Comparison of Different Types of Gradient Descent

Type	Update Rule	Computational Cost	Convergence Speed	When to Use	Downsides
Batch Gradient Descent (BGD)	$\theta^{(t+1)} = \theta^{(t)} - \alpha \nabla J(\theta)$	High (entire dataset per update)	Slow for large datasets	When the dataset is **small** and you need **precise convergence**	Computationally expensive for **large datasets**
Stochastic Gradient Descent (SGD)	$\theta^{(t+1)} = \theta^{(t)} - \alpha \nabla J(\theta; x^{(i)})$	Low (one sample per update)	Fast, but high variance	When the dataset is **large** and you need faster updates	Noisy updates, doesn't converge smoothly
Mini-Batch Gradient Descent (MBGD)	$\theta^{(t+1)} = \theta^{(t)} - \alpha \nabla J(\theta; B)$	Medium (batch size B)	Balanced speed and stability	When you need a **trade-off between speed and accuracy**	Still has some noise but is computationally better than BGD
Momentum-Based Gradient Descent	$v^{(t)} = \beta v^{(t-1)} + \nabla J(\theta)$ $\theta^{(t+1)} = \theta^{(t)} - \alpha v^{(t)}$	Medium	Faster than standard SGD	When training is **noisy** and you want **smoother updates**	Requires tuning of β
Nesterov Accelerated Gradient (NAG)	$v^{(t)} = \beta v^{(t-1)} + \nabla J(\theta - \beta v^{(t-1)})$ $\theta^{(t+1)} = \theta^{(t)} - \alpha v^{(t)}$	Medium	Faster than momentum	When you need **faster convergence** without overshooting	Requires additional computations

Adagrad	$\theta^{(t+1)} = \theta^{(t)} - \frac{\alpha}{\sqrt{G_{ii}^{(t)} + \epsilon}} \nabla J(\theta)$	Medium	Adaptive step size	When features have **different learning rates**	Learning rate shrinks too fast
RMSprop	$G_{ii}^{(t)} = \beta G_{ii}^{(t-1)} + (1 - \beta)(\nabla J(\theta))^2$ $\theta^{(t+1)} = \theta^{(t)} - \frac{\alpha}{\sqrt{G_{ii}^{(t)} + \epsilon}} \nabla J(\theta)$	Medium	Adaptive learning rate	Good for **non-stationary problems**	Requires tuning β
Adam (Adaptive Moment Estimation)	$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla J(\theta)$ $v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\nabla J(\theta))^2$ $\theta^{(t+1)} = \theta^{(t)} - \frac{\alpha}{\sqrt{v_t} + \epsilon} m_t$	Medium	Fastest and most stable	Default choice for **deep learning**	Slightly more computationally expensive

1. What is an Epoch in Machine Learning?

An **epoch** is **one complete pass** through the **entire training dataset** during model training.

Intuition:

Imagine you're trying to **memorize a book**:

- **One read-through of the entire book = One epoch.**
- If you read it **multiple times**, you're going through **multiple epochs**.
- The more epochs, the better you memorize, but **too many epochs can lead to overfitting**.

Epochs in the Context of Gradient Descent

In **batch gradient descent**:

- Each epoch means the model has **seen every training example once** and updated the weights accordingly.

In **stochastic gradient descent (SGD)**:

- One epoch means **each training example** has been used once, but the model **updates weights after each example**.

In **mini-batch gradient descent**:

- One epoch means **each training sample has been used once**, but the model updates weights **batch-by-batch**.

1. Batch Gradient Descent (BGD)

Intuition

- Computes the **average gradient** over **all data points** before updating parameters.
- Leads to **smooth convergence**, but is **slow** for large datasets.

Mathematical Formulation

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \nabla J(\theta)$$

where:

- $\nabla J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \nabla J(\theta; x_i)$ (gradient over all training samples).
- α = learning rate.

Pseudo Code

plaintext

```
1. Initialize θ randomly.
2. Repeat until convergence:
   a. Compute gradient: ∇J(θ) over entire dataset.
   b. Update parameters: θ = θ - α ∇J(θ).
```

Python Implementation

python

Copy Edit

```python
import numpy as np

def batch_gradient_descent(X, y, theta, alpha, epochs):
    m = len(y)
    for _ in range(epochs):
        gradient = (1/m) * X.T @ (X @ theta - y)
        theta -= alpha * gradient
    return theta
```

2. Stochastic Gradient Descent (SGD)

Intuition

- Updates parameters after **each individual data point**.
- **Faster** but updates are **noisy**.

Mathematical Formulation

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \nabla J(\theta; x^{(i)})$$

where $x^{(i)}$ is **one randomly chosen data point**.

Pseudo Code

plaintext

```
1. Initialize θ randomly.
2. Repeat until convergence:
   a. For each training example (x_i, y_i):
       i. Compute gradient: ∇J(θ; x_i).
       ii. Update parameters: θ = θ - α ∇J(θ; x_i).
```

Python Implementation

python

```python
def stochastic_gradient_descent(X, y, theta, alpha, epochs):
    m = len(y)
    for _ in range(epochs):
        for i in range(m):
            gradient = X[i].T @ (X[i] @ theta - y[i])
            theta -= alpha * gradient
    return theta
```

3. Mini-Batch Gradient Descent (MBGD)

Intuition

- Uses **a small batch** instead of **one sample (SGD)** or **all samples (BGD)**.
- Provides **faster updates** with **less noise**.

Mathematical Formulation

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \nabla J(\theta; B)$$

where B is a **randomly sampled mini-batch**.

Pseudo Code

plaintext Copy Edit

```plaintext
1. Initialize θ randomly.
2. Repeat until convergence:
   a. Randomly sample a mini-batch B.
   b. Compute gradient: ∇J(θ; B).
   c. Update parameters: θ = θ - α ∇J(θ; B).
```

Python Implementation

python

Copy Edit

```python
def mini_batch_gradient_descent(X, y, theta, alpha, batch_size, epochs):
    m = len(y)
    for _ in range(epochs):
        indices = np.random.permutation(m)
        X_shuffled, y_shuffled = X[indices], y[indices]
        for i in range(0, m, batch_size):
            X_batch = X_shuffled[i:i+batch_size]
            y_batch = y_shuffled[i:i+batch_size]
            gradient = (1/batch_size) * X_batch.T @ (X_batch @ theta - y_batch)
            theta -= alpha * gradient
    return theta
```

4. Momentum-Based Gradient Descent

Intuition

- Introduces a **velocity term** that smooths updates.

Mathematical Formulation

$$v^{(t)} = \beta v^{(t-1)} + (1 - \beta)\nabla J(\theta)$$

$$\theta^{(t+1)} = \theta^{(t)} - \alpha v^{(t)}$$

Python Implementation

python Copy Edit

```python
def momentum_gradient_descent(X, y, theta, alpha, beta, epochs):
    m = len(y)
    v = np.zeros_like(theta)
    for _ in range(epochs):
        gradient = (1/m) * X.T @ (X @ theta - y)
        v = beta * v + (1 - beta) * gradient
        theta -= alpha * v
    return theta
```

1. Nesterov Accelerated Gradient (NAG)

Intuition

- Similar to **Momentum-based Gradient Descent**, but **computes the gradient ahead of time.**
- Instead of updating θ using the past velocity, it **computes the gradient at a future step.**
- Helps prevent **overshooting**.

Mathematical Formulation

1. Compute a **look-ahead step** using the velocity:

$$\tilde{\theta} = \theta^{(t)} - \beta v^{(t-1)}$$

2. Compute gradient at **this look-ahead position**:

$$v^{(t)} = \beta v^{(t-1)} + \alpha \nabla J(\tilde{\theta})$$

3. Update parameters:

$$\theta^{(t+1)} = \theta^{(t)} - v^{(t)}$$

Pseudo Code

plaintext

```plaintext
1. Initialize θ randomly, v = 0.
2. Repeat until convergence:
   a. Compute look-ahead step: θ_temp = θ - β * v.
   b. Compute gradient at θ_temp: g = ∇J(θ_temp).
   c. Update velocity: v = β * v + α * g.
   d. Update parameters: θ = θ - v.
```

Python Implementation

python

```python
def nesterov_accelerated_gradient(X, y, theta, alpha, beta, epochs):
    m = len(y)
    v = np.zeros_like(theta)  # Initialize velocity
    for _ in range(epochs):
        theta_lookahead = theta - beta * v  # Compute look-ahead position
        gradient = (1/m) * X.T @ (X @ theta_lookahead - y)  # Compute gradient at l
        v = beta * v + alpha * gradient  # Update velocity
        theta -= v  # Update parameters
    return theta
```

---

2. Adagrad (Adaptive Gradient)

Intuition

- **Adapts the learning rate** for each parameter based on **past gradients**.
- Features with **rare occurrences** get **larger updates**, while **frequent features** get **smaller updates**.
- **Issue**: The learning rate **shrinks too fast**.

Mathematical Formulation

1. Accumulate squared gradients:
$$G_{ii}^{(t)} = G_{ii}^{(t-1)} + (\nabla J(\theta))^2$$
2. Update parameters using an **adaptive step size**:
$$\theta^{(t+1)} = \theta^{(t)} - \frac{\alpha}{\sqrt{G_{ii}^{(t)} + \epsilon}} \nabla J(\theta)$$

Pseudo Code

plaintext Copy Edit

```plaintext
1. Initialize θ randomly, G = 0.
2. Repeat until convergence:
   a. Compute gradient g = ∇J(θ).
   b. Update squared gradient sum: G = G + g^2.
   c. Compute adaptive step size: α / sqrt(G + ε).
   d. Update parameters: θ = θ − (α / sqrt(G + ε)) * g.
```

Python Implementation

python Copy Edit

```python
def adagrad_optimizer(X, y, theta, alpha, epsilon, epochs):
    m = len(y)
    G = np.zeros_like(theta)  # Initialize accumulated squared gradients
    for _ in range(epochs):
        gradient = (1/m) * X.T @ (X @ theta - y)  # Compute gradient
        G += gradient ** 2  # Accumulate squared gradients
        theta -= (alpha / (np.sqrt(G) + epsilon)) * gradient  # Update parameters
    return theta
```

3. RMSprop (Root Mean Square Propagation)

Intuition

- **Fixes Adagrad's problem** of decaying learning rates too fast.
- Uses an **exponentially moving average** of squared gradients instead of summing them.
- Works **well for non-stationary problems** (where gradients change over time).

Mathematical Formulation

1. Compute **exponential moving average** of squared gradients:

$$G_{ii}^{(t)} = \beta G_{ii}^{(t-1)} + (1-\beta)(\nabla J(\theta))^2$$

2. Update parameters:

$$\theta^{(t+1)} = \theta^{(t)} - \frac{\alpha}{\sqrt{G_{ii}^{(t)} + \epsilon}} \nabla J(\theta)$$

Pseudo Code

plaintext Copy Edit

```
1. Initialize θ randomly, G = 0.
2. Repeat until convergence:
   a. Compute gradient g = ∇J(θ).
   b. Update exponentially weighted squared gradient sum:
      G = β * G + (1 − β) * g^2.
   c. Compute adaptive step size: α / sqrt(G + ε).
   d. Update parameters: θ = θ − (α / sqrt(G + ε)) * g.
```

Python Implementation

python

Copy Edit

```python
def rmsprop_optimizer(X, y, theta, alpha, beta, epsilon, epochs):
    m = len(y)
    G = np.zeros_like(theta)  # Initialize exponentially weighted squared gradients
    for _ in range(epochs):
        gradient = (1/m) * X.T @ (X @ theta - y)  # Compute gradient
        G = beta * G + (1 - beta) * (gradient ** 2)  # Update exponentially weighte
        theta -= (alpha / (np.sqrt(G) + epsilon)) * gradient  # Update parameters
    return theta
```

5. Adam (Adaptive Moment Estimation)

Intuition

- Uses **momentum (first moment) and adaptive learning rate (second moment)**.

Mathematical Formulation

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

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

$$\theta^{(t+1)} = \theta^{(t)} - \frac{\alpha}{\sqrt{v_t} + \epsilon} m_t$$

Python Implementation

python Copy Edit

```python
def adam_optimizer(X, y, theta, alpha, beta1, beta2, epsilon, epochs):
    m, v = np.zeros_like(theta), np.zeros_like(theta)
    for _ in range(epochs):
        gradient = X.T @ (X @ theta - y)
        m = beta1 * m + (1 - beta1) * gradient
        v = beta2 * v + (1 - beta2) * (gradient ** 2)
        theta -= alpha * m / (np.sqrt(v) + epsilon)
    return theta
```