points.

Why Convexity Matters for Optimization

Convexity defines how "smooth" or "bumpy" a loss landscape is. In convex functions, optimization is straightforward — the slope always points toward the global minimum. In non-convex functions (like those in deep learning), optimization is messy and can get stuck in suboptimal regions.

Convex functions look like smooth valleys or U-shaped bowls. The gradient always points directly toward the lowest point, making optimization predictable and stable.

Why Convex Functions Are Preferred

- **Global Convergence:** Any local minimum is also the global minimum.
- **Predictable Behavior:** Gradient Descent will converge reliably if learning rate is chosen well.
- **Debug-Friendly:** No sudden jumps or traps — easy to visualize and tune.
- **Explainable Optimization:** The loss surface and descent path are straightforward.

What Is a Non-Convex Function?

A **non-convex function** breaks the convexity rule. It has multiple "valleys" and "peaks," which create:

- **Multiple local minima:** Points where the algorithm might get stuck.
- **Saddle points:** Flat regions where the gradient is zero but not a true minimum.
- **Complex loss surfaces:** No guaranteed smooth path to the best (global) solution.

Why They're Challenging: Deep learning models have non-convex loss functions because of:

- Multiple layers and nonlinear activations,
- Interactions between features and weights,

- High-dimensional parameter spaces.

This means the optimization landscape is full of local minima and saddle points, making convergence difficult. Gradient descent can:

- Settle at a **local minimum** instead of the global one,
- Get stuck at a **saddle point**,
- Oscillate if learning rate is too high.

Real-World Implications: In practice, non-convex optimization means:

- Convergence is not guaranteed.
- Training can take longer or fail entirely.
- Model performance depends heavily on initialization, optimizer, and tuning.

Techniques to Handle Non-Convexity

To improve convergence in non-convex landscapes, several strategies are used:

- **Adam / RMSProp:** Adaptive optimizers that adjust learning rates dynamically based on recent gradients.
- **Momentum:** Helps escape shallow minima and smooths oscillations by adding inertia to updates.
- **Dropout & Batch Normalization:** Regularization techniques that simplify the optimization surface.
- **Multiple Initializations:** Training from different starting points increases the chance of finding the global minimum.
- **Early Stopping:** Prevents overfitting and stops training when loss plateaus.
- **Gradient Clipping:** Limits extreme updates caused by exploding gradients.

In Summary

- **Convex functions** are predictable — gradient descent always finds the global minimum.
- **Non-convex functions** are realistic for deep learning but messy — multiple minima, saddle points, and longer convergence.
- The goal in deep learning is to reach a **good enough local minimum**, even if not the absolute global one.

1.4 Convergence

After understanding convex and non-convex functions, it becomes clear that **convergence** — the point where gradient descent stabilizes — depends heavily on the shape of the loss surface and the choice of hyperparameters.

Gradient Descent works by iteratively updating the model's weights to minimize the loss function. The algorithm is said to have **converged** when one of the following conditions is met:

- The loss function reaches a minimum (global or local).
- The improvement in loss between consecutive iterations becomes negligibly small.

In convex problems, convergence typically means reaching the **global minimum**. However, in complex non-convex neural networks, the loss surface contains multiple valleys — and convergence usually happens at a **good local minimum**, which is often sufficient for high model performance.

Intuition

In deep learning, the goal is not necessarily to find the absolute lowest point on the loss surface, but to find a region where the loss is small and stable — producing consistent, generalizable predictions.

1.5 Challenges in Convergence

Several factors influence whether gradient descent converges smoothly or becomes unstable:

- **Learning Rate (η):**
  - If η is too high, updates overshoot the minimum, causing oscillations or divergence.
  - If η is too low, the updates are too small, resulting in painfully slow convergence.
  - A well-tuned learning rate balances speed and stability.
- **Vanishing and Exploding Gradients:** In deep networks, gradients can either shrink toward zero (*vanish*) or grow uncontrollably large (*explode*) as they propagate through layers. This hinders learning, especially in very deep architectures or RNNs.

  Common Solutions:
  - *Batch Normalization* — stabilizes activations across layers.
  - *Proper Weight Initialization* — e.g., Xavier or He initialization reduces gradient instability.
  - *Gradient Clipping* — caps excessively large gradients to maintain numerical stability.
- **Complex Loss Surfaces:** Non-convex landscapes with multiple local minima and saddle points can mislead the optimizer. Techniques like momentum or adaptive optimizers (Adam, RMSProp) help the algorithm escape such traps.

1.6 Vanishing and Exploding Gradients

One of the most critical challenges in training deep neural networks is the instability of gradients as they are propagated backward through many layers. This problem is known as the **Vanishing or Exploding Gradient Problem**.

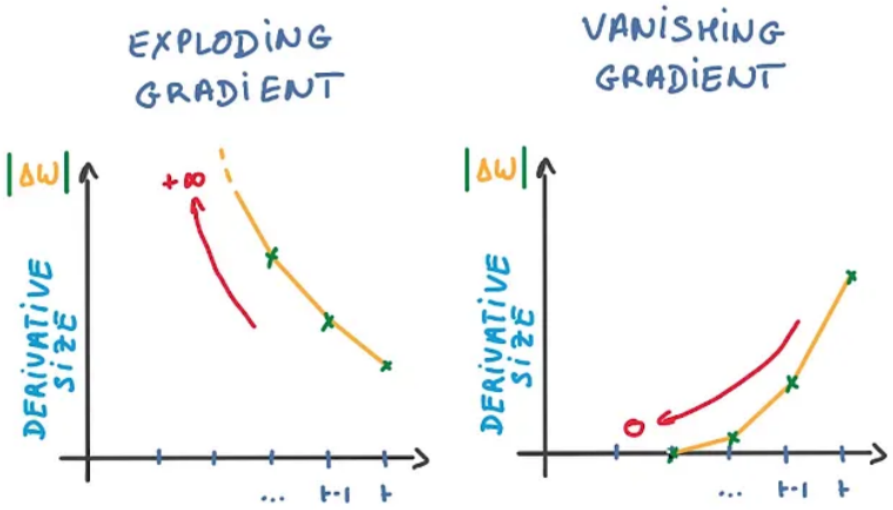

Figure 6: Illustration of Exploding (left) and Vanishing (right) Gradients. Gradients can either grow uncontrollably large or shrink toward zero as they propagate backward through layers.

What Happens During Backpropagation

When training a deep neural network, gradients are propagated backward from the output layer to the input layer. At each layer, the gradient is multiplied by the derivative of the activation function and the layer's weights.

Mathematically, for a weight w_i at layer i:

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial a_i} \cdot \frac{\partial a_i}{\partial z_i} \cdot \frac{\partial z_i}{\partial w_i}$$

If the derivatives in each layer are small (less than 1), repeated multiplication causes the overall gradient to **shrink exponentially** — leading to the *vanishing gradient* problem. Conversely, if the derivatives are large (greater than 1), they can **grow exponentially** — causing the *exploding gradient* problem.

Exploding Gradients (Left Plot)

- Occurs when gradients increase exponentially during backpropagation.
- The weights grow uncontrollably large, leading to numerical instability.
- The model's parameters "explode," resulting in **overflow errors**, extremely large losses, or the network failing to converge.
- Common in deep or recurrent networks when initial weights are too large or learning rate is high.

In the plot: The derivative size (vertical axis) rapidly increases as we move backward through layers (horizontal axis), heading toward $+\infty$.

Typical Symptoms:

- Loss becomes NaN (Not a Number).
- Training diverges after a few epochs.
- Weights oscillate or blow up to huge values.

Solutions:

- Use **gradient clipping** to cap maximum gradient values.
- Initialize weights carefully (He or Xavier initialization).
- Reduce the learning rate.
- Apply normalization layers like BatchNorm.

Vanishing Gradients (Right Plot)

- Occurs when gradients shrink exponentially as they move backward through the network.
- Lower layers receive near-zero updates — effectively "freezing" learning in early layers.
- Common when using activation functions like sigmoid or tanh, which have small derivatives for extreme input values.

In the plot: The derivative size drops to nearly zero as we move backward through layers, causing Δw (weight updates) to become insignificant.

Typical Symptoms:

- Model trains extremely slowly.
- Lower (earlier) layers fail to learn meaningful representations.
- Gradients disappear entirely (flat loss surface).

Solutions:

- Use **ReLU** or variants (Leaky ReLU, ELU) instead of sigmoid/tanh.
- Employ **Batch Normalization** to stabilize activations.
- Use proper initialization (He initialization for ReLU-based networks).
- Consider skip connections (ResNet) to shorten gradient paths.

In Summary

- **Exploding gradients:** Gradients become excessively large, causing unstable updates.
- **Vanishing gradients:** Gradients shrink toward zero, halting learning in early layers.
- Both can be mitigated by proper initialization, normalization, and optimizer choice.

1.7 Summary of the Gradient Update Process

1. **Forward Pass:** Data passes through the network to produce predictions.
2. **Loss Calculation:** The prediction is compared with the target to compute loss.
3. **Backpropagation:** Compute gradients of the loss with respect to each weight.
4. **Weight Update:** Update the weights using an optimization algorithm (gradient descent).
5. **Repeat:** Continue for multiple epochs until convergence or a stopping criterion is met.

2 Enhancements, Variants, and Optimization Techniques

To improve convergence speed, stability, and model generalization, several advanced optimization methods build upon the basic gradient descent framework.

2.0.1 Momentum

Momentum accelerates learning and helps escape local minima by including a fraction of the previous weight update in the current update:

$$\Delta w_m = -\eta \frac{\partial E_{pm}}{\partial w} + \alpha \Delta w_{m-1}$$

where:

- η — learning rate,
- α — momentum coefficient (typically 0.5–0.9).

Momentum smooths oscillations and speeds convergence along consistent descent directions. **Adaptive momentum methods**, such as **Simple Adaptive Momentum (SAM)**, dynamically adjust α based on the direction of weight updates, improving speed and reducing computational cost.

2.0.2 Adaptive Learning Rate Optimizers

Optimizers that adjust learning rates automatically include:

- **AdaGrad:** Scales learning rates inversely with the square root of the sum of past squared gradients. Effective for sparse data but can cause η to become too small over time.
- **RMSProp:** Uses an exponential moving average of squared gradients to maintain a stable learning rate.
- **Adam:** Combines RMSProp and momentum by maintaining moving averages of both gradients and squared gradients, with bias correction for initial steps.

2.0.3 Variants of Gradient Descent

- **Batch Gradient Descent:** Uses the full training dataset each update—accurate but computationally expensive.
- **Stochastic Gradient Descent (SGD):** Updates weights per sample—fast but noisy.
- **Mini-batch Gradient Descent:** Updates after small sample groups, balancing speed and stability. Smaller batches add noise (helping generalization); larger batches risk overfitting.

2.0.4 Regularization Techniques

Regularization improves generalization and stability:

- **Dropout:** Randomly removes neurons during training to prevent co-adaptation.
- **Weight Decay (L1/L2):** Adds penalty terms to constrain weights and limit model complexity.
- **Early Stopping:** Stops training when validation loss increases.
- **Batch Normalization:** Normalizes activations to mitigate internal covariate shift and stabilize training.

2.0.5 Learning Rate Scheduling & Hyperparameter Tuning

- **Learning Rate Scheduling:** Gradually decreases or cycles the learning rate to improve convergence near minima.
- **Hyperparameter Tuning:** Selecting optimal learning rate, momentum, batch size, and regularization parameters is crucial. Validation on hold-out datasets ensures robust performance.

2.0.6 Model Validation During Training

Validation monitors generalization, guides early stopping, and assists hyperparameter tuning. It ensures that the optimization process produces models that perform well beyond the training data.