5.Adagrad Optimizer

Introduction

The **Adagrad** (Adaptive Gradient) optimizer is a revolutionary advancement in gradient descent optimization that introduces the concept of **dynamic learning rates**. Unlike traditional gradient descent methods where the learning rate remains fixed throughout training, Adagrad automatically adapts the learning rate based on the historical gradients, making it particularly effective for various machine learning tasks.

Traditional Weight Update Formula

In standard gradient descent, the weight update formula is:

$$w_t = w_{t-1} - \eta \cdot \frac{\partial \mathcal{L}}{\partial w_{t-1}}$$

Where:

- w_t = weights at time step t
- w_{t-1} = weights at previous time step
- η = learning rate (fixed value, typically between 0 and 1)
- $\frac{\partial \mathcal{L}}{\partial w_{t-1}}$ = gradient of loss function with respect to weights

The Problem with Fixed Learning Rates

Traditional optimizers use a **fixed learning rate** (e.g., $\eta = 0.001$) throughout the entire training process. This approach has limitations:

- **Initially**: The algorithm should take bigger steps to converge faster
- **Near convergence**: The algorithm should take smaller steps to avoid overshooting the global minimum

The ideal scenario requires the learning rate to be:

- **High** at the beginning for faster convergence

- **Low** near the global minimum for precision

Adagrad Solution: Dynamic Learning Rate

Adagrad addresses this limitation by making the learning rate **dynamic** rather than fixed. The modified weight update formula becomes:

$$w_t = w_{t-1} - \eta' \cdot \frac{\partial \mathcal{L}}{\partial w_{t-1}}$$

Where η' is the **dynamic learning rate** defined as:

$$\eta' = \frac{\eta}{\sqrt{\alpha_t + \epsilon}}$$

Understanding the Time Step (t)

Before diving into the components, it's crucial to understand what **t** represents in Adagrad:

t = Time step or iteration number during training

- **t = 1**: First training iteration/epoch
- **t = 2**: Second training iteration/epoch
- **t = 3**: Third training iteration/epoch
- And so on throughout the training process...

Important Note: The parameter **t** refers to the temporal sequence of training iterations, **NOT** the layers of a neural network.

Key Components

1. Dynamic Learning Rate (η')

The dynamic learning rate is calculated as:

$$\eta' = \frac{\eta}{\sqrt{\alpha_t + \epsilon}}$$

Where:

- η = initial learning rate (base learning rate)

- α_t = accumulated squared gradients up to time t
- ϵ = small constant to prevent division by zero

2. Accumulated Squared Gradients (α_t) - Multiple Parameters

Important Reality Check: In real neural networks, we have **multiple weights** that need to be updated simultaneously. Adagrad maintains **separate α values for each weight parameter**.

For a network with multiple weights $(w_1, w_2, w_3, ..., w_n)$, Adagrad calculates:

$$\alpha_t^{(w_j)} = \sum_{i=1}^{t} \left(\frac{\partial \mathcal{L}}{\partial w_j}\bigg|_{iteration\ i} \right)^2$$

Where **j** represents each individual weight parameter (w_1, w_2, w_3, etc.).

Step-by-Step Calculation for Each Weight:

At each iteration, for every weight w_j:

$$\alpha_t^{(w_j)} = \alpha_{t-1}^{(w_j)} + \left(\frac{\partial \mathcal{L}}{\partial w_j}\bigg|_{current\ iteration} \right)^2$$

Concrete Example with Multiple Weights:

Consider a simple network with 3 weights (w_1, w_2, w_3):

Iteration 1 (t=1):

- Gradient for w_1 = 0.5 → $\alpha_1^{(w_1)} = 0 + (0.5)^2 = 0.25$
- Gradient for w_2 = 0.3 → $\alpha_1^{(w_2)} = 0 + (0.3)^2 = 0.09$
- Gradient for w_3 = 0.8 → $\alpha_1^{(w_3)} = 0 + (0.8)^2 = 0.64$

Iteration 2 (t=2):

- Gradient for w_1 = 0.2 → $\alpha_2^{(w_1)} = 0.25 + (0.2)^2 = 0.29$
- Gradient for w_2 = 0.4 → $\alpha_2^{(w_2)} = 0.09 + (0.4)^2 = 0.25$
- Gradient for w_3 = 0.1 → $\alpha_2^{(w_3)} = 0.64 + (0.1)^2 = 0.65$

Key Insight: Each weight gets its **own adaptive learning rate** based on its **individual gradient history**!

3. Epsilon (ϵ)

A small positive constant (typically 10^{-8}) added to prevent division by zero when $\alpha_t = 0$.

Complete Adagrad Update Formula - Multiple Parameters

In real neural networks with multiple weights, the complete Adagrad weight update formula for each weight parameter w_j is:

$$w_j^{(t)} = w_j^{(t-1)} - \frac{\eta}{\sqrt{\alpha_t^{(w_j)} + \epsilon}} \cdot \frac{\partial \mathcal{L}}{\partial w_j^{(t-1)}}$$

Where:

- $w_j^{(t)}$ = weight j at iteration t
- $w_j^{(t-1)}$ = weight j at previous iteration
- $\alpha_t^{(w_j)}$ = accumulated squared gradients for weight j up to iteration t

Multiple Weight Updates Simultaneously:

For a network with weights (w_1, w_2, w_3):

$$w_1^{(t)} = w_1^{(t-1)} - \frac{\eta}{\sqrt{\alpha_t^{(w_1)} + \epsilon}} \cdot \frac{\partial \mathcal{L}}{\partial w_1^{(t-1)}}$$

$$w_2^{(t)} = w_2^{(t-1)} - \frac{\eta}{\sqrt{\alpha_t^{(w_2)} + \epsilon}} \cdot \frac{\partial \mathcal{L}}{\partial w_2^{(t-1)}}$$

$$w_3^{(t)} = w_3^{(t-1)} - \frac{\eta}{\sqrt{\alpha_t^{(w_3)} + \epsilon}} \cdot \frac{\partial \mathcal{L}}{\partial w_3^{(t-1)}}$$

Important: Each weight gets its **own adaptive learning rate** because each has its own $\alpha_t^{(w_j)}$ value!

How Adagrad Works

Adaptive Behavior

1. **Early Training** (t is small):
   - α_t is relatively small
   - η' is relatively large

    - Takes bigger steps for faster convergence
2. **Later Training** (t is large):
    - α_t accumulates and becomes large
    - η' becomes smaller
    - Takes smaller, more precise steps

Example of Learning Rate Evolution

Consider an initial learning rate $\eta = 0.01$:

- At $t = 1$: $\eta' = 0.01$ (relatively high)
- At $t = 2$: $\eta' = 0.005$ (decreasing)
- At $t = 3$: $\eta' = 0.003$ (further decrease)

Advantages of Adagrad

Primary Advantage: Dynamic Learning Rate

- **Automatic adaptation**: No manual tuning of learning rate schedules
- **Faster initial convergence**: Larger steps when far from optimum
- **Precise final convergence**: Smaller steps when near optimum
- **Parameter-specific adaptation**: Different learning rates for different parameters

Disadvantages of Adagrad

Major Limitation: Vanishing Learning Rate

As training progresses in very deep neural networks, **for each weight parameter**:

$$\alpha_t^{(w_j)} = \sum_{i=1}^{t} \left(\frac{\partial \mathcal{L}}{\partial w_j}\bigg|_{iteration\ i} \right)^2 \text{ becomes very large}$$

This leads to **individual learning rates** becoming very small:

$$\eta'^{(w_j)} = \frac{\eta}{\sqrt{\alpha_t^{(w_j)} + \epsilon}} \approx 0$$

Consequences of Vanishing Learning Rate

When $\eta'^{(w_j)} \approx 0$ for any weight w_j:

$$w_j^{(t)} = w_j^{(t-1)} - \eta'^{(w_j)} \cdot \frac{\partial \mathcal{L}}{\partial w_j^{(t-1)}} \approx w_j^{(t-1)}$$

This means:

- **Weight updates stop**: $w_j^{(t)} \approx w_j^{(t-1)}$ for affected weights
- **Training stagnates**: No further learning occurs for those parameters
- **Premature convergence**: Algorithm may stop before reaching optimal solution
- **Different weights affected differently**: Some weights may stop updating while others continue (based on their individual gradient histories)

Summary

Adagrad optimizer introduces the groundbreaking concept of **adaptive learning rates** that:

- Start high for faster initial convergence
- Gradually decrease as training progresses
- Automatically adapt without manual intervention

However, the accumulation of squared gradients can cause the learning rate to become too small in deep networks, leading to premature training termination. This limitation paved the way for improved optimizers like **AdaDelta** and **RMSprop**, which address this vanishing learning rate problem.

Key Takeaway

Adagrad = Adaptive Gradient Descent with Dynamic Learning Rate

The fundamental innovation is transforming the fixed learning rate into a dynamic, self-adjusting parameter that improves convergence behavior automatically.

6.Adadelta and RMSprop Optimizers: Dynamic Learning Rate with Exponential

Weighted Averaging

Problem Statement

The main issue with traditional gradient descent is that the learning rate parameter α_t can become very high, leading to unstable training and poor convergence. Adadelta and RMSprop address this problem by introducing dynamic learning rate mechanisms.

Core Innovation: Dynamic Learning Rate

Instead of using a fixed learning rate, both optimizers implement a dynamic learning rate formula:

$$\text{Dynamic Learning Rate} = \frac{\text{learning rate}}{\sqrt{s_{dw} + \epsilon}}$$

Where:

- s_{dw} replaces the problematic α_t term
- ϵ is a small constant to prevent division by zero (typically 10^{-8})

Exponential Weighted Average for Smoothing

The key innovation is computing s_{dw} using **exponential weighted averaging** to smooth the updates:

$$s_{dw}^{(t)} = \beta \cdot s_{dw}^{(t-1)} + (1-\beta) \cdot \left(\frac{\partial \mathcal{L}}{\partial w^{(t-1)}}\right)^2$$

Components Breakdown:

- β: Momentum parameter (typically 0.95)
- $s_{dw}^{(t-1)}$: Previous smoothed gradient magnitude
- $\left(\frac{\partial \mathcal{L}}{\partial w^{(t-1)}}\right)^2$: Squared gradient at previous timestep
- $(1-\beta)$: Weighting factor for current gradient

Initialization

The algorithm begins with:

$$s_{dw}^{(0)} = 0$$

How the Restriction Mechanism Works

When $\beta = 0.95$:

- Previous values get weighted by 0.95
- Current squared gradient gets weighted by $1 - 0.95 = 0.05$

This restriction prevents the gradient magnitude from growing too large, even when the squared gradient $\left(\frac{\partial \mathcal{L}}{\partial w}\right)^2$ becomes very large.

Weight Update Formula

The final weight update combines the dynamic learning rate with the gradient:

$$w^{(t)} = w^{(t-1)} - \frac{\eta}{\sqrt{s_{dw}^{(t)} + \epsilon}} \cdot \frac{\partial \mathcal{L}}{\partial w^{(t-1)}}$$

Where:

- $w^{(t)}$: New weights at timestep t
- $w^{(t-1)}$: Previous weights at timestep $t-1$
- η: Base learning rate
- $\frac{\eta}{\sqrt{s_{dw}^{(t)} + \epsilon}}$: Dynamic learning rate

Key Advantages

1. Dynamic Learning Rate Adaptation

- Automatically adjusts learning rate based on gradient history
- Prevents exploding gradients from causing unstable training
- No manual tuning of learning rate schedules required

2. Exponential Weighted Smoothing

- Smooths out noisy gradient estimates
- Maintains memory of past gradients through the β parameter

- Provides more stable convergence compared to raw gradient methods

3. Parameter Restriction

- Constrains the influence of very large gradients through the $(1-\beta)$ weighting
- Prevents the optimizer from making overly aggressive updates
- Maintains training stability across different problem types

Adadelta vs RMSprop

Both optimizers implement the same core mechanism with slight variations:

- **Same principle**: Dynamic learning rate with exponential weighted averaging
- **Same smoothing**: Both use the exponential weighted average formula
- **Different origins**: Developed by different researchers independently
- **Minor implementation differences**: Slight variations in default parameters and specific formulations

Algorithm Summary

```
Initialize: s_dw^(0) = 0
For each timestep t:
    1. Compute gradient: ∂L/∂w^(t−1)
    2. Update smoothed magnitude: s_dw^(t) = β·s_dw^(t−1) + (1−β)·(∂L/∂w^(t−1))²
    3. Compute dynamic learning rate: η_dynamic = η/√(s_dw^(t) + ε)
    4. Update weights: w^(t) = w^(t−1) − η_dynamic·∂L/∂w^(t−1)
```

This approach effectively solves the problem of α_t becoming too large while providing adaptive, smooth convergence for neural network training.

7.Adam Optimizer

Introduction

The Adam (Adaptive Moment Estimation) optimizer is considered one of the most effective optimization algorithms in machine learning. It combines the best features of two previously discussed optimizers:

1. **SGD with Momentum** - for noise reduction and smoother convergence
2. **RMSprop** - for dynamic learning rates and gradient scaling

Adam essentially merges these approaches to create a robust optimizer that works well across most neural network architectures including ANNs, CNNs, and RNNs.

Core Weight Update Formula

The fundamental weight update equation in Adam optimizer is:

$$w_t = w_{t-1} - \eta' \cdot v_{dw}$$

Where:

- w_t = weights at time step t
- w_{t-1} = weights at previous time step
- η' = dynamic learning rate (explained below)
- v_{dw} = **velocity (momentum) term for weights** - this is the "smart gradient"

Understanding v_{dw} (Velocity/Momentum Term)

v_{dw} represents the **momentum term for weights** and is a key component that makes Adam superior to basic gradient descent:

What the notation means:

- **"v"** stands for "velocity" (from physics analogy of momentum)
- **"dw"** stands for "derivative with respect to weights" ($\frac{\partial L}{\partial w}$)

What v_{dw} actually is:

- An **exponentially weighted moving average** of past gradients
- A "smoothed gradient" that reduces noise and maintains directional momentum
- The "smart gradient" that Adam uses instead of raw gradients

Why use v_{dw} instead of raw gradients?

- **Noise Reduction**: Raw gradients can be noisy; v_{dw} smooths them out
- **Faster Convergence**: Maintains momentum in consistent gradient directions
- **Better Navigation**: Helps escape local minima and navigate saddle points
- **Stability**: Prevents erratic parameter updates

Core Bias Update Formula

Similarly, for bias updates:

$b_t = b_{t-1} - \eta' \cdot v_{db}$

Where:

- b_t = bias at time step t
- b_{t-1} = bias at previous time step
- η' = dynamic learning rate (same as for weights)
- v_{db} = **velocity (momentum) term for bias** - the "smart gradient" for bias updates

Note: v_{db} serves the same purpose as v_{dw} but for bias parameters - it's the smoothed, momentum-enhanced version of bias gradients.

Dynamic Learning Rate Calculation

The dynamic learning rate η' is computed as:

$$\eta' = \frac{\eta}{\sqrt{s_{dw}} + \epsilon}$$

Where:

- η = base learning rate
- s_{dw} = exponentially weighted average of squared gradients (from RMSprop)
- ϵ = small constant (typically 10^{-8}) to prevent division by zero

Exponentially Weighted Average for Squared Gradients (RMSprop Component)

The smoothing of squared gradients is calculated using:

$$s_{dw}^{(t)} = \beta_2 \cdot s_{dw}^{(t-1)} + (1 - \beta_2) \cdot \left(\frac{\partial L}{\partial w^{(t-1)}} \right)^2$$

Component Breakdown:

- $s_{dw}^{(t)}$ = smoothed squared gradients at time t
- β_2 = decay parameter (typically 0.999)
- $s_{dw}^{(t-1)}$ = previous smoothed squared gradients
- $(1 - \beta_2)$ = weight for current gradient contribution
- $\left(\frac{\partial L}{\partial w^{(t-1)}} \right)^2$ = squared gradient of loss with respect to weights

Momentum Calculation for Weights

The momentum term v_{dw} is computed as:

$v_{dw}^{(t)} = \beta_1 \cdot v_{dw}^{(t-1)} + (1 - \beta_1) \cdot \frac{\partial L}{\partial w^{(t-1)}}$

Component Breakdown:

- $v_{dw}^{(t)}$ = **momentum for weights at time t** (the "smart gradient")
- β_1 = momentum decay parameter (typically 0.9) - controls how much past momentum to retain
- $v_{dw}^{(t-1)}$ = previous momentum for weights - represents "memory" of past gradients
- $(1 - \beta_1)$ = weight for current gradient contribution (typically 0.1)
- $\frac{\partial L}{\partial w^{(t-1)}}$ = current gradient of loss with respect to weights

Key Insight: This formula creates a weighted combination where:

- 90% comes from past momentum (memory of previous directions)
- 10% comes from the current gradient (new information)
- This smooths out noise while maintaining overall direction

Momentum Calculation for Bias

Similarly, the momentum term v_{db} for bias is:

$$v_{db}^{(t)} = \beta_1 \cdot v_{db}^{(t-1)} + (1 - \beta_1) \cdot \frac{\partial L}{\partial b^{(t-1)}}$$

Component Breakdown:

- $v_{db}^{(t)}$ = **momentum for bias at time t** (the "smart gradient" for bias)
- $v_{db}^{(t-1)}$ = previous momentum for bias - represents "memory" of past bias gradients
- $\frac{\partial L}{\partial b^{(t-1)}}$ = current gradient of loss with respect to bias

Key Point: The same momentum principle applies to bias updates - creating a smoothed version of bias gradients that reduces noise and maintains directional consistency.

Key Features of Adam Optimizer

1. Momentum Component

- Uses exponentially weighted averages (EWA) of gradients
- Provides smoothening of weight updates
- Reduces noise in the optimization process

2. RMSprop Component

- Implements dynamic learning rates
- Uses exponentially weighted averages of squared gradients
- Prevents learning rates from becoming too large or too small

3. Smoothening Mechanism

- Both momentum and squared gradient terms use exponential smoothing
- Helps maintain stability during optimization
- Prevents abrupt changes in parameter updates

Complete Adam Update Process

The complete Adam optimization process involves these steps:

1. **Calculate momentum terms:**
   - $v_{dw}^{(t)} = \beta_1 \cdot v_{dw}^{(t-1)} + (1 - \beta_1) \cdot \frac{\partial L}{\partial w^{(t-1)}}$
   - $v_{db}^{(t)} = \beta_1 \cdot v_{db}^{(t-1)} + (1 - \beta_1) \cdot \frac{\partial L}{\partial b^{(t-1)}}$
2. **Calculate squared gradient averages:**

- $s_{dw}^{(t)} = \beta_2 \cdot s_{dw}^{(t-1)} + (1 - \beta_2) \cdot \left(\frac{\partial L}{\partial w^{(t-1)}}\right)^2$

3. **Compute dynamic learning rate:**
   - $\eta' = \frac{\eta}{\sqrt{s_{dw}} + \epsilon}$
4. **Update parameters:**
   - $w_t = w_{t-1} - \eta' \cdot v_{dw}$
   - $b_t = b_{t-1} - \eta' \cdot v_{db}$

Why Adam is Recommended

Adam optimizer is widely recommended because it:

- **Combines multiple optimization techniques** into one robust algorithm
- **Converges quickly** due to adaptive learning rates
- **Works well across different architectures** (ANNs, CNNs, RNNs)
- **Handles sparse gradients effectively**
- **Requires minimal hyperparameter tuning**

The combination of momentum smoothing and adaptive learning rates makes Adam particularly effective for most machine learning applications, making it the default choice for many practitioners.