Artificial Intelligence and
Machine Learning Club
Unionville High School

Multi-layer Perceptrons: An Introduction Machine Learning and Artificial Intellegence

Peter Lu

349293191@gapps.yrdsb.ca

Lesson notes for students wishing to enhance their
understanding and knowledge of artificial
intelligence and machine learning

Meeting 1 Supplemental
Unionville High School
ON, Canada

October 14, 2024

Abstract

This document presents comprehensive course notes on the foundational concepts of artificial intelligence, with a particular focus on multilayer perceptrons (MLPs). It begins by introducing the fundamental principles of artificial intelligence and neural networks, the concept of **gradient descent**, **loss functions,** and **back-propagation**. It examines the architecture of MLPs and how layers of interconnected neurons can model complex, non-linear relationships in data.

A significant portion of the notes is dedicated to loss functions, which quantify the difference between predicted and actual outcomes, serving as a crucial guide for model optimization. Various types of loss functions are discussed, explaining their suitability for different tasks.

The document also goes into optimization algorithms and the role of gradient descent in training neural networks. It explores how gradient descent and its variants are used to adjust the weights and biases of the network to minimize the loss function effectively. Practical considerations such as learning rates, convergence criteria, computational efficiency, and some calculus are also addressed.

These lesson notes aim to equip readers with a solid understanding of multilayer perceptrons, loss functions, and optimization techniques. This foundational knowledge is not essential for advancing in the club but is highly recommended if you want to learn more about the field of artificial intelligence and applying neural network models to real-world problems.

Contents

1 Introduction

1.1 What is Machine Learning?

Machine Learning (ML) is a subset of Artificial Intelligence focused on enabling machines to learn from data and improve their performance **without being explicitly programmed**. ML algorithms identify patterns, make predictions, and derive insights from large datasets, allowing systems to adapt and enhance their functionality over time.

In this note, we will mostly be focusing on the Multilayer Perceptron. The Multilayer Perceptron is a class of feedforward artificial neural network that consists of at least three layers of nodes: an input layer, one or more hidden layers, and an output layer. Each node, except for the input nodes, is a neuron that uses a nonlinear activation function.

1.2 Prerequisite Concepts

Before diving too deep into the theory of machine learning, there are a couple of important concepts one should be familiar with, including:

- Basic understanding of linear algebra (I mean really basic)
- Fundamental calculus concepts (Idea of derivatives and slopes)
- Probability and statistics fundamentals (Go to math club)
- Basic programming skills (preferably in Python)

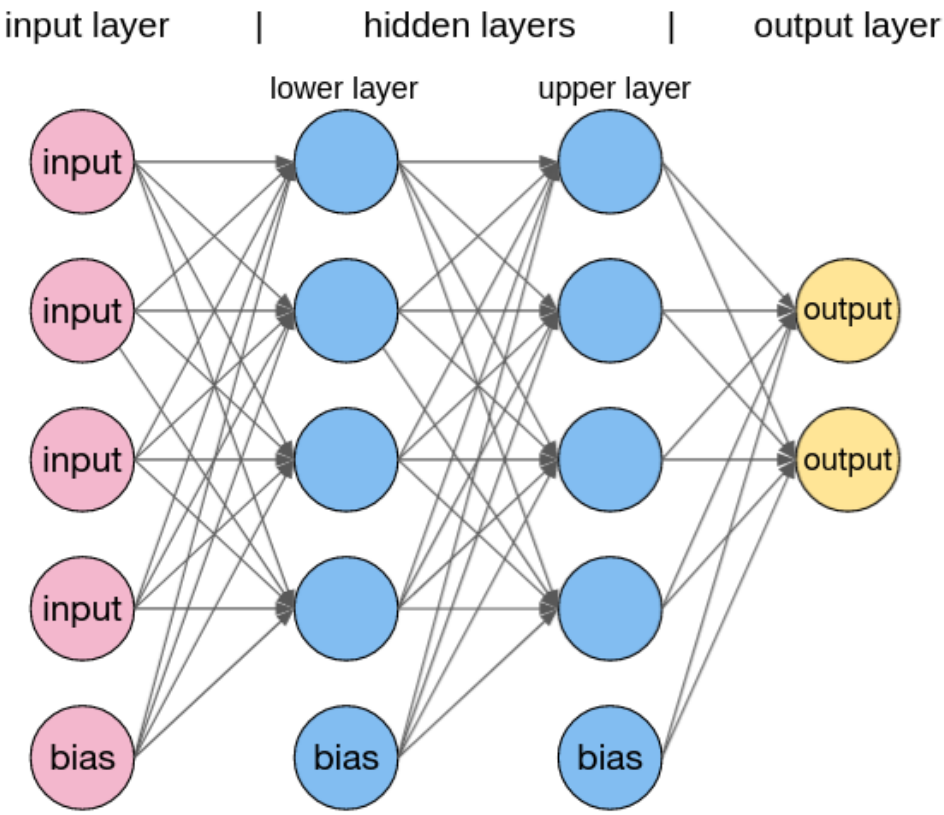

Figure 1.1: The Multilayer Perceptron

2 Learning Paradigms

2.1 Overview of Learning Paradigms

A learning paradigm in machine learning refers to the overarching framework or methodology that defines how an algorithm **learns from data**. It encompasses the type of data used, the learning process, the objectives, and the strategies employed to model and solve problems. These paradigms provide structured approaches to tackling challenges in ML by categorizing problems and solutions into distinct frameworks.

2.2 Supervised Learning

2.2.1 Definition

Supervised Learning is the most prevalent and widely used learning paradigm in machine learning. It involves training algorithms on **labeled datasets**, where each training example (input) is paired with an **output label**. The primary goal is for the model to learn a mapping from inputs to outputs, enabling it to make accurate predictions on unseen data.

2.2.2 Key Characteristics

- **Labeled Data**: Requires a dataset where each input instance is associated with a correct output label.
- **Objective**: Learn a function that maps inputs to desired outputs.
- **Feedback Mechanism**: Uses the known labels to evaluate and improve the model's predictions through error correction (e.g., Loss Functions and Back-Propagation).

2.2.3 Types of Supervised Learning

Classification

- **Definition**: Assigning inputs to predefined categories or classes.

- **Examples**: Email spam detection (spam or not), image recognition (identifying objects within images).
- **Algorithms**: Logistic Regression, Support Vector Machines (SVM), Decision Trees, Random Forests, Neural Networks.

Regression

- **Definition**: Predicting continuous numerical values based on input features.
- **Examples**: Predicting house prices, forecasting stock market trends, estimating product demand.
- **Algorithms**: Linear Regression, Ridge Regression, Lasso Regression, Support Vector Regression (SVR), Neural Networks.

2.3 Other Learning Paradigms

- Unsupervised Learning
- Semi-Supervised Learning
- Reinforcement Learning
- Self-Supervised Learning
- Transfer Learning

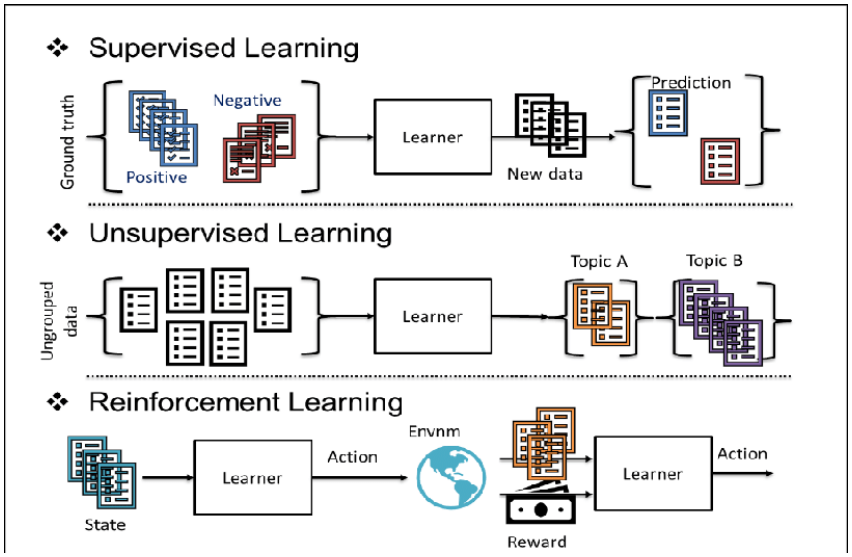

Figure 2.1: Machine Learning Paradigms

3 Loss Functions & Evaluation

3.1 Introduction to Model Evaluation

In machine learning, developing a robust model involves not only selecting the right algorithms and training them effectively but also rigorously evaluating their performance. Two fundamental concepts that are part of this process are model evaluation and loss functions. Understanding these concepts is essential for building models that generalize well to unseen data and perform reliably in real-world applications.

3.2 Loss Functions

3.2.1 Definition

(Loss Function) A mathematical function that quantifies the difference between the predicted outputs of a model and the actual target values.

A loss function, also known as a cost function or objective function, is a mathematical formula that measures the discrepancy between a model's predictions and the true values in the training data. By assigning a numerical value to this difference, loss functions provide a way to evaluate how well a machine learning model is performing during the training process. Due to the nature of machine learning **not being explicitly programmed**, the loss function is a way of letting the model know how wrong it is. In the next chapter about optimization, we will discuss how to tell the machine learning model how to fix its wrongdoings.

3.2.2 Mean Squared Error (MSE)

Mean Squared Error (MSE) is the most fundamental metric/loss function used in regression tasks. It measures the average squared difference between the observed actual outcomes and the outcomes predicted by a model. It quantifies the quality of a predictor or estimator by calculating the average of the squares of the errors—that is, the average squared difference between the estimated values and the actual value.

Mathematical Definition

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

where:

- n is the number of observations.
- y_i is the actual value for the i-th observation.
- $\hat{y}_i$ is the predicted value for the i-th observation.

MSE is used to evaluate the performance of a regression model. A lower MSE indicates a better fit of the model to the data. When a regression model is set to minimize the MSE, the model parameters (hidden values) are adjusted to reduce the prediction errors.

Calculation Example

Let's see an example of MSE in action by considering a simple linear regression example to illustrate how the Mean Squared Error (MSE) is used.

Suppose we have the following dataset with $n = 5$ observations:

Table 3.1: Actual Values

Observation	**Actual Value (y_i)**
1	3
2	5
3	2
4	7
5	4

Assume our regression model predicts the following values:

$$\hat{y} = \{2.8, 5.2, 2.5, 6.9, 3.9\}$$

Compute the squared errors for each observation as follows:
The **Mean Squared Error (MSE)** is then calculated as:

$$\text{MSE} = \frac{0.35}{5} = 0.07$$

An MSE of 0.07 indicates that, on average, the squared difference between the actual and predicted values is 0.07. Depending on the context and scale of the data, this can be considered a good or poor fit. It's essential to compare MSE values across different models or datasets to assess relative performance.

Table 3.2: Squared Errors Calculation

Observation	y_i	$\hat{y}_i$	$(y_i - \hat{y}_i)^2$
1	3	2.8	$(3 - 2.8)^2 = 0.04$
2	5	5.2	$(5 - 5.2)^2 = 0.04$
3	2	2.5	$(2 - 2.5)^2 = 0.25$
4	7	6.9	$(7 - 6.9)^2 = 0.01$
5	4	3.9	$(4 - 3.9)^2 = 0.01$
Sum of Squared Errors			$0.04 + 0.04 + 0.25 + 0.01 + 0.01 = 0.35$

3.3 Common Loss Functions

Loss functions vary depending on the type of machine learning task:

For Classification:

- Binary Cross-Entropy (Log Loss): Measures the performance of a binary classifier by comparing predicted probabilities to actual binary labels.

$$\mathcal{L}(y, p) = -\left[y \cdot \log(p) + (1 - y) \cdot \log(1 - p)\right]$$

Explanation of the Components

  - $\mathcal{L}(y, p)$: The loss function, where y is the true label and p is the predicted probability.
  - y: Binary indicator (0 or 1) representing the true class label.
  - p: Predicted probability of the instance being in the positive class (i.e., $y = 1$).
  - log: Natural logarithm function.

- Categorical Cross-Entropy: Extends binary cross-entropy to multi-class classification problems.

$$\mathcal{L}(y, \mathbf{p}) = -\sum_{i=1}^{C} y_i \cdot \log(p_i)$$

Explanation of the Components

  - $\mathcal{L}(y, \mathbf{p})$: The loss function, where y is the true label in one-hot encoded form and $\mathbf{p}$ is the predicted probability distribution over all classes.
  - C: The total number of classes.

- y_i: Binary indicator (0 or 1) if class label i is the correct classification for the input.
- p_i: Predicted probability of the input belonging to class i.
- log: Natural logarithm function.

- Hinge Loss: Used primarily with Support Vector Machines (SVMs) for "maximum-margin" classification.

$$\mathcal{L}_{\text{Hinge}} = \max(0, 1 - y_i \cdot f(x_i))$$

Explanation of the Components

- $\mathcal{L}_{\text{Hinge}}$: The Hinge Loss function.
- y_i: The true label for the i^{th} observation, where $y_i \in \{-1, 1\}$.
- $f(x_i)$: The predicted score (not necessarily a probability) for the i^{th} observation. In SVMs, this is typically the output of the decision function.
- $\max(0, 1 - y_i \cdot f(x_i))$: Ensures that the loss is zero when the data point is correctly classified with a margin of at least 1.

For Regression:

- Mean Squared Error (MSE): Calculates the average of the squares of the differences between predicted and actual values.
  See above subsection 3.2.2

- Mean Absolute Error (MAE): Computes the average of the absolute differences between predictions and actual values.

$$\mathcal{L}_{\text{MAE}} = \frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y}_i|$$

Explanation of the Components

- $\mathcal{L}_{\text{MAE}}$: The Mean Absolute Error loss function.
- n: The number of observations in the dataset.
- y_i: The actual target value for the i^{th} observation.
- $\hat{y}_i$: The predicted value for the i^{th} observation.
- $|y_i - \hat{y}_i|$: The absolute error for the i^{th} observation.

- Huber Loss: Combines MSE and MAE, being quadratic for small errors and linear for large errors, which makes it robust to outliers.

$$\mathcal{L}_{\text{Huber}}(a) = \begin{cases} \frac{1}{2}a^2 & \text{for } |a| \leq \delta \\ \delta(|a| - \frac{1}{2}\delta) & \text{otherwise} \end{cases}$$

$$\text{where } a = y_i - \hat{y}_i$$

Explanation of the Components

- $\mathcal{L}_{\text{Huber}}(a)$: The Huber Loss function.
- a: The residual or error, calculated as the difference between the actual value y_i and the predicted value $\hat{y}_i$.
- δ: A threshold parameter that determines the point at which the loss function transitions from quadratic to linear.

3.4 Conclusion

Loss functions are fundamental components in machine learning. They provide a quantitative measure of the discrepancy between a model's predictions and the actual target values. They guide the optimization algorithms to adjust model parameters in a manner that minimizes prediction errors. The selection of an appropriate loss function is extremely, as it directly influences the model's performance, robustness, and ability to generalize to unseen data.

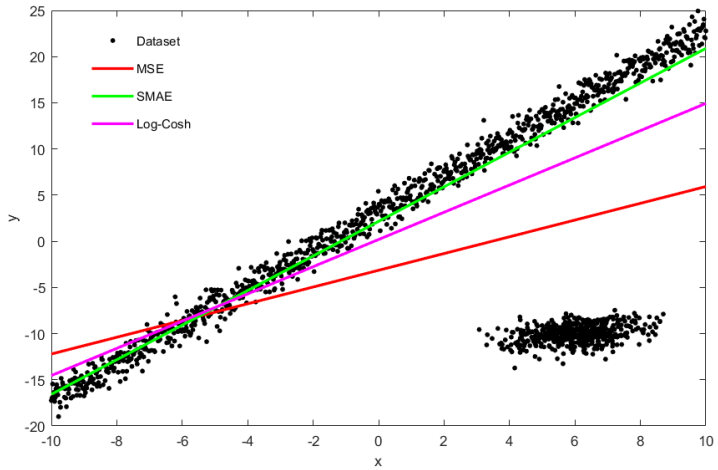

Figure 3.1: Different Loss Functions when faced with Outliers

4 Optimization & Gradient Descent

4.1 Introduction to Optimization

Optimization in machine learning is the delicate art of adjusting the parameters of a model to minimize the loss function. Effective optimization ensures that the model learns efficiently and generalizes well to new data. Most machine learning tasks involve minimizing a loss (cost) function, but some scenarios require maximization (e.g., maximizing accuracy).

4.1.1 Loss Landscape

A **loss landscape** is a multidimensional surface that represents the values of the loss function over the parameter space of a machine learning model. Each point in this space corresponds to a specific configuration of the model's parameters (e.g., weights and biases in a neural network), and the height of the surface at that point indicates the loss (error) associated with that parameter configuration. Formally, for a model with parameters θ, the loss landscape is the graph of the loss function $\mathcal{L}(\theta)$:

$$\mathcal{L}(\theta) = \mathcal{L}(\theta_1, \theta_2, \ldots, \theta_n)$$

where n is the number of parameters in the model.

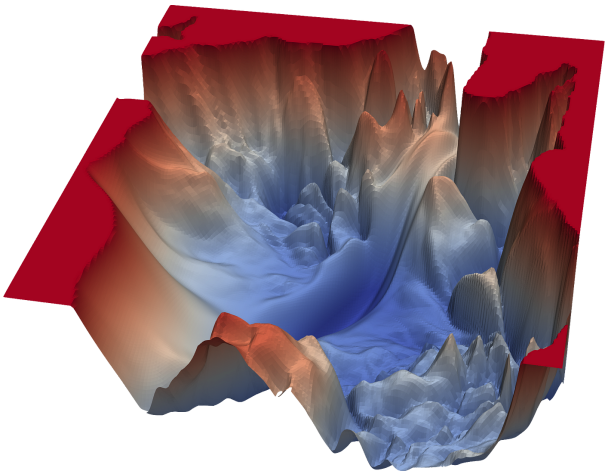

Figure 4.1: Example of a Loss Landscape

In the context of model optimization, the goal is find the global minima to maximize model accuracy (minimize loss).

4.2 Gradient Descent

4.2.1 Basic Concept

Gradient Descent is one of the most fundamental and widely used optimization algorithms in machine learning and deep learning. It is used to minimize an objective function, typically a loss function, by iteratively moving towards the steepest descent as defined by the negative of the gradient(slope of the loss landscape). Understanding Gradient Descent is crucial for effectively training models, as it directly influences the convergence speed and the quality of the final model parameters.

Mathematical Formulation

Given a differentiable loss function $\mathcal{L}(\theta)$, where θ represents the parameters of the model (e.g., weights and biases in a neural network), Gradient Descent aims to find the parameter values that minimize $\mathcal{L}(\theta)$.

The update rule for Gradient Descent is defined as:

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \cdot \nabla_\theta \mathcal{L}(\theta)$$

- θ_{old}: Current parameters.
- θ_{new}: Updated parameters.
- η: Learning rate, a hyperparameter that controls the step size.
- $\nabla_\theta \mathcal{L}(\theta)$: Gradient of the loss function with respect to the parameters.

4.2.2 How Gradient Descent Works

Gradient Descent operates by iteratively adjusting the model parameters in the direction that most reduces the loss function. The process can be broken down into the following steps:

1. **Initialization:** Start with an initial guess for the parameters θ, which can be randomly selected or based on some heuristic.
2. **Compute Gradient:** Calculate the gradient of the loss function $\nabla_\theta \mathcal{L}(\theta)$ at the current parameter values. The gradient indicates the direction of the steepest ascent.

3. **Update Parameters:** Adjust the parameters by moving a small step in the opposite direction of the gradient:

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \cdot \nabla_\theta \mathcal{L}(\theta)$$

4. **Iteration:** Repeat the gradient computation and parameter update steps until convergence, i.e., until the changes become negligible.

Example: Gradient Descent in Linear Regression

Consider a simple linear regression problem where we aim to fit a line $y = \theta_0 + \theta_1 x$ to a set of data points (x_i, y_i).

Loss Function

The Mean Squared Error (MSE) loss function for linear regression is defined as:

$$\mathcal{L}(\theta_0, \theta_1) = \frac{1}{n} \sum_{i=1}^{n} (y_i - (\theta_0 + \theta_1 x_i))^2$$

Gradient Computation

The gradients of the loss function with respect to θ_0 and θ_1 are:

$$\frac{\partial \mathcal{L}}{\partial \theta_0} = -\frac{2}{n} \sum_{i=1}^{n} (y_i - (\theta_0 + \theta_1 x_i))$$

$$\frac{\partial \mathcal{L}}{\partial \theta_1} = -\frac{2}{n} \sum_{i=1}^{n} x_i (y_i - (\theta_0 + \theta_1 x_i))$$

where $\frac{\partial \mathcal{L}}{\partial \theta_0}$ and $\frac{\partial \mathcal{L}}{\partial \theta_1}$ are the partial derivative of the loss function in respect of θ_0 and θ_1 respectively.

Parameter Update Rules

Using Gradient Descent, the update rules for θ_0 and θ_1 are:

$$\theta_0 := \theta_0 - \eta \cdot \frac{\partial \mathcal{L}}{\partial \theta_0}$$

$$\theta_1 := \theta_1 - \eta \cdot \frac{\partial \mathcal{L}}{\partial \theta_1}$$

Iterative Process

1. **Initialize:** Start with initial guesses for θ_0 and θ_1, say $\theta_0 = 0$ and $\theta_1 = 0$.

2. **Compute Gradients:** Calculate the partial derivatives of the loss function with respect to each parameter.
3. **Update Parameters:** Adjust θ_0 and θ_1 using the update rules.
4. **Repeat:** Continue the process until the loss converges to a minimum.

Visualization of Gradient Descent

Visualizing Gradient Descent helps in understanding how the algorithm navigates the loss landscape towards the minimum.

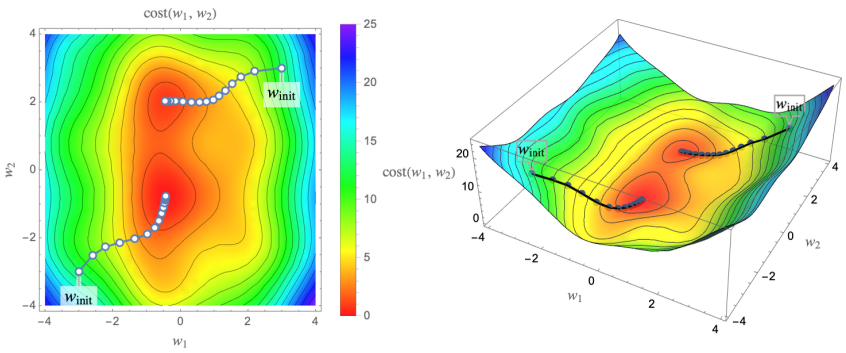

Figure 4.2: A 2D visualization of Gradient Descent navigating the loss landscape towards the minimum.

Learning Rate (η)

The learning rate is a crucial hyperparameter in Gradient Descent that determines the size of the steps taken towards the minimum.

- **Too Small:** Leads to slow convergence, requiring many iterations.
- **Too Large:** Can cause overshooting, leading to divergence or oscillations around the minimum.
- **Optimal Learning Rate:** Balances convergence speed and stability.
- **Strategies for Choosing Learning Rate:**
  - **Manual Tuning:** Experimenting with different values.
  - **Learning Rate Schedules:** Adjusting the learning rate over time (e.g., decay schedules).
  - **Adaptive Methods:** Algorithms that adjust the learning rate automatically (e.g., Adam, RMSprop).

Several variants of Gradient Descent have been developed to address different challenges related to convergence speed, computational efficiency, and handling large datasets.

Momentum and Accelerated Gradient Descent

To address some of the limitations of standard Gradient Descent, techniques like Momentum are employed. Momentum helps accelerate Gradient Descent in relevant directions and dampens oscillations.

Momentum

The Momentum update rule introduces a velocity term that accumulates the gradient vectors in directions of persistent reduction.

$$v_t = \gamma v_{t-1} + \eta \cdot \nabla_\theta \mathcal{L}(\theta)$$

$$\theta = \theta - v_t$$

- γ: Momentum coefficient (typically between 0.9 and 0.99).
- **Benefits:** Helps navigate ravines and accelerates convergence.

Adaptive Learning Rate Methods

Adaptive methods adjust the learning rate for each parameter dynamically based on past gradients, improving convergence speed and robustness.

4.2.3 Variants of Gradient Descent

1. Batch Gradient Descent

Batch Gradient Descent computes the gradient of the loss function using the entire training dataset. It guarantees convergence to the global minimum for convex loss functions but can be computationally expensive for large datasets.

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \cdot \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta \mathcal{L}(\theta; x_i, y_i)$$

- **Pros:** Stable convergence, deterministic updates.
- **Cons:** High computational cost, memory inefficiency for large datasets.

2. Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent updates the parameters using the gradient computed from a single randomly selected training example. This introduces noise into the updates, which can help escape local minima but may lead to oscillations around the minimum.

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \cdot \nabla_\theta \mathcal{L}(\theta; x_i, y_i)$$

- **Pros:** Faster updates, can handle large datasets, introduces noise that can help escape local minima.
- **Cons:** Noisy convergence, may never settle at the exact minimum.

3. Mini-Batch Gradient Descent

Mini-Batch Gradient Descent strikes a balance between Batch and Stochastic Gradient Descent by computing the gradient on small subsets of the data (mini-batches). This approach combines the efficiency of batch processing with the robustness of stochastic updates.

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \cdot \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta \mathcal{L}(\theta; x_i, y_i)$$

- **Pros:** Efficient computation, reduced variance in updates compared to SGD, better convergence properties.
- **Cons:** Requires careful selection of mini-batch size.

4.3 Backpropagation

Backpropagation, short for "backward propagation of errors," is an algorithm for computing the gradient of the loss function with respect to each weight in the network. It is the way neural networks compute the gradient (slope) of the loss landscape for gradient descent.

4.3.1 Intuition Behind Backpropagation

Backpropagation works by calculating how much each weight in a neural network contributes to the error (loss). It does this by finding the derivative of the loss function with respect to each weight, which tells us the slope or sensitivity of the error to changes in that weight. The steeper the slope, the more adjusting that weight will impact the loss. Using these slopes, the network updates the weights in the direction that reduces the error, improving its performance through gradient descent. By finding the partial derivative in respect to each model parameter, the gradient descent algorithm can systematically decrease the loss function, thus increasing accuracy.

4.3.2 Chain Rule Application

Backpropagation leverages the chain rule from calculus to compute the gradient of the loss function with respect to each parameter. Essentially, it decomposes the complex network into a series of simpler computations, allowing gradients to be propagated backward from the output to the input.

4.3.3 Error Propagation

The core idea is to quantify how a change in each parameter affects the overall error. By systematically adjusting parameters in the direction that most reduces the error, the network learns to make more accurate predictions.

4.3.4 Local Adjustments for Global Optimization

Each parameter update is based on local information (the immediate gradient). However, when combined across all parameters and iterations, these local adjustments lead to a global optimization of the network's performance.

4.4 Forward Pass

Consider a simple neural network with the following architecture:

- **Input Layer:** 1 neuron
- **Hidden Layer:** 1 neuron with sigmoid activation
- **Output Layer:** 1 neuron (linear activation)

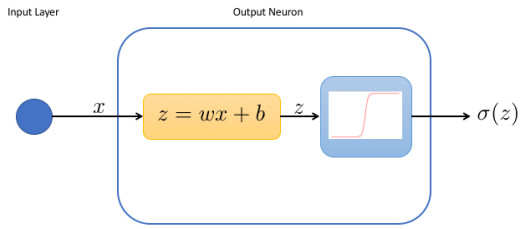

Figure 4.3: single neuron

The forward pass involves computing the output of the network given an input by propagating the input through each layer.

4.4.1 Input to Hidden Layer

$$z_1 = w_1 x + b_1$$
$$h = \sigma(z_1) = \frac{1}{1 + e^{-z_1}}$$

Where:

- x is the input.
- w_1 is the weight connecting the input to the hidden neuron.
- b_1 is the bias of the hidden neuron.
- $\sigma(z)$ is the sigmoid activation function.
- h is the activation of the hidden neuron.

4.4.2 Hidden to Output Layer

$$z_2 = w_2 h + b_2$$
$$y = z_2$$

Where:

- w_2 is the weight connecting the hidden neuron to the output neuron.
- b_2 is the bias of the output neuron.
- y is the output of the network.

4.4.3 Loss Function

We use the Mean Squared Error (MSE) as the loss function:

$$L = \frac{1}{2}(y - t)^2$$

Where t is the target value.

4.5 Backward Pass (Backpropagation)

Backpropagation computes the gradients of the loss function with respect to each weight and bias, facilitating their update to minimize the loss.

4.5.1 Gradients for Output Layer

$$\frac{\partial L}{\partial y} = y - t$$
$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial w_2} = (y - t) \cdot h$$
$$\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial b_2} = (y - t) \cdot 1 = y - t$$

4.5.2 Gradients for Hidden Layer

$$\frac{\partial L}{\partial h} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial h} = (y - t) \cdot w_2$$
$$\sigma'(z_1) = h(1 - h)$$
$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial h} \cdot \sigma'(z_1) \cdot \frac{\partial z_1}{\partial w_1} = (y - t)w_2h(1 - h)x$$
$$\frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial h} \cdot \sigma'(z_1) \cdot \frac{\partial z_1}{\partial b_1} = (y - t)w_2h(1 - h) \cdot 1$$

4.6 Parameter Updates

Using Gradient Descent, we update each parameter by subtracting a fraction (learning rate η) of its gradient.

$$w_2 \leftarrow w_2 - \eta \frac{\partial L}{\partial w_2} = w_2 - \eta(y - t)h$$
$$b_2 \leftarrow b_2 - \eta \frac{\partial L}{\partial b_2} = b_2 - \eta(y - t)$$
$$w_1 \leftarrow w_1 - \eta \frac{\partial L}{\partial w_1} = w_1 - \eta(y - t)w_2h(1 - h)x$$
$$b_1 \leftarrow b_1 - \eta \frac{\partial L}{\partial b_1} = b_1 - \eta(y - t)w_2h(1 - h)$$

4.7 Numerical Example

Let's walk through a numerical example to illustrate the backpropagation process.

4.7.1 Given:

- Input: $x = 1$

- Target: $t = 0.5$
- Initial Weights and Biases:

$$w_1 = 0.4$$
$$b_1 = 0.1$$
$$w_2 = 0.3$$
$$b_2 = 0.2$$

- Learning Rate: $\eta = 0.1$

4.7.2 Forward Pass

$$\begin{aligned}
z_1 &= w_1 x + b_1 = 0.4 \cdot 1 + 0.1 = 0.5 \\
h &= \sigma(z_1) = \frac{1}{1 + e^{-0.5}} \approx 0.622459 \\
z_2 &= w_2 h + b_2 = 0.3 \cdot 0.622459 + 0.2 \approx 0.3867377 \\
y &= z_2 \approx 0.3867377 \\
L &= \frac{1}{2}(y - t)^2 = \frac{1}{2}(0.3867377 - 0.5)^2 \approx 0.005488
\end{aligned}$$

4.7.3 Backward Pass

$$\begin{aligned}
\frac{\partial L}{\partial y} &= y - t = 0.3867377 - 0.5 = -0.1132623 \\
\frac{\partial L}{\partial w_2} &= (y - t)h = -0.1132623 \cdot 0.622459 \approx -0.070506 \\
\frac{\partial L}{\partial b_2} &= y - t = -0.1132623 \\
\frac{\partial L}{\partial h} &= (y - t)w_2 = -0.1132623 \cdot 0.3 \approx -0.0339787 \\
\sigma'(z_1) &= h(1 - h) = 0.622459 \cdot (1 - 0.622459) \approx 0.2350037 \\
\frac{\partial L}{\partial w_1} &= -0.0339787 \cdot 0.2350037 \cdot 1 \approx -0.007994 \\
\frac{\partial L}{\partial b_1} &= -0.0339787 \cdot 0.2350037 \approx -0.007994
\end{aligned}$$

4.7.4 Parameter Updates

$$
\begin{aligned}
w_2 &\leftarrow 0.3 - 0.1 \cdot (-0.070506) = 0.3 + 0.0070506 \approx 0.3070506 \\
b_2 &\leftarrow 0.2 - 0.1 \cdot (-0.1132623) = 0.2 + 0.01132623 \approx 0.21132623 \\
w_1 &\leftarrow 0.4 - 0.1 \cdot (-0.007994) = 0.4 + 0.0007994 \approx 0.4007994 \\
b_1 &\leftarrow 0.1 - 0.1 \cdot (-0.007994) = 0.1 + 0.0007994 \approx 0.1007994
\end{aligned}
$$

4.7.5 Summary of Updates

Parameter	Initial Value	Updated Value
w_1	0.4	0.4007994
b_1	0.1	0.1007994
w_2	0.3	0.3070506
b_2	0.2	0.21132623

Table 4.1: Parameter Updates after One Iteration

4.8 Conclusion

Backpropagation is an algorithm that enables neural networks to learn from data by iteratively minimizing the loss function. Through the application of calculus, specifically the chain rule, it systematically adjusts the network's parameters to improve performance. Understanding both the mathematical foundation and the intuitive concepts behind backpropagation is crucial for effectively designing and training neural networks.

5 The Multilayer Perceptron

5.1 Neurons and Activation Functions

5.1.1 Neurons

Neurons are the fundamental units of any deep learning algorithm. Each neuron receives input, applies weights and biases, and passes the result through an activation function.

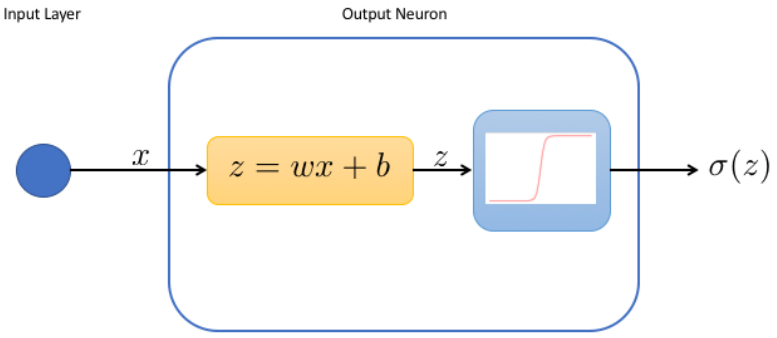

Figure 5.1: A single neuron with an activation function

An neuron is the basic building block of an artificial neural network. It's inspired by the biological neurons in the human brain but operates in a simplified mathematically to process information.

A neuron has 3 main parts, the weight, the bias, and the activation function.

5.2 Components of an Artificial Neuron

5.2.1 Inputs (Features)

- Data points or signals received by the neuron.
- Represented as $x_1, x_2, x_3, \ldots, x_n$.

5.2.2 Weights (w)

- Each input has an associated weight ($w_1, w_2, w_3, \ldots, w_n$).
- Weights determine the importance or strength of each input.
- Adjustable parameters that the network learns during training.

5.2.3 Bias (b)

- An additional parameter that allows the activation function to be shifted.
- Helps the model fit the data better by providing flexibility.

5.2.4 Summation Function

- Calculates the weighted sum of inputs plus the bias.
- Mathematically:

$$z = w_1x_1 + w_2x_2 + \ldots + w_nx_n + b$$

5.2.5 Activation Function (σ)

- Introduces non-linearity into the model, enabling it to learn complex patterns.
- Common activation functions include Sigmoid, ReLU (Rectified Linear Unit), and Tanh.

5.2.6 Output (y)

- The final output of the neuron after applying the activation function.
- Mathematically:

$$y = \sigma(z)$$

5.3 How Does a Neuron Work?

Let's break down the neuron's operation step-by-step with a simple example.

Suppose we have a neuron with two inputs trying to decide whether to activate (output 1) or not (output 0) based on the inputs.

Inputs and Weights

$$x_1 = 0.5 \quad w_1 = 0.8$$
$$x_2 = 0.3 \quad w_2 = 0.2$$

Bias

$$b = 0.1$$

Summation

$$z = (w_1 \times x_1) + (w_2 \times x_2) + b = (0.8 \times 0.5) + (0.2 \times 0.3) + 0.1 = 0.4 + 0.06 + 0.1 = 0.56$$

Activation Function

Using the **Sigmoid** activation function:

$$\sigma(z) = \frac{1}{1 + e^{-z}} = \frac{1}{1 + e^{-0.56}} \approx 0.63$$

Output

The neuron outputs approximately **0.63**. If using a threshold (e.g., 0.5), the neuron activates (outputs 1) since $0.63 > 0.5$.

5.3.1 Visualization

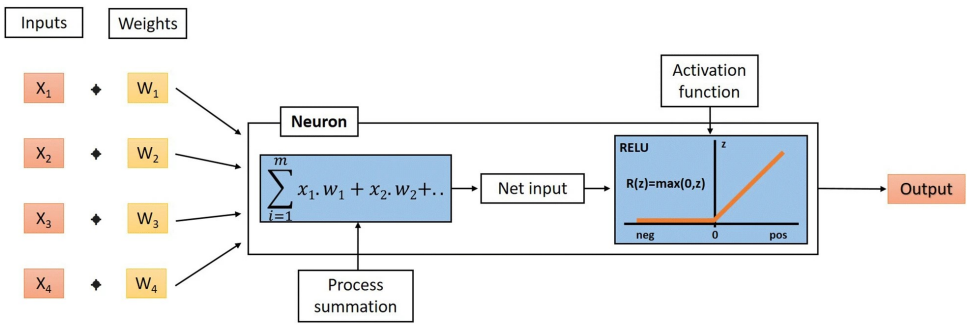

Figure 5.2: Visualization of an Artificial Neuron

Figure 5.2: Inputs x_1 and x_2 are multiplied by weights w_1 and w_2, summed with bias b, passed through the activation function σ, and produce the output y.

5.3.2 Activation Functions

Activation functions introduce non-linearity into the network, allowing it to learn and model complex patterns. Common activation functions include:

- Sigmoid
- Tanh
- ReLU (Rectified Linear Unit)
- Leaky ReLU

The most commonly used activation fuctions is the ReLU (Rectified Linear Unit) and the Sigmoid.

5.3.3 Why are Activation Functions Important

Activation functions are crucial in neural networks because they introduce non-linearity, allowing the model to learn complex patterns. Without them, the network would just be a series of linear transformations (matrix multiplications), no matter how many layers there are. A stack of linear layers can only model linear relationships, meaning it can't capture complex structures like curves or decision boundaries required for tasks like image recognition or speech processing.

For example, if you use only linear functions (like $y = Wx + b$), any combination of these remains a linear function, so the model's output is limited to straight lines. Activation functions like ReLU, sigmoid, or tanh break this limitation by introducing non-linearity, enabling the network to approximate more complex, curved functions and learn from diverse data distributions. This makes neural networks powerful enough to model intricate relationships in the data.

5.3.4 Rectified Linear Unit (ReLU)

ReLU is one of the most widely used activation functions in deep learning due to its simplicity and effectiveness.

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

Properties

- Computationally efficient.
- Mitigates the vanishing gradient problem.
- Can lead to "dead neurons" where neurons output zero for all inputs.

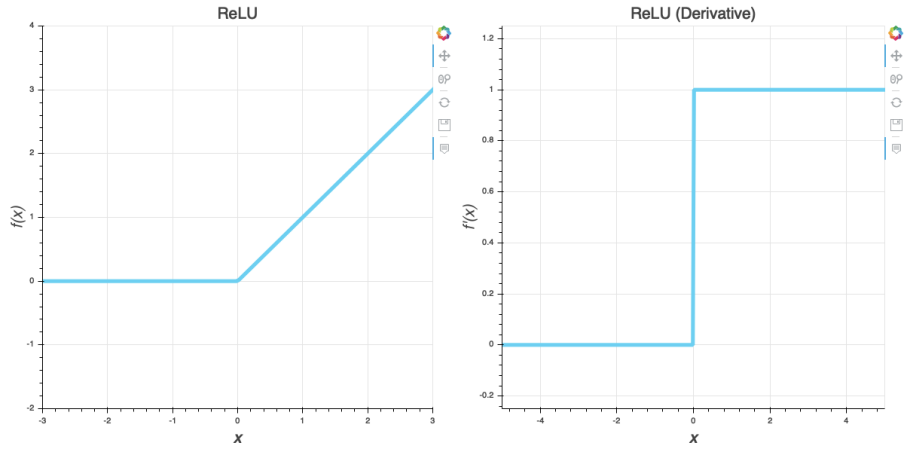

Figure 5.3: ReLU and its derivative

5.3.5 Sigmoid Function

The sigmoid activation function maps input values to an output range between 0 and 1.

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

Properties

- Smooth gradient.
- Outputs values in the (0, 1) range, suitable for binary classification.
- Prone to the vanishing gradient problem.

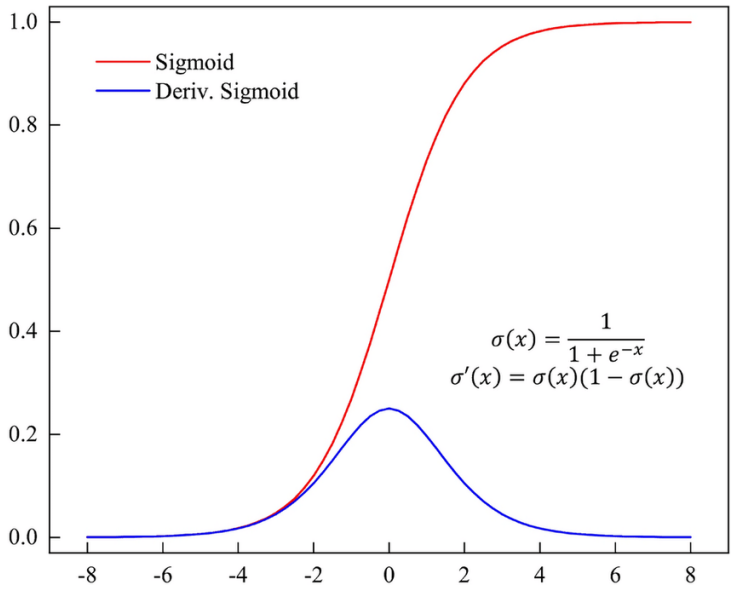

Figure 5.4: Sigmoid and its derivative

5.3.6 Softmax Function

Primarily used in the output layer for multi-class classification problems, the Softmax function converts logits (the numbers being outputed after passing through the neural net) into probabilities.

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

where K is the number of classes.

Properties

- Outputs sum to 1, representing probabilities.
- Sensitive to outliers.

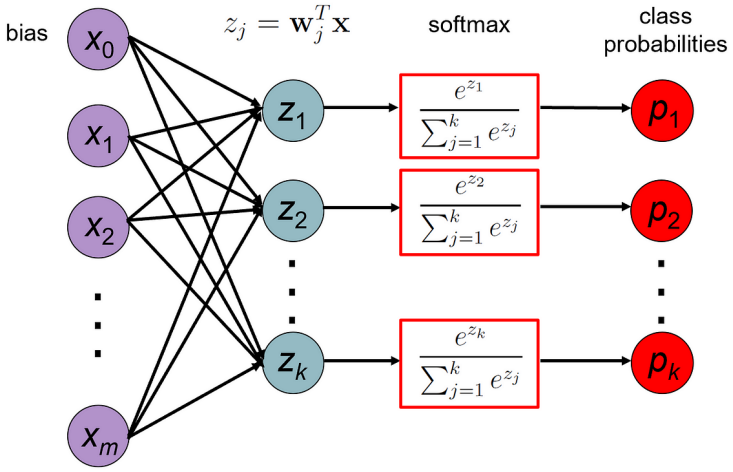

Figure 5.5: The Softmax function in a Neural Network

5.4 The Multilayer Perceptron

A Multi-Layer Perceptron (MLP) is a fundamental type of neural network that is used in various machine learning tasks like classification, regression, and pattern recognition. MLPs are essential for learing about more complex neural network architectures and form the backbone of many deep learning models (CNNs and Transformers).

A Multi-Layer Perceptron (MLP) is a class of feedforward artificial neural networks. This means the data goes through the model without loops. They consist of at least three layers of nodes:

- Input Layer: Receives the initial data.

- Hidden Layers: Intermediate layers that process inputs from the previous layer.
- Output Layer: Produces the final output.

Each node (neuron) in these layers is connected to nodes in the subsequent layer, and each connection has an associated weight that adjusts as the network learns.

5.5 Forward Propagation

Forward propagation is the process by which input data passes through the network to produce an output.

5.5.1 Steps in Forward Propagation

1. **Input Layer:** Receive input features $\mathbf{x} = [x_1, x_2, \ldots, x_n]$.
2. **Hidden Layers:** For each hidden layer, compute the weighted sum and apply the activation function.
3. **Output Layer:** Compute the final output using the activation function appropriate for the task.

5.6 Example of a Multilayer Perceptron

To further better our understanding of multilayer perceptrons, consider the following example.

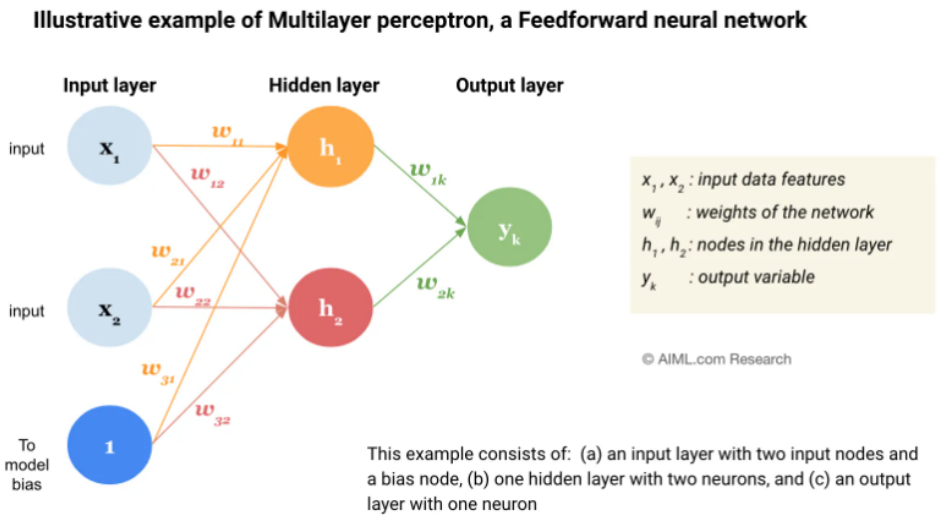

Figure 5.6: MLP with 2 inputs and 1 hidden layer

1. **Input Layer**: The inputs x_1 and x_2 enter the MLP on the left side. These 2 input neurons pass information to the hidden layers.

2. **Hidden Layer**: The hidden layer consists of 2 neurons h_1 and h_2 with weights $w_{11}, w_{12}, w_{21}, w_{22}$ and the bias weights w_{31}, w_{32}. Each neuron in the hidden layer then computes its final value by summing up the data from the input neurons multiplied by the **weights** and the **bias**.

$$h_i = \sum_{j=1}^{n} x_j w_{ji} + b$$

where h_i is the hidden neuron, n is the number of neurons in the previous layer, and b is the bias term associated with neuron h_i. In this simplified example there is no activation functions but generally, value the neuron possesses after the summation will be passed through the activation function.

Lets assume the use of the **sigmoid** activation function

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

3. **Output Layer**: The output layer is responsible for organizing the logits (model data) into information that is usable (probabilities, predictions). In classification, the logits of the model must be passed through a **Softmax** function to create predicted probabilities

$$y_k = \sum_{j=1}^{n} x_j w_{jk}$$

$$\text{Softmax}(y_i) = \frac{e^{y_i}}{\sum_{j=1}^{K} e^{y_j}}$$

5.7 Conclusion

Multilayer Perceptrons are by far one of the most important machine learning architectures to understand. They work by combining many layers of neurons and activation functions to capture linear and non-linear relationships in data.

6 Conclusion of Note

6.1 Summary of Key Points

Throughout this document, we have explored the foundational aspects of machine learning, with a particular emphasis on Multi-layer Perceptrons (MLPs), loss functions, and optimization techniques. We began by defining machine learning and outlining the prerequisites necessary for understanding its core concepts. The exploration of various learning paradigms, especially supervised learning, provided a framework for distinguishing between different types of machine learning tasks such as classification and regression.

A significant portion of our discussion focused on loss functions, and their role in quantifying the discrepancy between model predictions and actual outcomes. We examined key loss functions like Mean Squared Error (MSE), Mean Absolute Error (MAE), and Cross-Entropy, and their applications and mathematical representations.

In the optimization chapter, we introduced Gradient Descent and its variants, explaining how these algorithms iteratively adjust model parameters to minimize the loss function. The detailed walkthrough of backpropagation showed how neural networks learn by propagating errors backward through the network, tuning of weights and biases. Finally, the chapter on MLPs taught you the architecture of neural networks, the function of neurons and activation functions, and the mechanics of forward and backward propagation within these models.

6.2 Future Directions

The field of neural networks and artificial intelligence is rapidly evolving and there are many ways for further exploration and advancement. Your future studies could include more complex neural network architectures such as Convolutional Neural Networks (CNNs) for image processing, Recurrent Neural Networks (RNNs) for sequence modeling, and Transformers for natural language processing.

Another area you should look at are advanced optimization algorithms that enhance

training efficiency and model robustness. Techniques like Adam, RMSprop, and AdaGrad offer adaptive learning rates that can improve convergence rates and handle sparse gradients more effectively. Research into these and other optimization methods can lead to more stable and faster training processes, especially for large-scale neural networks.

6.3 Final Thoughts

Learning a new field like machine learning and artificial intelligence could be pretty daunting at first. The concepts of MLPs, loss functions, and optimization techniques are quite hard to comprehend. So if you even understood a bit of this, you are already doing well. I hope we can see you every week at AI club so we can learn more about AI together. It is the future after all!

7 Additional Resources

To further enhance your understanding of machine learning and neural networks, the following resources are highly recommended:

7.1 StatQuest with Josh Starmer

StatQuest is a YouTube channel hosted by Josh Starmer that provides explanations of statistical and machine learning concepts. The videos break down topics into simple, digestible pieces, that make them accessible for learners at all levels.

- **StatQuest YouTube Channel**
- Topics covered include machine learning algorithms, statistics, and data science fundamentals.

7.2 Machine Learning by Andrew Ng

Professor Andrew Ng's lectures are renowned for their comprehensive introduction to machine learning, covering both theoretical foundations and practical implementations.

- **Machine Learning Course on Coursera**
- Topics include supervised learning, unsupervised learning, neural networks, and best practices in machine learning.

7.3 Deep Learning Specialization by Andrew Ng

For a more in-depth exploration of neural networks and deep learning, this specialization goes into advanced topics and provides hands-on experience with real-world applications.

- **Deep Learning Specialization on Coursera**
- Courses cover convolutional neural networks, recurrent neural networks, and sequence models.

7.4 Books and Publications

- **Deep Learning** by Ian Goodfellow, Yoshua Bengio, and Aaron Courville: A comprehensive resource that covers a wide range of deep learning topics, from basic concepts to advanced techniques.
- **Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow** by Aurélien Géron: A practical guide that provides step-by-step instructions for implementing machine learning models.

7.5 Online Communities and Forums

Engaging with the machine learning community can provide additional support and insights.

- **Stack Overflow**: A platform to ask technical questions and share knowledge with professionals and enthusiasts.
- **r/MachineLearning on Reddit**: A community for discussions, news, and resources related to machine learning.