Artificial Neural Networks And Deep Learning

Theory

Christian Rossi

Academic Year 2024-2025

Abstract

Neural networks have matured into flexible and powerful non-linear data-driven models, effectively tackling complex tasks in both science and engineering. The emergence of deep learning, which utilizes neural networks to learn optimal data representations alongside their corresponding models, has significantly advanced this paradigm.

In the course, we will explore various topics in depth. We will begin with the evolution from the Perceptron to modern neural networks, focusing on the feedforward architecture. The training of neural networks through backpropagation and algorithms like Adagrad and Adam will be covered, along with best practices to prevent overfitting, including cross-validation, stopping criteria, weight decay, dropout, and data resampling techniques.

The course will also delve into specific applications such as image classification using neural networks, and we will examine recurrent neural networks and related architectures like sparse neural autoencoders. Key theoretical concepts will be discussed, including the role of neural networks as universal approximation tools, and challenges like vanishing and exploding gradients.

We will introduce the deep learning paradigm, highlighting its distinctions from traditional machine learning methods. The architecture and breakthroughs of convolutional neural networks (CNNs) will be a focal point, including their training processes and data augmentation strategies.

Furthermore, we will cover structural learning and long-short term memory (LSTM) networks, exploring their applications in text and speech processing. Topics such as autoencoders, data embedding techniques like word2vec, and variational autoencoders will also be addressed.

Finally, we will discuss transfer learning with pre-trained deep models, examine extended models such as fully convolutional CNNs for image segmentation (e.g., U-Net) and object detection methods (e.g., R-CNN, YOLO), and explore generative models like generative adversarial networks (GANs).

Contents

CHAPTER 1

Deep learning

1.1 Introduction

Definition (*Machine Learning*)**.** A computer program is considered to learn from experience E with respect to a specific class of tasks T and a performance measure P if its performance at tasks in T, as measured by P, improves with experience E.

Given a dataset $\mathcal{D} = x_1, x_2, \ldots, x_N$, Machine Learning can be broadly categorized into three types:

- *Supervised learning*: in this type of learning, the model is provided with desired outputs $t_1, t_2, \ldots, t_N$ and learns to produce the correct output for new input data. The primary tasks in supervised learning are:
  - *Classification*: the model is trained on a labeled dataset and returns a label for new data.
  - *Regression*: the model is trained on a dataset with numerical values and returns a number as the output.
- *Unsupervised learning*: here, the model identifies patterns and regularities within the dataset $\mathcal{D}$ without being provided with explicit labels. The main task in unsupervised learning is:
  - *Clustering*: the model groups similar data elements based on inherent similarities within the dataset.
- *Reinforcement learning*: in this approach, the model interacts with the environment by performing actions $a_1, a_2, \ldots, a_N$ and receives rewards $r_1, r_2, \ldots, r_N$ in return. The model learns to maximize cumulative rewards over time by adjusting its actions.

1.2 Deep Learning

Deep Learning, a subset of Machine Learning, focuses on utilizing large datasets and substantial computational power to automatically learn data representations. In certain cases, traditional classification may fail due to the presence of irrelevant or redundant features in the dataset.

Deep Learning addresses this issue by learning optimal features directly from the data, which are then used by Machine Learning algorithms to perform more accurate classifications. Essentially, Deep Learning involves learning how to represent data in a way that improves the performance of Machine Learning models.

1.3 Perceptron

In the 1940s, computers were already proficient at executing tasks exactly as programmed and performing arithmetic operations with impressive speed. However, researchers envisioned machines that could do much more. They wanted computers that could handle noisy data, interact directly with their environment, function in a massively parallel and fault-tolerant way, and adapt to changing circumstances. Their quest was for a new computational model, one that could surpass the constraints of the Von Neumann Machine.

1.3.1 Human neurons

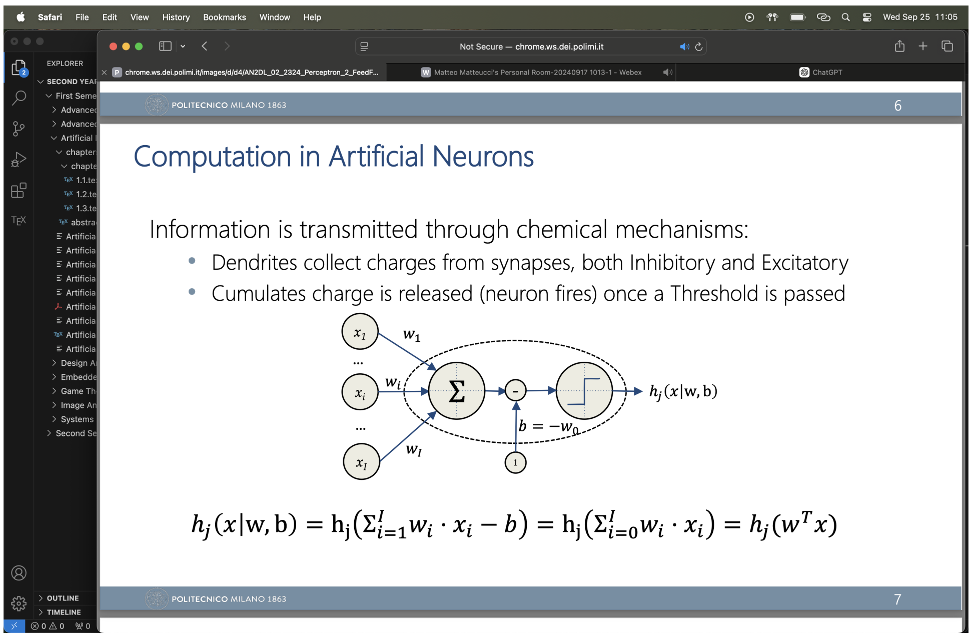
The human brain contains an enormous number of computing units, with approximately 100 billion neurons, each connected to around 7,000 other neurons through synapses. In adults, this results in a total of 100 to 500 trillion synaptic connections, while in a three-year-old child, this number can reach up to 1 quadrillion synapses.

The brain's computational model is characterized by its distributed nature among simple, non-linear units, its redundancy which ensures fault tolerance, and its intrinsic parallelism. The perceptron, a computational model inspired by the brain, reflects these principles.

Information in the brain is transmitted through chemical processes. Dendrites gather signals from synapses, which can be either inhibitory or excitatory. When the cumulative charge reaches a certain threshold, the neuron fires, releasing the charge.

1.3.2 Artificial neuron

The mathematical model of a neuron is represented as follows:



Computation in Artificial Neurons

Information is transmitted through chemical mechanisms:

- Dendrites collect charges from synapses, both Inhibitory and Excitatory
- Cumulates charge is released (neuron fires) once a Threshold is passed

$$h_j(x|\mathrm{w}, \mathrm{b}) = \mathrm{h_j}(\Sigma_{i=1}^{I} w_i \cdot x_i - b) = \mathrm{h_j}(\Sigma_{i=0}^{I} w_i \cdot x_i) = h_j(w^T x)$$

Figure 1.1: Artificial neuron

In this model, the output function $h_j(\mathbf{x}|\mathbf{w}, b)$ is defined as:

$$h_j(\mathbf{x}|\mathbf{w}, b) = h_j\left(\sum_{i=1}^{I} w_i x_i - b\right) = h_j\left(\sum_{i=0}^{I} w_i x_i\right) = h_j\left(\mathbf{w}^T \mathbf{x}\right)$$

The function used in an artificial neuron can either be a step function, with values ranging from 0 to 1, or a sine function, with values ranging from -1 to 1.

History Several researchers were actively investigating models for the brain during the mid-20th century. In 1943, Warren McCulloch and Walter Pitts proposed the Threshold Logic Unit, also referred to as the Linear Unit, where the activation function was a threshold unit, equivalent to the Heaviside step function. A few years later, in 1957, Frank Rosenblatt developed the first Perceptron, with weights encoded in potentiometers, and weight adjustments during learning were performed by electric motors. By 1960, Bernard Widrow introduced a significant advancement by representing the threshold value as a bias term in the ADALINE (Adaptive Linear Neuron or later, Adaptive Linear Element).

Example:
Consider a neuron designed to implement the OR operation:

x_0	x_1	x_2	**OR**
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

The corresponding neuron is illustrated below:

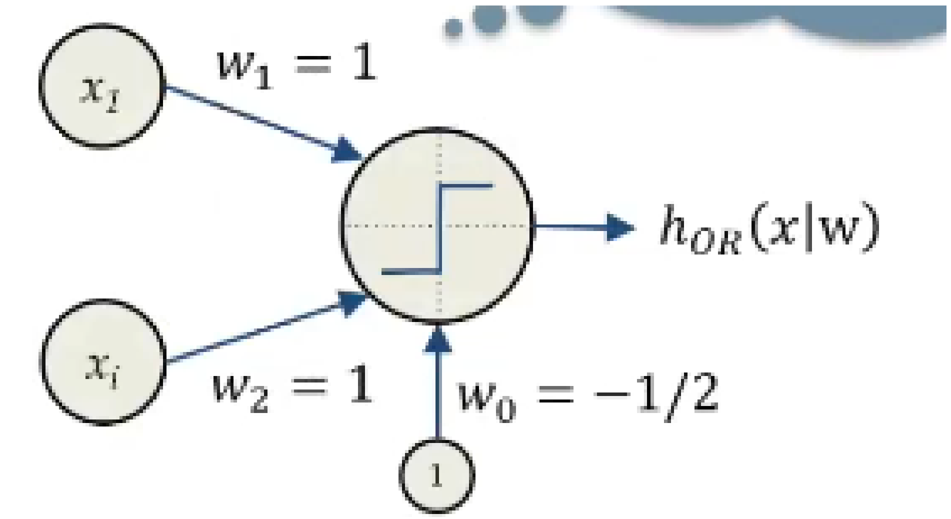


Figure 1.2: OR artificial neuron

The output function for this neuron is defined as:

$$h_{\mathrm{OR}}(w_0 + w_1x_1 + w_2x_2) = h_{\mathrm{OR}}\left(-\frac{1}{2} + x_1 + x_2\right) = \begin{cases} 1 & \text{if } \left(-\frac{1}{2} + x_1 + x_2\right) > 0 \\ 0 & \text{otherwise} \end{cases}$$

Now, consider a neuron designed to implement the AND operation:

x_0	x_1	x_2	**AND**
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

The corresponding neuron is shown below:

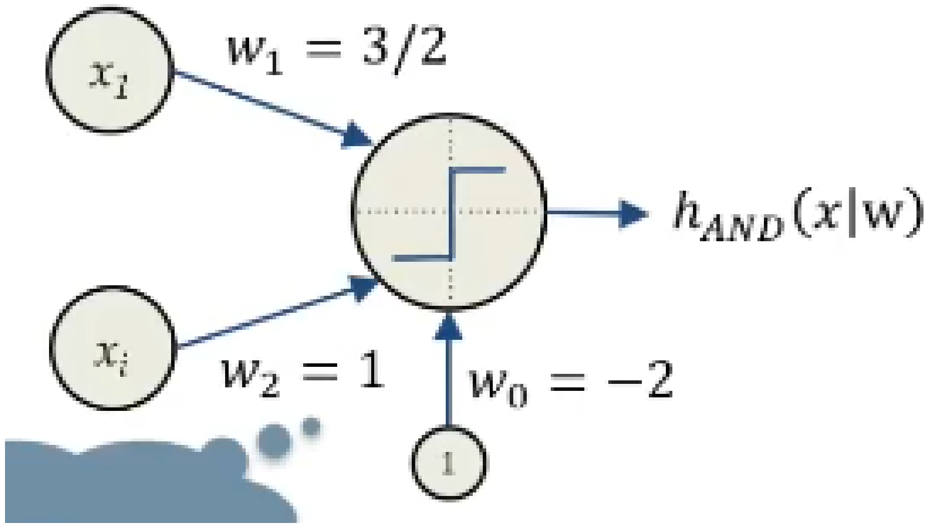


Figure 1.3: AND artificial neuron

The output function for this neuron is given by:

$$h_{\text{AND}}(w_0 + w_1x_1 + w_2x_2) = h_{\text{AND}}\left(-2 + \frac{3}{2}x_1 + x_2\right) = \begin{cases} 1 & \text{if } \left(-2 + \frac{3}{2}x_1 + x_2\right) > 0 \\ 0 & \text{otherwise} \end{cases}$$

1.3.3 Hebbian learning

The strength of a synapse increases based on the simultaneous activation of the corresponding input and the desired target. Hebbian learning can be summarized as follows:

1. Begin with a random initialization of the weights.
2. Adjust the weights for each sample individually (online learning), and only when the sample is not correctly predicted.

Mathematically, this is expressed as:

$$\begin{cases} w_i^{k+1} = w_i^k + \Delta w_i^k \\ \Delta w_i^k = \eta x_i^k t^k \end{cases} \implies w_i^{k+1} = w_i^k + \eta x_i^k t^k$$

Here, η represents the leraning rate, x_i^k is the i-th input to the perceptron at time k, and t^k is the desired output at time k.

Example:
We aim to learn the weights necessary to implement the OR operator with a sinusoidal output. The modified OR truth table is as follows:

x_0	x_1	x_2	**OR**
1	-1	-1	-1
1	-1	1	1
1	1	-1	1
1	1	1	1

We begin with random weights:

$$\mathbf{w} = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}$$

The learning rate is set to $\eta = \frac{1}{2}$. The output function is defined as:

$$h(\mathbf{w}^T\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w}^T\mathbf{x} > 0 \\ 0 & \text{if } \mathbf{w}^T\mathbf{x} = 0 \\ -1 & \text{if } \mathbf{w}^T\mathbf{x} < 0 \end{cases}$$

The training involves iterating through the data records and adjusting the weights for incorrectly classified samples until all records are correctly predicted.

Starting from the first row, we have:

$$y_{\text{first row}} = x_0w_0 + x_1w_1 + x_2w_2 = 1 \cdot 0 + (-1) \cdot 0 + (-1) \cdot 0 = 0$$

This does not match the expected output of -1. We adjust the weights:

$$w_0^{\text{new}} = w_0 + \eta x_0 y = 0 + \frac{1}{2} \cdot 1 \cdot (-1) = -\frac{1}{2}$$

$$w_1^{\text{new}} = w_1 + \eta x_1 y = 0 + \frac{1}{2} \cdot (-1) \cdot (-1) = \frac{1}{2}$$

$$w_2^{\text{new}} = w_2 + \eta x_2 y = 0 + \frac{1}{2} \cdot (-1) \cdot (-1) = \frac{1}{2}$$

Now, the weights vector is:

$$\mathbf{w} = \begin{bmatrix} -\frac{1}{2} & \frac{1}{2} & \frac{1}{2} \end{bmatrix}$$

For the second row, we have:

$$y_{\text{second row}} x_0 w_0 + x_1 w_1 + x_2 w_2 = 1 \cdot \left(-\frac{1}{2}\right) + (-1) \cdot \frac{1}{2} + 1 \cdot \frac{1}{2} = -\frac{1}{2}$$

This does not match the expected output of 1. We adjust the weights:

$$w_0^{\text{new}} = w_0 + \eta x_0 y = \left(-\frac{1}{2}\right) + \frac{1}{2} \cdot 1 \cdot 1 = 0$$

$$w_1^{\text{new}} = w_1 + \eta x_1 y = \frac{1}{2} + \frac{1}{2} \cdot (-1) \cdot 1 = 0$$

$$w_2^{\text{new}} = w_2 + \eta x_2 y = \frac{1}{2} + \frac{1}{2} \cdot 1 \cdot 1 = 1$$

Now, the weights vector is:

$$\mathbf{w} = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$$

For the third row, we have:

$$y_{\text{third row}} x_0 w_0 + x_1 w_1 + x_2 w_2 = 1 \cdot 0 + 1 \cdot 0 + (-1) \cdot 1 = -1$$

This does not match the expected output of 1. We adjust the weights:

$$w_0^{\text{new}} = w_0 + \eta x_0 y = 0 + \frac{1}{2} \cdot 1 \cdot 1 = \frac{1}{2}$$

$$w_1^{\text{new}} = w_1 + \eta x_1 y = 0 + \frac{1}{2} \cdot 1 \cdot 1 = \frac{1}{2}$$

$$w_2^{\text{new}} = w_2 + \eta x_2 y = 1 + \frac{1}{2} \cdot (-1) \cdot 1 = \frac{1}{2}$$

Now, the weights vector is:

$$\mathbf{w} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \end{bmatrix}$$

For the third row, we have:

$$y_{\text{fourth row}} x_0 w_0 + x_1 w_1 + x_2 w_2 = 1 \cdot \frac{1}{2} + 1 \cdot \frac{1}{2} + 1 \cdot \frac{1}{2} = \frac{3}{2}$$

This matches the expected output of 1.

After verifying the outputs for all rows, we recognize that further iterations (epochs) are needed for full convergence. We repeat the training until all records produce the desired outputs.

After multiple epochs, the final weights vector is:

$$\mathbf{w} = \begin{bmatrix} -\frac{1}{2} & 1 & 1 \end{bmatrix}$$

The number of epochs required depends on both the initialization of the weights and the order in which the data is presented.

A perceptron computes a weighted sum and returns the sign (thresholding) of the result:

$$h_j(\mathbf{x}|\mathbf{w}) = h_j\left(\sum_{i=0}^{I} w_i x_i\right) = \text{Sign}(w_0 + w_1 x_1 + \cdots + w_I x_I)$$

This forms a linear classifier, where the decision boundary is represented by the hyperplane:

$$w_0 + w_1 x_1 + \cdots + w_I x_I = 0$$

The linear boundary explains how the perceptron implements Boolean operators. However, if the dataset does not have a linearly separable boundary, the perceptron fails to work. In such cases, alternative approaches are needed, including non-linear boundaries or different input representations. This concept forms the basis for Multi-Layer Perceptrons (MLPs).

CHAPTER 2

Feed Forward Neural Networks

2.1 Introduction

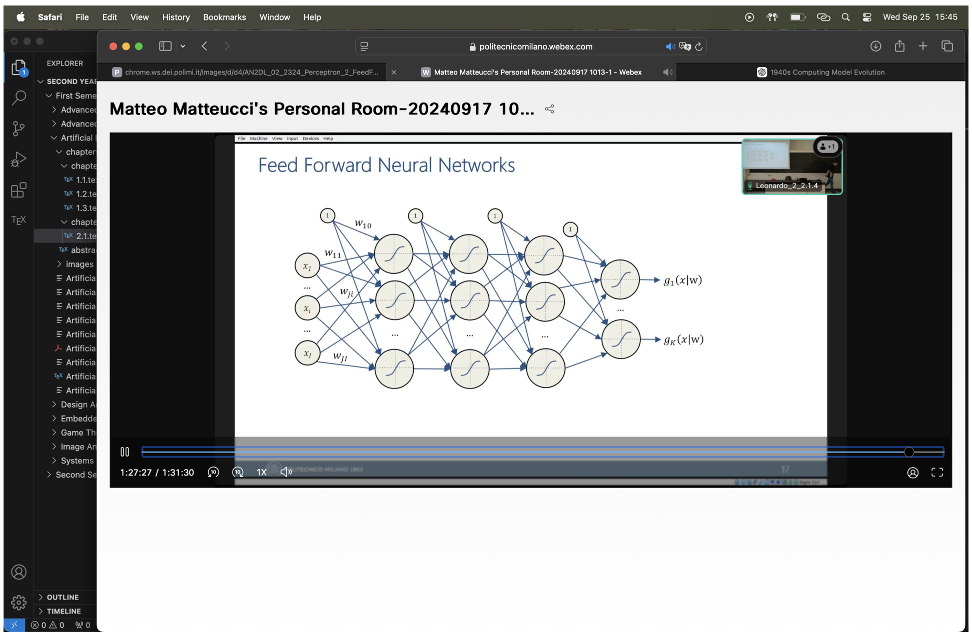


Figure 2.1: Multi-Layer Perceptron architecture

A Multi-Layer Perceptron (MLP) is a type of feedforward neural network (FFNN) that consists of multiple layers of nodes, each layer being fully connected to the next. The MLP architecture is composed of three primary components:

- *Input layer*: this layer receives the input data and passes it to the subsequent layers. The size of this layer depends on the specific problem and the input features.
- *Hidden layers*: these intermediate layers perform the actual computations, transforming the input into more abstract representations. The number of hidden layers and the number of neurons in each hidden layer are determined through hyperparameter tuning, which is often based on a trial-and-error approach.

- *Output layer*: this layer produces the final output of the network, which could be a prediction, classification, or another result. Its size depends on the nature of the problem, such as the number of classes in classification tasks.

The MLP is inherently a non-linear model, characterized by the number of neurons in each layer, the choice of activation functions, and the values of the connection weights. The connections between layers are represented by weight matrices, denoted as $W^{(l)} = \{w_{ij}^{(l)}\}$, where l is the layer index. If a layer has J nodes and receives I inputs, the corresponding weight matrix has dimensions $J \times (I+1)$ accounting for the bias term.

The output of each neuron depends solely on the outputs from the previous layer, allowing for forward propagation of information. The learning of weights in an FFNN is achieved through a technique known as backpropagation, which iteratively adjusts the weights to minimize the error in the network's predictions.

2.2 Activation functions

Activation functions play a critical role in neural networks by introducing non-linearity into the model. Common activation functions include:

- *Linear*: $g(a) = a$ with a derivative of $g'(a) = 1$.
- *Sigmoid*: $g(a) = \dfrac{1}{1+e^{-a}}$ with a derivative of $g(a)(1-g(a))$. This function is widely used due to its simplicity and ability to model probabilities.
- *Hyperbolic tangent* (Tanh): $g(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}$ with a derivative of $1 - g(a)^2$. This function is often preferred for hidden layers as it outputs values in the range $[-1, 1]$, centering the data.

The choice of the activation function is a design decision, influenced by the nature of the task and the structure of the network.

2.2.1 Output layer

The activation function for the output layer depends on the type of problem being addressed:

- *Regression*: in regression tasks, where the output spans the real number domain $\mathbb{R}$, a linear activation function is typically used for the output neuron.
- *Binary classification*: the choice of activation depends on the coding of the class labels:
    1. For classes coded as $\Omega_0 = -1$, $\Omega_1 = 1$, a Tanh activation function is appropriate.
    2. For classes coded as $\Omega_0 = 0$, $\Omega_1 = 1$, a Sigmoid activation function is commonly used, as it can be interpreted as representing the posterior probability of a class.
- *Multi-class classification*: for problems with K classes, the output layer contains K neurons, one for each class. The classes are typically encoded using one-hot encoding, e.g., $\Omega_0 = \begin{bmatrix}0 & 0 & 1\end{bmatrix}$, $\Omega_1 = \begin{bmatrix}0 & 1 & 0\end{bmatrix}$, and $\Omega_2 = \begin{bmatrix}1 & 0 & 0\end{bmatrix}$. The output neurons utilize a softmax activation function:

$$y_k = \frac{e^{z_k}}{\sum_k e^{z_k}} = \frac{e^{\sum_j w_{kj} h_j (\sum_i^I w_{ji} x_i)}}{\sum_{k=1}^K e^{\sum_j w_{kj} h_j (\sum_i^I w_{ji} x_i)}}$$

Here, z_k is the activation value of the k-th output neuron. The softmax function normalizes the output vector, providing class probabilities.

2.2.2 Hidden layers

For the hidden layers, activation functions such as the Sigmoid or Tanh are commonly used. These functions introduce non-linearity, allowing the network to model complex patterns in the data.

Theorem 2.2.1. *A single hidden layer feedforward neural network with S-shaped activation functions (such as Sigmoid or Tanh) can approximate any measurable function to any desired degree of accuracy on a compact set.*

This theorem implies that a single hidden layer can theoretically represent any function, though it does not guarantee that the learning algorithm will find the necessary weights. In practice, an excessively large number of hidden units may be required, and the network may struggle to generalize, particularly if overfitting occurs. However, for classification tasks, typically only one additional hidden layer is needed to achieve satisfactory performance.

2.3 Training

Training a neural network involves learning a set of parameters, such as weights $\mathbf{w}$, that allow the model $y(x_n|\mathbf{w})$ approximate the target t_n as closely as possible, given a training set $\mathcal{D} = \{\langle x_1, t_1\rangle, \ldots, \langle x_N, t_N\rangle\}$ we want to find model parameters such that for new data $y_n(x_n|\theta) \sim t_n$. This process can be viewed as finding parameters that generalize well to new data, such that $g(x_n|\mathbf{w}) \sim t_n$.

In regression and classification tasks, this goal is typically achieved by minimizing the error between the predicted outputs and the true labels. For a neural network, the error is often represented as the Sum of Squared Errors (SSE):

$$E(\mathbf{w})_{\text{SSE}} = \sum_{n}^{N} (t_n - g(x_n|\mathbf{w}))^2$$

Here, the SSE represents the error function, and for a feedforward neural network, this error is a non-linear function of the weights, making the optimization process more challenging.

2.3.1 Nonlinear optimization

To minimize a generic error function $J(\mathbf{w})$, we rely on optimization techniques. The goal is to find the weights $\mathbf{w}$ that minimize the error by solving:

$$\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}} = 0$$

However, for neural networks, closed-form solutions are generally not available due to the non-linearity of the model. Instead, we employ iterative methods like gradient descent, which adjusts the weights incrementally in the direction that reduces the error. The steps for gradient descent are as follows:

1. Initialize the weights $\mathbf{w}$ to small random values.

2. Iterate until convergence:

$$w^{k+1} = w^k - \eta \frac{\partial J(w)}{\partial w}\bigg|_{w^k}$$

   Here, η is the learning rate, controlling the step size in each iteration.

In cases where the error function has multiple local minima, the final solution depends on the initial starting point. To address this, we can introduce a momentum term that helps the optimization process avoid being trapped in local minima:

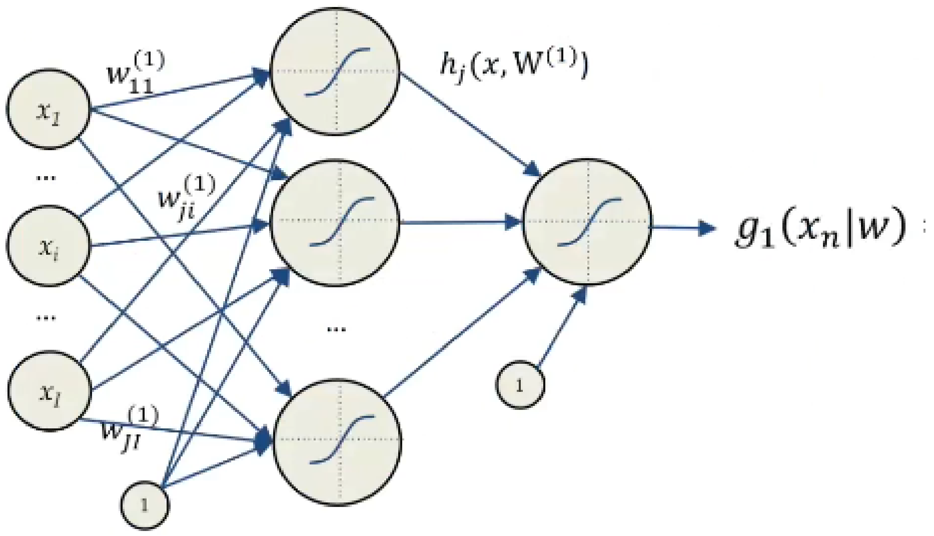
$$w^{k+1} = w^k - \eta \frac{\partial E(w)}{\partial w}\bigg|_{w^k} - \alpha \frac{\partial E(w)}{\partial w}\bigg|_{w^k}$$

Here, α represents the momentum coefficient, which encourages the optimization to keep moving in the same direction, effectively smoothing out oscillations and escaping shallow local minima.

Multiple restarts To improve the likelihood of finding the global minimum, especially in complex non-convex error surfaces, multiple restarts of the optimization from different random initializations can be used. This increases the chances of converging to a better solution by exploring various regions of the parameter space.

2.3.2 Gradient descent

Example:
Consider the following FFNN.



The output of the network is defined as:

$$g_1(x_n|\mathbf{w}) = g_1\left(\sum_{j=0}^{J} w_{1,j}^{(2)} h_j \left(\sum_{i=0}^{I} w_{j,i}^{(1)} x_{i,n}\right)\right)$$

Here, h_j represents the activation function of the hidden neurons, and $w_{i,j}^{(1)}$ and $w_{i,j}^{(2)}$ are the weights of the first and second layers, respectively.

We aim to minimize the sum of squared errors (SSE) between the predicted output and the target values:

$$E(\mathbf{w}) = \sum_{n=1}^{N} (t_n - g_1(x_n|\mathbf{w}))^2$$

Let's compute the weight update for $w_{3,5}^{(1)}$ using gradient descent. This weight corresponds to the connection between the 5th input and the 3rd hidden neuron. After calculating the derivative of the error function with respect to $w_{3,5}^{(1)}$, we obtain the following update rule:

$$\frac{\partial E(\mathbf{w})}{\partial w_{3,5}^{(1)}} = -2 \sum_{n}^{N} (t_n - g_1(x_n, \mathbf{w})) g_1'(x_n, \mathbf{w}) w_{1,3}^{(2)} h_3' \left(\sum_{i=0}^{I} w_{3,1}^{(1)} x_{i,n} \right) x_{5,n}$$

This expression includes the derivative of the output function g_1', the weight $w_{1,3}^{(2)}$ and the derivative of the hidden neuron activation function h_3'

In practice, using all data points for weight updates (i.e., batch gradient descent) can be computationally expensive, especially for large datasets. The gradient of the error function for batch gradient descent is given by:

$$\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} = \frac{1}{N} \sum_{n}^{N} \frac{\partial E(x_n, \mathbf{w})}{\partial \mathbf{w}}$$

However, this can be inefficient, so instead, we can use stochastic gradient descent (SGD), where the gradient is computed using a single sample at each iteration:

$$\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} \approx \frac{\partial E_{\text{SGD}}(\mathbf{w})}{\partial \mathbf{w}} = \frac{\partial E(x_n, \mathbf{w})}{\partial \mathbf{w}}$$

SGD is faster and unbiased but introduces high variance in the updates, which can cause the optimization process to oscillate.

A middle ground between batch gradient descent and SGD is mini-batch gradient descent, which uses a subset of samples (mini-batch) to compute the gradient:

$$\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} \approx \frac{\partial E_{\text{MB}}(\mathbf{w})}{\partial \mathbf{w}} = \frac{1}{M} \sum_{n \in \text{minibatch}}^{M<N} \frac{\partial E(x_n, \mathbf{w})}{\partial \mathbf{w}}$$

This approach provides a good balance between the computation cost and the variance of the updates, allowing for faster convergence while maintaining stability.

2.3.3 Gradient descent computation

The gradient descent process can be computed automatically from the structure of the neural network using backpropagation. This method allows for efficient weight updates that can be performed in parallel and locally, requiring just two passes through the network.

Let x be a real number, and consider two functions $f : \mathbb{R} \to \mathbb{R}$ and g: $\mathbb{R} \to \mathbb{R}$. Now, define the composite function $z = f(g(x)) = f(y)$, where $y = g(x)$. Using the chain rule, the derivative of z with respect to x is:

$$\frac{dz}{dx} = \frac{dz}{dy} = \frac{dy}{dx} = f'(y)g'(x) = f'(g(x))g'(x)$$

This concept extends naturally to backpropagation in neural networks. For example, consider the weight update for the weight $w_{j,i}^{(1)}$. Using the chain rule, we can express the partial derivative of the error function E with respect to $w_{j,i}^{(1)}$ as:

$$\frac{\partial E(w_{j,i}^{(1)})}{\partial w_{j,i}^{(1)}} = \underbrace{-2\sum_{n}^{N}(t_n - g_1(x_n,\mathbf{w}))}_{\frac{\partial E}{\partial g(x_n,\mathbf{w})}} \underbrace{g_1'(x_n,\mathbf{w})}_{\frac{\partial g(x_n,\mathbf{w})}{\partial w_{1,j}^{(2)} h_j(\cdot)}} \underbrace{w_{1,j}^{(2)}}_{\frac{\partial w_{1,j}^{(2)} h_j(\cdot)}{\partial h_j(\cdot)}} \underbrace{h_j'\left(\sum_{i=0}^{I} w_{j,i}^{(1)} x_{i,n}\right)}_{\frac{\partial h_j(\cdot)}{\partial w_{j,i}^{(1)} x_i}} \underbrace{x_i}_{\frac{\partial w_{j,i}^{(1)} x_i}{\partial w_{j,i}^{(1)}}}$$

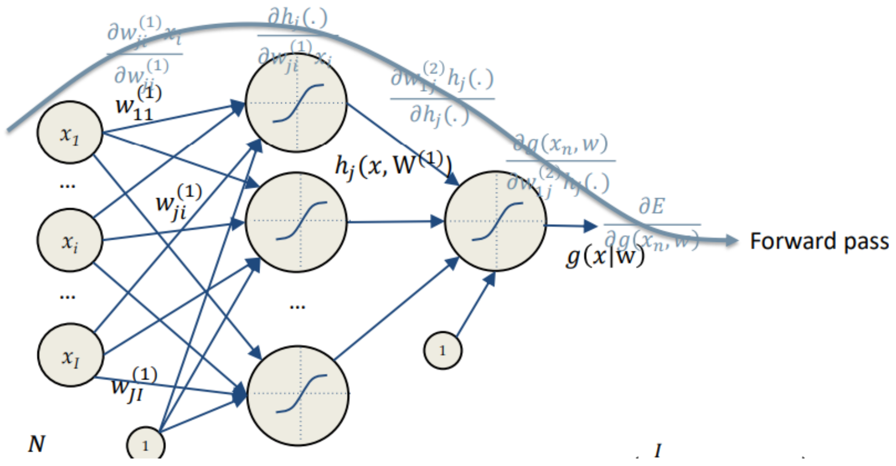
$$= \frac{\partial E}{\partial g(x_n,\mathbf{w})} \cdot \frac{\partial g(x_n,\mathbf{w})}{\partial w_{1,j}^{(2)} h_j(\cdot)} \cdot \frac{\partial w_{1,j}^{(2)} h_j(\cdot)}{\partial h_j(\cdot)} \cdot \frac{\partial h_j(\cdot)}{\partial w_{j,i}^{(1)} x_i} \cdot \frac{\partial w_{j,i}^{(1)} x_i}{\partial w_{j,i}^{(1)}}$$

The gradient descent can be computed efficiently using the forward-backward pass strategy:

1. *Forward pass*: during the forward pass, the input propagates through the network to compute the output of each neuron. The local derivatives for each neuron (dependent only on its immediate inputs) are also computed. These computations do not depend on the other neurons in the network, making it possible to store this information locally.

2. *Backward pass*: in the backward pass, the stored values from the forward pass are used to propagate the gradients back through the network. This involves computing the partial derivatives of the error with respect to each weight and updating them accordingly using the chain rule.

By separating the forward and backward computations, if any part of the network, such as the error function, changes, only the relevant parts need to be recomputed. This flexibility allows for a more efficient calculation of the gradients and weight updates.



$\frac{\partial w_{ji}^{(1)} x_i}{\partial w_{ji}^{(1)}}$
$\frac{\partial h_j(.)}{\partial w_{ji}^{(1)} x_i}$
$\frac{\partial w_{1j}^{(2)} h_j(.)}{\partial h_j(.)}$
$\frac{\partial g(x_n, w)}{\partial w_{1j}^{(2)} h_j(.)}$
$\frac{\partial E}{\partial g(x_n, w)}$
$w_{11}^{(1)}$
x_1
x_i
x_I
$w_{ji}^{(1)}$
$w_{JI}^{(1)}$
$h_j(x, W^{(1)})$
$g(x|w)$
Forward pass
N
1
1
I

Figure 2.2: Forward and backward passes in Neural Networks

This approach allows for a systematic and parallelizable way of calculating the gradient, minimizing the computation needed for each update and ensuring that the network can be trained efficiently.

2.4 Loss function

Consider an independent and identically distributed sample drawn from a Gaussian distribution with a known variance, σ^2. The goal is to estimate the parameter μ using Maximum Likelihood Estimation (MLE), which selects parameters that maximize the probability of the observed data.

Let $\boldsymbol{\theta} = \begin{pmatrix}\theta_1 & \theta_2 & \dots & \theta_p\end{pmatrix}^T$ represent the vector of parameters. The task is to find the MLE of $\boldsymbol{\theta}$. Here is the step-by-step approach:

1. *Construct the likelihood function*: the likelihood function $L(\mu)$ is the probability of the data given μ, assuming a Gaussian distribution. The joint likelihood of the data $x_1, x_2, \dots, x_N$ is:

$$L(\mu) = \Pr(x_1, x_2, \dots, x_N|\mu, \sigma^2) = \prod_{n=1}^{N} \Pr(x_n|\mu, \sigma^2) = \prod_{n=1}^{N} \frac{1}{\sqrt{2\pi}} e^{-\dfrac{(x_n-\mu)^2}{2\sigma^2}}$$

2. *Log-likelihood function*: since the likelihood is a product, it is convenient to take the logarithm to transform it into a sum, yielding the log-likelihood:

$$l(\mu) = \log\left(\prod_{n=1}^{N} \frac{1}{\sqrt{2\pi}} e^{-\dfrac{(x_n-\mu)^2}{2\sigma^2}}\right) = \sum_{n=1}^{N} \log \frac{1}{\sqrt{2\pi}} e^{-\dfrac{(x_n-\mu)^2}{2\sigma^2}}$$

Simplifying:

$$l(\mu) = N \log \frac{1}{\sqrt{2\pi}} - \frac{1}{2\sigma^2} \sum_{n}^{N} (x_n - \mu)^2$$

3. *Compute the gradient of the log-likelihood*: to find the MLE for μ, we need to maximize the log-likelihood by taking its derivative with respect to μ and setting it to zero:

$$\frac{\partial l(\mu)}{\partial \mu} = \frac{\partial}{\partial \mu}\left(N \log \frac{1}{\sqrt{2\pi}\sigma} - \frac{1}{2\sigma^2} \sum_{n}^{N} (x_n - \mu)^2\right)$$

Focusing on the second term, the derivative is:

$$\frac{\partial l(\mu)}{\partial \mu} = -\frac{1}{2\sigma^2} \frac{\partial}{\partial \mu} \sum_{n}^{N} (x_n - \mu)^2 = \frac{1}{2\sigma^2} \sum_{n}^{N} 2(x_n - \mu)$$

4. *Solve the equation for MLE*: setting the derivative equal to zero to maximize the likelihood:

$$\frac{1}{2\sigma^2} \sum_{n}^{N} 2(x_n - \mu) = 0$$

This simplifies to:

$$\sum_{n}^{N} x_n = \sum_{n}^{N} \mu$$

Hence, the MLE for μ is:

$$\mu^{\text{MLE}} = \frac{1}{N} \sum_{n}^{N} x_n$$

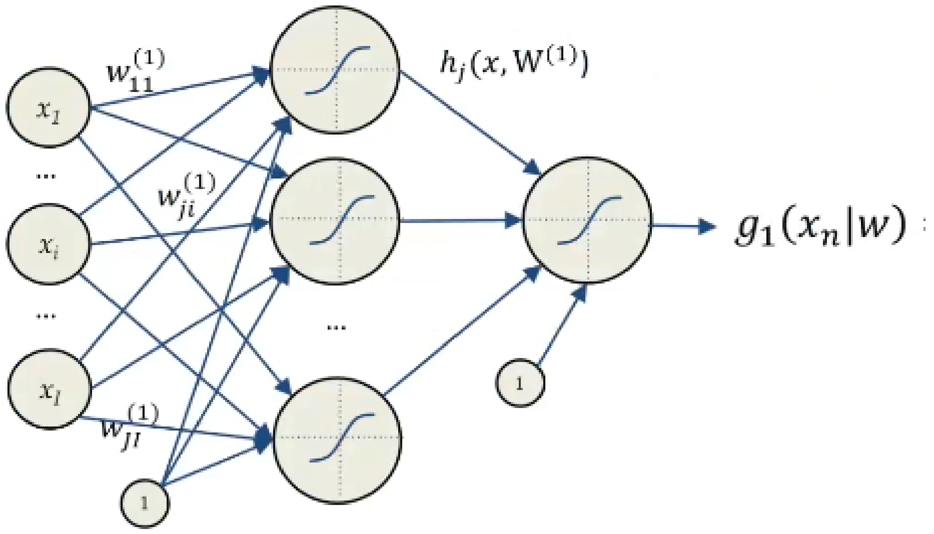
This is the sample mean, which is the optimal estimate for μ under the MLE framework.

To maximize or minimize the log-likelihood, we can apply several techniques:

- *Analytical methods*: directly solve the equations for the MLE, as done above.
- *Optimization techniques*: use methods such as Lagrange multipliers for constrained optimization problems.
- *Numerical methods*: apply iterative approaches like gradient descent when closed-form solutions are intractable.

In our case, we derived the MLE for μ analytically as the sample mean, but for more complex models, numerical optimization techniques may be necessary.

Example:
Consider the following FFNN.



The output of the network is defined as:

$$g_1(x_n|\mathbf{w}) = g_1\left(\sum_{j=0}^{J} w_{1,j}^{(2)} h_j\left(\sum_{i=0}^{I} w_{j,i}^{(1)} x_{i,n}\right)\right)$$

Here, h_j represents the activation function of the hidden neurons, and $w_{i,j}^{(1)}$ and $w_{i,j}^{(2)}$ are the weights of the first and second layers, respectively. The goal is to approximate a target function t having N observations:

$$t_n = g(x_n|\mathbf{w}) + \epsilon_n \qquad \epsilon_n \sim N(0, \sigma^2) \to t_n \sim N(g(x_n|\mathbf{w}), \sigma^2)$$

To estimate the weights $\mathbf{w}$ in a model $g(x_n|\mathbf{w})$ using Maximum Likelihood Estimation (MLE), follow these steps:

1. *Write the likelihood function $L(\mathbf{w}) = P(data|\mathbf{w})$ for the data*: assume that the observed values t_n are drawn from a Gaussian distribution with known variance σ^2, and that the model $g(x_n|\mathbf{w})$ provides the mean of the distribution. The probability of each observation is given by:

$$\Pr(t_n|g(x_n|\mathbf{w}), \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(t_n - g(x_n|\mathbf{w}))^2}{2\sigma^2}}$$

The likelihood function for the entire dataset is the product of these probabilities:

$$L(\mathbf{w}) = \Pr(t_1, t_2, \ldots, t_N | g(x|\mathbf{w}), \sigma^2) = \prod_{n=1}^{N} \Pr(t_n | g(x_n|\mathbf{w}), \sigma^2)$$

Substituting the expression for each $\Pr(t_n)$, we get:

$$L(\mathbf{w}) = \prod_{n=1}^{N} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(t_n - g(x_n|\mathbf{w}))^2}{2\sigma^2}}$$

2. *Write the log-likelihood function* $l(\mathbf{w}) = \log P(\mathit{data}|\mathbf{w})$:taking the logarithm of the likelihood function simplifies the product into a sum:

$$l(\mathbf{w}) = \log \left(\prod_{n=1}^{N} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(t_n - g(x_n|\mathbf{w}))^2}{2\sigma^2}} \right)$$

This simplifies to:

$$l(\mathbf{w}) = \sum_{n=1}^{N} \log \frac{1}{\sqrt{2\pi\sigma^2}} - \frac{(t_n - g(x_n|\mathbf{w}))^2}{2\sigma^2}$$

Further simplifying:

$$l(\mathbf{w}) = N \log \frac{1}{\sqrt{2\pi\sigma^2}} - \frac{1}{2\sigma^2} \sum_{n=1}^{N} (t_n - g(x_n|\mathbf{w}))^2$$

3. *Optimize the weights* $\mathbf{w}$ *to maximize the log-likelihood*: to find the weights $\mathbf{w}$ that maximize the log-likelihood, we solve the following optimization problem:

$$\mathbf{w}^{\text{MLE}} = \underset{\mathbf{w}}{\operatorname{argmax}}\, l(\mathbf{w})$$

This is equivalent to minimizing the sum of squared errors:

$$\mathbf{w}^{\text{MLE}} = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{n=1}^{N} \left(t_n - g(x_n|\mathbf{w})\right)^2$$

This optimization problem is typically solved using numerical techniques like gradient descent, depending on the complexity of $g(x_n|\mathbf{w})$.

4. *Approximate the posterior probability for t*:f or binary classification where $t_n \in \{0, 1\}$ and $t_n \sim \text{Bernoulli}(g(x_n|\mathbf{w}))$, the likelihood for the data is given by:

$$\Pr(t_n | g(x_n|\mathbf{w})) = g(x_n|\mathbf{w})^{t_n} \cdot (1 - g(x_n|\mathbf{w}))^{1-t_n}$$

The likelihood function for the entire dataset is:

$$L(\mathbf{w}) = \prod_{n=1}^{N} g(x_n|\mathbf{w})^{t_n} \cdot (1 - g(x_n|\mathbf{w}))^{1-t_n}$$

5. *Compute the log-likelihood*: taking the logarithm of the likelihood function gives:

$$l(\mathbf{w}) = \sum_{n=1}^{N} [t_n \log g(x_n|\mathbf{w}) + (1 - t_n) \log(1 - g(x_n|\mathbf{w}))]$$

This expression is known as the cross-entropy loss:

$$-\sum_{n=1}^{N} t_n \log g(x_n|\mathbf{w}) + (1 - t_n) \log(1 - g(x_n|\mathbf{w}))$$

2.4.1 Loss function selection

The choice of an appropriate loss function is crucial for defining the task and guiding the learning process. The loss function not only measures the error between the predicted and actual values but also influences the optimization behavior of the model. Designing a loss function requires careful consideration of various factors:

- *Leverage knowledge of the data distribution*: when selecting a loss function, it is important to incorporate any prior knowledge or assumptions regarding the underlying data distribution. For example, if the data is normally distributed, a squared error loss might be appropriate, while for binary classification tasks, cross-entropy loss is typically used.
- *Exploit task-specific and model knowledge*: a good loss function should align with the goals of the task at hand. For instance, in classification problems, we want to maximize the probability of correct predictions, and in regression, we aim to minimize the difference between predicted and true values. Tailoring the loss function to the model and its intended use case can significantly improve performance.
- *Creativity in loss function design*: in some cases, predefined loss functions may not fully capture the nuances of the problem, and this is where creativity can play a role. Custom loss functions can be designed by combining multiple loss components or incorporating domain-specific constraints to better align with the task's objectives.

In summary, selecting the right loss function is both a science and an art—it requires a blend of theoretical insights, practical understanding of the problem domain, and sometimes, creative thinking to balance accuracy and interpretability.

2.4.2 Perceptron loss function

Hyperplanes Consider a hyperplane $L : \mathbf{w}^T x + w_0 = 0 \in \mathbb{R}^2$. Any two points x_1 and x_2 lying on the hyperplane L can be characterized by their relationship to this hyperplane. The normal vector $\mathbf{w}^*$ to the hyperplane can be defined as follows:

$$\mathbf{w}^* = \frac{\mathbf{w}}{\|\mathbf{w}\|}$$

For any point x_0 on the hyperplane L, we can express the signed distance of any point x from the hyperplane L as:

$$\mathbf{w}^{*^T}(x - x_0)\frac{1}{\|\mathbf{w}\|} = \frac{1}{\|\mathbf{w}\|}\left(\mathbf{w}^T x + w_0\right)$$

It can be shown that the error function minimized by the Hebbian rule is related to the distance of misclassified points from the decision boundary. When coding the perceptron output as 1 or -1, we can represent the following conditions:

- If an output that should be 1 is misclassified, then $\mathbf{w}^T x + w_0 < 0$.
- Conversely, for an output that should be -1, we have $\mathbf{w}^T x + w_0 > 0$.

The objective then becomes to minimize the following loss function:

$$D(\mathbf{w}, w_0) = -\sum_{i \in M} t_i (\mathbf{w}^T x_i + w_0)$$

Here, $D(\mathbf{w}, w_0)$ is non-negative and proportional to the distance of the misclassified points from the decision boundary defined by $\mathbf{w}^T x + w_0 = 0$.

To minimize the error function $D(\mathbf{w}, w_0)$ using stochastic gradient descent, we take the gradients with respect to the model parameters:

$$\frac{\partial D(\mathbf{w}, w_0)}{\partial \mathbf{w}} = -\sum_{i \in M} t_i \cdot x_i \qquad \frac{\partial D(\mathbf{w}, w_0)}{\partial w_0} = -\sum_{i \in M} t_i$$

Stochastic gradient descent is applied iteratively for each misclassified point:

$$\begin{cases} \mathbf{w}_{k+1} = \mathbf{w}_k + \eta t_i \cdot x_i \\ w_{0,k+1} = w_{0,k} + \eta t_i \end{cases}$$

In this context, η represents the learning rate. This iterative approach is akin to Hebbian learning and effectively implements stochastic gradient descent, allowing the perceptron to adjust its weights and bias based on the misclassified instances.