Neuronske mreže i statističko učenje

Softić, Alen

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Alen Softić

NEURAL NETWORKS AND STATISTICAL LEARNING

Diplomski rad

Voditelj rada:
prof. dr. sc. Miljenko Huzak

Zagreb, svibanj, 2024.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

$\forall$

”for all”

Contents

Introduction

Artificial Neural Networks are a mathematical model originally developed to simulate the human brain. Their development began soon after the advent of computers in the fifties and sixties, and at the time they were referred to as *electronic* or *artificial brains* and *thinking machines*. Today, artificial neural networks are used in *Machine Learning* tasks by treating the computational units in a learning model as brain neurons. Although the biological analogy of neural networks is an exciting one and evokes comparisons with science fiction, the mathematical understanding of artificial neural networks is more mundane.

Artificial neural networks are theoretically capable of learning any mathematical function with sufficient training data. Some variants like *Recurrent Neural Networks* are known to be *Turing complete*, meaning that they can simulate any learning algorithm given sufficient data. The biggest obstacle is that the amount of data required to learn even simple tasks is often extraordinarily large, which causes a corresponding increase in computation time. Nevertheless, given that the speed of computers is increasing rapidly and more powerful paradigms like quantum computing are on the horizon, the computational issues might not turn out to be as critical as imagined.

Major events in *Artificial Intelligence* (AI) research can be found in Figure 0.1, along with indications of the *first AI winter* and the current *AI summer*. Topics covered in this thesis are:

1943**: The McCulloch-Pitts Neuron** The first mathematical model of a biological neuron

1957**: The Rosenblatt's Perceptron** The first hardware implementation of a mathematical neuron model

1959**: ADALINE** The first learning model developed in terms of optimization

1976**: Backpropagation algorithm** A recursive algorithm allowing efficient training of multilayered networks

1986**: Multilayer Perceptron** A classic multilayered artificial neural network model used even today

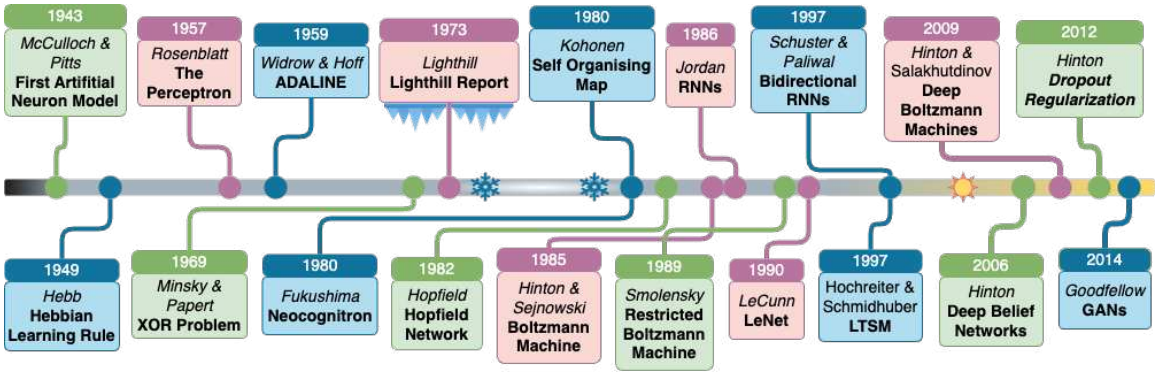


Figure 0.1: AI Timeline

Thesis outline

The main goal of this thesis is to explore artificial neural networks in a *binary classification* setting. This includes defining the statistical models, deriving their training algorithms, and justifying their use even through statistical learning theory. By unifying many important works in the field into a cohesive unit, this thesis provides a self-consistent introduction to the field of artificial neural networks. It is separated into two main chapters that comprise its title.

Chapter 1 describes the workings of *Biological Neural Networks* and their mechanisms, setting the stage for artificial neural networks. Through early artificial neural networks, we reach the *Multilayer Perceptron* and the *Backpropagation learning algorithm*, which are the principal components of the chapter.

Chapter 2 describes the foundations of *Statistical Learning Theory*, focusing on artificial neural networks. It justifies the models presented in Chapter 1 in a theoretical setting and presents interesting results concerning the learning process of neural networks.

Chapter 1

Neural Networks

1.1 Biological Neural Networks

The *neuron* (or nerve cell) is the fundamental anatomical and functional unit of the nervous system. It is an extension of a simple cell with two types of appendages (protrusions from the cell surface): multiple *dendrites* and an *axon*. Four main components of a neuron are the dendrites, the axon, the *soma* (cell body), and the *synapse*. They are shown in Fig. 1.1. The average human brain consists of nearly 10^{11} neurons of various types: unipolar, bipolar, multipolar, and pseudounipolar.

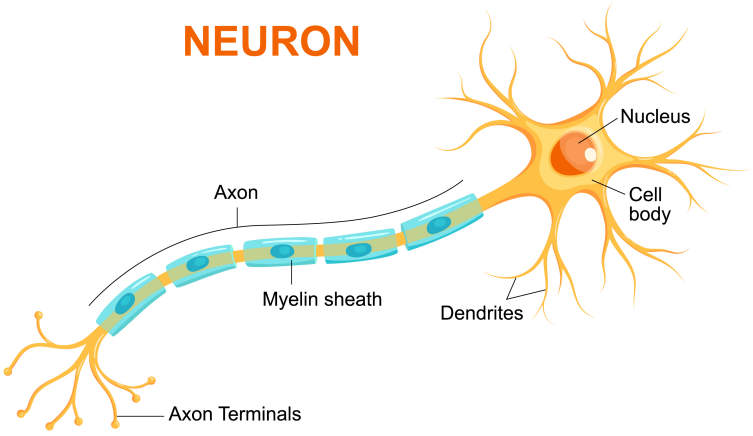


Figure 1.1: Schematic drawing of a multipolar neuron

The soma contains the cell nucleus. Dendrites branch into a short bushy network around the cell to receive input from other neurons, whereas the axon stretches out for a longer distance. The axon is an output channel to other neurons; it branches into strands and

substrands to connect to the dendrites and cell bodies of other neurons. The connecting junction is called a synapse, and each neuron has $10^4 - 10^5$ synaptic connections. For that reason, the biological neural network models are called *connectionist models*. Dendrites receive signals from other neurons and pass them onto the cell body to be processed, and the resulting signal is transferred through an axon. This process goes on and on throughout the entire *neural network*.

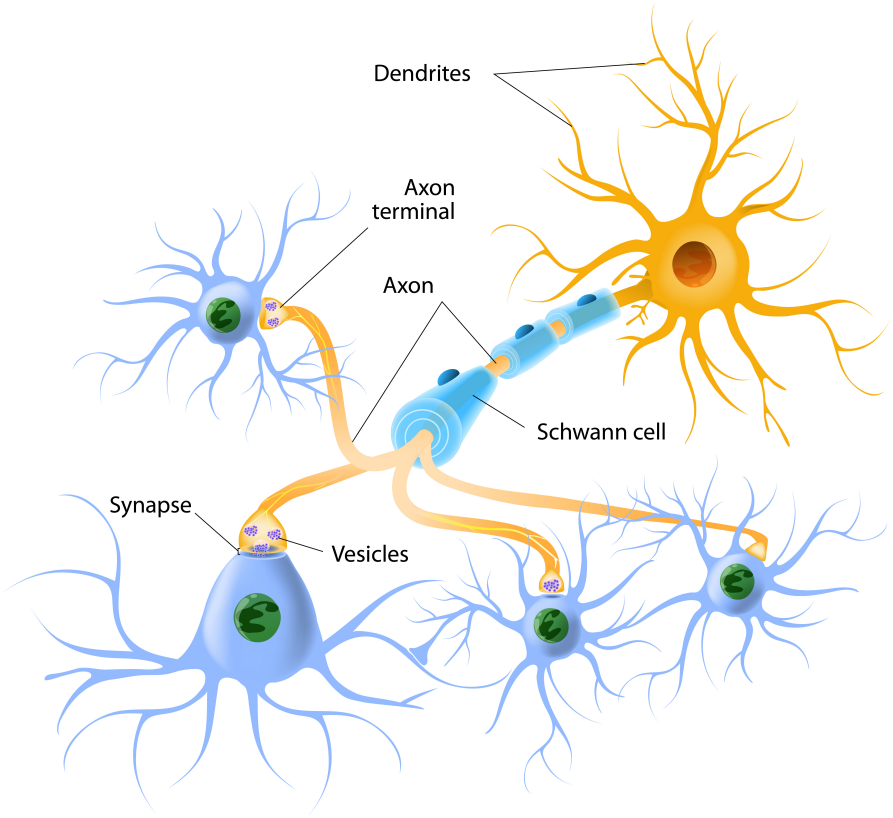


Figure 1.2: Synaptic connections

Like any other cell, neurons have a membrane potential, that is, an electric potential difference between the cell's interior and exterior. The cell membrane has an electrical *resting potential* of $-70mV$, however, unlike an ordinary cell, the neuron is *excitable*. Because of inputs from the dendrites, the cell may not be able to maintain the $-70mV$ resting potential, resulting in an *action potential* – an electrical pulse transmitted down the axon. Signals are thus propagated from neuron to neuron by a complicated electrochemical reaction.

When the potential is above a threshold, an electrical pulse (action potential) is sent along the axon. After releasing the pulse, the neuron returns to its resting potential. The action potential causes a release of certain biochemical agents for transmitting messages to the dendrites of nearby neurons. These biochemical transmitters may have either an *excitatory* or *inhibitory* effect on neighboring neurons. A synapse that increases the potential is excitatory, whereas a synapse that decreases it is inhibitory.

Synaptic connections exhibit *plasticity* – a long-term change in the strength of connections in response to the pattern of stimulation, where neurons can also form new connections with other neurons. Synaptic plasticity is a basic biological mechanism underlying *learning* and memory.

1.2 Artificial Neural Networks

As mentioned, this thesis covers artificial neural networks as a machine learning algorithm in *supervised learning* setting, used for binary classification. Meaning, that we are provided with a *training set* consisting of input-output pairs *(labelled data)*, with the output values being strictly 0 or 1 in this thesis. Also common output values are -1 and 1.

Section 1.1 was purposely detailed to show the upcoming similarity between biological neural networks (BNNs) and artificial neural networks (ANNs). Later, we will see how ANNs are a natural progression and generalisation of simpler machine learning algorithms such as *Linear regression* or *Logistic regression*. In other words, motivation from biology may seem unnecessary. However, major biological breakthroughs in discovering the workings of the human brain often led to significant improvements in ANNs as well.

The aforementioned bio-mechanisms of BNNs are simulated in ANNs. Let's recap; The human nervous system contains cells, which are referred to as neurons. The neurons are connected with axons and dendrites, and the connecting regions between them are called synapses. These connections are illustrated in Fig. 1.2. The strengths of these synaptic connections change through time in response to external stimuli. This change is how learning takes place in living organisms.

In ANNs, the computational units are the *artificial neurons*, but for simplicity, we'll refer to them as neurons from now on. Neurons are connected through weights, which serve the same role as the strengths of synaptic connections in biological neurons. The sign of the weight often corresponds to excitatory and inhibitory effects on the receiving neurons. Each neuron receives inputs scaled with weights and computes a certain *activation function*, equivalent to chemical signal processing in soma once it receives signals from other brain neurons through its dendrites. The output of a neuron is the value of the activation function (usually a number between 0 and 1 in our setting) and it is sent out to be received by other neurons as an input parameter (once scaled with an according weight), just like an electrical pulse through the axon.

An artificial neural network computes a function of the inputs by propagating the computed values from the input neurons to the output neuron(s) using the weights as intermediate parameters, the same way the signals are propagated from neuron to neuron in a BNN. Learning occurs by changing the weights connecting the neurons, analogous to synaptic plasticity. Just as external stimuli are needed for learning in biological organisms, the artificial neural networks are provided by the training data containing examples of input-output pairs of the function that is to be learned.

The training data provides feedback on the correctness of the weights based on how well the predicted output matches the true output label in the training data. One can view the errors made by the ANN as a kind of unpleasant feedback in a biological organism, such as pain or discomfort. This negative feedback is what leads to an adjustment in the synaptic strengths. Similarly, the weights between neurons are adjusted in a neural network in response to *prediction errors*. The weights are changed (or updated) to make more precise predictions in future iteration, mimicking learning in biological organisms. This duality between BNNs and ANNs is shown in Table 1.1.

BNNs	ANNs
Nerve cell	Neuron
Dendrites & Synapses	Weighted connections
Soma	Net input
Axon	Activation function
Synaptic plasticity	Weight update
External stimuli	Training set

Table 1.1: Simplified comparison of ANN's and BNN's components

The biological comparison is often criticized as a very poor replication of the workings of the human brain. Nevertheless, the principles of neuroscience have often been useful in designing *neural network architectures*. A different view is that neural networks are built as higher-level abstractions of the classical models that are commonly used in machine learning. The most basic units of computation in the neural network *(perceptrons)* are inspired by traditional machine learning algorithms like linear regression and logistic regression, and artificial neural networks gain their power by combining many such basic computational units.

From this point of view, a neural network can be viewed as a computational graph of elementary units in which greater power is gained by connecting them in particular ways. By combining multiple units, one is increasing the power of the model to learn more complicated functions. How these units are combined also plays a role. Designing the network architecture requires some understanding and insight from the analyst. Furthermore, sufficient training data is also required to learn a larger number of weights in these expanded computational graphs.

Many neural network architectures that have shown extraordinary performance were not created by randomly connecting computational units. *Deep neural networks* themselves mirror the fact that biological neural networks gain much of their power from depth as well. Furthermore, biological networks are connected in ways we don't fully understand yet, and in the few cases that the biological structure is understood at some level, significant breakthroughs have been achieved by designing artificial neural networks along those lines. A classical example of this type of architecture are the *convolutional neural networks*, used mostly for image recognition.

Short history of ANN development

Before moving to ANN models, let's mention some important historical events we'll cover in the next section related to their development.

In their 1943 paper, *"A logical calculus of the ideas immanent in nervous activity"*, McCulloch and Pitts [12] proposed that a neuron can be modelled as a simple threshold device to perform a logic function. Later, in 1949, Hebb [6] proposed the *Hebbian rule* to describe how learning affects the synapses between neurons. He stated that synaptic connections could adapt to different stimuli over time, where the connections that were used frequently together would gradually become stronger, while those that were not used would fade away. In 1952, based on the physical properties of cell membranes, Hodgkin and Huxley [7] modelled neuronal firing and action potential as a set of evolution equations, receiving a Nobel Prize in 1963 for their work. In the late 1950s, Rosenblatt [16] proposed the perceptron model, and Widrow and Hoff [19] proposed the *ADALINE* (adaptive linear element) model, trained with a least mean squares (LMS) method. Finally, in their 1969 book *"Perceptrons"*, Minsky and Papert [13] mathematically proved that the perceptron cannot be used for a more complex logic function, also known as the *XOR problem*. This, in conjunction with the Lighthill[1] report [10] led to a lack of funding in the field of neural networks and the stagnation in AI research known as the *first AI winter*, lasting from early 1970s through 1980s.

[1]The Lighthill Debate of 1973: `https://www.youtube.com/watch?v=yReDbeY7ZMU&list=PLhThm05V6bZPbfpbAyzFEU-qVT-OkpwLA` (visited on 16.04.2024).

1.3 McCulloch–Pitts Neuron Model

We are starting with the simplest ANN model called the *McCulloch–Pitts Neuron Model* or the *M-P Neuron*, also referred to as a *Linear Threshold Gate*. It's the earliest model proposed, and generally used for implementing logic functions. This is due to a limited understanding of the workings of the human brain at the time. Neurons were modelled as computing elements described in propositional logic: *"... neural events and the relations among them can be treated by means of propositional logic"* [12].

The McCulloch–Pitts neuron model consists of a single neuron called the M-P neuron. It takes an n-dimensional boolean vector[2] $X = [x_1, x_2, \ldots, x_n]^T \in \{0,1\}^n$ as its input. For each input x_i, there is a corresponding weight w_i, and we denote the vector of weights by $W = [w_1, w_2, \ldots, w_n]^T \in \{-1, 1\}^n$. Values -1 and 1 represent the inhibitory and excitatory behaviour respectively. The *threshold parameter* θ sets the threshold value for neuron activation. The McCulloch-Pitts neuron processes information by aggregating the weighted inputs and assessing whether their sum surpasses the threshold using an *activation function* ϕ. The resulting output of the M-P neuron is $y \in \{0, 1\}$.

Schematics of the M-P Neuron model are shown in Figure 1.3. It has two layers, the input and the output layer. Since it's only the output layer that does the calculations, we say that the M-P neuron has one *computational layer*. For this reason, it would be considered a *single-layer network*.

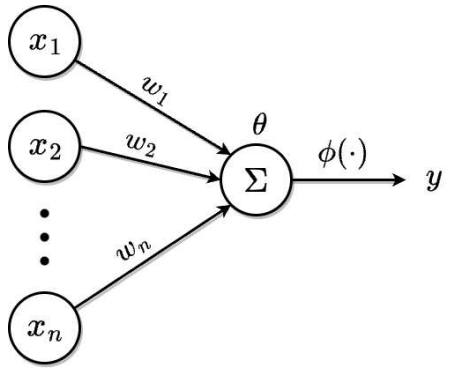


Figure 1.3: The McCulloch-Pitts neuron

For $k, n \in \mathbb{N}$, set $\mathcal{X} \subseteq \mathbb{R}^n$ denotes the *input space*, set $\mathcal{Y}$ the *output space*, and $\Omega \subseteq \mathbb{R}^k$ the set of *network states*. If unspecified, we take $\mathcal{X}$ and Ω to be $\mathbb{R}^n$, and $\mathcal{Y}$ to be $\{0, 1\}$.

The output space in the binary classification setting can generally be any set $\{a, b\} \subseteq \mathbb{R}$ such that $a \neq b$, but in this thesis is fixed to $\{0, 1\}$. The input space $\mathcal{X}$ is the set of allowed input values of real numbers. While this may seem restricting, more often than not problems have inputs that can be encoded in real numbers. Examples are images, sound etc.

[2]By vector we mean a column vector and we'll identify column vectors with n-tuples: $\mathbb{R}^n \equiv \mathbb{R}^{n\times 1}$.

The set of network states represents the possible values of *network parameters*, usually presented as vectors and matrices. In the case of the M-P neuron, the set of network states comprises all possible values for weights and the threshold.

Definition 1.3.1. Any Borel measurable function $f : \mathbb{R} \longrightarrow \mathbb{R}$ is called an *activation function*.

Definition 1.3.2. Function $\phi : \mathbb{R} \longrightarrow [0, 1]$ defined as

$$\phi(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases}$$

is called the *threshold activation function* or the *hard-limiter activation function*.

Remark 1.3.3. The threshold activation function is an activation function, justifying the name. Its preimage is always a Borel set; either $\emptyset$, $\langle -\infty, 0\rangle$, $\langle 0, +\infty]$ or $\mathbb{R}$.

Definition 1.3.4. Let ϕ be the threshold activation function, $n \in \mathbb{N}$ any natural number, $\mathcal{X} = \{0, 1\}^n$ the input space, $\mathcal{Y} = \{0, 1\}$ the output space, and $\Omega \subseteq \{-1, 1\}^n \times \mathbb{R}$ the set of network states. Function $y : \Omega \times \mathcal{X} \longrightarrow \mathcal{Y}$ defined as

$$y(W, \theta, X) = \phi\left(W^T X - \theta\right)$$

is called the *McCulloch-Pitts Neuron* or the *M-P Neuron*.

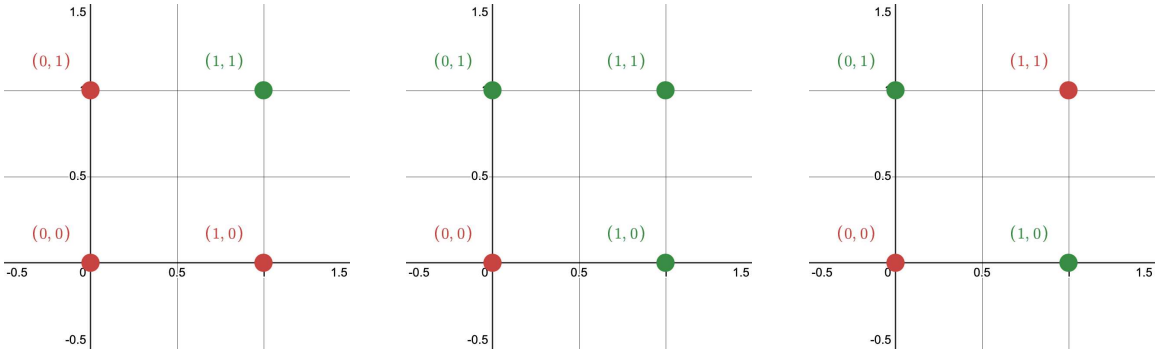
There isn't a *training phase* for the McCulloch-Pitts neuron model. The weights and the threshold parameter are not being *learned* in this setting, but rather chosen prematurely to solve a specific problem, such as implementing a logic (or *boolean*) function.

Definition 1.3.5. For $n \in \mathbb{N}$, boolean functions AND, OR, $XOR : \{0, 1\}^n \longrightarrow \{0, 1\}$ are defined as:

$$AND(x_1, x_2, \ldots, x_n) = \begin{cases} 1, & \forall i \in \{1, 2, \ldots, n\}\ x_i = 1 \\ 0, & \text{otherwise} \end{cases},$$

$$OR(x_1, x_2, \ldots, x_n) = \begin{cases} 1, & \exists i \in \{1, 2, \ldots, n\}\ x_i \neq 0 \\ 0, & \text{otherwise} \end{cases},$$

$$XOR(x_1, x_2, \ldots, x_n) = \begin{cases} 1, & x_1 + x_2 + \cdots + x_n \text{ is odd} \\ 0, & \text{otherwise} \end{cases}.$$



(a) *AND* function (b) *OR* function (c) *XOR* function

Figure 1.4: Graphical representation of boolean functions

Provided inputs are two-dimensional, we can visualise these functions in a 2D plane using colours green and red to represent the TRUE and FALSE values respectively.

Implementing the *AND* & *OR* logical functions with M-P neurons is simple, and the solution is not unique.

Proposition 1.3.6. *Let $n \in \mathbb{N}$ be any natural number, $\mathcal{X} = \{0,1\}^n$ the input space, $\mathcal{Y} = \{0,1\}$ the output space, and $\Omega = \{-1,1\}^n \times \mathbb{R}$ the set of network states. Then, there exist (W_1, θ_1), $(W_2, \theta_2) \in \Omega$ such that the M-P neuron $y : \Omega \times \mathcal{X} \longrightarrow \mathcal{Y}$ satisfies*

$$y(W_1, \theta_1, X) = AND(X),$$
$$y(W_2, \theta_2, X) = OR(X),$$

for all $X \in \mathcal{X}$.

Proof. Let $n \in \mathbb{N}$ be arbitrary. To implement the *AND* function, we set the vector of weights to $W_1 = \mathbb{1}_n$, where $\mathbb{1}_n = [1, 1, \ldots, 1]^T \in \mathbb{R}^n$, and the threshold parameter to $\theta_1 = n - 0.5$. The M-P neuron therefore assesses whether the number of incoming input signals is greater than $n - 0.5$, which can only be true in case of $X = \mathbb{1}_n$.

Similarly, for the *OR* function, we use the same vector of weights $W_2 = \mathbb{1}_n$, and set the threshold parameter to $\theta_2 = 0.5$. The threshold will be surpassed in case the M-P neuron receives at least one signal from the input, or $X \neq 0 \cdot \mathbb{1}_n$.

Therefore, $(W_1, \theta_1) = (\mathbb{1}_n, n - 0.5)$, and $(W_2, \theta_2) = (\mathbb{1}_n, 0.5)$ give a solution.

□

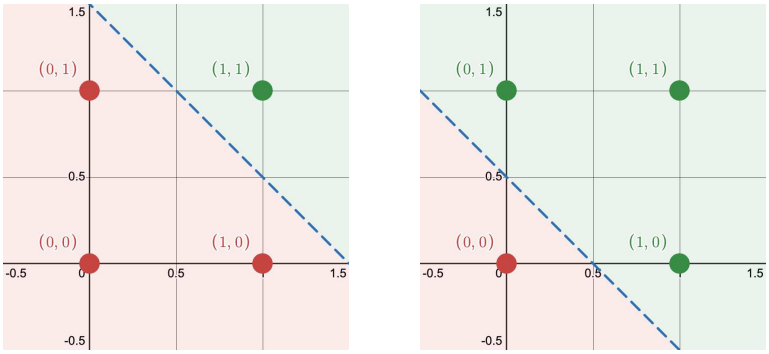
The solution is not unique. Interestingly, there are no parameters $(W, \theta) \in \Omega$ such that the M-P neuron implements the *XOR* function even in the simples case when $n = 2$. This implies that a more advanced approach is necessary in order to solve the *XOR problem.*

The XOR problem

Looking at the geometry of the M-P neuron model, one can easily see why it can't implement the *XOR* function. To reach said geometry, let's look at the following format the M-P neurons output denoted simply by y, given X, W, and θ:

$$y = \begin{cases} 1, & W^T X - \theta \geq 0 \\ 0, & W^T X - \theta < 0 \end{cases}.$$

In 2D case, the equation $w_1 x_1 + w_2 x_2 - \theta = 0$ is a line equation. This line serves as a *decision boundary* of a 2D plane, splitting it into 2 regions. When implementing boolean functions, we are essentially looking for weights W and threshold θ such that each *decision region* made by the decision boundary contains strictly all TRUE or all FALSE values. An example of such decision regions for the *AND* & *OR* boolean functions is shown below.



(a) *AND* function (b) *OR* function

Figure 1.5: Decision regions of boolean functions

Definition 1.3.7. Let $n \in \mathbb{N}$ and $A, B \subseteq \mathbb{R}^n$. We say that A and B are *linearly separable in* $\mathbb{R}^n$ if there exists a vector $w \in \mathbb{R}^n$ and a real number θ, such that $w^T a - \theta > 0$ and $w^T b - \theta < 0$ for all $a \in A$, $b \in B$. Otherwise, sets A and B are called *linearly inseparable*.

Definition 1.3.8. For any two linearly separable sets in $\mathbb{R}^n$ and their given w and θ from definition 1.3.7, the set $\{x \in \mathbb{R}^n : w^T x - \theta = 0\}$ is called the *(linear) decision boundary* (or a *separating hyperplane*), and the two halfspaces it divides the $\mathbb{R}^n$ into are called the *decision regions*. Such vector w is called the *normal* (to the separating hyperplane).

Definition 1.3.9. We say that a function $f : \mathbb{R}^n \supseteq D \longrightarrow \{0, 1\}$ is linearly separable in $\mathbb{R}^n$ if the preimages $f^{-1}(\{0\}), f^{-1}(\{1\}) \subseteq \mathbb{R}^n$ are linearly separable in $\mathbb{R}^n$.

Since its decision boundary is a line, the M-P neuron can only be used to implement linearly separable functions. From their graphs, it's obvious that *AND* & *OR* functions are linearly separable, and the decision boundaries from Proposition 1.3.6 are shown in Figure 1.5. Conversely, the graph of the *XOR* function from Figure 1.4c shows how any decision region containing points $(0, 1)$ and $(1, 0)$ will contain at least one more point from $\{(0, 0), (1, 1)\}$.

As mentioned, this realisation led to stagnation in ANN research after the initial excitement. However, it is possible to solve the XOR problem, but it requires combining multiple M-P neurons, allowing the network to encompass more complicated boolean functions. The true power of ANNs comes from combining multiple neurons, together with using a non-linear activation function.

To overcome the limitations of the M-P neuron, Frank Rosenblatt proposed the classical perception model in 1957 [15]. It is a more generalised computational model compared to the McCulloch-Pitts neuron, where real-valued weights can be *learned* over time.

1.4 Rosenblatt's Perceptron

At the time, there were two approaches to brain information storage and memory: the *coded representations* approach and the *connectionist* approach.

The coded representation approach states that sensory information is stored in the form of coded representations – a one-to-one mapping between the sensory stimuli and the stored pattern. Meaning, that information is precisely stored in the brain in a way you would be able to retrieve it from a particular location. So precisely in fact, that you could tell the contents encoded in the neurons by looking at how they are connected.

The connectionist approach Roseblatt advocated states that there is no unique cluster of neurons, wired in a specific way such that they encode the memory. Instead, memory is a series of associations among a set of neurons that tend to react in the presence of any familiar stimuli.

Rosenblatt's perceptron, the so-called *photo-perceptron*, was intended to emulate the functionality of the human eye. In today's terms, we would say it was designed for image recognition.[3] It consisted of three "systems" composed of individual units (neurons): the S-system (sensory system), the A-system (association system), and the R-system (response system).

Stimuli would hit the retina sensory units (S-units) generating a binary response. The S-system was essentially the input layer and it was connected with fixed weights to the A-system at random. Signs of the weights had the same excitatory/inhibitory interpretation

[3]Demonstration of the photo-perceptron: `www.youtube.com/watch?v=cNxadbrN_aI&t=1s` (visited on 16.04.2024).

as the M-P neuron. Each A-unit in the following system receives an impulse from several S-units and transmits an output to one or more R-units. The S-units were connected at random to the R-units as well, as that seemed to have been observed in actual biological neurons.

The A-units are connected to the R-units with adjustable weights. They were adjustable resistors (potentiometers) controlled by small electrical motors. It was the turning of hundreds of these motors, making adjustments to the weights, that made up the learning phase of the system. Due to this weight update paradigm, Rosenblatt proposed the R-system serves a memory function of the machine.

The R-system was the output system, usually containing a smaller number of units. It served two purposes: displaying the result in some way (e.g. lights), and feeding back impulses to the A-system with the intent of learning the correct weights. Rosenblatt [16] called this feedback paradigm the ”back-propagating error correction”.

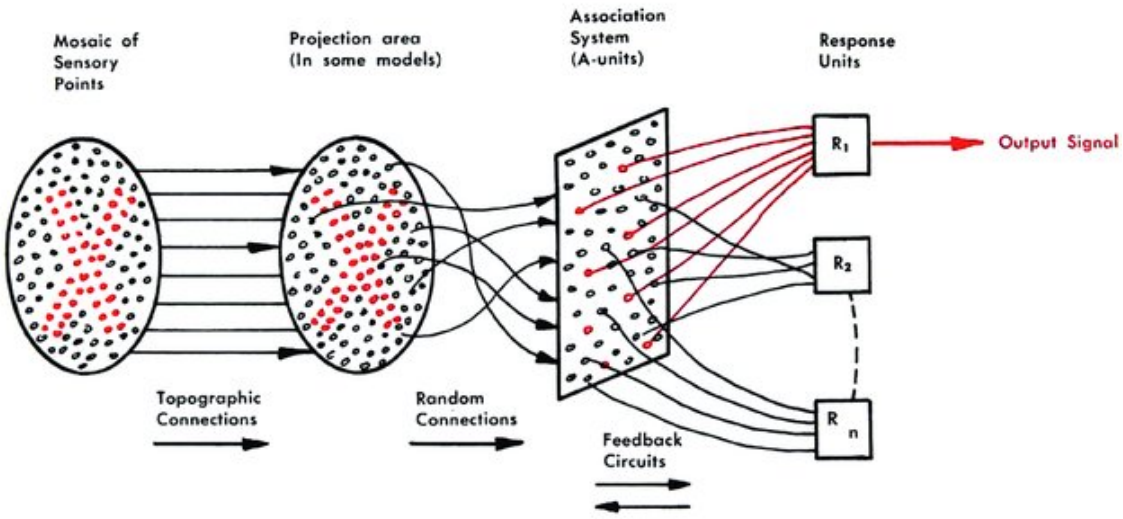


Figure 1.6: Rosenblatt’s photo-perceptron architecture

Rosenblatt experimented with many different perceptron models and called this three-layered perceptron network the *alpha-perceptron*. The first implementation was in software for the IBM 704 computer, but was subsequently implemented in custom-built hardware in 1957 called the ”Mark I Perceptron”. In today’s terms, the alpha-perceptron would closely resemble a feed-forward three-layer neural network with two *hidden layers*. We refer to all layers between the input and the output layers as hidden layers. The feed-forward type of neural network architecture is characterised by data flowing from the input to the output through multiple in-between layers in a sequential matter.

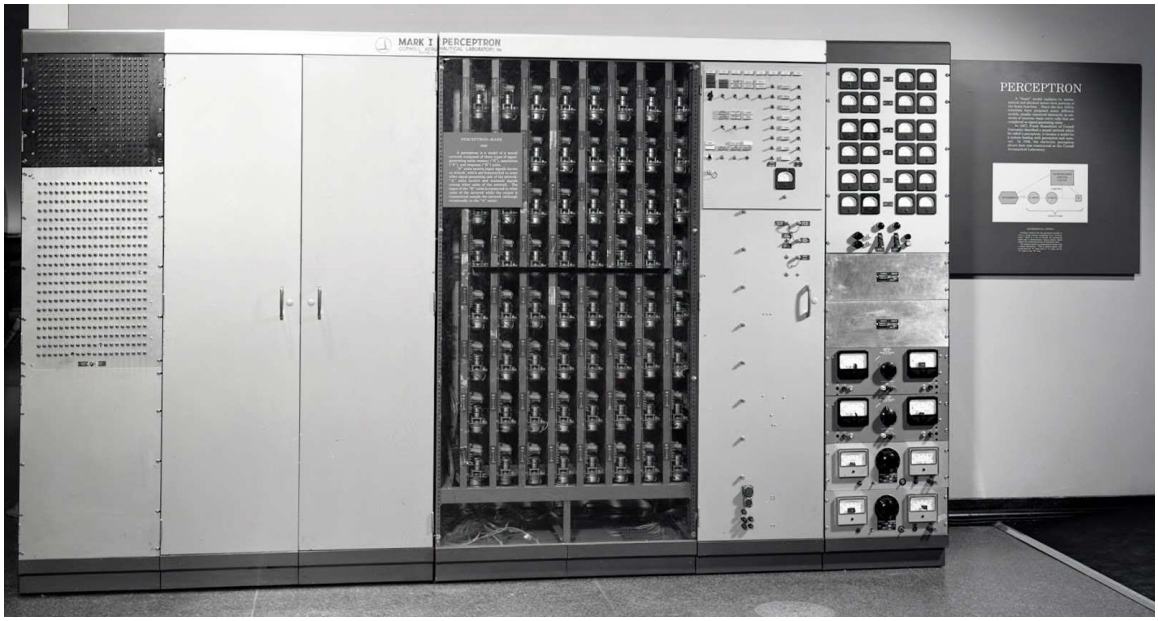


Figure 1.7: Mark I Perceptron displayed at the Smithsonian musem

Solving the XOR problem

Let's show how a simple extension of the M-P neuron can solve the XOR problem. We will construct a multilayered neural network capable of implementing the boolean *XOR* function with two inputs. It will be highly motivated by Rosenblatt's alpha-perceptron, but still within the M-P neuron's restrictions.

The *XOR* function can be written as a composition of *AND*, *OR*, and *NOT* functions.

Definition 1.4.1. Boolean function $NOT : \{0,1\} \longrightarrow \{0,1\}$ is defined as

$$NOT(x) = \begin{cases} 1, & x = 0 \\ 0, & x = 1 \end{cases}.$$

We can also write it as $NOT(x) = 1 - x$, and its implementation with the M-P neuron is immediately clear. By setting $W = [w_1] = -1$, and $\theta = -1$ we obtain the solution.

It follows from mathematical logic that for statements A and B, their *XOR* function can be written as $A \oplus B = \overline{A}B \vee A\overline{B}$, where $\oplus$ stands for the *XOR* function, multiplication for the *AND* function, $\vee$ for the *OR* function, and $\overline{(\cdot)}$ for the *NOT* function. Rewriting it to fit our setting we get

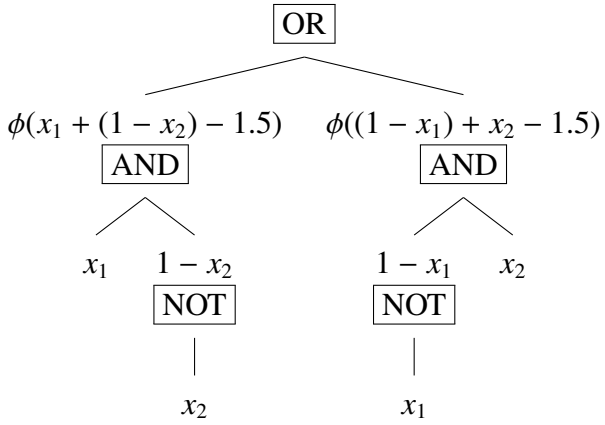
$$\begin{aligned} XOR(x_1, x_2) &= OR(AND(x_1, NOT(x_2)), AND(x_2)) \\ &= OR \circ (AND, AND) \circ ((x_1, NOT(x_2)), (NOT(x_1), x_2)). \end{aligned} \tag{1.1}$$

From Section 1.3 we know how to implement the *AND* and the *OR* functions. For the threshold activation function ϕ we have:

$$\begin{aligned} AND(x_1, x_2) &= \phi(x_1 + x_2 - 1.5) \\ OR(x_1, x_2) &= \phi(x_1 + x_2 - 0.5) \\ NOT(x_1) &= 1 - x_1. \end{aligned} \tag{1.2}$$

The solution combines equations (1.1) and (1.2), and is shown in a tree-like structure here:

$$\phi(\phi(x_1 + (1 - x_2) - 1.5) + \phi((1 - x_1) + x_2 - 1.5) - 0.5) \tag{1.3}$$



Some expressions inside the activation functions can be simplified, leading to desired weights and threshold parameters in the neural network. The tree structure from above rotated 90° clockwise would give us an ANN-like structure. Figure 1.8 shows a generalised version of such a neural network.

When constructing a neural network, we keep track of all parameters in vectors and matrices. Regarding the network from Figure 1.8, the first *weight matrix*, containing weights connecting the input layer to the hidden layer, is $W_1 \in \{-1, 1\}^{2\times 2}$. Matrix $W_2 \in \{-1, 1\}^{2\times 1}$ contains weights connecting the hidden layer to the output layer, and the threshold vectors $\theta_1 \in \mathbb{R}^{2\times 1}$, $\theta_2 \in \mathbb{R}$ contain the threshold parameters for the first and second layer respectively. Finally, we mark the vector of intermediate values calculated by the hidden layer with $H \in \{0, 1\}^{2\times 1}$, the input vector with $X \in \mathbb{R}^n$, and the output with $y \in \mathbb{R}$.

Vectors and matrices are of the following form:

$$X = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix},\ H = \begin{bmatrix} h_1 \\ h_2 \end{bmatrix},\ W_1 = \begin{bmatrix} w_{11}^{(1)} & w_{12}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} \end{bmatrix},\ W_2 = \begin{bmatrix} w_{11}^{(2)} \\ w_{21}^{(2)} \end{bmatrix},\ \theta_1 = \begin{bmatrix} \theta_1^{(1)} \\ \theta_2^{(1)} \end{bmatrix},\ \theta_2 = \begin{bmatrix} \theta_1^{(2)} \end{bmatrix}.$$

Note 1.4.2. We extend the definition of a real function $f : \mathbb{R} \longrightarrow \mathbb{R}$ to matrices by letting $f(M) = [f(m_{ij})] \in \mathbb{R}^{m\times n}$ for any matrix $M = [m_{ij}] \in \mathbb{R}^{m\times n}$.

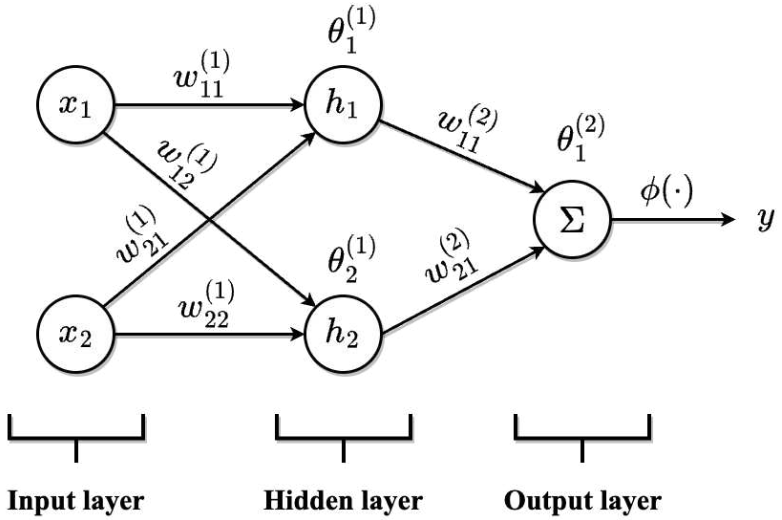


Figure 1.8: Schematics of a two-layer neural network

Values calculated by the hidden layer and the output layer are:

$$\begin{aligned} H &= \phi(W_1^T X - \theta_1), \\ y &= \phi(W_2^T H - \theta_2) \\ &= \phi(W_2^T \phi(W_2^T X - \theta_1) - \theta_2). \end{aligned} \tag{1.4}$$

Note 1.4.3. Matrix multiplication WX is more often seen in the literature instead of $W^T X$, along with indexing the weight matrices by $W^{(l)}$, instead of W_l used here due to it being transposed. In that case, the weight of the connection of the i-th neuron in the l-th layer and the j-th neuron in the $(l+1)$-th layer be represented by $w_{ji}^{(l)}$, instead of the current $w_{ij}^{(l)}$.

Finally, setting the weights and thresholds to the following values gives us the solution to the XOR problem:

$$W_1 = \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}, \; W_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \; \theta_1 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}, \; \theta_2 = \begin{bmatrix} 0.5 \end{bmatrix}.$$

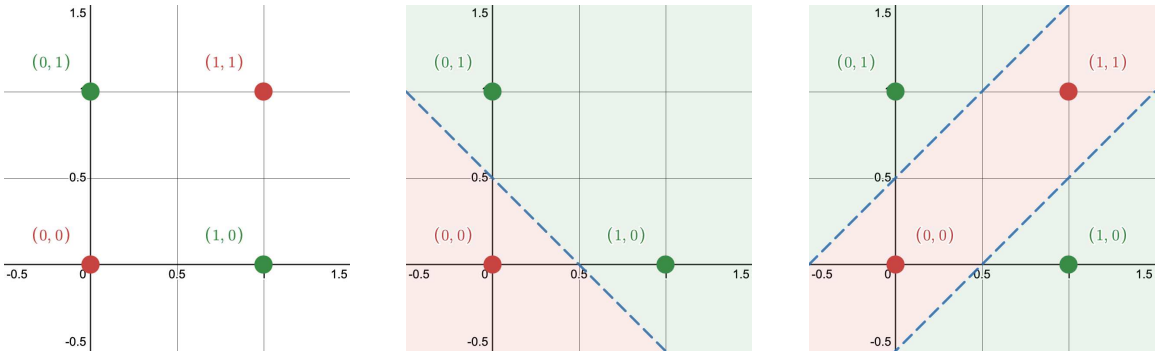
We'll use the logic truth table to check the results by evaluating all possible values of x_1 and x_2 on the equations given by (1.3), going from the bottom to the top. The top node of the tree is essentially the neural network y from relation (1.4).

Looking at the last row of the h_1 and h_2 columns in Table 1.2, we notice how the original pair $(x_1, x_2) = (1, 1)$ was transformed into $(h_1, h_2) = (0, 0)$ by the hidden layer. This in-between step allowed us to transform the original dataset into a linearly separable one, shown in Figure 1.9b.

x_1	x_2	$x_1 - x_2 - 0.5$	$x_2 - x_1 - 0.5$	h_1	h_2	$h_1 + h_2 - 0.5$	y	$XOR(x_1, x_2)$
0	0	-0.5	-0.5	0	0	-0.5	0	0
0	1	-1.5	0.5	0	1	0.5	1	1
1	0	0.5	-1.5	1	0	0.5	1	1
1	1	-0.5	-0.5	0	0	-0.5	0	0

Table 1.2: *XOR* truth table with intermediate parameter calculation

Figure 1.9c shows the decision regions of the original dataset, the untransformed *XOR* function. The decision regions look a bit odd, but this is due to the restrictions on weights in the M-P neuron model.



(a) *XOR* function (b) *XOR* transformed (c) *XOR* decision regions

Any logic function can be implemented using a network of M-P neurons. This is because we can implement the universal *NAND* function, which is a negated *AND* function. The *NAND* function possesses a property called the *functional completeness*, meaning that any boolean function can be implemented using only a composition of *NAND* functions.

Minsky and Papert knew that multiple layers would be able to solve the XOR problem, even suggesting "...a universal computer could be built entirely out of linear threshold modules." in their 1969 book *Perceptrons* [13]. Why was their book such devastation to AI research, contributing to the first AI winter and the *"end of connectionism"*? They had conjectured, based on their "intuitive judgment" [13, p. 232], that extensions of the perceptron architecture would be subject to limitations similar to those suffered by one-layer perceptrons. Another major concern at the time was the lack of efficient training algorithms for multi-layered networks. This consequently led to disbelief in the capabilities of even the multilayered ANNs, which were known to be able to solve more complicated problems, as it seemed they couldn't be trained.

Perceptron Learning Algorithm

The earliest learning algorithm was the *perceptron learning algorithm*, proposed by Rosenblatt. It wasn't presented in terms of optimization, but the goal was always to minimize the number of misclassifications.

Figure 1.3 shows the basic structure of the perceptron. Using a common *feature engineering trick*, we can remove the parameter θ by introducing an additional *bias neuron*. It has a fixed output value of 1, and the weight w_{n+1} that scales it will take the place of θ. In many settings, θ is called the *bias*. Generally, we won't distinguish this added neuron and the weight w_{n+1} from others in any particular way, and the term "bias" won't be explicitly used. Figure 1.10 shows the schematics. Additionally, we'll denote the output of the perceptron with $\hat{y}$ as it's no longer used for function implementation, but rather a prediction of the true output value y.

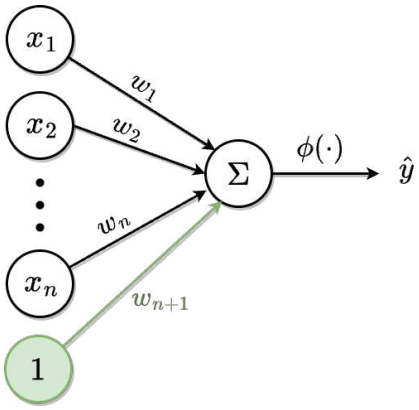


Figure 1.10: Perceptron with added bias neuron

Definition 1.4.4. Let ϕ be the threshold activation function, $n \in \mathbb{N}$ any natural number, $\mathcal{X}, \Omega \subseteq \mathbb{R}^n$ an input space and the set of network states respectively, and $\mathcal{Y} = \{0, 1\}$ the output space. Function $\hat{y} : \Omega \times \mathcal{X} \longrightarrow \mathcal{Y}$ defined as

$$\hat{y}(W, X) = \phi\left(W^T X\right)$$

is called the *perceptron*.

Definition 1.4.5. Let $\mathcal{X}$ be an input space and $\mathcal{Y}$ an output space. Any finite subset $\mathcal{T} \subseteq \mathcal{X} \times \mathcal{Y}$ is called a *training set*.

For $\mathcal{X} \subseteq \mathbb{R}^n$ and $\mathcal{Y} = \{0, 1\}$, we say that a training set $\mathcal{T} \subseteq \mathcal{X} \times \mathcal{Y}$ is linearly separable in $\mathbb{R}^n$ if sets $\mathcal{T}_0$ and $\mathcal{T}_1$ are linearly separable in $\mathbb{R}^n$, where

$$\mathcal{T}_0 = \{X \in \mathbb{R}^n : (X, 0) \in \mathcal{T}\}, \text{ and } \mathcal{T}_1 = \{X \in \mathbb{R}^n : (X, 1) \in \mathcal{T}\}.$$

Let $\mathcal{T} = \{(X_1, y_1), (X_2, y_2), \ldots, (X_N, y_N)\}$ be a training set. Our goal is to minimize the number of misclassifications made by the perceptron $\hat{y}$ on the training set $\mathcal{T}$. Let $\mathcal{M}(W) \subseteq \mathcal{T}$ denote the set of misclassified input-output pairs made by the perceptron in state $W \in \Omega$. Assuming the perceptron misclassified some input-output pair $(X_i, y_i) \in \mathcal{T}$ using the weight matrix W (meaning that $\phi(W^T X_i) = \hat{y}_i \neq y_i$), there are two possibilities:

$$1.\ W^T X_i \geq 0 \implies \hat{y}_i = 1 \text{ and } y_i = 0,$$

$$2.\ W^T X_i < 0 \implies \hat{y}_i = 0 \text{ and } y_i = 1.$$

Denoting the *error of i-the prediction* by $e_i = y_i - \hat{y}_i$, we observe that $e_i \cdot \operatorname{sgn}(W^T X_i)$ is always a negative integer in case of misclassification.[4] The number of misclassifications $|\mathcal{M}(W)|$ is given by

$$|\mathcal{M}(W)| = -\sum_{\mathcal{T}} e_i \cdot \operatorname{sgn}(W^T X_i).$$

We are looking for an optimal weight matrix W_* that minimizes the number of misclassifications $|\mathcal{M}(W)|$. It should possess the following property:

$$W_* \in \arg\min_{W \in \Omega} |\mathcal{M}(W)|,$$

where $\arg\min(\cdot)$ represents the set of all optimal matrices, as it is not necessarily unique.

Since the sign function is not especially optimizable, we aim to find a differentiable function in terms of W that optimizes the same problem. By omitting the sign function entirely, we are also minimizing the number of misclassifications. This is because by minimizing $-e_i W^T X_i$ we minimize its sign as well, and in case of misclassification we have $\operatorname{sgn}(-e_i W^T X_i) = -e_i \operatorname{sgn}(W^T X_i) = 1$. The task now is finding W_* such that

$$W_* \in \arg\min_{W \in \Omega} \sum_{\mathcal{T}} -e_i W^T X_i\,. \tag{1.5}$$

Relation (1.5) is called the *perceptron criterion*.

Optimization is achieved through *epochs*. Each time an algorithm passes through the entire training set, it's said to have completed an epoch. It can be thought of as an instance of a for-loop moving through the entire dataset. The perceptron learning algorithm doesn't minimize the sum from (1.5) directly, but rather goes through the entire training set in one epoch, and upon encountering a misclassification, updates the weight matrix by following some *weight update rule*. Assuming that at some moment $k \in \mathbb{N}$, the perceptron in state W_k misclassifies X_i, we wish to update the weight matrix W_k with the intent of minimizing the function $f(W) = -e_i W^T X_i$. In case the weight matrix has been updated in the current epoch, the process repeats until no misclassifications are made in the entire epoch.

[4] For our purposes, $\operatorname{sgn}(x) = 1$ for $x \geq 0$, and $\operatorname{sgn}(x) = -1$ for $x < 0$.

The weight update rule is simply a *gradient descent* method for finding a minimum of a function. The gradient of a function is the direction of the quickest ascend, and by moving in the opposite direction by some step $\eta > 0$, called the *learning rate*, we're hoping for the quickest descent to a minimum. Iterating this process gives the gradient descent algorithm, and for some differentiable real or vector function f, the iterations are given by

$$x_{k+1} = x_k - \eta \cdot \nabla f(x_k)\,.$$

While dependent on W, the error of i-the prediction e_i only serves as a normalizing constant, and it will be considered as such when differentiating. For some fixed constants $e_i \in \mathbb{R}$ and $X_i \in \mathbb{R}^n$, function $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ defined as $f(W) = -e_i W^T X_i$ satisfies

$$\nabla f(W) = -e_i X_i\,.$$

Let W_k represent the vector of weights after k updates. Then, upon encountering the misclassified pair $(X_i, y_i) \in \mathcal{T}$, the weights are updated by the following rule:

$$\begin{aligned} W_{k+1} &= W_k - \eta(-e_i X_i) \\ &= W_k + \eta e_i X_i\,. \end{aligned}$$

Let $\mathcal{M}_k \subseteq \mathcal{T}$ denote all misclassified data points at some moment k, made by the perceptron $\hat{y}$ in state W_k. We'll use this $\mathcal{M}_k$ notation to simplify the algorithm in writing.

Algorithm 1: The perceptron learning algorithm

Input: Training set $\mathcal{T}$, Learning rate η
Initialize: $k = 0$, $\mathcal{M}_0 = \mathcal{T}$, $W_0 = 0 \cdot \mathbb{1}_n$
Algorithm:
while $|\mathcal{M}_k| > 0$ **do**
  Pick $(X_i, y_i) \in \mathcal{M}_k$ *at random ;*
  $W_{k+1} = W_k + \eta e_i X_i$ *;*
  $\mathcal{M}_{k+1} = \{(X_i, y_i) \in \mathcal{T} : e_i W_{k+1}^T X_i < 0\}$ *;*
  $k \leftarrow k + 1$ *;*
end
$W_* \leftarrow W_k$;
$k_* \leftarrow k$;
Output: W_* , k_*

Number $k_* \in \mathbb{N}$ is the number of weight updates required before optimal W_* is found. Such W_* clearly satisfies the perceptron criterion (1.5), as $|\mathcal{M}(W_*)| = 0$. We'll show how the perceptron learning algorithm terminates in the case of a linearly separable training set, and specify an upper bound on the required number of iterations.

Reminder. The Cauchy-Schwartz inequality states that $X^T Y \leq \|X\| \, \|Y\|$, for all $X, Y \in \mathbb{R}^n$.

Theorem 1.4.6 (Novikoff)**.** *Let $\mathcal{T} = \{(X_1, y_1), \ldots, (X_N, y_N)\}$ be linearly separable training set, and W a unit vector such that $\phi(W^T X_i) = y_i$ for all $i = 1, 2, \ldots, N$. Let $R = \max_i \|X_i\|$, and $\gamma = \min_i |W^T X_i|$. Then the perceptron learning algorithm outputs $k_* \leq R^2/\gamma^2$.*

Proof. Let $W_1, W_2, \ldots, W_k$ be a sequence of iterates generated by the Algorithm 1, and (X_i, y_i) the training pair from the k-th update $W_k = W_{k-1} + \eta e_i X_i$. Then,

$$W^T W_k = W^T W_{k-1} + \eta e_i W^T X_i \Rightarrow$$
$$\{e_i W^T X_i \geq \gamma \text{ by assumption, since } 0 \leq e_i W^T X_i = |W^T X_i|\} \Rightarrow$$
$$W^T W_k \geq W^T W_{k-1} + \eta\gamma \Rightarrow$$
$$\{\text{recursion and } W_0 = 0\} \Rightarrow$$
$$W^T W_k \geq k\eta\gamma.$$

Using the Cauchy-Schwartz inequality we get

$$\|W\| \, \|W_k\| \geq W^T W_k \geq k\eta\gamma \Rightarrow$$
$$\{\|W\| = 1 \text{ by assumption}\} \Rightarrow$$
$$\|W_k\| \geq k\eta\gamma \Rightarrow$$
$$\|W_k\|^2 \geq k^2\eta^2\gamma^2. \tag{1.6}$$

On the other hand,

$$\|W_k\|^2 = W_k^T W_k = (W_{k-1} + \eta e_i X_i)^T (W_{k-1} + \eta e_i X_i) \Rightarrow$$
$$\|W_k\|^2 = \|W_{k-1}\|^2 + 2\eta e_i W_{k-1}^T X_i + \eta^2 e_i^2 \|X_i\|^2 \Rightarrow$$
$$\{e_i W_{k-1}^T X_i \leq 0 \text{ due to misclassification, } e_i^2 = 1, \ \|X_i\|^2 \leq R^2\} \Rightarrow$$
$$\|W_k\|^2 \leq \|W_{k-1}\|^2 + \eta^2 R^2 \Rightarrow$$
$$\{\text{recursion and } W_0 = 0\} \Rightarrow$$
$$\|W_k\|^2 \leq k\eta^2 R^2. \tag{1.7}$$

Combining (1.6) and (1.7) we get

$$k^2\eta^2\gamma^2 \leq \|W_k\|^2 \leq k\eta^2 R^2 \Rightarrow$$
$$k \leq \frac{R^2}{\gamma^2}.$$

□

Algorithm 1 does not terminate for a linearly inseparable training sets, and the weights W_k may exhibit a cyclic behaviour. Lastly, it's worth noting that the learning rate η does not affect the stability of the perceptron learning algorithm, and affects the convergence rate only for nonzero initial weight vector. A common value of η is 0.5 [4].

1.5 Adaptive Linear Neuron Model

Adaptive Linear Neuron, or *ADALINE* (also referred to as the *LMS learning*) is a learning algorithm developed by Bernard Widrow and Ted Hoff at Stanford University in 1959 [19]. It was the first approach to the problem of learning from an optimization point of view. The method was a direct application of linear regression to binary target values.

For a training set $\mathcal{T}$ and $(X_i, y_i) \in \mathcal{T}$, ADALINE uses $\hat{y}(W, X_i) = W^T X_i$ as its prediction of y_i, and works with this linear function through the entire training process. Once the training is finished and optimal weights are found, only then do we apply an activation function to classify the inputs. Meaning, that ADALINE adjusts the weights before applying the activation function, while the perceptron learning algorithm adjusts weights after applying the activation function.

In the perceptron learning algorithm, the errors take values only in $\{-1, 0, 1\}$, while in the case of ADALINE, the errors can be arbitrary real values, suggesting that ADALINE may learn from the "correct" predictions as well. This does come with a downside: the perceptron learning algorithm refrains from penalizing excessively large (absolute) values of $W^T X_i$ since they are equally squashed to 0 or 1 as those near the threshold. This is due to the "all-or-none" behaviour of the threshold activation function. On the other hand, using real-valued predictions will inappropriately penalize such points of over-performance.

ADALINE learns by minimizing a *cost function* rather than the number of misclassifications. The cost function measures the error made by the prediction model on a given set of examples, and the most commonly seen one is the *Mean Squared Error*. While often synonymous, in this thesis, we differentiate between a cost function and a *loss function*.

Definition 1.5.1. Any non-negative Borel measurable function $L : \mathbb{R}^2 \longrightarrow \mathbb{R}$ is called a *loss function*.

Definition 1.5.2. Let $\mathcal{X}$ be an input space, $\mathcal{Y}$ an output space, Ω the set of network states, $N \in \mathbb{N}$, and L a loss function. For any (parametric) prediction $\hat{y} : \Omega \times \mathcal{X} \longrightarrow \mathbb{R}$ of $y \in \mathcal{Y}$, function $C : \Omega \times (\mathcal{X} \times \mathcal{Y})^N \longrightarrow \mathbb{R}$ defined as

$$C(W, \mathcal{T}) = \frac{1}{|\mathcal{T}|} \sum_{(X,y)\in\mathcal{T}} L(y, \hat{y}(W, X))$$

is called a *cost function*.

Remark 1.5.3. Regarding the above definition, the training set $\mathcal{T}$ of size N is considered as an element of $(\mathcal{X} \times \mathcal{Y})^N$ to satisfy the definition of a cost function. Additionally, we'll denote the cost function by $C_{\mathcal{T}}(W)$ when differentiating with respect to W.

Loss function L defined by $L(y, \hat{y}) = (y - \hat{y})^2$ is called the *squared loss*, and the cost function associated with the squared loss is the *Mean Square Error* or the MSE cost function.

Ideally, we would minimize the cost function using a gradient descent method, however, it may be too computationally demanding. The learning algorithm would need to compute the prediction for every input pattern in the training set, and then average the squared error for all training instances in each iteration. This is likely infeasible for larger training sets and a different method is needed.

One approach is to split the training set into the so-called *mini-batches* by partitioning it into equally sized subsets. In a given epoch, the training algorithm will go through the training set mini-batch by mini-batch, and update the weights based on the gradient of the cost function evaluated on a mini-batch instead of the entire training set. The training set is randomly partitioned into mini-batches at the beginning of each epoch. The *mini-batch size*, denoted by b, is an important *hyperparameter* in learning methods. Due to the architectures of processors, mini-batch size is often picked as a power of 2, with recommendations[5] of not going too large and staying somewhere in the $b = 32$ range [11].

Note 1.5.4. If it's not possible to partition the training set into equally sized parts, either one subset will be of a different size, or the training set will be randomly resized in each epoch. For this reason, we'll assume that the size of the training set is always divisible by the mini-match size b.

The approach described is called the *mini-batch learning* or *mini-batch gradient descent*. In case the entire training set $\mathcal{T}$ is used for the cost function calculation, meaning that $b = |\mathcal{T}|$ and the "mini-batch" is the entire training set, the method is called the *batch learning* or the *batch gradient descent*. Another special case is $b = 1$, in which case the method is called the *stochastic gradient descent* or *SGD*. SGD is stochastic because the mini-batches are always chosen randomly, meaning that in every epoch the algorithm goes through the training set randomly instead of deterministically.

For training set $\mathcal{T}$, a cost function C, and a b-sized mini-batch $\mathcal{T}_{(b)}$ of $\mathcal{T}$, the mini-batch gradient descent method with a learning rate $\eta > 0$ gives the weight update rule as

$$W_{k+1} = W_k - \eta \cdot \nabla C_{\mathcal{T}_{(b)}}(W_k), \quad k \geq 0.$$

The loss function will often be denoted by L_W when evaluating a parametric prediction $\hat{y}$ of y given the state W. Due to the linearity of a differential and the definition of a cost function, we have the following relation:

$$\nabla C_{\mathcal{T}_{(b)}}(W) = \frac{1}{|\mathcal{T}_{(b)}|} \sum_{(X,y)\in\mathcal{T}_{(b)}} \nabla L_W(y, \hat{y}(W, X)).$$

For this reason, it's sufficient to calculate $\nabla L_W(y, \hat{y}(W, X))$ when deriving a mini-batch gradient descent algorithm for any particular cost function.

[5] Yann LeCun: "Training with large minibatches is bad for your health. More importantly, it's bad for your test error. Friends don't let friends use mini-batches larger than 32." - `https://twitter.com/ylecun/status/989610208497360896?lang=en` (visited on 16.04.2024).

ADALINE Learning Algorithm

Widrow and Hoff's ADALINE learning algorithm is the SGD method with the MSE cost function and a linear prediction $\hat{y}(W, X) = W^T X$ of an output y. For each input-output pair $(X_i, y_i) \in \mathcal{T} = \{(X_1, y_1), (X_2, y_2), \ldots, (X_N, y_N)\}$, we'll denote the prediction $\hat{y}(W, X_i)$ of y_i by $\hat{y}_i(W) = W^T X_i$. For the squared loss L_w we denote the *loss at the i-th example* by $L_i(W) = L_w(y_i, \hat{y}_i) = (y_i - \hat{y}_i(W))^2 = (y_i - W^T X_i)^2$. Then,

$$\mathrm{MSE}_{\mathcal{T}}(W) = \frac{1}{|\mathcal{T}|} \sum_{(X,y)\in\mathcal{T}} L_w(y, \hat{y}(W, X)) = \frac{1}{N} \sum_{i=1}^{N} L_i(W)\,. \tag{1.8}$$

For one-sized mini-batch $\mathcal{T}_{(1)} = \{(X_i, y_i)\}$ we have

$$\begin{aligned} \mathrm{MSE}_{\mathcal{T}_{(1)}}(W) &= \frac{1}{|\mathcal{T}_{(1)}|} \sum_{(X,y)\in\mathcal{T}_{(1)}} (y - \hat{y}(W, X))^2 \\ &= (y_i - \hat{y}_i(W))^2 = (y_i - W^T X_i)^2 \\ &= L_i(W), \end{aligned}$$

and

$$\nabla\,\mathrm{MSE}_{\mathcal{T}_{(1)}}(W) = \nabla L_i(W) = -2(y_i - W^T X_i) X_i\,.$$

Ignoring the constant 2, as it is multiplied by the learning rate η, a single instance of the SGD weight update rule is given by

$$W_{k+1} = W_k + \eta(y_i - W_k^T X_i) X_i\,.$$

For a sufficiently small learning rate η_{SGD}, it's possible to show that SGD minimizes the global MSE error of batch learning, and if the learning rate η_{batch} is also sufficiently small, SGD approaches batch learning and the two methods produce the same result [4].

The *test error* is a measure of an error made by the prediction model on an independent set of input-output pairs called the *test set*, denoted by $\mathcal{T}_{test}$. As the test set is not used during the model's training phase, it provides a better metric for assessing how well a model generalizes to unseen data. The value of the test error is the cost function evaluated on a test set, and similarly, the cost function evaluated on the training set is called the *training error*. While the training error will always decrease in gradient descent methods, the test error will decrease to a minimum and then begin to rise again, since after a certain point the network is being *overtrained*.

A procedure called the *early stopping rule* can deal with the issue of ovetraining (or *overfitting*), and is therefore a form of *regularization*. It states that the learning algorithm should halt once the test error increases or does not significantly decrease for s epochs. The non-significant decrease of the test error is measured by a relative difference between test

errors of the two successive epochs. It's denoted by r, and its insignificance means that r is smaller than some *relative error lower bound* r^*. The ADALINE learning algorithm will terminate in case a maximum number of epochs k_{max} is reached, or by an early stopping rule which does not select the last parameter just before the algorithm terminates, but rather the one with the lowest test error (ignoring those among the final s epochs that might have caused an insignificant decrease in the test error).

Algorithm 2: The ADALINE Learning Algorithm

Input: Training set $\mathcal{T} = \{(X_1, y_1), (X_2, y_1), \ldots, (X_N, y_N)\}$, Test set $\mathcal{T}_{test}$,
Learning rate η, Maximum number of epochs k_{max},
Relative error lower bound r^*, Early stopping parameter s

Initialize: $k^* = k = j = 0$, $W_0 = RAND$, $\varepsilon_0 = \text{MSE}(W_0, \mathcal{T}_{test})$, $r_0 = r^* + 1$

Algorithm:

while $k < k_{max}$ *and* $j < s$ **do**

  Randomly shuffle $\mathcal{T} \Rightarrow \mathcal{T} = \{(X_{(1)}, y_{(1)}), (X_{(2)}, y_{(2)}), \ldots, (X_{(N)}, y_{(N)})\}$;

  for $i = 1, 2, \ldots, N$ **do**

    $W_{Nk+i} = W_{Nk+i-1} + \eta(y_{(i)} - W_{Nk+i-1}^T X_{(i)}) X_{(i)}$;

  end

  $k \leftarrow k + 1$;

  $\varepsilon_k = \text{MSE}(W_{Nk+i}, \mathcal{T}_{test})$;

  $d_k = \varepsilon_{k^*} - \varepsilon_k$;

  $r_k = |d_k| / \varepsilon_k$;

  if $d_k \leq 0$ *or* $r_k < r^*$ **then** `/* No significant improvement */`

    $j \leftarrow j + 1$;

    continue ; `/* Move on to the next epoch */`

  else

    $k^* \leftarrow k$; `/* Memorize the new best epoch */`

    $W_* \leftarrow W_{Nk}$;

    $j \leftarrow 0$; `/* Reset the early stopping parameter */`

  end

end

Output: W_*

Using the learning rate $0 < \eta < 2/\max_i \|X_i\|^2$ ensures convergence. The *α-LMS learning rule* is a modification to the ADALINE (LMS learning rule) obtained by normalizing the input vector so that the weights change independently of its magnitude:

$$W_{k+1} = W_k + \eta(y_i - W_k^T X_i) \frac{X_i}{\|X_i\|} .$$

For the convergence of the α-LMS rule, η should be selected as $0 < \eta < 2$ [20].

The number $W_*^T X_i$ is interpreted as the probability of X_i belonging to class 1, and in case it's larger than 0.5, the output is predicted as 1, and 0 otherwise. The class prediction of y_i is therefore given by $\phi(W_*^T X_i - 0.5)$. This 0.5 threshold isn't fixed, and can be adjusted for various reasons. The probability interpretation is somewhat misleading, as the values of $W_*^T X_i$ are not strictly in $[0, 1]$. Of course, logistic regression was known at the time and would be a more appropriate method to use in this situation, but Algorithm 2 is what Widrow[6] and Hoff implemented in their machine at the time. Their main contribution wasn't the model itself, but rather an optimization approach to the problem of learning, and ADALINE had a major impact on the field of adaptive signal processing.

1.6 Multilayer Perceptron

The perceptrons presented in Section 1.4 are types of a *single-layer perceptron*. In defining the single-layer perceptron we allow arbitrary activation function. An important nonlinear activation function often used is the *sigmoid activation function*.

Definition 1.6.1. Let Φ be any activation function, $n \in \mathbb{N}$ any natural number, $\mathcal{X}, \Omega \subseteq \mathbb{R}^n$ an input space and the set of network states respectively, and $\mathcal{Y} \subseteq \mathbb{R}$ an output space. Function $\hat{y} : \Omega \times \mathcal{X} \longrightarrow \mathcal{Y}$ defined as

$$\hat{y}(W, X) = \Phi(W^T X)$$

is called a *Single-Layer Perceptron* or *SLP*.

Definition 1.6.2. For any $\beta > 0$ we define a *sigmoid activation function* (or *logistic activation function*) $\sigma_\beta : \mathbb{R} \longrightarrow [0, 1]$ as

$$\sigma_\beta(x) = \frac{1}{1 + e^{-\beta x}}.$$

For $\beta = 1$, function σ_1 is denoted by σ.

The threshold activation function can be thought of as a discrete estimate of the sigmoid activation function. In fact, both functions belong to the same class of activation functions called the *sigmoidal activation functions*.

Definition 1.6.3. Any non-decreasing function $\Psi : \mathbb{R} \longrightarrow [0, 1]$ with the property

$$\lim_{x \to -\infty} \Psi(x) = 0, \qquad \lim_{x \to +\infty} \Psi(x) = 1$$

is called a *sigmoidal function*.

[6] Video lectures and demonstration of the ADALINE by Bernard Widrow: `https://www.youtube.com/watch?v=hc2Zj55j1zU` , `https://www.youtube.com/watch?v=skfNlwEbqck` (visited on 16.04.2024).

Remark 1.6.4. Sigmoidal functions are measurable as non-decreasing functions, and are therefore activation functions. We'll refer to any activation function with a sigmoidal property as a *sigmoidal activation function*.

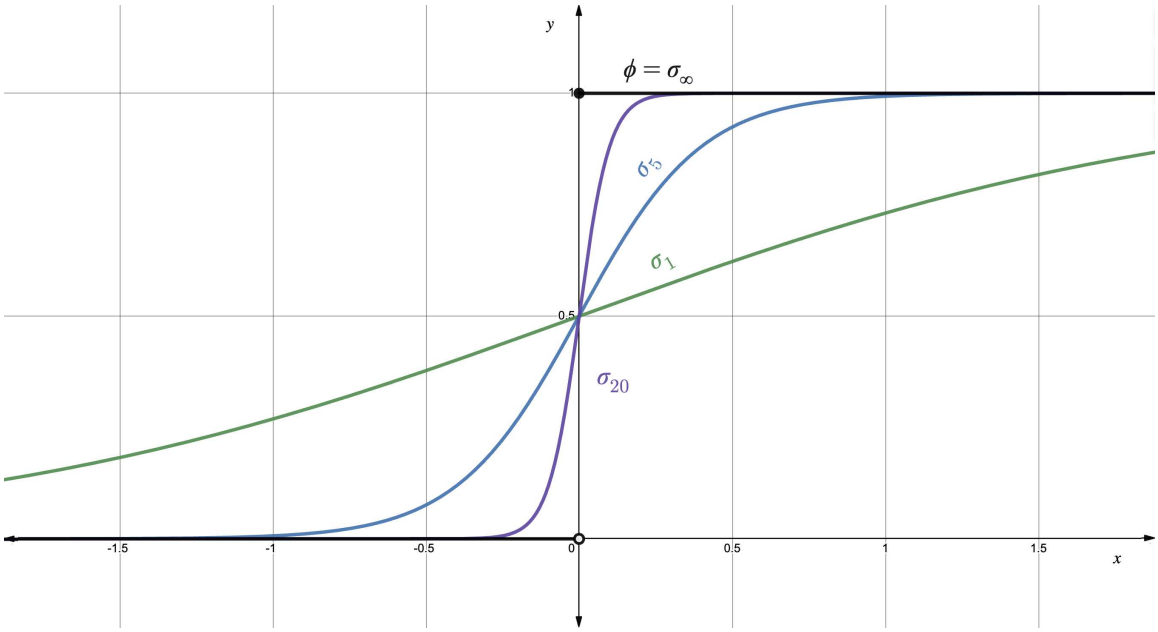


Figure 1.11: Sigmoidal functions σ_1, σ_5, σ_{20}, and the threshold activation function ϕ

Multilayer neural networks contain more than one computational layer. The additional intermediate layers between the input and the output layer are referred to as *hidden layers*. The *Multilayer perceptron* combines SLPs in the same way the M-P neural network in Section 1.4, constructed to solve the XOR problem, combines multiple M-P neurons. All adjacent layers are fully connected one to another, meaning that every neuron in a given layer connects to all neurons of the following layer. Additionally, all neurons in the same layer use the same activation function.

Let's assume there are k hidden layers in the multilayer perceptron, and the l-th hidden layer contains J_l neurons and uses Φ_l as its activation function. The weights connecting the input layer with n neurons to the first hidden layer are contained in matrix $W_1 \in \mathbb{R}^{n \times J_1}$, whereas the weights connecting the l-th to the $(l+1)$-th hidden layer are contained in matrix $W_{l+1} \in \mathbb{R}^{J_l \times J_{l+1}}$. The output layer contains only a single neuron and uses Φ_{k+1} as its activation function. The final matrix $W_{k+1} \in \mathbb{R}^{J_k \times 1}$ contains the weights connecting the last J_k-th hidden layer to the output layer. We'll use a vector representation to denote the values outputted by the l-th hidden layer and denote it by $H_l = \Phi_l(W_l^T H_{l-1}) = [h_1^{(l)}, h_2^{(l)}, \ldots, h_{J_l}^{(l)}]^T$, where $l = 1, 2, \ldots, k$. Definition of the l-th hidden layer may be extended to $l = 0$ by setting $H_0 = X$, where X is the input vector. This will prove useful for various definitions.

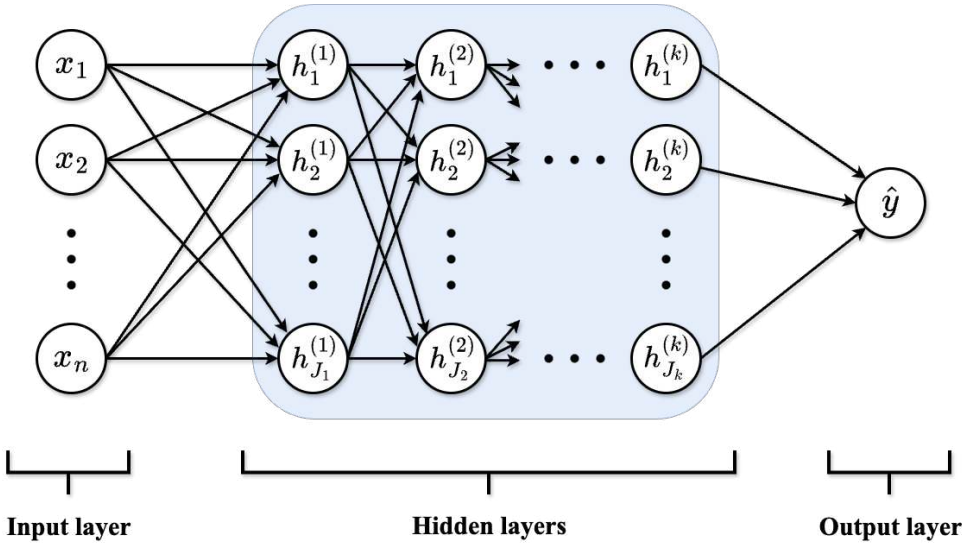


Figure 1.12: Schematics of the Multilayer Perceptron

Definition 1.6.5 (Multilayer Perceptron)**.** For any $n, k \in \mathbb{N}$, let $\mathcal{X} \subseteq \mathbb{R}^n$ be an input space, $\mathcal{Y} \subseteq \mathbb{R}$ an output space, and let $\Phi_1, \Phi_2, \ldots, \Phi_{k+1}$ be arbitrary activation functions. For $J_1, \ldots, J_k \in \mathbb{N}$, let $\Omega \subseteq \mathbb{R}^{n\times J_1} \times \mathbb{R}^{J_1\times J_2} \times \mathbb{R}^{J_2\times J_3} \times \ldots \times \mathbb{R}^{J_{k-1}\times J_k} \times \mathbb{R}^{J_k\times 1}$ be the set of network states. Function $\hat{y} : \Omega \times \mathcal{X} \longrightarrow \mathcal{Y}$ defined by the following recursion:

$$\begin{aligned} H_1 &= \Phi_1(W_1^T X) \text{ for } W_1 \in \mathbb{R}^{n\times J_1}, \\ H_{l+1} &= \Phi_{l+1}(W_{l+1}^T H_l) \text{ for } W_{l+1} \in \mathbb{R}^{J_l\times J_{l+1}}, \quad l = 1, 2, \ldots, k\,, \\ \hat{y}(\omega, X) &= \Phi_{k+1}(W_{k+1}^T H_k) \text{ for } W_{k+1} \in \mathbb{R}^{J_k\times 1}, \end{aligned}$$

is called the *Multilayer perceptron* or *MLP*.

Remark 1.6.6. The network state ω is given by an array of matrices $(W_1, W_2, \ldots, W_{k+1})$ representing the weights of connections between successive layers. For $l = 1, 2, \ldots, k+1$, vector $Z_l = W_l^T H_{l-1} \in \mathbb{R}^{J_l}$ is called the *l-th vector of pre-activation values*, and vector $H_l = \Phi_l(Z_l) \in \mathbb{R}^{J_l}$ is called the *l-th vector of post-activation values*.

Backpropagation Learning Algorithm

The *Backpropagation Learning Algorithm* (or simply *Backpropagation*) once again uses a gradient descent method to update the network state ω through epochs, with the goal of minimizing a cost function. The name ”backpropagation” is interpreted as follows: after providing an input pattern, the output calculated by the network is compared with a given target and the error is propagated backwards, updating the weights (in backward order) and establishing a closed-loop control system.

Unlike single-layer networks, in the case of multilayer networks, the cost function is a complicated composition of multiple weight matrices composed with activation functions throughout all layers. Backpropagation allows the extension of neural networks to many layers by providing an efficient way of calculating the gradient of the cost function with respect to the weights. Dynamic programming is used to effectively calculate the derivative of the cost function with respect to each weight, and the *Generalised Chain Rule* of differential calculus plays a major role. The Backpropagation Learning Algorithm contains two main phases, referred to as the *forward pass* and *backward pass*.

1. Forward pass: The network state ω is fixed in the forward pass. The input training examples are fed into the neural network, resulting in a forward cascade of computations across the network and the output prediction.

2. Backward pass: The input-output training pairs, and their predictions given the network state ω are fixed, with ω now being variable. The gradients of the cost function with respect to all the different weights are calculated and used to update the weights. Since these gradients are calculated in the backward direction, starting from the output node, this process is referred to as the backward pass.

Note 1.6.7. A differential of a function $f : \mathbb{R}^m \longrightarrow \mathbb{R}^n$ at some point $a \in \mathbb{R}^m$ is most often denoted by $D_a f$ or by $Df(a)$ in literature. However, we will use the following notation:

$$\frac{df}{da} = D_a f.$$

The differential above is not to be confused with the Leibniz-style notation of a derivative in real analysis, in which, the above term would represent a mapping $a \longmapsto f'(a)$.

Theorem 1.6.8 (Generalised Chain Rule [17])**.** *Let $m, n, p \in \mathbb{N}$, and $A \subseteq \mathbb{R}^m$ be any open set such that $f : A \longrightarrow \mathbb{R}^n$ is differentiable in $a \in A$. Let $B \subseteq \mathbb{R}^n$ be an open set, $f(A) \subseteq B$ and $g : B \longrightarrow \mathbb{R}^p$ differentiable function in $f(a) = b \in B$. Then the composition $g \circ f : A \longrightarrow \mathbb{R}^p$ is differentiable in $a \in A$ and*

$$\frac{d(g \circ f)}{da} = \frac{dg(f(a))}{df(a)} \frac{df(a)}{da}.$$

□

Remark 1.6.9. We'll often use a shorthand notation

$$\frac{dg}{df} \frac{df}{da} = \frac{dg(f(a))}{df(a)} \frac{df(a)}{da},$$

dropping the argument $f(a)$ (and subsequently a), whenever they are long compositions of functions themselves and hinder the readability. A major drawback of this notation is

the fact that 'f' is treated both as a point and as a function. However, it allows for a more elegant derivation of backpropagation. In terms of Jacobian matrices, the generalised chain rule states that

$$\nabla(g \circ f)(a) = \nabla g(f(a)) \cdot \nabla f(a) =: \nabla g(f) \cdot \nabla f(a)\,.$$

Informal Overview of Backpropagation

Consider an arbitrary MLP $\hat{y}$ from Definition 1.6.5. Let $K = k + 1$ be the total number of computational layers, and $J_K = 1$. Let L be any differentiable loss function and $\mathcal{T} = \{(X_1, y_1), (X_2, y_2), \ldots, (X_N, y_N)\}$ a training set. Additionally, the output layer will be denoted by H_K (along with $\hat{y}$), and given a state $\omega = (W_1, W_2, \ldots, W_K)$, we can write it as:

$$H_K(\omega, X) = \Phi_K(W_K^T \Phi_{K-1}(W_{K-1}^T \cdots W_2^T \Phi_1(W_1^T X))). \tag{1.9}$$

Let $(X, y) \in \mathcal{T}$ be fixed. In Remark 1.6.6, we've defined the vectors of pre and post1-activation values for all $l =, 2, \ldots, K$ and $H_0 = X$ as $Z_l = W_l^T H_{l-1}$ and $H_l = \Phi_l(Z_l)$. To provide an intuitive overview of the idea behind backpropagation, we will consider each vector Z_l as a function of the state ω and keep the input vector X that generated it in the forward pass fixed. This is what happens in the backward pass of the backpropagation. Specifically, let's consider the value of the weight in W_l at index (i, j) as the only variable in each Z_l and (informally) calculate the differential of H_K at $w_{ij}^{(l)}$. By using the chain rule and the Equation (1.9), we may presume that

$$\frac{dH_K}{dw_{ij}^{(l)}} = \frac{dH_K}{dZ_K}\frac{dZ_K}{dH_{K-1}}\frac{dH_{K-1}}{dZ_{K-1}} \cdots \frac{dH_{l+1}}{dZ_{l+1}}\frac{dZ_{l+1}}{dH_l}\frac{dH_l}{dZ_l}\frac{dZ_l}{dw_{ij}^{(l)}}\,. \tag{1.10}$$

Given a training example (X, y), the *loss function at* (X, y) is the function $L_y : \mathbb{R} \longrightarrow \mathbb{R}$ defined by $L_y(x) = L(y, x)$. We now calculate the differential of $(L \circ H_K)$ at $w_{ij}^{(l)}$ as

$$\frac{d(L \circ H_K)}{dw_{ij}^{(l)}} = \frac{dL}{dH_K}\frac{dH_K}{dw_{ij}^{(l)}} \overset{(1.10)}{=} \frac{dL}{dH_K}\frac{dH_K}{dZ_K}\frac{dZ_K}{dH_{K-1}} \cdots \frac{dZ_{l+1}}{dH_l}\frac{dH_l}{dZ_l}\frac{dZ_l}{dw_{ij}^{(l)}}\,. \tag{1.11}$$

While it is possible to compute this derivative for each weight independently, such an approach is inefficient. Equation (1.11) shows how the beginning part of the expression is constantly repeated, and how it grows as we move to shallower layers (smaller l). This is the key observation that leads us to the idea of calculating the Jacobian of the loss function layer by layer, starting from the output layer and moving backwards toward the input layer, with each new calculation using some prior information from the layers that came before.

Expressions (1.10) and (1.11) are presented in terms of vectors of pre-activation values Z_l and vectors post-activation values and H_l, and such derivatives are not strictly defined. For this reason, we'll represent the MLP as a true function in the language of calculus, rather than a cascade of matrix multiplications. Then, by applying the generalised chain rule to such an MLP, we will obtain the differentials of interest.

Multilayer Perceptron as a Function

To represent an MLP as a function, first, we will define the *pre-activation functions* Z_l for some fixed weight matrices $W_l \in \mathbb{R}^{J_{l-1} \times J_l}$ to mimic l-th vector of pre-activation values given by the matrix multiplication $W_l^T H_{l-1}$. Then, to mimic the l-th vector of post-activation values $\Phi_l(W_l^T H_{l-1})$ for some fixed activation function Φ_l, we define the *post-activation functions* H_l. Their iterative composition will be the MLP from Definition 1.6.5.

Note 1.6.10. There won't be any confusion between the l-th pre-activation function Z_l and the vector of l-th pre-activation values with the same notation Z_l. The vector of l-th pre-activation values will always be associated with some training example (X_p, y_p), and in such case will be denoted by $Z_l^{(p)}$, as it is essentially some matrix multiplying X_p. On the other hand, the l-th pre-activation function Z_l will be defined as a general function, independent of training examples (and dependent only on some fixed matrix W_l).

Definition 1.6.11. Let $m, n \in \mathbb{N}$, and $W \in \mathbb{R}^{m \times n}$. The *pre-activation function given (a matrix) W* is a function $Z_W : \mathbb{R}^m \longrightarrow \mathbb{R}^n$, $Z_W = (z_1^{(W)}, z_2^{(W)}, \ldots, z_n^{(W)})$, where each component $z_i^{(W)} : \mathbb{R}^m \longrightarrow \mathbb{R}$ is defined as

$$z_i^{(W)}(x_1, x_2, \ldots, x_m) = \sum_{k=1}^{m} x_k w_{ki}\,, \qquad i = 1, 2, \ldots, n\,.$$

Proposition 1.6.12. *Let $m, n \in \mathbb{N}$, $W \in \mathbb{R}^{m \times n}$, and $Z_W : \mathbb{R}^m \longrightarrow \mathbb{R}^n$ be any pre-activation function given a matrix W. Then,*

$$\nabla Z_W(x_1, x_2, \ldots, x_n) = W^T \in \mathbb{R}^{n \times m}\,.$$

Proof.

$$\frac{\partial z_i^{(W)}}{\partial x_j} = \frac{\partial}{\partial x_j} \sum_{k=1}^{m} x_k w_{ki} = w_{ji} \implies \nabla Z_W(x_1, x_2, \ldots, x_n) = W^T \in \mathbb{R}^{n \times m}\,.$$

□

To formally define the post-activation functions, we need to use *pojections*.

Definition 1.6.13. Let $n \in \mathbb{N}$ and $i \in \{1, 2, \ldots, n\}$. Function $\pi_i : \mathbb{R}^n \longrightarrow \mathbb{R}$ defined as

$$\pi_i(x_1, x_2, \ldots, x_n) = x_i$$

is called a *projection onto the i-the coordinate*.

Proposition 1.6.14. *For any $n \in \mathbb{N}$ and any $i \in \{1, 2, \ldots, n\}$, let $\pi_i : \mathbb{R}^n \longrightarrow \mathbb{R}$ be a projection onto the i-the coordinate. Then*

$$\frac{\partial \pi_i}{\partial x_j} = \delta_{ij}\,,$$

where δ_{ij} the Kronecker delta function.

Proof.

$$\frac{\partial \pi_i}{\partial x_j} = \frac{\partial}{\partial x_j}\pi_i(x_1, x_2, \ldots, x_n) = \frac{\partial}{\partial x_j}x_i = \delta_{ij}$$

□

Definition 1.6.15. Let $n \in \mathbb{N}$. For any activation function Φ, we define the *post-activation function given (an activation function)* Φ as $H_\Phi : \mathbb{R}^n \longrightarrow \mathbb{R}^n$, $H_\Phi = (h_1^\Phi, h_2^\Phi, \ldots, h_n^\Phi)$, where each component $h_i^\Phi : \mathbb{R}^n \longrightarrow \mathbb{R}$ is defined as

$$h_i^\Phi(x_1, x_2, \ldots, x_n) = (\Phi \circ \pi_i)(x_1, x_2, \ldots, x_n) = \Phi(x_i)\,, \qquad i = 1, 2, \ldots, n\,.$$

Proposition 1.6.16. *Let Φ be any differentiable activation function, $n \in \mathbb{N}$ any natural number, and $H_\Phi : \mathbb{R}^n \longrightarrow \mathbb{R}^n$ a post-activation function given Φ. Then*

$$\nabla H_\Phi(x_1, x_2, \ldots, x_n) = \begin{bmatrix} \Phi'(x_1) & 0 & \ldots & 0 \\ 0 & \Phi'(x_2) & \ldots & 0 \\ \vdots & & \ddots & \vdots \\ 0 & \ldots & 0 & \Phi'(x_n) \end{bmatrix} =: \operatorname{diag}(\Phi'(X)) \in \mathbb{R}^{n\times n}\,.$$

Proof.

$$\frac{\partial h_i^\Phi}{\partial x_j} = \frac{\partial}{\partial x_j}(\Phi \circ \pi_i)(X) = \frac{\partial \Phi(\pi_i(X))}{\partial \pi_i(X)}\frac{\partial \pi_i(X)}{\partial x_j} = \delta_{ij}\,\Phi'(x_i)$$

□

The MLP can now be expressed as a function $\hat{y} : \mathbb{R}^n \longrightarrow \mathbb{R}$. For $J_0 = n$, $J_K = 1$ and matrices $W_l \in \mathbb{R}^{J_{l-1}\times J_l}$, $l = 1, 2, \ldots, K$, we'll denote the l-th pre-activation function given a matrix W_l by Z_l instead of Z_{w_l}, and similarly, the post-activation functions given activation function Φ_l will be denoted by H_l. The MLP in a state $(W_1, W_2, \ldots, W_K)$ is defined as

$$\hat{y}(X) = (H_K \circ Z_K \circ H_{K-1} \circ Z_{K-1} \circ \cdots \circ H_2 \circ Z_2 \circ H_1 \circ Z_1)(X)\,. \tag{1.12}$$

From now on, we assume all activation functions Φ_l are differentiable. We can now derive the expression for the differential of MLP $\hat{y}$ at any point $X \in \mathbb{R}^n$ using the chain rule:

$$\frac{d\hat{y}}{dX} = \frac{dH_K}{dZ_K}\frac{dZ_K}{dH_{K-1}}\frac{dH_{K-1}}{dZ_{K-1}} \cdots \frac{dH_2}{dZ_2}\frac{dZ_2}{dH_1}\frac{dH_1}{dZ_1}\frac{dZ_1}{dX}\,. \tag{1.13}$$

Deriving the backpropagation Learning Algorithm

Our differential of interest is similar to that in (1.13), however, once we reach the l-th layer and the l-th pre-activation function Z_l, we consider it as a function of a single variable. The variable in question will be the value of the weight $w_{ij}^{(l)}$ in the matrix W_l. Such a mapping will be denoted by $Z_l^{(ij)} : \mathbb{R} \longrightarrow \mathbb{R}$, and will always be associated with the $(l-1)$-th vector of post-activation values. This is reasonable, as these vectors always precede the l-th pre-activation function in the forward propagation, and we may consider them available. In short, $Z_l^{(ij)}$ is the l-th pre-activation function Z_l evaluated at the $l-1$-th post-activation vector, except that the matrix W_l is variable at index (i, j). To be precise,

$$Z_l^{(ij)}(x) = \left(\sum_{k=1}^{J_{l-1}} h_k^{(l-1)} w_{k1}^{(l)} \, , \, \ldots \, , \, \sum_{\substack{k=1 \\ k \neq i}}^{J_{l-1}} h_k^{(l-1)} w_{kj}^{(l)} + h_i^{(l-1)} x \, , \, \ldots \, , \, \sum_{k=1}^{J_{l-1}} h_k^{(l-1)} w_{kJ_l}^{(l)} \right) \in \mathbb{R}^{J_l}. \quad (1.14)$$

We can immediately see that $\nabla Z_l^{(ij)}(x) = \left[0, \ldots, 0, h_i^{(l-1)}, 0, \ldots, 0\right]^T = h_i^{(l-1)} e_j \in \mathbb{R}^{J_l \times 1}$.

As mentioned in the Note 1.6.10, the l-th vectors of pre and post-activation values are always associated with some training example $(X_p, y_p) \in \mathcal{T}$. As such, we denote them by $Z_l^{(p)}$ and $H_l^{(p)}$, where $Z_l^{(p)} = Z_l(X_p)$ and $H_l^{(p)} = H_l(X_p)$. Let $Z_{p,l}^{(i,j)} : \mathbb{R} \longrightarrow \mathbb{R}^{J_l}$ be the function defined by (1.14), only now associated with the post-activation vector $H_{l-1}^{(p)}$. The loss function at (X_p, y_p) is defined as $L_p(x) = L(y_p, x)$, and the composition of functions

$$L_p \circ H_K \circ Z_K \circ \cdots \circ Z_{l+1} \circ H_l \circ Z_{p,l}^{(i,j)}. \quad (1.15)$$

is well-defined for all $l = 1, 2, \ldots, K$, $i = 1, 2, \ldots, J_{l-1}$, $j = 1, 2, \ldots, J_l$ and $p = 1, 2, \ldots, N$. We define the following matrices for all possible indices $l, (i, j)$, and all $(X_p, y_p) \in \mathcal{T}$:

$$\left[\Delta L_p(W_l)\right]_{ij} = \nabla L_p(H_K^{(p)}) \, \nabla H_K(Z_K^{(p)}) \, \cdots \nabla Z_{l+1}(H_l^{(p)}) \, \nabla H_l(Z_l^{(p)}) \, \nabla Z_{p,l}^{(i,j)}(w_{ij}^{(l)}). \quad (1.16)$$

Now, we move on to the main idea behind the backpropagation. For fixed indices p, i, j, we once again observe that the beginning part of the Equation (1.16) is being repeated, and expands for smaller values of l. We define the parameter $\delta_l^{(p)}$, often referred to as the *local gradient* at (X_p, y_p), as the beginning part of the Equation (1.16):

$$\delta_l^{(p)} = \nabla L_p(H_K^{(p)}) \, \nabla H_K(Z_K^{(p)}) \, \nabla Z_K(H_{K-1}^{(p)}) \, \cdots \nabla Z_{l+1}(H_l^{(p)}) \, \nabla H_l(Z_l^{(p)}). \quad (1.17)$$

The local gradients allow us to simplify the matrix definition from (1.16):

$$\left[\Delta L_p(W_l)\right]_{ij} = \delta_l^{(p)} \, \nabla Z_{p,l}^{(i,j)}(w_{ij}^{(l)}). \quad (1.18)$$

The following recursion emerges for the local gradients:

$$
\begin{aligned}
\delta_K^{(p)} &= \nabla L_p(H_K^{(p)})\,\nabla H_K(Z_K^{(p)}) \\
\delta_{K-1}^{(p)} &= \nabla L_p(H_K^{(p)})\,\nabla H_K(Z_K^{(p)})\,\nabla Z_K(H_{K-1}^{(p)})\,\nabla H_{K-1}(Z_{K-1}^{(p)}) \\
&= \delta_K^{(p)}\,\nabla Z_K(H_{K-1}^{(p)})\,\nabla H_{K-1}(Z_{K-1}^{(p)}) \\
&\vdots \\
\delta_l^{(p)} &= \nabla L_p(H_K^{(p)})\,\nabla H_K(Z_K^{(p)})\,\nabla Z_K(H_{K-1}^{(p)})\cdots\nabla Z_{l+1}(H_l^{(p)})\,\nabla H_l(Z_l^{(p)}) \\
&= \delta_{l+1}^{(p)}\,\nabla Z_{l+1}(H_l^{(p)})\,\nabla H_l(Z_l^{(p)}) \\
&\vdots \\
\delta_2^{(p)} &= \nabla L_p(H_K^{(p)})\,\nabla H_K(Z_K^{(p)})\,\nabla Z_K(H_{K-1}^{(p)})\cdots\nabla Z_3(H_2^{(p)})\,\nabla H_2(Z_2^{(p)}) \\
&= \delta_3^{(p)}\,\nabla Z_3(H_2^{(p)})\,\nabla H_2(Z_2^{(p)}) \\
\delta_1^{p} &= \nabla L_p(H_K^{(p)})\,\nabla H_K(Z_K^{(p)})\,\nabla Z_K(H_{K-1}^{(p)})\cdots\nabla Z_3(H_2^{(p)})\,\nabla H_2(Z_2^{(p)})\,\nabla Z_2(H_1^{(p)})\,\nabla H_1(Z_1^{(p)}) \\
&= \delta_2^{(p)}\,\nabla Z_2(H_1^{(p)})\,\nabla H_1(Z_1^{(p)}).
\end{aligned}
$$

It allows us to efficiently calculate the matrix $\Delta L_p(W_l)$ for each training example (X_p, y_p).

Given a training set $\mathcal{T} = \{(X_1, y_1), (X_2, y_2), \ldots, (X_N, y_N)\}$ and a cost function $C_\mathcal{T}$ associated with the loss function L, we define a matrix

$$\Delta C_\mathcal{T}(W_l) = \frac{1}{N}\sum_{p=1}^{N} \Delta L_p(W_l). \tag{1.19}$$

Both matrices are purposely denoted by Δ, as they allow us to use them in the update step of a gradient descent method to update the elements of the matrix W_l all at once, and the gradient descent itself is often referred to as a *delta rule*. Given a learning rate $\eta > 0$, the weight update step in the gradient descent method can now be stated as

$$W_l^{(k+1)} = W_l^{(k)} + \eta \cdot \Delta C_\mathcal{T}(W_l^{(k)}), \quad k \geq 0\,. \tag{1.20}$$

The backpropagation works as follows: In a single epoch, the algorithm performs the forward propagation on every training example in order to calculate all vectors of pre and post-activation values. Additionally, during this forward pass, the algorithm evaluates all necessary derivatives that constitute the elements of the local gradients. Now, the backward pass starts and the local gradients are all calculated in the backward direction for each training example. Once the algorithm goes through the entire training set, the matrices from (1.18) and (1.19) are calculated, and the weights are updated via the delta rule (1.20).

The stopping rule of the backpropagation algorithm is the same as the ADALINE's, that is, the algorithm halts either after reaching the maximum number of epochs, or by an early stopping procedure.

Mini-batch Backpropagation Learning Algorithm

Lastly, the backpropagation learning algorithm is presented in the most general mini-batch gradient descent optimization setting. We will need a slightly different notation. Let

$$\mathcal{P}_{[b]}(\mathcal{T}) = \left\{\mathcal{T}^1_{(b)}, \mathcal{T}^2_{(b)}, \ldots, \mathcal{T}^M_{(b)}\right\}$$

be a partition of $\mathcal{T}$ into M b-sized mini-batches. For every $m = 1, 2, \ldots, M$ and every $p = 1, 2, \ldots, b$ we denote the p-th training exmaple in the m-th mini-batch by $(X^{(m)}_p, y^{(m)}_p)$. The associated l-th vectors of pre and post-activation values are denoted by $Z^{(m,p)}_l$ and $H^{(m,p)}_l$, and they are the results of the forward propagation on training examples $(X^{(m)}_p, y^{(m)}_p)$. The pre and post-activation functions Z_l and H_l remain unchanged as they are in no way dependent on the training set. On the other hand, $Z^{(i,j)}_{(m,p),l} : \mathbb{R} \longrightarrow \mathbb{R}^{J_l}$ is the function defined by (1.14), only now associated with the vector of post-activation values $H^{(m,p)}_{l-1}$. The loss function at $(X^{(m)}_p, y^{(m)}_p)$ is defined as

$$L_{(m,p)}(x) = L(y^{(m)}_p, x),$$

and we examine the following composition of functions:

$$L_{(m,p)} \circ H_K \circ Z_K \circ \cdots \circ Z_{l+1} \circ H_l \circ Z^{(i,j)}_{(m,p),l} \,.$$

The local gradients $\delta^{(m,p)}_l$ at $(X^{(m)}_p, y^{(m)}_p)$ are defined by

$$\delta^{(m,p)}_l = \nabla L_{(m,p)}(H^{(m,p)}_K)\, \nabla H_K(Z^{(m,p)}_K) \cdots \nabla Z_{l+1}(H^{(m,p)}_l)\, \nabla H_l(Z^{(m,p)}_l).$$

Naturally, they satisfy the same recurrence relation. Lastly, the matrices of interest are defined as

$$\left[\Delta L_{(m,p)}(W_l)\right]_{ij} = \delta^{(m,p)}_l\, \nabla Z^{(i,j)}_{(m,p),l}(w^{(l)}_{ij}), \tag{1.21}$$

and

$$\Delta C_{\mathcal{T}^{(m)}_{(b)}}(W_l) = \frac{1}{b} \sum_{p=1}^{b} \Delta L_{(m,p)}(W_l). \tag{1.22}$$

The mini-batch backpropagation learning algorithm has the following weight update rule for each mini-batch $\mathcal{T}^{(m)}_{(b)} \in \mathcal{P}_{[b]}(\mathcal{T})$, where $m = 1, 2, \ldots, M$:

$$W^{(k+1)}_l = W^{(k)}_l + \eta \cdot \Delta C_{\mathcal{T}^{(m)}_{(b)}}(W^{(k)}_l), \quad k \geq 0\,. \tag{1.23}$$

The weight matrices are updated M times in a given epoch, once for each mini-batch. The algorithm is essentially the same as the (batch) backpropagation described earlier, and it's presented in Algorithm 3.

Algorithm 3: The Mini-batch Backpropagation Learning Algorithm

Input: Training set $\mathcal{T} = \{(X_1, y_1), (X_2, y_2), \ldots, (X_N, y_N)\}$, Test set $\mathcal{T}_{test}$, Batch size b, Learning rate η, Maximum number of epochs t_{max}, Relative error lower bound r^*, Early stopping parameter s

Initialize: $k = t^* = t = j = 0$, $\omega_0 = RAND$, $\varepsilon_0 = C_{\mathcal{T}_{test}}(\omega_0)$, $r_0 = r^* + 1$, $M = N/b$

Algorithm:

while $t < t_{max}$ *or* $j < s$ **do**

  $\mathcal{P}_{[b]}(\mathcal{T}) \leftarrow \{\mathcal{T}^1_{(b)}, \mathcal{T}^2_{(b)}, \ldots, \mathcal{T}^M_{(b)}\}$; /* Randomly Partition $\mathcal{T}$ */

  for $m = 1, 2, \ldots, M$ **do** /* For each mini-batch */

    for $p = 1, 2, \ldots, b$ **do** /* For each training example */

      for $l = 1, 2, \ldots, K$ **do** /* Forward pass */

        $Z_l^{(m,p)} = W_l^T H_{l-1}^{(m,p)}$; /* $H_0^{(m,p)} = X_p^{(m)}$ */

        $H_l^{(m,p)} = \Phi_l(Z_l^{(m,p)})$;

        $\nabla H_l(Z_l^{(m,p)}) = \mathrm{diag}(\Phi'(Z_l^{(m,p)}))$;

      end

      $\delta_K^{(m,p)} = \nabla L_{(m,p)}(H_K^{(m,p)}) \nabla H_K(Z_K^{(m,p)})$;

      $\left[\Delta L_{(m,p)}(W_K)\right]_{ij} = \delta_K^{(m,p)} \nabla Z^{(i,j)}_{(m,p),K}(w_{ij}^{(K)})$; /* $\nabla L_{(m,p)}(W_K)$ */

      for $l = K - 1, \ldots, 1$ **do** /* Backward pass */

        $\delta_l^{(m,p)} = \delta_{l+1}^{(m,p)} \nabla Z_{l+1}(H_l^{(m,p)}) \nabla H_l(Z_l^{(m,p)})$;

        $\left[\Delta L_{(m,p)}(W_l)\right]_{ij} = \delta_l^{(m,p)} \nabla Z^{(i,j)}_{(m,p),l}(w_{ij}^{(l)})$; /* $\nabla L_{(m,p)}(W_l)$ */

      end

    end

    for $l = 1, 2, \ldots, K$ **do** /* Update the network state ω */

      $\Delta C_{\mathcal{T}^m_{(b)}}(W_l) = \frac{1}{b} \sum_{p=1}^{b} \Delta L_{(m,p)}(W_l)$;

      $W_l^{(k+1)} = W_l^{(k)} + \eta \cdot \Delta C_{\mathcal{T}^m_{(b)}}(W_l^{(k)})$;

    end

    $k \leftarrow k + 1$;

  end

  $t \leftarrow t + 1$; $\varepsilon_t = C_{\mathcal{T}_{test}}(\omega_k)$; $d_t = \varepsilon_{t^*} - \varepsilon_t$; $r_t = |d_t|/\varepsilon_t$;

  if $d_t \le 0$ *or* $r_t < r^*$ **then**

    $j \leftarrow j + 1$;

    continue ; /* Continue on to the next epoch */

  else

    $t^* \leftarrow t$; /* Memorize the new best epoch */

    $\omega_* \leftarrow \omega_k$;

    $j \leftarrow 0$; /* Reset the early stopping parameter */

  end

end

Output: ω_*

Universal Approximation Theorem

In this section we explore the approximation capabilities of the MLP class of functions. MLP is often referred to as a *universal function approximator*, and can be used for classification of linearly inseparable patterns and function approximation in regression problems. By approximating a function defining some decision boundary, we can convert the classification problem into a regression problem. This leads us to cases of approximating a classification function directly or approximating the decision boundary, and from it defining a classifier.

Sets C^n and M^n are defined as follows:

$$C^n = \{f : \mathbb{R}^n \longrightarrow \mathbb{R} \mid \text{ f is continuous}\}, \; M^n = \{f : \mathbb{R}^n \longrightarrow \mathbb{R} \mid \text{ f is Borel measurable}\}.$$

Consider the MLP Definition 1.6.5. For any $n \in \mathbb{N}$, let $\mathcal{A}_n(\Psi)$ denote a set of all MLPs with one hidden layer of arbitrary width ($k = 1$, $J_1 \in \mathbb{N}$), defined on the input space $\mathcal{X} = \mathbb{R}^n$, and set of network states $\Omega = \mathbb{R}^{n\times J_1} \times \mathbb{R}^{J_1\times 1}$, with $\Phi_1 = \Psi$ being any sigmoidal activation function and Φ_2 an identity function. Such MLPs are of the form $\hat{y}(W_1, W_2, X) = \Phi_2(W_2^T\Phi_1(W_1^T X)) = W_2^T\Psi(W_1^T X)$, and $\mathcal{A}_n(\Psi)$ can be written as:

$$\mathcal{A}_n(\Psi) = \left\{X \mapsto W_2^T\Psi(W_1^T X) : X \in \mathbb{R}^n,\; J_1 \in \mathbb{N},\; W_1 \in \mathbb{R}^{n\times J_1},\; W_2 \in \mathbb{R}^{J_1}\right\}.$$

The final scalar product can be more conveniently written in a summation form, resulting in a more interpretable definition of set $\mathcal{A}_n(\Psi)$:

$$\mathcal{A}_n(\Psi) = \left\{X \mapsto \sum_{i=1}^{N} \alpha_i \Psi(W^T X) : X, W \in \mathbb{R}^n,\; N \in \mathbb{N},\; \alpha_i \in \mathbb{R}\; \forall i = 1, 2, \ldots, N\right\}.$$

$\mathcal{A}_n(\Psi)$ contains all finite linear combinations of a fixed sigmoidal function Ψ composed with any linear function of X. Such functions are measurable, meaning that $\mathcal{A}_n(\Psi) \subseteq M^n$, and for a continuous sigmoidal function Ψ we have $\mathcal{A}_n(\Psi) \subseteq C^n$.

Definition 1.6.17. For any $n \in \mathbb{N}$, $K \subseteq \mathbb{R}^n$, and $f, g : \mathbb{R} \longrightarrow \mathbb{R}$ we define $\rho_K(f, g)$ as

$$\rho_K(f, g) = \sup_{x\in K} |f(x) - g(x)|.$$

Definition 1.6.18. Let $n \in \mathbb{N}$. Given a probability measure μ on Borel measure space $(\mathbb{R}^n, B^n)$ and measurable functions $f, g : \mathbb{R} \longrightarrow \mathbb{R}$ we define $\rho_\mu(f, g)$ as

$$\rho_\mu(f, g) = \inf_{\varepsilon>0} \mu(\{x \in \mathbb{R}^n : |f(x) - g(x)| > \varepsilon\}) < \varepsilon\,.$$

Note 1.6.19. Both ρ_K and ρ_μ are *metrics* in *metric spaces* of functions (C^n, ρ_K) for compact sets K and (M^n, ρ_μ), and the following theorem is generally stated in terms of *density* of $\mathcal{A}_n(\Psi)$ in those metric spaces. For simplicity, we deter from the use of such terminology.

Theorem 1.6.20 (Universal Approximation Theorem)**.** *Let Ψ be any sigmoidal function and $n \in \mathbb{N}$. Then the following is true:*

1. $\forall \varepsilon > 0 \, \forall f \in C^n \, \exists g \in \mathcal{A}_n(\Psi)$ *such that*

$$\rho_K(f, g) < \varepsilon$$

for any compact set $K \subseteq \mathbb{R}^n$.

2. $\forall \varepsilon > 0 \, \forall f \in M^n \, \exists g \in \mathcal{A}_n(\Psi)$ *such that*

$$\rho_\mu(f, g) < \varepsilon$$

for any probability measure μ on $(\mathbb{R}^n, B^n)$.

□

Note 1.6.21. Any finite measure instead of a probability measure would have sufficed, however, we are exploring neural networks in a probabilistic context.

Universal Approximation Theorem states that MLP with only a single hidden layer can approximate any continuous function uniformly on any compact set and any measurable function arbitrarily well with respect to ρ_μ, regardless of the sigmoidal activation function (continuous or not), regardless of the dimension of the input space n, and regardless of the probability measure μ.

It's worth noting that the Universal Approximation Theorem is purely an existence theorem, providing no instructions on how to construct such a network or the necessary boundaries on the number of hidden units. It's referred to as an *arbitary width case* in a class of universal approximation theorems concerning neural networks.

Ideally, we would have the same uniform approximation result for measurable functions as well, as it is a stronger result. This proves to be too demanding, but the following theorem shows we can come arbitrarily close to the desired result.

Theorem 1.6.22. *Let Ψ be any sigmoidal function, $n \in \mathbb{N}$, and μ any probability measure on $(\mathbb{R}^n, B^n)$. For every $\varepsilon > 0$ and every measurable function $f \in M^n$ there exists a compact set $K \subseteq \mathbb{R}^n$ and $g \in \mathcal{A}_n(\Psi)$ such that $\mu(K) \geq 1 - \varepsilon$, and for every $x \in K$ we have $|f(x) - g(x)| < \varepsilon$.* □

Theorem 1.6.22 shows that for any measurable function, there is a single hidden layer MLP that approximates it to any desired degree of accuracy on some compact set K spanning any desirable percentage of the input space. In the case of a classifier, this result states that the total measure of the incorrectly classified points can be made arbitrarily small.

Let's assume a binary classification function f, defined on some compact set $D \subseteq \mathbb{R}^n$, is of the form χ_K for some closed set $K \subseteq D$, where χ_K is the *characteristic function* of the set K. Then the function Δ defined as

$$\Delta(x, K) = \inf_{y \in K} |x - y|$$

is a continuous function of x, and measures the distance of any point $x \in \mathbb{R}^n$ to set K. Setting

$$f_\varepsilon(x) = \max\left\{0, \frac{\varepsilon - \Delta(x, K)}{\varepsilon}\right\}$$

we get $f_\varepsilon(x) = 1$ for $x \in K$ and $f_\varepsilon(x) = 0$ for points x farther than ε away from K. Moreover, $f_\varepsilon(x)$ continuous for all $x \in D$.

By universal approximation theorem, there exists a function $g \in \mathcal{A}_n(\Psi)$ such that $|g(x) - f_\varepsilon(x)| < 0.5$ on the entire domain D. We can use this g as an approximate decision function: $g(x) \geq 0.5$ guesses that $x \in K$, while $g(x) < 0.5$ guesses that $x \in K^{\mathsf{c}}$. This decision procedure is correct for all $x \in K$ and for all x at a distance at least ε away from K, since f_ϵ is exactly χ_K for such points, and g is less than 0.5 distance from f_ε. Meaning, that $g(x) > 0.5$ on K, and $g(x) < 0.5$ for all points x at least ε away from K. If x is within ε distance of K, its classification depends on the particular choice of g. These observations say that points sufficiently far away from, and points inside the closed decision region can be classified correctly. In contrast, Theorem 1.6.22 says that there is an MLP that makes the measure of incorrectly classified points as small as desired but does not guarantee their location.

The Universal Approximation Theorem 1.6.20 presented in this section is an adaptation of Theorem 2.4 from Hornik *et al.* (1989) [8], where its proof can be found, along with the proof of Theorem 1.6.22. Very similar universal approximation results were independently proved the same year by both Cybenko [3] and Funahashi [5]. One might ask how important is the use of sigmoidal activation functions for universal approximation, and whether an arbitrary activation function would have sufficed. Interestingly, Leshno *et al.* in 1993 [9] and later Pinkus in 1999 [14] showed that the universal approximation property is equivalent to having a nonpolynomial activation function.

Why are deep neural networks so widely used if a single hidden layer is enough for universal approximation? The failures in practical applications of the Universal Approximation Theorem can be attributed to inadequate learning, an insufficient number of hidden units, insufficient training data or the presence of a stochastic rather than a deterministic relationship between input and output. Deep neural networks provide a more efficient way of encompassing high non-linearity with the use of additional hidden layers, while a single hidden layer network essentially approximates a function with *step* (or *Heaviside*) functions – which is theoretically achievable, but not very practical.

Chapter 2

Statistical Learning theory

Chapter 2 is still within the restricted framework mentioned in the Introduction , that is, the use of artificial neural networks for supervised learning problems in a binary classification setting, with the output label being either 0 or 1. Results presented in this chapter are entirely adapted from Anthony and Bartlett's *Neural Network Learning: Theoretical Fundations* [2] (Part one: Pattern Classification with Binary-Output Neural Networks), where the proof of each statement can be found.

Understanding the behaviour of machine learning models and algorithms used for solving information processing problems (like pattern recognition and prediction) is crucial for the design of effective *learning systems*. Three main questions naturally arise.

The first one addresses the *approximation properties* of a learning system: we associate a *class of mappings* between the input patterns and the output labels with every learning system, and the question is whether this class is sufficiently powerful to accurately approximate the ”true relationship” between the input patterns and their labels. This ”true relationship” isn't necessarily a mapping from an input space to the output space. The second key issue is a statistical one, concerning *estimation*: since the underlying relationship between the input patterns and their labels is unknown, and we receive only a finite amount of data (regarding said relationship) through the training set, how much data suffices to model the relationship with the desired accuracy? The third key question concerns the *computational efficiency* of learning algorithms: how efficiently can we make use of the training data to choose an accurate model of the underlying relationship?

This chapter focuses on the estimation question. The question of approximation was discussed in Chapter 1, with the perceptron's approximation capabilities being rather limited, as shown by the XOR problem in Section 1.3. On the other hand, we've shown how the MLP possesses the universal approximation property. Additionally, we've touched upon the computational efficiency of the perceptron learning algorithm by establishing an upper bound on the number of required update steps before finding the optimal weights.

2.1 The Learning Problem

The notion of learning is formally defined in the language of probability theory, but first, we motivate the definition with a less technical discussion. The notation throughout this chapter is the same as in Chapter 1, with the input space being denoted by $\mathcal{X}$, the set of network states by Ω, and the output space by $\mathcal{Y}$.

In a supervised learning environment, a learning algorithm describes how to change the state of the network in response to training data. We'll assume that the training set is a *finite sample* drawn randomly from a fixed probability distribution $\mathbb{P}$ on $\mathcal{X}\times\mathcal{Y}$, where $\mathcal{Y} = \{0, 1\}$. As such, for a given $X \in \mathcal{X}$, both $(X, 0)$ and $(X, 1)$ may have positive probabilities, meaning that neither 0 or 1 is the "correct" label for that X. Therefore, there isn't necessarily an underlying classification function $f : \mathcal{X} \longrightarrow \mathcal{Y}$ such that $\mathbb{P}(\{(X, f(X)) : X \in \mathcal{X}\}) = 1$. Even if such a function representing the "true relationship" did exist, we do not assume that the neural network is capable of computing it. This is a more general approach, allowing a classification problem in which some inputs are ambiguous, or in which there is some noise corrupting the input patterns or the output labels. The aim of successful learning is that, after training on a large enough sequence of labelled examples, the neural network computes a function that matches, almost as closely as it can, the process generating the data; that is, we hope that the classification of subsequent unseen examples is close to the best performance that the network can possibly manage.

As mentioned, in statistical learning theory, the training set is considered to be a realisation of a *random sample* generated by an unknown distribution $\mathbb{P}$ on $\mathcal{X} \times \mathcal{Y}$. More precisely, let $\mathcal{I}$ be a σ-algebra on the input space $\mathcal{X}$, and $\mathcal{O}$ a σ-algebra on the output space $\mathcal{Y}$. For $\mathcal{Z} = \mathcal{X} \times \mathcal{Y}$ and a product σ-algebra $\mathcal{V} = \mathcal{I} \otimes \mathcal{O}$ on $\mathcal{Z}$, let $\mathbb{P}$ be a probability on a measure space $(\mathcal{Z}, \mathcal{V})$. Functions $X : \mathcal{Z} \longrightarrow \mathcal{X}$ and $y : \mathcal{Z} \longrightarrow \mathcal{Y}$ defined by $X(i, o) = i$ and $y(i, o) = o$ are random variables, and (X, y) is a random vector with distribution $\mathbb{P}$.

Similarly, given a probability space $(\mathcal{Z}^m, \mathcal{V}^m, \mathbb{P}^m)$, we define the m-sized random sample $Z_1 = (X_1, y_1),\ Z_2 = (X_2, y_2), \ldots, Z_N = (X_m, y_m)$ by $X_k((i_1, o_1), (i_2, o_2), \ldots, (i_m, o_m)) = i_k$, and $y_k((i_1, o_1), (i_2, o_2), \ldots, (i_m, o_m)) = o_k$. Now, $Z_1, Z_2, \ldots, Z_m$ are independent identically distributed random variables with distribution $\mathbb{P}$, and we assume they generate our training set $\mathcal{T}$, now represented by $z \in \mathcal{Z}^m$ – a realisation of a random vector $Z = (Z_1, Z_2, \ldots, Z_m)$.

Note 2.1.1. In Chapter 2, random vectors (X, y) and (X_k, y_k) have the same notation as elements of a training set $\mathcal{T}$ in Chapter 1. This shouldn't cause any confusion since the random sample vector $((X_1, y_1), (X_2, y_2), \ldots, (X_m, y_m))$ is denoted by $Z = (Z_1, Z_2, \ldots, Z_m)$, and its realisation representing (an instance of) a training set $\mathcal{T}$ by $z = (z_1, z_2, \ldots, z_m)$. Therefore, vectors (X, y) and (X_k, y_k) aren't needed to represent the training data points, and are here strictly used as random vectors.

The notion of a loss function is slightly modified to fit the statistical learning framework, in that, we restrict its domain to $\mathcal{Y} \times \mathcal{Y}$. Now, a loss function is any non-negative mapping $L : \mathcal{Y} \times \mathcal{Y} \longrightarrow \mathbb{R}$ that is $(\mathcal{O} \otimes \mathcal{O}, B(\mathbb{R}))$-measurable.

For any loss function L and probability distribution $\mathbb{P}$ on $(\mathcal{Z}, \mathcal{V})$, we define a class of functions $\mathcal{H}_{\mathbb{P},L}$ as

$$\mathcal{H}_{\mathbb{P},L} = \{h : \mathcal{X} \longrightarrow \mathcal{Y} \mid \mathbb{E}(L(y, h(X)) < +\infty\}, \tag{2.1}$$

where $\mathbb{E}$ is mathematical expectation with respect to $\mathbb{P}$. Elements of $\mathcal{H}_{\mathbb{P},L}$ are often called *hypotheses*, and for any hypothesis $h \in \mathcal{H}_{\mathbb{P},L}$, we define the *error of h with respect to* $\mathbb{P}$ *and* L (simply called *the error of h*) as

$$\mathrm{er}_{\mathbb{P},\mathrm{L}}(h) = \mathbb{E}(L(y, h(X)).$$

The error $\mathrm{er}_{\mathbb{P},\mathrm{L}}(h)$ measures how accurately h approximates the relationship between the input patterns and their labels generated by $\mathbb{P}$, with respect to some loss function L. Since the distribution $\mathbb{P}$ is unknown, we will approximate the error of h with respect to $\mathbb{P}$ and L on an m-sized sample $z = (z_1, z_2, \ldots, z_m) \in \mathcal{Z}^m$ drawn from $(\mathcal{Z}^m, \mathcal{V}^m, \mathbb{P}^m)$, where $z_k = (i_k, o_k) \in \mathcal{Z}$, by defining the *sample error function* $\hat{\mathrm{er}}_{\mathrm{m,L}} : \mathcal{Z}^m \times \mathcal{H}_L \longrightarrow \mathbb{R}$ as

$$\hat{\mathrm{er}}_{\mathrm{m,L}}(z, h) = \hat{\mathrm{er}}_{\mathrm{m,L}}(z_1, z_2, \ldots, z_m, h) = \frac{1}{m} \sum_{i=k}^{m} L(o_k, h(i_k)).$$

The sample error $\hat{\mathrm{er}}_{\mathrm{m,L}}$ is an unbiased and strongly consistent estimator of the true error $\mathrm{er}_{\mathbb{P},\mathrm{L}}$, once evaluated on random sample vector $Z = (Z_1, Z_2, \ldots, Z_m)$.

For any $\mathcal{H} \subseteq \mathcal{H}_{\mathbb{P},L}$ the *approximation error of class* $\mathcal{H}$ is defined as

$$\mathrm{opt}_{\mathbb{P},\mathrm{L}}(\mathcal{H}) = \inf_{h \in \mathcal{H}} \mathrm{er}_{\mathbb{P},\mathrm{L}}(h).$$

The approximation error describes how accurately the best function in $\mathcal{H}$ approximates the relationship between X and y determined by the probability distribution $\mathbb{P}$. For any predetermined $\varepsilon \in (0, 1)$ called the *accuracy parameter*, the learning algorithm aims to produce near-optimal $h^* \in \mathcal{H}$, such that

$$\mathrm{er}_{\mathbb{P},\mathrm{L}}(h^*) < \mathrm{opt}_{\mathbb{P},\mathrm{L}}(\mathcal{H}) + \varepsilon.$$

We say that such h^* is *ε-good for* $\mathbb{P}$.

It is possible that an unrepresentative training sample will be drawn and mislead a learning algorithm. Therefore, no learning algorithm can guarantee that a computed hypothesis will always be ε-good. Instead, we wish to find a sufficiently large training sample to ensure that the computed hypothesis will be ε-good with a probability of at least $1 - \delta$, for some predetermined $\delta \in (0, 1)$ called the *confidence parameter*. This is formalised into a notion called the *Probably Approximately Correct learning*.

Formal Definition of Learning

For any loss function L, let $\mathcal{H}_L$ denote the following class of functions:

$$\mathcal{H}_L = \bigcap_{\mathbb{P}} \mathcal{H}_{\mathbb{P},L},$$

where $\mathbb{P}$ is a probability distribution on $(\mathcal{Z}, \mathcal{V})$. This function class is used to define a learning algorithm as a mapping invariant of the (unknown) distribution $\mathbb{P}$.

Definition 2.1.2 (PAC learning)**.** Let $\mathcal{H} \subseteq \mathcal{H}_L$ ba any class of functions. A mapping

$$A : \bigcup_{m=1}^{\infty} \mathcal{Z}^m \longrightarrow \mathcal{H}$$

is a *learning algorithm for* $\mathcal{H}$ if for every $\varepsilon \in (0,1)$ and for every $\delta \in (0,1)$ there exists $m_0 \in \mathbb{N}$ such that for every $m \geq m_0$ and every probability distribution $\mathbb{P}$ on $(\mathcal{Z}, \mathcal{V})$, function $\mathrm{er}_{\mathbb{P},\mathrm{L}}(A(\cdot)) : \mathcal{Z}^m \longrightarrow [0, +\infty\rangle$ is measurable and

$$\mathbb{P}^m \left(\mathrm{er}_{\mathbb{P},\mathrm{L}}(A(Z)) < \mathrm{opt}_{\mathbb{P},\mathrm{L}}(H) + \varepsilon \right) \geq 1 - \delta.$$

We say that $\mathcal{H}$ is (PAC) *learnable* if there exists a learning algorithm for $\mathcal{H}$, and the quantity $m_0(\varepsilon, \delta)$ is called the *sufficient sample size* for (ε, δ)-learning of $\mathcal{H}$ by A. A measure of the efficiency of a learning algorithm is the smallest sufficient sample size for (ε, δ)-learning of $\mathcal{H}$ by A, called a *sample complexity of* A and defined as

$$m_{A,\mathcal{H}}(\varepsilon, \delta) = \min \{ m \in \mathbb{N} : m \text{ is sufficient sample size for } (\varepsilon, \delta)\text{-learning of } \mathcal{H} \text{ by } A \}.$$

The *inherent sample complexity of* $\mathcal{H}$ is defined as $m_{\mathcal{H}}(\varepsilon, \delta) = \min_A m_{A,\mathcal{H}}(\varepsilon, \delta)$, where the minimum is taken over all learning algorithms for $\mathcal{H}$. The inherent sample complexity $m_{\mathcal{H}}(\varepsilon, \delta)$ provides an absolute lower bound on the sample size needed for (ε, δ)-learning of $\mathcal{H}$, regardless of the learning algorithm.

An equivalent definition of PAC learning states that a function A is a learning algorithm for $\mathcal{H}$ if for every $m \in \mathbb{N}$ and every $\delta \in (0,1)$ there exists $\varepsilon_0 \in (0,1)$ such that for every probability distribution $\mathbb{P}$ on $(\mathcal{Z}, \mathcal{V})$ the following holds:

$$\mathbb{P}^m \left(\mathrm{er}_{\mathbb{P},\mathrm{L}}(A(Z)) < \mathrm{opt}_{\mathbb{P},\mathrm{L}}(H) + \varepsilon_0(m, \delta) \right) \geq 1 - \delta,$$

and $\lim_m \varepsilon_0(m, \delta) = 0$ for every $\delta \in (0,1)$. We refer to such $\varepsilon_0(m, \delta)$ as an *estimation error bound for algorithm* A, and define the *estimation error* $\varepsilon_{A,\mathcal{H}}(m, \delta)$ *of* A to be the smallest possible estimation error bound allowing the (m, δ)-learning of $\mathcal{H}$ by an algorithm A. Also, the *inherent estimation error of* $\mathcal{H}$ is analogously defined as $\varepsilon_{\mathcal{H}}(m, \delta) = \min_A \varepsilon_{A,\mathcal{H}}(m, \delta)$.

It's worth noting that the function class $\mathcal{H}_L$ is not PAC learnable. The central question of this chapter is to determine under which conditions is a given class of functions $\mathcal{H} \subseteq \mathcal{H}_L$ learnable and, if so, how to design a learning algorithm for $\mathcal{H}$. In analysing learning algorithms, the results are often presented either in the form of sample complexity bounds or estimation error bounds. It is usually straightforward to transform between the two.

2.2 Learning Finite Function Classes

The aim of learning is to produce a function $h^* \in \mathcal{H}$ having a near-minimum error $\mathrm{er}_{\mathbb{P},\mathrm{L}}(h^*)$, as close as possible to the optimum class approximation error $\mathrm{opt}_{\mathbb{P},\mathrm{L}}(\mathcal{H})$. Given that the true errors of the functions in $\mathcal{H}$ are unknown, it seems natural to use the sample error as its estimate, hoping that if a function h has a small sample error, then it has a small true error as well. In this section, we show that this is true for finite function classes.

For any loss function L and any (possibly infinite) function class $\mathcal{H} \subseteq \mathcal{H}_L$ we define a *sample error minimization (SEM) algorithm for a function class $\mathcal{H}$ and a loss function L* to be any function $A : \bigcup_{m=1}^{\infty} \mathcal{Z}^m \longrightarrow \mathcal{H}$, such that for every $m \in \mathbb{N}$ and every $z \in \mathcal{Z}^m$

$$A(z) \in \arg\min_{h \in \mathcal{H}} \hat{\mathrm{er}}_{\mathrm{m,L}}(z, h).$$

So far, every definition allowed for a general output space $\mathcal{Y}$ and a general loss function $L : \mathcal{Y} \times \mathcal{Y} \longrightarrow \mathbb{R}$. This is because statistical learning theory is similarly developed in different learning settings. For the rest of this chapter, however, we concentrate on developing a learning theory strictly for the binary classification setting where $\mathcal{Y} = \{0, 1\}$, and for a specific loss function $L : \{0, 1\}^2 \longrightarrow \mathbb{R}$ called the *0–1 loss function* and defined by

$$L(i, j) = 1 - \delta_{ij} = \chi_{i \neq j},$$

where δ_{ij} is the Kronceker's delta function. From now on, the output space is fixed to $\{0, 1\}$.

For any $(\mathcal{I}, \mathcal{O})$-measurable function $h : \mathcal{X} \longrightarrow \mathcal{Y}$ and the $0-1$ loss function L, mathematical expectation $\mathbb{E}(L(y, h(X)))$ is always finite. Since $\mathbb{E}(\chi_A) = \mathbb{P}(A) \leq 1$ for every event $A \in \mathcal{V}$, we have

$$\mathbb{E}(L(y, h(X))) = \mathbb{E}(\chi_{y \neq h(X)}) = \mathbb{P}(y \neq h(X)) \leq 1. \tag{2.2}$$

In such case, the function class $\mathcal{H}_{\mathbb{P},L}$ defined in (2.1), and subsequently the function class $\mathcal{H}_L$, is simply the set of all $(\mathcal{I}, \mathcal{O})$-measurable functions $h : \mathcal{X} \longrightarrow \mathcal{Y}$.

Theorem 2.2.1. *Let L be the $0-1$ loss function, $\mathcal{H} \subseteq \mathcal{H}_L$ any finite class of functions, and A any SEM algorithm for $\mathcal{H}$ and the $0-1$ loss function L. Then A is a learning algorithm for $\mathcal{H}$ with sample complexity*

$$m_{A,L}(\varepsilon, \delta) \leq \frac{2}{\varepsilon^2} \ln\left(\frac{2\,|\mathcal{H}|}{\delta}\right).$$

□

Applications to Perceptrons

Let $n \in \mathbb{N}$ be any natural number. The M-P neuron from Definition 1.3.4 is a function $F : \Omega \times \mathcal{X} \longrightarrow \mathcal{Y}$ defined by $F(W, \theta, X) = \phi(W^T X - \theta)$, where ϕ is the threshold activation function, $\mathcal{X} = \{0, 1\}^n$ the input space, $\Omega \subseteq \{-1, 1\}^n \times \mathbb{R}$ the set of network states, and $\mathcal{Y}$ the output space. We'll consider these parameters fixed throughout this subsection.

The threshold parameter θ can generally be an arbitrary real number. However, this assumption is too relaxed for implementing boolean functions. We can significantly reduce the number of possible threshold parameters without the M-P neuron losing any of its representation capabilities. Since the M-P neuron essentially adds and subtracts at most n zeros and ones, any whole integer in $[-n - 1, n + 1]$ will surely suffice as the threshold parameter. We'll denote said integers by $\mathbb{Z}_n = [-n - 1, n + 1] \cap \mathbb{Z}$, and the set of network states for the M-P neuron is now defined by $\Omega_n = \{-1, 1\}^n \times \mathbb{Z}_n$. Such M-P neurons form the following function class:

$$\mathcal{H}_{\text{M-P}} = \{h_\omega : \mathcal{X} \longrightarrow \mathcal{Y} \mid \omega \in \Omega_n \text{ and } h_\omega(X) = F(\omega, X)\}.$$

The cardinality of $\mathcal{H}_{M\text{-}P}$ can be calculated as $|\mathcal{H}_{M\text{-}P}| = |\Omega| = |\{0, 1\}^n| \cdot |\mathbb{Z}_n| = 2^n(2n + 3)$.

A generalised version of the described M-P neuron is the *k-bit perceptron*, where the weights and threshold are expressible in binary as arrays of k zeros and ones. Let $\mathcal{H}_{k\text{-bit}}$ denote the set of all k-bit perceptrons of n inputs. Since there are 2^k possibilities for each weight and threshold, we have $|\mathcal{H}_{k\text{-bit}}| = 2^{k(n+1)}$.

By using the relation (2.2) and the fact that all functions in both $\mathcal{H}_{\text{M-P}}$ and $\mathcal{H}_{k\text{-bit}}$ are measurable, Theorem 2.2.1 guarantees that these function classes are learnable by any SEM algorithm given the $0-1$ loss function.

Corollary 2.2.1.1. *Let L be the $0-1$ loss function, A any SEM algorithm for the function classes $\mathcal{H}_{M\text{-}P}$ and L, and A' any SEM algorithm for the function classes $\mathcal{H}_{k\text{-}bit}$ and L. Then, A and A' are learning algorithms for $\mathcal{H}_{M\text{-}P}$ and $\mathcal{H}_{k\text{-}bit}$ respectively. Additionally, the following bound on the sample complexity holds for $\mathcal{H}_{M\text{-}P}$*

$$m_{A,\mathcal{H}_{M\text{-}P}}(\varepsilon, \delta) \leq \frac{2}{\varepsilon^2}\left(n \ln(2) + \ln(2n + 3) + \ln\left(\frac{2}{\delta}\right)\right),$$

and similarly, for the k-bit perceptron function class $\mathcal{H}_{k\text{-}bit}$

$$m_{A',\mathcal{H}_{k\text{-}bit}} \leq \frac{2}{\varepsilon^2}\left(k(n + 1) \ln(2) + \ln\left(\frac{2}{\delta}\right)\right).$$

□

2.3 Vapnik-Chervonenkis Dimension

The previous section showed how finite classes of functions are learnable. However, many interesting function classes are not finite. For example, the number of functions computed by the perceptron with real-valued weights is infinite. Many other neural networks, such as MLPs, can also be represented as a parameterized function class with an infinite parameter set. We'll see that learning is possible provided the function class is not too complex, and the measure of complexity we examine is the *Vapnik-Chervonenkis dimension*.

For any finite $S \subseteq \mathcal{X}$ we define the restriction of $\mathcal{H} = \{h \mid h : \mathcal{X} \longrightarrow \mathcal{Y}\}$ to the set S by $\mathcal{H}|_S = \{h|_S : h \in \mathcal{H}\}$. If $\mathcal{H}|_S$ is the set of all possible functions from S to $\{0, 1\}$, then clearly, $\mathcal{H}$ is as powerful as it can be in classifying the points in S. We can view the cardinality of $\mathcal{H}|_S$ (and in particular how it compares with $2^{|S|}$) as a measure of the classification complexity of $\mathcal{H}$ with respect to the set S. Since we are in the binary classification setting, it holds that $|H|_S| \leq 2^{|S|}$ with possible strict inequality.

Definition 2.3.1 (VC-dimension)**.** Let $\mathcal{H} = \{h \mid h : \mathcal{X} \longrightarrow \mathcal{Y}\}$ be an arbitrary class of functions. We define the *Vapnik-Chervonenkis dimension* (or *VC-dimension*) of $\mathcal{H}$ as

$$\text{VCdim}(\mathcal{H}) = \sup_{\substack{S \subseteq \mathcal{X} \\ |S| = p}} \{p \in \mathbb{N} : |\mathcal{H}|_S| = 2^p\}.$$

For any finite-dimensional vector space of functions $V = \{f \mid f : \mathcal{X} \longrightarrow \mathbb{R}\}$ and any function $g : \mathcal{X} \longrightarrow \mathbb{R}$ we define the function class $\mathcal{H}_g(V) = \{\phi(f + g) : f \in V\}$, where ϕ is the threshold activation function. The following theorem shows the connection between the VC-dimension of function class $\mathcal{H}_g(V)$ and the dimension of the associated vector space V.

Theorem 2.3.2. *Let $V = \{f \mid f : \mathcal{X} \longrightarrow \mathbb{R}\}$ be a finite-dimensional vector space and $g : \mathcal{X} \longrightarrow \mathbb{R}$ any function. Then* $\text{VCdim}(\mathcal{H}_g(V)) = \dim(V)$.

□

We can apply Theorem 2.3.2 to the (real-weight) perceptron from Definition 1.4.4. For the threshold activation function ϕ, we denote the class of functions calculated by the perceptron of n inputs by

$$\mathcal{P}_n = \{h_w : \mathcal{X} \longrightarrow \mathcal{Y} \mid h_w(X) = \phi(W^T X),\ W \in \mathbb{R}^n\}.$$

A perceptron is clearly a thresholded dot product of vectors $X, W \in \mathbb{R}^n$. If we consider the dual vector space of $\mathbb{R}^n$, by Riesz representation theorem we know that each $f \in (\mathbb{R}^n)^*$ can be represented as a unique dot product such that $f(X) = f_w(X) = W^T X$. Now, since

$$\mathcal{P}_n = \{\phi(f) : f \in (\mathbb{R}^n)^*\} = \mathcal{H}_0((\mathbb{R}^n)^*),$$

we have

$$\text{VCdim}(\mathcal{P}_n) = \text{VCdim}(\mathcal{H}_0((\mathbb{R}^n)^*)) = \dim((\mathbb{R}^n)^*) = n.$$

Sample Error Minimization

For an m-sized random sample vector $Z = (Z_1, Z_2, \ldots, Z_m)$ from $(\mathcal{Z}^m, \mathcal{V}^m, \mathbb{P}^m)$, the sample error $\hat{\text{er}}_{\text{m,L}}(Z, h)$ is an unbiased and strongly consistent estimator of the true error $\text{er}_{\mathbb{P},\text{L}}(h)$. This leads to the following result showing *uniform convergence* property of sample error on any function class with a finite VC-dimnsion.

Theorem 2.3.3 (Uniform Convergence)**.** *Let L be a $0-1$ loss function and $\mathcal{H} \subseteq \mathcal{H}_L$ any function class with a finite VC-dimension d. Then, for every $\varepsilon \in (0,1)$ and every $m \in \mathbb{N}$ such that $m \geq d$, it holds that*

$$\mathbb{P}^m\left(\sup_{h\in\mathcal{H}}\left|\text{er}_{\mathbb{P},\text{L}}(h) - \hat{\text{er}}_{\text{m,L}}(Z,h)\right| > \varepsilon\right) < \left(m^d + 1\right)\exp\left(-\frac{\varepsilon^2 m}{8}\right)$$

for every probability distribution $\mathbb{P}$ on $(\mathcal{Z}, \mathcal{V})$.

□

The upper bound in Theorem 2.3.3 tends to zero as m tends to infinity, which essentially leads any function class with a finite VC-dimension being PAC learnable by any SEM algorithm. Additionally, VC-dimension will replace the cardinality of finite function classes in the upper bounds on estimation error and sample complexity in the learning problem of infinite function classes.

Theorem 2.3.4. *Let L be a $0-1$ loss function and $\mathcal{H} \subseteq \mathcal{H}_L$ any function class with a finite VC-dimension d. Let A be any sample error minimization algorithm for $\mathcal{H}$ and L. Then A is a learning algorithm for $\mathcal{H}$ and its sample complexity satisfies*

$$m_{A,\mathcal{H}}(\varepsilon,\delta) \leq \frac{64}{\varepsilon^2}\left(2d\ln\left(\frac{12}{\varepsilon}\right) + \ln\left(\frac{4}{\delta}\right)\right),$$

and for every $m \geq d/2$ the estimation error of A satisfies

$$\varepsilon_{A,\mathcal{H}}(m,\delta) \leq \left(\frac{32}{m}\left(d\ln\left(\frac{2em}{d}\right) + \ln\left(\frac{4}{\delta}\right)\right)\right)^{1/2}.$$

□

Corollary 2.3.4.1. *Let L be a $0-1$ loss function and $\mathcal{P}_n \subseteq \mathcal{H}_L$ the class of functions calculated by a perceptron on n inputs. Then, any sample error minimization algorithm A for $\mathcal{P}_n$ and L is a learning algorithm for $\mathcal{P}_n$. Furthermore, A has sample complexity*

$$m_{A,\mathcal{P}_n}(\varepsilon,\delta) \leq \frac{64}{\varepsilon^2}\left(2n\ln\left(\frac{12}{\varepsilon}\right) + \ln\left(\frac{4}{\delta}\right)\right).$$

□

Any k-bit perceptron is a perceptron, and specifically, for the k-bit perceptron of n inputs we have $\mathcal{H}_{k\text{-bit}} \subseteq \mathcal{P}_n$. Let $S \subseteq \mathcal{X}$ be an arbitrary finite set. Then,

$$(\mathcal{H}_{k\text{-bit}})|_S \subseteq (\mathcal{P}_n)|_S \implies \text{VCdim}(\mathcal{H}_{k\text{-bit}}) \leq \text{VCdim}(\mathcal{P}_n) = n.$$

We may compare the upper bounds on sample complexities of Corrolary 2.3.4.1, given in terms of VC-dimension, and that of Corrolary 2.2.1.1 given in terms of a finite function class cardinality, that is, $|\mathcal{H}_{k\text{-bit}}|$. Corrolary 2.2.1.1 gives an upper bound of

$$m_{A,L}(\varepsilon, \delta) \leq \frac{2}{\varepsilon^2} \ln\left(\frac{2\,|\mathcal{H}_{k\text{-bit}}|}{\delta}\right) = \frac{2}{\varepsilon^2}\left(k(n+1) + \ln\left(\frac{2}{\delta}\right)\right) \tag{2.3}$$

on the sample complexity of any SEM learning algorithm for the k-bit perceptron of n inputs. On the other hand, Corrolary 2.3.4.1 gives the sample complexity bound of

$$m_{A,L}(\varepsilon, \delta) \leq \frac{64}{\varepsilon^2}\left(2\,\text{VCdim}(\mathcal{H}_{k\text{-bit}}) \ln\left(\frac{12}{\varepsilon}\right) + \ln\left(\frac{4}{\delta}\right)\right) = \frac{64}{\varepsilon^2}\left(2n \ln\left(\frac{12}{\varepsilon}\right) + \ln\left(\frac{4}{\delta}\right)\right) \tag{2.4}$$

for any SEM algorithm. Sample complexity from (2.4), given in terms of VC-dimension, in no way depends on k, the number of bits used for the binary representation of the values of weights and thresholds. For many values of ε and δ, relation (2.4) gives the worse sample complexity bound between the two. However, as it has no explicit dependency on k, interestingly, for large enough values of k, there exist ranges of ε and δ for which the bound from in terms of VC-dimension is better than the bound given in terms of $|\mathcal{H}_{k\text{-bit}}|$.

2.4 Fundamental Theorem of Statistical Learning

Can the upper bound on the sample complexity of an SEM algorithm be significantly lower? To answer this, we need to provide a lower bound on the sample complexity of an SEM algorithm as well. Additionally, might there exist some non-SEM algorithm with significantly lower sample complexity bound, compared to that of an SEM algorithm, also adequate for PAC learning? The following Theorem resolves these questions.

Theorem 2.4.1. *Let L be a $0-1$ loss function and $\mathcal{H} \subseteq \mathcal{H}_L$ any class of functions. Then $\mathcal{H}$ is learnable if and only if it has a finite VC-dimension. Furthermore, there exist constants $c_1, c_2 > 0$ such that the inherent sample complexity of the learning problem for $\mathcal{H}$ satisfies*

$$\frac{c_1}{\varepsilon^2}\left(\text{VCdim}(\mathcal{H}) + \ln\left(\frac{1}{\delta}\right)\right) \leq m_{\mathcal{H}}(\varepsilon, \delta) \leq \frac{c_2}{\varepsilon^2}\left(\text{VCdim}(\mathcal{H}) + \ln\left(\frac{1}{\delta}\right)\right)$$

for all $0 < \varepsilon < 1/40$ and $0 < \delta < 1/20$.

□

Remark 2.4.2. We'll use the *big* Θ *notation*[1] to describe the conclusion of the Theorem 2.4.1. Let $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ be any function. If there exisit some real constants $c_1, c_2 > 0$, a constant vector $a = (a_1, a_2, \ldots, a_n) \in \mathbb{R}^n$, and a function $g : \mathbb{R}^n \longrightarrow \mathbb{R}$ such that $c_1\, g(x) \leq f(x) \leq c_2\, g(x)$ for all $x \in \mathbb{R}^n$ such that $x_i \geq a_i$, we write $f(x) = \Theta(g(x))$.

Theorem 2.4.1 shows how no PAC learning algorithm can significantly outperform the SEM algorithm, and how the inherent sample complexity of any function class with a finite VC-dimension $\mathcal{H}$ satisfies

$$m_{\mathcal{H}}(\varepsilon, \delta) = \Theta\left(\frac{1}{\varepsilon^2}\left(\ln\left(\frac{1}{\delta}\right)\right)\right).$$

Another important consequence of Theorem 2.4.1 is the fact that if a class of functions is learnable, then it necessarily has a finite VC-dimension.

Combining all of the results so far, we obtain the central theorem of this chapter.

Theorem 2.4.3 (Fundamental Theorem of Statistical Learning)**.** *For the* $0-1$ *loss function* L *and any function class* $\mathcal{H} \subseteq \mathcal{H}_L$*, the following statements are equivalent.*

1. $\mathcal{H}$ *is PAC learnable.*
2. $\mathrm{VCdim}(\mathcal{H}) < \infty$.
3. $\mathcal{H}$ *has the following uniform convergence property of the sample error function: For every* $m \in \mathbb{N}$ *and every* $\delta \in (0, 1)$ *there exists* $\varepsilon_0 \in (0, 1)$ *such that*

$$\mathbb{P}^m\left(\sup_{h \in \mathcal{H}} \left|\mathrm{er}_{\mathbb{P},\mathrm{L}}(h) - \hat{\mathrm{er}}_{\mathrm{m},\mathrm{L}}(Z, h)\right| > \varepsilon_0(m, \delta)\right) < \delta,$$

   for every probability distribution $\mathbb{P}$ *on* $(\mathcal{Z}, \mathcal{V})$*, and*

$$\varepsilon_{\mathcal{H}}(m, \delta) = \Theta\left(\left(\frac{1}{m}\ln\left(\frac{1}{\delta}\right)\right)^{1/2}\right) \xrightarrow{m \to \infty} 0.$$

4. *The inherent sample complexity of* $\mathcal{H}$ *satisfies*

$$m_{\mathcal{H}}(\varepsilon, \delta) = \Theta\left(\frac{1}{\varepsilon^2}\ln\left(\frac{1}{\delta}\right)\right).$$

5. *The inherent estimation error of* $\mathcal{H}$ *satisfies*

$$\varepsilon_{\mathcal{H}}(m, \delta) = \Theta\left(\left(\frac{1}{m}\ln\left(\frac{1}{\delta}\right)\right)^{1/2}\right).$$

□

[1] Big Θ notation is simply the combination of the standard big $\mathcal{O}$ and big Ω notation, where $f(x) = \Theta(g(x))$ if and only if $f(x) = \mathcal{O}(g(x))$ and $f(x) = \Omega(g(x))$.

2.5 Vapnik-Chervonenkis Dimension of Multillayer Networks

Lastly, we calculate the VC-dimensions of multilayer networks. While the theorems that follow apply to a larger class of neural networks called the *feed-foreward networks*, we'll present them in terms of the most general neural network defined in Chapter 1 – the MLP.

Reminder. Concerning the MLP Definition 1.6.5, let $n, k \in \mathbb{N}$ and $J_1, J_2, \ldots, J_k \in \mathbb{N}$ be arbitrary natural numbers. Additionally, let $p \in \mathbb{N}$ denote the total number of adjustable parameters and $c \in \mathbb{N}$ denote the number of computational units of an MLP. Whenever all of the activation functions $\Phi_1, \Phi_2, \ldots, \Phi_{k+1}$ are the threshold activation function ϕ, we refer to such an MLP as a *Linear Threshold Network*. On the other hand, whenever all of the activation functions $\Phi_1, \Phi_2, \ldots, \Phi_{k+1}$ are the sigmoid activation function σ, we refer to such an MLP as a *Sigmoid Network*. For the threshold activation function ϕ and a sigmoid network f, we refer to the composition $F(X) = \phi(f(X) - 0.5)$ as a *Binary Sigmoid Network*.

A linear threshold network with one computational unit ($c = 1$) and n adjustable parameters ($p = n$) is exactly the perceptron of n inputs, known to have the VC-dimesion exactly n, which is equal to the number of parameters p. The following Theorem gives a general upper bound on the VC-dimension of any linear threshold network.

Theorem 2.5.1. *Let $\mathcal{H}_{LTN}$ be the class of functions computable by a linear threshold network with p adjustable parameters and c computation units. Then*

$$\text{VCdim}(\mathcal{H}_{LTN}) \leq 2p \log_2\left(\frac{2c}{\ln(2)}\right).$$

□

The following theorem provides an upper bound on the VC-dimension of binary sigmoid networks in terms of the number of adjustable parameters p and the number of computational units c.

Theorem 2.5.2. *Let $\mathcal{H}_{BSN}$ be the class of functions computable by a binary sigmoid network with p adjustable parameters and c computation units. Then*

$$\text{VCdim}(\mathcal{H}_{BSN}) \leq (pc)^2 + 11pc \log_2(18pc^2).$$

□

The Fundamental Theorem of Statistical Learning guarantees that function classes $\mathcal{H}_{\text{LTN}}$ and $\mathcal{H}_{\text{BSN}}$ are PAC learnable with respect to the $0-1$ loss function. This integrates every artificial neural network encountered in Chapter 1 into the statistical learning theory developed in this chapter.

Bibliography

[1] Charu C. Aggarwal et al., *Neural networks and deep learning*, Springer, 2018.

[2] M. Anthony and P. Bartlett, *Neural network learning - theoretical foundations.*, Cambridge University Press, 2002.

[3] G. Cybenko, *Approximation by superpositions of a sigmoidal function*, Mathematics of Control, Signals and Systems **2** (1989), 303 – 314, https://web.njit.edu/~usman/courses/cs675_fall18/10.1.1.441.7873.pdf, visited on 2024-04-16.

[4] Ke Lin Du and M. N.S. Swamy, *Neural networks and statistical learning*, Springer Publishing Company, Incorporated, 2013.

[5] Ken Ichi Funahashi, *On the approximate realization of continuous mappings by neural networks*, Neural Networks **2** (1989), no. 3, 183–192.

[6] D. O. Hebb, *Organization of behavior*, Wiley, 1949, https://pure.mpg.de/rest/items/item_2346268_3/component/file_2346267/content, visited on 2024-04-16.

[7] A. L. Hodgkin and A. F. Huxley, *A quantitative description of ion currents and its applications to conductance and excitation in nerve membranes*, Journal of Physiology **117** (1952), 500—544.

[8] K. Hornik, M. Stinchcombe, and H. White, *Multilayer feedforward networks are universal approximators*, Neural Networks **2** (1989), 359 – 366, https://cognitivemedium.com/magic_paper/assets/Hornik.pdf, visited on 2024-04-16.

[9] M. Leshno, V. Ya. Lin, A. Pinkus, and S. Schocken, *Multilayer feedforward networks with a nonpolynomial activation function can approximate any function.*, Neural Networks **6** (1993), no. 6, 861–867.

[10] J. Lighthill, *Artificial intelligence: A general survey*, Artificial Intelligence **5** (1973), no. 10, 1–21, https://www.chilton-computing.org.uk/inf/literature/reports/lighthill_report/p001.htm, visited on 2024-04-16.

[11] D. Masters and C. Luschi, *Revisiting small batch training for deep neural networks*, 2018, https://arxiv.org/pdf/1804.07612.pdf, visited on 2024-04-16.

[12] W.S. McCulloch and W. Pitts, *A logical calculus of the ideas immanent in nervous activity*, Bulletin of Mathematical Biophysics **2** (1943), 115 – 133, https://www.cs.cmu.edu/~./epxing/Class/10715/reading/McCulloch.and.Pitts.pdf, visited on 2024-04-16.

[13] M. Minsky and S. Papert, *Perceptrons; an introduction to computational geometry*, MIT Press, 1969.

[14] A. Pinkus, *Approximation theory of the mlp model in neural networks*, Acta Numerica **8** (1999), 143–195.

[15] F. Rosenblatt, *The perceptron - a perceiving and recognizing automaton*, Tech. Rep. 85-460-1, Cornell Aeronautical Laboratory, Ithaca, New York, January 1957.

[16] _____, *Principles of neurodynamics: Perceptrons and the theory of brain mechanisms*, Spartan Books, 1962.

[17] W. Rudin, *Principles of mathematical analysis*, International series in pure and applied mathematics, McGraw-Hill, 1976, ISBN 9780070856134, https://web.math.ucsb.edu/~agboola/teaching/2021/winter/122A/rudin.pdf, visited on 2024-04-16.

[18] D. Rumelhart, G. Hinton, and R. Williams, *Learning internal representations by error propagation*, Nature **323** (1986), no. 9, 533–536, https://gwern.net/doc/ai/nn/1986-rumelhart-2.pdf, visited on 2024-04-16.

[19] B. Widrow and M. E. Hoff, *Adaptive switching circuits*, Convention Record of IRE Eastern Electronic Show & Convention (WESCON1960) **4** (1960), 96–104, https://www-isl.stanford.edu/~widrow/papers/c1960adaptiveswitching.pdf, visited on 2024-04-16.

[20] B. Widrow and M. A. Lehr, *30 years of adaptive neural networks: Perceptron, madaline, and backpropagation*, Proceedings of the IEEE **78** (1990), 1415 – 1442, https://www-isl.stanford.edu/~widrow/papers/j199030years.pdf, visited on 2024-04-16.

Sažetak

Prvo poglavlje započinjemo s principima rada neurona i bioloških neuronskih mreža. Nakon dane motivacije, započinjemo s njihovim matematičkim modeliranjem umjetnim neuronskim mrežama. Prvo se susrećemo s McCulloch-Pittsovim neuronom – prvim matematičkim modelom neurona. Kako ima određene nedostatke, uvodimo perceptron kao općenitiji model umjetnog neurona. Uz perceptrona dolazi algoritam perceptronskog učenja te ADALINE algoritam učenja. Nadalje, uvodimo višeslojni perceptron kao matematički model bioloških neuronskih mreža te predstavljamo tzv. "Backpropagation" algoritam učenja.

Drugo poglavlje formalizira problem učenja jezikom Teorije Vjerojatnosti. Problem učenja smatramo pronalaskom optimalnih parametara umjetne neuronske mreže. Svaka umjetna neuronska mreža, kao parametarska funkcija, generira određenu klasu funkcija koje dobivamo variranjem njenih parametara. Cilj drugog poglavlja je odgovoriti na pitanje ima li smisla tražiti optimalne parametre modela unutar dane klase funkcija koju sam model generira. Pokazuje se da je odgovor potvrdan. Naime, postoji metrika zvana Vapnik-Chervonenkisova dimenzija, pridružena proizvoljnoj klasi funkcija, čija konačnost u potpunosti karakterizira naučivost same klase. Ovaj rezultat je iskazan kao Osnovni Teorem Statističkog Učenja. Na samom kraju pokazujemo kako sve klase funkcija koje generiraju umjetne neuronske mreže definirane u prvom poglavlju imaju konačnu Vapnik-Chervonenkisovu dimenziju.

Summary

We start the first chapter by explaining the workings of neurons and biological neural networks. After providing some motivation, we begin their mathematical modelling with artificial neural networks. We define the McCulloch-Pitts neuron – the first mathematical model of a neuron. As it has certain limitations, we introduce the perceptron as a more general model of an artificial neuron. Along with the perceptron come the perceptron learning algorithm and the ADALINE learning algorithm. Furthermore, we introduce the multilayer perceptron as a mathematical model of biological neural networks and present the so-called "Backpropagation" learning algorithm.

The second chapter formalizes the problem of learning in the language of Probability Theory. We consider the learning problem as the search for the optimal parameters of an artificial neural network. Each artificial neural network, as a parametric function, generates a certain class of functions that we obtain by varying its parameters. The goal of the second chapter is to answer the question of whether it makes sense to search for the optimal model parameters within a given class of functions generated by the model itself. It turns out that the answer is yes. Namely, there is a metric called the Vapnik-Chervonenkis dimension, associated with an arbitrary class of functions, whose finiteness completely characterizes the learnability of said class. This result is stated as the Fundamental Theorem of Statistical Learning. At the very end, we show how all classes of functions generated by artificial neural networks defined in the first chapter have a finite Vapnik-Chervonenkis dimension.

Životopis

Rođen sam 03.08.1997. godine u Slavonskom Brodu, gdje sam pohađao Osnovnu školu Đuro Pilar, nakon koje upisujem Tehničku školu Slavonski Brod. Godine 2016. upisujem preddiplomski sveučilišni studij Matematike na Prirodoslovno-matematičkom fakultetu u Zagrebu. Nakon stjecanja titule prvostupnika matematike, odlučujem se za diplomski sveučilišni studij Matematičke Statistike na istom fakultetu. Tijekom diplomskog studija najviše su me interesirala područja Strojnog Učenja i Umjetne Inteligencije.