Task 2: Training on Understanding Artificial Neural Networks (ANNs)

1. Task Overview

Artificial Neural Networks (ANNs) are among the most fascinating tools in modern artificial intelligence. At their core, they are systems built to mimic the way our brains process information. Just as biological neurons receive signals, process them, and pass them on, ANNs consist of artificial “neurons” that work together to recognize patterns, make decisions, and even learn from their mistakes.

Although the idea has been around since the 1940s, the real breakthrough came when researchers realized that single neurons alone could only solve very simple problems. The invention of **multilayer architectures** and the **backpropagation** learning algorithm changed everything. Suddenly, these networks could learn far more complex relationships, opening the door to tasks like image recognition, speech processing, and natural language understanding.

At a high level, an ANN works like this: data enters through the **input layer**, passes through one or more **hidden layers** where it is transformed, and finally reaches the **output layer**, which produces the result. The connections between neurons have **weights**, which determine how strongly signals are passed. These weights are adjusted during training so that the network’s predictions improve over time. The “adjustment” process is guided by a **loss function**—a measure of how far the network’s predictions are from the desired answers—and powered by forward and backward propagation.

In this chapter, we will build our understanding step-by-step. We’ll start with the simplest unit, the **perceptron**, and then see how it grows into the more capable **multilayer perceptron (MLP)**. We’ll explore how information flows forward through the network, how errors flow backward to update the weights, why **activation functions** are essential for learning, and how different **loss functions** shape the training process.

By the end, you will see not just the mathematics behind ANNs, but also the intuition that makes them work. The aim is simple: when you finish this chapter, the inner workings of neural networks should feel less like a black box—and more like a clear, logical system you can confidently work with.

2. Task Objectives

This task focuses on building a foundational understanding of Artificial Neural Networks (ANNs), from their basic components to complex multilayer architectures and training mechanisms. The objectives below guide the learning process to ensure comprehension of both theoretical principles and practical intuition, summarized in Table 1.

Objective Type	Details
Ideological and Political Objectives	Cultivate a rigorous scientific attitude toward understanding AI fundamentals, emphasizing the importance of foundational knowledge in responsibly developing and applying neural network technologies, and appreciating the historical progress that enables modern ANNs.
Learning Objectives	1. Grasp the core components of ANNs, including artificial neurons, layers (input, hidden, output), weights, activation functions, and the role of backpropagation in training. 2. Understand the evolution from simple perceptrons to multilayer perceptrons (MLPs), and how architectural advancements enabled solving complex problems like pattern recognition and decision-making.
Skill Objectives	1. Explain the flow of information in ANNs: how input data propagates forward through layers, how loss functions measure prediction errors, and how backpropagation adjusts weights to minimize these errors. 2. Analyze the role of key elements (activation functions, loss functions) in network performance, and differentiate between their applications in various tasks (e.g., classification vs. regression).
Literacy Objectives	1. Develop intuition for the "black box" of neural networks, translating mathematical concepts into logical, understandable mechanisms that drive learning and prediction. 2. Gain the ability to critically evaluate how ANNs solve problems, laying the groundwork for further exploration into advanced architectures and real-world applications.

Table 1: Task Objectives for Understanding Artificial Neural Networks (ANNs)

3. Task Implementation

3.1 Perceptron and Multilayer Perceptron

The perceptron is the simplest form of an artificial neural network, introduced by **Frank Rosenblatt** in 1958. Think of it as a single decision-making unit that takes several inputs, applies weights to them, adds them up, and then decides whether to “fire” or not based on a threshold.

Mathematically, it works like this:

1. Each input x_i is multiplied by a weightw_i.
2. All these weighted inputs are summed, along with a bias term$b$.
3. The result is passed through an **activation function**—for the original perceptron, this was a simple step function that outputs 1 if the sum is above a threshold, and 0 otherwise.

In formula form:

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

where f is the activation function.

A perceptron is great for problems where the data can be separated by a straight line (in two dimensions) or a flat plane (in higher dimensions). However, it has a big limitation—it cannot solve problems where the classes are not linearly separable.

To overcome the perceptron’s limitations, researchers introduced the **Multilayer Perceptron (MLP)**. This model stacks multiple layers of neurons as shown in Figure 1:

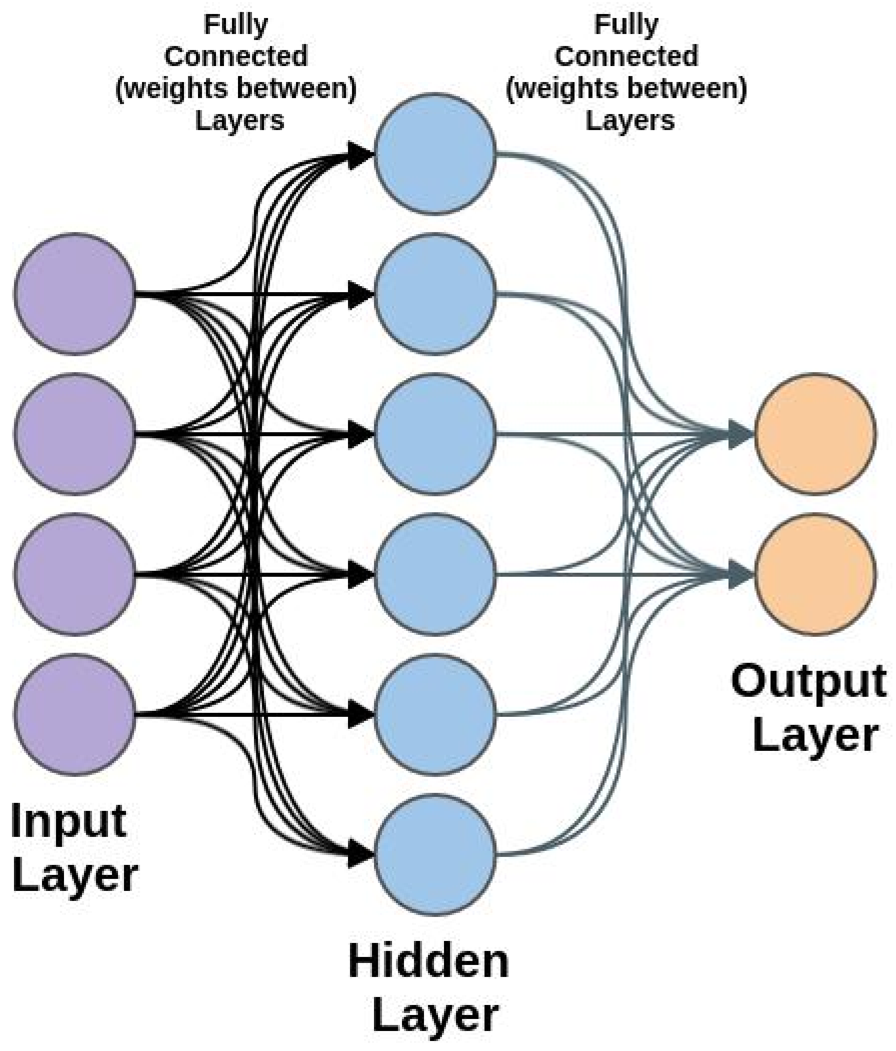
Fully Connected (weights between) Layers
Fully Connected (weights between) Layers
Input Layer
Hidden Layer
Output Layer

Figure 1 : Multilayer Perceptron

- **Input layer** – receives the raw data.
- **Hidden layers** – transform the inputs through intermediate computations.
- **Output layer** – produces the final prediction or decision.

In an MLP, neurons in one layer connect to *all* neurons in the next layer—this is called a **fully connected** architecture. The presence of one or more hidden layers allows the network to model **nonlinear relationships**, meaning it can solve far more complex problems than a single perceptron ever could.

The key innovation that made MLPs trainable was the **backpropagation algorithm**. Instead of just adjusting weights randomly or heuristically, backpropagation calculates how much each weight contributed to the error and updates them accordingly. This makes learning efficient, even for deep networks with many layers.

The MLP is the foundation of modern neural networks. Even though today's models—like convolutional networks for images or recurrent networks for sequences—have specialized architectures, they all follow the same core principles that the MLP introduced:

- **Layered structure** of neurons.
- **Weighted connections** adjusted through learning.
- **Activation functions** to introduce nonlinearity.
- **Loss functions** to guide optimization.

Understanding the perceptron and the MLP is like learning the alphabet before writing a novel—once you grasp these basics, you can understand almost any neural network architecture.

3.2 Forward and Backward Propagation

Once we have an MLP in place, the next big question is: **How does it actually learn?** This is where the ideas of **forward propagation** and **backward propagation** come in. You can think of them as the “two halves” of the learning process—first we make a guess, then we figure out how wrong we were and fix it.

3.2.1 Forward Propagation – Making a Prediction

Forward propagation is the “flow” of information from the input layer to the output layer. It’s like a relay race where data passes through each neuron, layer by layer, until we get a final answer.

Here’s what happens step-by-step:

1. **Inputs enter the network** – Suppose we are classifying handwritten digits. The pixel values of the image become the inputs to the first layer.
2. **Weighted sums are computed** – Each input is multiplied by its weight, all the products are added together, and a bias term is included.
3. **Activation functions are applied** – This adds nonlinearity, helping the network learn complex patterns.
4. **The output moves to the next layer** – This repeats until the output layer produces a final result, like “7” or “cat” or a probability score.

At this stage, the network hasn’t “learned” anything new—it’s just making a prediction with the current weights.

3.2.2 Backward Propagation – Learning from Mistakes

Once the network makes a prediction, we compare it to the correct answer using a **loss function**. The loss tells us *how far off* the prediction was. Backward propagation is about figuring out which weights caused the error and how to adjust them.

The process works like this:

1. **Calculate the loss** – For example, if the network predicted 0.8 for “cat” when the correct answer was 1.0, the loss measures that difference.
2. **Send the error backwards** – We start from the output layer and work our way

back, calculating how much each neuron contributed to the error.

3. **Update the weights** – Using a method called **gradient descent**, each weight is nudged in the direction that reduces the loss. The size of the nudge is controlled by the **learning rate**—too big, and we overshoot; too small, and learning is slow.

3.2.3 Training process

Forward propagation is like *guessing*; backward propagation is like *learning from the guess*.
This forward-backward cycle happens many times—often thousands or millions—during training. Each cycle makes the network a little better at its task, until its predictions are as accurate as possible given the data.

The magic of ANNs lies in this loop:

1. **Guess** (forward)
2. **Check** (loss)
3. **Fix** (backward)

Repeat enough times, and a network that once made random guesses can end up beating humans at recognizing images or understanding speech.

3.3 Activation Functions

Activation functions are at the heart of neural networks, giving them the ability to model complex, nonlinear relationships. Without them, no matter how many layers we stack, the network would behave like a single large linear model—incapable of capturing the intricate patterns found in real-world data. An activation function transforms the weighted sum of inputs into an output signal for the next layer. The choice of this transformation is critical: it influences not only how well the network learns but also how quickly and stably it converges. In this section, we will explore six widely used activation functions, their mathematical forms, and their strengths and weaknesses.

3.3.1 Sigmoid Function

The sigmoid function is one of the earliest and most widely known activation functions. It is a smooth, continuous curve that maps any input into a value between 0 and 1, which makes it particularly suitable for modeling probabilities. It has an S-shaped curve and provides a gradual transition, which is helpful for differentiable learning. Sigmoid was widely used in early neural networks for tasks such as binary classification, where outputs represent probabilities of two possible classes. However, the function's outputs saturate for very large positive or negative inputs, causing very small gradients and slowing down learning.

The mathematical form of the sigmoid function is given by:

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

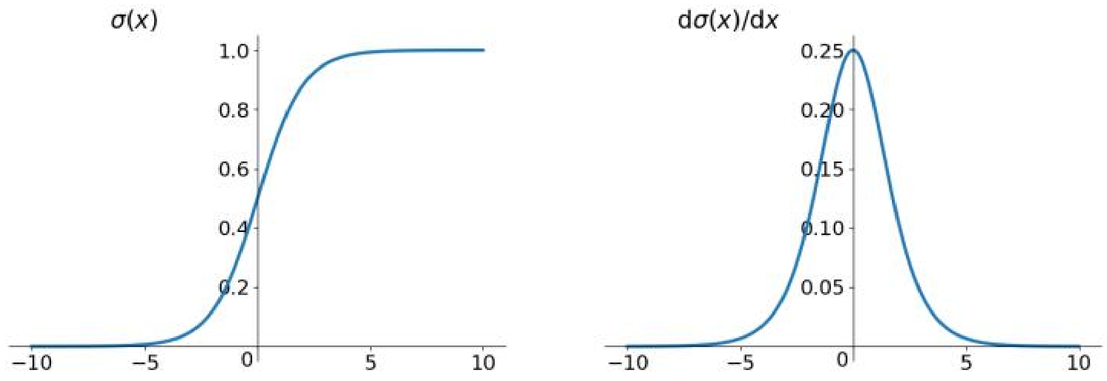

Figure 2: Sigmoid function – smooth S-shaped curve mapping inputs to the range (0,1).

Its smoothness and differentiability made it popular in early neural networks, especially for binary classification tasks, as its output can be interpreted as a probability. However, for very large positive or negative inputs, the gradient becomes extremely small—a problem known as the **vanishing gradient**. When this happens, learning slows dramatically. To address this, researchers turned to a similar function with a wider output range: the hyperbolic tangent.

3.3.2 Hyperbolic Tangent (tanh)

The hyperbolic tangent, or tanh function, is a scaled version of the sigmoid. It maps inputs to a range between -1 and 1, which centers the data around zero. This zero-centering helps gradients propagate more naturally, improving learning dynamics. Like sigmoid, tanh is smooth and differentiable, making it suitable for gradient-based optimization. The function provides a stronger response than sigmoid for inputs around zero, which often accelerates convergence. Despite these advantages, tanh still suffers from vanishing gradients for large inputs, which motivated the development of more robust alternatives like ReLU.

The mathematical form of the tanh function is given by:

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

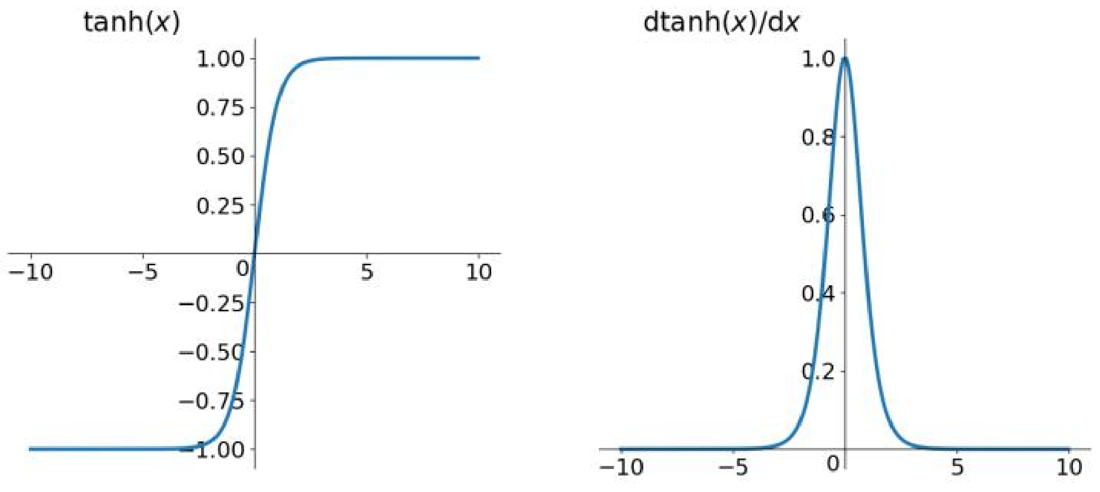

Figure 3: Tanh function – outputs range from -1 to 1, centered at zero.

3.3.3 Rectified Linear Unit (ReLU)

The Rectified Linear Unit, or ReLU, is now the standard for hidden layers in most modern networks. Unlike sigmoid and tanh, ReLU introduces nonlinearity without saturating in the positive domain. It outputs zero for negative inputs and passes positive inputs unchanged. This simple mechanism allows networks to train much faster and avoids the vanishing gradient problem in the positive range. ReLU is computationally efficient and often leads to better performance on deep architectures. However, neurons can “die” if they consistently receive negative inputs, producing zero gradients and never activating again.

The mathematical form of ReLU is given by:

$$f(x) = \max(0,x)$$

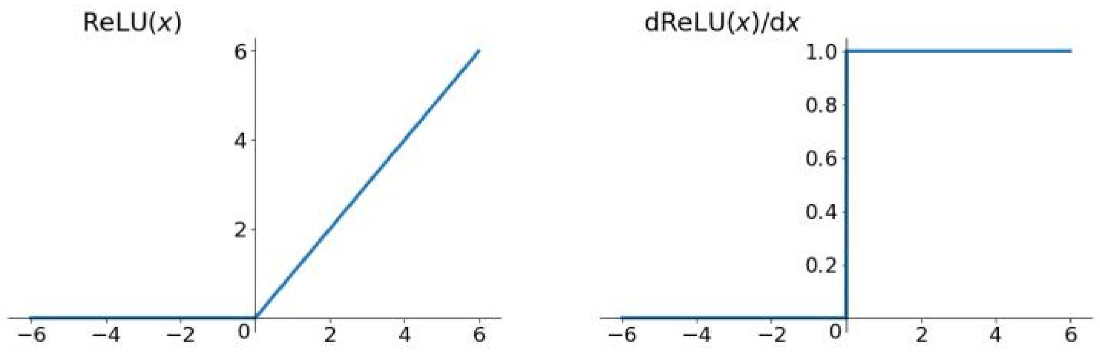

Figure 4: ReLU function – passes positive inputs unchanged, outputs zero for negative inputs.

To overcome this drawback, variants such as Leaky ReLU and Parametric ReLU were introduced, which allow small negative outputs to keep neurons alive.

3.3.4 Leaky ReLU and Parametric ReLU (PReLU)

Leaky ReLU and Parametric ReLU modify the standard ReLU by allowing a small slope for negative inputs. This ensures that neurons continue to learn even when inputs are negative. In Leaky ReLU, the negative slope is a small fixed constant, whereas in Parametric ReLU, the slope becomes a learnable parameter adjusted during training. This flexibility improves network robustness and reduces the risk of dead neurons while preserving the computational advantages of ReLU.

The mathematical form of Leaky ReLU/PReLU is given by:

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases}$$

where α is a small constant in Leaky ReLU or a learnable parameter in PReLU.

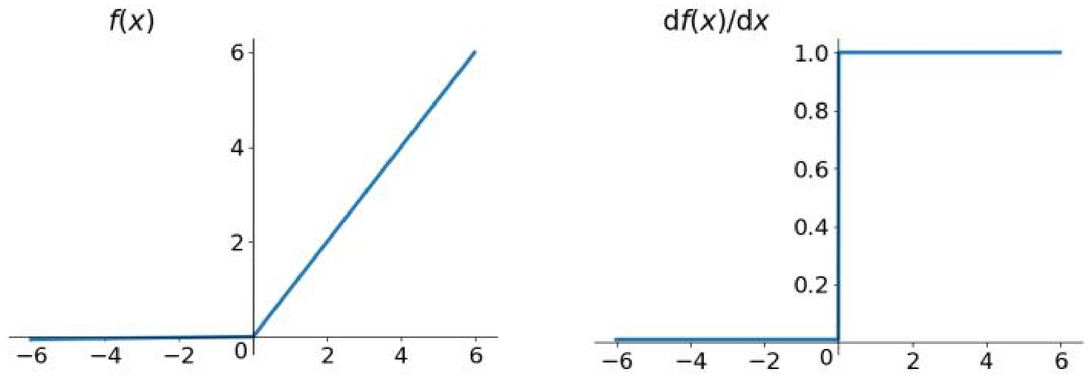

Figure 5: Leaky and Parametric ReLU – allow small negative outputs to prevent dead neurons.

While these functions handle negative inputs better, the slope is either fixed or learned per neuron and may not fully adapt to the range of negative values. To address this, the Exponential Linear Unit was proposed, which smooths the negative domain further.

3.3.5 Exponential Linear Unit (ELU)

ELU improves upon ReLU and its variants by providing a smooth, continuous function for negative inputs. For positive values, it behaves like ReLU, but for negative values, it gradually approaches a negative asymptote. This prevents dead neurons and pushes the outputs closer to zero mean, which can stabilize and accelerate learning. ELU is particularly useful in deeper networks where zero-centered activations help gradient flow. Although computationally slightly more expensive due to the exponential operation, ELU can improve convergence and generalization in many cases.

The mathematical form of ELU is given by:

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha (e^x - 1) & \text{if } x \leq 0 \end{cases}$$

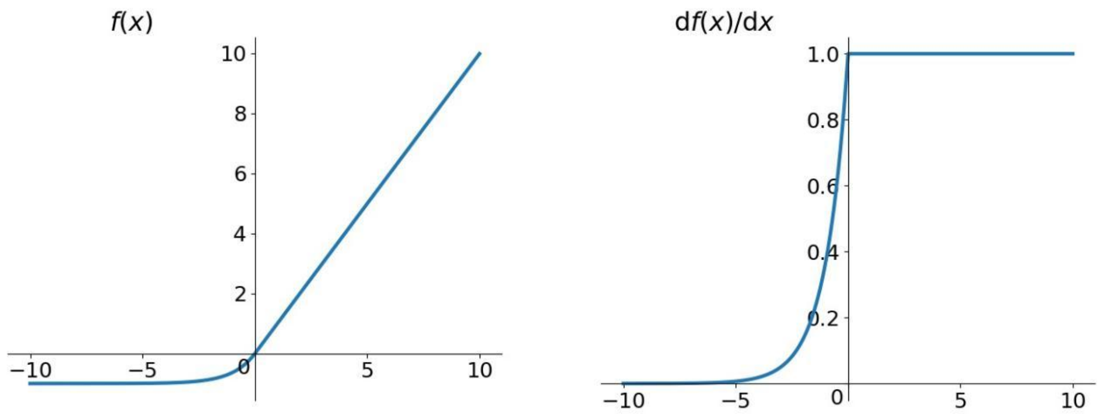

Figure 6: ELU – smooths the transition for negative values and pushes outputs closer to zero mean.

For final outputs in classification tasks, however, we typically use a function that transforms scores into probabilities: the Softmax function.

3.3.6 Softmax Function

The Softmax function is used in the output layer for multi-class classification. It converts raw scores into a probability distribution across all classes, ensuring the sum of probabilities equals one. Softmax not only allows interpretable outputs but also facilitates cross-entropy loss optimization, which is the standard for classification problems. Unlike hidden layer activations, Softmax is typically applied only at the final layer, where each output corresponds to a distinct class.

The mathematical form of Softmax is given by:

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

3.3.7 Choosing the Right Activation Function

For hidden layers, **ReLU** is the default choice due to its simplicity, computational efficiency, and ability to avoid vanishing gradients in positive inputs. If dead neurons are observed, **Leaky ReLU** or **Parametric ReLU** are effective alternatives, providing small negative slopes that keep neurons alive. For networks where zero-centered activations and smoother negative outputs are advantageous, especially in very deep architectures, **ELU** can improve convergence and training stability.

For output layers, the selection depends on the task type. **Softmax** is preferred for multi-class classification problems, while **sigmoid** is suitable for binary classification. Regression tasks often omit activation in the output layer to allow unrestricted continuous values. The choice of activation function, while sometimes guided by conventions, often benefits from careful experimentation to match the dataset and model architecture.

3.4 Loss and Cost Functions

In a neural network, the **loss function** is a crucial component that measures how well the model is performing. It quantifies the difference between the predicted outputs and the actual target values, providing a guide for adjusting the network's parameters during training. Closely related is the **cost function**, which is typically the average loss over all training examples. While "loss" often refers to an individual prediction error, the cost function represents the overall model error, which the optimization algorithms aim to minimize. Proper choice of a loss function is critical, as it directly affects the speed of learning, convergence stability, and ultimately, the performance of the model.

Mathematically, if y_i is the true output and $\hat{y}_i$ is the predicted output for the i-th sample, the **loss** for that sample is given by $L(y_i, \hat{y}_i)$, and the **cost function** $J$over$N$samples is generally defined as:

$$J(\theta) = \frac{1}{N} \sum_{i=1}^{N} L(y_i, \hat{y}_i)$$

whereθrepresents all model parameters. The cost function provides a single value representing the model's overall error, which the training process seeks to minimize using optimization algorithms like gradient descent.

Different tasks—regression or classification—require different types of loss functions, tailored to the nature of the outputs and the desired optimization behavior.

3.4.1 Loss Functions for Regression

Regression tasks aim to predict continuous values. Common loss functions in regression include **Mean Squared Error (MSE)**, **Mean Absolute Error (MAE)**, **Root Mean Squared Error (RMSE)**, and **Huber Loss**. Each has its characteristics and use cases.

3.4.1.1 Mean Squared Error (MSE)

Mean Squared Error measures the average of the squared differences between predicted and true values. By squaring the errors, larger discrepancies are penalized more heavily, which can help the model prioritize reducing significant mistakes. MSE is widely used because it is differentiable and works well with gradient-based optimization. However, it is sensitive to outliers since large errors are amplified by squaring.

The mathematical form of MSE is given by:

\text{MSE} = \frac{1}{N} \sum_{i = 1}^{N} (\hat{y}_i − y_i)^2

3.4.1.2 Mean Absolute Error (MAE)

Mean Absolute Error calculates the average absolute difference between predictions and true values. Unlike MSE, MAE treats all errors linearly, making it less sensitive to outliers. It provides a more robust measure when the dataset contains extreme values. However, the gradient of MAE is constant (or undefined at zero), which can make optimization slower compared to MSE.

The mathematical form of MAE is given by:

$$\text{MAE} = \frac{1}{N} \sum_{i=1}^{N} |\hat{y}_i - y_i|$$

3.4.1.3 Root Mean Squared Error (RMSE)

Root Mean Squared Error is simply the square root of MSE. It retains the property of penalizing large errors more than small ones but returns results in the same units as the original output, making interpretation easier. RMSE is often preferred when it is important to understand errors in terms of the same scale as the predicted quantity.

The mathematical form of RMSE is given by:

$$\text{RMSE} = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (\hat{y}_i - y_i)^2}$$

3.4.1.4 Huber Loss

Huber Loss combines the advantages of MAE and MSE. For small errors, it behaves like MSE, providing smooth gradients and fast convergence. For large errors, it behaves like MAE, reducing sensitivity to outliers. This makes Huber Loss robust in situations where the dataset contains occasional extreme values but still benefits from smooth optimization for smaller errors. The transition between the two regimes is controlled by a parameter δ.

The mathematical form of Huber Loss is given by:

$$L_\delta(y, \hat{y}) = \begin{cases} \frac{1}{2} (\hat{y} - y)^2 & \text{for } |\hat{y} - y| \leq \delta \\ \delta \cdot (|\hat{y} - y| - \frac{1}{2} \delta) & \text{otherwise} \end{cases}$$

3.4.2 Loss Functions for Classification

Classification tasks require predicting discrete class labels. Loss functions in classification evaluate how well the predicted probabilities match the true class labels. Common examples include **Binary Cross-Entropy**, **Categorical Cross-Entropy**, and **Sparse Categorical Cross-Entropy**.

3.4.2.1 Binary Cross-Entropy (BCE)

Binary Cross-Entropy is used for binary classification problems, where each output is either 0 or 1. It measures the dissimilarity between predicted probabilities and actual class labels. BCE heavily penalizes confident but incorrect predictions, encouraging the network to produce probabilities that reflect the true likelihood of each class.

The mathematical form of BCE is given by:

$$L = -\frac{1}{N} \sum_{i=1}^{N} \big[y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i) \big]$$

3.4.2.2 Categorical Cross-Entropy (CCE)

Categorical Cross-Entropy generalizes BCE for multi-class classification where the target label is one-hot encoded. It calculates the negative log-likelihood of the predicted probability assigned to the correct class. CCE encourages the network to assign higher probabilities to the correct class while penalizing errors on other classes.

The mathematical form of CCE is given by:

$$L = -\sum_{i=1}^{N} \sum_{c=1}^{C} y_{i,c} \log(\hat{y}_{i,c})$$

where C is the number of classes, and $y_{i,c}$ is 1 if sample i belongs to class c and 0 otherwise.

3.4.2.3 Sparse Categorical Cross-Entropy

Sparse Categorical Cross-Entropy is a variant of CCE that is computationally more efficient when class labels are integers rather than one-hot encoded vectors. The underlying principle is identical to CCE, but it reduces memory overhead and simplifies data representation.

The mathematical form of Sparse Categorical Cross-Entropy is given by:

$$L = -\sum_{i=1}^{N} \log(\hat{y}_{i,y_i})$$

where $y_i$is the integer label of the correct class for sample i.

3.4.2.4 Choosing the Right Loss Function

For regression tasks, the choice of loss function depends on sensitivity to outliers and interpretability. **MSE** or **RMSE** is appropriate when large errors must be penalized strongly. **MAE** is preferable when the dataset contains outliers. **Huber Loss** strikes a balance, providing robustness and smooth gradients.

For classification tasks, **Binary Cross-Entropy** is ideal for two-class problems, whereas **Categorical Cross-Entropy** is standard for multi-class outputs with one-hot encoding. **Sparse Categorical Cross-Entropy** is useful for multi-class tasks with integer labels, offering computational efficiency without compromising accuracy. Careful selection of the loss function ensures stable training and better generalization of the neural network.