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, in which 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.

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.

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.

1.3.2 Artificial neuron

The mathematical model of a neuron is represented as follows:

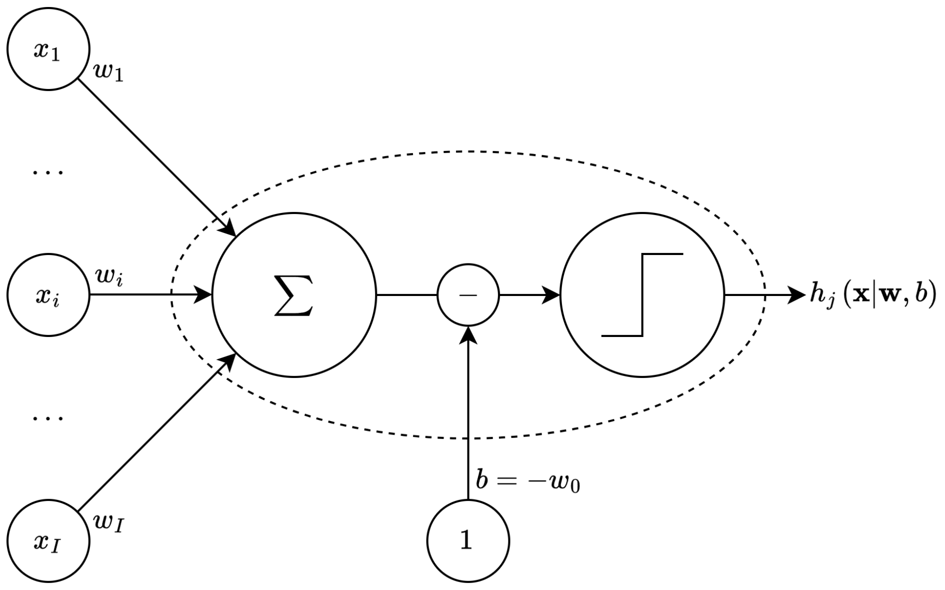


Figure 1.1: Artificial neuron

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, 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:

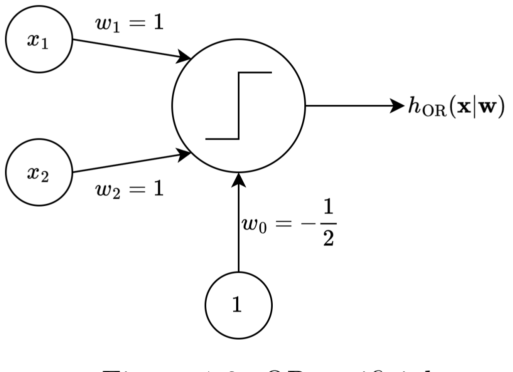


Figure 1.2: OR artificial neuron

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:

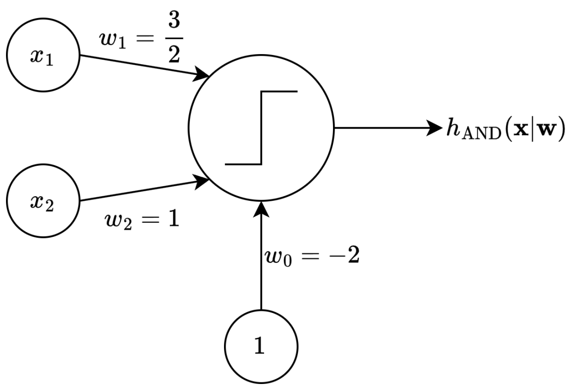


Figure 1.3: AND artificial neuron

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:

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

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:

$$y_{\text{second row}} x_0w_0 + x_1w_1 + x_2w_2 = 1 \cdot \left(-\frac{1}{2}\right) + (-1) \cdot \frac{1}{2} + 1 \cdot \frac{1}{2} = -\frac{1}{2}$$

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

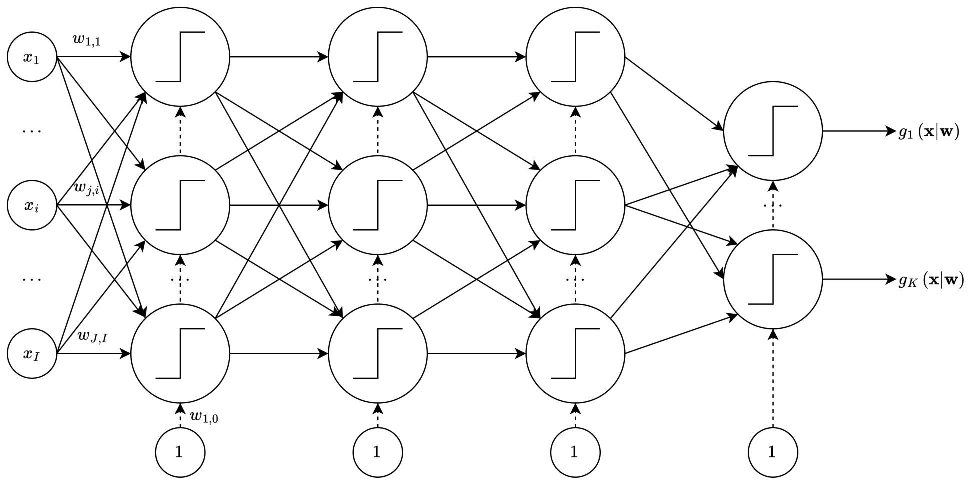


Figure 2.1: Multi-Layer Perceptron architecture

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.

2.2.3 Alternative activation functions

Activation functions like sigmoid or hyperbolic tangent tend to saturate, meaning their gradients become close to zero, or at least less than one, in many cases. Since backpropagation relies on gradient multiplication, this saturation causes gradients to vanish as they propagate backward through the network. This leads to the well-known vanishing gradient problem, where learning becomes extremely slow or even impossible, especially in deep networks. Although this problem is particularly evident in Recurrent Neural Networks, it also affects deep feed-forward networks, and for a long time, it was a significant obstacle to neural network training.

A popular solution to this issue is the Rectified Linear Unit (ReLU) activation function, defined as:

$$g(a) = \mathrm{ReLU}(a = \max(0, a))$$

Its derivative is:

$$g'(a) = 1_{a>0}$$

ReLU has several notable advantages:

- *Faster convergence*: stochastic Gradient Descent converges approximately six times faster with ReLU compared to sigmoid or tanh activations.
- *Sparse activation*: since only a portion of the hidden units are activated (i.e., those with positive input), this leads to a more efficient representation.
- *Efficient gradient propagation*: ReLU avoids both the vanishing and exploding gradient problems, making it easier to train deep networks.
- *Efficient computation*: ReLU is computationally efficient, as it only involves simple thresholding at zero.
- *Scale-invariance*: the output of ReLU is invariant to the scale of its input, meaning $\max(0, xa) = a\max(0, x)$.

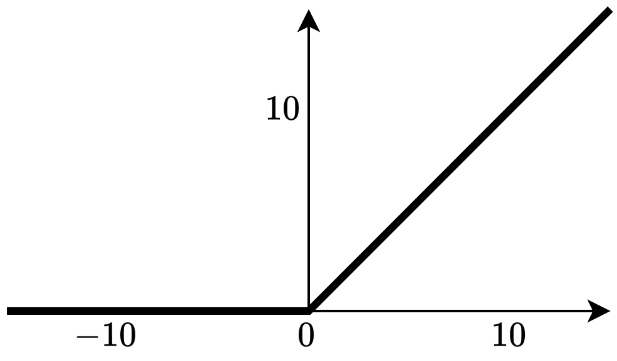


Figure 2.2: Rectified Linear Unit

Despite its advantages, ReLU has some potential downsides:

- *Non-differentiability at zero*: while ReLU is non-differentiable at zero, it is differentiable everywhere else. This non-differentiability rarely causes significant issues in practice.
- *Non-zero centered output*: ReLU outputs are non-zero-centered, which can lead to imbalanced updates when using gradient descent.
- *Unbounded output*: since ReLU does not cap its output, the activations can grow very large, potentially leading to exploding gradients under high learning rates.
- *Dying neurons*: a key issue with ReLU is the dying neuron problem, where neurons can get stuck with negative outputs for all inputs. When this happens, their gradients become zero, effectively killing those neurons, as they stop updating during training.

ReLU variants Several variants of ReLU have been developed to address some of its limitations:

- *Leaky ReLU*: this variant addresses the dying ReLU problem by allowing a small, non-zero gradient for negative input values:

  $$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0.01x & \text{otherwise} \end{cases}$$

  By using a small slope for negative inputs, Leaky ReLU ensures that neurons are less likely to die.

- *Exponential Linear Unit* (ELU): ELU aims to bring mean activations closer to zero, which can accelerate learning. It introduces an alpha parameter that controls the saturation for negative values:

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

  The α parameter is typically tuned by hand, and ELU provides smoother, more balanced outputs.

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:

$$w^{k+1} = w^k - \eta \left.\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}}\right|_{w^k}$$

   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:

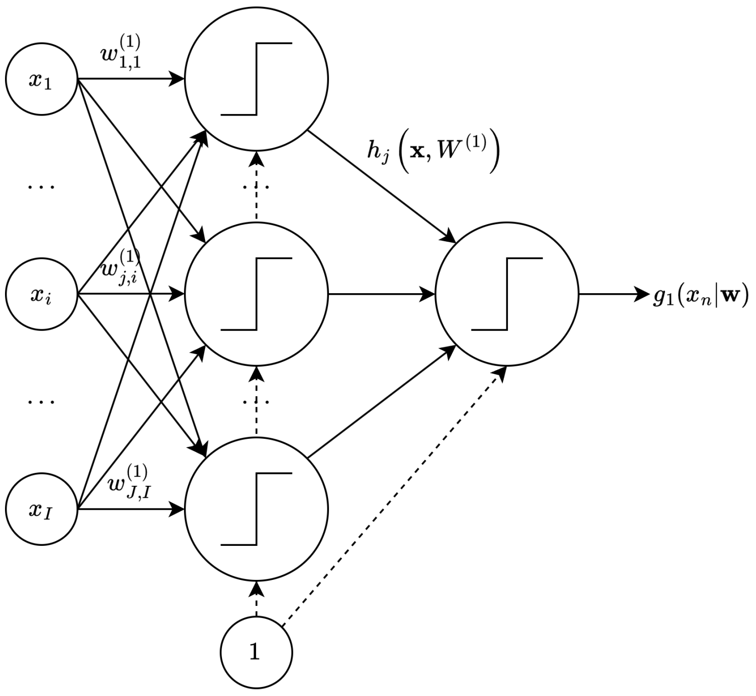
$$w^{k+1} = w^k - \eta \left.\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}\right|_{w^k} - \alpha \left.\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}\right|_{w^k}$$

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 feed-forward neural network:



The output of the network is defined as:

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

Here, h_j represents the activation function of the hidden neurons, and $w^{(1)}_{i,j}$ and $w^{(2)}_{i,j}$ 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 $w^{(1)}_{3,5}$ 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 $w^{(1)}_{3,5}$, we obtain the following update rule:

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

This expression includes the derivative of the output function g_1', the weight $w^{(2)}_{1,3}$ 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 $w^{(1)}_{j,i}$. Using the chain rule, we can express the partial derivative of the error function E with respect to $w^{(1)}_{j,i}$ as:

$$\begin{aligned}\frac{\partial E(w^{(1)}_{j,i})}{\partial w^{(1)}_{j,i}} &= \underbrace{-2\sum_{n}^{N}(t_n - g_1(x_n, \mathbf{w}))}_{\frac{\partial E}{\partial g(x_n,\mathbf{w})}} \underbrace{g'_1(x_n, \mathbf{w})}_{\frac{\partial g(x_n,\mathbf{w})}{\partial w^{(2)}_{1,j} h_j(\cdot)}} \underbrace{w^{(2)}_{1,j}}_{\frac{\partial w^{(2)}_{1,j} h_j(\cdot)}{\partial h_j(\cdot)}} \underbrace{h'_j\left(\sum_{i=0}^{I} w^{(1)}_{j,i} x_{i,n}\right)}_{\frac{\partial h_j(\cdot)}{\partial w^{(1)}_{j,i} x_i}} \underbrace{x_i}_{\frac{\partial w^{(1)}_{j,i} x_i}{\partial w^{(1)}_{j,i}}} \\ &= \frac{\partial E}{\partial g(x_n, \mathbf{w})} \cdot \frac{\partial g(x_n, \mathbf{w})}{\partial w^{(2)}_{1,j} h_j(\cdot)} \cdot \frac{\partial w^{(2)}_{1,j} h_j(\cdot)}{\partial h_j(\cdot)} \cdot \frac{\partial h_j(\cdot)}{\partial w^{(1)}_{j,i} x_i} \cdot \frac{\partial w^{(1)}_{j,i} x_i}{\partial w^{(1)}_{j,i}}\end{aligned}$$

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.

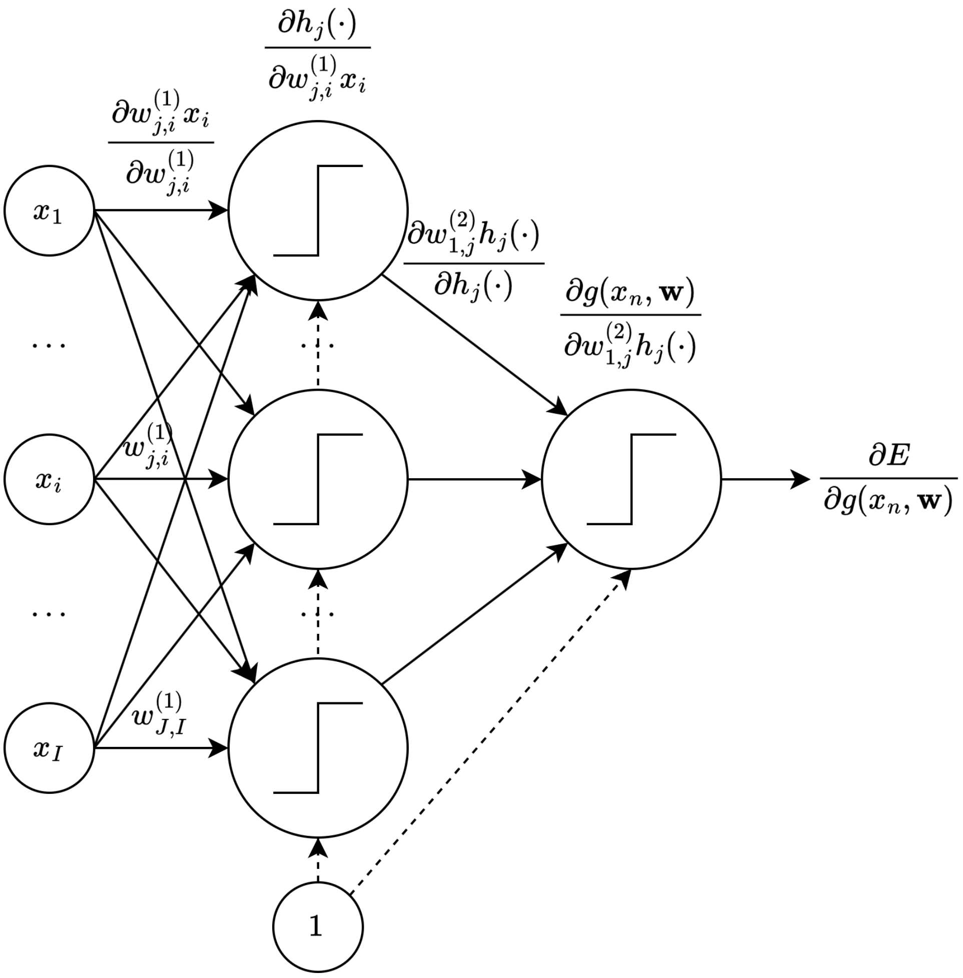

$\frac{\partial h_j(\cdot)}{\partial w_{j,i}^{(1)} x_i}$
$\frac{\partial w_{j,i}^{(1)} x_i}{\partial w_{j,i}^{(1)}}$
x_1
$\frac{\partial w_{1,j}^{(2)} h_j(\cdot)}{\partial h_j(\cdot)}$
$\frac{\partial g(x_n, \mathbf{w})}{\partial w_{1,j}^{(2)} h_j(\cdot)}$
$\dots$
$w_{j,i}^{(1)}$
x_i
$\frac{\partial E}{\partial g(x_n, \mathbf{w})}$
$\dots$
$w_{J,I}^{(1)}$
x_I
1

Figure 2.3: 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.

2.3.4 Batch normalization

To achieve faster convergence in neural networks, it is beneficial for the inputs to be whitened, meaning they should have a zero mean and unit variance while also being uncorrelated. This practice helps mitigate covariate shift. However, in addition to external covariate shifts, networks can experience internal covariate shifts, where the distribution of each layer's inputs changes during training. Batch normalization is a powerful technique to address this issue.

Batch normalization normalizes the activations of each layer, forcing them to take on values that approximate a unit Gaussian distribution at the start of training. It is typically implemented as follows:

- A BatchNorm layer is added after fully connected or convolutional layers and before the activation functions (nonlinearities).
- This layer effectively acts as a preprocessing step at each level of the network but is integrated in a differentiable manner, allowing for backpropagation.

Algorithm The input comprises values of x over a mini-batch $\mathcal{B} = \{x_1, \ldots, x_m\}$. The outputs are the parameter to be learned γ, β with respect to the given min-batch. Batch normalization

Algorithm 1 Batch normalization

```
1: Compute mini-batch mean: $\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$
2: Compute mini-batch variance: $\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$
3: Normalize: $\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$
4: Scale and shift: $y_i \leftarrow \gamma \hat{x}_i + \beta$
```

has several notable benefits:

- *Improved gradient flow*: it enhances the flow of gradients through the network, making it easier to train deep architectures.
- *Higher learning rates*: allows for the use of higher learning rates, leading to faster convergence.
- *Reduced dependency on weight initialization*: the strong dependence on initial weights is mitigated, as the normalization process stabilizes the learning.
- *Regularization effect*: acts as a form of regularization, slightly decreasing the need for techniques like dropout.

By incorporating batch normalization into neural network architectures, practitioners can significantly enhance training efficiency and model performance.

2.3.5 Weight initialization

The effectiveness of gradient descent in neural networks is highly dependent on how weights are initialized at the start. Various approaches to weight initialization can significantly impact the convergence speed and stability of the network:

- *Zero initialization*: setting all weights to zero is ineffective because it results in zero gradients, meaning no learning occurs. Symmetry is never broken, and the model fails to converge.

- *Large weights*: Initializing with large weights is also problematic. In deep networks, large initial weights can cause gradients to explode as they propagate back through the layers, leading to instability and slow convergence.
- *Small weights* (Gaussian distribution): initializing weights with small values from a Gaussian distribution, such as $\mathbf{w} \sim \mathcal{N}(0, \sigma^2) = 0.01$, works for shallow networks but may pose challenges for deeper architectures. Small initial weights in deep networks can cause the gradients to shrink as they pass through each layer, slowing down learning and potentially leading to the vanishing gradient problem.

In deep networks, weight initialization plays a critical role in avoiding the vanishing or exploding gradient problems:

- *Small weights*: if the initial weights are too small, gradients shrink during backpropagation, eventually becoming so small that learning effectively halts.
- *Large weights*: if the initial weights are too large, gradients can grow excessively, leading to unstable learning or divergence.

To address these issues, specific initialization methods have been proposed to ensure that the gradients remain balanced as they propagate through the network.

Xavier initialization Xavier initialization, proposed by Glorot and Bengio, aims to maintain the variance of the activations across layers, ensuring that signals neither vanish nor explode. Suppose we have an input vector $\mathbf{x}$ with I components, and a linear neuron with random weights $\mathbf{w}$. The output of the neuron is:

$$h_j = w_{j,1}x_1 + \cdots + w_{j,i}x_i + \cdots + w_{j,I}x_I$$

The variance of each term $w_{j,i}x_i$ is:

$$\mathrm{Var}(w_{j,i}x_i) = \mathrm{Var}(w_{j,i})\mathrm{Var}(x_i)$$

Assuming x_i and w_i are independent and identically distributed , the variance of the output h_j becomes:

$$\mathrm{Var}(h_j) = I \times \mathrm{Var}(w_i)\mathrm{Var}(x_i)$$

To keep the variance of the output the same as the input, Xavier initialization sets the weights as:

$$\mathbf{w} \sim \mathcal{N}\left(0, \frac{1}{n_{\text{in}}}\right)$$

Here, n_{in} is the number of input units to the neuron. For the gradients, Glorot and Bengio derived that the variance of the weights should also consider the number of output units n_{out}, leading to the refined initialization:

$$\mathbf{w} \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}} + n_{\text{out}}}\right)$$

This approach works well for activation functions like sigmoid or hyperbolic tangent.

He initialization More recently, He initialization was proposed specifically for networks using ReLU activations. Since ReLU activation functions only pass positive values, He initialization sets a slightly larger variance to prevent shrinking gradients. It uses the following distribution:

$$\mathbf{w} \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}}}\right)$$

This method helps maintain better gradient flow, especially in deep networks with ReLU activations, and is widely adopted in modern architectures.

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:

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

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:

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

   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:

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

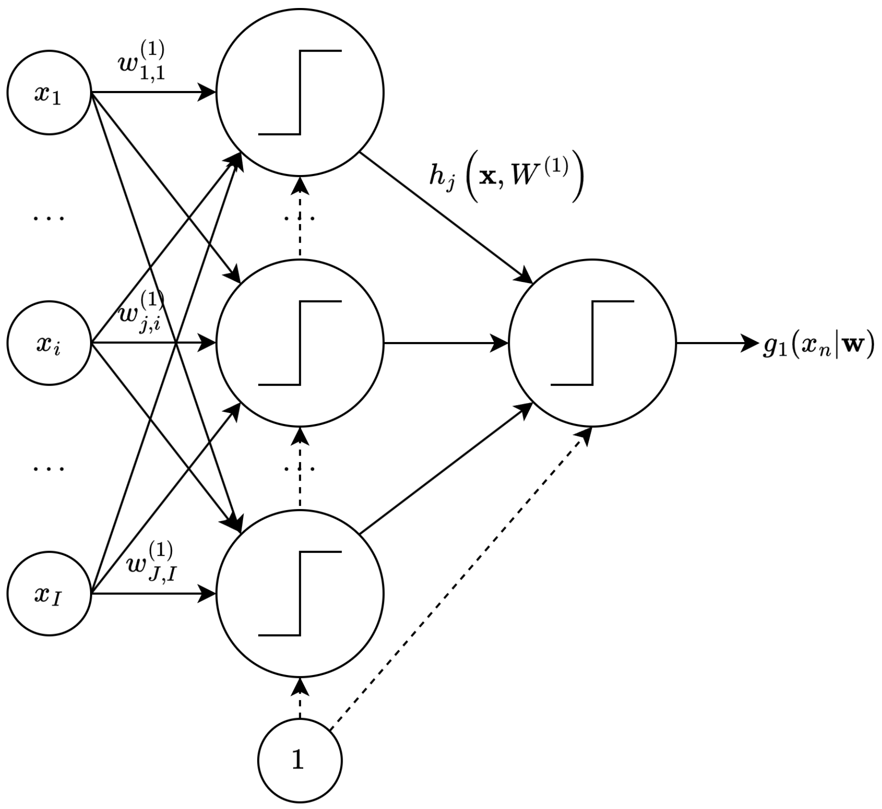
4. *Solve the equation for MLE*: setting the derivative equal to zero to maximize the likelihood:
$$\frac{1}{2\sigma^2}\sum_{n}^{N} 2(x_n - \mu) = 0$$
This simplifies to:
$$\sum_{n}^{N} x_n = \sum_{n}^{N} \mu$$
Hence, the MLE for μ is:
$$\mu^{\text{MLE}} = \frac{1}{N}\sum_{n}^{N} x_n$$
This is the sample 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 Feed Forward Nwural Network.



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:

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

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

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

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

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

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

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

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:

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

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.

2.4.3 Adaptive learning

To effectively navigate the challenges of local minima in the optimization landscape, various techniques can be employed, one of which is momentum. This method helps to accelerate gradient descent in the relevant direction and dampens oscillations. The update rule with momentum is given by:

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

Here, $\mathbf{w}^k$ represents the weights at iteration k, η is the learning rate, and α is the momentum term.

Nestorov accelerated gradient An enhancement to the momentum method is the Nesterov accelerated gradient. This approach first makes a momentum-based update and then adjusts the gradient based on the new position. The two-step update process is defined as follows:

1. First, a momentum-based prediction of the next position:

$$\mathbf{w}^{k+\frac{1}{2}} = \mathbf{w}^k - \alpha \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}\bigg|_{\mathbf{w}^{k-1}}$$

2. Then, updating the weights using the gradient evaluated at the predicted position:

$$\mathbf{w}^{k+1} = \mathbf{w}^k - \eta \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}\bigg|_{\mathbf{w}^{k+\frac{1}{2}}}$$

Layer-specific learning rates In deep networks, different neurons in each layer may learn at varying rates due to differences in gradient magnitudes across layers. Early layers are often prone to vanishing gradients, which can hinder learning. To address this, it is beneficial to employ separate adaptive learning rates for different layers. Several algorithms have been proposed to achieve this:

- *Resilient propagation* (Rprop): introduced by Riedmiller and Braun in 1993, this algorithm adjusts the weight updates based on the sign of the gradient, ensuring that the step sizes remain appropriate.
- *Adaptive gradient* (AdaGrad): developed by Duchi et al. in 2010, AdaGrad adapts the learning rate for each parameter based on the historical gradients, allowing for a more tailored approach.
- *RMSprop*: a combination of Stochastic Gradient Descent and Rprop, proposed by Tieleman and Hinton in 2012. RMSprop utilizes an exponentially decaying average of squared gradients to normalize the learning rate.
- *AdaDelta*: introduced by Zeiler et al. in 2012, this method extends AdaGrad by not accumulating past squared gradients, thus overcoming its diminishing learning rates issue.
- *Adam* (Adaptive Moment Estimation): proposed by Kingma and Ba in 2012, Adam combines the benefits of both momentum and RMSprop, using estimates of both the first and second moments of the gradients to adaptively adjust learning rates.

2.5 Overfitting

When learning a function, a single-layer model can theoretically approximate any measurable function to a high degree of accuracy on a compact set. However, this does not guarantee that we can find the necessary weights. In fact, an exponential number of hidden units may be required, making it impractical in practice, especially if the model does not generalize well to unseen data.

Underfitting occurs when the model is too simple to capture the underlying patterns in the data, leading to poor performance. To mitigate this, we can increase the model's complexity. However, if we increase the complexity too much, the model may adhere too closely to the training data, resulting in overfitting. Overfitting causes the model to capture noise rather than the true signal, reducing its ability to generalize to new, unseen data.

The goal of training is to find an approximation that balances complexity and generality, allowing the model to perform well not only on the training set but also on future data. This process involves finding a good balance between underfitting and overfitting, aiming for a model that generalizes well to new, unseen data rather than simply memorizing the training data.

The error or loss on the training data is not a reliable indicator of how well the model will perform on future data. There are several reasons for this:

- The model has been trained on the same data, so any estimate of performance based on the training set will likely be overly optimistic.
- New data will rarely match the training data exactly, meaning the model may encounter patterns it hasn't seen before.
- It's possible to find patterns even in random data, leading to the false belief that the model is performing well.

To accurately assess model performance, it's essential to evaluate it on a separate test set, which represents new, unseen data.

To properly evaluate the model's ability to generalize, we follow these steps:

1. *Test on an independent dataset*: it's critical to evaluate the model on a dataset that was not used during training.
2. *Data splitting*: split the available data into training, validation, and test sets. The test set is hidden and only used for final evaluation.
3. *Cross-validation*: perform random subsampling (with replacement) to split the data. In classification problems, ensure the class distribution is preserved using stratified sampling.

Definition (*Training dataset*)**.** The training dataset is the entire dataset available for building the model.

Definition (*Training set*)**.** The training set is subset of the data used to learn the model parameters.

Definition (*Test set*)**.** The test set is a separate subset of the data used for the final evaluation of the model's performance.

Definition (*Validation set*)**.** The validation set is a subset of the data used to fine-tune model hyperparameters and perform model selection.

Definition (*Training data*)**.** The training data is the data used during the training phase for both fitting and selection of the model.

Definition (*Validation data*)**.** The validation data is the data used to assess the model's performance during training, aiding in hyperparameter tuning and model selection.

2.5.1 Cross validation

Cross-validation is a technique used to estimate the performance of a model by utilizing the available training data for both training (parameter fitting and model selection) and error estimation on unseen data. This helps assess the model-s ability to generalize to new data without the need for a separate test set in the early stages of model development. The possible tecniques are:

- *Hold-out validation*: when a large amount of data is available, it's common to split the dataset into distinct subsets for training and validation. This allows for a straightforward estimation of model performance on unseen data. However, this method may not be suitable when the available data is limited, as it reserves part of the data for validation, reducing the amount used for training.

- *Leave-one-out cross-validation* (LOOCV): in cases where data is scarce, LOOCV can be applied. Here, the model is trained on all but one data point, which is then used for validation. This process is repeated for each data point, ensuring that every instance is used for both training and validation. While this method provides an unbiased estimate of model performance, it can be computationally expensive, particularly for large datasets.

- *K-fold cross-validation*: a more practical and commonly used approach is K-fold cross-validation. In this method, the dataset is randomly split into K equally sized subsets (folds). The model is trained on $K-1$ folds and validated on the remaining fold. This process is repeated K times, with each fold serving as the validation set once. The final model performance is computed as the average error across all K iterations. K-fold cross-validation offers a good balance between computational efficiency and performance estimation, and it is often preferred over LOOCV as it tends to provide more stable results with fewer repetitions.

2.5.2 Early stopping

Overfitting in neural networks often manifests as a monotonically decreasing training error as the number of gradient descent iterations k increases (especially when using stochastic gradient descent). However, while the model continues to perform better on the training set, its ability to generalize to unseen data can degrade beyond a certain point. Early stopping is a regularization technique designed to prevent overfitting by halting the training process when the model's performance on a validation set starts to deteriorate. The steps needed to use early stopping are:

1. *Hold out a validation set*: reserve a portion of the data for validation, separate from the training set.

2. *Train on the training set*: perform gradient descent iterations using the training set.

3. *Cross-Validate on the hold-out set*: continuously evaluate the model on the validation set during training.

4. *Stop training based on validation error*: when the validation error starts to increase (indicating the model is beginning to overfit the training data), halt the training process. This is where the model achieves its best generalization.

Model selection and evaluation occur at multiple levels:

- *Parameter level*: this involves learning the model's parameters, such as the weights $\mathbf{w}$ of a neural network, through optimization during training.

- *Hyperparameter level*: yhis involves choosing the structural components of the model, such as the number of layers L or the number of hidden neurons $J^{(l)}$ in each layer l. These hyperparameters significantly affect the model's capacity and generalization.

2.5.3 Weight decay

Regularization aims to constrain the freedom"of the model based on prior assumptions, in order to reduce overfitting. In the context of neural networks, weight decay is one of the most commonly used regularization techniques. It helps control model complexity by discouraging large weights, which can lead to overfitting.

Typically, we maximize the data likelihood when learning the model parameters:

$$w_{\text{MLE}} = \underset{\mathbf{w}}{\operatorname{argmax}} \Pr(\mathcal{D}|\mathbf{w})$$

However, to impose additional structure and reduce model complexity, we can introduce a Bayesian perspective by incorporating a prior over the weights $\mathbf{w}$. This leads to maximum a posteriori (MAP) estimation:

$$\begin{aligned} w_{\text{MAP}} &= \underset{\mathbf{w}}{\operatorname{argmax}} \Pr(\mathbf{w}|\mathcal{D}) \\ &= \underset{\mathbf{w}}{\operatorname{argmax}} \Pr(\mathcal{D}|\mathbf{w}) \Pr(\mathbf{w}) \end{aligned}$$

In practice, small weights tend to improve the generalization performance of neural networks, which can be reflected by assuming a Gaussian prior on the weights:

$$\Pr(\mathbf{w}) \sim \mathcal{N}(0, \sigma_{\mathbf{w}}^2)$$

By incorporating this prior, we can expand the optimization problem as follows:

$$\begin{aligned} \hat{\mathbf{w}} &= \underset{\mathbf{w}}{\operatorname{argmax}} \Pr(\mathbf{w}|\mathcal{D}) \\ &= \underset{\mathbf{w}}{\operatorname{argmax}} \Pr(\mathcal{D}|\mathbf{w}) \Pr(\mathbf{w}) \\ &= \underset{\mathbf{w}}{\operatorname{argmax}} \prod_{n=1}^{N} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(t_n - g(x_n|\mathbf{w})^2)}{2\sigma^2}} \prod_{q=1}^{Q} \frac{1}{\sqrt{2\pi}\sigma_{\mathbf{w}}} e^{-\frac{(w_q)^2}{2\sigma_{\mathbf{w}^2}}} \end{aligned}$$

Simplifying this, we get the following objective function:

$$\hat{\mathbf{w}} = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{n=1}^{N} \frac{(t_n - g(x_n|\mathbf{w})^2)}{2\sigma^2} + \sum_{q=1}^{Q} \frac{(w_q)^2}{2\sigma_{\mathbf{w}^2}}$$

This can be rewritten as:

$$\hat{\mathbf{w}} = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{n=1}^{N}(t_n - g(x_n|\mathbf{w})^2) + \gamma \sum_{q=1}^{Q}(w_q)^2$$

Here, γ is a regularization parameter that controls the strength of weight decay. Larger values of γ will penalize larger weights more strongly, encouraging smaller weights and thus reducing model complexity.

Gamma selection To select the optimal value of γ, we can use cross-validation as follows:

1. *Split the data*: divide the dataset into training and validation sets.
2. *Minimize for different γ values*: for each candidate value of γ, minimize the following training error:
$$\mathrm{E}_{\gamma}^{\text{train}} \sum_{n=1}^{N_{\text{train}}}(t_n - g(x_n|\mathbf{w})^2) + \gamma \sum_{q=1}^{Q}(w_q)^2$$
3. *Evaluate the model*: compute the validation error for each candidate γ:
$$\mathrm{E}_{\gamma}^{\text{validation}} \sum_{n=1}^{N_{\text{validation}}}(t_n - g(x_n|\mathbf{w})^2)$$
4. *Choose the optimal γ^**: select the value of γ that results in the best validation error.
5. *Final model training*: combine all data and minimize the objective with the selected γ^* that results in the best validation error.
$$\mathrm{E}_{\gamma^*} \sum_{n=1}^{N}(t_n - g(x_n|\mathbf{w})^2) + \gamma^* \sum_{q=1}^{Q}(w_q)^2$$

2.5.4 Dropout

Dropout is a stochastic regularization technique used to mitigate overfitting by randomly dropping out or deactivating certain neurons during training. By doing this, the model is forced to learn more robust and independent feature representations, preventing hidden units from relying too heavily on one another (a phenomenon known as co-adaptation). For each training iteration:

1. *Neuron deactivation*: each hidden unit is set to zero with a probability $\mathrm{Pr}_j^{(l)}$, where l denotes the layer and j the unit within that layer. This effectively turns off the neuron for that iteration.
2. *Applying a mask*: a mask is applied to the layer to randomly drop out neurons, removing part of the signal. This generates a sub-network from the original neural network, on which the training continues.
3. *Repeating with new masks*: at each training step, a new mask is applied, creating a different sub-network from the original. Training proceeds as if each sub-network is a distinct model.

4. *Reactivating all neurons at test time*: once training is complete, all neurons are reactivated for the full network during testing. The behavior of the full network is approximately the averaged result of all the sub-networks trained during dropout. To compensate for the dropouts during training, weight scaling is applied at test time, effectively averaging the outputs of all possible sub-networks.

Benefits One of the key advantages of dropout is that it helps prevent co-adaptation among neurons. By randomly deactivating certain neurons during training, the network is forced to distribute the learning process more evenly across all neurons, making it less dependent on specific units. This encourages the network to learn more robust and independent feature representations, improving generalization. Additionally, dropout can be viewed as a form of implicit ensemble learning. Each time a subset of neurons is dropped, the network effectively trains a different sub-network. At inference time, the full network's output can be seen as the average prediction of all these sub-networks, leading to a more generalizable model.

Dropout can be applied selectively to specific layers, typically to the more densely connected ones, rather than across every layer in the network. This targeted regularization can be more effective in reducing overfitting while maintaining computational efficiency. A good practice when training neural networks is to first train the model to the point of overfitting, which ensures that the problem is solvable and that the model has sufficient capacity to learn the task. Once overfitting is observed, techniques such as early stopping, dropout, or weight decay can be introduced to reduce overfitting and improve generalization.