Backpropagation and Gradient-Based Optimization in Multilayer Perceptrons

Naman Goel

January 7, 2026

Abstract

This report provides a comprehensive treatment of backpropagation and gradient-based optimization in multilayer perceptrons. We establish the mathematical foundations through the chain rule and derive the backpropagation algorithm. Subsequently, we examine fundamental optimization techniques including gradient descent variants, adaptive learning rate methods such as Adam and RMSprop, loss functions for regression and classification, and activation functions with analysis of their gradient-flow properties.

1 Introduction

Deep neural network training fundamentally relies on computing gradients efficiently and using them to navigate the loss landscape. The ability to minimize a loss function through iterative parameter updates enables learning from data. Two components are essential: (1) a well-defined loss function quantifying prediction error, and (2) an optimization algorithm that systematically reduces this loss. Understanding these mechanisms is foundational to deep learning practice.

2 The Chain Rule and Gradient Computation

2.1 Forward Propagation as Compositional Functions

Consider a multilayer network with L layers. The forward propagation computes a sequence of transformations: the input passes through each layer, undergoing affine transformation followed by nonlinear activation. Mathematically, we can express the forward computation as a composition of functions.

Let us denote the input as $\mathbf{a}^{[0]}$, which corresponds to the original feature vector $\mathbf{x}$. For each layer ℓ from 1 to L, we compute the pre-activation as a weighted sum plus bias:

$$\mathbf{z}^{[\ell]} = \mathbf{W}^{[\ell]}\mathbf{a}^{[\ell-1]} + \mathbf{b}^{[\ell]} \tag{1}$$

Following this, we apply a nonlinear activation function denoted by σ:

$$\mathbf{a}^{[\ell]} = \sigma(\mathbf{z}^{[\ell]}) \tag{2}$$

The final layer output $\mathbf{a}^{[L]}$ represents the network predictions. The loss function measures discrepancy between predictions and targets: the loss $\mathcal{L}$ depends on both the network parameters (weights $\mathbf{W}$ and biases $\mathbf{b}$) and the target labels $\mathbf{y}$.

2.2 The Chain Rule for Multivariate Functions

Fundamental to backpropagation is the chain rule from calculus. For composite functions, the derivative of an output with respect to an earlier input can be computed by multiplying derivatives along the path. Specifically, if a quantity $\mathcal{L}$ depends on an intermediate quantity $\mathbf{z}^{[\ell]}$, which in turn depends on parameters $\mathbf{W}^{[\ell]}$, then:

The gradient of the loss with respect to the weight matrix $\mathbf{W}^{[\ell]}$ is obtained by multiplying the gradient flowing backward from the loss through all subsequent layers, then through the current layer. This composition of partial derivatives along the computational path gives us the required gradient.

2.3 Gradient Propagation Through Layers

To compute how the loss changes with respect to parameters deep in the network, we trace the computational path. The key insight is that we can express the gradient as a product of local gradients at each layer. Instead of computing all these products separately for each parameter, the backpropagation algorithm computes them efficiently by traversing the network in reverse, reusing intermediate computations.

3 Backpropagation Algorithm

3.1 Mathematical Formulation and Error Signals

Backpropagation works by introducing error signals that propagate backward through the network. For the final layer, the error signal is the gradient of the loss with respect to the post-activation values. This is obtained by differentiating the loss function with respect to the output.

For hidden layers, the error signal must be backpropagated through both the activation function and the weight matrix. Specifically, if we have computed the error signal for layer $\ell + 1$, we can compute the error signal for layer ℓ by:

1. Multiplying by the weight matrix transpose (routing error backward through connections)
2. Multiplying element-wise by the derivative of the activation function (accounting for nonlinearity)

Once we have the error signal at each layer, we can compute the gradient with respect to weights by multiplying this error signal by the pre-activation values from the previous layer.

3.2 Algorithm Description

The backpropagation algorithm proceeds in two phases:

Forward Pass: Compute all layer activations $\mathbf{z}^{[\ell]}$ and post-activations $\mathbf{a}^{[\ell]}$ for $\ell = 1, \ldots, L$, storing these values for later use. Compute the final loss $\mathcal{L}$.

Backward Pass: Initialize the error signal at the output layer from the gradient of the loss function. Then, for each layer ℓ from L down to 1:

1. Compute how the loss changes with respect to weights in this layer
2. Compute how the loss changes with respect to pre-activations

3. Compute how the loss changes with respect to the activation from the previous layer (for propagation)

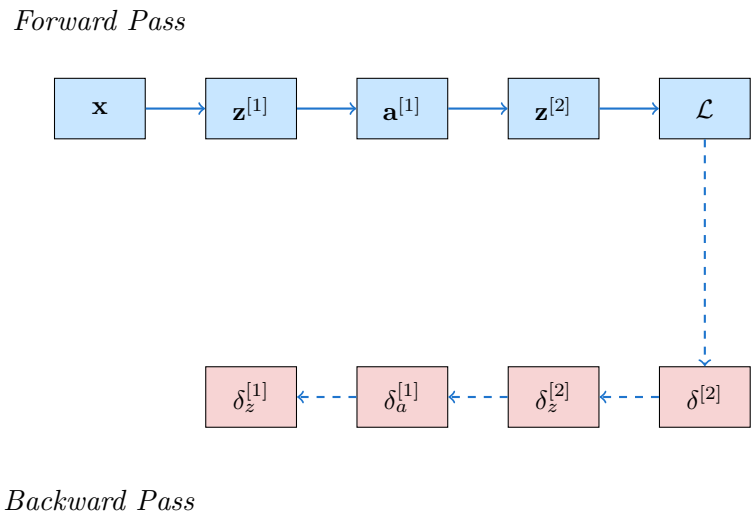

Figure 1: Forward and backward propagation through a two-layer network.

3.3 Computational Efficiency

A naive approach would recompute gradients independently for each parameter, requiring $\mathcal{O}(L^2)$ forward passes through the network. Backpropagation reduces this to $\mathcal{O}(L)$ by traversing the computational graph once backward and reusing intermediate activations stored during the forward pass. This efficiency makes training of deep networks computationally feasible.

Algorithm 1 Backpropagation Algorithm

Input: Batch $\{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^m$, network parameters $\{\mathbf{W}^{[\ell]}, \mathbf{b}^{[\ell]}\}$
Forward Pass: For each layer $\ell = 1$ to L:
    Compute pre-activation: $\mathbf{z}^{[\ell]} = \mathbf{W}^{[\ell]}\mathbf{a}^{[\ell-1]} + \mathbf{b}^{[\ell]}$
    Compute activation: $\mathbf{a}^{[\ell]} = \sigma(\mathbf{z}^{[\ell]})$
Compute loss: $\mathcal{L} = \ell(\mathbf{a}^{[L]}, \mathbf{y})$
Backward Pass: Initialize output error signal
For each layer $\ell = L$ down to 1:
    Compute weight gradient using error signal and previous activation
    Propagate error signal backward through activation function and weights
Update: Apply gradient-based update to all parameters

4 Loss Functions

4.1 Mean Squared Error for Regression

For regression problems where we predict continuous values, the Mean Squared Error loss is standard:

$$\mathcal{L}_{\text{MSE}} = \frac{1}{2n}\sum_{i=1}^{n} \|\hat{\mathbf{y}}_i - \mathbf{y}_i\|^2 \tag{3}$$

where $\hat{\mathbf{y}}_i$ denotes the network prediction and $\mathbf{y}_i$ the target for sample i. The squared error penalizes deviations nonlinearly: large errors receive proportionally larger penalties. The gradient with respect to network output takes a particularly simple form: it is proportional to the prediction error itself.

4.2 Cross-Entropy Loss for Classification

In multi-class classification, we model the problem probabilistically. The network outputs L logits (one per class), which are converted to class probabilities through the softmax function. The softmax function ensures outputs form a valid probability distribution that sums to one.

Cross-entropy loss measures the divergence between the predicted probability distribution and the true distribution (represented as one-hot encoded labels):

$$\mathcal{L}_{\text{CE}} = -\frac{1}{n}\sum_{i=1}^{n} \mathbf{y}_i^T \log(\text{softmax}(\mathbf{z}_i)) \tag{4}$$

A remarkable property of the softmax-cross-entropy combination is that when we differentiate with respect to the pre-softmax logits, we obtain a remarkably simple gradient: the difference between predicted and true probabilities. This simplicity facilitates efficient training.

4.3 Binary Cross-Entropy

For binary classification problems, where we distinguish between two classes, binary cross-entropy provides an appropriate loss:

$$\mathcal{L}_{\text{BCE}} = -\frac{1}{n}\sum_{i=1}^{n} [y_i \log(p_i) + (1 - y_i) \log(1 - p_i)] \tag{5}$$

where p_i is the network's predicted probability of the positive class. This formulation applies equal weight to both classes in expectation and is the standard choice for binary classification.

5 Gradient Descent and Optimization

5.1 Vanilla Gradient Descent

The fundamental principle of gradient-based optimization is to move parameters in the direction opposite to the gradient, which points toward increasing loss. The parameter update rule is:

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \nabla_{\mathbf{W}} \mathcal{L} \tag{6}$$

where η denotes the learning rate, a positive hyperparameter controlling step size. The learning rate plays a critical role: values that are too small result in slow convergence, while values that are too large can cause divergence or oscillation around the minimum.

5.2 Stochastic Gradient Descent

Computing gradients on the entire dataset can be computationally expensive and may not be necessary. Stochastic gradient descent instead computes gradients on small random subsets (mini-batches) of the data:

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \nabla_{\mathbf{W}} \mathcal{L}_{\mathcal{B}} \tag{7}$$

where $\mathcal{B}$ denotes a mini-batch. This approach provides multiple benefits: it reduces computational cost per update, introduces beneficial noise that can help escape local minima, and enables processing of datasets larger than available memory.

5.3 Momentum Methods

To accelerate convergence, momentum methods accumulate a velocity vector that carries information from previous gradient updates. The parameter update becomes:

$$\mathbf{v} \leftarrow \beta \mathbf{v} + \nabla_{\mathbf{W}} \mathcal{L} \tag{8}$$

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \mathbf{v} \tag{9}$$

where β (typically 0.9) controls the momentum coefficient. This approach smooths the trajectory through parameter space, reducing oscillations in directions with alternating gradients and accelerating movement in consistent directions.

5.4 Nesterov Accelerated Gradient

Nesterov momentum provides a refinement that evaluates the gradient at a lookahead position, providing a correction mechanism that reduces overshooting. This variant often achieves slightly faster convergence than standard momentum.

6 Adaptive Learning Rate Methods

6.1 RMSprop: Per-Parameter Learning Rates

Rather than using a fixed learning rate for all parameters, adaptive methods compute individual learning rates based on the historical magnitude of gradients. RMSprop maintains an exponentially weighted average of squared gradients:

The algorithm maintains a running average of squared gradients for each parameter. When updating a parameter, the learning rate is divided by the square root of this accumulated sum (with a small constant added for numerical stability). Parameters receiving consistently large gradients have their effective learning rate reduced, while parameters with small gradients maintain larger effective learning rates.

6.2 Adagrad: Cumulative Gradient History

Adagrad accumulates the sum of squared gradients from the beginning of training without decay. This provides the most aggressive adaptation but suffers from learning rates eventually shrinking to near zero as the accumulated sum grows without bound.

6.3 Adam: Combining Momentum and Adaptive Rates

Adam combines the benefits of momentum methods with adaptive learning rates. It maintains two exponentially weighted moving averages: one for gradients (first moment) and one for squared gradients (second moment). The algorithm applies bias correction to account for initialization effects, particularly important in early training.

The update rule employs both moments:

$$\text{First moment} = \beta_1 \times \text{previous first moment} + (1 - \beta_1) \times \text{current gradient} \tag{10}$$

$$\text{Second moment} = \beta_2 \times \text{previous second moment} + (1 - \beta_2) \times \text{squared current gradient} \tag{11}$$

The parameter update uses the ratio of these moments. With default hyperparameters ($\beta_1 = 0.9$, $\beta_2 = 0.999$), Adam has proven effective across diverse problems with minimal tuning.

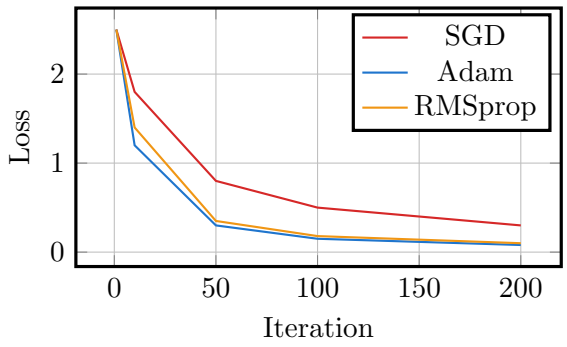

Figure 2: Convergence behavior of different optimizers on a typical deep learning task. Adam typically achieves the fastest early-stage convergence.

7 Learning Rate Scheduling

7.1 Step Decay

A common strategy reduces the learning rate by a constant factor every fixed number of epochs. This approach provides coarse-grained control: early training uses moderate learning rates for exploration, while later training uses smaller learning rates for refinement.

7.2 Exponential Decay

Exponential decay smoothly decreases the learning rate following an exponential curve. This provides a principled approach to gradually reducing learning rates throughout training without discrete jumps.

7.3 Cosine Annealing

Cosine annealing schedules follow a cosine curve, smoothly reducing learning rates from initial to final values. This strategy has theoretical justification and empirically produces good results.

7.4 1 Cycle Learning Rate Policy

The 1 Cycle policy increases learning rate from a minimum to maximum over the first portion of training, then decreases to the minimum over the remainder. This strategy allows exploration early while providing refinement later, and has been shown empirically to improve both convergence speed and generalization.

8 Activation Functions

8.1 Sigmoid Activation

The sigmoid function maps any input to the interval $(0, 1)$, making it historically popular for binary classification:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{12}$$

The function is differentiable with derivative $\sigma'(z) = \sigma(z)(1 - \sigma(z))$. However, the derivative is maximized at $z = 0$ with value 0.25, and rapidly approaches zero for large magnitude inputs.

8.2 Hyperbolic Tangent

The hyperbolic tangent function outputs values in the range $(-1, 1)$ and is zero-centered, offering advantages over sigmoid:

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{13}$$

The derivative reaches maximum value 1 at the origin. Although tanh still exhibits saturation for large magnitude inputs, its zero-centering and stronger gradients make it preferable to sigmoid in practice.

8.3 Rectified Linear Unit

ReLU has become the standard activation in modern deep networks:

$$\text{ReLU}(z) = \max(0, z) \tag{14}$$

The function is computationally efficient and has constant gradient (1) for positive inputs, avoiding saturation in the active region. However, neurons can become permanently inactive if they fall into the region where inputs are negative.

8.4 Leaky ReLU

Leaky ReLU addresses the dying ReLU problem by allowing a small gradient in the negative region:

$$\text{LeakyReLU}(z) = \max(\alpha z, z) \tag{15}$$

where typically $\alpha \in (0, 1)$ (commonly 0.01). This ensures all neurons maintain some gradient signal.

8.5 Softmax for Multi-Class Output

The softmax function converts L logits to a probability distribution:

$$\text{softmax}(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{L} e^{z_k}} \tag{16}$$

The outputs sum to one and are strictly positive, making them valid probabilities. When combined with cross-entropy loss, the gradient has a particularly clean form.

9 Gradient Flow: Vanishing and Exploding Gradients

9.1 The Vanishing Gradient Problem

In deep networks, gradients propagate backward through many layers via multiplication of local derivatives. Consider a chain of layers where each contributes a factor to the overall gradient. If each factor is less than one in magnitude, multiplying many such factors produces a very small result. This is the vanishing gradient problem.

Sigmoid and tanh activations exhibit this property: their derivatives are bounded by small constants (0.25 for sigmoid, 1 for tanh at best). In a deep network with 50 layers, the gradient product may be reduced by factors of 10^{-30} or smaller, rendering gradient signals to early layers negligible.

9.2 Consequences

When gradients vanish, parameters in deep layers receive nearly zero gradient signals. These parameters effectively cease learning meaningful updates. The network essentially becomes shallow only the final layers learn significantly while early layers stagnate near initialization.

9.3 Vanishing Gradient Solutions

Several approaches address this problem:

1. Use ReLU activation where the derivative is 1 for active neurons, preventing exponential decay

2. Employ careful weight initialization schemes that maintain activation ranges

3. Apply batch normalization to reset activation distributions

4. Introduce skip connections (residual networks) providing direct gradient paths

9.4 The Exploding Gradient Problem

Conversely, if gradient factors exceed one in magnitude, their product grows exponentially. This causes gradient-based updates to become extremely large, leading to parameter divergence and training instability.

9.5 Exploding Gradient Solutions

To prevent exploding gradients:

1. Employ careful weight initialization avoiding large initial values
2. Apply gradient clipping, scaling gradients to bounded norms
3. Use L2 regularization penalizing large weight magnitudes
4. Reduce learning rate if divergence is observed

10 Practical Implementation Considerations

10.1 Gradient Checking

Before deploying a backpropagation implementation, numerical gradient checking verifies correctness. Compute finite difference approximations and compare against analytical gradients:

$$\text{Numerical gradient} \approx \frac{\mathcal{L}(\mathbf{W} + \epsilon) - \mathcal{L}(\mathbf{W} - \epsilon)}{2\epsilon} \tag{17}$$

Relative error should be less than 10^{-7} for single precision or 10^{-5} for double precision when the analytical gradient is correct.

10.2 Monitoring Training Dynamics

During training, several diagnostics indicate proper functioning:

1. Training loss should decrease monotonically initially
2. Check for NaN or infinite loss values indicating divergence
3. Monitor weight and gradient statistics per layer
4. Plot training and validation loss curves to detect overfitting
5. Verify activation distributions remain in reasonable ranges

11 Conclusion

Backpropagation and gradient-based optimization form the mathematical foundations enabling deep learning. The algorithm efficiently computes gradients through the chain rule, allowing parameter updates that minimize the loss function. Understanding various optimization algorithms, loss functions, and activation functions provides practitioners with tools to construct effective training pipelines. The challenges of gradient flow in deep networks motivate architectural innovations and training techniques addressed in subsequent reports.

References

- Nielsen, M. A. (2015). Neural networks and deep learning. Determination Press.
- Kingma, D. P., Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.
- Rumelhart, D. E., Hinton, G. E., Williams, R. J. (1986). Learning representations by back-propagating errors. Nature, 323(6088), 533-536.
- Goodfellow, I., Bengio, Y., Courville, A. (2016). Deep learning. MIT press.
- Nesterov, Y. (2013). Introductory lectures on convex optimization. Springer Science+Business Media.