

1.Gradient Descent Optimizer

Introduction

Gradient descent is a fundamental optimization algorithm used in machine learning and deep learning to minimize loss functions. It serves as the backbone for training neural networks by iteratively adjusting model parameters (weights and biases) to reduce prediction errors.

What is an Optimizer?

An optimizer is responsible for updating the model parameters during the training process. In neural networks:

- **Role:** Minimize the loss function by adjusting weights and biases
- **Process:** Works during backward propagation
- **Goal:** Find the optimal set of parameters that minimize prediction errors

Neural Network Context

Consider a two-layer neural network:

Input Layer (X_1, X_2, X_3) → Hidden Layer (L_1) → Output Layer (\hat{y})

The optimizer updates all weights in the network to minimize the difference between predicted output (\hat{y}) and actual output (y).

Gradient Descent Algorithm

Gradient descent is an iterative optimization algorithm that moves in the direction of steepest descent of the loss function.

Core Concept

1. **Start** with randomly initialized weights
2. **Calculate** the gradient of the loss function with respect to weights

3. **Update** weights in the opposite direction of the gradient
4. **Repeat** until convergence or maximum iterations reached

Mathematical Foundation

Loss Function

For Mean Squared Error (MSE):

- **Loss Function:**

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

- **Cost Function:**

$$J = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

Where:

- y = actual output
- \hat{y} = predicted output
- n = number of data points

Gradient Calculation

The gradient represents the partial derivative of the cost function with respect to each weight:

$$\nabla J(w) = \frac{\partial J}{\partial w}$$

Weight Update Formula

The fundamental weight update formula for gradient descent:

$$w_{new} = w_{old} - \alpha \cdot \frac{\partial J}{\partial w_{old}}$$

Where:

- w_{new} = updated weight
- w_{old} = current weight

- α = learning rate
- $\frac{\partial J}{\partial w_{old}}$ = gradient of cost function with respect to weight

Learning Rate (α)

The learning rate controls the step size during optimization:

- **Too high:** May overshoot the minimum and cause divergence
- **Too low:** Slow convergence, may get stuck in local minima
- **Optimal:** Balanced convergence speed and stability

Forward and Backward Propagation

Forward Propagation

1. Input data flows through the network
2. Weights multiply inputs, biases are added
3. Activation functions are applied
4. Final output \hat{y} is produced
5. Loss/cost function is calculated

Backward Propagation

1. Gradients are calculated using chain rule
2. Error propagates backward through the network
3. Weights are updated using the gradient descent formula
4. Process repeats for next epoch

Epochs vs Iterations

Epoch

- **Definition:** One complete pass through the entire training dataset
- **Process:** Forward propagation → Loss calculation → Backward propagation
- **In Standard Gradient Descent:** Uses all data points in each epoch

Iteration

- **Definition:** One update of the model parameters
- **In Standard Gradient Descent:** 1 Epoch = 1 Iteration
- **Reason:** All data points are processed together before weight update

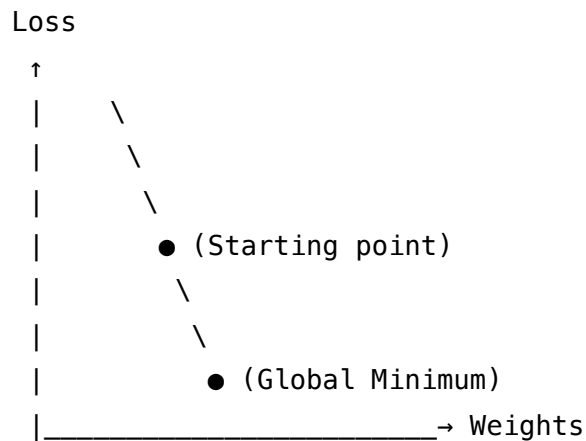
Example with 1000 Data Points

Standard Gradient Descent:

- Epoch 1: Process all 1000 points → Calculate cost → Update weights (1 iteration)
- Epoch 2: Process all 1000 points → Calculate cost → Update weights (1 iteration)
- And so on...

Gradient Descent Curve

The optimization process can be visualized as a parabolic curve:



Key Points:

- **Global Minimum:** The optimal point where loss is minimized
- **Slope at Minimum:** Equals zero, indicating convergence
- **Weight Updates:** Move toward the global minimum

Implementation Details

Algorithm Steps:

1. **Initialize** weights randomly
2. **For each epoch:**
 - Forward propagation with all data points
 - Calculate cost function:

$$J = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

- Backward propagation: compute gradients
- Update weights:

$$w = w - \alpha \cdot \nabla J(w)$$

3. **Repeat** until convergence criteria met

Convergence Criteria:

- Loss function reaches acceptable threshold
- Maximum number of epochs reached
- Gradient magnitude becomes very small

Advantages and Disadvantages

Advantages

1. **Guaranteed Convergence:** For convex functions, converges to global minimum
2. **Stable Updates:** Uses information from entire dataset
3. **Smooth Convergence:** Less noisy compared to other variants
4. **Theoretical Foundation:** Well-established mathematical properties

Disadvantages

1. **Resource Intensive:** Requires large amounts of RAM and GPU memory
2. **Scalability Issues:** Difficult to handle very large datasets (millions of records)
3. **Slow for Large Datasets:** Must process entire dataset before each update

4. **Memory Requirements:** All data points must fit in memory simultaneously
5. **Computational Cost:** High computational overhead for large datasets

Resource Requirements Example:

- **1,000 data points:** Manageable on standard hardware
- **1,000,000 data points:** Requires high-end hardware or cloud computing
- **Billions of data points:** Often impractical with standard gradient descent

Code Examples

Python Implementation (Pseudocode)

```
def gradient_descent(X, y, learning_rate=0.01, epochs=1000):
    # Initialize weights randomly
    weights = np.random.randn(X.shape[1])
    bias = 0

    for epoch in range(epochs):
        # Forward propagation (all data points)
        predictions = np.dot(X, weights) + bias

        # Calculate cost function
        cost = np.mean((y - predictions) ** 2)

        # Calculate gradients
        dw = -(2/len(y)) * np.dot(X.T, (y - predictions))
        db = -(2/len(y)) * np.sum(y - predictions)

        # Update weights (backward propagation)
        weights = weights - learning_rate * dw
        bias = bias - learning_rate * db

        # Print progress
        if epoch % 100 == 0:
            print(f"Epoch {epoch}, Cost: {cost}")

    return weights, bias
```

Mathematical Notation

For each epoch:

$$\hat{y} = X \cdot w + b$$

$$J = \frac{1}{m} \sum_{i=1}^m (y_i - \hat{y}_i)^2$$

$$\frac{\partial J}{\partial w} = -\frac{2}{m} X^T (y - \hat{y})$$

$$\frac{\partial J}{\partial b} = -\frac{2}{m} \sum_{i=1}^m (y_i - \hat{y}_i)$$

$$w = w - \alpha \cdot \frac{\partial J}{\partial w}$$

$$b = b - \alpha \cdot \frac{\partial J}{\partial b}$$

Best Practices

1. Learning Rate Selection

- Start with $\alpha = 0.01$
- Use learning rate scheduling
- Monitor loss curves for optimal adjustment

2. Feature Scaling

- Normalize input features to similar scales
- Use standardization:

$$x_{standardized} = \frac{x - \mu}{\sigma}$$

where μ is the mean and σ is the standard deviation

- Helps achieve faster convergence

3. Weight Initialization

- Use appropriate initialization schemes
- Avoid zero initialization
- Consider Xavier or He initialization

4. Monitoring Convergence

- Plot loss curves
- Check for oscillations or divergence
- Implement early stopping criteria

5. Hardware Considerations

- Ensure sufficient RAM for dataset size
- Use GPU acceleration when available
- Consider cloud computing for large datasets

2. Stochastic Gradient Descent (SGD) Optimizer

Introduction

Stochastic Gradient Descent (SGD) is an optimization algorithm that differs from standard Gradient Descent by processing one data point at a time instead of the entire dataset. This approach fundamentally changes how weight updates occur during model training.

How SGD Works

Basic Process

In SGD, the training process follows this pattern:

For each epoch:

1. **Iteration 1:** Take the first data point

- Compute \hat{y} (prediction)
- Calculate loss function: $L = f(y, \hat{y})$
- Update weights using gradient:

$$w^{(t+1)} = w^{(t)} - \alpha \cdot \frac{\partial L}{\partial w}$$

2. **Iteration 2:** Take the second data point

- Repeat the same process

3. Continue until all data points are processed

Mathematical Formulation

For a dataset with n data points, SGD updates weights as:

Weight Update Formula:

$$w^{(t+1)} = w^{(t)} - \alpha \cdot \frac{\partial L(w^{(t)}, x^{(i)}, y^{(i)})}{\partial w}$$

Loss Function for Single Sample:

$$L^{(i)} = \frac{1}{2} \cdot (y^{(i)} - \hat{y}^{(i)})^2$$

Prediction Formula:

$$\hat{y}^{(i)} = w_0 + w_1 x_1^{(i)} + w_2 x_2^{(i)} + \dots + w_m x_m^{(i)}$$

Gradient Calculation:

$$\frac{\partial L^{(i)}}{\partial w_j} = \frac{\partial L^{(i)}}{\partial \hat{y}^{(i)}} \cdot \frac{\partial \hat{y}^{(i)}}{\partial w_j} = -(y^{(i)} - \hat{y}^{(i)}) \cdot x_j^{(i)}$$

Where:

- $w^{(t)}$ = weights at iteration t
- α = learning rate (typically $\alpha = 0.01, 0.001$, etc.)
- $x^{(i)}$ = i-th input sample
- $y^{(i)}$ = i-th target value
- L = loss function for single sample

Iterations per Epoch:

$$\text{Total Iterations per Epoch} = \frac{n}{1} = n$$

Total Training Iterations:

$$\text{Total Iterations} = E \times n$$

Where E = number of epochs

Epoch Structure

Example with 1000 Data Points

Epoch 1:

- └─ Iteration 1: Process data point 1 → Update weights
- └─ Iteration 2: Process data point 2 → Update weights
- └─ Iteration 3: Process data point 3 → Update weights
- └─ ...
- └─ Iteration 1000: Process data point 1000 → Update weights

Complete one epoch = 1000 iterations

This process continues for multiple epochs (100, 200, etc.) until the loss value stops decreasing significantly.

Advantages of SGD

1. Resource Efficiency

Memory Requirement:

$$\text{Memory Usage} = O(1) \text{ per iteration}$$

- Works with limited RAM (4GB, 8GB systems)
- Processes one record at a time
- No need for massive computational resources
- Suitable for large datasets that don't fit in memory

2. Online Learning Capability

- Can start learning immediately with the first data point
- Suitable for streaming data applications
- Faster initial progress compared to batch methods

Disadvantages of SGD

1. High Time Complexity

Time Analysis:

$$\text{Time per Epoch} = n \times (t_{\text{forward}} + t_{\text{backward}})$$

Time Complexity Formula:

$$\text{Total Training Time} = \frac{E \times n \times (t_{\text{forward}} + t_{\text{backward}})}{\text{Parallel Processing Factor}}$$

Where E = number of epochs, n = number of samples

For large datasets example:

If $n = 1,000,000$ data points and $E = 100$ epochs

$$\text{Total iterations} = E \times n = \frac{100 \times 1,000,000}{1} = 100,000,000 \text{ iterations}$$

2. Slow Convergence

Convergence Rate:

$$\text{Convergence Rate} = O\left(\frac{1}{\sqrt{t}}\right)$$

Expected Error after t iterations:

$$\mathbb{E}[\|w^{(t)} - w^*\|^2] \leq \frac{C}{\sqrt{t}}$$

Where:

- w^* = optimal weights
- C = constant depending on problem
- t = number of iterations

Learning Rate Decay:

$$\alpha^{(t)} = \frac{\alpha_0}{1 + \text{decay_rate} \times t}$$

Or alternatively:

$$\alpha^{(t)} = \frac{\alpha_0}{\sqrt{t}}$$

3. Noisy Convergence Path

The Noise Problem

Unlike standard Gradient Descent which has smooth convergence:

Standard GD Path: Smooth curve \rightarrow Global Minima

SGD Path: Zigzag pattern with noise \rightarrow Eventually Global Minima

Mathematical Explanation

Gradient Variance in SGD:

$$\text{Var}[\nabla L(w, x^{(i)}, y^{(i)})] = \mathbb{E}[(\nabla L^{(i)} - \mathbb{E}[\nabla L^{(i)}])^2]$$

For individual sample:

$$\text{Var}[\nabla L^{(i)}] = \sigma^2$$

For batch of size n :

$$\text{Var}[\nabla L_{\text{batch}}] = \frac{\sigma^2}{n}$$

Signal-to-Noise Ratio:

$$\text{SNR} = \frac{|\mathbb{E}[\nabla L^{(i)}]|^2}{\text{Var}[\nabla L^{(i)}]}$$

Higher SNR = Less noise = Smoother convergence

Noise Reduction Factor:

$$\text{Noise Reduction} = \frac{\sqrt{\text{Batch Size}}}{\sqrt{1}} = \sqrt{\text{Batch Size}}$$

- For SGD (batch size = 1): No noise reduction
- For Mini-batch (batch size = k): \sqrt{k} noise reduction

Noise Characteristics


- Loss function oscillates during training
- Path to global minima is not smooth
- Weight updates are noisy due to single sample gradients
- Creates a "jumping" behavior around the optimal solution

Convergence Pattern Comparison

Gradient Descent

Loss:  Global Minima (Smooth)

Stochastic Gradient Descent

Loss:  Global Minima (Noisy)

When to Use SGD

Recommended Scenarios:

1. **Large Datasets:** When dataset size > available memory
2. **Limited Resources:** Systems with constraints on RAM/processing power
3. **Online Learning:** Real-time data processing requirements
4. **Quick Prototyping:** When you need fast initial results

Not Recommended When:

1. **Small Datasets:** Batch processing is more efficient
2. **High Precision Required:** Noise can be problematic
3. **Time Constraints:** Slow convergence may be limiting

Key Takeaways

1. **Core Concept:** SGD processes one data point per iteration using:

$$w^{(t+1)} = w^{(t)} - \alpha \cdot \frac{\partial L}{\partial w}$$

2. **Resource Efficient:** Solves memory limitations of standard GD
3. **Time Trade-off:** Lower resource usage but higher time complexity
4. **Noisy Convergence:** Path to optimum is not smooth with variance

$$\text{Var}[\nabla L^{(i)}] = \sigma^2$$

5. **Scalability:** Excellent for very large datasets

Formula Summary

Weight Update (Core SGD):

$$w^{(t+1)} = w^{(t)} - \alpha \cdot \frac{\partial L(w^{(t)}, x^{(i)}, y^{(i)})}{\partial w}$$

Loss Functions:

$$L^{(i)} = \frac{1}{2} \cdot (y^{(i)} - \hat{y}^{(i)})^2$$

$$\text{MSE Loss} = \frac{1}{n} \sum_{i=1}^n (y^{(i)} - \hat{y}^{(i)})^2$$

Epoch Completion:

$$1 \text{ Epoch} = \frac{n \text{ iterations}}{1 \text{ sample per iteration}} = n \text{ iterations}$$

Time Complexity:

$$\text{Total Time} = O(E \times n \times (t_{\text{forward}} + t_{\text{backward}}))$$

Where:

- E = number of epochs
- n = dataset size
- t_{forward} = forward pass time per sample

- t_{backward} = backward pass time per sample

Memory Complexity:

$$\text{Memory per Iteration} = O(1)$$

$$\text{Total Memory} = O(\text{model_parameters} + 1_sample)$$

Convergence Properties:

$$\mathbb{E}[\|w^{(t)} - w^*\|^2] \leq \frac{C}{\sqrt{t}}$$

$$\alpha^{(t)} = \frac{\alpha_0}{1 + \lambda t} \quad \text{or} \quad \frac{\alpha_0}{\sqrt{t}}$$

$$\text{Var}[\nabla L^{(i)}] = \sigma^2$$

3. Mini-batch Stochastic Gradient Descent (Mini Batch SGD)

Introduction to Batch Size Concept

In Mini-batch Stochastic Gradient Descent (SGD), we introduce a crucial concept called **batch size** that significantly improves the training process. This approach combines the benefits of both Gradient Descent and Stochastic Gradient Descent while mitigating their respective disadvantages.

Understanding Batch Size with a Practical Example

Let's consider a dataset with **100,000 data points** (100K records). To implement mini-batch SGD effectively:

Key Parameters:

- **Total data points:** 100,000

- **Batch size:** 1,000 records
- **Number of iterations per epoch:** $\frac{\text{Total data points}}{\text{Batch size}} = \frac{100,000}{1,000} = 100$ iterations

Epoch Structure

In each epoch, we complete 100 iterations where:

- Each iteration processes exactly 1,000 data points
- Forward propagation uses all 1,000 data points in the batch
- Backward propagation calculates gradients based on the entire batch

Mathematical Foundation

Cost Function for Mini-batch

For a batch of size B (in our example, $B = 1000$), the cost function using Mean Squared Error is:

$$J(\theta) = \frac{1}{B} \sum_{i=1}^B (y^{(i)} - \hat{y}^{(i)})^2$$

Where:

- $J(\theta)$ is the cost function
- B is the batch size (1000 in our example)
- $y^{(i)}$ is the actual output for the i -th sample
- $\hat{y}^{(i)}$ is the predicted output for the i -th sample

Weight Update Process

The mini-batch SGD optimizer updates weights using:

$$\theta_{t+1} = \theta_t - \alpha \cdot \frac{1}{B} \sum_{i=1}^B \nabla_{\theta} J^{(i)}(\theta_t)$$

Where:

- θ_t represents weights at iteration t
- α is the learning rate
- $\nabla_{\theta} J^{(i)}(\theta_t)$ is the gradient for the i -th sample

Training Process Flow

Complete Epoch Execution:

1. **Iteration 1:** Process data points 1-1000
 - Forward propagation with 1000 samples
 - Calculate cost function
 - Backward propagation and weight update
2. **Iteration 2:** Process data points 1001-2000
 - Repeat the same process
3. **Continue until Iteration 100:** Process data points 99001-100000
 - Complete the epoch

Resource Optimization

Memory Requirements by Batch Size:

Batch Size	Approximate RAM Requirement	Processing Efficiency
1 (Pure SGD)	8 GB	High noise, slow convergence
1,000	8 GB	Balanced performance
5,000	16 GB	Lower noise, faster convergence

The batch size selection depends on available computational resources and desired training characteristics.

Noise Reduction Analysis

Gradient Descent Behavior Comparison:

1. **Standard Gradient Descent:** Smooth convergence path but computationally expensive
2. **Pure SGD:** Fast updates but extremely noisy convergence
3. **Mini-batch SGD:** Balanced approach with reduced noise

Noise Reduction Formula:

The variance in gradient estimates decreases as:

$$\text{Variance} \propto \frac{1}{B}$$

Where B is the batch size. Larger batch sizes lead to more stable gradient estimates.

Advantages of Mini-batch SGD

1. Increased Convergence Speed

- Faster than pure SGD due to more stable gradient estimates
- More efficient than full-batch gradient descent
- Optimal balance between computation and convergence stability

2. Reduced Noise

- Significantly less noisy compared to pure SGD
- Gradient estimates are more reliable
- Smoother convergence path toward global minimum

3. Efficient Resource Usage

- Optimal utilization of available RAM
- Better parallelization opportunities
- Improved GPU/CPU utilization

Mathematical representation of efficiency:

$$\text{Efficiency} = \frac{\text{Convergence Speed}}{\text{Computational Resources}} \times \frac{1}{\text{Noise Level}}$$

Disadvantages and Limitations

1. Residual Noise

Despite improvements over pure SGD, some noise still exists in the convergence path:

$$\text{Remaining Noise} = f\left(\frac{1}{\sqrt{B}}\right)$$

2. Convergence Time

While faster than pure SGD, convergence may still require significant time depending on:

- Problem complexity
- Learning rate selection
- Batch size choice

Convergence Path Visualization

The convergence behavior can be mathematically described as:

Pure SGD Path:

$$\theta_{t+1} = \theta_t - \alpha \nabla_{\theta} J^{(i)}(\theta_t) + \epsilon_{\text{high}}$$

Mini-batch SGD Path:

$$\theta_{t+1} = \theta_t - \alpha \frac{1}{B} \sum_{i=1}^B \nabla_{\theta} J^{(i)}(\theta_t) + \epsilon_{\text{reduced}}$$

Where $\epsilon_{\text{reduced}} < \epsilon_{\text{high}}$ represents the noise reduction achieved.

Future Optimization Strategies

The next step in optimization involves **smoothing techniques** to further reduce the remaining noise:

Momentum-based Approaches:

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

$$\theta_{t+1} = \theta_t - \alpha v_t$$

Where β is the momentum coefficient that helps smooth the optimization path.

Conclusion

Mini-batch SGD represents a significant improvement over pure SGD by:

- Reducing gradient noise through batch averaging
- Improving computational efficiency
- Maintaining reasonable memory requirements
- Providing faster convergence compared to pure SGD

The key insight is that the batch size B creates a trade-off between:

$$\text{Noise Level} \propto \frac{1}{\sqrt{B}} \quad \text{vs} \quad \text{Memory Usage} \propto B$$

This balance makes mini-batch SGD the preferred optimization method for most deep learning applications, setting the foundation for more advanced optimization techniques like momentum, Adam, and other adaptive learning rate methods.

4.SGD with Momentum:

$$v_t = \beta v_{t-1} + (1 - \beta) \nabla J(w_t)$$

$$w_{t+1} = w_t - \alpha v_t$$

5.AdaGrad:

$$G_t = G_{t-1} + (\nabla J(w_t))^2$$

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{G_t + \epsilon}} \nabla J(w_t)$$

Adam:

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

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

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{v_t} + \epsilon} m_t$$