Optimization Algorithms in Deep Learning

by Dr. Rishikesh Yadav

September 29, 2025

Assistant Professor, School of Mathematical and Statistical Sciences, IIT Mandi, India

Table of Contents

1. Recap: Regression and Classification
2. Role of Gradients and Hessians in Optimization
3. Famous Optimization Algorithms Used in Deep Learning

3.1 Newton-Raphson Algorithm

3.2 Gradient Descent and its Variants

Batch Gradient Descent: The Standard One

Stochastic Gradient Descent (SGD)

Mini-batch Gradient Descent

3.3 Momentum Algorithm: Accelerating Gradient Descent

3.4 Adaptive Learning Rates-Based Algorithms

Adagrad: Adaptive Gradient Algorithm

RMSProp: Root Mean Square Propagation

Adam: Adaptive Moment Estimation

3.5 Theoretical Convergence of Optimizers

4. Practicals: Training Regression and Classification

Recap: Regression and Classification

Supervised Learning: General Framework

- **Goal:** Learn (estimate) a function $f(\mathbf{x}; \mathbf{w})$ that maps several input features $\mathbf{x}$ to an output variable y.

$$y \approx f(\mathbf{x}; \mathbf{w}) \quad \text{(there will be some error terms as well)}$$

- **Data:** A collection of input–output pairs

$$\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{n}$$

  - Inputs (features): $\mathbf{x}_i = (x_{i1}, x_{i2}, \dots, x_{ip})^\top$
  - Outputs (targets): y_i (numeric or categorical)
- **Learning:** Estimate the unknown parameters (weights in machine learning) $\mathbf{w}$ by minimizing a suitable **loss function** $\mathcal{L}(\mathbf{w})$.
- **Prediction:** Once $\hat{\mathbf{w}}$ is obtained, use

$$\hat{y} = f(\mathbf{x}; \hat{\mathbf{w}})$$

  for new input $\mathbf{x}$.

Regression Models

- **Task:** Predict a continuous output $y \in \mathbb{R}$.
- **One simpler model (Assume f is linear in w):** Multiple Linear Regression (MLR)

$$y_i = w_0 + w_1 x_{i1} + w_2 x_{i2} + \cdots + w_p x_{ip} + \varepsilon_i$$

- **Loss function (Mean Squared Error):**

$$\mathcal{L}(\mathbf{w}) = \sum_{i=1}^{n} (y_i - \mathbf{w}^\top \mathbf{x}_i)^2$$

- **Ordinary Least Squares (OLS) solution:**

$$\hat{\mathbf{w}} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$$

where

$$\mathbf{X} = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1p} \\ 1 & x_{21} & x_{22} & \cdots & x_{2p} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & x_{n2} & \cdots & x_{np} \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

- **Important Note**: For all regression types of problems, we might not get the closed-form expressions of $\hat{\mathbf{w}}$.

Classification Models

- **Task:** Predict a categorical output $y \in \{1, 2, \ldots, K\}$.
- **Model:** Logistic regression (binary or multinomial)

$$P(Y = k \mid \mathbf{x}, \mathbf{w}) = \frac{\exp(\mathbf{w}_k^\top \mathbf{x})}{\sum_{j=1}^{K} \exp(\mathbf{w}_j^\top \mathbf{x})}$$

- **Loss function (Cross-Entropy / Log-Loss):**

$$\mathcal{L}(\mathbf{w}) = -\sum_{i=1}^{n} \sum_{k=1}^{K} \mathbf{1}\{y_i = k\} \log P(Y = k \mid \mathbf{x}_i, \mathbf{w})$$

- **Estimation:** No closed-form solution; parameters $\mathbf{w}$ are found using iterative optimization methods (e.g., gradient descent, Adam, Newton–Raphson etc.)
- **Prediction:** Assign the class with the highest predicted probability

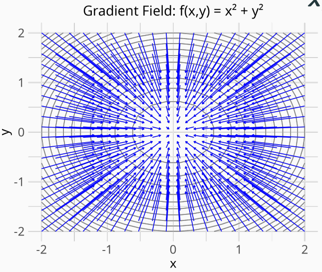
$$\hat{y}_i = \arg \max_{k \in \{1, \ldots, K\}} P(Y = k \mid \mathbf{x}_i, \mathbf{w})$$

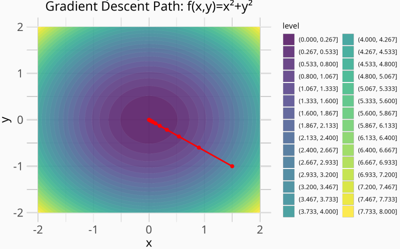
Role of Gradients and Hessians in Optimization

Role of Gradients in Optimization

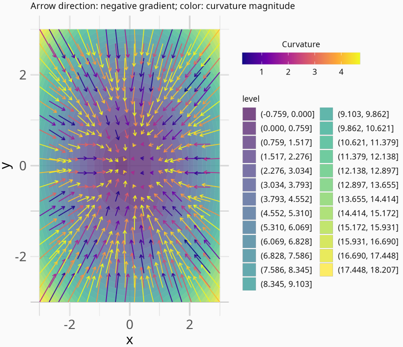
- The **gradient** $\nabla f(x)$ points in the direction of the steepest increase of the function.
- Optimization algorithms use the **negative gradient** to move towards a minimum.
- **Intuition**:
    - If $\nabla f(\boldsymbol{x}) = 0$, we are at a **stationary point**.
    - Gradient magnitude $\|\nabla f(\boldsymbol{x})\|$ indicates how steep the surface is.
- **Example**: Gradient Descent updates

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \eta \nabla f(\boldsymbol{x}_k)$$





- The **Hessian matrix** $H(\mathbf{x}) = \nabla^2 f(\mathbf{x})$ captures the **curvature** of the function.
- **Key roles:**
  - Determines if a stationary point is a minimum, maximum, or saddle.
  - Guides second-order methods (e.g., Newton's method).



  - Positive definite Hessian $\Rightarrow$ local minimum.
  - Negative definite Hessian $\Rightarrow$ local maximum.
  - Indefinite Hessian $\Rightarrow$ saddle point.

- **Example:** Newton-Raphson update

$$\mathbf{x}_{k+1} = \mathbf{x}_k - H(\mathbf{x}_k)^{-1} \nabla f(\mathbf{x}_k)$$

Famous Optimization Algorithms Used in Deep Learning

Agenda

3. Famous Optimization Algorithms Used in Deep Learning

3.1 Newton-Raphson Algorithm

3.2 Gradient Descent and its Variants

Batch Gradient Descent: The Standard One

Stochastic Gradient Descent (SGD)

Mini-batch Gradient Descent

3.3 Momentum Algorithm: Accelerating Gradient Descent

3.4 Adaptive Learning Rates-Based Algorithms

Adagrad: Adaptive Gradient Algorithm

RMSProp: Root Mean Square Propagation

Adam: Adaptive Moment Estimation

3.5 Theoretical Convergence of Optimizers

Newton-Raphson Method

- **What is Newton–Raphson?** An iterative optimization method using both the **gradient** and the **Hessian**:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \left[\nabla_w^2 L(\boldsymbol{w}_t)\right]^{-1} \nabla_w L(\boldsymbol{w}_t)$$

- Uses curvature information (Hessian) for optimal step size and direction
- **Application Example:** classical Logistic regression with small, well-conditioned datasets

Advantages over Gradient Descent:

✓ **Quadratic convergence** near optimum (vs. linear)

✓ **No learning rate** η to tune - step size is adaptive

✓ **Accounts for curvature** - follows natural shape of loss landscape

Geometric Intuition of Hessian in Newton's Method

- **Hessian (H)** captures the **curvature of the loss surface**:
$$H = \nabla^2 \mathcal{L}(\boldsymbol{w})$$
- **Newton update:**
$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - H^{-1} \nabla \mathcal{L}(\boldsymbol{w}_t)$$
- **Geometric intuition:**
    - Directions with **steep curvature** $\rightarrow$ move **smaller steps**
    - Directions with **flat curvature** $\rightarrow$ move **larger steps**
    - Hessian inverse rescales gradient according to curvature.

Geometric Intuition of Hessian in Newton's Method

- **Hessian (H)** captures the **curvature of the loss surface**:

$$H = \nabla^2 \mathcal{L}(\boldsymbol{w})$$

- **Newton update:**

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - H^{-1} \nabla \mathcal{L}(\boldsymbol{w}_t)$$

- **Geometric intuition:**
  - Directions with **steep curvature** $\rightarrow$ move **smaller steps**
  - Directions with **flat curvature** $\rightarrow$ move **larger steps**
  - Hessian inverse rescales gradient according to curvature.

When Hessian is singular or non-invertible:

- Occurs at **saddle points**, flat regions, or redundant parameters.

Geometric Intuition of Hessian in Newton's Method

- **Hessian (H)** captures the **curvature of the loss surface**:

$$H = \nabla^2 \mathcal{L}(\boldsymbol{w})$$

- **Newton update:**

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - H^{-1} \nabla \mathcal{L}(\boldsymbol{w}_t)$$

- **Geometric intuition:**
  - Directions with **steep curvature** $\rightarrow$ move **smaller steps**
  - Directions with **flat curvature** $\rightarrow$ move **larger steps**
  - Hessian inverse rescales gradient according to curvature.

When Hessian is singular or non-invertible:

- Occurs at **saddle points**, flat regions, or redundant parameters.
- Using a **generalized inverse (pseudoinverse)** H^+ is possible:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - H^{+} \nabla \mathcal{L}(\boldsymbol{w}_t)$$

Geometric Intuition of Hessian in Newton's Method

- **Hessian (H)** captures the **curvature of the loss surface**:
$$H = \nabla^2 \mathcal{L}(\boldsymbol{w})$$
- **Newton update:**
$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - H^{-1} \nabla \mathcal{L}(\boldsymbol{w}_t)$$
- **Geometric intuition:**
  - Directions with **steep curvature** $\rightarrow$ move **smaller steps**
  - Directions with **flat curvature** $\rightarrow$ move **larger steps**
  - Hessian inverse rescales gradient according to curvature.

When Hessian is singular or non-invertible:

- Occurs at **saddle points**, flat regions, or redundant parameters.
- Using a **generalized inverse (pseudoinverse)** H^+ is possible:
$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - H^{+} \nabla \mathcal{L}(\boldsymbol{w}_t)$$
- **Intuition:** move along **curved directions only**, ignore flat directions.

Geometric Intuition of Hessian in Newton's Method

- **Hessian (H)** captures the **curvature of the loss surface**:

$$H = \nabla^2 \mathcal{L}(\boldsymbol{w})$$

- **Newton update:**

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - H^{-1} \nabla \mathcal{L}(\boldsymbol{w}_t)$$

- **Geometric intuition:**
  - Directions with **steep curvature** $\rightarrow$ move **smaller steps**
  - Directions with **flat curvature** $\rightarrow$ move **larger steps**
  - Hessian inverse rescales gradient according to curvature.

When Hessian is singular or non-invertible:

- Occurs at **saddle points**, flat regions, or redundant parameters.
- Using a **generalized inverse (pseudoinverse)** H^+ is possible:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - H^+ \nabla \mathcal{L}(\boldsymbol{w}_t)$$

- **Intuition:** move along **curved directions only**, ignore flat directions.
- **Effect:** slower or less precise convergence, but still usable.

Geometric Intuition of Hessian in Newton's Method

- **Hessian (H)** captures the **curvature of the loss surface**:
$$H = \nabla^2 \mathcal{L}(\boldsymbol{w})$$
- **Newton update:**
$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - H^{-1} \nabla \mathcal{L}(\boldsymbol{w}_t)$$
- **Geometric intuition:**
  - Directions with **steep curvature** $\rightarrow$ move **smaller steps**
  - Directions with **flat curvature** $\rightarrow$ move **larger steps**
  - Hessian inverse rescales gradient according to curvature.

When Hessian is singular or non-invertible:

- Occurs at **saddle points**, flat regions, or redundant parameters.
- Using a **generalized inverse (pseudoinverse)** H^+ is possible:
$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - H^{+} \nabla \mathcal{L}(\boldsymbol{w}_t)$$
- **Intuition:** move along **curved directions only**, ignore flat directions.
- **Effect:** slower or less precise convergence, but still usable.
- **Alternative practical fix: Damped Newton**:
$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - (H + \lambda I)^{-1} \nabla \mathcal{L}(\boldsymbol{w}_t)$$
ensures positive definiteness and stable updates.

From Newton-Raphson to Gradient Descent

- **Hessian Computation:** $\nabla^2_w \mathcal{L}(\boldsymbol{w})$ requires $O(p^2)$ memory
  - Modern DL: millions of parameters $\rightarrow$ infeasible storage
- **Matrix Inversion:** $[\nabla^2_w \mathcal{L}(\boldsymbol{w})]^{-1}$ costs $O(p^3)$ computationally
- **Non-Convexity:** Hessian may not be positive definite $\rightarrow$ convergence issues
- **Sensitivity:** Requires careful initialization and well-conditioned problems

Gradient Descent: The Practical Choice

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \eta \nabla_w \mathcal{L}(\boldsymbol{w}_t)$$

- **Memory:** Stores only gradient ($O(p)$) vs. Hessian ($O(p^2)$)
- **Speed:** $O(p)$ per update vs. Newton's $O(p^3)$
- **Robustness:** Works reliably even in non-convex landscapes
- **Scalability:** Mini-batch version enables training on massive datasets

Agenda

3. Famous Optimization Algorithms Used in Deep Learning

3.1 Newton-Raphson Algorithm

3.2 Gradient Descent and its Variants

- Batch Gradient Descent: The Standard One
- Stochastic Gradient Descent (SGD)
- Mini-batch Gradient Descent

3.3 Momentum Algorithm: Accelerating Gradient Descent

3.4 Adaptive Learning Rates-Based Algorithms

- Adagrad: Adaptive Gradient Algorithm
- RMSProp: Root Mean Square Propagation
- Adam: Adaptive Moment Estimation

3.5 Theoretical Convergence of Optimizers

Gradient Descent: The Core Idea

- **Goal:** Minimize a loss function $L(w)$ to find the best model parameters w for regression/classification.
- **Intuition:** Find the lowest point in a valley by always walking downhill.

The Update Rule:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \eta \nabla_w \mathcal{L}(\boldsymbol{w}_t)$$

- $\boldsymbol{w}$: Model parameters (e.g., weights)
- $\mathcal{L}(\boldsymbol{w})$: Loss function (e.g., Mean Squared Error, Log Loss)
- η: Learning rate (step size)
- $\nabla_w \mathcal{L}(\boldsymbol{w})$: Gradient (direction of steepest ascent)

Motivation: Fundamental algorithm for training models like Linear Regression and Logistic Regression.

Batch Gradient Descent: The Standard Version

Algorithm:

1. Initialize parameters $\boldsymbol{w}$ randomly.
2. **Compute Gradient** over the **entire dataset**:

$$\nabla_w \mathcal{L}(\boldsymbol{w}) = \frac{1}{n} \sum_{i=1}^{n} \nabla_w \text{Loss}(f_w(\boldsymbol{x}^{(i)}), y^{(i)})$$

3. **Update:** $\boldsymbol{w} = \boldsymbol{w} - \eta \nabla_w \mathcal{L}(\boldsymbol{w})$
4. Repeat until convergence.

Characteristics:

- ✓ **Pros:** Stable convergence. Guaranteed for convex functions.
- × **Cons:** Very slow for large datasets. One update requires a full data pass.

Stochastic Gradient Descent (SGD)

Core Idea: Use a **single, random** training example $(\boldsymbol{x}^{(i)}, y^{(i)})$ to compute a noisy gradient.

Algorithm per Epoch (one complete pass through data):

1. Shuffle the entire dataset.
2. For each example (input) i in the dataset:
    2.1 Compute gradient for one example: $\nabla_w \mathcal{L}(\boldsymbol{w}; \boldsymbol{x}^{(i)}, y^{(i)})$
    2.2 Update immediately: $\boldsymbol{w} = \boldsymbol{w} - \eta \nabla_w \mathcal{L}(\boldsymbol{w}; \boldsymbol{x}^{(i)}, y^{(i)})$

Key Properties:

- ✓ **Pros:** Extremely fast per update. Can escape local minima due to noise.
- × **Cons:** Very noisy path. Loss may fluctuate heavily. Harder to converge precisely.

Mini-batch Gradient Descent

Core Idea: The best compromise. Use a small random subset (a **mini-batch**) of size b to compute the gradient.

Algorithm per Epoch:

1. Shuffle the dataset.
2. For each batch of b examples:
   2.1 Compute gradient for the batch:
   $\nabla_w \mathcal{L}(\boldsymbol{w}) = \frac{1}{b} \sum_{k=1}^{b} \nabla_w \text{Loss}(f_w(\boldsymbol{x}^{(k)}), y^{(k)})$
   2.2 Update parameters: $\boldsymbol{w} = \boldsymbol{w} - \eta \nabla_w \mathcal{L}(\boldsymbol{w})$

Key Properties:

- ✓ **Pros:** Efficient and leverages GPU parallelism. More stable than SGD.
- × **Cons:** Introduces the batch size b as a new hyperparameter to tune.

Comparison: GD, SGD, and Mini-batch GD

Criterion	Batch GD	Stochastic GD	Mini-batch GD
Gradient	Full dataset	Single example	Small batch (b)
Speed/Update	Slow	Very Fast	Fast
Stability	Smooth	Noisy	Moderate
Memory	High	Low	Medium
Parallelization	Difficult	No	Excellent
Use Case	Small datasets	Large datasets	Deep Learning

Conclusion: For most modern machine learning tasks, especially deep learning, Mini-batch Gradient Descent is the preferred algorithm.

Epochs and Iterations

- **Iteration:** One parameter update using a single batch of data.

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \eta \nabla_w \mathcal{L}(\boldsymbol{w}_t)$$

- **Epoch:** One complete pass over the entire dataset.
- Relationship between them:

$$\text{Iterations per epoch} = \frac{N}{\text{Batch Size}}$$

- Each epoch revisits all training samples once; multiple epochs refine model parameters progressively.

Interpretation:

- **Iterations** = micro updates within an epoch
- **Epoch** = one full cycle through the dataset.

Data Division in Gradient Descent Variants

Method	Batch Size	Description	Key Property
Batch GD	N	Uses all data per update	Stable but slow
Stochastic GD (SGD)	1	One sample per update	High variance, fast convergence
Mini-Batch GD	b (e.g. 32–512)	Uses small random batches per update	Balanced trade-off between noise and stability

Important:

- At the start of each *epoch*, the dataset is **randomly shuffled** and then divided into batches. Each batch is used once per epoch — i.e., **sampling without replacement**. (This is widely used now)
- In **Online SGD**, samples are drawn **randomly with replacement**:
  - Each iteration picks a random data point.
  - Some samples may be seen multiple times before others.
  - There is no strict notion of "epoch".
  - Commonly used in **streaming** or **very large-scale** settings. Barely used

Choosing the Learning Rate η: A Practical Guide

- **The Trade-off:** Newton-Raphson had no η (Hessian adaptive), but GD requires careful tuning

Effects of Learning Rate:

× **Too Large (η big):** Overshooting, divergence, unstable loss

Rule of Thumb: Start with $\eta = 0.01$ or 0.001 and adjust based on loss curve behavior.

Choosing the Learning Rate η: A Practical Guide

- **The Trade-off:** Newton-Raphson had no η (Hessian adaptive), but GD requires careful tuning

Effects of Learning Rate:

× **Too Large (η big):** Overshooting, divergence, unstable loss

× **Too Small (η small):** Extremely slow convergence, stuck in local minima

Rule of Thumb: Start with $\eta = 0.01$ or 0.001 and adjust based on loss curve behavior.

Choosing the Learning Rate η: A Practical Guide

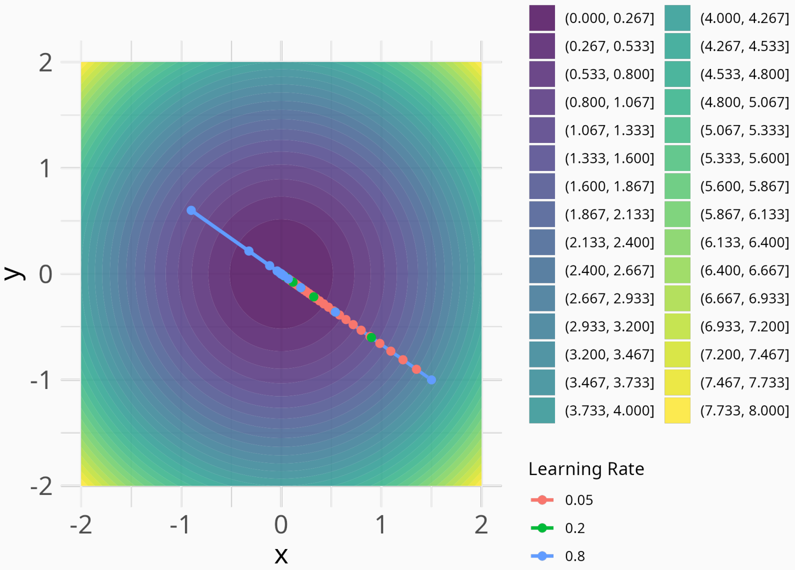
- **The Trade-off:** Newton-Raphson had no η (Hessian adaptive), but GD requires careful tuning

Effects of Learning Rate:

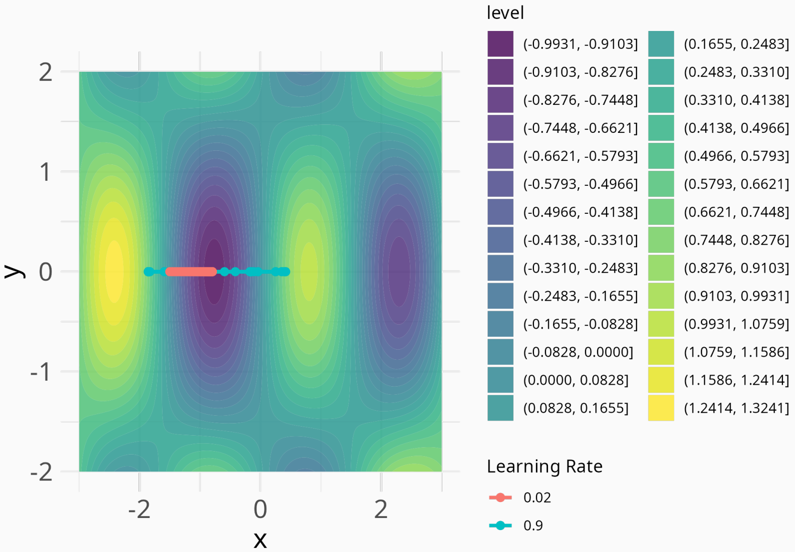
- × **Too Large (η big):** Overshooting, divergence, unstable loss
- × **Too Small (η small):** Extremely slow convergence, stuck in local minima
- ✓ **Just Right:** Stable, efficient convergence to good minimum

Rule of Thumb: Start with $\eta = 0.01$ or 0.001 and adjust based on loss curve behavior.

Gradient Descent with Different Learning Rates: Convex



Gradient Descent with Different Learning Rates: Non-Convex



Small learning rate **stays**; large learning rate **jumps to next basin**.

Practical Strategies:

- **Grid Search:** Try values like 0.1, 0.01, 0.001, 0.0001
- **Learning Rate Schedule:** Start large, decrease over time (e.g., $\eta_t = \frac{\eta_0}{1+t}$)
- **Adaptive Methods:** Use algorithms like the momentum algorithm, Adam, Adagrad that auto-tune η

Agenda

3. Famous Optimization Algorithms Used in Deep Learning

3.1 Newton-Raphson Algorithm

3.2 Gradient Descent and its Variants

Batch Gradient Descent: The Standard One

Stochastic Gradient Descent (SGD)

Mini-batch Gradient Descent

3.3 Momentum Algorithm: Accelerating Gradient Descent

3.4 Adaptive Learning Rates-Based Algorithms

Adagrad: Adaptive Gradient Algorithm

RMSProp: Root Mean Square Propagation

Adam: Adaptive Moment Estimation

3.5 Theoretical Convergence of Optimizers

Momentum: Inertia in Optimization

Motivation:

- Plain (Vanilla) SGD oscillates heavily in ravines (narrow valleys).
- Wastes time moving back-and-forth in directions of steep curvature.
- **Idea:** build inertia — keep moving in the average descent direction.

Update Rule:

$$\boldsymbol{m}_t = \beta \boldsymbol{m}_{t-1} + (1-\beta)\boldsymbol{g}_t$$

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \eta \boldsymbol{m}_t$$

- $\boldsymbol{g}_t$: current gradient
- $\boldsymbol{m}_t$: velocity (exponentially averaged gradients).
- β: momentum coefficient (0.9 typical).

Characteristics:

✓ Faster convergence in ravines.
✓ Reduces oscillations.
× May overshoot if β too high.

Momentum: Inertia in Optimization

Motivation:

- Plain (Vanilla) SGD oscillates heavily in ravines (narrow valleys).
- Wastes time moving back-and-forth in directions of steep curvature.
- **Idea:** build inertia — keep moving in the average descent direction.

Update Rule:

$$\boldsymbol{m}_t = \beta \boldsymbol{m}_{t-1} + (1-\beta)\boldsymbol{g}_t$$

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \eta \boldsymbol{m}_t$$

- $\boldsymbol{g}_t$: current gradient
- $\boldsymbol{m}_t$: velocity (exponentially averaged gradients).
- β: momentum coefficient (0.9 typical).

Characteristics:

✓ Faster convergence in ravines.

✓ Reduces oscillations.

× May overshoot if β too high.

Look at the YouTube lecture of Andrew NG: watch?v=k8fTYJPd3_I

Agenda

3. Famous Optimization Algorithms Used in Deep Learning

3.1 Newton-Raphson Algorithm

3.2 Gradient Descent and its Variants

Batch Gradient Descent: The Standard One

Stochastic Gradient Descent (SGD)

Mini-batch Gradient Descent

3.3 Momentum Algorithm: Accelerating Gradient Descent

3.4 Adaptive Learning Rates-Based Algorithms

Adagrad: Adaptive Gradient Algorithm

RMSProp: Root Mean Square Propagation

Adam: Adaptive Moment Estimation

3.5 Theoretical Convergence of Optimizers

The Need for Adaptive Learning Rates

- **Problem with Fixed η:** Same learning rate for all parameters, all time
- **Real-World Data:** Features have different scales and frequencies
- **Sparse Features:** Some features appear rarely but are important

The Need for Adaptive Learning Rates

- **Problem with Fixed η:** Same learning rate for all parameters, all time
- **Real-World Data:** Features have different scales and frequencies
- **Sparse Features:** Some features appear rarely but are important

Why Adaptation Helps:

- **Sparse Gradients:** Rare features need larger updates when they appear
- **Ill-Conditioned Problems:** Loss landscape has different curvatures in different directions
- **Training Dynamics:** Need larger steps initially, smaller steps near convergence

The Need for Adaptive Learning Rates

- **Problem with Fixed η:** Same learning rate for all parameters, all time
- **Real-World Data:** Features have different scales and frequencies
- **Sparse Features:** Some features appear rarely but are important

Why Adaptation Helps:

- **Sparse Gradients:** Rare features need larger updates when they appear
- **Ill-Conditioned Problems:** Loss landscape has different curvatures in different directions
- **Training Dynamics:** Need larger steps initially, smaller steps near convergence

Goal: Automate the per-parameter learning rate adjustment during training

Adagrad: Adaptive per-Parameter Learning

- **Core Idea:** Scale learning rates by historical gradient magnitudes
- Parameters with large gradients get smaller learning rates, and vice versa

Update Rule:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \frac{\eta}{\sqrt{\boldsymbol{G}_t + \epsilon}} \odot \boldsymbol{g}_t$$

where:

- $\boldsymbol{G}_t$: sum of squares of all past gradients (per parameter)
- ϵ: small constant for numerical stability
- $\boldsymbol{g}_t = \nabla_w \mathcal{L}(\boldsymbol{w}_t)$: gradient vector at time step t
- $\odot$: element-wise multiplication

Characteristics:

✓ Excellent for sparse data and features

× Learning rate decreases too aggressively over time

RMSProp: Fixing Adagrad's Aggressive Decay

- **Improvement:** Use exponentially weighted moving average of gradients
- Prevents learning rate from vanishing too quickly

Update Rule:

$$\boldsymbol{v}_t = \beta \boldsymbol{v}_{t-1} + (1-\beta)\boldsymbol{g}_t^2$$

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \frac{\eta}{\sqrt{\boldsymbol{v}_t + \epsilon}} \odot \boldsymbol{g}_t$$

Key Innovation:

- β: decay rate (typically 0.9) - controls how much history to keep
- Recent gradients matter more than ancient ones
- Learning rates adapt to recent gradient behavior

RMSProp: Fixing Adagrad's Aggressive Decay

- **Improvement:** Use exponentially weighted moving average of gradients
- Prevents learning rate from vanishing too quickly

Update Rule:

$$\boldsymbol{v}_t = \beta \boldsymbol{v}_{t-1} + (1-\beta)\boldsymbol{g}_t^2$$

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \frac{\eta}{\sqrt{\boldsymbol{v}_t + \epsilon}} \odot \boldsymbol{g}_t$$

Key Innovation:

- β: decay rate (typically 0.9) - controls how much history to keep
- Recent gradients matter more than ancient ones
- Learning rates adapt to recent gradient behavior

Result: More stable and practical than Adagrad

Adam: Combining Momentum and Adaptive Rates

- **Best of Both Worlds:** Momentum (like physical inertia) + Adaptive learning rates
- **Most Popular:** Default choice for many deep learning applications

Algorithm [1]:

1. Compute momentum: $\boldsymbol{m}_t = \beta_1 \boldsymbol{m}_{t-1} + (1 - \beta_1)\boldsymbol{g}_t$
2. Compute gradient magnitudes: $\boldsymbol{v}_t = \beta_2 \boldsymbol{v}_{t-1} + (1 - \beta_2)\boldsymbol{g}_t^2$
3. Bias correction: $\hat{\boldsymbol{m}}_t = \frac{\boldsymbol{m}_t}{1-\beta_1^t}$, $\hat{\boldsymbol{v}}_t = \frac{\boldsymbol{v}_t}{1-\beta_2^t}$
4. Update: $\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \frac{\eta}{\sqrt{\hat{\boldsymbol{v}}_t}+\epsilon}\hat{\boldsymbol{m}}_t$

Default Parameters: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$

[1] Kingma, Diederik P., and Jimmy Ba (2014). "Adam": A Method for Stochastic Optimization.

Optimizers with Gradient Descent Variants

Recall: Gradient Descent Variants:

- **Batch GD** — use the full dataset to compute gradient.
- **Mini-batch GD** — compute gradient on small batches.
- **Stochastic GD (SGD)** — compute gradient on a single sample.

Optimizers with Gradient Descent Variants

Recall: Gradient Descent Variants:

- **Batch GD** — use the full dataset to compute gradient.
- **Mini-batch GD** — compute gradient on small batches.
- **Stochastic GD (SGD)** — compute gradient on a single sample.

Optimizers (can be applied to any variant):

- Momentum
- Adagrad
- RMSProp
- Adam

Optimizers with Gradient Descent Variants

Recall: Gradient Descent Variants:

- **Batch GD** — use the full dataset to compute gradient.
- **Mini-batch GD** — compute gradient on small batches.
- **Stochastic GD (SGD)** — compute gradient on a single sample.

Optimizers (can be applied to any variant):

- Momentum
- Adagrad
- RMSProp
- Adam

Key idea:

- Each optimizer only requires a gradient to update parameters.

Optimizers with Gradient Descent Variants

Recall: Gradient Descent Variants:

- **Batch GD** — use the full dataset to compute gradient.
- **Mini-batch GD** — compute gradient on small batches.
- **Stochastic GD (SGD)** — compute gradient on a single sample.

Optimizers (can be applied to any variant):

- Momentum
- Adagrad
- RMSProp
- Adam

Key idea:

- Each optimizer only requires a gradient to update parameters.
- Gradient can come from **full dataset, mini-batch, or single sample**.

Optimizers with Gradient Descent Variants

Recall: Gradient Descent Variants:

- **Batch GD** — use the full dataset to compute gradient.
- **Mini-batch GD** — compute gradient on small batches.
- **Stochastic GD (SGD)** — compute gradient on a single sample.

Optimizers (can be applied to any variant):

- Momentum
- Adagrad
- RMSProp
- Adam

Key idea:

- Each optimizer only requires a gradient to update parameters.
- Gradient can come from **full dataset, mini-batch, or single sample**.
- Therefore, all optimizers are compatible with all gradient descent variants.

Choosing Optimizers: Adam vs SGD + Momentum

Adam: Default Choice for Most Problems

- Handles **sparse gradients** and **non-stationary objectives** well

Choosing Optimizers: Adam vs SGD + Momentum

Adam: Default Choice for Most Problems

- Handles **sparse gradients** and **non-stationary objectives** well
- Adapts learning rate per parameter automatically

Choosing Optimizers: Adam vs SGD + Momentum

Adam: Default Choice for Most Problems

- Handles **sparse gradients** and **non-stationary objectives** well
- Adapts learning rate per parameter automatically
- Fast convergence in deep networks, embeddings, RNNs, Transformers

Choosing Optimizers: Adam vs SGD + Momentum

Adam: Default Choice for Most Problems

- Handles **sparse gradients** and **non-stationary objectives** well
- Adapts learning rate per parameter automatically
- Fast convergence in deep networks, embeddings, RNNs, Transformers

Choosing Optimizers: Adam vs SGD + Momentum

Adam: Default Choice for Most Problems

- Handles **sparse gradients** and **non-stationary objectives** well
- Adapts learning rate per parameter automatically
- Fast convergence in deep networks, embeddings, RNNs, Transformers

Cases Where Simpler Optimizers May Be Better (SGD + Momentum)

- Extremely large, sparse, high-dimensional problems (Adagrad sometimes better)
- Memory constraints — fewer moving averages needed
- Desire for more controlled, predictable convergence

Adaptive Methods: Summary & Comparison

Method	Key Idea	Best For
Adagrad	Scale by sum of all past gradients	Sparse data, NLP
RMSProp	Exponentially weighted average	Non-stationary objectives
Adam	Momentum + adaptive learning rates	General purpose, deep learning
SGD + Momentum	Physical inertia metaphor	Well-tuned convex problems

Adaptive Methods: Summary & Comparison

Method	Key Idea	Best For
Adagrad	Scale by sum of all past gradients	Sparse data, NLP
RMSProp	Exponentially weighted average	Non-stationary objectives
Adam	Momentum + adaptive learning rates	General purpose, deep learning
SGD + Momentum	Physical inertia metaphor	Well-tuned convex problems

Practical Recommendations:

- **Start with Adam:** Good default for most problems

Adaptive Methods: Summary & Comparison

Method	Key Idea	Best For
Adagrad	Scale by sum of all past gradients	Sparse data, NLP
RMSProp	Exponentially weighted average	Non-stationary objectives
Adam	Momentum + adaptive learning rates	General purpose, deep learning
SGD + Momentum	Physical inertia metaphor	Well-tuned convex problems

Practical Recommendations:

- **Start with Adam:** Good default for most problems
- **Fine-tune with SGD:** For ultimate performance (with careful tuning)

Adaptive Methods: Summary & Comparison

Method	Key Idea	Best For
Adagrad	Scale by sum of all past gradients	Sparse data, NLP
RMSProp	Exponentially weighted average	Non-stationary objectives
Adam	Momentum + adaptive learning rates	General purpose, deep learning
SGD + Momentum	Physical inertia metaphor	Well-tuned convex problems

Practical Recommendations:

- **Start with Adam:** Good default for most problems
- **Fine-tune with SGD:** For ultimate performance (with careful tuning)
- **Sparse data:** Consider Adagrad or its variants

Adaptive Methods: Summary & Comparison

Method	Key Idea	Best For
Adagrad	Scale by sum of all past gradients	Sparse data, NLP
RMSProp	Exponentially weighted average	Non-stationary objectives
Adam	Momentum + adaptive learning rates	General purpose, deep learning
SGD + Momentum	Physical inertia metaphor	Well-tuned convex problems

Practical Recommendations:

- **Start with Adam:** Good default for most problems
- **Fine-tune with SGD:** For ultimate performance (with careful tuning)
- **Sparse data:** Consider Adagrad or its variants
- **Use default parameters first**, then experiment if needed

Agenda

3. Famous Optimization Algorithms Used in Deep Learning

3.1 Newton-Raphson Algorithm

3.2 Gradient Descent and its Variants

Batch Gradient Descent: The Standard One

Stochastic Gradient Descent (SGD)

Mini-batch Gradient Descent

3.3 Momentum Algorithm: Accelerating Gradient Descent

3.4 Adaptive Learning Rates-Based Algorithms

Adagrad: Adaptive Gradient Algorithm

RMSProp: Root Mean Square Propagation

Adam: Adaptive Moment Estimation

3.5 Theoretical Convergence of Optimizers

Theoretical Convergence for Convex Problems

Convex Optimization: Convergence Behavior of Optimizers

Optimizer	Convergence Behavior
SGD	Converges to global minimum with proper learning rate decay; slow if learning rate too small.
Momentum	Accelerates convergence in ravines; smooths oscillations.
Adagrad	Convergent due to adaptive learning rate; may become too slow for long training.
RMSProp	Stable convergence; handles noisy gradients well.
Adam	Fast convergence by combining momentum + adaptive learning; widely used in practice.

Theoretical Convergence for Convex Problems

Convex Optimization: Convergence Behavior of Optimizers

Optimizer	**Convergence Behavior**
SGD	Converges to global minimum with proper learning rate decay; slow if learning rate too small.
Momentum	Accelerates convergence in ravines; smooths oscillations.
Adagrad	Convergent due to adaptive learning rate; may become too slow for long training.
RMSProp	Stable convergence; handles noisy gradients well.
Adam	Fast convergence by combining momentum + adaptive learning; widely used in practice.

Key Points:

- All optimizers eventually reach global minimum.
- Convergence rate and stability depend on learning rate and batch size.

Theoretical Convergence for Non-Convex Problems

Non-Convex Optimization: Convergence Behavior of Optimizers

Optimizer	Convergence Behavior
SGD	May oscillate; can get trapped in local minima.
Momentum	Helps escape shallow local minima; can oscillate in complex landscapes.
Adagrad	Learning rate decays too fast; may stop before reaching good minima.
RMSProp	Adapts learning rate; more stable in non-convex landscapes.
Adam	Fast convergence; widely effective, but may overfit or oscillate if not tuned.

Theoretical Convergence for Non-Convex Problems

Non-Convex Optimization: Convergence Behavior of Optimizers

Optimizer	**Convergence Behavior**
SGD	May oscillate; can get trapped in local minima.
Momentum	Helps escape shallow local minima; can oscillate in complex landscapes.
Adagrad	Learning rate decays too fast; may stop before reaching good minima.
RMSProp	Adapts learning rate; more stable in non-convex landscapes.
Adam	Fast convergence; widely effective, but may overfit or oscillate if not tuned.

Key Points:

- Global minimum is not guaranteed.
- Optimizer choice affects ability to escape local minima and convergence speed.

Summary of Algorithm Evolution

- Newton-Raphson: second order, converges fast, but memory heavy

References: Goodfellow et al., 2016; Ruder, 2016; Kingma & Ba, 2014.

Summary of Algorithm Evolution

- Newton-Raphson: second order, converges fast, but memory heavy
- Gradient Descent: first-order, scalable

References: Goodfellow et al., 2016; Ruder, 2016; Kingma & Ba, 2014.

Summary of Algorithm Evolution

- Newton-Raphson: second order, converges fast, but memory heavy
- Gradient Descent: first-order, scalable
- SGD and mini batch GD: mini-batches, stochastic updates

References: Goodfellow et al., 2016; Ruder, 2016; Kingma & Ba, 2014.

Summary of Algorithm Evolution

- Newton-Raphson: second order, converges fast, but memory heavy
- Gradient Descent: first-order, scalable
- SGD and mini batch GD: mini-batches, stochastic updates
- Momentum: smoother and faster convergence

References: Goodfellow et al., 2016; Ruder, 2016; Kingma & Ba, 2014.

Summary of Algorithm Evolution

- Newton-Raphson: second order, converges fast, but memory heavy
- Gradient Descent: first-order, scalable
- SGD and mini batch GD: mini-batches, stochastic updates
- Momentum: smoother and faster convergence
- AdaGrad/RMSProp/Adam: adaptive learning rates for deep networks

References: Goodfellow et al., 2016; Ruder, 2016; Kingma & Ba, 2014.

Summary of Algorithm Evolution

- Newton-Raphson: second order, converges fast, but memory heavy
- Gradient Descent: first-order, scalable
- SGD and mini batch GD: mini-batches, stochastic updates
- Momentum: smoother and faster convergence
- AdaGrad/RMSProp/Adam: adaptive learning rates for deep networks
- **Key Takeaway:** Evolution driven by dataset size, high-dimensional parameters, and non-convex loss landscapes

References: Goodfellow et al., 2016; Ruder, 2016; Kingma & Ba, 2014.

Practicals: Training Regression and Classification

Gradient Descent for Training Binary Classification (Logistic Regression)

- **Loss function** (Binary Cross-Entropy / Log-Loss):

$$\mathcal{L}(\boldsymbol{w}) = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i \log \hat{y}_i + (1-y_i)\log(1-\hat{y}_i)\right],$$

where $\hat{y}_i = \sigma(\boldsymbol{w}^\top \boldsymbol{x}_i)$ and $\sigma(z) = \frac{1}{1+e^{-z}}$.

Gradient Descent for Training Binary Classification (Logistic Regression)

- **Loss function** (Binary Cross-Entropy / Log-Loss):

$$\mathcal{L}(\boldsymbol{w}) = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i \log \hat{y}_i + (1-y_i)\log(1-\hat{y}_i)\right],$$

  where $\hat{y}_i = \sigma(\boldsymbol{w}^\top \boldsymbol{x}_i)$ and $\sigma(z) = \frac{1}{1+e^{-z}}$.

- **Gradient**:

$$\nabla\mathcal{L}(\boldsymbol{w}) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)\boldsymbol{x}_i$$

Gradient Descent for Training Binary Classification (Logistic Regression)

- **Loss function** (Binary Cross-Entropy / Log-Loss):

$$\mathcal{L}(\boldsymbol{w}) = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i \log \hat{y}_i + (1-y_i)\log(1-\hat{y}_i)\right],$$

  where $\hat{y}_i = \sigma(\boldsymbol{w}^\top \boldsymbol{x}_i)$ and $\sigma(z) = \frac{1}{1+e^{-z}}$.

- **Gradient**:

$$\nabla\mathcal{L}(\boldsymbol{w}) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)\boldsymbol{x}_i$$

- Gradient Descent: **Parameter update rule**:

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta \cdot \nabla\mathcal{L}(\boldsymbol{w}^{(t)})$$

  where η is the learning rate.

Lab: Binary Classification Problems

Look at the Jupyter notebook in the https://github.com/yadavrishikesh/Deep-Learning-Slides-Code/blob/main/code/DL_Optim

- For **linearly separable** classification problem: DL_OptimizationAlgorithm_Classification_LinearSep.ipynb
- For **non-linearly separable** classification problem: DL_OptimizationAlgorithm_Classification_NotLinearSep.ipynb

Gradient Descent for Training Linear Regression

- **Loss function** (Mean Squared Error, MSE):

$$\mathcal{L}(\boldsymbol{w}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2, \quad \hat{y}_i = w_0 + w_1 x_i$$

Gradient Descent for Training Linear Regression

- **Loss function** (Mean Squared Error, MSE):

$$\mathcal{L}(\boldsymbol{w}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2, \quad \hat{y}_i = w_0 + w_1 x_i$$

- **Gradients**:

$$\frac{\partial \mathcal{L}}{\partial w_1} = -\frac{2}{n}\sum_{i=1}^{n} x_i(y_i - \hat{y}_i), \qquad \frac{\partial \mathcal{L}}{\partial w_0} = -\frac{2}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)$$

Gradient Descent for Training Linear Regression

- **Loss function** (Mean Squared Error, MSE):

$$\mathcal{L}(\boldsymbol{w}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2, \quad \hat{y}_i = w_0 + w_1 x_i$$

- **Gradients**:

$$\frac{\partial \mathcal{L}}{\partial w_1} = -\frac{2}{n}\sum_{i=1}^{n} x_i(y_i - \hat{y}_i), \qquad \frac{\partial \mathcal{L}}{\partial w_0} = -\frac{2}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)$$

- **Parameter updates**:

$$w_1 \leftarrow w_1 - \eta\frac{\partial \mathcal{L}}{\partial w_1}, \qquad w_0 \leftarrow w_0 - \eta\frac{\partial \mathcal{L}}{\partial w_0}$$

Exercise: Regression Problems

Look at the Jupyter notebook in the https://github.com/yadavrishikesh/Deep-Learning-Slides-Code/blob/main/code/DL_Optim and

- Answer the questions from the Notebook: DL_OptimizationAlgorithm_Regression_Exercise.ipynb