Backpropagation and GD Optimization

Idant Srivastava

December 2025

Introduction

Multi-Layer Perceptrons (MLPs) are fundamental neural network architectures that learn complex patterns through iterative optimization of their parameters. The main component of this learning is backpropagation, an algorithm that efficiently computes gradients of the loss function with respect to parameters. Gradient-based optimization techniques enable MLPs to adjust their weights and biases to minimize errors.

Contents

Chapter 1

Backpropagation

Backpropagation is an algorithm for computing gradients of a loss function with respect to the weights and biases of a neural network. It applies the chain rule of calculus systematically from the output layer back to the input layer.

1.1 Architecture of an MLP

An MLP comprises sequential layers of neurons. For a single training example, it computes:

$$z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$$

$$a^{[l]} = \sigma(z^{[l]})$$

- $W^{[l]}$:Weight matrix
- $b^{[l]}$: Bias matrix
- $\sigma(.)$: Activation function

The activation function is used to make the linear combination of vectors, non-linear so that they can easily find patterns. The functions could be tanh, ReLU, sigmoid, softmax etc.

1.2 Loss Functions

The loss function quantifies the error between a model's predicted output and the true value. This provides a single numerical score that guides the model's learning process by adjusting the weights and biases in order to minimize the loss.

1.2.1 Mean Squared Error (MSE)

This is used for regression:

$$L = \frac{1}{2m} \sum_{i=1}^{m} ||a^{[L][i]} - y^{[i]}||$$

1.2.2 Cross Entropy Loss

This is used for binary classification:

$$L = -\frac{1}{m} \sum_{i=1}^{m} [y^{[i]} \log a^{[L][i]} + (1 - y^{[i]}) \log(1 - a^{[L][i]})]$$

This is used for multi-classification (categorical):

$$L = -\frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} y_k^{(i)} \log a_k^{[L](i)}$$

1.3 Backpropagation

Backpropagation is the core algorithm for training neural networks, using gradient descent to minimize errors by propagating backward through the network, layer by layer, to adjust weights and biases efficiently, allowing the network to learn complex patterns and improve prediction accuracy over time.

1.3.1 Derivatives of the output layer

$$\frac{\partial J}{\partial z^{[L]}} = a^{[l]} - y$$

1.3.2 Derivatives of the hidden layers

At layer l:

$$\delta^{[l]} = \frac{\partial J}{\partial z^{[l]}}$$

1.3.3 Gradient of Parameters

$$\frac{\partial J}{\partial W^{[l]}} = \delta^{[l]} (a^{[l-1]})^T$$

$$\frac{\partial J}{\partial b^{[l]}} = \delta^{[l]}$$

1.3.4 Updating the parameters

$$W^{[l]} = W^{[l]} - \eta \frac{\partial J}{\partial W^{[l]}}$$

$$b^{[l]} = b^{[l]} - \eta \frac{\partial J}{\partial b^{[l]}}$$

Here, η is the learning rate.

1.4 Varients of Gradient Descent

- Batch Gradient Descent (BGD): Computes gradient on the entire dataset, giving stable convergence but slow updates for large data
- Stochastic Gradient Descent (SGD): Updates after each single training example, making it fast but introducing high variance (noise) in updates, potentially avoiding.
- Mini-Batch Gradient Descent (MBGD): Uses small batches (e.g., 32, 64 samples) to balance BGD's stability and SGD's speed, offering efficient updates.

Chapter 2

Activation Functions

Activation functions in deep learning are crucial mathematical functions applied to a neuron's output, introducing non-linearity to allow networks to learn complex patterns. Otherwise, the network would just be a linear model.
Some common activation functions are:

2.1 Sigmoid

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

The sigmoid squashes inputs to the range (0,1).

2.2 Tanh

$$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

Tanh outputs range from (-1,1) and has stronger gradients than sigmoid, but still suffers from vanishing gradients.

2.3 ReLU

$$\text{ReLU}(z) = \max(0, z)$$

ReLU addresses vanishing gradients for positive inputs and is computationally efficient. However, it can suffer from "dying ReLU" when neurons output zero for all inputs.

2.4 Softmax

$$a_k = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}$$

The softmax activation is a function that transforms a vector of real numbers into a probability distribution. It is primarily used in the output layer of neural networks for multi-class classification problems

Chapter 3

Gradient-Based Optimization Algorithms

Gradient-based optimization algorithms like Momentum and RMSProp are advanced variants of the basic gradient descent algorithm, designed to improve convergence speed and stability by adapting the learning rates for each parameter individually.

3.1 Momentum

Concept

Momentum accelerates gradient descent by accumulating an exponentially decaying moving average of past gradients.
It reduces oscillations in directions with high curvature and speeds up convergence along gentle slopes.

Mathematical Foundation

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

- v_t: It is the velocity term
- β: Momentum coefficient
- η: Learning rate

3.2 RMSProp

Concept

Root Mean Square Propagation is an adaptive learning rate optimization algorithm that speeds up training by adjusting learning rates per parameter, preventing them from becoming too small and smoothing oscillations by using an exponentially decaying average of squared gradients to adapt step sizes for faster and stable convergence.

Mathematical Foundation

$$s_t = \beta s_{t-1} + (1-\beta)\nabla J(\theta_t)^2$$
$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{s_t + \epsilon}}\nabla J(\theta_t)$$

3.3 Adam

Concept

The Adam Optimizer (Adaptive Moment Estimation) is an adaptive optimization algorithm that efficiently updates neural network weights by combining Momentum and RMSProp to compute adaptive learning rates for each parameter, enabling faster convergence and better performance.

Mathematical Foundation

Compute biased moment estimates

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla J(\theta_t)$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\nabla J(\theta_t)^2$$

Bias correction

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

Parameter update

$$\theta_{t+1} = \theta_t - \alpha\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Chapter 4

Vanishing and Exploding Gradients

Vanishing and exploding gradients are training problems in deep neural networks where gradients become extremely small (vanish) or large (explode) during backpropagation, hindering learning. Vanishing gradients stop early layers from updating, causing stagnation, while exploding gradients cause unstable, large weight updates, leading to divergence.

4.1 Causes

- Deep networks without proper weight initialization.
- Improper learning rate.

4.2 Solutions

- Use ReLU or Leaky ReLU activations, which have constant derivatives for positive inputs.
- Use batch normalization to maintain stable gradient magnitudes.
- Proper weight initialization schemes like Xavier or He initialization.
- Gradient clipping: cap gradient value at a threshold.

Chapter 5

Learning Rate

The learning rate η controls the step size in parameter space. A learning rate that is too large causes oscillations or divergence, as updates overshoot minima. A learning rate that is too small results in slow convergence and potential trapping in poor local minima.

Learning Rate Schedules

To balance fast initial progress with fine-tuning, learning rate schedules gradually decrease η.

- Step decay: $\eta_t = \eta_0 \gamma^{(\frac{t}{k})}$ where k is the step size,γ is the drop rate, and t is number of epochs.
- Exponential decay: $\eta_t = \eta_0 e^{-\lambda t}$
- Inverse time decay: $\eta_t = \frac{\eta_0}{1+\lambda t}$

Adaptive optimizers like Adam reduce sensitivity to initial learning rate choice, but learning rate scheduling can still improve performance.