MSE Master of Science in Engineering

Deep Learning

Lecture Notes for the module TSM-DeLearn (Part 1)

Prof. Dr. Klaus Zahn

Lucerne University of Applied Sciences and Arts
Engineering and Architecture

February 2023

Table of Content

1 Introduction to Artificial Intelligence and Deep Learning

Ever since humans have dreamed of “intelligent” machines i.e., of machines that think[1]. One famous but fake sample is the so-called “Mechanical Turk” that pretended to be able to play chess in the late 18th century[2].

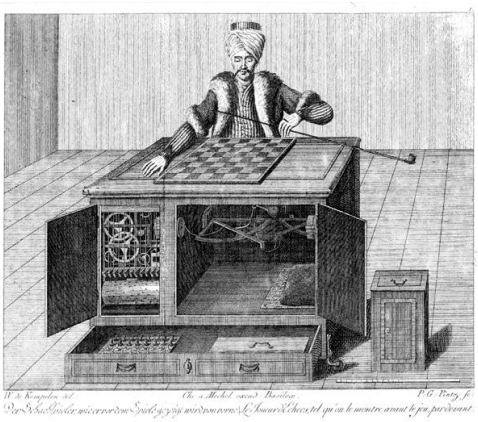


Figure 1: *«Mechanical Turk» a fake machine that pretended to play chess.*

The field of Artificial Intelligence (AI) emerged in the mid-fifties of the last century[3]. Since then, it has experienced various ups and downs, with the most recent hype – starting some 10 years ago – being triggered by so-called Deep Learning (DL). In Figure 2 the relation of DL to AI is shown. In fact, DL comprises specific types of Neural Networks (NN). NNs are a part of Machine Learning (ML), which itself is field of AI. While the terms are often used interchangeably in the public domain, they should be clearly distinguished.

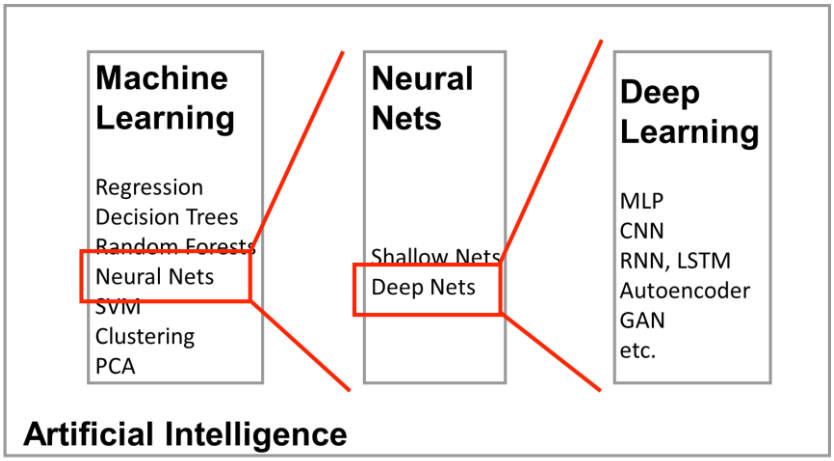


Figure 2: *The diagram shows the relation between Artificial Intelligence, Machine Learning, Neural Networks and Deep Learning.*

1.1 Successful Applications of Machine Learning and Deep Learning

As already mentioned, the last decade has seen the development of numerous ML systems often based on DL. Two major fields of development and application are Computer Vision (CV) and Natural Language Processing (NLP). In the following we will give a few examples for both[4].

[1] While we certainly have an intuitive idea of “intelligence” its formal definition is far from being obvious [2].
[2] https://en.wikipedia.org/wiki/Mechanical_Turk.
[3] We will give some historical reference in chapter 2.
[4] For a list of interesting applications see e.g., http://www.yaronhadad.com/deep-learning-most-amazing-applications/

1.1.1 Computer Vision

Due to the possibility of automatic object detection and recognition in images based on Convolutional Neural Networks (CNN) (Figure 3), various applications emerged or improved over the last decade.

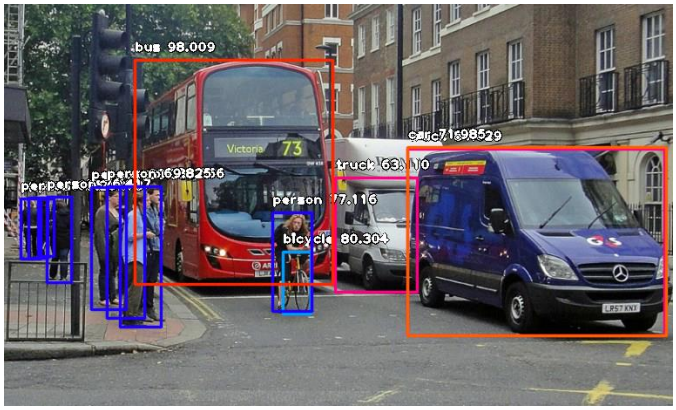


Figure 3: *Automatic object detection and recognition is now available with high accuracy[5].*

1.1.1.1 Optical Character Recognition

Optical Character Recognition OCR) (printed or handwritten) has been studied since the 1960s and its accuracy was considerably improved by DL techniques.

Figure 4: *The MNIST database[6] of handwritten digits is frequently used for illustration of the OCR task.*

1.1.1.2 Automatic Image Caption Generation

DL system can be used to automatically tag an image or a video i.e., describe the semantic content of a given scene.

Figure 5: *Example for automatic image caption generation[7].*

[5] https://towardsdatascience.com/object-detection-with-10-lines-of-code-d6cb4d86f606
[6] http://yann.lecun.com/exdb/mnist/
[7] https://github.com/akshaypunwatkar/Image-captioning-on-flickerdata

1.1.1.3 Self-Driving cars

Figure 6: *Article on the "Chaos" of 2004 Darpa Grand Challenge[8].*

While the 2004 Darpa Grand Challenge[9], a driverless car competition, ended with a complete failure of all vehicles (Figure 6) nowadays vehicles manage to navigate autonomously in complex environments (Figure 7).

Figure 7: *Example of an autonomous vehicle navigating in an urban environment[10].*

1.1.1.4 Medical diagnosis

Automatic treatment of data from medical imaging techniques have been used since the 1980s. During the last decade, based on DL systems, considerable progress could be achieved. The example given in Figure 8 shows results of a team from HSLU with a DL system able to recognize skin eczema.

Figure 8: *Example of a computer program that learned to recognize skin eczema[11].*

[8] https://www.wired.com/story/autonomous-car-chaos-2004-darpa-grand-challenge/
[9] https://en.wikipedia.org/wiki/DARPA_Grand_Challenge_(2004)
[10] https://youtu.be/__EoOvVkEMo
[11] https://hub.hslu.ch/informatik/mit-rechenpower-gegen-hautekzeme/

1.1.1.5 Lip synchronisation from audio signal

Figure 9: *Lip synchronisation from an audio signal[12]. The approach described here [3] shows an interesting hierarchy of methods.*

In the example given in Figure 9 a DL system was trained to produce a video, that reproduces the lip movement of a person based exclusively on the audio recording.

1.1.2 Natural Language Processing

Natural language processing (NLP) deals with the automatic processing of spoken or written language. Different subtopics that evolved considerably through the last decade from the progress in DL are given below.

1.1.2.1 Automatic Speech Recognition

Automatic Speech Recognition (ASR) i.e., the possibility to recognize spoken language from an audio signal and transform it to text, considerably improved using DL systems (in particular Recurrent Neural Networks, RNN) during the last decade and is now the basis of various applications. Based on ASR communication with a personal assistant (Alexa, Cortana, Siri) is now ubiquitous[13].

Figure 10: *Automatic speech recognition based on Recurrent Neural Networks is now achieving impressive performance.*

1.1.2.2 Machine Translation

Machine translation also has been a topic since the 1960s and till the 2000s results were not very encouraging. Meanwhile, free translation programs based on DL are available offering very good quality[14,15]. However, languages with very simple grammar like Chinese may require context knowledge for a correct translation and still require certain manual tweaking for a good quality (Figure 11).

[12] https://www.youtube.com/watch?v=MVBe6_o4cMI
[13] https://youtu.be/UOEIH2l9z7c
[14] https://translate.google.com
[15] https://www.deepl.com

Figure 11: *Comparison of a machine translation from Chinese to French for two different SW packages (top: https://translate.google.com , bottom: https://www.deepl.com).*

In that context one should also note that the so-called “human performance”, expression which is frequently cited in the media in relation with nowadays DL systems, is somewhat relative. Thus, any German native speaker would understand the following sentence without too much trouble, even though only the first and the last letters of each word are correct while the centre part has been permutated:

Gmäess eneir Sutide eneir elgnihcesn Uvinisterät ist es nchit witihcg, in wlecehr Rneflogheie die Bstachuebn in eneim Wrot snid, das ezniige was wcthiig ist, ist, dass der estre und der leztte Bstabchue an der ritihcegn Pstoiion snid. Der Rset knan ein ttoaelr Bsinöldn sien, tedztorm knan man ihn onhe Pemoblre lseen. Das ist so, wiel wir nciht jeedn Bstachuebn enzelin leesn, snderon das Wrot als gseatems.

Any machine learning program however will fail completely to translate even a single (of the longer) word.

1.1.2.3 Text Classification

Automatic Text Classification consists of automatically assigning a document to one or more membership classes. This is a fundamental task in many scenarios. For example, in social media monitoring it is essential to classify tweets referring to a specific “brand” as positive or negative opinions. Or in the case of search engines, it is possible to significantly improve their accuracy if the indexed documents are classified in relation to the topic at hand – so that users can more easily identify the texts that interest them[16].

Exercise:

In groups of 2, address the following question(s):

- Machines (IBM “Deep Blue”) could play chess at best human level back in 1997. With Go this took almost 20 years longer[17]. Why?
- How could you design (just conceptually) a simple program playing chess? Think about the value of the figures and the possible moves they can make:

Symbol	♙	♘	♗	♖	♕
Piece	pawn	knight	bishop	rook	queen
Value	1	3	3	5	9

- What could be the problem with Go?

[16] https://stackabuse.com/text-classification-with-python-and-scikit-learn/
[17] https://en.wikipedia.org/wiki/AlphaGo

1.2 Definition of Deep Learning

Because Deep Learning is a subtopic of Machine Learning (c.f. Figure 2) we start with a definition of the latter.

1.2.1 Definition of Machine Learning

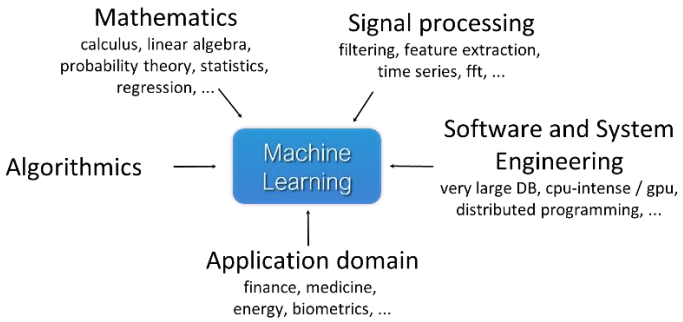


Figure 12: *Relation of Machine Learning with connected fields.*

Machine Learning is at the convergence of different fields (Figure 12). Like the term *machine intelligence*[1] there is no unique definition of Machine Learning and different versions are used. Three examples are cited below:

> Machine learning consists of computer methods that analyse observation data to automatically detect patterns, and then use the uncovered patterns to perform tasks based on new unobserved data.

> Machine Learning could be defined as a set of methods that automatically detect patterns in data, and then use the uncovered patterns to predict future data, or to perform other kinds of decision making under uncertainty [4].

> A machine learning program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E [5].

The following Figure 13 tries to illustrate these abstract definitions somewhat more in detail:

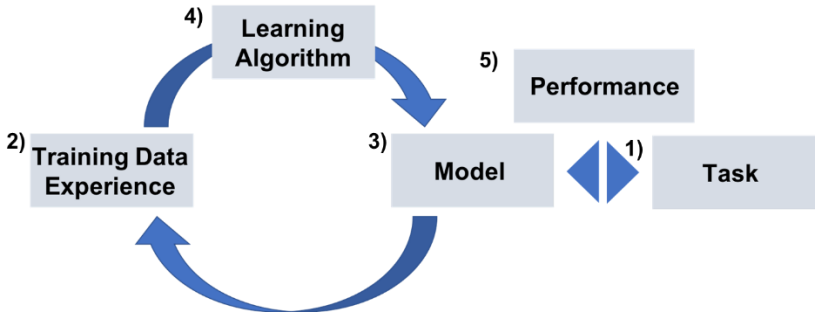


Figure 13: *Machine Learning involves shaping a model (hypothesis function) that can capture the structure seen in the data.*

1) At the beginning, we have a problem – i.e., a task – which we try to solve with ML concepts.
2) We have some sample (“training”) data available related to this task. Depending on the context this can be raw observed data or pre-processed data with manually engineered features.
3) A model suitable to solve the task-problem will be defined with the goal to capture (some of)

the patterns observed in the training data.

4) A learning algorithm will allow to optimize the parameters of the model through a suitable cost function (or loss, benefit, reward function).
5) The performance of our ML model for the task will be as good as it captures the statistical distribution hidden in the data.

We will try to further illustrate this definition of the ML steps using the following concrete example of a face recognition task: imagine we want to recognize a set of faces in a series of images. Seven different categories are possible as shown in Figure 14.

Figure 14: *The task will be the recognition of different faces in an image.*

The problem of face recognition can be divided in three subtasks:

1) Face detection: find the region(s) of face(s) in the image.
2) Feature extraction: find suitable characteristics allowing to characterize the different categories.
3) Face matching: match a given face with one of the possible categories.

To illustrate the three steps the Betaface API[18] is used and the results as shown in the following Figure 15 are obtained.

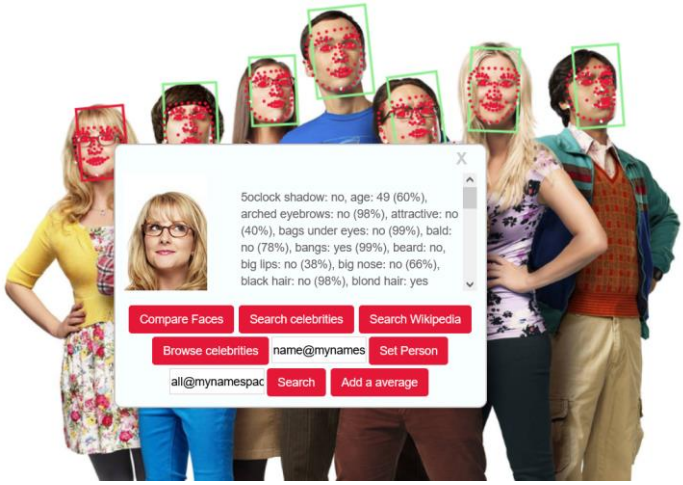


Figure 15: *Face recognition using the Betaface API.*

[18] https://www.betafaceapi.com/demo.html

The three subtasks are performed as follows with the API:

1) After the upload of the image, the faces are detected automatically (green rectangles).
2) In addition, the feature extraction step represented by the red dots is performed automatically in the background. The dots correspond to the position of the eyes, the nose, and the mouth[19].
3) For the recognition step, one of the faces can be selected and “Compare Faces” or “Search celebrities” can be selected.

Now we can make the definition given in Figure 13 more concrete:

Supervised Learning:
The goal of the ML algorithm is to extract most relevant features **x** from the raw observation data **o** and learn a mapping from inputs **x** to outputs **y** given a set of example data pairs **(x,y)** called the training set.

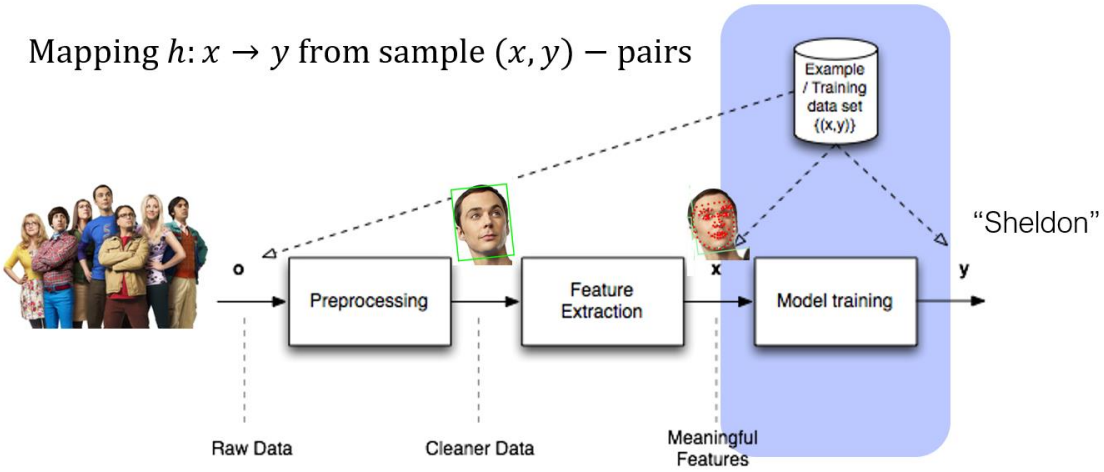


Figure 16: *Supervised machine learning approach for the face recognition problem.*

In our case the features **x** could be the information coded in the red dots and the output **y** is the name of the respective person. The training set consists of a certain number of feature sets **x** extracted for all seven faces. This learning paradigm is called Supervised Learning because the match between the features **x** and the output **y** is given. We will address two further important ML schemes below.
When applying this ML scheme immediately the following questions or problems arise:

1) Problem of sufficient training data
   A large quantity of training data labelled and validated by humans is required consisting of face images for the seven classes under various conditions (below class “Sheldon”).

[19] We do not claim that these features are actually used by the API for matching the faces and use them only for illustration purposes.

2) Can we deal with high variability i.e., do we generalize well enough
The data may show a high variability (i.e., growing barb changing hair style) which may increase even further the requirements on the training set size and raise the question, whether under such challenging conditions our ML algorithm will be able to generalize i.e., will be able to deal with unknown data showing such high variability.

3) Optimal choice of features
It is a priori unclear, which features **x** extracted from the raw face data **o** will deal best with the high variability of the data set and provide maximum generalization performance. A lot of time and effort will be required to design and optimize these hand-craft features.

We will show in the next chapter that Deep Learning addresses exactly these problems, in particular it will provide a solution to the manual feature extraction problem. Before that, we will make two further points about the testing of the ML algorithms and additional ML paradigms.

1.2.1.1 Testing of a ML algorithm

The flow chart in Figure 16 represents the training step. In addition, we must test the performance of our ML model as shown in Figure 17. This will require addition data independent from the training set and further increase the requirements on the set size.

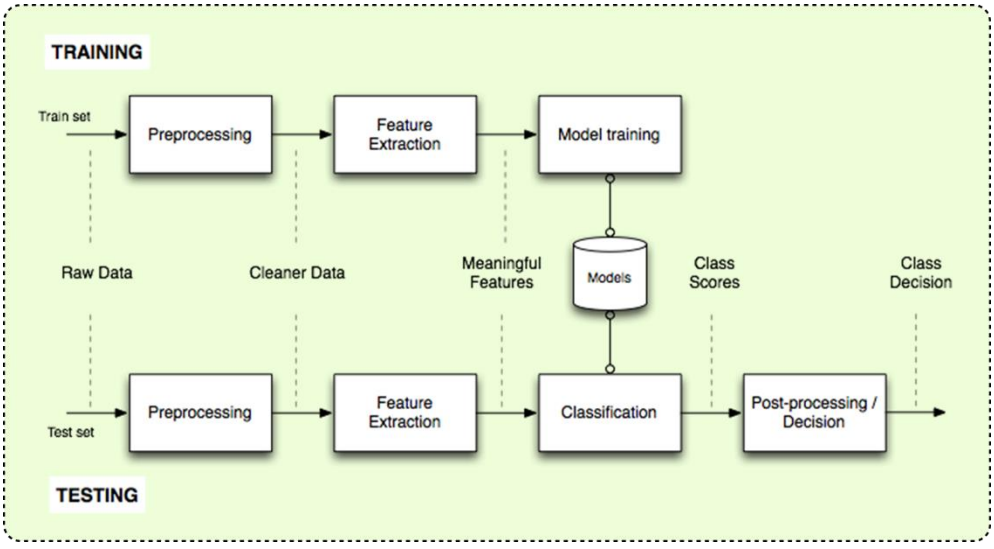


Figure 17: *Testing the model after the training process is important to evaluate the performance of the ML approach.*

1.2.1.2 Further Machine Learning Paradigms

In addition to the supervised ML approach discussed above, two other learning approaches are important to practice:

Unsupervised Learning
While for supervised learning the mapping between the input features **x** and the output **y** is given, the goal of unsupervised learning is to discover patterns or structures in the input **x** automatically i.e., form unlabelled data.

Unsupervised Learning:
The goal of the ML algorithm is to discover interesting **structures** from inputs **x** given a set of data called the training set.

Typical applications include automatic clustering of data into groups, where ML determines both visual features for clustering and the groups themselves. Some typical fields of applications are given in Figure 18 below.

Figure 18: *Typical fields of application for unsupervised ML approaches.*

Reinforcement Learning

Reinforcement Learning:
The goal of the ML algorithm is to train an autonomous agent to perform a task by trial and error, without any guidance from the human operator.

During the training process the agent takes actions on a trial-and-error basis in an environment and optimization is done by maximizing some reward function.

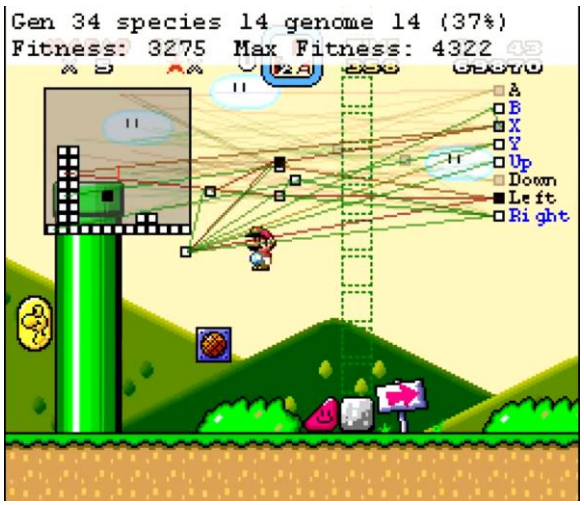


Figure 19: *Reinforcement learning allows computers to "play" video games*[20].

[20] https://www.youtube.com/watch?v=qv6UVOQ0F44

Two typical applications are shown in Figure 19, a computer trained to play the game Super Mario, and Figure 20, robots trained to perform various tasks, respectively.

Figure 20: *Reinforcement learning used to train robots performing various tasks*[21].

Exercise:

In groups of 2, do the following:

- Watch (parts of) the video with explanation on how computers learn a game, here Super Mario: https://www.youtube.com/watch?v=qv6UVOQ0F44
- From the video and through discussions try to figure out how the main principles of reinforcement learning work.
- Be prepared to explain these principles in simple terms to students of the other groups.

1.2.2 Definition of Deep Learning

DL is a recent trend in the field of machine learning at the convergence of:

Availability of increasingly larger training data sets

The following Figure 21 shows the (increasing) size of various well-known datasets over the time. The scale is double logarithmic thus could be interpreted as a form of Moore's law[22].

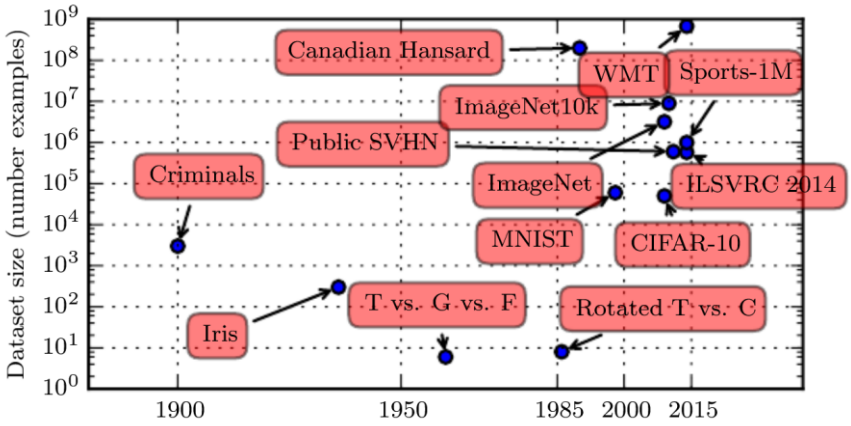


Figure 21: *Increasing data set over time. Note the double logarithmic scale [1].*

[21] https://youtu.be/rVlhMGQgDkY , https://youtu.be/tf7IEVTDjng
[22] https://en.wikipedia.org/wiki/Moore%27s_law

We already stressed the importance of sufficiently large datasets in the previous chapter (c.f. point 2) and this becomes even more relevant for the training of deep ML architectures.

Availability of increasingly larger processing performance
Due to the steady increase in computing power in particular also due to the availability of Graphics Processing Unit (GPU) allowing to massively parallelize certain algorithms, the complexity of the ML algorithms steadily increased over time. Figure 22 shows the increasing size of various artificial neural network architectures over time, represented by the blue dots, and compared to biological NNs. Thus e.g., in the 1990s a typical neural network consisted of some few hundred neurons corresponding to 50k parameters in total. At that time, the training would require months of CPU time on a CPU. Currently NN have on the order of a million neurons, corresponding to some 100M parameters. Nevertheless, training of such large NNs is feasible in "only" days/weeks of GPU time.

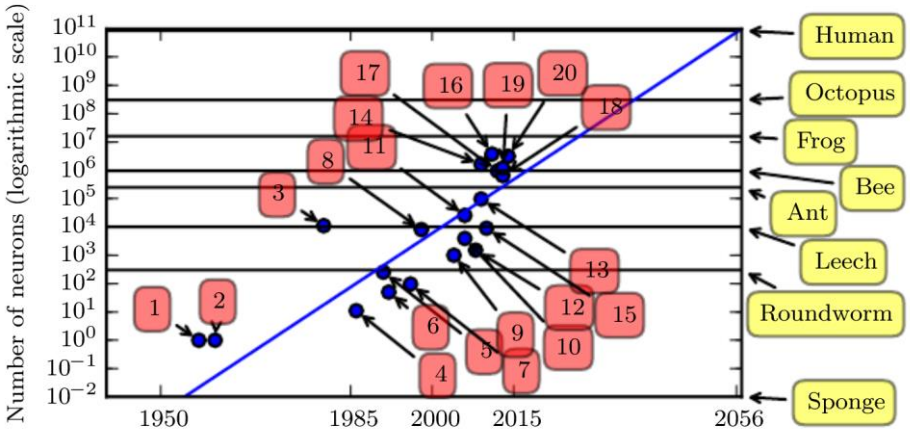


Figure 22: *Increasing size of various artificial neural networks over time, represented by the blue dots, and compared to biological NNs[23]. Since the introduction of hidden units, artificial neural networks have doubled in size roughly every 2.4 years [1].*

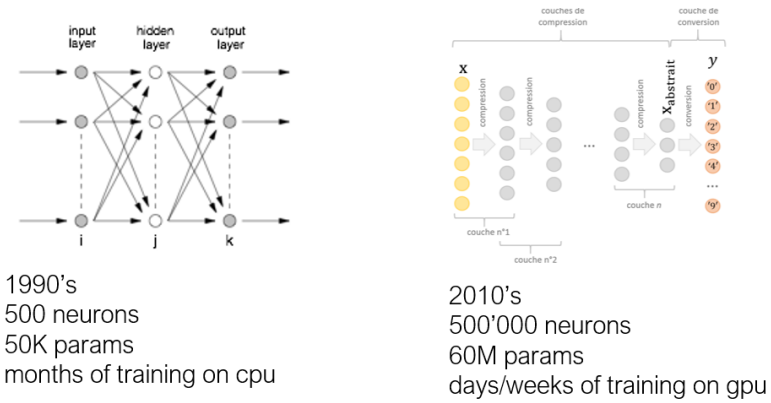


Figure 23: *Evolution of NN-size and training time from the 1990s till now.*

Availability of new algorithms
In addition to the increase in dataset size and processing power, ML algorithms have also improved, a topic we will discuss in the following chapters, especially in section 7.

So far, we have only explained the driving forces of DL, but not its main advantages with respect to standard or "classical" ML approaches. Recall the problems we stated in the previous chapter concerning the size of the available training dataset and the manual optimization of the feature extraction for maximum generalization performance i.e., points 1), 2), and 3). In Figure 24 (top) the ML approach as discussed in the previous chapter is shown. It involves the manual labelling of the faces in the images (Preprocessing) to obtain the clean input data for the feature extraction step. Then a further manual step is required, being the optimization of features to extract the input data **x** for the ML algorithm.

[23] For a legend of the numbers c.f. [1], part 1 Introduction, page 23.

The corresponding DL approach is shown in bottom part of the Figure 24. Deep ML architecture can work directly on the raw input data **o**, detect the faces (i.e., perform the pre-processing step) and automatically extract the features. Thus, the manual labelling step of the faces in the images is done automatically, which will considerably increase the available data set (only the final class must be defined manually). Furthermore, the manual feature engineering step is done automatically which will considerably simplify the application and optimize the generalization performance. Especially the last point was one of the major gains of DL with respect to ML (Figure 25).

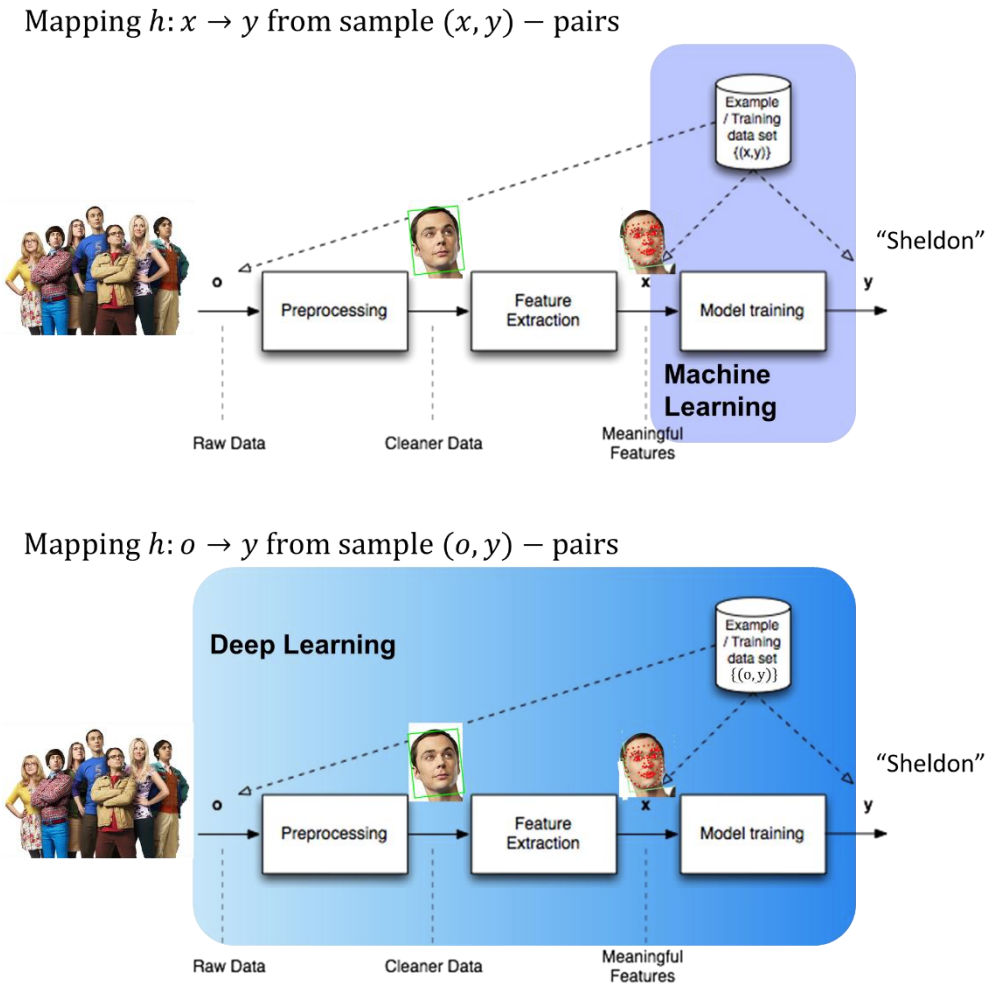


Figure 24: *The DL approach (bottom) can work directly on the raw input data **o** and automize the Preprocessing and Feature Extraction/Optimization step, which are both manual for the ML approach (top).*

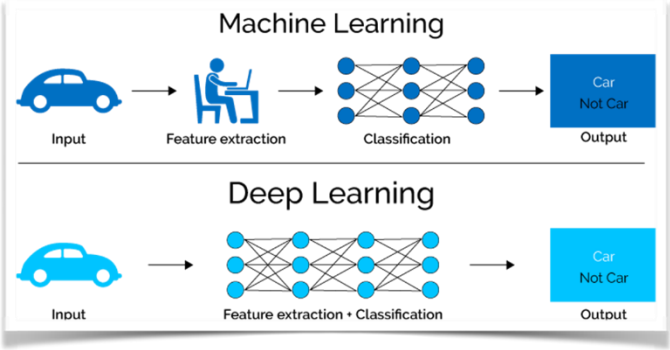


Figure 25: *DL (bottom) allows to skip the step of the hand-engineered features in ML (top) which will considerably save time and improve the result.*

2 First Neural Networks – a Historical Perspective

2.1 Biological Neural Systems

Neuroscience i.e., the scientific study of biological neurons and the nervous system in general was an important driver for the early developments of the field of artificial intelligence. Historically, in Neural Network Science the Neuro-Scientific Perspective was prevalent with the goal to understand the brain and the principles underlying human intelligence as well as the intention to model intelligent behaviour by artificial neural networks mimicking the biological brain[24]. Nowadays, NN-science is not necessarily related to an attempt to model the biological functions. The focus is simply the search for models and learning algorithms that fulfil the desired tasks, independent of whether a counterparts exist on the biological side. Nevertheless, the understanding of the biological neuron and its connections to other neurons forming a Neural Network has interesting analogies to Artificial Neural Networks (ANN).

Like all biological systems, neurons are extremely complex, and the following simplified explanations are mainly intended to illustrate the ANN below. A typical neuron consists of a cell body, dendrites, and a single axon (Figure 26). The axon and dendrites are filaments connected to the compact cell body. The dendrites protrude a few hundred micrometres from the cell body, while the axon in humans can be up to one meter long. Via the dendrites the neuron receives excitatory or inhibitory signals from other neurons and accumulates them. If these signals cumulate up to a certain activation potential an output signal is sent down the axon ("the neuron fires"). This signal will then be received by other neurons through connections of the synaptic terminals.

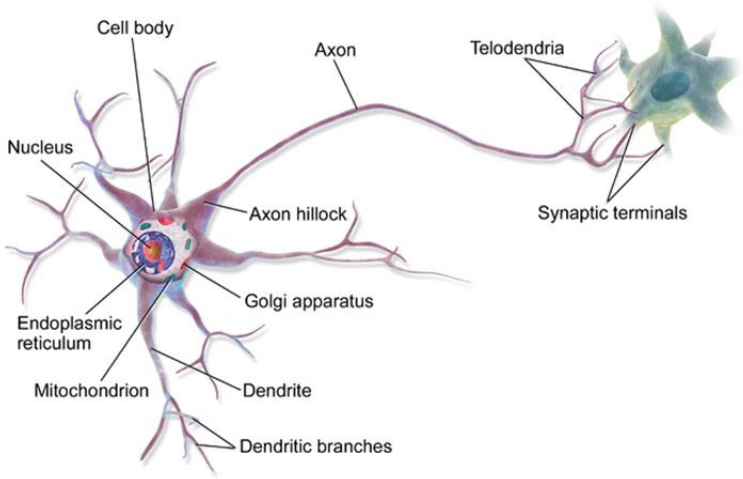


Figure 26: *Representation of a biological neuron (details see text)[25].*

Due to the mutual connections of the neurons, they form a complex neural network (Figure 27).

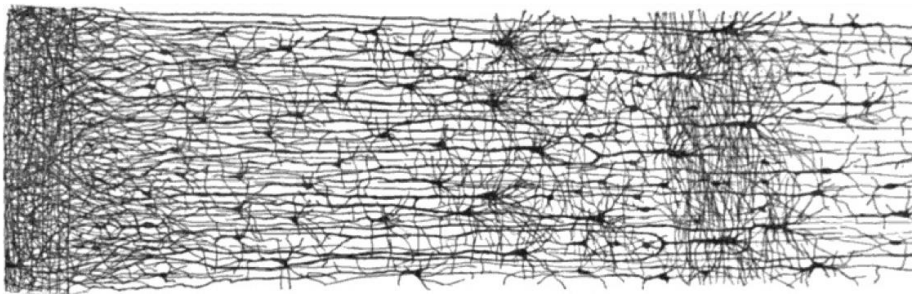


Figure 27: *Cross section of the visual cortex showing neurons connected to a network[26].*

There are further interesting correspondences between biological and artificial neural networks.

[24] A more recent initiative in that sense is the Human Brain Project of EC: https://www.humanbrainproject.eu/en/
[25] https://en.wikipedia.org/wiki/Neuron
[26] https://en.wikipedia.org/wiki/Cerebral_cortex

- Hierarchical structures:
  The visual cortex of the human brain is organized in hierarchical structures. Thus, the visual information generated by the eyes during vision is routed through different regions in the brain (Figure 28: V1, V2, ...). The different regions are responsible for different aspects of the seeing process:
  - Regions that are passed through earlier (activated earlier) are responsible for simple features/aspects, for recognising simple patterns like edges, corners, ...
  - Regions that are passed through later are responsible for aspects of higher complexity.

  This aspect observed in biological systems is also very important for artificial neural networks especially the architecture of CNNs.

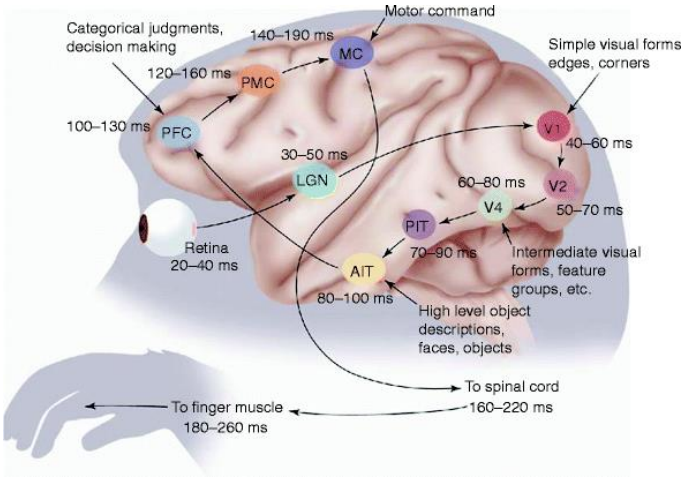


Figure 28: *The visual cortex of a human is organized in hierarchical structures. This bears similarities to CNNs.*

- Synaptic Pruning:
  During adolescence the number of neurons and synapses are reduced by up to 50% to build up more complex and more efficient structures. A similar idea is adopted for ANNs – a so called regularisation procedure that also aims at making the network structure in some sense more efficient (as we will see later in the course).

2.2 Artificial Neurons

2.2.1 McCulloch-Pitts Neuron

The first theoretical model for the activation of a neuron was developed in 1943 by Warren McCulloch and Walter Pitts (Figure 29, bottom). The idea was inspired by the functioning of biological neurons as discussed in the previous chapter. I.e., each neuron receives input signals from its dendrites and produces output signals along its (single) axon. The axon eventually branches out and connects via synapses to dendrites of other neurons, thus forming a neural network.
In the computational model of a neuron, it receives the signals x_k from the axons of the previous neurons (situated to the left and not drawn). The signals x_k are weighted with the synaptic strength at that synapse w_k in multiplicative manner $x_k \cdot w_k$. The idea is that the synaptic strengths w_k are learnable and control the strength of influence and direction of one neuron on another, which can be excitatory (positive weight) or inhibitory (negative weight). In the basic model, the dendrites carry the signal to the cell body where they all get summed up. If the final sum is above a certain threshold θ, the neuron will fire, sending a spike along its axon. For the McColloch-Pitts Neuron the activation function used is the Heaviside function, defined as:

$$H(z) = \begin{cases} 1 & (z \geq 0) \\ 0 & (z < 0) \end{cases}$$

Furthermore, the activations x_k are supposed to be binary and the weights w_k of magnitude 1:

$$x_k = 0{,}1$$

$$w_k = \pm 1 \quad \text{(+1 excitatory or -1 inhibitory)}$$

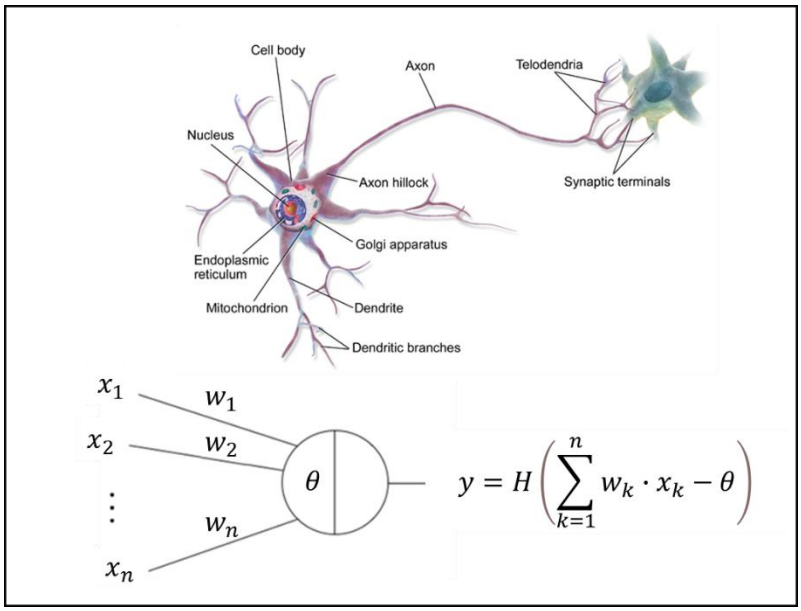


Figure 29: *The McCulloch-Pitts Neuron (bottom) was the first model for the activation of biological neurons (top) (details c.f. text).*

Despite the simple nature of the McCulloch-Pitts neuron its computational power is quite strong. McCulloch and Pitts showed that, in principle, any logical or arithmetic function can be computed with networks of such neurons[27]. Three examples for simple logical functions are given below:

AND: $y = H(x_1 + x_2 - 2)$

OR: $y = H(x_1 + x_2 - 1)$

XOR: $y = H(H(x_1 + x_2 - 1) + H(1 - x_1 - x_2) - 2)$

In Figure 30 the XOR function is (in addition to the formula above) represented in graphical notation, with the respective weights and thresholds given along the lines and in the neurons, respectively.

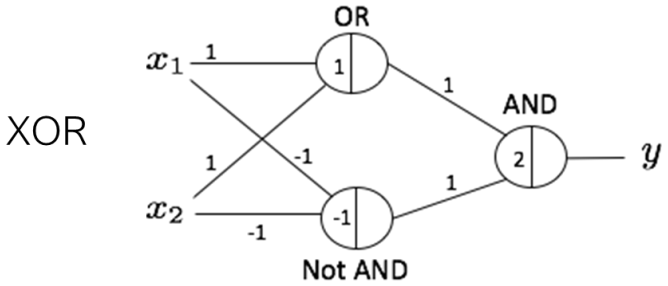


Figure 30: *Representation of the XOR function using three McCulloch-Pitts neurons. The binary inputs x_1 and x_2 will give the output y. The numbers along the lines represent the respective weights w_k, the numbers in the neurons represent the respective thresholds θ.*

Exercise:

In groups of 2, «prove» that the definitions given for OR and XOR are correct:

The straightforward way is to establish the missing tables like for AND

x_1	0	0	1	1
x_2	0	1	0	1
$x_1 + x_2$	0	1	1	2
$x_1 + x_2 - 2$	-2	-1	-1	0
y	0	0	0	1

[27] https://link.springer.com/article/10.1007%2FBF02478259

While the McCullogh-Pitts neurons is certainly an interesting concept, the following question immediately arises for a given problem:

How should the network structure and the weights be chosen?

In fact, no algorithm for selecting the weights is available and these must be determined manually. One first possible solution was given by the Hebb's Learning Hypothesis[28] in 1949 on how (biological) neurons might connect in networks. Its general idea is:

Any two cells or systems of cells that are repeatedly active at the same time tend to become "associated" so that activity in one facilitates activity in the other.
Hebb: *"… axon of the first cell develops synaptic knobs (…) in contact with the soma[29] of the second …"*.
In brief: *"Cells that fire together wire together."*

While this learning rule is not proven from a biological point of view it is nevertheless considered as the first rule for "self-organised learning". However, when taken as a principle for ANNs to adjust connection weights this does not provide a stable learning procedure.

2.2.2 Rosenblatt's Perceptron

A milestone in the development of ANN was the single Perceptron, referred to as Linear Threshold Unit (LTU), developed by Frank Rosenblatt in 1958. The activation function is calculated similar to the McCullogh-Pitts neurons but allows for arbitrary real inputs, weights and biases i.e.:

$$y = H\left(\sum_{k=1}^{n} w_k \cdot x_k + b\right) \quad \text{for } x_k, w_k, b \in \mathbb{R}$$

Due to the Heaviside function the output y still only takes the values 0 and 1.

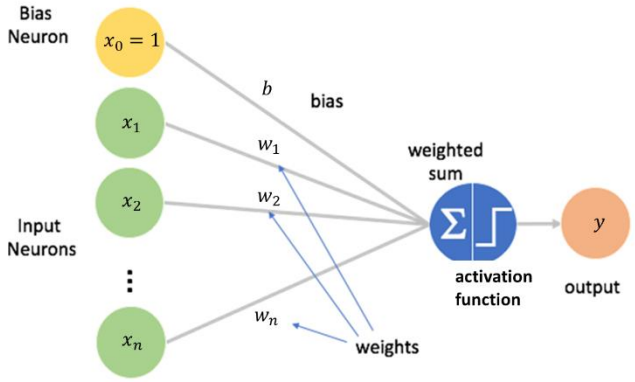


Figure 31: *Graphical representation of the single Perceptron also referred to as Linear Threshold Unit.*

In Figure 31 a graphical representation of the Perceptron is given. The input neurons x_k can take any real value. The neuron x_0 corresponding to the index 0 is the so-called bias neuron and its input is always equal to 1. The weights w_k and the bias b may take any real value and sum up according to the formula given above (represented by the Σ-sign in the figure). Finally, the Heaviside step function is applied to give the binary output y.
The bias specifies how much input signal is needed to activate the neuron or whether the neuron is already activated even if there is no input signal. In this sense, it can be seen as a kind of a priori information or default activation level for the neuron.

2.2.2.1 Computational Capabilities of a Single Perceptron

What kind of tasks can be performed with a single Perceptron unit?

[28] https://en.wikipedia.org/wiki/Hebbian_theory
[29] Cell body.

The set of points[30] $H_{w,b}$ defined as[31]

$$H_{w,b} = \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{w} \cdot \mathbf{x} + b = 0\}$$

represents a hyperplane in $\mathbb{R}^n$. Here both $\mathbf{w}$ and $\mathbf{x}$ are vectors of dimension n:

$$\mathbf{w} = \begin{pmatrix} w_1 \\ \vdots \\ w_n \end{pmatrix} \quad , \quad \mathbf{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$$

For any point $\mathbf{x}$ not lying on the hyperplane the result of $\mathbf{w} \cdot \mathbf{x} + b$ will be greater or smaller than zero, depending on which side of the hyperplane the point $\mathbf{x}$ is lying. Thus, the Perceptron is suited for **linearly separable binary classification** problems.

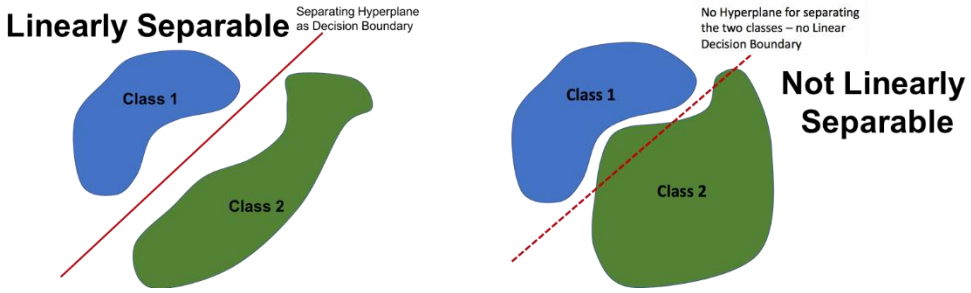


Figure 32: *The single Perceptron is suited for linearly separable binary classification problems.*

A simple example is given in Figure 33, with the values:

$$\mathbf{w} = \begin{pmatrix} w_1 \\ w_2 \end{pmatrix} = \begin{pmatrix} 0.5 \\ 1.0 \end{pmatrix} \; , \; b = -0.3$$

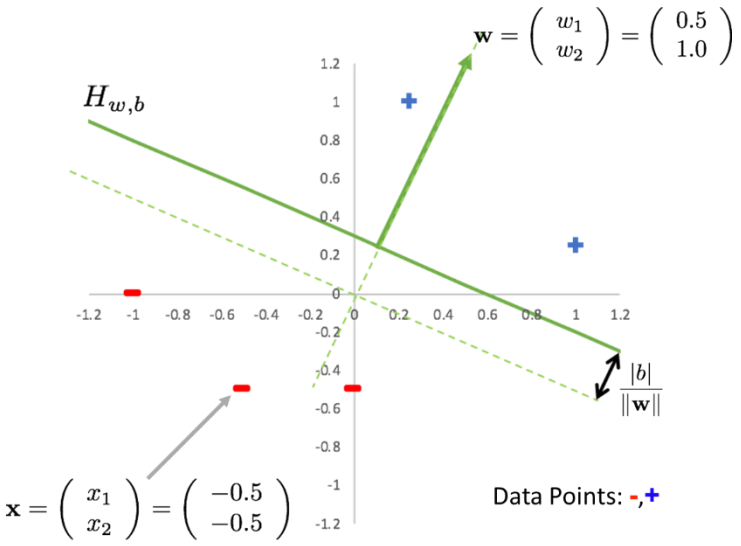


Figure 33: *Illustration of the decision boundary defined by a single Perceptron.*

As is well known form elementary geometry the vector $\mathbf{w}$ is orthogonal to the decision boundary and the value

[30] The indices w, b express the dependencies upon the weights and the bias.
[31] Here, the sum over the inputs x_k and weights w_k is written as scalar product of the vectors $\mathbf{x}$ and $\mathbf{w}$.

$$\frac{b}{\|\mathbf{w}\|}$$

represents the distance of the hyperplane to the origin.
If we calculate for the given point (-) $\mathbf{x} = (-0.5, -0.5)$ the result of $\mathbf{w} \cdot \mathbf{x} + b$ we find:

$$\mathbf{w} \cdot \mathbf{x} + b = 0.5 \cdot (-0.5) + 1 \cdot (-0.5) - 0.3 = -1.05$$

A value, which is effectively less than zero as is true for all other points $\mathbf{x}$ on this side of the hyperplane.

2.2.2.2 Perceptron Learning Algorithm

The main progress of the Perceptron with respect to the McCulloch-Pitts neuron is, that a robust learning algorithm can be formulated, the so-called *Perceptron Learning Algorithm*.
It is an iterative algorithm that will determine the weight vector $\mathbf{w}$ and bias b and works according to the following set of rules:
Given is a set of labelled data $\{(\mathbf{x}^{(i)}, y^{(i)}) | \; i = 1, \ldots, N\}$, where $\mathbf{x}^{(i)}$ are n-dimensional vectors.

1) Initialize the weight vector $\mathbf{w}$ and the bias b with zero or small random numbers
2) Iterate by updating the weight vector $\mathbf{w}$ and the bias b according to:
   a. Pick an arbitrary sample: $(\mathbf{x}^{(i)}, y^{(i)})$
   b. Compute the predicted value: $\hat{y}^{(i)} = H(\mathbf{w} \cdot \mathbf{x}^{(i)} + b)$
   c. Perform the parameter update:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \cdot (\hat{y}^{(i)} - y^{(i)}) \cdot \mathbf{x}^{(i)}$$

$$b \leftarrow b - \alpha \cdot \underbrace{(\hat{y}^{(i)} - y^{(i)})}_{0,+1,-1}$$

The learning rate $\alpha > 0$ (e.g. $\alpha = 0.1$) is a parameter of the algorithm.

The following can be stated:

- Obviously, an update occurs only in case that the selected sample $(\mathbf{x}^{(i)}, y^{(i)})$ is misclassified i.e., for: $\hat{y}^{(i)} = H(\mathbf{w} \cdot \mathbf{x}^{(i)} + b) \neq y^{(i)}$
- The learning rule searches for a weight vector $\mathbf{w}$ and a bias b which define a hyperplane that separates the points associated with the two classes. This is only possible for linearly separable input sets.
- In case of misclassified points, the weight update rule leads to a correction of the weight vector $\mathbf{w}$ such that the hyperplane is tilted to (rather) bring the misclassified points to the correct side. The bias update rule will simply pull the hyperplane closer to the misclassified points by keeping its orientation.

In the following Figure 34 the Perceptron Learning Algorithm is illustrated by applying one update step.

On the left-hand side, the situation before the update is shown. The values of the weights and bias are:

$$\mathbf{w}_0 = \begin{pmatrix} 1 \\ 0.5 \end{pmatrix}, b_0 = 0.5$$

The index 0 indicates, that both values are the start values at iteration step zero. The corresponding decision boundary is shown in black. The datapoint (-) $\mathbf{x} = (0, -0.5)$ is misclassified (it is above the boundary on the + side i.e., $y^{(i)} = 0$ but $\hat{y}^{(i)} = 1$) and therefore used to apply the update step. We use a learning rate $\alpha = 0.5$ (note $\hat{y}^{(i)} - y^{(i)} = 1$).

- Update rule for weights: $\mathbf{w}_1 = \begin{pmatrix} 1 \\ 0.5 \end{pmatrix} - \alpha \begin{pmatrix} 0 \\ -0.5 \end{pmatrix} = \begin{pmatrix} 1 \\ 0.75 \end{pmatrix}$
- Update rule for bias: $b_1 = b_0 - \alpha = 0$

The new hyperplane (green) on the right-hand side of Figure 34 results, which represents a perfect separation of the two pointsets. We see in fact, that the weight update tilts the boundary counter clock-wise such that the misclassified point moves to the correct side. Furthermore, the bias update pushes the boundary upwards towards the origin with the same goal.

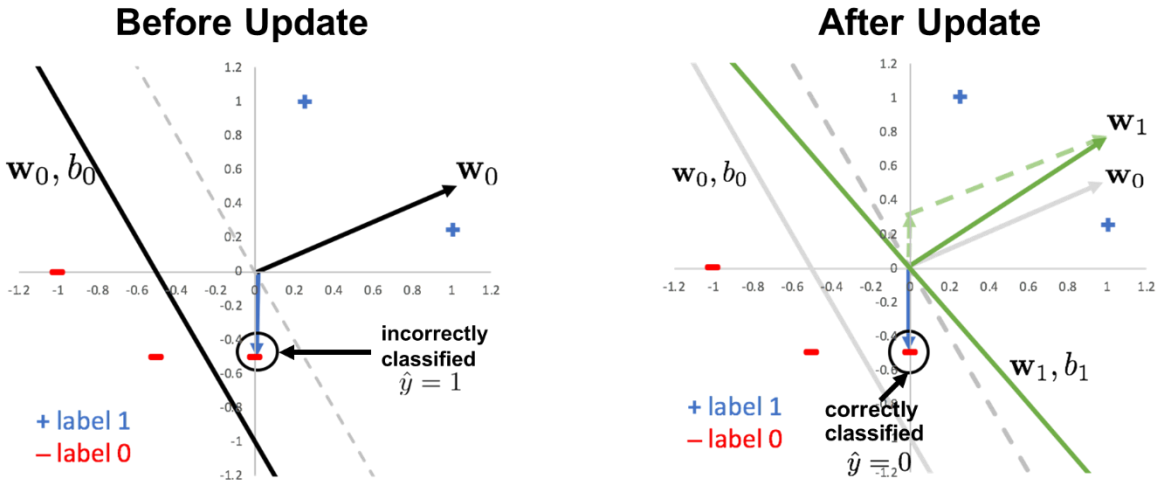


Figure 34: *Illustration of the Perceptron Learning Algorithm (details see text).*

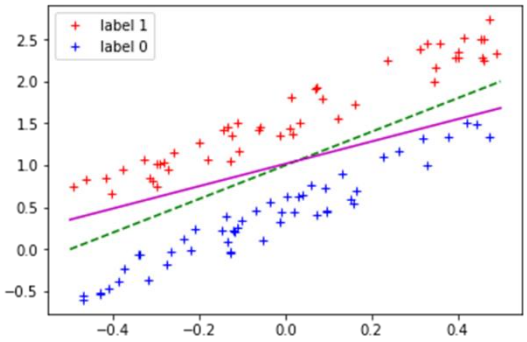
Exercise:
In groups of 2 discuss what happens, if the Perceptron Learning Algorithm is applied to a set that is not linearly separable.

The following theorem can be proved:

Perceptron Convergence Theorem:
The Perceptron Learning Algorithm converges in a finite number of steps to a weights vector and bias that separates the two classes – provided that the two classes are linearly separable.

Nevertheless, the following two points should be noted:

- The solutions obtained for linearly separable inputs is not unique and not optimal. The optimal solution (with the widest separating corridor) is known as the linear support vector machine (SVM).
  Thus, in the example shown to the right the solid magenta-coloured line represents one solution of the Perceptron Learning Algorithm. It is far from being the optimal (dotted green) solution obtained from an SVM.

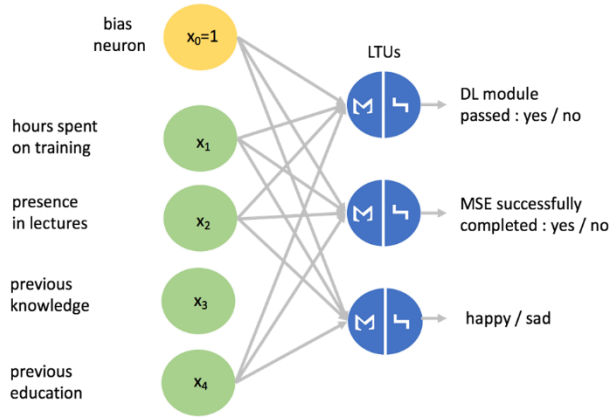


- The Perceptron may be a reasonable model also for problems that are not linearly separable. However, the application of the Perceptron Learning Algorithm does not converge in this case.

2.3 Artificial Neural Networks

2.3.1 A first Neural Network: Single Layer LTUs

Combining many Perceptron units will build a so-called neural network. Figure 35 shows an example of a single layer network with just one output layer, represented by the blue nodes. The green nodes are the input neurons and each of them is connected to each of the output neurons forming a so-called *fully connected layer*.
From an application point of view, the network in Figure 35 classifies input instances simultaneously into m different binary classes i.e., is a multi-output classifier that can perform multiple tasks. The outputs may be related to each other or not.



Some of the outputs may be related to each other, others not.

Student	Training	Presence	Prev Knowledge	Prev Education	Module passed	MSE successfully completed	Happy / sad
a	90	100%	medium	ec.	1	1	1
b	60	85%	high	c.s.	1	1	1
c	70	100%	medium	c.s.	1	1	0
d	40	30%	high	c.s.	0	1	1
e	10	90%	low	sys. eng.	1	1	1
f	85	50%	high	c.s.	1	1	1
g	30	20%	low	sys. eng.	0	0	1
h	70	30%	high	el.eng.	1	0	0
…	…	…	…	…			

Figure 35: *Example of an LTU analysing the relation of a student's effort in a module with various criteria characterizing the results obtained.*

A further example with higher practical relevance (and which we will study extensively) is shown in Figure 36. Based on the MNIST-dataset[6] the classification of handwritten digits is performed with a single layer LTU. The input vector consists of the grayscale values of the 28 x 28-pixel sized image patch representing the given digit. The output will be the numerical value of the digit represented by the image patch. In contrary to the example given in Figure 35, here the input data is classified into m different *exclusive* classes. Therefore, the output values ('labels') in the dataset would be prepared as so-called *one-hot vectors* with only one non-zero entry.

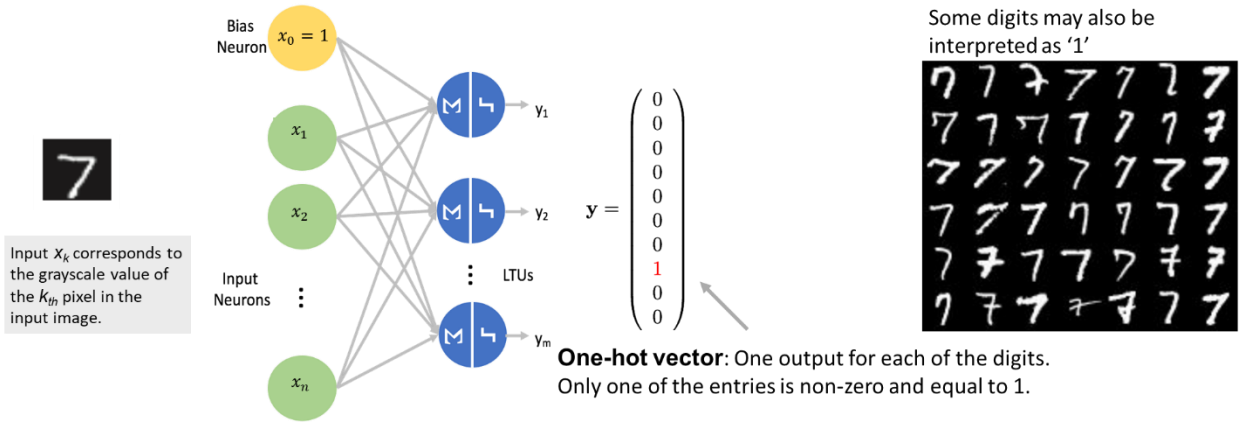


Figure 36: *Example of an LTU classifying instances simultaneously to m different exclusive classes.*

Exercise:
In groups of 2, discuss the following:

- What is the size of the input vector in Figure 36?
- Arrange a set of classes (you may chose digits) in 2D- (or if you are good in sketching) 3D-space. Consider the representational capacity of a single layer LTU and construct

cases, where the classification would or would not work.

- How could single layer LTUs be extended to solve the non-working cases you constructed.

2.3.1.1 The XOR Problem

The representational capacity of the single Perceptron is given by a hyperplane separating the feature space in two parts and therefore allowing to classify linearly separable binary set. A single layer LTU, which consists of m output neurons, represents an extension to m hyperplanes and therefore allows to solve classification problems of m classes that are mutually linearly separable.
One well-known example that cannot be solved with the single layer LTUs is the XOR function. Mathematically it is represented as

$$h(x_1, x_2) = \begin{cases} 0 & (x_1 = x_2) \\ 1 & (x_1 \neq x_2) \end{cases}$$

where x_1 and x_2 can take the values 0 and 1.
Graphically the XOR problem can be represented as a binary classification task with two sets as shown in the following Figure 37 with the green squares and yellow triangles. It is clear, that these two pointsets cannot be separated by a single layer LTU which immediately represents a considerable limitation for this approach. In fact, the discovery of this XOR problem was one of the reasons for the end of the first AI hype in the 1960s[32].

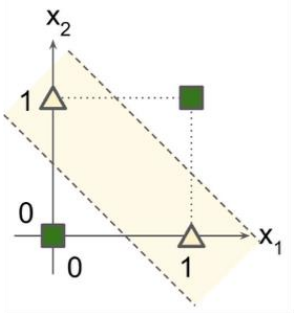


Figure 37: *The XOR problem represented as a binary classification problem in 2D.*

2.3.2 Multi-Layer Perceptron

A Multi-Layer Perceptron (MLP) is simply a stack of single layer LTUs (Figure 38). It is composed of an input layer, one or more hidden layers (layers of LTUs) and a final output layer. Input layer and hidden layers include a bias neuron and are fully connected to the next layer. The activation function (so far, we only discussed the Heaviside step function) may be replaced by smooth functions as we will see in the next chapters.

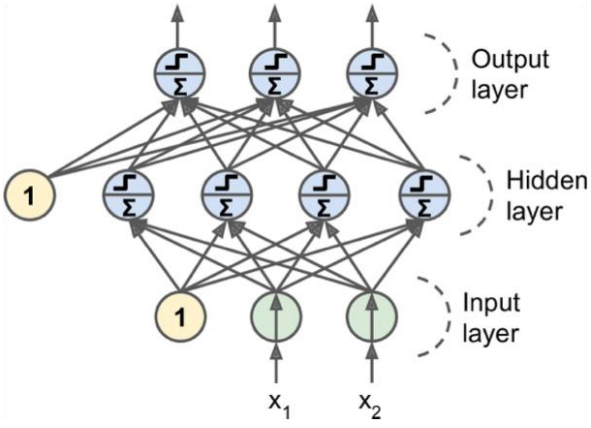


Figure 38: *An MLP is composed an input layer, one or several hidden layers and an output layer.*

[32] c.f. [1], part 1 Introduction, figure 1.7, page 13.

The superior representational capacity of the MLP over the single-layer LTU becomes immediately apparent when revisiting the XOR problem, because an MLP with a hidden layer can be used to give a simple solution (Figure 39). We will see later that an MLP with at least one hidden layer can represent essentially any well-behaved function mapping the input space $\mathbb{R}^n$ to the output space $\mathbb{R}^M$ with arbitrary accuracy.

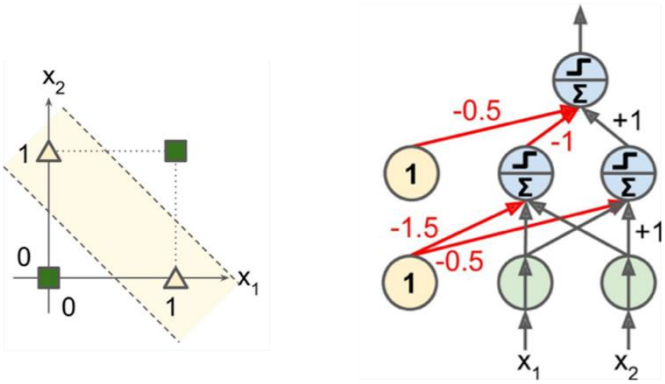


Figure 39: *Solution of the XOR-problem based on a MLP with one hidden layer.*

3 Learning and Optimisation

In this chapter we will formulate the machine learning task (Figure 13) in formal mathematical language[33]. The exemplary task we want to solve is to automatically predict the correct digit from a handwritten image (Figure 40). We assume a hypothetical mapping $f(\mathbf{x})$ fulfilling this task, which however we do not have any knowledge of. Therefore, we construct a model $h_\theta(\mathbf{x})$ that should approximate this mapping $f(\mathbf{x})$. The subscript θ of the model function $h_\theta(\mathbf{x})$ represents the parameters that it depends on, and which will be optimized during the learning step, and which is represented in Figure 41. Based on a set of training data $(\mathbf{x}, y)$ ($\mathbf{x}$ representing the input data i.e., the images and y the corresponding labels i.e., the digits) we will iteratively correct i.e., optimize the parameters θ of the model function $h_\theta(\mathbf{x})$ to minimize the discrepancy between the true labels y and the model prediction $\hat{y} = h_\theta(\mathbf{x})$. This discrepancy is quantified with a so-called Cost-function.

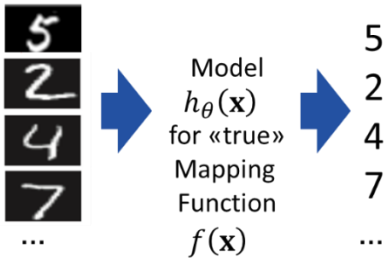


Figure 40: *We want to solve a classification task i.e., find the correct digits (right) to the input images (left).*

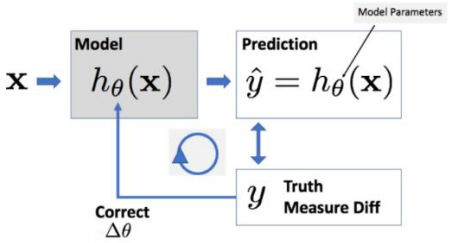


Figure 41: *Formalization of the learning procedure required to optimize our model $\boldsymbol{h_\theta(\mathbf{x})}$ (details c.f. text).*

In chapter 2.2.2 we already studied an example of such a learning procedure. There, the model was given by the Perceptron as represented in Figure 31, the parameters were the weight vector $\mathbf{w}$ and bias b (i.e., $\theta = (\mathbf{w}, b)$), and the optimization step was given by the perceptron learning algorithm. The cost function was simply based on the number of differences $(\hat{y}^{(i)} - y^{(i)})$ between the true labels $y^{(i)}$ and the prediction of the Perceptron $\hat{y}^{(i)} = h_\theta(\mathbf{x}^{(i)})$. In this chapter we will extend this concept to the so-called "generalized Perceptron" and introduce the following new concepts:

1. A smooth activation function i.e., the sigmoid function will be used instead of the Heaviside step function.
2. The Perceptron learning algorithm will be replaced by the Gradient Descent Algorithm.
3. Two new cost functions will be presented, the Mean Squared Error and the Cross Entropy.

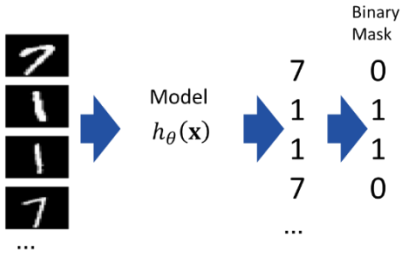


Figure 42: *To simplify the problem we start with a binary classification task based on a set of two digits only.*

[33] Before reading further it might be worth having a look at the document summarizing the conventions used for the mathematical notations [6].

We will nevertheless not tackle right away the classification task described in Figure 40 with 10 different classes but start with the simpler binary classification problem based on two digits only (Figure 42). As dataset we will use the MNIST digits which we will quickly introduce in the following section.

3.1 MNIST Dataset

MNIST[6] is a freely available dataset of handwritten digits (Figure 4), which is widely used for testing and illustration of classification tasks. Its relevance is twofold. First it was one of the first sets comprising a comparatively large number of samples (70'000, c.f. Figure 21) and second, the image patches are of size 28 x 28-pixel "only" thus keeping the requirements on the computer resources (CPU power, memory) reasonable. Two sets are available:

- Original MNIST
  70'000 images
  28 x 28-pixel image size, grey value range [0, 255]
  Available for download via reference given here [6] or various Python frameworks (scikit-learn, keras, tensor flow)

- Lightweight MNIST ('MNIST light')
  1'797 images
  8x8-pixel image size, grey value range [0, 16]
  Installed as part of scikit-learn library

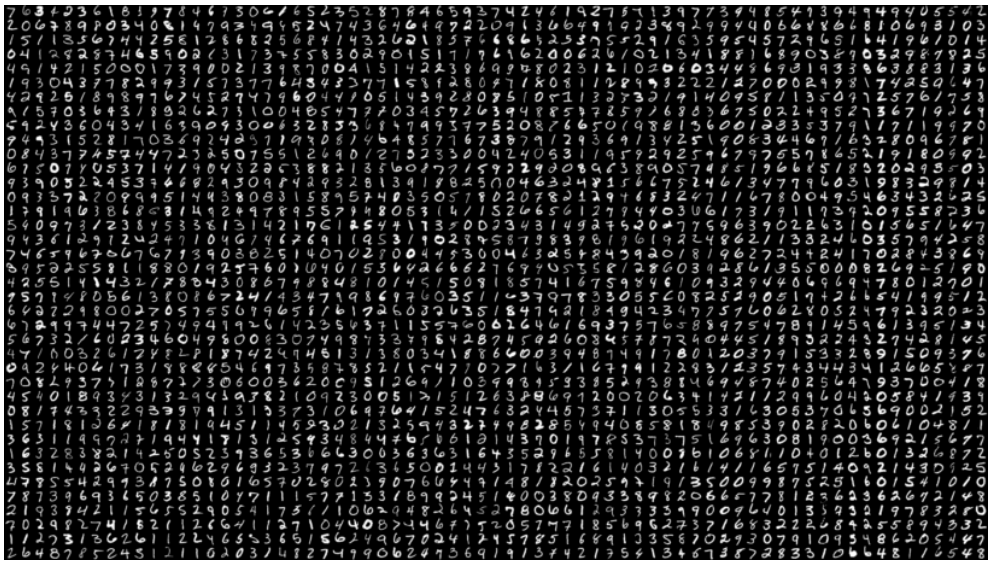


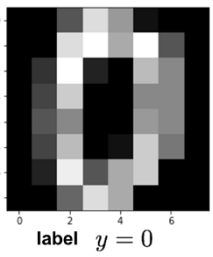
Figure 43: *A subset of the 70'000 MNIST patches containing handwritten (American English[34]) digits.*

Exercise:
Using the provided iPython notebooks perform the following tasks:

- Start using the iPython notebook `load_mnist_light-stud.ipynb`. You can use it to download all 1797 images and corresponding labels.

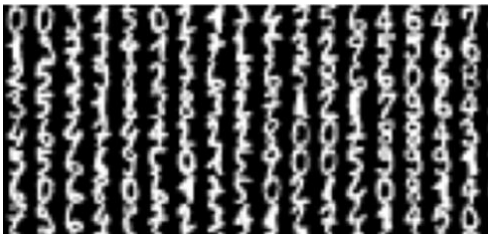
- Extend the notebook to print the content of the first image (as 8 x 8 matrix) and plot the corresponding image:

```
[[ 0.  0.  5. 13.  9.  1.  0.  0.]
 [ 0.  0. 13. 15. 10. 15.  5.  0.]
 [ 0.  3. 15.  2.  0. 11.  8.  0.]
 [ 0.  4. 12.  0.  0.  8.  8.  0.]
 [ 0.  5.  8.  0.  0.  9.  8.  0.]
 [ 0.  4. 11.  0.  1. 12.  7.  0.]
 [ 0.  2. 14.  5. 10. 12.  0.  0.]
 [ 0.  0.  6. 13. 10.  0.  0.  0.]]
```



[34] Note that a 1 is written using a single vertical bar only.

- Extend the notebook to print n x n tiles of digits in one single figure.



- Now continue with the iPython notebook `load_mnist-stud.ipynb` and repeat the above steps for the MNIST dataset.

- Finally test the application of the following strategies for selecting two digits.

```
#select the two digits for your training and test set
digit_1 = 1
digit_2 = 7

x_sel_1 = x[y == digit_1,:]
x_sel_2 = x[y == digit_2,:]

x_sel = np.append(x_sel_1, x_sel_2, 0)
y_sel = np.append(np.zeros((x_sel_1.shape[0],1)),
                  np.ones((x_sel_2.shape[0],1)), 0)
```

Selecting two digits

Selecting subsets of images corresponding to digits.

Concatenating subsets and creating labels

Note:
When programming a new algorithm, a good strategy is to start with the MNIST-light dataset because the processing will be considerably faster than the full MNIST data. Once the algorithm is stable you can switch to MNIST and do the necessary refinements.

3.2 Data Preparation

Before we start with the actual learning procedure, we must prepare the data accordingly. This will be discussed in this chapter.

3.2.1 Split of data in Training and Test Set

As already discussed in Figure 17 we must test our model after the learning procedure. This must be done on an independent data set, because we want to make a prediction about the ability of our model to generalize to unknown data. Therefore, we split the data into two subsets[35]:

- Training Set: Used for learning the task.
- Test Set: Used for testing how well the learned model performs.

The typical split ratios used depend on the available data size:

	Small Datasets	Large Datasets
Training	70-80%	99%
Testing	20-30%	1%

The following points should be considered:

- Training and test set should share the same characteristics:
  Data acquisition may change over time. E.g., the first and second halves of the image set may have been acquired under different lighting conditions (day <-> night). To ensure identical characteristics the data should be shuffled randomly before splitting.

- Training and test set should be kept strictly separated:
  During learning, no information contained in the test set may be used to adjust parameters of the model. As mentioned above, this is the only way to make an independent statement about the performance of the model on previously unseen data.

[35] Splitting the dataset into three parts (with an additional validation set) will be treated later.

3.2.1.1 Example: MNIST Dataset split using scikit-learn

Several libraries have built-in functionality to do the split in training and test set. In Figure 44 below, this is performed with scikit-learn for the MNIST dataset. The code snippet is from practical work exercise `MNIST_binary_classifier_stud.ipynb` (Week 2)[36].

```
from sklearn import model_selection as ms

#define train and test split
x_train, x_test, y_train, y_test = ms.train_test_split(x_sel, y_sel,
                                    test_size=0.20, random_state=1)

print(x_train.shape, x_test.shape, y_train.shape, y_test.shape)
```

```
(12136, 784) (3034, 784) (12136, 1) (3034, 1)
```

Figure 44: *Example for the split in training and test data based on scikit-learn for MNIST data.*

3.2.2 Data Normalisation – Scaling and Centring

Data normalization is the process of scaling and centring the data:

- Scaling means to bring the data to the same scale.
  It improves the numerical stability, the convergence speed and accuracy of the learning algorithms. We will illustrate the reason for that that below.

- Centring means to balance the data around zero.
  This also improves the robustness of the learning algorithms and is particularly important for stabilising the convergence in deep networks. Probably it also improves the convergence speed and accuracy of the learning algorithms.

Illustration of Scaling of input data:

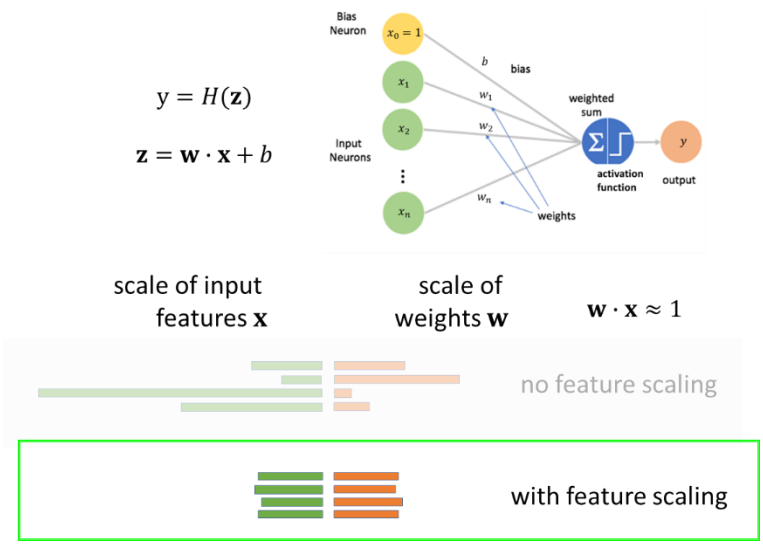


Figure 45: *Feature scaling will lead to weights having also similar scales which will be beneficial for the stability of the learning process.*

The numerical values of the features contained in the input[37] data $\mathbf{x}$ may be on different scales, ranging over different orders of magnitude. E.g., the input data for a classification problem on large cities

[36] Depending on the framework (here scikit-learn) you may need to do some suitable "data wrangling" (such as the transposition of `numpy` arrays).
[37] This may also apply to the output data y e.g., in case we have a regression task.

may contain the following features:

1. Size in km measuring the perimeter (~ 100)
2. Population (~ 1'000'000)
3. Yearly gross product in $ (~ 10'000'000'000)

If we leave the input data on such different scales the corresponding weights in the Perceptron will also have very different scales. The reason for that is illustrated in Figure 45. The activation function $H(\mathbf{z})$[38] has a dimensionless argument $\mathbf{z}$ and therefore, the weight component w_k will have the inverse unit and inverse scale of the corresponding input feature x_k such that the value of the product will be of unit magnitude i.e., $\mathbf{w} \cdot \mathbf{x} = \sum w_k \cdot x_k \approx 1$. Thus, if we scale the input features to have the same magnitude this will be also true for the weights (Figure 45). This is the preferred case as the learning algorithms can focus on learning the importance of features and not their scale.
Different scaling and normalization schemes are currently applied, and we will start with a scaling-only scheme.

3.2.2.1 Min-Max Rescaling

[39]The input values $\mathbf{x}$ consist of a set of training vectors $\mathbf{x}^{(i)}$ with index $i = 1..m$. Each vector consists of components $x_k^{(i)}$ with index $k = 1..n_x$. For the example given above with the large city classification each vector $\mathbf{x}^{(i)}$ (each city) consists of three components ($n_x = 3$) corresponding to the size, population, and gross product. To prepare the scaling, for each component k the minimum and maximum $(\min_k, \max_k)$ over all samples are determined:

$$\min_k = \min_{1 \le i \le m} \{x_k^{(i)}\}\ ,\ \max_k = \max_{1 \le i \le m} \{x_k^{(i)}\}$$

Thus, for all cities the maximum and minimum size, population, and gross product are determined. It should be noted that the minimum and maximum is taken over the *training data* set only to keep the strict separation of the learning process from the test set.
Then the following scaling scheme is applied, which will map all values, independent from their input scales to the final range $[0,1]$:

$$x_k'^{(i)} = \frac{x_k^{(i)} - \min_k}{\max_k - \min_k}$$

```
img = x[0,:].reshape((8,8))
print(img)
```

```
[[ 0.  0.  5. 13.  9.  1.  0.  0.]
 [ 0.  0. 13. 15. 10. 15.  5.  0.]
 [ 0.  3. 15.  2.  0. 11.  8.  0.]
 [ 0.  4. 12.  0.  0.  8.  8.  0.]
 [ 0.  5.  8.  0.  0.  9.  8.  0.]
 [ 0.  4. 11.  0.  1. 12.  7.  0.]
 [ 0.  2. 14.  5. 10. 12.  0.  0.]
 [ 0.  0.  6. 13. 10.  0.  0.  0.]]
```

Computes the min and max over all the pixels.

```
x /= np.max(x)
print(img)
```

```
[[0.     0.     0.3125 0.8125 0.5625 0.0625 0.     0.    ]
 [0.     0.     0.8125 0.9375 0.625  0.9375 0.3125 0.    ]
 [0.     0.1875 0.9375 0.125  0.     0.6875 0.5    0.    ]
 [0.     0.25   0.75   0.     0.     0.5    0.5    0.    ]
 [0.     0.3125 0.5    0.     0.     0.5625 0.5    0.    ]
 [0.     0.25   0.6875 0.     0.0625 0.75   0.4375 0.    ]
 [0.     0.125  0.875  0.3125 0.625  0.75   0.     0.    ]
 [0.     0.     0.375  0.8125 0.625  0.     0.     0.    ]]
```

Figure 46: *Min-Max Rescaling applied to (a single) MNIST light image.*

[38] In Figure 45 the Heaviside function is used but the same would be true for other types of activation functions (c.f. chapter 3.8).
[39] For the notation convention refer to [6].

It is important to note that for images the minimum and maximum values are determined over *all* features i.e., pixels of the images. Recall e.g., that for the MNIST dataset each vector $\mathbf{x}^{(i)}$ represents an image with $n_x = 28 \times 28 = 784$ components corresponding to the pixels of the image patches. Because all pixels of the image are *already at the same scale* there is no need to scale them independently. This is illustrated in the Figure 46, where the scaling is applied to one single MNIST light image. Note that the value of $\min_k = 0$ is skipped in the formula and only the division by $\max_k$ is applied.
Apart from Min-Max Rescaling where no centring is applied the following two normalization schemes (i.e., including centring) are applied.

3.2.2.2 Min-Max Normalisation

Min-Max Normalisation is very similar to Min-Max Rescaling apart from the fact that now centring is applied via the following scheme mapping all values, independent from their input scales to the final range$[-1,1]$:

$$x_k'^{(i)} = 2 \cdot \frac{x_k^{(i)} - \min_k}{\max_k - \min_k} - 1$$

3.2.2.3 z-Normalisation

This normalisation scheme is inspired by statistics as it leads to a distribution of each input component having zero mean and unit-variance. Therefore, in a first step the mean and variance for each input component is determined (on the *training data* only):

$$\mu_k = \frac{1}{m}\sum_{i=1}^{m} x_k^{(i)} \qquad \sigma_k^2 = \frac{1}{m}\sum_{i=1}^{m}\left(x_k^{(i)} - \mu_k\right)^2$$

Then the normalization scheme is applied, which leads to zero mean and unit variance for each input component:

$$x_k'^{(i)} = \frac{x_k^{(i)} - \mu_k}{\sigma_k}$$

As already discussed for Min-Max Rescaling, in case that the input data are images the mean μ_k and variance σ_k^2 is calculated over all pixel i.e., all input features k.

3.3 Generalised Perceptron

As already noted in the introduction to this chapter 3 we will use the example task of image classification, i.e., recognizing a particular digit based on the MNSIT dataset, to formulate the learning approach in a general way. Using this "toy model" allows us to introduce and illustrate general learning concepts more concretely by applying them directly to a practical example. The two main concepts that we will present in the following will be the gradient descent optimization scheme and two new cost functions (MSE and CE). However, in a first step we must extend the Rosenblatt Perceptron (Figure 31) to the so-called generalised Perceptron which means that we have to replace the original Heaviside activation function by a smooth version.

3.3.1 Sigmoid Activation Function

The Heaviside $H(z)$ step function, while being conceptionally simple, has the major inconvenience of not being suitable for optimization schemes like Gradient Descent (GD). GD makes use of the derivate of the activation function, which is equal to zero everywhere for $H(z)$ except for $z = 0$ where it is equal to a Dirac impulse. A smooth version of the Heaviside function $H(z)$ is given by the so-called sigmoid function $\sigma(z)$ shown in Figure 47[40]. As is true for $H(z)$ the function $\sigma(z)$ rises from 0 to 1 but in a smooth manner and therefore – even infinitely often – differentiable[41].

[40] Later, in chapter 3.8 we will present all activation functions commonly used for ML models.
[41] Properties of the sigmoid function will be studied in the practical work exercises.

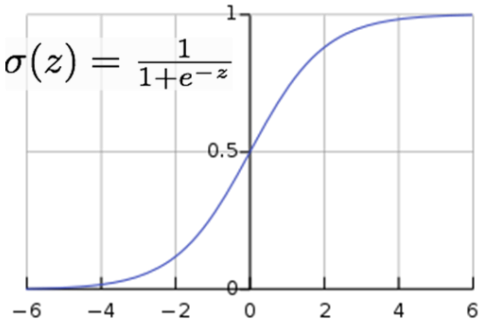


Figure 47: *The sigmoid activation function is a smooth generalisation of the Heaviside step function.*

Now, based on the sigmoid function we can generalise the Perceptron to the version shown in Figure 48 below, i.e., with one single output neuron only. The output is no longer a binary 'yes' (1) or 'no' (0), but a numeric value in the interval $[0,1]$. Class labels can be assigned according to the rule:

$$\text{yes:}\quad \hat{y} = h_\theta(\mathbf{x}) \geq 0.5 \qquad \text{no:}\quad \hat{y} = h_\theta(\mathbf{x}) < 0.5$$

$$\hat{y} = h_\theta(\mathbf{x}) = \sigma(\mathbf{w} \cdot \mathbf{x} + b)$$

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

$$\theta = (\mathbf{w}, b)$$

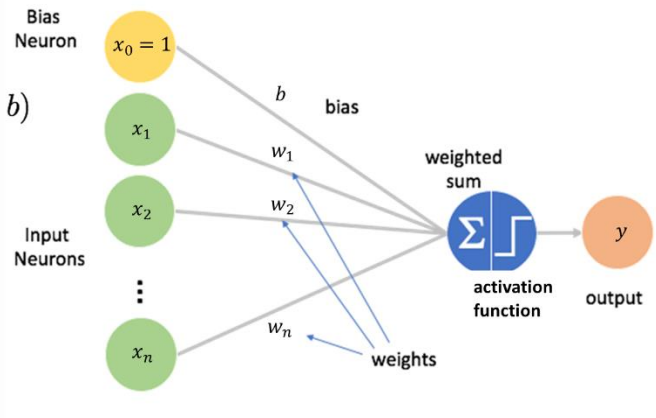


Figure 48: *The generalised perceptron using the sigmoid activation function.*

As for the Rosenblatt Perceptron we require to optimize the parameters $\boldsymbol{\theta} = (\mathbf{w}, b)$ i.e., the weight vector $\mathbf{w}$ and bias b. This will be done by the Gradient Descent scheme which we will introduce now.

3.4 Gradient Descent

Gradient Descent is a general scheme to optimize a function with respect to some parameters. Here we want to apply this scheme to optimize the model parameters $\boldsymbol{\theta} = (\mathbf{w}, b)$ such that the predictions $\hat{y}^{(i)} = h_\theta(\mathbf{x}^{(i)})$ of our model are "close" to the true outcomes $y^{(i)}$. The requires some notion of distance between $\hat{y}^{(i)}$ and $y^{(i)}$ which is provided by the concept of so-called cost functions. We will introduce two cost functions, namely the Mean Squared Error (MSE) and Cross Entropy (CE). Since the first is quite intuitive, we will start with it.

3.4.1 Mean Squared Error Cost Function

As the name suggests, the Mean Squared Error (MSE) cost function just determines the average squared distance between the true outcomes $y^{(i)}$ and the predictions $\hat{y}^{(i)} = h_\theta(\mathbf{x}^{(i)})$:

$$J_{MSE}(\boldsymbol{\theta}) = \frac{1}{2m}\sum_{i=1}^{m}\left(\hat{y}^{(i)} - y^{(i)}\right)^2 = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta\left(\mathbf{x}^{(i)}\right) - y^{(i)}\right)^2$$

The sum extends over the m input vectors $\mathbf{x}^{(i)}$ and outcomes $y^{(i)}$ and the pre-factor $1/m$ takes care of the average. The additional factor $1/2$ is just for "cosmetic" reasons because it cancels out with a factor of two coming from the derivative applied later.
The learning procedure (Figure 41) can be simply formulated by determining of the minimum of the

cost function $J_{MSE}(\theta)$ with respect to the parameters θ. However, in general no closed form solution can be given, and an iterative optimisation scheme must be applied. This is the GD optimisation which will now formulate first in its general form and then specifically for the MSE cost function.

3.4.2 General formulation of Gradient Descent

The minimisation of any cost function $J(\boldsymbol{\theta})$ based on the GD schemes works as follows:

1. Start with some initial value $\boldsymbol{\theta}_0$ for the parameter vector $\boldsymbol{\theta}$.
   (e.g., random values or all 0)
2. Iteratively update the parameter vector $\boldsymbol{\theta}$ by
   a. Computing the gradient of the cost function at the current position $\boldsymbol{\theta}_t$: $\nabla_\theta J(\boldsymbol{\theta}_t)$
   b. Step in the negative gradient direction according to:
   $$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \cdot \nabla_\theta J(\boldsymbol{\theta}_t) \quad (\alpha \text{ is the learning rate})$$
3. Stop when the change in parameter vector update step ($\boldsymbol{\theta}_t \rightarrow \boldsymbol{\theta}_{t+1}$) is “small”

Figure 49 illustrates GD using a cost function $J(\boldsymbol{\theta})$ dependent on a two-dimensional parameter vector $\boldsymbol{\theta} = (\theta_0, \theta_1)$. Then $J(\boldsymbol{\theta})$ can be represented as a surface in 3D space. As is well known form multidimensional analysis, the derivative

$$-\nabla_\theta J(\boldsymbol{\theta}_t) = -\begin{bmatrix} \partial_{\theta_0} J(\theta_t) \\ \partial_{\theta_1} J(\theta_t) \end{bmatrix}$$

is a 2D vector pointing in direction of the strongest descent of the function $J(\boldsymbol{\theta}_t)$. Thus, when starting at the point indicated by the star, the GD algorithm will lead successively along the black trajectory to the indicated local minimum, which however is not necessarily the global minimum.

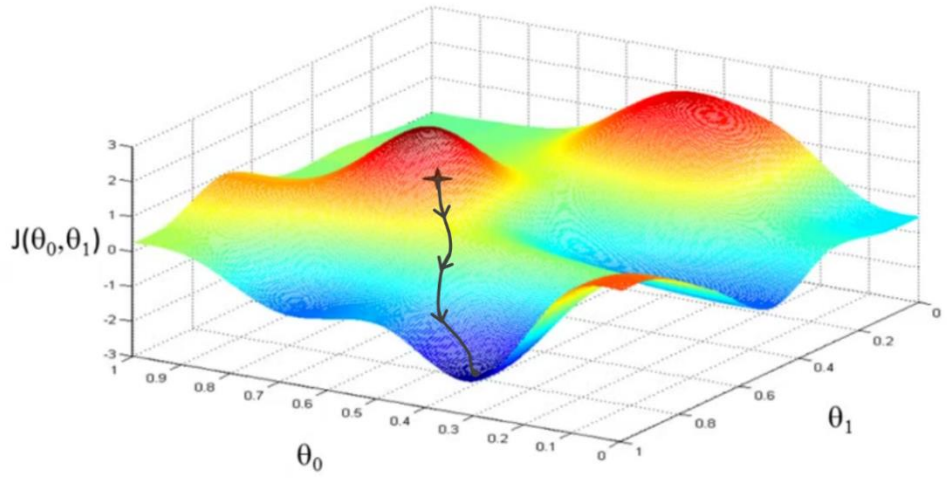


Figure 49: *Gradient descent always moves along the (locally) steepest descent and eventually reaches a local (but not necessarily global) minimum.*

While the idea of GD is conceptually simple, its use is far from being trivial and we will later discuss several improvements with respect to the standard GD update rule given above. Nevertheless, the following points should be noted here already:

- The GD learning principle does not depend on the type of model and can be applied if gradients can be computed (e.g., also works for deep neural networks).
- It works ‘locally’ (by using local function properties) – designed to find local but not necessarily global minimum.
- The position of the local minimum approached may depend very sensitively upon the starting position (Figure 50).

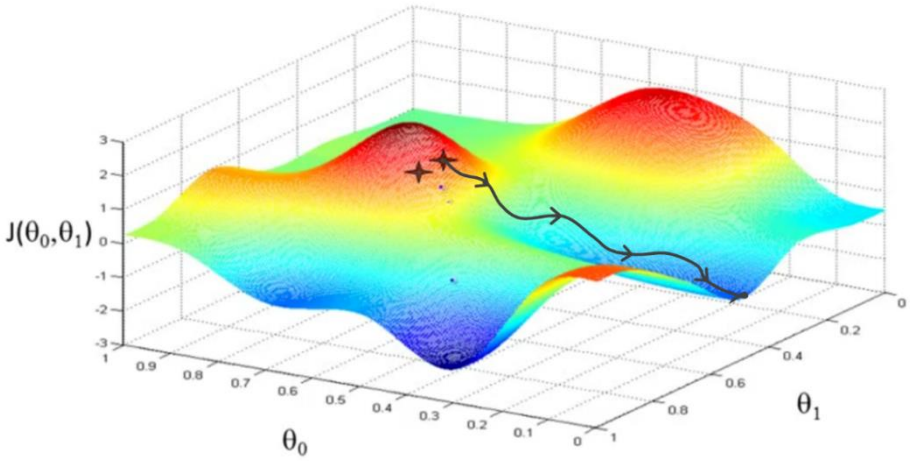


Figure 50: *In case the start position of the GD is only slightly changed, a completely different minimum position might be reached.*

- The learning scheme can get stuck in critical points where the gradient is zero. Without additional information, it cannot distinguish between local minima, local maxima, or saddle points.
- GD is iterative and iteratively approaches a critical point and may fluctuate around it.
- Thus, it is not guaranteed to converge e.g., if learning rate is chosen too large.

Having introduced the general GD optimisation scheme, we can now formulate the update rule specifically for the MSE cost function of the generalised Perceptron (Figure 48).

3.4.3 GD Update for MSE Cost of generalised Perceptron

For the GD update rule (chapter 3.4.2) we require the partial derivates of the MSE cost function

$$J_{MSE}(\boldsymbol{\theta}) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta\left(\mathbf{x}^{(i)}\right) - y^{(i)}\right)^2$$

with respect to the components θ_k of the parameter vector $\boldsymbol{\theta} = (\mathbf{w}, b)$ i.e.:

$$\frac{\partial}{\partial\theta_k} J_{MSE}(\boldsymbol{\theta})$$

Here, the parameter components θ_k can be either a weight component w_k or the bias b. We also recall that prediction of the model is given by $\hat{y}^{(i)} = h_\theta\left(\mathbf{x}^{(i)}\right) = \sigma(\mathbf{w}\cdot\mathbf{x}^{(i)} + b)$.
Because $J_{MSE}(\boldsymbol{\theta})$ is essentially a sum of identical terms, we can focus on one of these terms and start with the derivation of the following relation:

$$\frac{\partial}{\partial\theta_k}\left(h_\theta\left(\mathbf{x}^{(i)}\right) - y^{(i)}\right)^2 = 2\cdot\left(h_\theta\left(\mathbf{x}^{(i)}\right) - y^{(i)}\right)\cdot\frac{\partial}{\partial\theta_k}h_\theta\left(\mathbf{x}^{(i)}\right) =^{(1)}$$

$$= 2\cdot\left(h_\theta\left(\mathbf{x}^{(i)}\right) - y^{(i)}\right)\cdot\left(1 - h_\theta\left(\mathbf{x}^{(i)}\right)\right)\cdot h_\theta\left(\mathbf{x}^{(i)}\right)\cdot\frac{\partial}{\partial\theta_k}\left(\mathbf{w}\cdot\mathbf{x}^{(i)} + b\right)$$

In the step indicated by $=^{(1)}$ we used the following relation for the sigmoid function,

$$\sigma'(z) = \sigma(z)\cdot(1 - \sigma(z))$$

which can be proven by elementary calculus.
Thus, it basically only remains to determine the last term in the above relation being the derivative of:

$$\frac{\partial}{\partial \theta_k}(\mathbf{w} \cdot \mathbf{x}^{(i)} + b)$$

Here we must distinguish two cases depending on whether the parameter components θ_k is a weight component w_k or the bias b:

$$\frac{\partial}{\partial \theta_k}\left(\mathbf{w} \cdot \mathbf{x}^{(i)} + b\right) = \begin{cases} x_k^{(i)} & (\theta_k = w_k) \\ 1 & (\theta_k = b) \end{cases}$$

Putting everything together we obtain the following result (using the replacement $\hat{y}^{(i)} = h_\theta\left(\mathbf{x}^{(i)}\right)$):

$$\nabla_{\mathbf{w}} J_{MSE}(\mathbf{w}, b) = \frac{1}{m}\sum_{i=1}^{m} \hat{y}^{(i)} \cdot (1 - \hat{y}^{(i)}) \cdot (\hat{y}^{(i)} - y^{(i)}) \cdot \mathbf{x}^{(i)}$$

$$\nabla_b J_{MSE}(\mathbf{w}, b) = \frac{1}{m}\sum_{i=1}^{m} \hat{y}^{(i)} \cdot (1 - \hat{y}^{(i)}) \cdot (\hat{y}^{(i)} - y^{(i)})$$

Note that the upper equation is a vector equation, with on the left-hand-side the gradient with respect to the weight vector $\nabla_{\mathbf{w}}$ and on the right-hand-side a weighted sum of all input vectors $\mathbf{x}^{(i)}$.
We will try to illustrate this result by discussing the different terms:

1. The sum expresses the average over the training set:

   $$\frac{1}{m}\sum_{i=1}^{m}(..)$$

2. The Measure of uncertainty is given by:

   $$\hat{y}^{(i)} \cdot (1 - \hat{y}^{(i)})$$

   This term is small either if $\hat{y}^{(i)}$ is close to zero or close to 1 i.e., if the prediction is clear. This term is large if $\hat{y}^{(i)}$ is around 0.5 i.e., at high uncertainty.

3. The error signals are given by:

   $$\left(\hat{y}^{(i)} - y^{(i)}\right) \cdot \mathbf{x}^{(i)}$$
   $$\left(\hat{y}^{(i)} - y^{(i)}\right)$$

   The first term $\sim\mathbf{x}^{(i)}$ contributes more for samples with larger mismatch. This is like the Perceptron Learning Rule but note that the prediction $\hat{y}^{(i)}$ is different.

With these formulas for the gradient of $J_{MSE}(\boldsymbol{\theta})$ with respect to the parameter vector $\boldsymbol{\theta} = (\mathbf{w}, b)$ and application of the general GD update rule (c.f. chapter 3.4.2) it is now straight forward to apply GD to the MSE cost function. We will show some results for the MNIST database below[42].

3.4.3.1 Binary Classification for generalised Perceptron using GD and MSE Cost

We will perform binary classification (c.f. Figure 42) of digits 1 and 7[43], which are represented with a total of 7877 and 7293 in MNIST, respectively. We will apply the following settings:

- The generalised Perceptron (Figure 48), i.e., single layer LTU will be used for classification.
- The sigmoid activation function will be used.
- The MSE cost function will be used.
- The split ratio applied to training and test set sizes is 80% to 20%, giving absolute numbers of 12136 and 3024, respectively.
- Min-Max-Rescaling is applied to the input data (c.f. chapter 3.2.2.1).
- Weights and bias are initialized to zero.

[42] The corresponding iPython notebooks will be part of the practical work.
[43] We will see later that the final error of the prediction may vary considerably depending on the choice of digits.

- A learning rate $\alpha = 0.5$ is used.
- A total of 500 GD optimisation steps (so-called *epochs*) are applied.

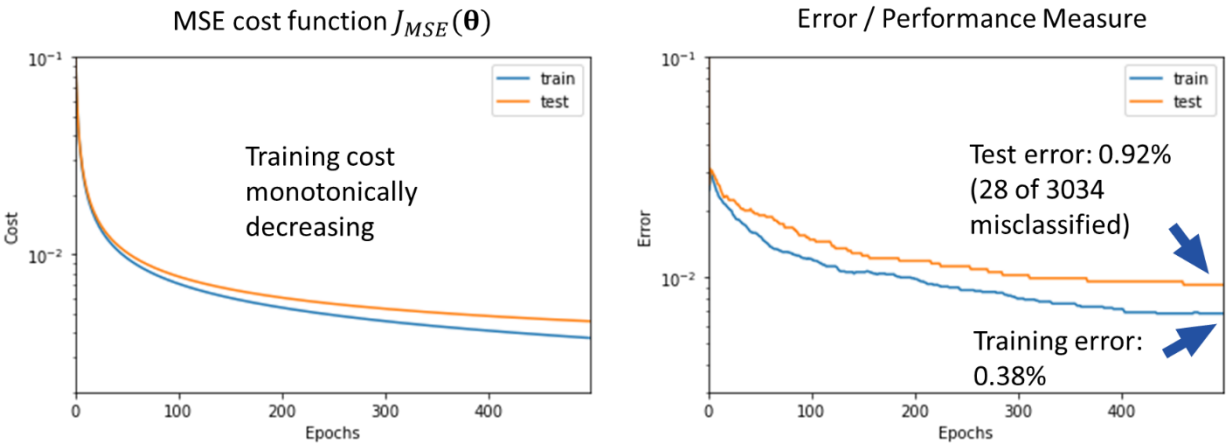


Figure 51: *Results of a binary classification of the digits 1 and 7 from the MNIST data set (details see text).*

The results are shown in Figure 51 above. On the left-hand-side the MSE cost functions for the training and the test data set are plotted as a function of the number of epochs. As expected, the cost function decreases monotonically because in each individual GD optimisation step a direction towards the current local minimum is searched.
The right-hand-side of Figure 51 shows the fraction of misclassified samples (error rate) as performance measure. Its definition is given in the Figure 52 below. We use the value of 0.5 as threshold to decide whether a digit was classified as 1 or 7 (c.f. Figure 48).

Fraction of misclassified samples as **performance measure**:

$$\text{error} = \frac{|\{\hat{y} \neq y\}|}{m_{\text{test}}}$$

with $\hat{y} = round(h_\theta(\mathbf{x}))$

(m_{test} = # test samples)

Figure 52: *The fraction of misclassified samples (with respect to the total training or testing set size) is used as performance measure.*

1 7 1 7 1 7 7
1 1 7 7 7 7 7
7 7 1 1 7 7 7
1 7 7 7 7 7 1

Figure 53: *The 28 misclassified digits from the testing set with their respective ground truth given to the right.*

Figure 53 finally shows the 28 misclassified digits from the test data set (corresponding to an error rate of 0.92%) with their respective ground truth given on the right-hand-side.

Despite these already encouraging results the question of further improvements immediately raises related to the following points/choices that were made:

1. What is the effect of different learning rates?
2. What is the effect of more training epochs?

3. What influence has the choice of the parameter initialisation?
4. Are there other, possibly better cost functions?

We will come to point 4. soon and introduce the so-called Cross Entropy Cost function, which is motivated by statistics. Before that, we want to study some issues related to the questions 1. and 2. above.

Choice of Learning rate

The Figure 54 below shows the effect of increasing learning rate (from left to right) on the cost and error rate. Taking our first trial ($\alpha = 0.5$) as benchmark we see that a lower learning rate of $\alpha = 0.1$ (left) leads to a slower decrease of both cost and error rate as indicated by the initial slope (dotted line) in the error rate plot[44]. On the contrary when increasing the learning rate to $\alpha = 2.0$ (right) an even faster decrease (than for $\alpha = 0.5$) can be observed. Thus, increasing the learning rate generally seems to be a good option[45]. However, this is only half of the story as is illustrated in Figure 55.

For a "good" choice of learning rate the GD algorithm will lead smoothly to the minimum (left). However, if the learning rate is chosen too large, the GD algorithm may oscillate around the minimum and never converge. On the other hand, if we choose the learning rate too small the convergence might be very slow, and GD may hardly or never converge either. Thus, the learning rate needs to be well tuned to the given problem and an optimal value for one problem may not work for another. Nevertheless, thanks to data normalisation the learning rate is largely independent of the scale of the input data.

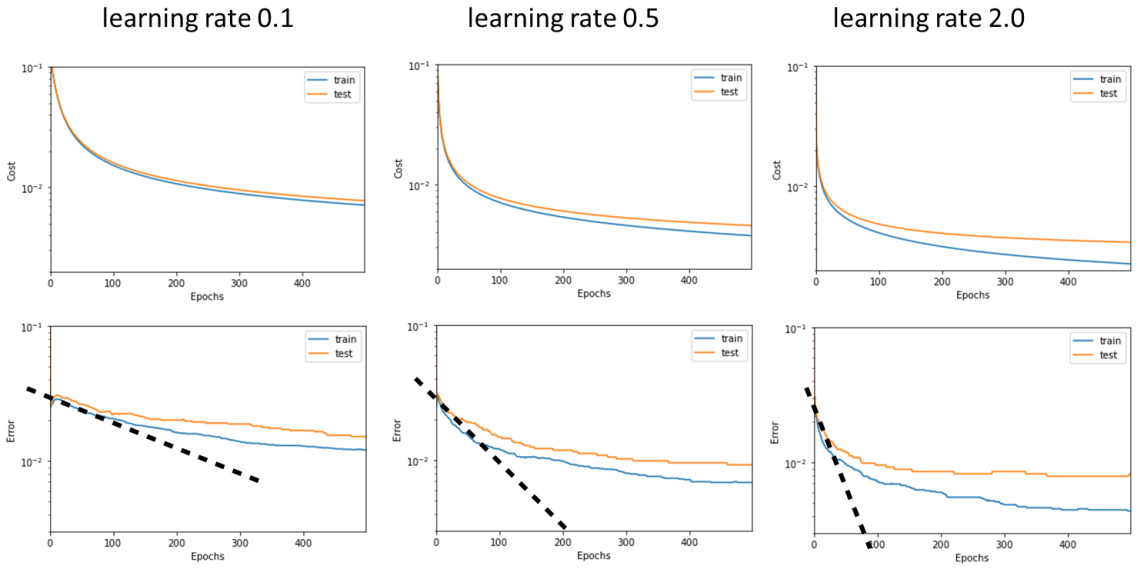


Figure 54: *Effect of different learning rates on cost (top) and error rate (bottom).*

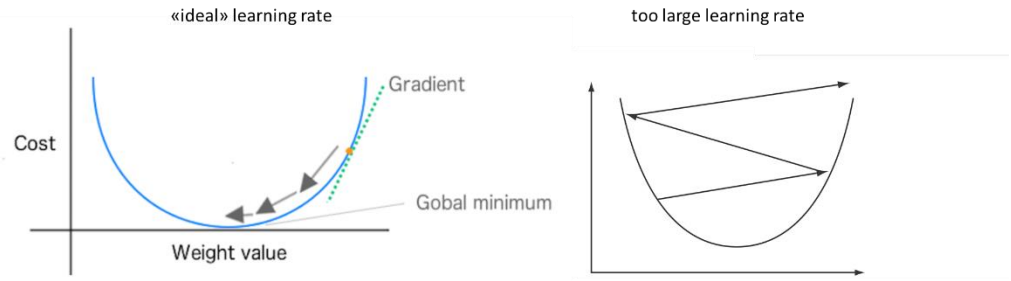


Figure 55: *Illustration of a «good» choice of learning rate (left), where GD moves smoothly to the minimum. A too large learning rate leads to GD oscillating around the minimum (right).*

[44] The error rate starts for epoch number 0 at value of around 0.5, which represents a so-called "dummy" predictor just choosing randomly between the two options for the digits 1 and 7.

[45] It should be noted that the learning rates used here for the illustration purpose are atypically large and usually considerably smaller values are used.

Increasing number of epochs

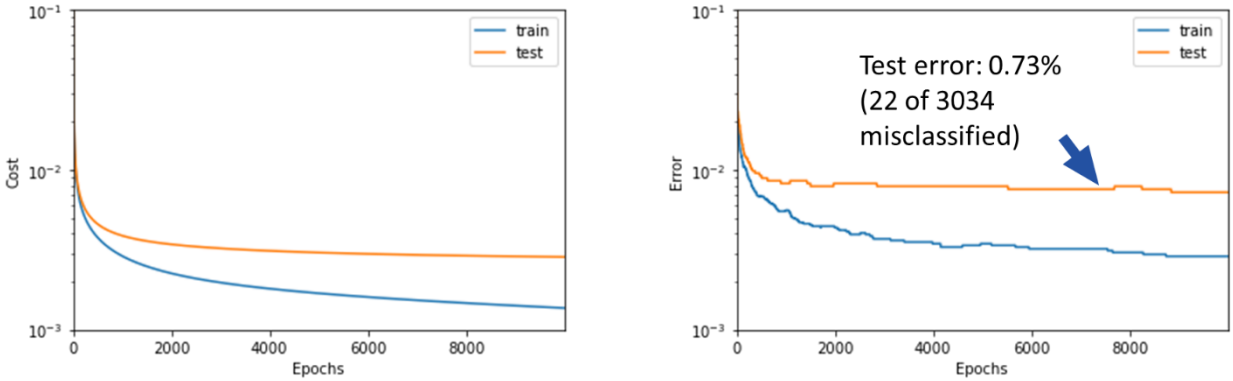


Figure 56: *Cost function and error rate for 10'000 epochs. After some 1000 epochs the test error rate will not decrease any further.*

In Figure 56 and Figure 57 the behaviour of our binary classifier for training some 10'000 epochs is shown. It can be noted that the testing error rate will not decrease any further after some 1000 epochs and will essentially represent the same set of misclassified images below. On the other hand, the training cost will continue to decrease i.e., the algorithm will further integrate information from the training data set trying to optimize the performance. This very important topic of so-called over-fitting will be treated in one of the following chapters.

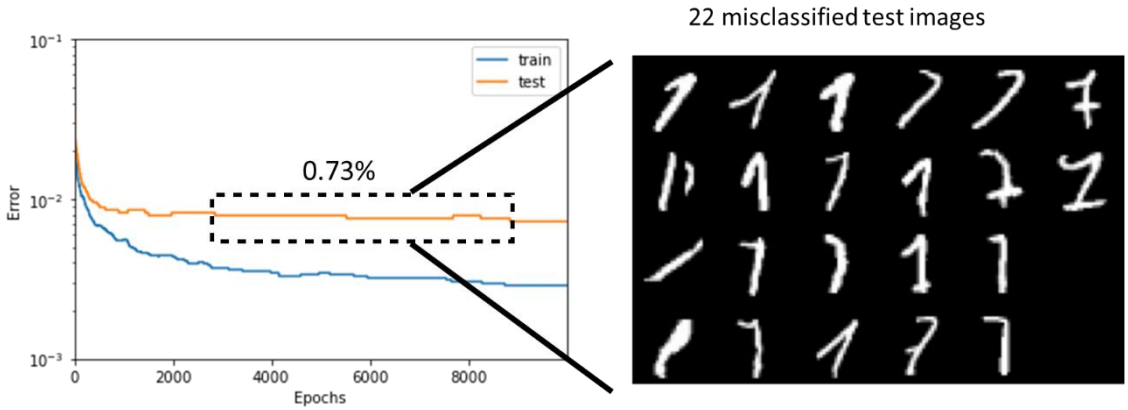


Figure 57: *Like Figure 56 above. The test error rate will not decrease after some 1000 epochs corresponding essential always to the 22 misclassified images shown.*

Thus, in summary we see that both the choice of the learning rate α and of the number of training epochs is far form being trivial and may influence considerably the training process (especially its duration) and the result. These questions are related to the general problem of hyperparameter tuning, which we will address in general later.

3.4.4 Cross Entropy Cost Function

While the MSE cost function is very intuitive it has several drawbacks. As discussed in chapter 3.4.3, point 2, the measure of uncertainty is given by the term:

$$\hat{y}^{(i)} \cdot (1 - \hat{y}^{(i)})$$

This expression is always positive and less or equal than $1/4$. When the model output $\hat{y}^{(i)}$ gets closer to either 0 or 1 (where the model is very confident in its prediction) this expression and the gradient can get very small. In result, the change in parameters can get very small and training can get stalled or stuck.
A cost function which is better suited to cope with this kind of problems is the Cross Entropy (CE) cost function. However, it is somewhat less intuitive to introduce than MSE. Therefore, we start from a well-known concept in statistics the so-called maximum likelihood estimator, which will lead us to CE cost.

3.4.4.1 Maximum Likelihood Estimator

Statistics provides a general framework to develop estimators and studies the bias and variance of these estimators, a topic which we will deal with later. One very general concept of obtaining a "good" estimator is the so-called maximum likelihood estimator (MLE) principle. We will introduce the MLE by formulating its principle for a general classification problem.
Consider a set of training data $(\mathbf{x}^{(i)}, y^{(i)})$ related to a classification task. We assume that there exists a true but unknown (distribution) function $p(y|\mathbf{x})$ that can predict the classes y given[46] the input $\mathbf{x}$. For a single input vector this reads $p(y^{(i)}|\mathbf{x}^{(i)})$. E.g., for our binary MNIST classification problem given an image $\mathbf{x}^{(i)}$ the function $p(y^{(i)}|\mathbf{x}^{(i)})$ would predict with certainty the correct class $y^{(i)}$. Now, we want to model $p(y|\mathbf{x})$ using an estimator $p_\theta(y|\mathbf{x})$ (or $p_\theta(y^{(i)}|\mathbf{x}^{(i)})$), with the dependency on the model parameter vector $\boldsymbol{\theta}$ denoted by the subscript. Our goal now is to optimize this estimator by tuning the parameter vector $\boldsymbol{\theta}$. The maximum likelihood estimator now states to select those parameter values $\boldsymbol{\theta}_{MLE}$ that make the observed data most probable (*most likely*) under that estimator. In mathematical terms this reads:

$$\boldsymbol{\theta}_{MLE} = \underset{\theta}{\operatorname{argmax}}\ p_\theta(y|\mathbf{x}) = \underset{\theta}{\operatorname{argmax}} \prod_{i=1..m} p_\theta(y^{(i)}|\mathbf{x}^{(i)})$$

Here the expression $p_\theta(y|\mathbf{x})$ denotes the probability for the entire training data. On the right-hand-side this expressed as the product over the (assumed) independent probabilities for each individual sample $p_\theta(y^{(i)}|\mathbf{x}^{(i)})$. As the product on the right-hand-side is prone to numerical underflow the logarithm[47] is taken leading to the final formulation of the maximum likelihood estimator:

$$\boldsymbol{\theta}_{MLE} = \underset{\theta}{\operatorname{argmax}} \sum_{i=1}^{m} \log\ p_\theta(y^{(i)}|\mathbf{x}^{(i)}) = \underset{\theta}{\operatorname{argmin}} \Bigg[\underbrace{-\sum_{i=1}^{m} \log\ p_\theta(y^{(i)}|\mathbf{x}^{(i)})}_{\text{CE Cost}} \Bigg]$$

The version on the right-hand-side is obtained by applying two changes, which mutually compensate: adding a minus sign and replacing the $\max$- by a $\min$-function. We conclude that the maximum likelihood estimator is based on the minimization of a suitable function given in square brackets on the right-hand-side which we will denote as Cross Entropy Cost.

$$J_{CE}(\theta) = -\frac{1}{m}\sum_{i=1}^{m} \log\ p_\theta(y^{(i)}|\mathbf{x}^{(i)})$$

Equation 1

Note the additional factor of $\frac{1}{m}$ which represents the division over all training samples and – because it is a constant factor – will not influence the optimization result. Thus, as for the MSE cost we calculate the *average* cross entropy cost over all training samples such that the scale of the CE cost will not depend on the number of samples.
While the formulation of the CE Cost above is generic, we will now evaluate its expression for the special case of our binary classification problem.

3.4.4.2 Cross Entropy Cost for binary Classification

To relate the above expression for the CE cost to our model function $h_\theta(\mathbf{x}^{(i)})$ for the binary classification task we must distinguish two possible outcomes depending on whether the desired class occurs $y^{(i)} = 1$ or does not $y^{(i)} = 0$ [48]:

$$p_\theta(y^{(i)} = 1, \mathbf{x}^{(i)}) = h_\theta(\mathbf{x}^{(i)})$$
$$p_\theta(y^{(i)} = 0, \mathbf{x}^{(i)}) = 1 - h_\theta(\mathbf{x}^{(i)})$$

[46] Thus, we formulate the maximum likelihood directly to estimate a conditional probability distribution, which is the most common case for supervised learning. Obviously, the maximum likelihood estimator can be formulated also for a non-conditional probability distribution.
[47] As the logarithm function is monotonically increasing this will not influence the result.
[48] For a Softmax-layer output, which we will study in chapter 3.6, we will be able to substitute $h_\theta(\mathbf{x}^{(i)})$ directly for the probability $p_\theta(y|\mathbf{x})$.

Therefore, the expression for the CE cost has two terms and can be written as follows:

$$J_{CE}(\boldsymbol{\theta}) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)} \cdot \log\, h_\theta\left(\mathbf{x}^{(i)}\right) + (1-y^{(i)}) \cdot \log\left(1 - h_\theta\left(\mathbf{x}^{(i)}\right)\right)\right]$$

Equation 2

3.4.5 GD Update for CE Cost of generalised Perceptron

This derivation will be very similar to the calculation of the GD update for MSE cost in chapter 3.4.3. Now our goal is to calculate the partial derivates of the CE cost function

$$J_{CE}(\boldsymbol{\theta}) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)} \cdot \log\, h_\theta\left(\mathbf{x}^{(i)}\right) + (1-y^{(i)}) \cdot \log\left(1 - h_\theta\left(\mathbf{x}^{(i)}\right)\right)\right]$$

with respect to the components θ_k of the parameter vector $\boldsymbol{\theta} = (\mathbf{w}, b)$ i.e.:

$$\frac{\partial}{\partial \theta_k} J_{CE}(\boldsymbol{\theta})$$

Again, we can focus on the derivate of a term under the sum (we include the minus sign):

$$-\frac{\partial}{\partial \theta_k}\left[y^{(i)} \cdot \log\, h_\theta\left(\mathbf{x}^{(i)}\right) + (1-y^{(i)}) \cdot \log\left(1 - h_\theta\left(\mathbf{x}^{(i)}\right)\right)\right] =$$

$$-\left[\frac{y^{(i)}}{h_\theta(\mathbf{x}^{(i)})} \cdot \frac{\partial}{\partial \theta_k} h_\theta\left(\mathbf{x}^{(i)}\right) + \frac{(1-y^{(i)})}{1-h_\theta(\mathbf{x}^{(i)})} \cdot \frac{\partial}{\partial \theta_k}\left[-h_\theta\left(\mathbf{x}^{(i)}\right)\right]\right] =$$

$$-\left[\frac{y^{(i)}}{h_\theta(\mathbf{x}^{(i)})} - \frac{(1-y^{(i)})}{1-h_\theta(\mathbf{x}^{(i)})}\right] \cdot \frac{\partial}{\partial \theta_k} h_\theta\left(\mathbf{x}^{(i)}\right) =^{(1)}$$

$$-\left[\frac{y^{(i)}}{h_\theta(\mathbf{x}^{(i)})} - \frac{(1-y^{(i)})}{1-h_\theta(\mathbf{x}^{(i)})}\right] \cdot \left(1 - h_\theta\left(\mathbf{x}^{(i)}\right)\right) \cdot h_\theta\left(\mathbf{x}^{(i)}\right) \cdot \frac{\partial}{\partial \theta_k}(\mathbf{w} \cdot \mathbf{x}^{(i)} + b) =$$

$$\left(h_\theta\left(\mathbf{x}^{(i)}\right) - y^{(i)}\right) \cdot \frac{\partial}{\partial \theta_k}(\mathbf{w} \cdot \mathbf{x}^{(i)} + b)$$

In the step indicated by $=^{(1)}$ we used again the following relation for the sigmoid function:

$$\sigma'(z) = \sigma(z) \cdot (1 - \sigma(z))$$

Now based on the result for the last term involving the derivative of $(\mathbf{w} \cdot \mathbf{x}^{(i)} + b)$, which we already obtained in chapter 3.4.3, we can finally conclude:

$$\nabla_{\mathbf{w}} J_{CE}(\mathbf{w}, b) = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)}) \cdot \mathbf{x}^{(i)}$$

$$\nabla_{b} J_{CE}(\mathbf{w}, b) = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})$$

Frequently, the two equations are summed up as follows:

$$\nabla_{\boldsymbol{\theta}} J_{CE}(\mathbf{w}, b) = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)}) \cdot \begin{bmatrix}\mathbf{x}^{(i)} \\ 1\end{bmatrix}$$

Here, the parameter vector $\boldsymbol{\theta} = (\mathbf{w}, b)$ can be either the weight vector $\mathbf{w}$ or the bias b.

Like the GD update rule for MSE in chapter 3.4.3, we can state the following:

1. The average over the training set is expressed by the sum:
$$\frac{1}{2m}\sum_{i=1}^{m}(..)$$
2. The error signal is given by the difference between the predicted probability $\hat{y}^{(i)} = h_\theta(\mathbf{x}^{(i)})$ and the label value $y^{(i)}$:
$$(\hat{y}^{(i)} - y^{(i)}) \cdot \begin{bmatrix}\mathbf{x}^{(i)} \\ 1\end{bmatrix}$$

We reformulate explicitly the GD update rule for CE cost for the Generalised Perceptron to compare it with the Learning Algorithm for the Rosenblatt Perceptron form chapter 2.2.2.2:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \cdot \frac{1}{m}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)}) \cdot \mathbf{x}^{(i)}$$
$$b \leftarrow b - \alpha \cdot \frac{1}{m}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})$$

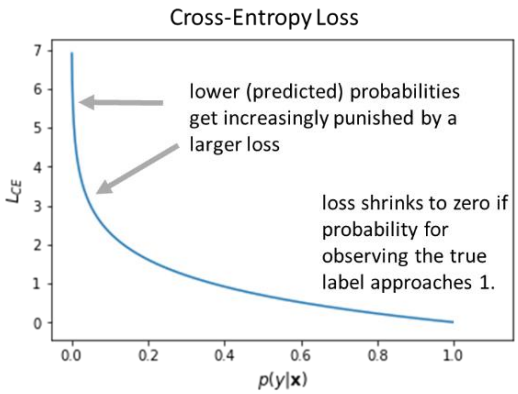
Exercise:

In groups of 2, discuss the similarities and differences between the GD update rule for CE and the Perceptron Learning Algorithm form chapter 2.2.2.2:

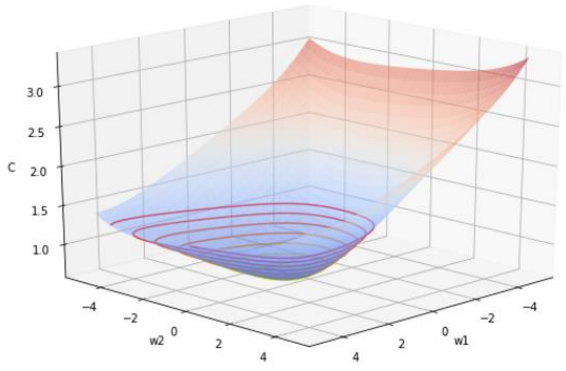
3.4.6 Differences between MSE and CE Cost GD Update

Having developed the two cost functions, we want to briefly discuss their major differences.

In the figure to the right the behaviour of the function $\log p_\theta(y|\mathbf{x})$ i.e., the CE cost is shown as a function of the probability $p_\theta(y|\mathbf{x})$. It is meant to give some intuition on why the CE Loss has superior performance as compared to the MSE Loss. The major point is that low probability values i.e., $p_\theta(y|\mathbf{x}) \rightarrow 0$ are increasingly punished for CE Loss, which tends to plus infinity, while for MSE Loss the values are limited to a maximum of 1.



Furthermore, the following point is important.
The CE cost function for the generalised perceptron is a convex function i.e., bowl-shaped as illustrated to the right[49].
Therefore, the above scheme is guaranteed to find the global minimum – if the learning rate is kept sufficiently small (to avoid oscillations around the minimum). For the MSE cost function such a behaviour cannot be guaranteed.



[49] It should be noted that for more complex architectures e.g., MLPs this is not the case.

In the Figure 58 the results on the binary classification problem involving the digits 1 and 7 on the MNIST dataset are compared for CE and MSE cost. These results confirm the general findings when comparing CE and MSE cost:

- Typically, CE cost shows faster learning
- Typically, CE cost has lower error

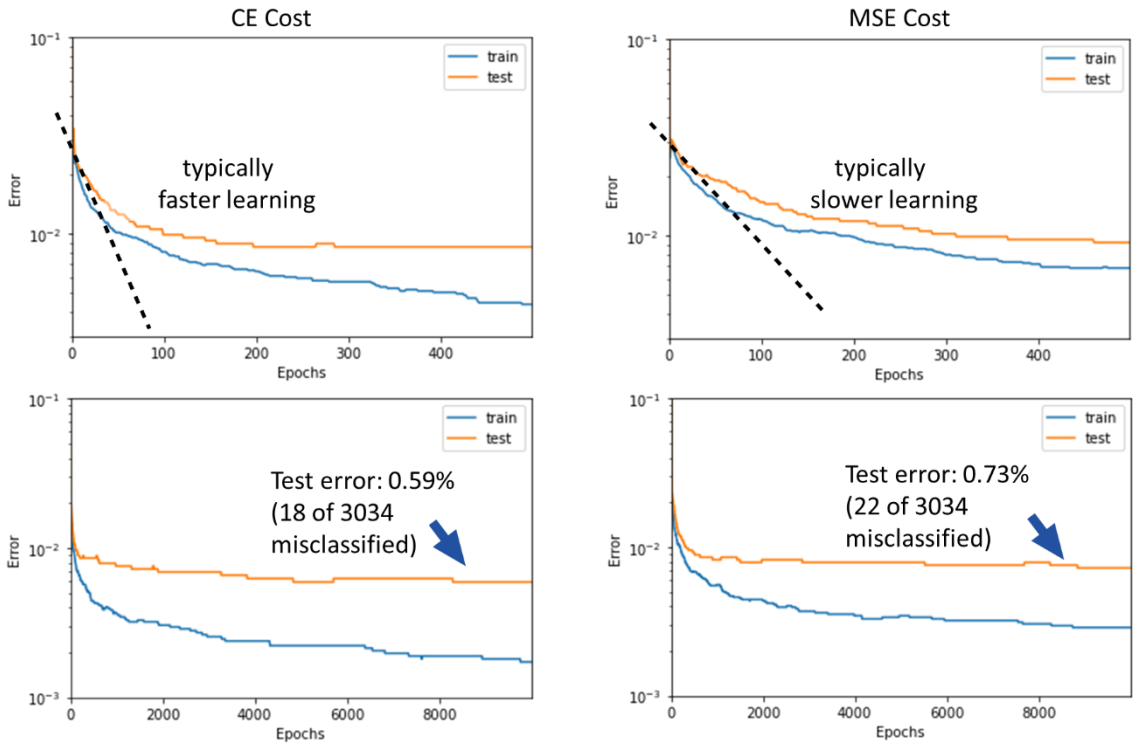


Figure 58: *Comparison of Cross Entropy (left) and Mean Squared Error (right) cost (details see text).*

3.4.7 Summary of Results on binary Classification

The Table 1 below summarizes the results on all pairs of digits for the binary classification problem on the MNIST dataset. The settings used to obtain these results were:

- Training for 5000 epochs
- Learning rate $\alpha = 0.5$
- Cross Entropy cost function
- Weight and biases (w_k, b) initialised with zero values

As noted in the preceding chapter 3.4.6 for cross entropy we obtain a convex cost function, and the results does not depend on the initial values. The fact that the confusion matrix shown below is not symmetric can therefore be only due to different training and test set choices when switching the digits $i \leftrightarrow j$. We note the following:

- The Digits '3', '5' and '8' are often involved in classifiers with high error rates: (3,5), (3,8), (5,8)
- For digits (5,3) the error rate is largest.
- Digits '0' and '1' are well distinguishable from all the others.

i \ j	0	1	2	3	4	5	6	7	8	9
0		0.17%	1.40%	0.71%	0.36%	1.17%	0.65%	0.60%	0.95%	0.61%
1	0.27%		0.64%	0.67%	0.41%	0.28%	0.17%	0.59%	1.56%	0.51%
2	1.08%	0.91%		2.94%	1.23%	1.54%	1.23%	0.98%	2.21%	0.82%
3	0.64%	0.87%	3.08%		0.39%	4.61%	0.25%	1.28%	3.15%	1.56%
4	0.36%	0.31%	1.38%	0.64%		1.07%	1.06%	1.31%	0.77%	2.76%
5	0.76%	0.39%	1.95%	4.01%	1.22%		1.59%	0.66%	3.69%	1.32%
6	0.69%	0.14%	1.15%	0.46%	0.69%	1.63%		0.07%	0.95%	0.18%
7	0.25%	0.26%	0.95%	1.25%	1.03%	0.62%	0.07%		0.71%	4.24%
8	0.73%	1.84%	2.64%	3.19%	0.92%	3.61%	0.88%	0.96%		1.49%
9	0.47%	0.40%	1.04%	1.45%	3.34%	1.17%	0.11%	3.61%	1.67%	

Table 1: *Summary of results for binary classification on MNIST dataset using CE cost.*

The fact that the cost function for CE is convex is rather an exception due to the single layer of the chosen model architecture. In general, this is not the case, and we expect different results for different initial conditions (recall Figure 50 where this is illustrated). We can nevertheless study this issue using MSE cost because it is not convex and will lead to varying results. Therefore, we proceed as follows:

- We apply normalization of the input data (i.e., not only scaling) i.e.:

$$x_k \sim \frac{1}{\sqrt{n}} \cdot N(0,1)$$

  Here, $N(0,1)$ means a random Gaussian variable with zero mean and unit variance.
- We choose random initial weights w_k (and bias zero) according to the following rule:

$$w_k \sim \frac{1}{\sqrt{n}} \cdot N(0,1)$$

- These choices are motivated by the following observation.
  Given the above conditions when evaluating the term

$$z = \sum_{k=1}^{n} w_k \cdot x_k + b$$

  the so-called *Logit*, we will again obtain a distribution $z \sim N(0,1)$[50], a property which is beneficial for deep architectures and which we will study in chapter 7.1.
- We will perform 10 runs for digits (5,3) with MSE cost
- Training will be done for 5000 epochs
- The learning rate is chosen to $\alpha = 0.1$

Table 2 below shows the results of the 10 independent runs with an average error of $\bar{e} = 4.1\%$ and a standard deviation $\Delta e = 0.07\%$. Figure 59 shows a set of some 100 misclassified digits.

run	1	2	3	4	5	6	7	8	9	10
error rate	4.42%	4.65%	4.50%	4.42%	4.46%	4.46%	4.39%	4.39%	4.42%	4.50%

Table 2: *Different error rates obtained for 10 runs for digits (5,3) with MSE cost.*

[50] We use the independence of w_k and x_k leading to $\mathrm{var}(w_k \cdot x_k) = \mathrm{var}(w_k) \cdot \mathrm{var}(x_k)$ (with zero mean).

Figure 59: *Set of 100 misclassified digits for a binary (5,3) classification on the MNIST data set.*

3.5 Extension of Gradient Descent – Stochastic and Mini Batch GD

So far, we have computed the gradient of the cost function defined on all the training set. This procedure is referred to as *Batch* Gradient Descent. This computation can be costly because it involves the evaluation of the model and its derivatives for *all* the samples in the training set.
This is not an issue for small models like the single (layer) perceptron but for deep neural networks and if the training dataset is large this becomes a problem.
The idea of so-called Stochastic Gradient Descent and Mini-Batch Gradient Descent is based on the following observation. Recall Equation 1 for the general CE cost (MSE cost similar). It is expressed as an arithmetic mean over the per sample contributions ("Loss")[51]:

$$J_{CE}(\boldsymbol{\theta}) = -\boxed{\frac{1}{m}\sum_{i=1}^{m}} \log\, p_\theta(y^{(i)}|\mathbf{x}^{(i)})$$

Equation 3

We may assume that the update direction $\nabla_{\boldsymbol{\theta}} J_{CE}(\boldsymbol{\theta})$ is already approximated with fewer (than all) samples. Thus, depending on the extend of the sum the following GD variants can be formulated:

- Batch GD: Averaging over all training samples
- Mini-Batch GD: Averaging only over a subset of training samples
- Stochastic GD: No averaging, just use a single randomly selected sample

It is important to note that if we do not apply Batch GD there is in general no guarantee to move in the parameter space towards a direction that leads to smaller values of the cost function. This will become especially evident for Stochastic GD.

3.5.1 Stochastic Gradient Descent

For Stochastic GD we only select one single sample for each update step. We refer to an epoch as a full iteration over all samples m i.e., doing m independent GD update step with one single sample each. The formal update scheme (analogue to chapter 3.4.2) for the binary classification with CE cost is as follows:

[51] I.e., when we refer to the entire sum, we use the term CE "Cost", when only referring to a single term, we use the term CE "Loss".

1. Start with some initial value $\boldsymbol{\theta}_0$ for the parameter vector $\boldsymbol{\theta_0} = (\mathbf{w_0}, b_0)$.
   (e.g., random values or all 0)
2. Iteratively update the parameter vector $\boldsymbol{\theta} = (\mathbf{w}, b)$ by
   a. Selecting one training sample randomly $(\mathbf{x}^{(i)}, y^{(i)})$
   b. Step in the negative gradient direction according to:

$$\mathbf{w_{t+1}} = \mathbf{w_t} - \alpha \cdot (h_\theta(\mathbf{x}^{(i)}) - y^{(i)}) \cdot \mathbf{x}^{(i)}$$
$$b_{t+1} = b_t - \alpha \cdot (h_\theta(\mathbf{x}^{(i)}) - y^{(i)})$$

3. Stop when the change in parameter vector update step $(\boldsymbol{\theta}_t \rightarrow \boldsymbol{\theta}_{t+1})$ is small

As we will see the convergence may be trickier to observe as we may select favourable and less favourable points in the training set. In practice we may compute a sliding averaged cost over the past updates.

Results for Binary Classification on MNIST Dataset

Figure 60 below shows first results obtained for binary classification of the digits (1,7) using the generalised Perceptron with CE cost function. 20 epochs with 12136 update steps of one sample each have be performed.

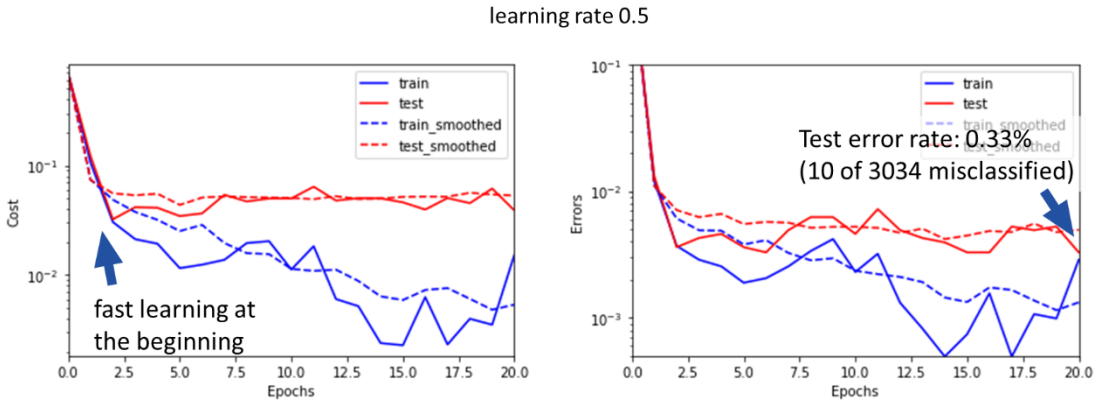


Figure 60: *Results of SGD for CE cost and error rate for binary classification of digits (1,7) on MNIST data set.*

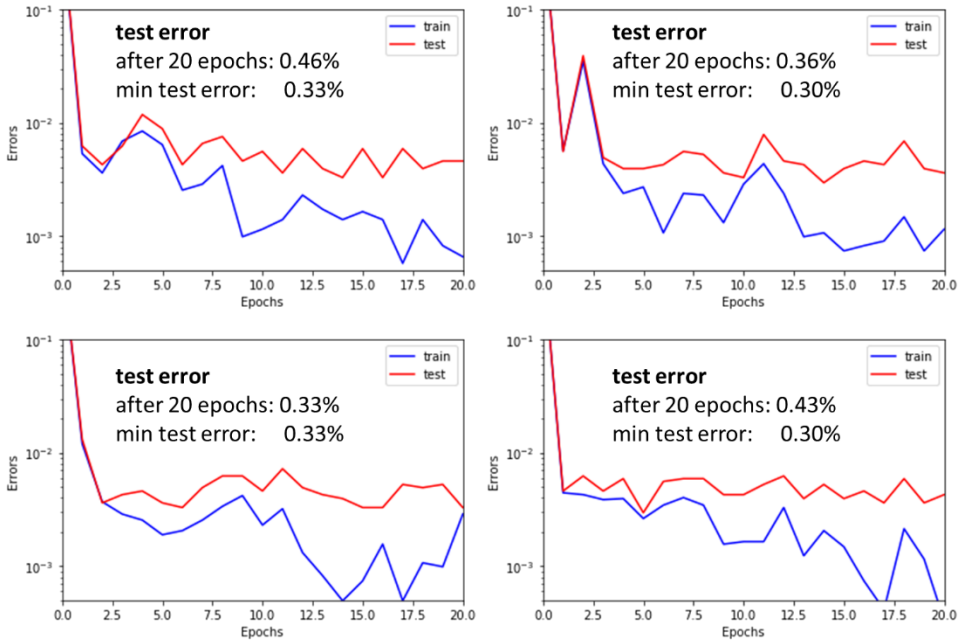


Figure 61: *Comparison of error rates for different runs using SGD (details see text).*

One immediately observes that the cost function and error rate are not monotonically decreasing but strongly fluctuate both for training and test set. Fluctuations are however smaller for the test set. We observer a fast learning for the first few epochs at the beginning of the optimization. When comparing with Batch GD (Figure 58) we observer that we require considerably less epochs for a "convergence" i.e., a decrease to a test error rate close to the optimum. But it should be noted that one epoch is more costly for SGD than for BGD in particular when computing the smoothed learning curves because this requires the evaluation of the entire sum over all m samples both for cost function and error rate.
When performing several runs even with identical parameter settings (Figure 61) the results will differ due to the random selection of the samples for each GD update step. Thus, the results have intrinsically a stochastic nature and must be interpreted accordingly.
Finally, Figure 62 shows the results for the same model as above now successively reducing the learning rate from left to right. While the value of $\alpha = 0.5$ was suitable for BGD we see that by choosing smaller learning rates we obtain much smoother behaviour of both cost function and error rates for SGD.

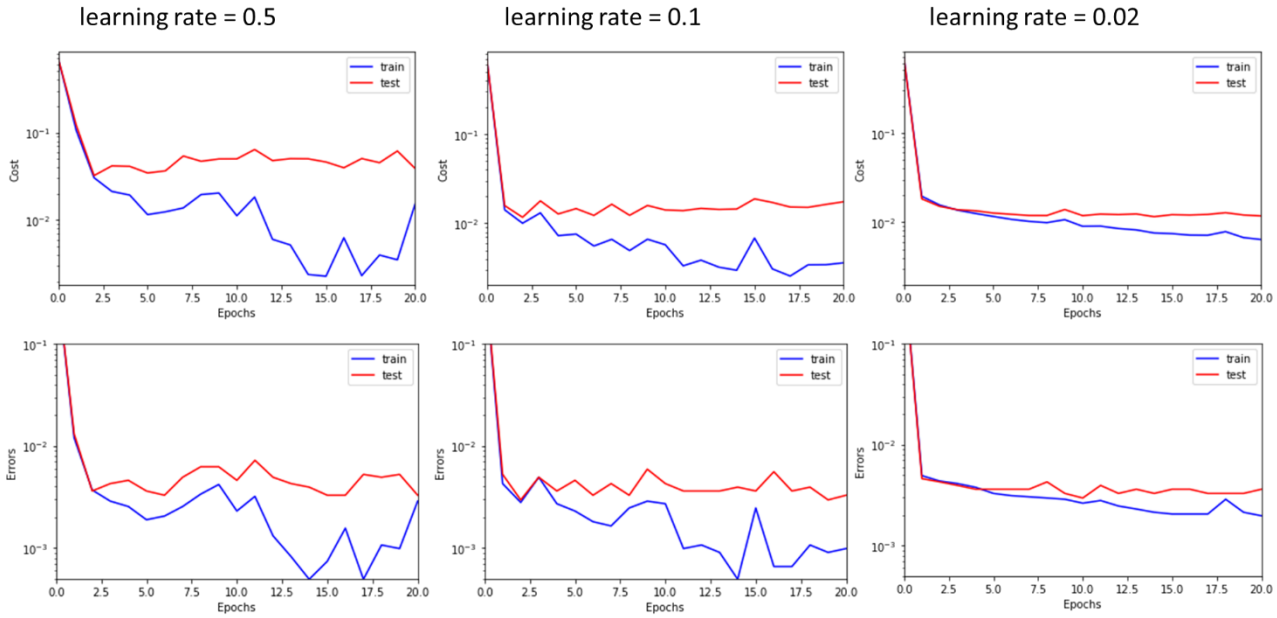


Figure 62: *Reducing the learning rate for SGD will reduce the fluctuations of both cost function and error rates.*

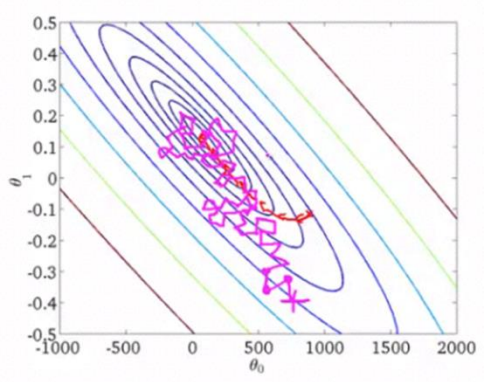
Exercise:
In groups of two discuss the following questions:

- Why do the cost function and error rate fluctuate less for the test set than for the training set?
- Why do the fluctuation of cost function and error rates decrease with decreasing learning rate α (Figure 62)?

Summarizing the characteristics of SGD, we can state:

General Characteristics

- SGD tends to move in direction of a local minimum, but not always.
- It never converges like batch gradient descent does but ends up fluctuating around the local minimum. If we are close enough to the minimum this is not a problem.
- It allows for escaping local minima, thus has a regularizing effect. We will address this important issue later.
- The learning principle is generalizable to many other "hypothesis families" (same as BGD in this respect).



Advantages

- SGD needs less epochs since parameters are updated for each training sample.
- It can handle very large sets of data and is great for learning on huge datasets that do not fit in memory ("out-of-core learning").
- It allows for incremental learning ("online learning") i.e., on-the-fly adjustment of the model parameters on new incoming data.

Disadvantage

- SGD cannot easily by parallelised.

3.5.2 Mini-Batch Gradient Descent

Mini-Batch Gradient Descent is in fact a compromise between Batch Gradient Descent and Stochastic Gradient Descent. The process is presented schematically in Figure 63. For each epoch, the m training samples are randomly shuffled and m/b equally sized batches (of sizes b each) are created. Then for each update step a batch is chosen randomly and a Batch GD step is performed by evaluating the sums (c.f. Equation 3) only over the samples contained in the batch. One epoch consists of one loop over all batches.

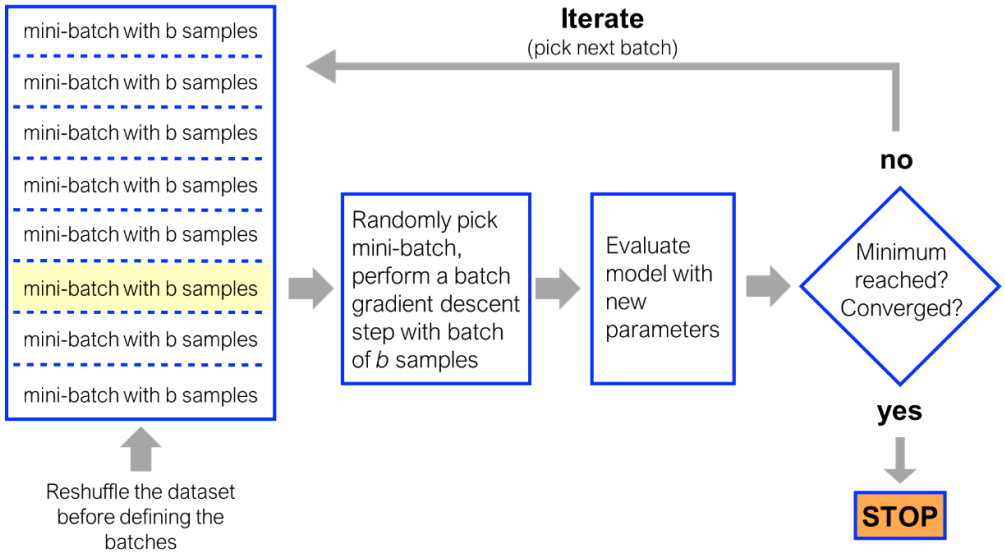


Figure 63: *Schematic presentation of MBGD (details see text).*

The formal update scheme (analogue to chapter 3.4.2) for the binary classification with CE cost is as follows:

1. Start with some initial value $\boldsymbol{\theta}_0$ for the parameter vector $\boldsymbol{\theta}_0 = (\mathbf{w}_0, b_0)$. (e.g., random values or all 0)
2. Iteratively update the parameter vector $\boldsymbol{\theta} = (\mathbf{w}, b)$ by
   a. Selecting one mini batch with indices $i_1, i_2, i_3, \ldots$ i.e., $\left(\mathbf{x}^{(i_1)}, y^{(i_1)}\right), \left(\mathbf{x}^{(i_2)}, y^{(i_2)}\right), \left(\mathbf{x}^{(i_3)}, y^{(i_3)}\right), \ldots$ randomly
   b. Step in the negative gradient direction according to:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \cdot \frac{1}{b} \sum_{j=1}^{b} \left(h_\theta(\mathbf{x}^{(i_j)}) - y^{(i_j)}\right) \cdot \mathbf{x}^{(i_j)}$$

$$b_{t+1} = b_t - \alpha \cdot \frac{1}{b} \sum_{j=1}^{b} \left(h_\theta(\mathbf{x}^{(i_j)}) - y^{(i_j)}\right)$$

3. Stop when the change in parameter vector update step ($\boldsymbol{\theta}_t \rightarrow \boldsymbol{\theta}_{t+1}$) is small

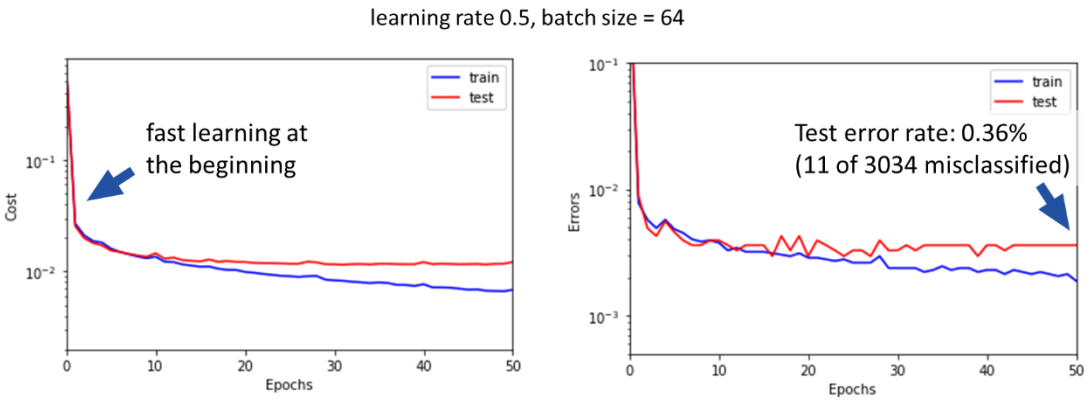


Figure 64: *Results of CE cost and Error rate for binary classification of digits (1,7) on MNIST data set.*

The Figure 64 above shows the results for our binary (1,7) classification on the MNIST dataset for MBGD. A batch size of $b = 64$ was chosen, the learning rate was $\alpha = 0.5$. We observer that for identical learning rate the learning curves are smoother than with SGD, however, the smoothness will obviously depend on batch size. After 50 epochs an almost optimal test error rate of ~0.36% was reached (11 out of 3034 wrong) i.e., we observe – like SGD – a much faster convergence than for BGD.

Summarizing the characteristics of MBGD, we can state:

General Characteristics

- MBGD tends to move in direction of a local minimum, but not always.
- It ends up wandering closely around the local minimum.
- It allows for escaping local minima, thus has a regularizing effect. We will address this important issue later.
- As for BGD, the learning principle is generalisable to many other “hypothesis families”.

Advantages

- MBGD is faster than BGD since parameters are updated for each mini-batch.
- It can handle very large sets of data and is great for learning on huge datasets that do not fit in memory (“out-of-core learning”).
- It allows for incremental learning (‘online learning’) i.e., on-the-fly adjustment of the model parameters on new incoming data.
- It can be parallelised on GPU and in HPC.

Disadvantage

- The only inconvenience is that we have an additional hyperparameter, the batch size, that needs to be optimised.

Table 3 below summarizes the properties of the three optimisation schemes. Note the difference in learning rate for the three graphs. The comparison shows that the three schemes allow for different trade-offs concerning the learning rate, the noise or fluctuation of the learning curves, the batch size, and the possibility of parallelisation.

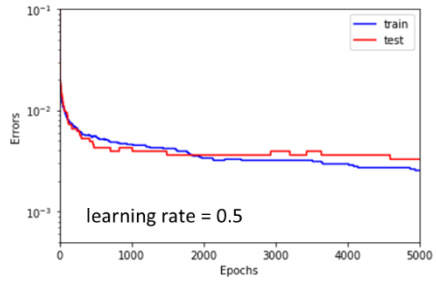
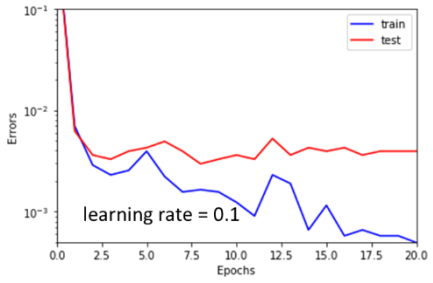
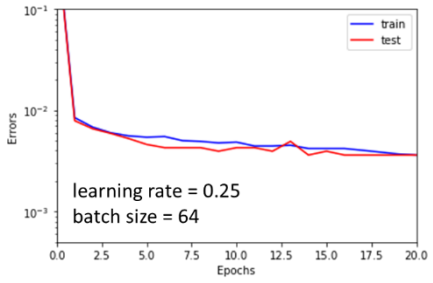
BGD	SGD	MBGD
		
Smooth Not wiggling	Wiggling, needs smoothing Wiggles around minimum	Slightly wiggling. Wiggles around minimum
Strictly decreasing cost	Not necessarily decreasing cost	Typically decreasing cost
Many epochs needed	Few epochs needed	Less epochs than BGD, more than SGD needed
Choose larger learning rate	Choose smaller learning rate	Choose medium learning rate (dependent on model)
No out-of-core support – all data in RAM ($\sim m$).	Out-of-core support - not all data to be kept in RAM of a single machine.	Out-of-core support - not all data to be kept in RAM of a single machine
Easy to parallelise	Not easy to parallelise	Easy to parallelise

Table 3: *Summary of the differences between BGD, SGD, and MBGD. Note the difference in learning rate.*

3.6 Multi-Class Classification and Softmax Activation

So far, we restricted our models to binary classification as represented in Figure 48. We now want to generalise our task to K independent classes as shown in Figure 65 below and will only use CE cost from now on. The index l $(0 \leq l < K)$ represents the class index and now we have K different output neurons. This means that our parameter space increases because each input neuron is connected to each output neurons such that we now have K independent parameter vectors $\boldsymbol{\theta}_l = (\mathbf{w}_l, b_l)$. However, because the output neurons have no "knowledge" of each other we basically have K independent binary classifiers where each digit is trained in comparison to all other digits – in a binary fashion. To choose between the K different outputs we would then simply choose the highest prediction $h_{\theta_l}(\mathbf{x})$ for all $0 \leq l < K$.

$$0 \leq l < K$$

$$h_{\theta_l}(\mathbf{x}) = \sigma(\mathbf{w}_l \cdot \mathbf{x} + b_l)$$

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

$$\theta_l = (\mathbf{w}_l, b_l)$$

$$\hat{y}(\mathbf{x}) = \underset{l}{argmax}\{h_{\theta_l}(\mathbf{x})\} \in \{0, 1, \ldots, K-1\}$$

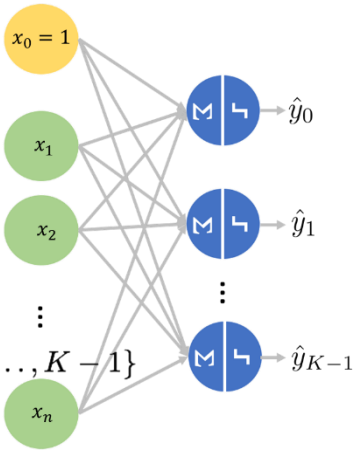


Figure 65: *Generalisation of the classification task to K independent classes (details see text).*

Figure 66 shows the result of such a procedure for our generalised perceptron applied to the MNIST dataset. For each digit the final test error after training of 50 epochs using MBGD with a batch size of 64 is shown. I.e., an error corresponds to the fact that the value of

$$\underset{l}{\text{argmax}}\left(h_{\theta_l}(\mathbf{x})\right)$$

taken over all $0 \leq l < 10$ binary classifiers does not give the correct prediction. The overall error rate is of the order of 9%.

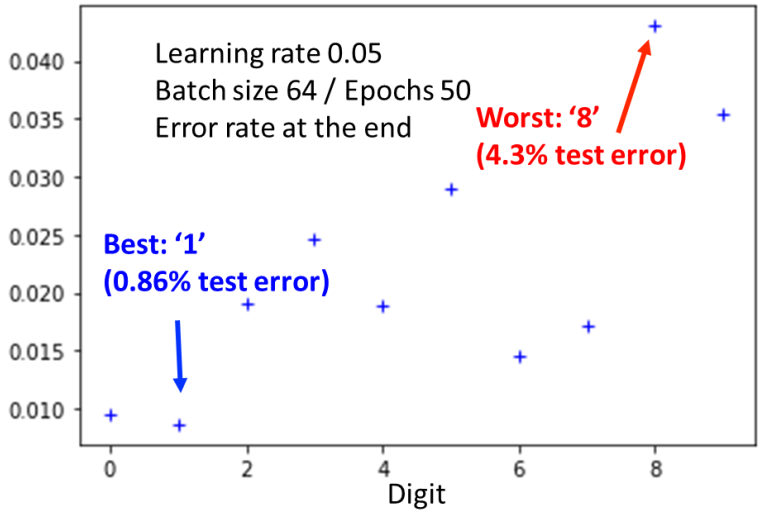


Figure 66: *Result of 10 mutual binary classifiers on the MNIST dataset for multi-class classification.*

Recall that the CE cost function for our binary classification task (Equation 2) can be interpreted as class probability (c.f. Equation 1 for the general definition of CE cost). The problem with the simple approach shown in Figure 65 is that the model is not trained to provide normed probabilities over all the K classes. For a binary classification this is not a problem because if $h_{\theta_l}(\mathbf{x})$ is the probability for one class the probability of the other class can be deduced by $1 - h_{\theta_l}(\mathbf{x})$, which we used to derive Equation 2. To extend our approach to K classes we therefore introduce the so-called Softmax function and corresponding layer (Figure 67).

$$0 \leq l < K$$

$$z_l = \mathbf{w}_l \cdot \mathbf{x} + b_l$$

$$h_{\theta,l}(\mathbf{x}) = p_l = \frac{\exp \mathbf{z}_l}{\sum_{j=0}^{K-1} \exp \mathbf{z}_j}$$

$$\theta_l = (\mathbf{w}_l, b_l)$$

$$\hat{y}(\mathbf{x}) = \underset{l}{argmax}\{h_{\theta_l}(\mathbf{x})\} \in \{0, 1, \ldots, K-1\}$$

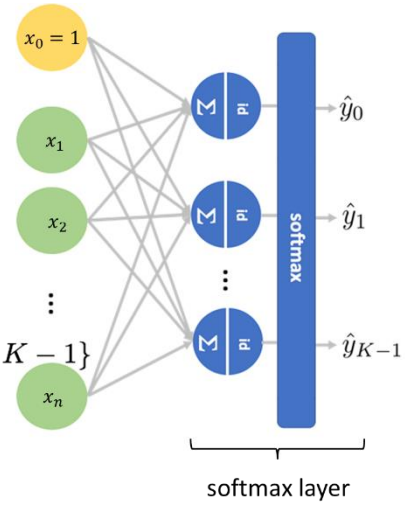


Figure 67: *General formulation of the Softmax layer providing normed probabilities over all K output classes.*

Therefore, we replace the non-linearity (sigmoid function) in the output neurons by the identity function ("id" in Figure 67) and normalize the K outputs (given by the Logits $z_l = \mathbf{w}_l \cdot \mathbf{x} + b_l$)[52] according to the following scheme:

$$h_{\theta,l}(\mathbf{x}) = \frac{\exp z_l}{\sum_{j=0}^{K-1} \exp z_j}$$

Because the sum is taken over all K possible outcome values, $h_{\theta,l}(\mathbf{x})$ is normalized:

[52] Note that the product $\mathbf{w}_l \cdot \mathbf{x}$ is in vector notation i.e., represents a scalar product of the vector $\mathbf{w}_l$ and $\mathbf{x}$.

$$\sum_{l=0}^{K-1} h_{\theta,l}(\mathbf{x}) = 1$$

Thus, we can interpret $h_{\theta,l}(\mathbf{x})$ as conditional probability $p_\theta(y = l|\mathbf{x})$ of finding output $y = l$ given the input $\mathbf{x}$. This has the advantage to use them directly for the formulation of the CE cost function, if we recall the general expression in Equation 1:

$$J_{CE}(\boldsymbol{\theta}) = -\frac{1}{m}\sum_{i=1}^{m} \log\, p_\theta\left(y^{(i)}\middle|\mathbf{x}^{(i)}\right) = -\frac{1}{m}\sum_{i=1}^{m} \log\, h_{\theta,y^{(i)}}\left(\mathbf{x}^{(i)}\right)$$

3.6.1 GD Update with Softmax

We can now apply our BGD schemes (chapters 3.4.2) (or its extension to MBGD or SGD) to the Softmax output. Therefore, we must calculate the gradient of the CE cost function with respect to the parameter vectors $\boldsymbol{\theta_l} = (\mathbf{w_l}, b_l)$ $(0 \leq l < K)$ i.e.:

$$\frac{\partial}{\partial \theta_l} J_{CE}(\boldsymbol{\theta})$$

Again, we can focus on the derivate of the terms under the sum. However, the situation is slightly more complicated because our parameter vector $\boldsymbol{\theta_l} = (\mathbf{w_l}, b_l)$ has now the subscript $l, (0 \leq l < K)$ which represents the output class. In the previous calculations (e.g., chapter 3.4.5) the subscript θ_k represented the vector character of $\boldsymbol{\theta}$. To keep things clear we will now directly calculate the derivative with respect to the vector $\mathbf{w_l}$ i.e., we will represent the vector character through the bold face letter and therefore can use the subscript for the output class:

$$\frac{\partial}{\partial \mathbf{w}_l} \log\, h_{\theta,y^{(i)}}\left(\mathbf{x}^{(i)}\right) = \frac{\partial}{\partial \mathbf{w}_l} \log\left[\frac{\exp z_{y^{(i)}}}{\sum_{j=0}^{K-1} \exp z_j}\right] =$$

$$\frac{\partial}{\partial \mathbf{w}_l}\left[z_{y^{(i)}} - \log \sum_{j=0}^{K-1} \exp z_j\right] =^{(1)} \left[\delta_{l,y^{(i)}} - \frac{\exp z_l}{\sum_{j=0}^{K-1} \exp z_j}\right] \cdot \mathbf{x}^{(i)} =$$

$$\left(\delta_{l,y^{(i)}} - h_{\theta,l}\left(\mathbf{x}^{(i)}\right)\right) \cdot \mathbf{x}^{(i)}$$

In the step indicated by $=^{(1)}$ we used:

1. The expression for the logits
$$z_j = \mathbf{w}_j \cdot \mathbf{x}^{(i)} + b_j$$
2. The derivative of the logit with respect to $\mathbf{w}_l$:
$$\frac{\partial}{\partial \mathbf{w}_l} z_j = \frac{\partial}{\partial \mathbf{w}_l}\left[\mathbf{w}_j \cdot \mathbf{x}^{(i)} + b_j\right] = \mathbf{x}^{(i)}$$
3. The fact that
$$\frac{\partial}{\partial \mathbf{w}_l} z_{y^{(i)}}$$
is zero unless $l = y^{(i)}$ which can be expressed by the Kronecker delta $\delta_{l,y^{(i)}}$.

We already know that the derivative with respect to b_l leads to the identical expression as obtained for $\mathbf{w}_l$ but without the final vector $\mathbf{x}^{(i)}$. Putting everything together we obtain the derivatives of the Softmax layer for GD update rules as follows:

$$\nabla_{\mathbf{w}_l} J_{CE}(\mathbf{w}, b) = -\frac{1}{m}\sum_{i=1}^{m}\left(\delta_{l,y^{(i)}} - h_{\theta,l}\left(\mathbf{x}^{(i)}\right)\right)\cdot \mathbf{x}^{(i)}$$

$$\nabla_{b_l} J_{CE}(\mathbf{w}, b) = -\frac{1}{m}\sum_{i=1}^{m}\left(\delta_{l,y^{(i)}} - h_{\theta,l}\left(\mathbf{x}^{(i)}\right)\right)$$

And the corresponding GD update steps:

$$\mathbf{w}_l \leftarrow \mathbf{w}_l - \alpha \cdot \nabla_{\mathbf{w}_l} J_{CE}(\mathbf{w}, b)$$

$$b_l \leftarrow b_l - \alpha \cdot \nabla_{b_l} J_{CE}(\mathbf{w}, b)$$

3.6.1.1 Results for MNIST Dataset

In Figure 68 the result for the Softmax layer and multi-class classification is shown. The parameters are chosen as follows:

- Optimisation scheme is MBGD
- Batch size is $b = 256$
- Learning rate is $\alpha = 0.05$
- Number of epochs 100

The final test error rate is 8.0%.

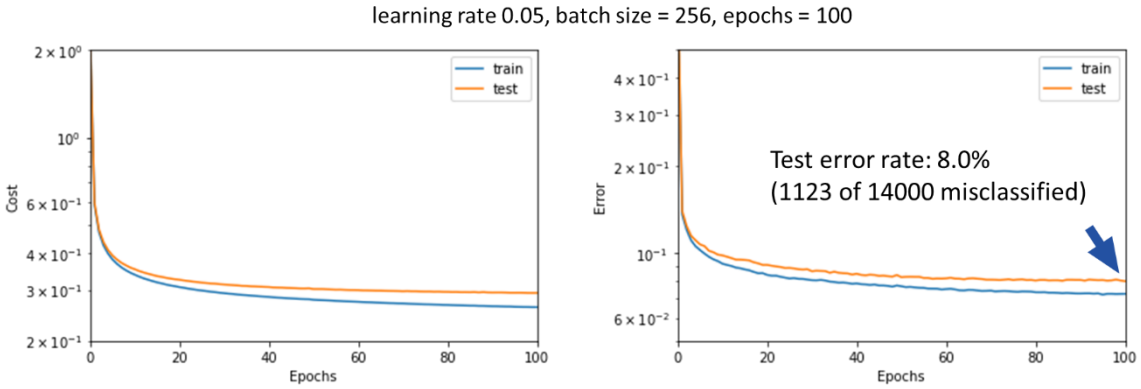


Figure 68: *Multi-class classification using MBGD on MNIST dataset using a Softmax layer.*

real label	false label
[8 3 7 1 5 8 4 0 7 7]	[2 2 9 8 0 1 9 8 4 9]
[7 0 9 9 2 3 3 4 9 5]	[9 3 3 4 7 8 7 9 4 9]
[7 5 7 2 7 7 4 8 9 9]	[4 8 8 9 9 4 9 1 4 7]
[9 2 7 7 0 2 1 8 4 3]	[8 4 9 1 8 7 8 2 5 0]
[7 9 5 7 9 5 5 8 8 2]	[2 7 6 1 2 8 3 5 2 1]
[3 5 6 8 5 2 3 4 2 2]	[5 1 9 4 3 7 5 6 3 3]
[9 8 6 5 4 4 5 5 6 7]	[5 1 2 6 9 6 6 0 4 9]
[0 4 3 9 2 8 5 8 9 3]	[3 9 5 4 4 2 2 9 7 8]
[7 8 8 2 3 1 6 8 8 6]	[9 3 9 4 5 8 5 5 9 2]
[6 7 5 3 5 9 7 8 9 8]	[8 2 2 5 4 7 1 7 1 1]

Figure 69: *Set of 100 misclassified MNIST digits, in the centre the correct, to the right the false labels.*

The Figure 69 shows 100 of the 1123 misclassified images with the correct (middle) and wrong (right) labels given. Finally, in Figure 70 the weight vectors $\mathbf{w}_l$ $(l = 0..9)$ for the ten output neurons are shown

arranged as 28x28 image patches i.e., corresponding to the original MNIST data size. As the terms $\mathbf{w}_l \cdot \mathbf{x}$ represents scalar products of two vectors this means that a given image patch from the MNIST dataset is multiplied pixel by pixel with the corresponding weight vector in Figure 70. Apparently, the weight vectors have learned some of the structure of the respective digit.

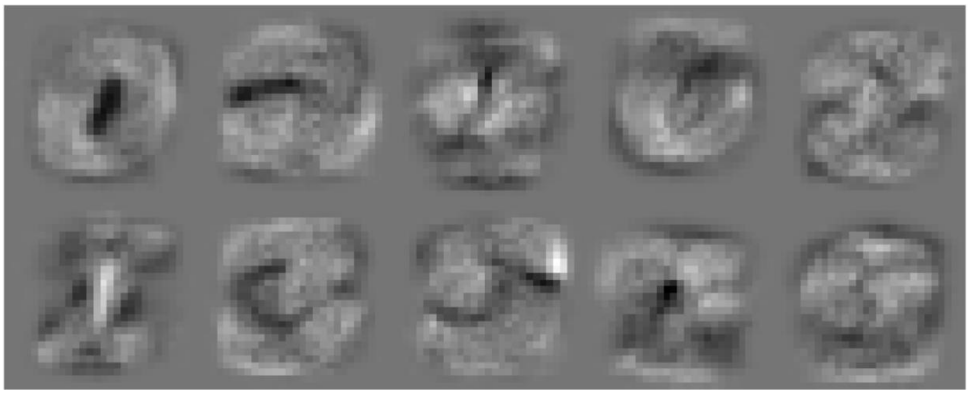


Figure 70: *The weight vectors $\mathbf{w}_l$ ($\boldsymbol{l} = \mathbf{0}..\mathbf{9}$) (top 0, 2, 4, 6, 8; bottom 1, 3, 5, 7, 9) after training organized as 28x28 patches (i.e., as the MNIST images). The weights apparently have learned the structure of the corresponding digits.*

Analysing these results immediately raises the question how good we may expect a classifier to be? The best performance on this data set is of the order of 0.2%[6]. To achieve this performance, larger models with higher representational capacity are needed including (much) more model parameters. While we currently lack the techniques to train those models, we will nevertheless have a first look at possible extensions.

3.7 Extended Model Capacity – MLP with one Hidden Layer

We present results obtained with a Multi-Layer Perceptron including one hidden layer with n_1 neurons. This hidden layer is of fully connected type and each input neuron is connected to each hidden neuron (●-●). In addition, each hidden neuron is connected to each neuron of the softmax output layer (●-●). Finally, each neuron (hidden and softmax) is connected to a bias neuron (●-●).

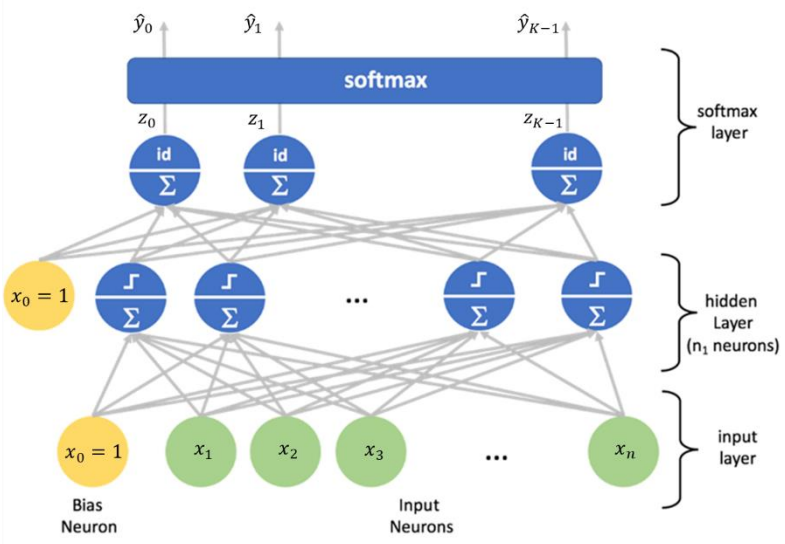


Figure 71: *A first model with considerably higher representational capacity including one hidden layer.*

Figure 72 to Figure 75 present the corresponding training curves as function of different hyperparameters (learning rates, batch sizes, number of hidden neurons). Training was done with MBGD using 100 epochs for each run. The curves are obtained with the ML framework tensorflow/keras which we will use in later chapters.

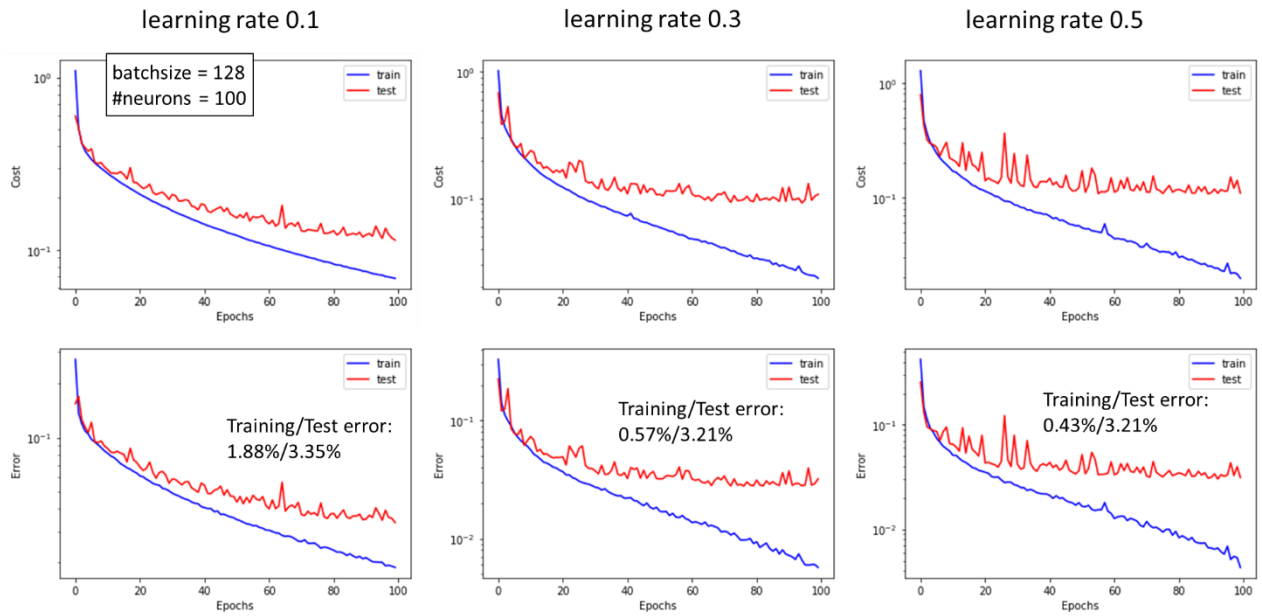


Figure 72: *Results for MLP model with single hidden layer (Figure 71) with varying learning rate.*

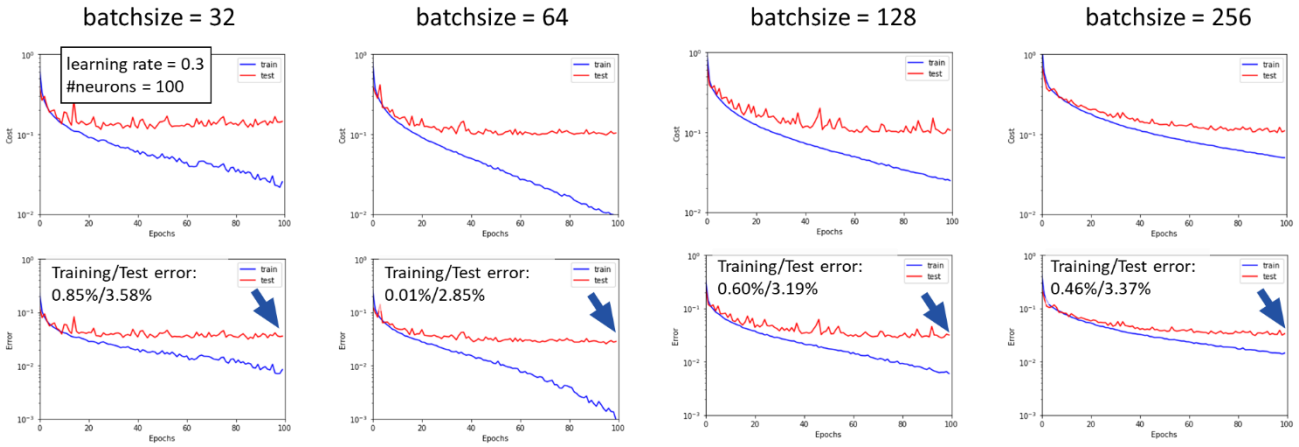


Figure 73: *Results for MLP model with single hidden layer (Figure 71) with varying batch size.*

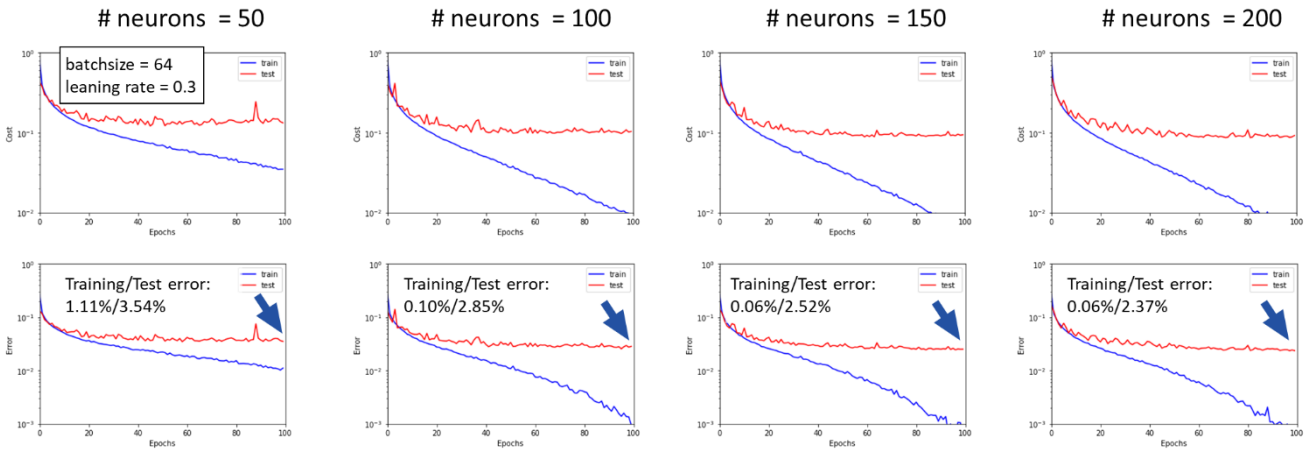


Figure 74: *Results for MLP model with single hidden layer (Figure 71) with varying number of hidden neurons.*

The following points can be noted:

- Increasing the learning rate increase the uncertainty in the GD step and thus the fluctuations of the test error rates (Figure 72). The error rates for the smallest learning rate (left) may not yet have reached their final values, as they are considerably larger than for the other curves.
- An intermediate batch size of 64 leads to the fasted convergence.
- Increasing the number of hidden neurons increases the representational capacity of the network and allows for smaller error rates (Figure 74).

To further exploit the representational capacity of this model (MLP with one hidden layer) the number of hidden neurons was increased even more (than in Figure 74) and the behaviour of the test error rate was observed. The batch size was chosen to 64 (presumably best value from Figure 73) and the learning rate to 0.3 (compromise between fluctuations and convergence, Figure 72). The results are shown in the following Figure 75. Apparently, the increase in number of hidden neurons reduces the test error rate from ~8.0% down to ~2.4%. However, the test error rate reaches a plateau at around 200 neurons with a final value of ~2.4% and no further decrease can be observed.

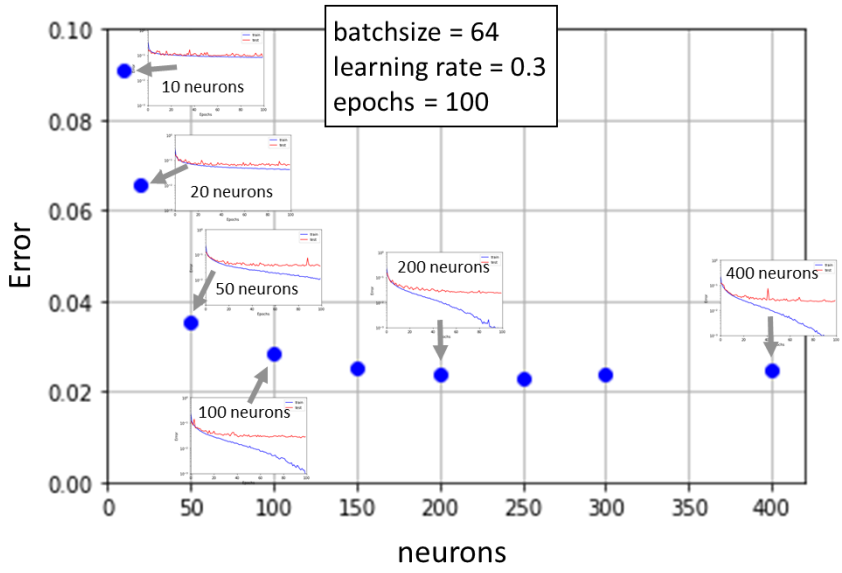


Figure 75: *Test error rate for MLP model with single hidden layer (Figure 71) over number of hidden neurons.*

From our first study on increased representational capacity, we can conclude that the addition of a hidden layers improves the result for the test error. However, various questions remain leading us to topics treated in the next chapters.

- What improvements are necessary to further boost the performance from ~2.4% down to ~0.2% test error, as given in this reference[6]? In the next chapters we will study:
  - Better strategies for choosing/evolving the learning rate and number of epochs, the so-called hyperparameter tuning.
  - Further improvements of the GD optimisation scheme.
  - A model architecture especially suited for images the so-called Convolutional Neural Networks (CNN)
- Could we just add more hidden layers to improve the performance? It turns out that this is not the case for MNIST. A possible explanation is that the correlation between the different pixels is captured already with a single layer and there is not much more structure beyond to be captured for MNIST.
- One major question remains which is how to train such MLP architectures that we just used for our preliminary study. We will develop these formulas when applying gradient descent to networks with hidden layers in the context of the so-called back-propagation algorithm.

3.8 A note about activation functions

We already saw the important role of the choice of activation function when moving from the Rosenblatt Perceptron (chapter 2.2.2) with Heaviside step function to the generalized Perceptron with sigmoid function as first step (chapter 3.3) and finally using the Softmax function (chapter 3.6). In this chapter we will give an overview of activation functions currently used in ML. Before that two important points shall be addressed.

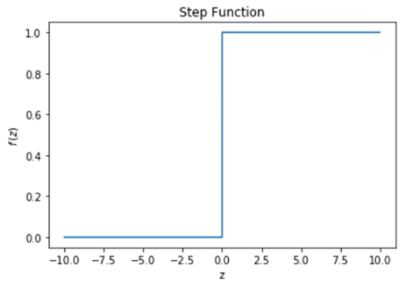
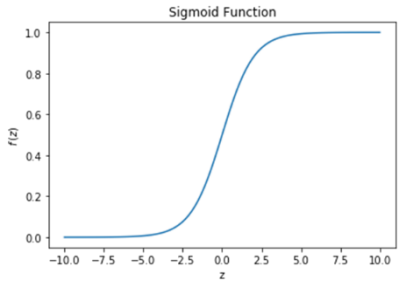
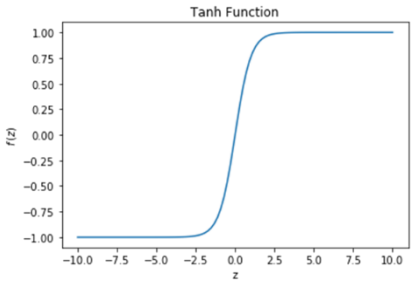
1. **Non-Linearities are crucial for sufficient representational capacity of the model**
   The activation function $f(z)$ maps the logit z to the final output $h(\mathbf{x})$ of the layer.

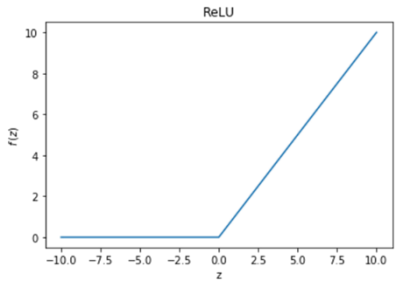
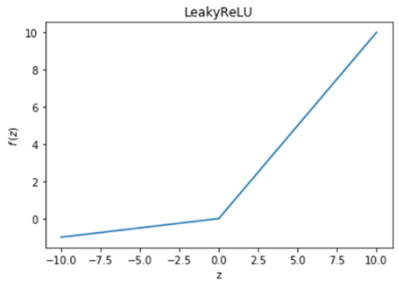
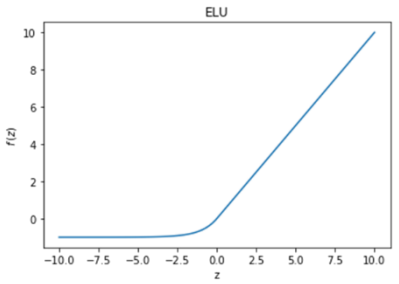
   $$h(\mathbf{x}) = f(z) = f(\mathbf{w} \cdot \mathbf{x} + b)$$

   The logit z is an affine function of the input values $\mathbf{x}$. If the activation function $f(z)$ is a pure linear function the entire layer would be affine. Stacking more layers on top of each other to an MLP would not change the affine nature of the full MLP and we would always keep the (low) representational capacity of an LTU with – optionally – final Softmax layer (c.f. Figure 67). Thus, non-linearities in the mapping between input and output of a neural network are crucial for gaining sufficient power for learning a task with sufficient accuracy.

2. **Robustness and Performance of Learning Algorithm**
   As we will see later the choice of activation function has an impact on the robustness and performance of the learning algorithm. Different activation functions have been introduced to improve the robustness and performance especially in the context of deep architectures.

The following Table 4 gives an overview of commonly used activation function and their major characteristics.

Heaviside	Sigmoid	Hyperbolic Tangent
		
$f(z) = \begin{cases} 1 & (z \geq 0) \\ 0 & (z < 0) \end{cases}$	$f(z) = \frac{1}{1+\exp(-z)}$	$f(z) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}$
As in Rosenblatt perceptron. Not differentiable i.e., gradient descent not possible. No practical use.	Most used in textbooks and in illustrative examples. In practice, typically used in output layers in binary classification. Smooth and differentiable i.e., gradient descent works. But saturation regions leading to vanishing gradients.	Often preferred over sigmoid since the output is centred around 0. In practice used e.g., in LSTM. Smooth i.e., gradient descent works. But saturation regions leading to vanishing gradients.

Rectified Linear Unit (ReLU)	Leaky ReLU	Exponential Linear Unit
		
$f(z) = \max(0, z)$	$f(z) = \max(\alpha z, z)$	$f(z) = \begin{cases} \alpha\,(\exp(z) - 1) & (z < 0) \\ z & (z \geq 0) \end{cases}$
Used as de facto standard. Introduced to alleviate the vanishing gradient problem. However, suffers from dying units problem for $z < 0$ where the activation and the gradient is zero.	Alleviates both, the vanishing gradient problem and the dying units problem. Uses a small hyperparameter $0 < \alpha < 1$ which makes sure that the unit never dies (typical $\alpha = 0.01$).	Similar to Leaky ReLU. Negative activation saturates at $-\alpha$ but the gradient vanishes at small negative values. More expensive to compute.

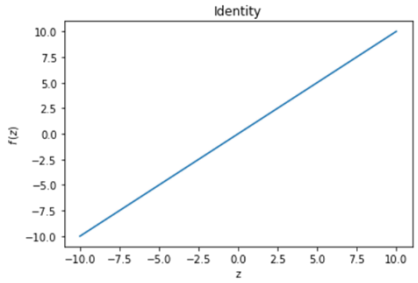
Identity	Sofmax	
		
$f(z) = z$	$f(z_l) = \frac{\exp z_l}{\sum_{j=0}^{K-1} \exp z_j}$	
Used only in specific cases such as in the last layer of a network for performing a regression task.	Used as last layer in a network for a classification task with K classes. Output vector can be interpreted as probability distribution.	.

Table 4: *Overview of all activation functions commonly used in NN architectures.*

3.9 Universal Approximation Theorem

In chapter 3.7 we started to have a look at larger models with higher representational capacity and more model parameters. Therefore, we increased the model size from a single layer to a multi-layer architecture with one hidden layer having increasing number of hidden neurons (Figure 75). This allowed us to significantly improve the classification result for our toy model.
Now, from a scientific point of view, it would be interesting to know whether the full representational capacity of a given architecture can be specified. It turns out, that the model studied in chapter 3.7 i.e., a MLP with one hidden layer, already has a very high representational capacity which is represented by the “Universal Approximation Theorem” formulated below. Because this theorem is expressed as a regression theorem, we will formulate our classification problem as such. Therefore, we will consider our models $h_\theta(\mathbf{x})$ as an approximation for a general mapping $f(\mathbf{x})$ from the input space $\mathbf{x} \in \mathbb{R}^{n_x}$ to the

output space $\mathbf{y} \in \mathbb{R}^{n_y}$ (in our case $\mathbf{x}$ are the images with $n_x = 784$ components and the results $\mathbf{y}$ with $n_y = 10$ are the Softmax output values). In fact, $f(\mathbf{x})$ corresponds to the hypothetical mapping fulfilling the given classification i.e., providing a perfect mapping $f(\mathbf{x}) = \mathbf{y}$ between the images and the corresponding correct output (Figure 40). The model $h_\theta(\mathbf{x})$ represents a family of functions that depend on the parameters θ which allows to approximate the function $f(\mathbf{x})$ such that:

$$\mathbf{y} = f(\mathbf{x}) \approx h_\theta(\mathbf{x}) = \hat{\mathbf{y}}$$

As before, we use $\hat{\mathbf{y}}$ as our estimator for the true outcome $\mathbf{y}$. The richness of this family of functions $h_\theta(\mathbf{x})$ determines their ability to represent more or less complex mappings $f(\mathbf{x})$. This ability is referred to as *representational capacity*[53]. A very general statement now proves [7] that shallow networks (i.e., networks with a single hidden layer) provide sufficient capacity for approximating quite general functions:

Universal Approximation Theorem

A feedforward network with a linear output layer and at least one hidden layer with a non-linear activation function (e.g. sigmoid) can approximate a large class of functions[54] $f: \mathbb{R}^{n_x} \to \mathbb{R}^{n_y}$ with arbitrary accuracy[55] – provided that the network is given a sufficient number of hidden units and the parameters are suitably chosen.

Exercise:
In groups of two discuss the following question:

- The statement made by the “Universal Approximation Theorem” is strong! What could it mean?
- Are all problems solved? What points are missing for a practical application?
- It makes a statement about functions. But we typically start with data. What is the connection?

We will now try to develop some intuition on the idea behind the Universal Approximation Theorem by “proving” it for the case of a 1-dimensional function.

3.9.1 Function Approximation with Sigmoids in 1D

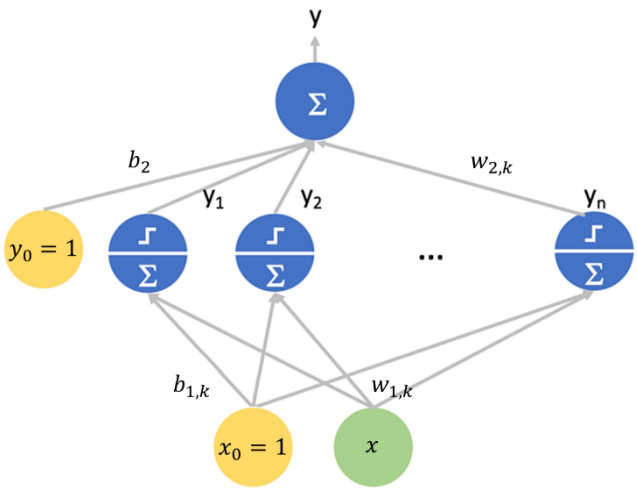


Figure 76: *Network architecture used to illustrate the Universal Approximation Theorem in 1D.*

[56]We will illustrate the Universal Approximation Theorem with a real valued function in one dimension

[53] This can be considered as a formal definition of this term, which we used so far only “intuitively”.
[54] E.g., any continuous function on compact support.
[55] Accuracy quantified with suitable distance measure (e.g., L^2-distance ~ integrated mean-square distance).
[56] The following was inspired by http://neuralnetworksanddeeplearning.com/chap4.html.

i.e., $f: \mathbb{R}^1 \to \mathbb{R}^1, f(x) = y$. The model $h_\theta(x)$ is given in Figure 76 having the following characteristics:

- One-dimensional real-valued input x.
- One hidden layer with n neurons and sigmoid activation function, weights $w_{1,k}$ and biases $b_{1,k}$.
- Linear output layer with weights $w_{2,k}$ and bias b_2.
- The linear output layer leads to a linear combination of sigmoid functions:

$$h_\theta(\mathbf{x}) = \sum_{k=1}^{n} w_{2,k} \cdot \sigma\left(w_{1,k} \cdot x + b_{1,k}\right) + b_2$$

The theorem now states that we can approximate any quite general function if we choose suitable parameters and a suitable number of neurons n. To illustrate this property, we will re-parametrise the sigmoid functions like:

$$\sigma(w \cdot x + b) = \sigma(w \cdot (x - s)) \quad \text{where} \quad s = -\frac{b}{w}$$

The value of s represents the position of the step and the parameter $1/w$ corresponds to the width of the step i.e., the larger w the smaller the width as indicated in the Figure 77 below.

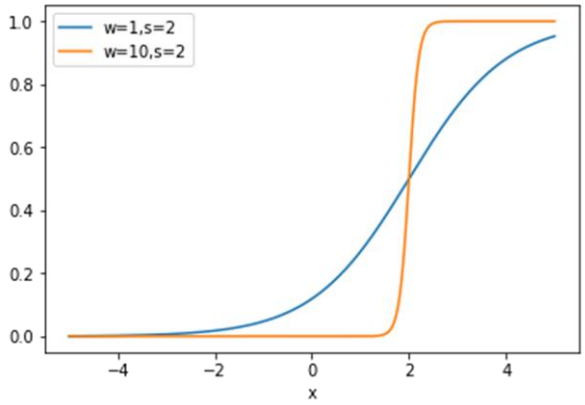


Figure 77: *Illustration of the effect of the parameter* ***w*** *on the slope of the step.*

Now we subtract two such step functions but shift each by a value of $\Delta/2$, but in opposite directions:

$$a \cdot [\sigma(w \cdot (x - s + \Delta/2)) - \sigma(w \cdot (x - s - \Delta/2))]$$

This will create a peak as shown in Figure 78 (right). The value of a just scales the final peak height (being 3 in the example). The left-hand side of Figure 78 shows the corresponding model.

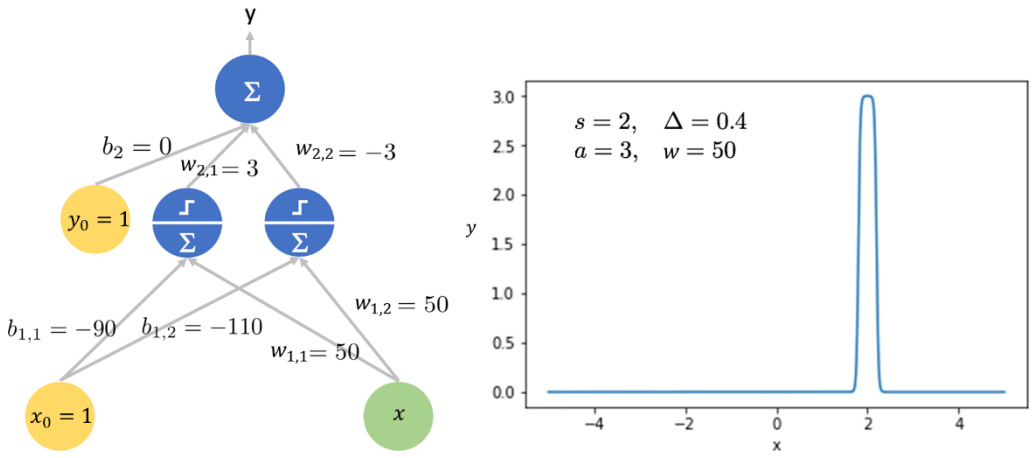


Figure 78: *Combination of two "steep" step functions allows to construct a peak.*

Linearly combining such peaks at different locations allows to construct arbitrary stepwise functions which permits to approximate a large class of functions as illustrated in Figure 79.

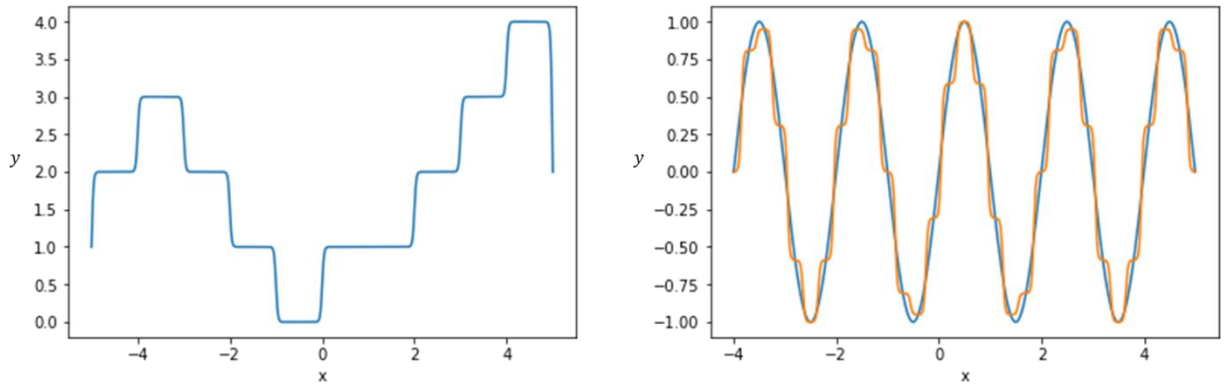


Figure 79: *Illustration that based on step function any function can be approximated.*

Extension to more general cases are straight forward. By increasing the number of input neurons, the function can be made to work on any input dimension n i.e., $f: \mathbb{R}^n \rightarrow \mathbb{R}^1, f(\mathbf{x}) = f(x_1, x_2, \dots x_n) = y$. The same can be done with respect to the output neurons to obtain the most general case i.e., $f: \mathbb{R}^{n_x} \rightarrow \mathbb{R}^{n_y}$. This is schematically illustrated in Figure 80 comparing the network structure for the one-dimensional case (left) with the general situation.

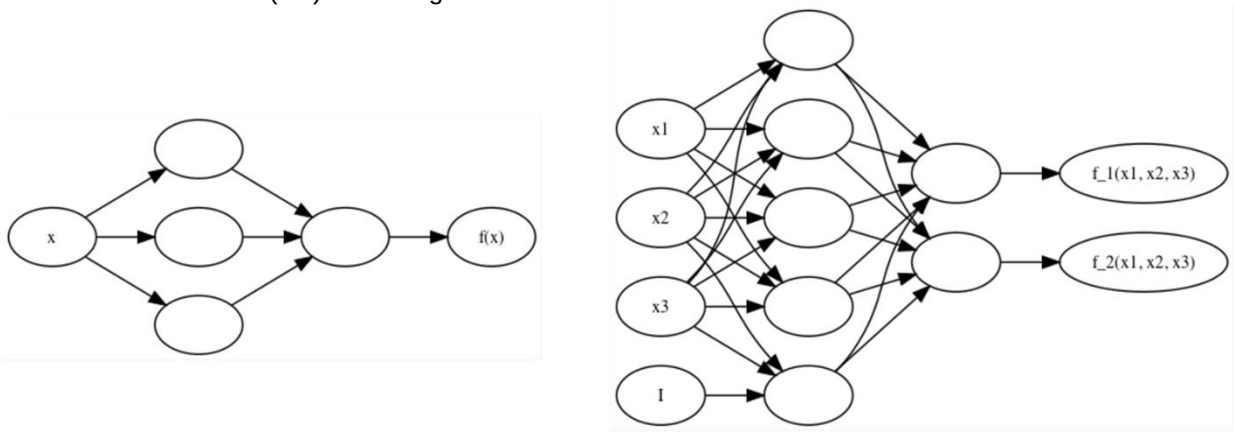


Figure 80: *A network for a function* $\boldsymbol{f}: \mathbb{R}^1 \rightarrow \mathbb{R}^1$ *(left) is easily extend to the general case* $: \mathbb{R}^3 \rightarrow \mathbb{R}^2$*(right).*

Figure 81 illustrates how this could be applied for function approximation $f: \mathbb{R}^2 \rightarrow \mathbb{R}^1$. In a first step 2D peaks are created by summing up 1D ridges oriented in different directions. The intensity in the centre is amplified in proportion to the number of ridges involved. Once a peak obtained the same strategy as illustrated in Figure 79 for 1D can be applied.
However, this strategy immediately raises the following questions:

- For improving the accuracy when modelling with step-functions, more and more neurons are needed, and many parameters need to be determined. In problems with high-dimensional input, an exponentially growing number of neurons is needed.
- Accordingly, more data sampled on a sufficiently fine grid is needed. With growing dimensionality in the input (e.g. image or audio data) this becomes infeasible. This is known as curse of dimensionality.
- If we just use the available data and interpolate with step-functions between data points, we significantly overfit on the training data.
- The theorem does not provide a scheme for efficiently learning the parameters from available

limited data.

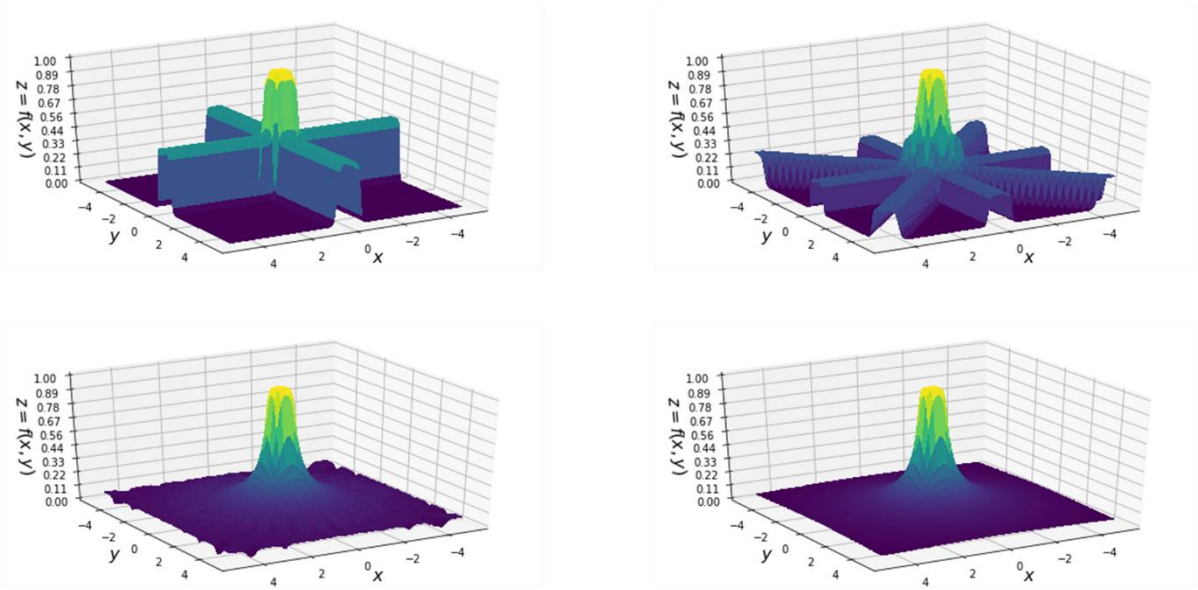


Figure 81: *Illustration of the creation of 2D peak functions with the help of 1D ridges.*

4 Bias and Variance of Model, Overfitting and Model Selection

The central challenge in machine learning is that our algorithm must perform well on new, previously unseen inputs – not just those on which our model was trained. The ability to perform well on previously unobserved inputs is called generalization [1]. This statement summarizes well the problem that we cannot simply infer from the performance of a model on the training data set on how well it will behave on previously unseen data. While we use the independent test data set to estimate this generalization performance, so far, we lack the formal framework to study the expected generalization error. In this chapter we will introduce the statistical concepts bias and variance of a model that will allow to understand the behaviour of the generalization error from a general point of view and that will naturally lead to the topic of under- and overfitting.

4.1 Bias and Variance

We first introduce the concepts of bias and variance using the formal language of statistics (which provides the natural framework for these terms) and then illustrate their behaviour using a toy model. As usual we assume that for our task (Figure 40) there exists a hypothetical mapping $f(\mathbf{x})$, which we do not know, and which we want to approximate with a model $h_\theta(\mathbf{x})$. We sample a training set of size m given by $D = \{(\mathbf{x}^{(i)}, y^{(i)}), i = 1..m\}$. Our choice for the model $h_\theta(\mathbf{x})$ (or model parameters θ) will obviously depend on the training set D, because sampling a second time will give a different set and lead to a different parameter choice for θ. We will indicate this dependency by the additional subscript D i.e., $h_{\theta,D}(\mathbf{x})$[57]. Therefore, we can consider $h_{\theta,D}(\mathbf{x})$ as a random variable depending on the data set D, which consists of m independent and identically distributed[58] data points provided by the mapping $f(\mathbf{x})$. Having introduced this notation of statistics we can now formulate bias and variance in a natural way:

- The bias of our model i.e., the estimator $h_{\theta,D}(\mathbf{x})$ for $f(\mathbf{x})$ is defined as[59]:

$$\text{bias}(h_\theta) = \mathbf{E}\left[h_{\theta,D}\right] - f$$

  Here $\mathbf{E}\left[h_{\theta,D}\right]$ means the expectation (or average) of $h_{\theta,D}(\mathbf{x})$ over the training sample set D. Thus, the bias measures how close the model is on the average to the true mapping.

- The variance of $h_{\theta,D}(\mathbf{x})$ is now nothing but the variance in the statistical sense, i.e., the expectation taken over the (squared) difference between the model and its expectation $\mathbf{E}\left[h_{\theta,D}\right]$:

$$\text{Var}(h_\theta) = \mathbf{E}\left[\left(h_{\theta,D} - \mathbf{E}\left[h_{\theta,D}\right]\right)^2\right]$$

  It measures how strong the model will fluctuation under variation of the training set D.

We will require these formulas to derive the bias variance trade-off later.

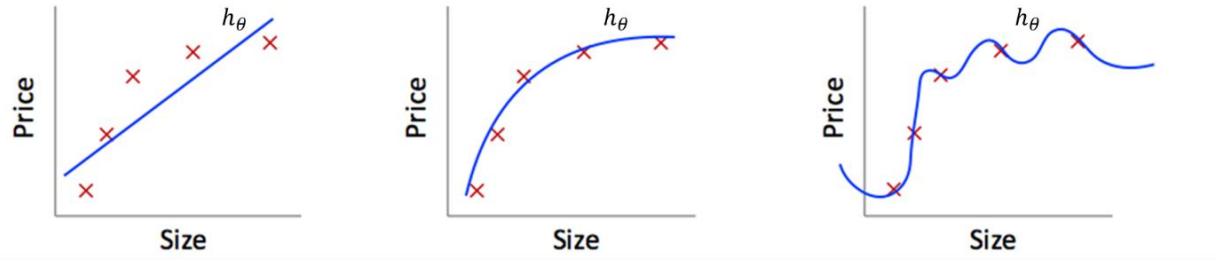


Figure 82: *Three models with different biases for a regression problem.*

[57] We will use this notation only in this section.
[58] This is frequently abbreviated as i.i.d.
[59] We skip the dependency on the input vector $\mathbf{x}$ for the sake of clarity.

The concept of varying bias is illustrated in Figure 82 and Figure 83 where we use a regression and a classification example. Figure 82 shows a simple regression problem where from left to right the capacity of the model function increases from a pure linear function, over a quadratic function to a polynomial of degree four. If the true mapping $f(\mathbf{x})$ corresponds to a smoothed version of the data points, we can qualitatively conclude from the drawings:

- left case:
  The low capacity of the linear model leads to a high bias i.e., to strong deviations from the true mapping $f(\mathbf{x})$. We will call this “underfitting”.

- Middle case:
  Increasing the capacity leads to a model h_θ that corresponds well to our assumption of the true mapping $f(\mathbf{x})$ and thus to a low bias. This seems to be a good compromise.

- Right case:
  Further increasing the capacity allows the model to fit all data points exactly however, this no longer corresponds to the underlying structure of the data i.e., the mapping $f(\mathbf{x})$. It is expected that changing (i.e. resampling) the data set will lead to a strong variation of the model h_θ i.e., to a high variance. We will call this “overfitting”.

Figure 83 illustrates the same idea but using a binary classification problem where the decision boundary of the model is represented. The findings just made can be applied here in an identical way.

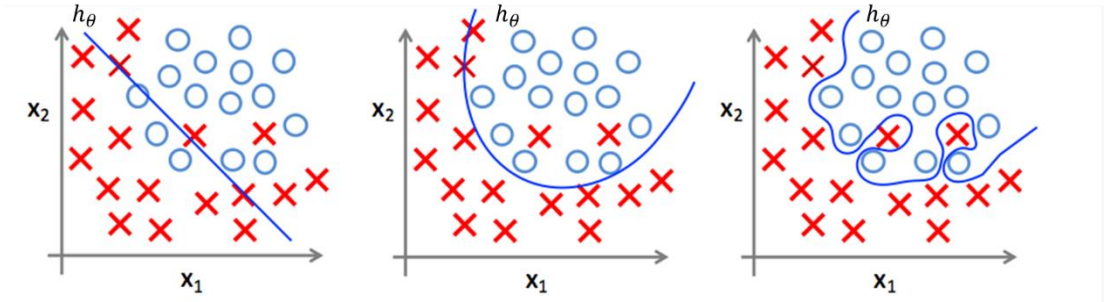


Figure 83: *Three models with different biases for a classification problem.*

While it is obvious in the two samples given above to choose the optimum case this is far from being trivial in the general case and we will develop strategies in the following to do so. As an intuitive approach the following principle of parsimony is frequently cited which is known as [60]Occam’s razor. It states:

> Among competing hypotheses that explain known observations equally well, we should choose the “simplest” one.

We will now illustrate the concepts of bias and variance introduce above using a simple toy model.

4.1.1 Illustrative Example

The Figure 84 shows our toy model, which is a simple regression problem. The true mapping $f(x)$ is shown on the left-hand side. On the right a set of $m = 30$ training data points was sampled, where some Gaussian random noise of standard deviation $\sigma = 0.1$ was added[61,62]:

$$D = \left\{\left(x^{(i)}, y^{(i)} = f\left(x^{(i)}\right) + \sigma \cdot \mathrm{N}(0,1)\right), i = 1..m\right\}$$

Equation 4

[60] https://en.wikipedia.org/wiki/William_of_Ockham
[61] $\mathrm{N}(0,1)$ is a Gaussian random variable with zero mean and unit variance.
[62] This represents noise introduced by the data generation processing (i.e., sensor noise) and will lead us later to the so-called irreducible error.

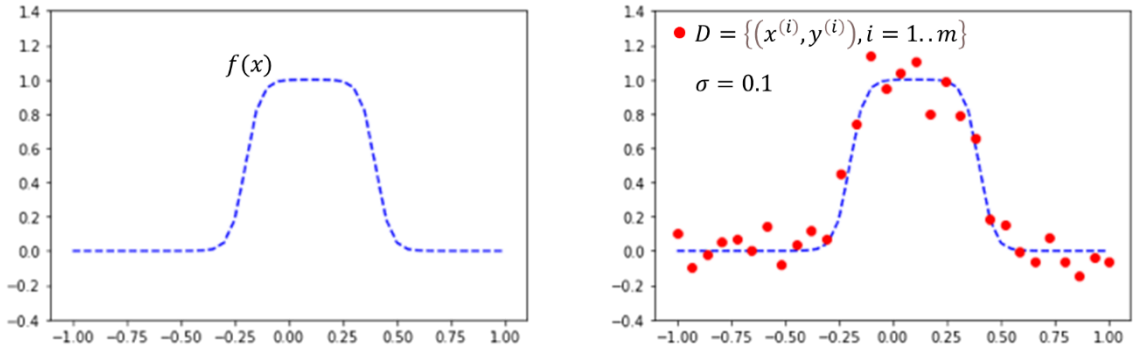


Figure 84: *Toy model used to illustrate the concepts of bias and variance.*

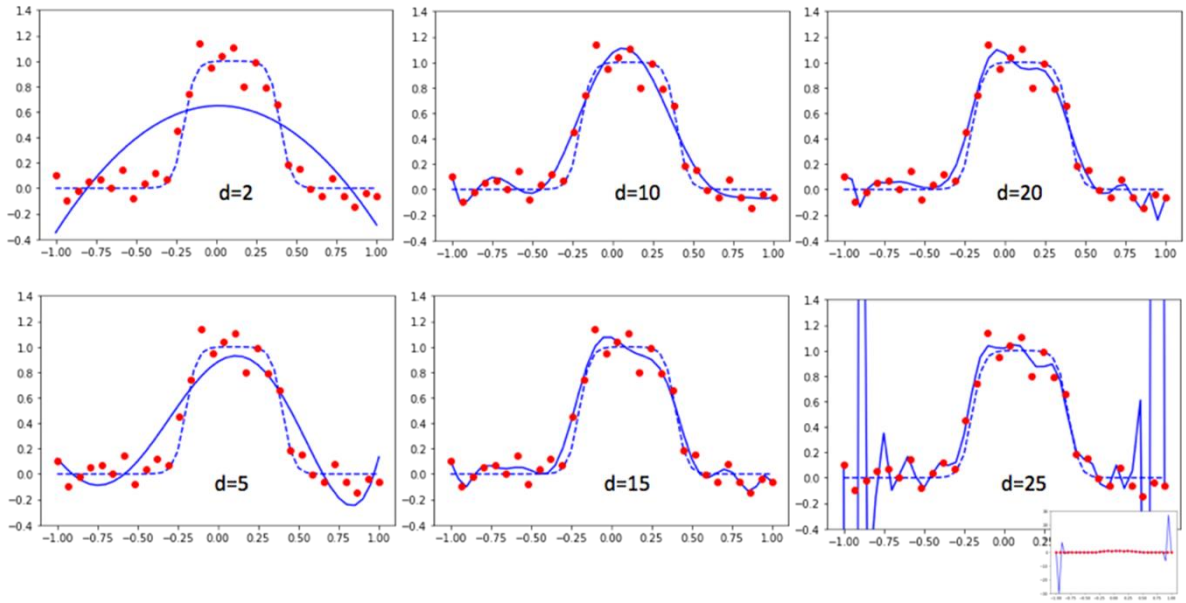


Figure 85: *Increasing the capacity of the model (polynomial degree **d**) will decrease its bias.*

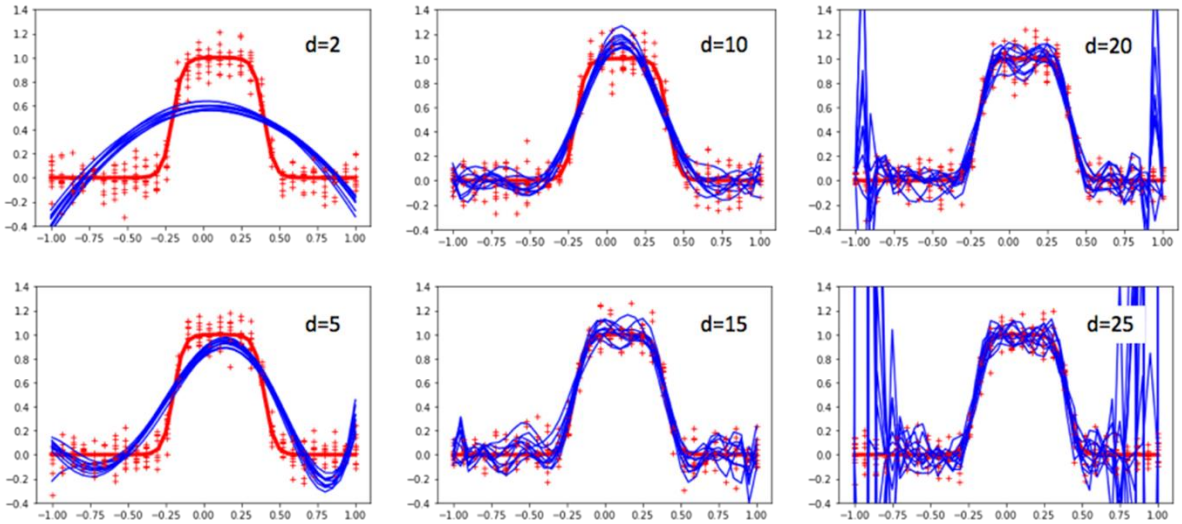


Figure 86: *Increasing the capacity of the model (polynomial degree **d**) will increase its variance.*

Our model will consist of a parametric family of polynomials with increasing degree d and thus increasing capacity:

$$h_\theta(\mathbf{x}) = \theta_0 + \theta_1 \cdot x + \theta_2 \cdot x^2 + .. + \theta_d \cdot x^d$$

Figure 85 shows a set of models obtained after optimizing the MSE cost for increasing polynomial degrees $d = 2,..,25$. It is obvious that the low-capacity models show a very high bias which decreases successively with increasing polynomial degrees d.
By resampling several training data sets for each capacity, we can now study the variance of the toy model (Figure 86). The quadratic model ($d = 2$) shows very few dependencies upon the change of the training data set, while as capacity increases these changes increase continuously, illustrating that the variance increases with the model capacity.
Figure 87 (left) shows the bias and variance of the different models, evaluated according to the formulas given at the beginning of this chapter 4.1, as a function of the model capacity d. It proves our qualitative observation that bias decreases and the variance increases with increasing capacity.

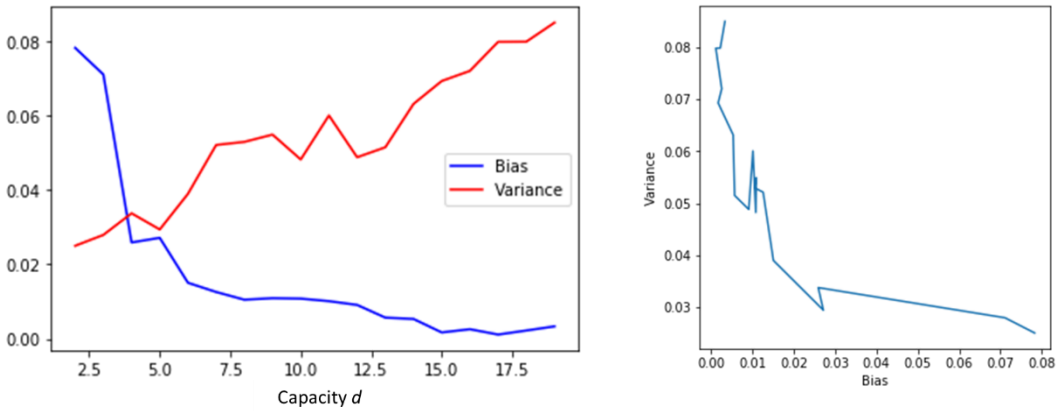


Figure 87: *Dependency of bias and variance as a function of the model capacity (polynomial degree* $\boldsymbol{d}$*).*

It turns out that this inter-dependency i.e., decreasing bias means increasing variance (right), is not a coincidence but results from a general theorem which we are going to prove now.

4.1.2 Bias-Variance Trade-off

We recall the formulas for the bias

$$\text{bias}(h_\theta) = \mathbf{E}\left[h_{\theta,D}\right] - f$$

and the variance

$$\text{Var}(h_\theta) = \mathbf{E}\left[\left(h_{\theta,D} - \mathbf{E}\left[h_{\theta,D}\right]\right)^2\right]$$

introduced at the beginning of this chapter 4.1, where the expectation is with respect to the training data set D.
We now formally introduce the concept of *generalization error* as the mean squared error (MSE) between the true mapping and the model:

$$\text{MSE} = \mathbf{E}\left[\left(h_{\theta,D} - f\right)^2\right]$$

For MSE cost function[63] this measures the expected deviation of the model $h_{\theta,D}$ from the true mapping f under variation of the data set D i.e., when applied to a new previously unseen data set.

We now perform the following rearrangements:

$$\mathbf{E}\left[\left(h_{\theta,D} - f\right)^2\right] = \mathbf{E}\left[\left(h_{\theta,D} - \mathbf{E}\left[h_{\theta,D}\right] + \mathbf{E}\left[h_{\theta,D}\right] - f\right)^2\right] =$$

[63] While these considerations are strictly true only for MSE cost (regression problems) they are usually applied also for to CE cost i.e., classification problems. Note nevertheless the explications in section 6.4.4.3 of [8].

$$\mathbf{E}\left[\left(h_{\theta,D} - \mathbf{E}[h_{\theta,D}]\right)^2\right] + \mathbf{E}\left[\left(\mathbf{E}[h_{\theta,D}] - f\right)^2\right] + 2 \cdot \underbrace{\mathbf{E}\left[\left(h_{\theta,D} - \mathbf{E}[h_{\theta,D}]\right) \cdot \left(\mathbf{E}[h_{\theta,D}] - f\right)\right]}_{0} =$$

$$\mathrm{Var}(h_\theta) + \mathrm{bias}(h_\theta)^2$$

Thus, we find a compact relationship known as:

Bias Variance Trade-off:

$$\mathrm{MSE} = \mathrm{bias}(h_\theta)^2 + \mathrm{Var}(h_\theta) + \sigma^2$$

We added (manually) the irreducible error variance σ^2, which corresponds to the sampling noise (c.f. Equation 4)[64]. This important relation states that the generalization or test error[65] is the sum of two terms the bias and the variance[66]. Because increasing the model capacity will decrease the bias but simultaneously increase variance, we must find the optimal capacity (trade-off) where the error is minimum (Figure 88). In the region with lower model capacity, where bias dominates the error, we speak of *underfitting*. Above the trade-off, where the error is mainly determined by the variance, we use the term *overfitting*.

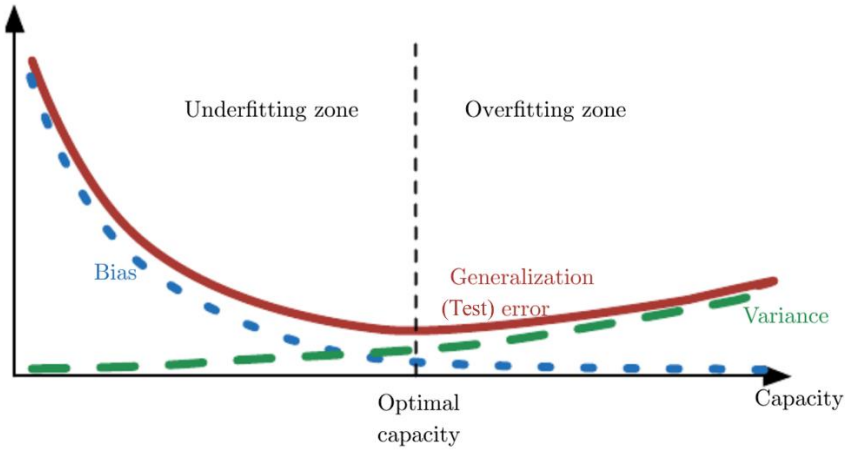


Figure 88: *The generalization or test error (if measured using MSE) is the sum of bias and variance.*

Analysing the MSE curves for the training and the test data set as function of the model capacity d of our toy model of chapter 4.1.1 also reveals the same behaviour (Figure 89). I.e., the test error shows a pronounced minimum in the centre region of the capacity corresponding to the optimal trade-off between bias and variance. The training error, which for low capacity follows closely the test error, nevertheless continues to decrease for increasing capacity.

Formally we can now define the problem of overfitting as follows:

Overfitting occurs when the learned hypothesis (trained model) fits the training data set very well - but fails to generalise to new examples.

Overfitting may occur if:

- the number of parameters of the model is too large i.e., the model capacity is too large.
- the training set is too small in comparison with the dimensionality of the input data (which introduces sampling noise).
- the training set is too noisy.

[64] The full derivation i.e., including σ can be found here: https://en.wikipedia.org/wiki/Bias%E2%80%93variance_tradeoff
[65] We use the independent test data set to estimate the generalization error.
[66] In addition, we have the irreducible error σ^2 but which can only be influenced by improving the data sampling process.

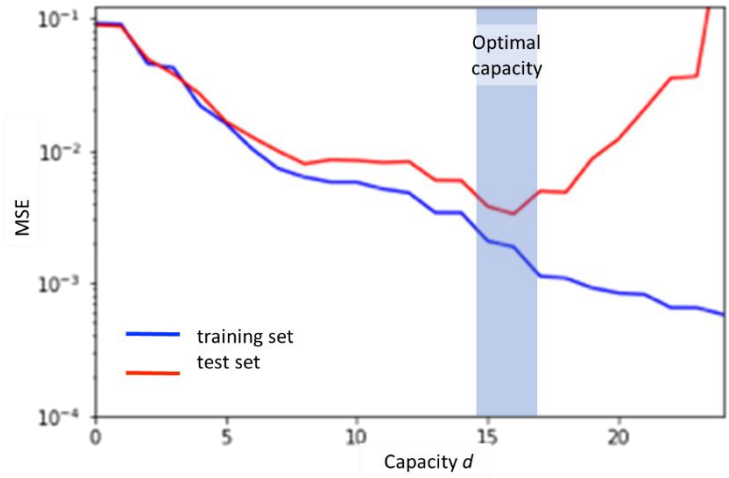


Figure 89: *The MSE curves for training and test set of our toy model (chapter 4.1.1) reveal the region of optimal capacity.*

In later chapters we will address the important topic of regularisation, which is meant to avoid overfitting i.e., to reduce the generalization error.

The analysis of the overfitting as shown in Figure 89 can be generalized as represented in Figure 90. Thus, the cost function (MSE or CE) or other performance measure can be analysed:

- for increasing model complexities
- for increasing training set sizes
- for increasing number of training epochs

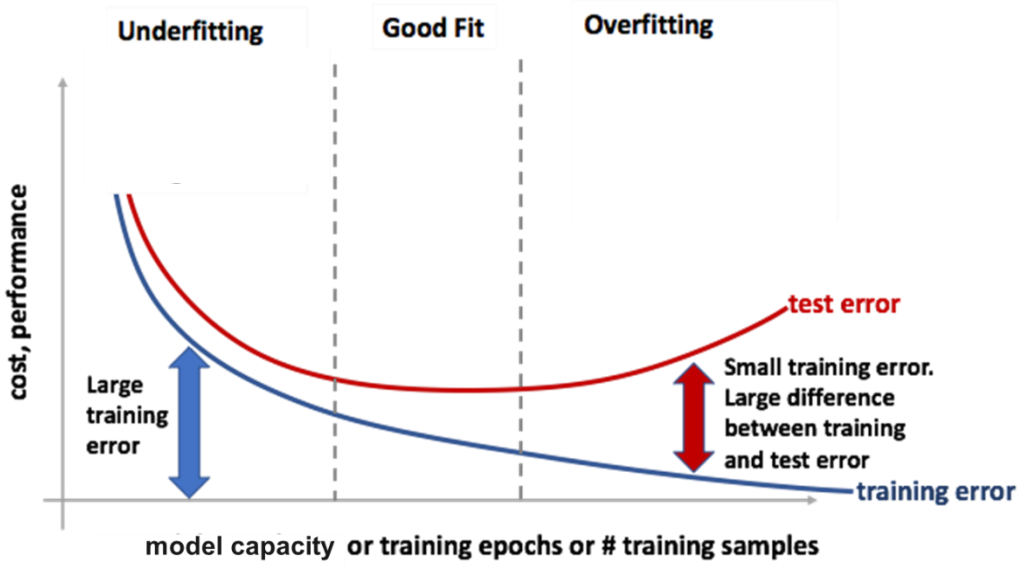


Figure 90: *General representation of the possibility to analyse overfitting.*

As illustrated in Figure 90 for optimal choice of model capacity we should be able to make

1. the training error (“bias error”) small.
2. the gap between training and test error (“variance error”) small.
3. the bias and variance error of comparable magnitude.

We will now present suitable strategies to selection the optimal model.

4.2 Model Selection Process

From the discussion in the previous paragraph in particular Figure 90 we learned that the minimization of the generalization error requires in principle to study the model performance on training *and* test data set simultaneously. However, we are not allowed to use any information from the test set during the training. To avoid this problem, we will further split the data into now *three* sets, as shown in the following Table 5. But before that we introduce the concept of hyperparameter:

> A hyperparameter is a parameter of the learning algorithm itself or a higher-level property of the model that *cannot* be not determined by optimising the cost on the training set.

Examples of hyperparameters are:

- Learning rate α
- Batch size b
- Activation function (Table 4)
- Number of hidden layers in a neural network
- Number of neurons in a layer of a neural network
- Degree d of the polynomial of our toy model (c.f. chapter 4.1.1)
- …etc. (more to come)

We can now extend the data sets by including the *validation* set which specifically allows to tune the hyperparameters of the ML model.

Training Set	Subset of trustable[1)] data used to train the model (choose the parameters that lead to a small cost). Should not be used for evaluating the model.	60%	98%
Validation Set	Subset of trustable data used to select models e.g., by selecting the best hyperparameters of the model.	20%	1%
Test Set	Subset of trustable data used to measure the performance of the finally selected model.	20%	1%

Table 5: *The available data set is split in three parts, training, validation, and test set.*

[1)]With trustable we mean:

- composed of data acquired under the same conditions,
- including the same characteristics (range of values, distribution, etc),
- sufficiently large to have confidence in the parameter estimates or evaluation metrics.

The typical split ratios given to the right are for small and large ($\gtrsim 10^6$) datasets (3rd and 4th column).

Based on these three data sets we can now formulate the standard process for Model Selection in its general form (Figure 91) which works as follows:

- We split the data according to Table 5 into training, validation, and test data. The test data set is hidden somewhere and not used till the final evaluation.
- The appropriate data normalization scheme (c.f. chapter 3.2.2) is applied.
- We chose our model h_θ that depends on a set of model parameters θ and additional hyperparameters.
- The model parameters θ are optimized using the training data set and an appropriate cost function (MSE or CE).
- The obtained model is evaluated using the validation set (e.g., validation cost is studied similar to Figure 90).
- If model improvements seem possible the hyperparameters (e.g., learning rate α) are adjusted and a new training iteration is performed.
- Once the hyperparameter tuning finalized and no further improvement observed, the final performance of the model on the test set is evaluated as estimate for the generalization error.

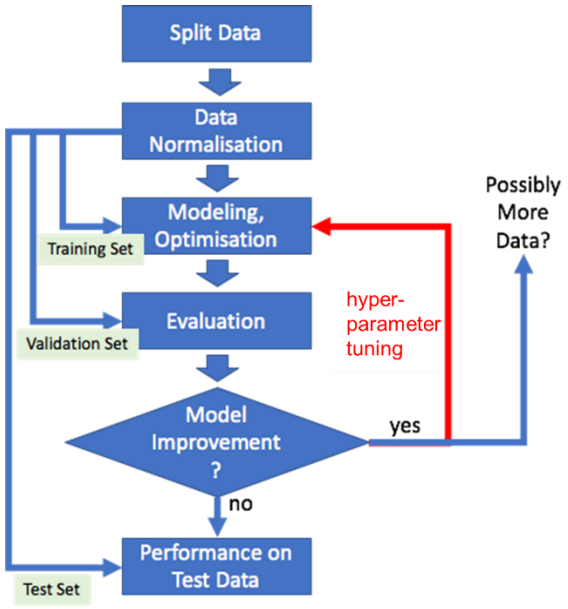


Figure 91: *Standard process for Model Selection process with an iterative improvement of the model using the validation set and final estimation of generalization error based on the test set.*

A further refinement of the model selection procedure that makes efficient use of the data for a more robust estimation of the validation set performance is the so-called k-fold Cross-Validation which we are going to present now.

4.2.1 k-fold Cross-Validation

The idea of cross validation is to determine the performance of the trained model not on a single but on k different validation sets. Therefore, the training set is split in k so-called folds. In Figure 92 this is illustrated for $k = 5$. Then for each fold the following procedure is applied:

- Use the selected fold as validation set.
- Train the model on the remaining $k - 1$ folds.
- Determine the performance of the model on the selected validation fold.

As overall model performance the average over the k folds are reported. In addition, the variance of the performance can be determine giving an idea on how confident the values are. The model leading to the best performance, among the k validation folds possible, is retained.

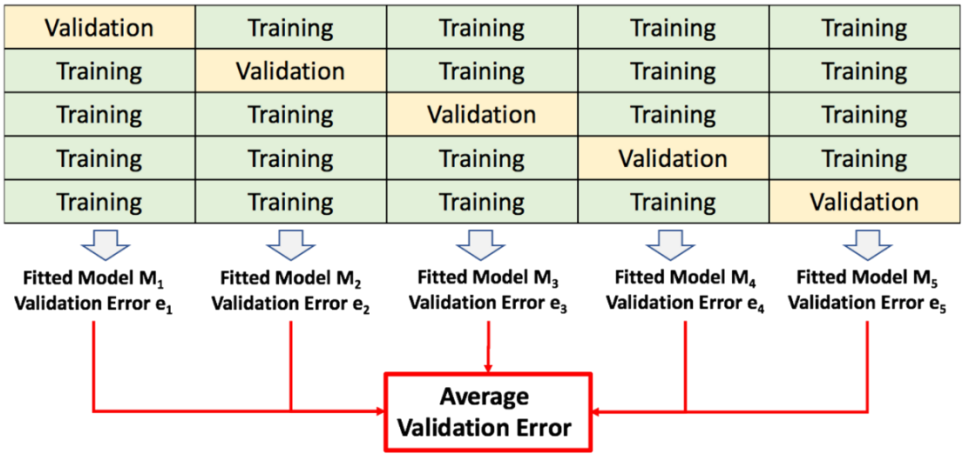


Figure 92: *The idea of cross validation using* $\boldsymbol{k = 5}$ *folds.*

An example of 5-fold cross validation applied to a k-nearest neighbour classifier[67] trained on cifar-10 dataset[68] is shown in Figure 93. The hyperparameter, which is tuned and given on the x-axis, represents the number of neighbours of the classifier. As just discussed, we can not only give the validation performance but also an estimation of its variance. This corresponds to the formal definition of the model variance as given in chapter 4.1.

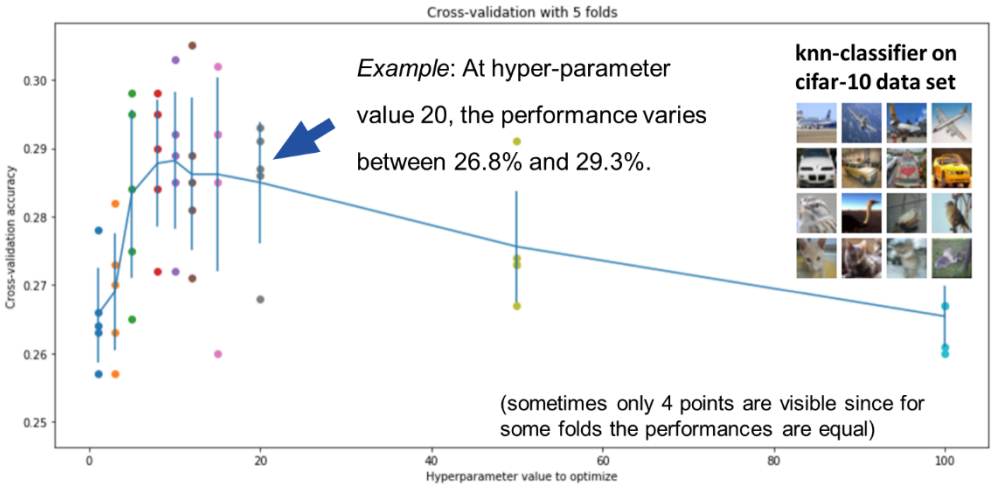


Figure 93: *Example of 5-fold cross-validation using a k-nearest neighbour classifier on cifar-10 dataset[69].*

4.2.2 Selecting a Split Ratio

In Table 5 we gave typical split ratios for training and validation sets. It may be instructive to consider different split ratios and study the behaviour of the corresponding cost functions. If we increase the split ratio with a value of zero corresponding to *no* training data, we expect:

- the training cost (or training error i.e., fraction of misclassified samples) to be increasing in the split ratio. This is because with increasing training data size it becomes more difficult to fit a model.

- the validation cost (or validation error) to be decreasing in the split ratio (the validation error becomes very wiggly for large split ratios). This is because the model trained with more data is expected to capture more details about the underlying problem.

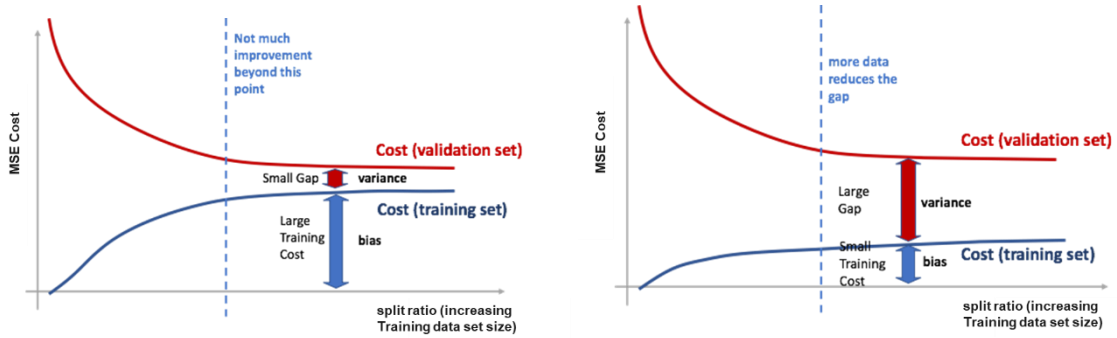


Figure 94: *Behaviour of training and validation cost as a function of the split ratio.*

This is illustrated for two cases in Figure 94[70]. In addition to the behaviour discussed above (training cost in- and validation cost de-creasing) two aspects are shown. To left, the case of underfitting is illustrated characterized by a high bias and low variance. To the right the inverse case with low bias but

[67] I.e., a sample is assigned the class which corresponds to the most frequent class of its k nearest neighbors.
[68] https://www.cs.toronto.edu/~kriz/cifar.html
[69] http://cs231n.stanford.edu/slides/2016/winter1516_lecture2.pdf
[70] C.f. also a similar discussion in [1], Figure 5.4, p.114.

hight variance i.e., the overfitting case is represented. As for Figure 90 this shows that the detailed observation and study of such *learning curves* provides precious information on the status of the model training process. We will now introduce further performance measures allowing to for even more learning curves.

5 Performance Measures

5.1 Confusion Matrix

When working with classification tasks it may be important to present the results of correctly and misclassified samples in well-arranged form. Therefore, a confusion matrix is used:

> A confusion matrix measures the test performance of a classification system on a per-class basis by indicating the number of samples of actual class **a** predicted as class **b**. The rows relate to the actual class labels **a**, and the columns to the predicted class labels **b**.

E.g., in the example given in Figure 95, the orange box indicates that 63 samples of total 221 of actual class **b** (8+131+63+19) are predicted as class **c**.

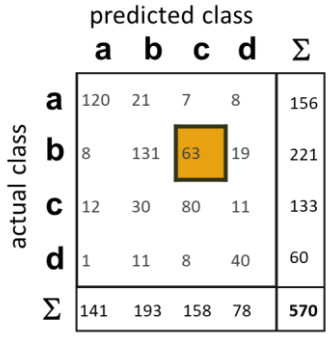


Figure 95: *Illustration of a confusion matrix which presents the number of the actual class labels versus the predicted ones.*

Figure 96 shows a further example from the MNIST dataset. Due to the well-arranged presentation, we can quickly identify particularly bad results which are the 60 digits '9' classified as '4' or the 65 digits '5' predicted as '8'.

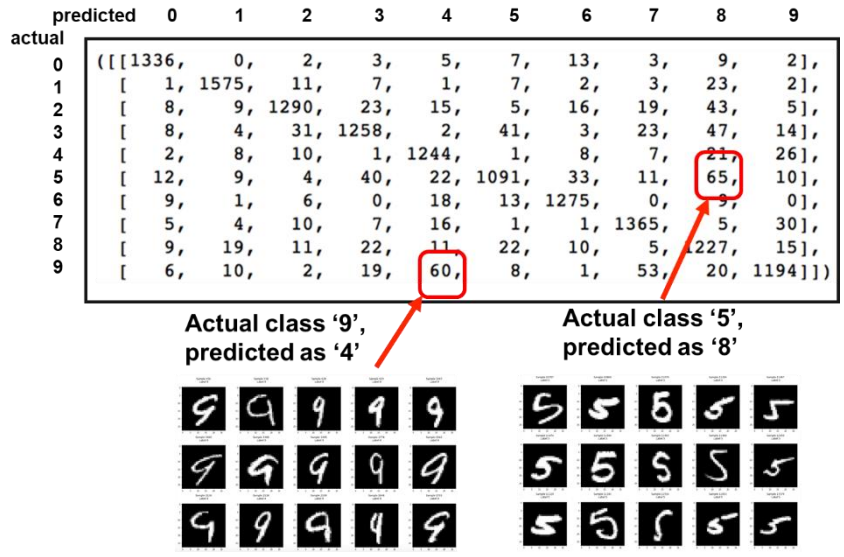


Figure 96: *Confusion matrix for MNIST dataset (details c.f. text).*

Based on the confusion matrix different performance measures can be obtained.

Accuracy:
The accuracy is simply the fraction of all correct predictions. From the confusion matrix it is obtained by summing up all diagonal elements and dividing by the number of all samples. In the example for MNIST given above we find a total of 12'855 correct classification with respect to a total of 14'000

samples in the test data set giving an accuracy of 91.8%.

$$accuray = \frac{\sum diagonal\ elements}{\#samples}$$

Error Rate:
The complement of the accuracy i.e., the fraction of misclassified samples is the error rate:

$$error\ rate = 1 - accuray$$

Important:
The accuracy may give a biased view of the performance in case of strongly un-balanced classes. If e.g., a single class represents already 95% of the data an accuracy of 95.5% would be very poor.

5.2 Confusion Table

When establishing a confusion matrix for a binary classification problem we use the term confusion table.

A confusion table is used to measure the classification performance of a two-class system

		Predicted		
		Positive	Negative	*Total*
Actual	Positive	**TP**	FN	(TP+FN)
	Negative	FP	**TN**	(FP+TN)
	Total	(TP+FP)	(FN+TN)	N

Figure 97: *Illustration of a confusion table i.e., a confusion matrix for a binary classification problem.*

Figure 97 illustrates an example. Because this concept is often used in the context of hypothesis testing the terms “Positve” and “Negative” classes and outcomes are frequently used. But the classes may as well represent the digits ‘1’ and ‘7’ of our MNIST data set. According to this notation one defines:

True Positives (TP)	Number or fraction of positive samples classified as such
True Negatives (TN)	Number or fraction of negative samples classified as such
False Positives (FP)	Number or fraction of negative samples classified as positive.
False Negatives (FN)	Number or fraction of positive samples classified as negative.

To develop the different performance measures, we will transform the confusion matrix from Figure 96 to a confusion table by considering one digit against all others. This will lead us in addition naturally to the “class” performance measures.
In Figure 98 we obtained a confusion table by considering the digit ‘5’ against all other digits. We see that this represents an example for a strongly unbalanced datasets because the ‘Negatives’ classes

represent 12’598 out of 14’000 i.e., 90%.

Example Digit ‘5’		Predicted		
		P	N	Total
Actual	P	1091	206	1297
	N	105	12598	12703
	Total	1196	12804	14000

Figure 98: *Confusion table obtained from Figure 96 by considering digit ‘5’ against all others.*

We now define the following “class” performance measures because we always consider the digit ‘5’ with respect to all other classes.

Per Class Accuracy:
The class accuracy is in analogy to above the fraction of correctly classified samples. For digit ‘5’ we obtain:
(1’091 + 12’598) / 14’000 = 97.8%.

$$class\ accuray = \frac{TP + TN}{N}$$

Example Digit ‘5’		Predicted		
		P	N	Total
Actual	P	1091	206	1297
	N	105	12598	12703
	Total	1196	12804	14000

Per Class Sensitivity (or Recall):
The class sensitivity is the ratio of correctly classified samples with respect to all positive samples. It tells us what fraction of all positive samples is recognised by the system.
For digit ‘5’ we obtain:
1091 / (1091 + 206) = 84.1%

$$class\ sensitivity = \frac{TP}{TP + FN}$$

Example Digit ‘5’		Predicted		
		P	N	Total
Actual	P	1091	206	1297
	N	105	12598	12703
	Total	1196	12804	14000

Per Class Precision:
The class precision is the ratio of correctly classified samples with respect to all positive predictions. It tells us what fraction of all positively classified samples is correctly classified.
For digit ‘5’ we obtain:
1091 / (1091 + 105) = 91.2%

$$class\ precision = \frac{TP}{TP + FP}$$

Example Digit ‘5’		Predicted		
		P	N	Total
Actual	P	1091	206	1297
	N	105	12598	12703
	Total	1196	12804	14000

As a trade-off between recall and precision serves the so-called

F-Score or F1-Score:
The F- or F1-Score is the harmonic mean between precision and recall.
For digit ‘5’ we obtain:
1091 / (1091+ (206+105)/2) = 87.5%

Example Digit ‘5’		Predicted		
		P	N	Total
Actual	P	1091	206	1297
	N	105	12598	12703
	Total	1196	12804	14000

$$F1\_score = \frac{2}{\frac{1}{precision} + \frac{1}{recall}} = \frac{TP}{TP + \frac{FP + NP}{2}}$$

It is now possible to obtain from all per class performance measures the so-called system performance measure by taking the average over all class measures. For the system F1-score the harmonic mean of system precision and system recall is used.

Exercise:
In groups of two discuss for what problems a high precision or high recall may be desirable.

6 Multi-Layer Perceptron

Based on the work done in the previous chapters we can now generalize our consideration on multi-layer architectures. Deep feedforward networks, also called feedforward neural networks, or multi-layer Perceptrons, are the quintessential deep learning models. These models are called *feedforward* because information flows through the function being evaluated from the input vector $\mathbf{x}$ through the intermediate computations used to define the output $\mathbf{y} = h_\theta(\mathbf{x})$. There are no feedback connections in which outputs of the model are fed back into itself. When feedforward neural networks are extended to include feedback connections, they are called *recurrent* neural which are discussed in the second part of the lecture.
We already studied the performance of an MLP using one single hidden layer on the MINST dataset (Figure 75) and the general representational capacity of such an architecture in the context of the Universal Approximation Theorem (chapter 3.9). Here we will generalize the formulation of the gradient decent update rule to MLP which will bring us to the so-called backpropagation algorithm. But before that, we will discuss a common problem of all Machine Learning Algorithms, the so-called *Curse of Dimensionality*.

6.1 Curse of Dimensionality

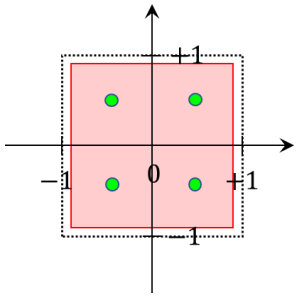
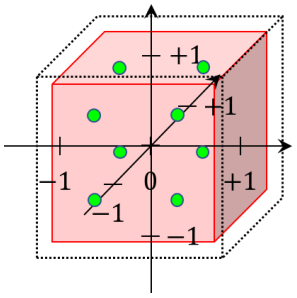
When we train a neural network, we want to approximate an unknown mapping $\mathbf{y} = f(\mathbf{x})$ based on our model $h_\theta(\mathbf{x})$ by an appropriate choice of parameters. This can be either a regression or a classification problem (Figure 82, Figure 83). Because this approximation is done based on a finite training data set only, we implicitly work with the so-called *Local Smoothness Assumption in Classical ML* and assume:

- The function to approximate does not vary much locally and stays roughly constant in small regions.
- This allows to generalise to variations in small regions from closely located training examples. A strategy adopted e.g., in clustering (k-means), k-nearest neighbours, decision trees, etc. and shallow neural nets.
- This implies that training examples are needed in all the regions of interest. If no or only few examples are available in a certain region, no (confident) predictions can be made.

From many successful applications we know that this assumption works fine if the dimensionality of the data is not too high. However, it fails in typical DL applications when dealing with image or speech/language data. The reason for that is the so-called Curse of Dimensionality. This expression was coined by Richard E. Bellmann[71]:

> "… when the dimensionality increases, the volume of the space increases so fast that the available data become sparse. This sparsity is problematic for any method that requires statistical significance. In order to obtain a statistically sound and reliable result, the amount of data needed to support the result often grows exponentially with the dimensionality."

We will illustrate this important point using the MNIST data set as an example in the following Table 6.

$d = 1$	$d = 2$	$d = 3$	$d = 784$
			

[71] https://en.wikipedia.org/wiki/Curse_of_dimensionality

$N = 2$	$N = 4$	$N = 8$	$N = 2^{784} \approx 10^{236}$
$\frac{2}{n}$	$\frac{2}{\sqrt{n}}$	$\frac{2}{\sqrt[3]{n}}$	$\frac{2}{\sqrt[784]{n}} \approx^{1} 1.97$ [1]for $n = 70'000$
0.99	0.99^2	0.99^3	$0.99^{784} \approx 0.0004$

Table 6: *Illustration of the effect of increasing dimension on different parameters (details see text).*

The first line shows the dimension of the respective features space we consider, starting from $d = 1$ till $d = 784$ being the dimension of the MNIST images (28x28). We will always consider the unit "cube" under the assumption of min-max-normalization of each input feature (second line, black dotted "cubes"). To apply the local smoothness assumption, we will try to place two training samples per dimension (green dots) which gives a total of 2^d for a dimension of d. We see immediately that for the MNIST feature space $d = 784$ this would require a total of 2^{784} data points which outnumbers by far the number of estimated atoms in the entire universe ($\approx 10^{80}$). If we cannot provide such high number, how "far" do we get with our available training set of $n = 70'000$ images? If we calculate the average distance between n equally distributed points in the unit cube we obtain $2/\sqrt[d]{n}$ with d being the dimension. If we determine the average distance of the $n = 70'000$ training sample in the unit cube of the $d = 784$ dimensional features space, we obtain a value of 1.97. Thus, the points are essentially all at the border of the unit cube at the maximum possible distance with respect to each other. Finally, we take for each linear dimension 99% of the centre part and determine the corresponding volume (red cube) which decreases with 0.99^d with respect to the volume of the unit cube. For the MNIST features space and corresponding dimensionality $d = 784$ we obtain only 0.0004 of the original volume in other words barely anything.
These considerations show that our intuition which was shaped in three dimensions is completely wrong in high dimension. Furthermore, we see, that even a comparatively large dataset (Figure 21) of 70'000 images for MNIST is vanishing small in a $d = 784$ dimensional features space and will in no way allow to apply the local smoothness assumption.

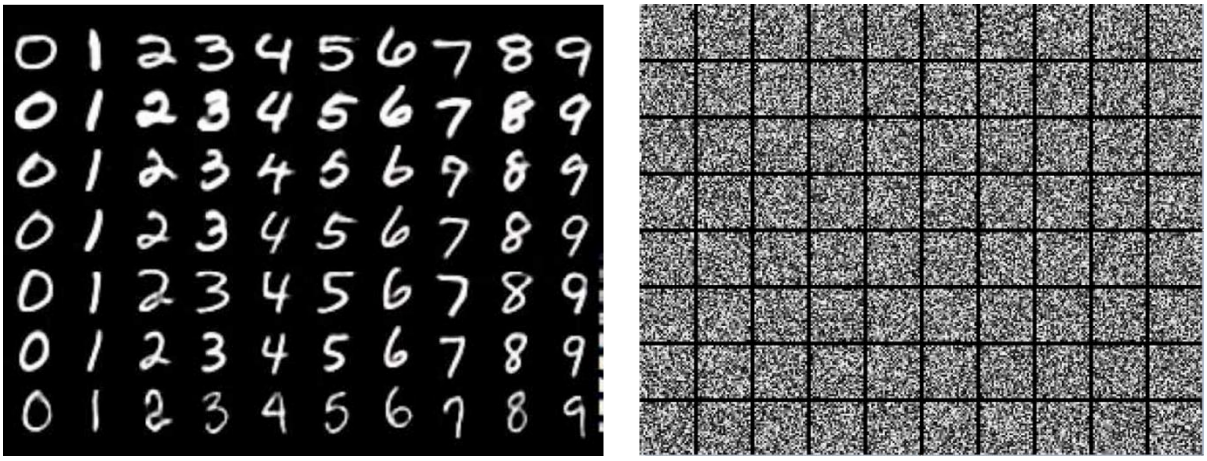


Figure 99: *MNIST images compared to a set of randomly sampled images in a 28x28 grid.*

But how can we then expect to make any reasonable prediction at all? The reason why this is nevertheless possible is the following. Consider the number of all possible 8-bit grayscale images that can be created on a 28x28 grid. For each pixel we have 256 possible values, and each pixel's grey value can be chosen independently giving a total of $256^{784} \approx 10^{1888}$ possible images! Even if the total humanity $\approx 10^{10}$ did nothing else but drawing numbers at a rate of one per second for a hole year this only gave a total of approximately $86400 \cdot 365 \cdot 10^{10} \approx 3 \cdot 10^{17}$ – a number still completely negligible with respect to all possible images. This means however, that relevant images only occupy a much smaller subspace of the full available feature space and the vast majority of all the possible images just represent noise without any semantic information (Figure 99, right). This situation is typically illustrated as in Figure 100, where a one-dimensional curve is imbedded into the two-dimensional space.

Thus, we consider, that the interesting information is concentrated on some lower dimensional manifold with much less degrees of freedom. The Local smoothness assumption may still hold on the submanifold, but not in the original input space.

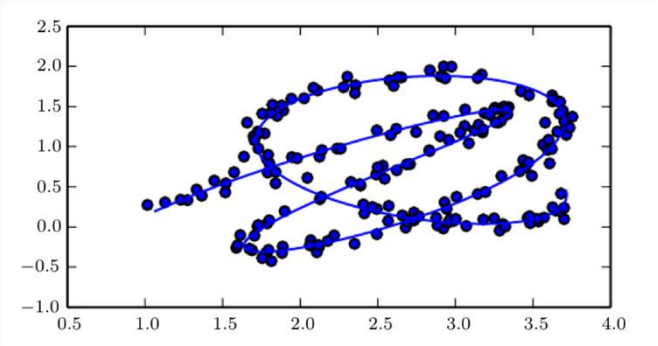


Figure 100: *Embedding of a one-dimensional curve into the two-dimensional space.*

In addition, we assume that a composition of features at multiple levels in a hierarchy is possible leading to the following core idea of DL:

- We assume that the data was generated by the composition of factors or features, potentially at multiple levels in a hierarchy.
- Learning then involves discovering a set of underlying factors of variation that can be described in terms of other, simpler underlying factors (arranged in a hierarchy).

One prominent example cited frequently in that context are convolutional neural networks (Figure 101). It can be shown that CNNs detect features of different complexity at different levels i.e., layers. Thus, with increasing layer index increasingly complex semantic information is extracted from the images[72]. Two points should be noted:

- Due to this hierarchy of features/concepts and the distributed representations an exponential gain in the number of examples needed for a given number of regions to be distinguished is obtained.
- In addition, better generalisation properties for a wide variety of tasks (as compared with local smoothness assumption) can be expected.

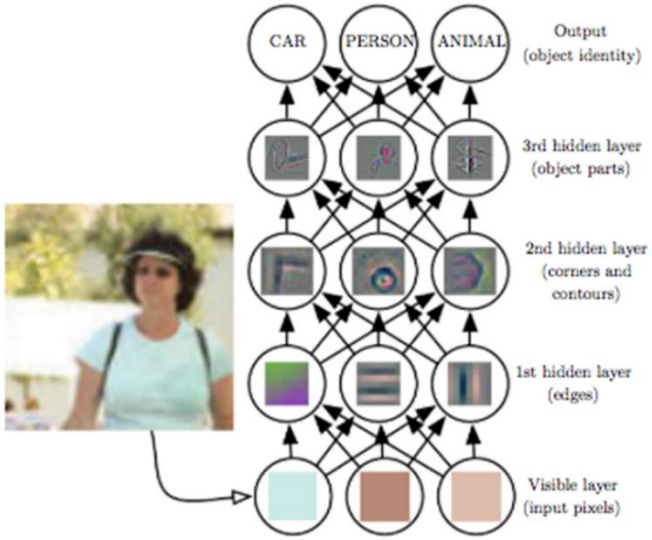


Figure 101: *CNN extract features at different levels of hierarchy.*

[72] C.f. also the discussion on biological NNs in chapter 2.1, in particular Figure 28.

6.2 Computational Graph

We will now address the problem of formalizing the architecture of MLP to automize the forward and backward pass for the data processing. Therefore, we define the following:

A computational graph is a directed graph where:
- *nodes* correspond to operations or input variables
- *edges* correspond to inputs of an operation which can originate from input variables or outputs of other operations.

Two types of input variables are possible: Input data and model parameters.

Our goal is to represent the MLP as a computational graph. A single layer could be represented as follows (c.f. Figure 102):

$$g(\mathbf{x};\mathbf{W};\mathbf{b}) = \sigma(\mathbf{W}\cdot\mathbf{x}+\mathbf{b})$$

The graph is denoted by g which depends on the input vector $\mathbf{x}$ and the parameters $\mathbf{W}$ and $\mathbf{b}$. We have three input nodes ("Input", 2 x "Var" in Figure 102) corresponding to the following variables with given dimensions[73]:

$$\mathbf{x}: \; n_x \text{ x } 1$$
$$\mathbf{W}: \; n_1 \text{ x } n_x$$
$$\mathbf{b}: \; n_1 \text{ x } 1$$

Furthermore, we have three nodes representing operations, which are from left to right:
- Matrix multiplication of $\mathbf{W}$ with $\mathbf{x}$.
- Vector addition of $\mathbf{W}\cdot\mathbf{x}$ and $\mathbf{b}$.
- Element-wise application of the sigmoid function $\sigma(..)$.

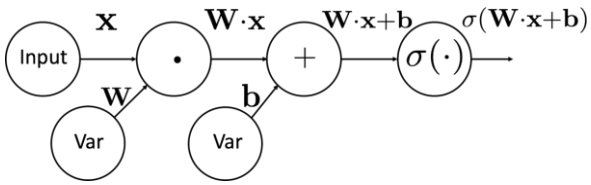


Figure 102: *A single MLP layer represented as a computational graph.*

Starting from this representation of a single MLP layer as a computational graph, we will now step by step increase the complexity to the most general result for a full MLP with a batch input $\mathbf{X}$ of size $n_x \text{ x } m$.
Starting with the next higher complexity of a two-layer architecture we obtain:

$$g\left(\mathbf{x};\mathbf{W}^{[1]};\mathbf{b}^{[1]};\mathbf{W}^{[2]};\mathbf{b}^{[2]}\right) = \sigma\left(\mathbf{W}^{[2]}\cdot\sigma\left(\mathbf{W}^{[1]}\cdot\mathbf{x}+\mathbf{b}^{[1]}\right)+\mathbf{b}^{[2]}\right)$$

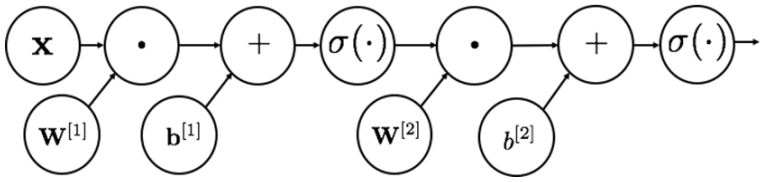


Figure 103: *Two-layer MLP network represented as graph.*

For the respective dimensions we have:

$$\mathbf{x}: \; n_x \text{ x } 1$$
$$\mathbf{W}^{[1]}: \; n_1 \text{ x } n_x$$

[73] We will eventually formulate all update rules in matrix-notation, so it is essential to be familiar with matrix calculus.

$$\mathbf{b}^{[1]}: \; n_1 \text{ x } 1$$
$$\mathbf{W}^{[2]}: \; n_2 \text{ x } n_1$$
$$\mathbf{b}^{[2]}: \; n_2 \text{ x } 1$$

We will however choose a more compact notation convention by collapsing the affine transformation and the application of the activation function into a single node:

$$\mathbf{a}^{[l]} = \sigma\left(\mathbf{W}^{[l]} \cdot \mathbf{x} + \mathbf{b}^{[l]}\right)$$

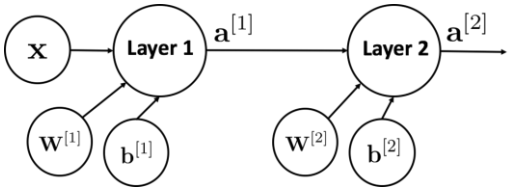


Figure 104: *Compact notation with a single node for the operation* $\mathbf{a}^{[l]} = \boldsymbol{\sigma}\left(\mathbf{W}^{[l]} \cdot \mathbf{x} + \mathbf{b}^{[l]}\right)$.

This type of layer is the abstraction level of most DL frameworks used to define the architecture of a DL model. An example for the tensorflow 2.0/keras API is given below:

```
import tensorflow as tf
mnist = tf.keras.datasets.mnist

(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(512, activation=tf.nn.relu),
  tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
```

Figure 105: *Definition of two layers of type "Dense" (i.e., fully connected) using the DL framework tensorflow/keras.*

It is now straightforward to represent a multi-layer Perceptron with arbitrary number of hidden layers as a computational graph (Figure 106). The input vectors $\mathbf{x}^{(i)}$ have the size $n_x \text{ x } 1$. Each hidden layer n_l $(1 \leq l \leq L-1)$ is characterized by the weight matrix $\mathbf{W}^{[l]}$ and bias vector $\mathbf{b}^{[l]}$. Each layer receives as input the activations from the previous layer $\mathbf{a}^{[l-1]}$ (being $\mathbf{x}^{(i)}$ for the first hidden layer). Each weight matrix $\mathbf{W}^{[l]}$ has the dimension $n_l \text{ x } n_{l-1}$, where n_{l-1} is the size of the activations from the previous layer $\mathbf{a}^{[l-1]}$ and n_l corresponds to the number of hidden neurons of layer l, which corresponds to the dimension of the output $\mathbf{a}^{[l]}$.
For a single layer of an MLP the notation, including the detailed vector and matrix indices, is shown in Figure 107. The dimensions are as follows:

$$\mathbf{a}^{[l-1]}: \; n_{l-1} \text{ x } 1$$
$$\mathbf{W}^{[l]}: \; n_l \text{ x } n_{l-1}$$
$$\mathbf{b}^{[l]}: \; n_l \text{ x } 1$$
$$\mathbf{z}^{[l]}: \; n_l \text{ x } 1$$
$$\mathbf{a}^{[l]}: \; n_l \text{ x } 1$$

It is now convenient to write the operation in compact vector/matrix notation:

$$\mathbf{z}^{[l]} = \mathbf{W}^{[l]} \cdot \mathbf{a}^{[l-1]} + \mathbf{b}^{[l]}$$
$$\mathbf{a}^{[l]} = g^{[l]}\left(\mathbf{z}^{[l]}\right)$$

Equation 5

Here $\mathbf{z}^{[l]}$ represents the so-called logit i.e., the input to the activation function, which is denoted by $g^{[l]}$. To illustrate the matrix-wise multiplication a concrete example is given in Figure 108 including the corresponding Python code snippet.

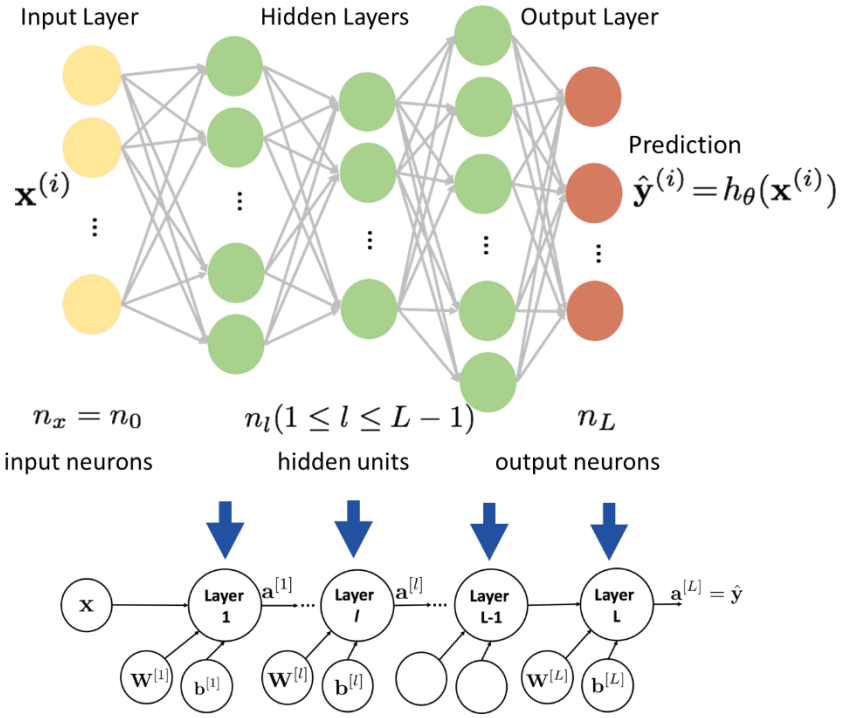


Figure 106: *Representation of a MLP as a computational graph.*

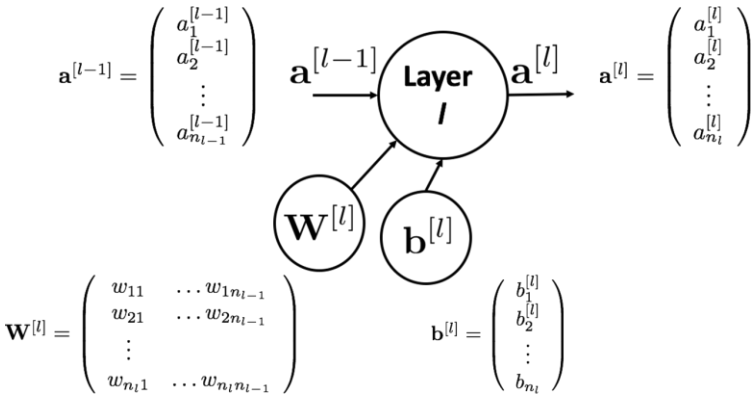


Figure 107: *Notation for single layer in MLP with indices explicitly given.*

$$\mathbf{a}^{[l-1]} = \begin{pmatrix} 1 \\ -1 \\ -0.5 \end{pmatrix} \qquad \mathbf{W}^{[l]} = \begin{pmatrix} 1 & -1 & -0.5 \\ 1 & 0.5 & -1 \\ -0.2 & 0.3 & 0.5 \\ -1 & 1 & 1 \end{pmatrix} \qquad \mathbf{b}^{[l]} = \begin{pmatrix} -1 \\ -1 \\ 1 \\ 1 \end{pmatrix}$$

$$\mathbf{z}^{[l]} = \mathbf{W}^{[l]} \cdot \mathbf{a}^{[l-1]} + \mathbf{b}^{[l]} = \begin{pmatrix} 1.25 \\ 0. \\ 0.25 \\ -1.5 \end{pmatrix} \qquad \mathbf{a}^{[l]} = \sigma(\mathbf{z}^{[l]}) = \begin{pmatrix} 0.78 \\ 0.50 \\ 0.56 \\ 0.18 \end{pmatrix}$$

```
import numpy as np
W = np.array([[1, -1, -0.5],[1, 0.5, -1], [-0.2, 0.3, 0.5],[-1, 1, 1]]).reshape(4,3)
b = np.array([-1, -1, 1, 1]).reshape(4,1)
aprev = np.array([1, -1, -0.5]).reshape(3,1)
z = np.matmul(W,aprev)+b
a = 1./(1.0+np.exp(-z))
```

Figure 108: *Example calculation for one layer to illustrate the correct matrix-wise multiplication.*

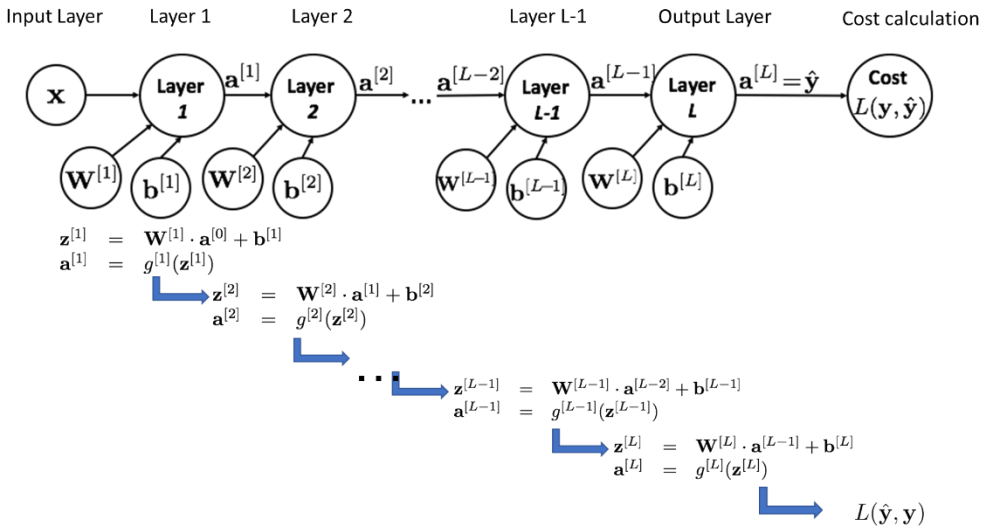


Figure 109: *Forward propagation for an MLP with arbitrary number of hidden layers.*

If one now strings together layer by layer according to the scheme shown in Figure 107 one obtains a full MLP with arbitrary number of hidden layers (Figure 109). This represents in principle the most general case. However, the forward (and the backward step coming next) is usually performed simultaneously on a full mini batch of m input vectors:

$$\mathbf{X} = \mathbf{A}^{[0]} = \begin{pmatrix} \vdots & \vdots & & \vdots \\ \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \cdots & \mathbf{x}^{(m)} \\ \vdots & \vdots & & \vdots \end{pmatrix}$$

Therefore, we extend the update from Equation 5 by transforming the activations and logits to matrices with m columns each:

$$\mathbf{A}^{[l]} = \begin{pmatrix} \vdots & \vdots & & \vdots \\ \mathbf{a}^{[l](1)} & \mathbf{a}^{[l](2)} & \cdots & \mathbf{a}^{[l](m)} \\ \vdots & \vdots & & \vdots \end{pmatrix} \qquad \mathbf{Z}^{[l]} = \begin{pmatrix} \vdots & \vdots & & \vdots \\ \mathbf{z}^{[l](1)} & \mathbf{z}^{[l](2)} & \cdots & \mathbf{z}^{[l](m)} \\ \vdots & \vdots & & \vdots \end{pmatrix}$$

$$\mathbf{Z}^{[l]} = \mathbf{W}^{[l]} \cdot \mathbf{A}^{[l-1]} + \mathbf{b}^{[l]}$$
$$\mathbf{A}^{[l]} = g^{[l]}\left(\mathbf{Z}^{[l]}\right)$$

Equation 6

The dimensions are as follows:

$\mathbf{A}^{[l-1]}$: n_{l-1} x m
$\mathbf{W}^{[l]}$: n_l x n_{l-1}
$\mathbf{b}^{[l]}$: n_l x 1
$\mathbf{Z}^{[l]}$: n_l x m
$\mathbf{A}^{[l]}$: n_l x m

It should be noted that in Equation 6 the product $\mathbf{W}^{[l]} \cdot \mathbf{A}^{[l-1]}$ is a standard matrix multiplication and that the addition of the bias $\mathbf{b}^{[l]}$ makes use of the broadcast functionality of numpy along the dimension of m. Based on this formulation the implementation of the practical work will be most efficient and should therefore be adapted. In Figure 110 again a concrete example is given including the corresponding code Python snippet.

$$\mathbf{A}^{[l-1]} = \begin{pmatrix} 1 & -1 \\ -1 & -0.5 \\ -0.5 & 1 \end{pmatrix} \quad \mathbf{W}^{[l]} = \begin{pmatrix} 1 & -1 & -0.5 \\ 1 & 0.5 & -1 \\ -0.2 & 0.3 & 0.5 \\ -1 & 1 & 1 \end{pmatrix} \quad \mathbf{b}^{[l]} = \begin{pmatrix} -1 \\ -1 \\ 1 \\ 1 \end{pmatrix}$$

$$\mathbf{Z}^{[l]} = \mathbf{W}^{[l]} \cdot \mathbf{A}^{[l-1]} + \mathbf{b}^{[l]} = \begin{pmatrix} 1.25 & -2.0 \\ 0.0 & -3.25 \\ 0.25 & 1.55 \\ -1.5 & 2.5 \end{pmatrix} \quad \mathbf{A}^{[l]} = \sigma(\mathbf{Z}^{[l]}) = \begin{pmatrix} 0.78 & 0.12 \\ 0.50 & 0.04 \\ 0.56 & 0.83 \\ 0.18 & 0.92 \end{pmatrix}$$

```
import numpy as np
W = np.array([[1, -1, -0.5],[1, 0.5, -1], [-0.2, 0.3, 0.5],[-1, 1, 1]]).reshape(4,3)
b = np.array([-1, -1, 1, 1]).reshape(4,1)
aprev = np.array([[1, -1, -0.5],[-1, -0.5, 1]]).T.reshape(3,2)
z = np.matmul(W,aprev)+b
a = 1./(1.0+np.exp(-z))
```

Figure 110: *Example calculation for one layer to illustrate the correct matrix-wise multiplication.*

6.3 Backpropagation

Having developed the formal representation of a MLP as a computational graph and based on the formulas for the forward pass in its most general form (Equation 6) we can now develop the corresponding equations for the backward pass the so-called backpropagation. These considerations go back to a work by Rumelhart [10]. We recall that the goal is to minimise the cost function $J(\boldsymbol{\theta})$ (quadratic for regression problems, cross-entropy for classification)

$$J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L\left(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)}\right) = \frac{1}{m}\sum_{i=1}^{m} L\left(h_{\theta,y^{(i)}}\left(\mathbf{x}^{(i)}\right), \mathbf{y}^{(i)}\right)$$

with respect to the model parameters, which for an MLP are given by:

$$\boldsymbol{\theta} = \left(\mathbf{W}^{[1]}, \mathbf{b}^{[1]}, \ldots, \mathbf{W}^{[L]}, \mathbf{b}^{[L]}\right)$$

As before $\hat{\mathbf{y}}^{(i)}$ and $\mathbf{y}^{(i)}$ denote, respectively, the prediction and the true outcome for the training sample $\mathbf{x}^{(i)}$. For the minimization step we require the update rules for the Gradient Descent:

$$\mathbf{W}^{[l]} \leftarrow \mathbf{W}^{[l]} - \alpha \cdot \frac{\partial J(\boldsymbol{\theta})}{\partial \mathbf{W}^{[l]}}$$

$$\mathbf{b}^{[l]} \leftarrow \mathbf{b}^{[l]} - \alpha \cdot \frac{\partial J(\boldsymbol{\theta})}{\partial \mathbf{b}^{[l]}}$$

I.e., we require the derivatives of the cost function $J(\boldsymbol{\theta})$ with respect to all the parameters $\mathbf{W}^{[l]}$ and $\mathbf{b}^{[l]}$ of the MLP. We will start by recalling some general notions of differential calculus.

6.3.1 Chain Rule of Differential Calculus

We define the function

$$y = f(x) = \sqrt{1 + x^2}$$

as concatenation of

$$g(z) = \sqrt{z}$$

and

$$h(x) = 1 + x^2$$

according to:

$$y = f(x) = g\big(h(x)\big)$$

We now want to calculate the derivative of:

$$\frac{dy}{dx} = \frac{df(x)}{dx}$$

While this is in principle straight forward, we want to obtain the result by applying the chain rule of calculus:

$$\frac{df(x)}{dx} = \frac{d}{dx} g\big(h(x)\big) = \frac{dg(z)}{dz}\bigg|_{z=h(x)} \cdot \frac{dh(x)}{dx}$$

To evaluate this formula, we require:

$$\frac{dg(z)}{dz} = \frac{d}{dz}\sqrt{z} = \frac{1}{2\sqrt{z}}$$

$$\frac{dh(x)}{dx} = \frac{d}{dx}(1 + x^2) = 2x$$

Putting everything together we find:

$$\frac{dy}{dx} = \frac{df(x)}{dx} = \frac{d}{dx} g\big(h(x)\big) = \frac{dg(z)}{dz}\bigg|_{z=h(x)} \cdot \frac{dh(x)}{dx} = \frac{1}{2\sqrt{z}}\bigg|_{z=h(x)} \cdot 2x = \frac{1}{2\sqrt{1+x^2}} \cdot 2x = \frac{x}{\sqrt{1+x^2}}$$

The advantage of applying the chain rule is, that we only require the derivatives of the functions with respect to their respective arguments i.e.,

$$\frac{dg(z)}{dz}$$

and

$$\frac{dh(x)}{dx}$$

We can now represent the forward propagation i.e., $y = f(x) = g\big(h(x)\big)$ and the backward propagation i.e., the determination of the derivative $\frac{dy}{dx} = \frac{df(x)}{dx}$ as a computational graph:

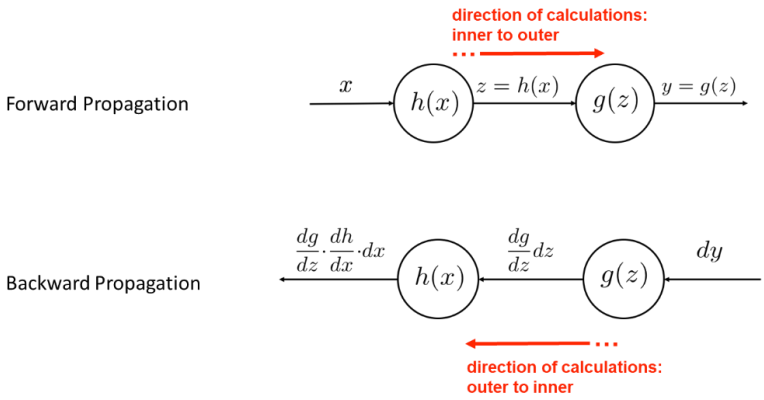


Figure 111: *Representation of the forward and backward propagation for a simple chain rule example.*

We see, that in the forward pass we just have to successively concatenate the function application while for the backward propagation we can simply multiply successively the respective derivatives. Based on these considerations we can now formulate the backpropagation in general form.

6.3.2 Backpropagation for single Node

The generalization of the example given in Figure 111 is presented in Figure 112. The forward pass is given by the application of the function $y = f(x)$. We assume that the change of the cost with respect to the output y of the layer is known i.e.:

$$dy \cdot \frac{\partial L}{\partial y}$$

By substituting

$$dy = \frac{\partial f(x)}{\partial x} \cdot dx$$

we can now propagate the error calculation further through the given layer to obtain

$$dx \cdot \frac{\partial f(x)}{\partial x} \cdot \frac{\partial L}{\partial y}$$

as input for the next layer left to the given one.

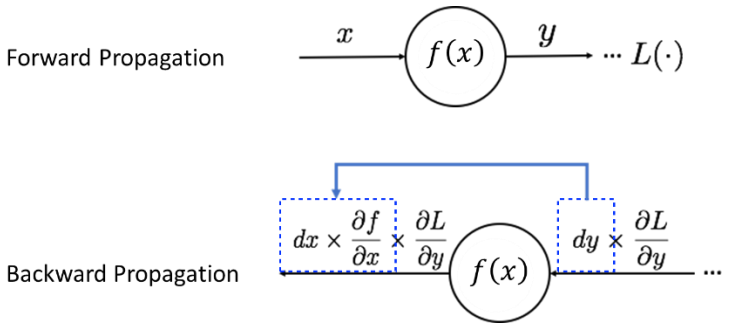


Figure 112: *General formulation of backpropagation for a single node.*

6.3.3 Backprop through a Single MLP Layer

We recall the formula for the forward pass for a single MLP layer (Figure 113).

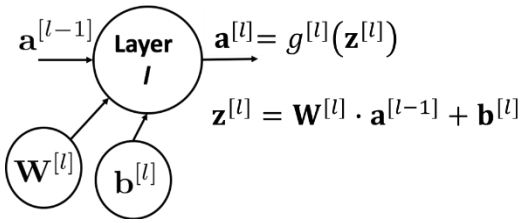


Figure 113: *Forward pass for a single MLP layer.*

Assuming that we have the quantity

$$\frac{\partial L}{\partial \mathbf{a}^{[l]}}$$

from the considerations of the previous sections, we must determine the following quantities:

$$\frac{\partial L}{\partial \mathbf{z}^{[l]}} \quad , \quad \frac{\partial L}{\partial \mathbf{W}^{[l]}} \quad , \quad \frac{\partial L}{\partial \mathbf{b}^{[l]}} \quad , \quad \frac{\partial L}{\partial \mathbf{a}^{[l-1]}}$$

We will in a first step determine these quantities by making the vector and matrix indices explicit. Then we will formulate a compact matrix-based notation.
With the matrix indices explicit we determine the j-th component of the activation $a_j^{[l]}$ of layer l as an element-wise application of the activation function $g^{[l]}$ on the j-th component of the logit $z_j^{[l]}$. The latter can be written as a sum over the weights $W_{ji}^{[l]}$ multiplied by the activations of the previous layer $a_i^{[l-1]}$ plus the corresponding bias term $b_j^{[l]}$:

$$a_j^{[l]} = g^{[l]}\left(z_j^{[l]}\right) = g^{[l]}\left(\sum_i W_{ji}^{[l]} \cdot a_i^{[l-1]} + b_j^{[l]}\right)$$

Equation 7

We will use this representation to determine the four quantities required for the backpropagation:

6.3.3.1 Determination of $\frac{\partial L}{\partial \mathbf{z}^{[l]}}$

$$\frac{\partial L}{\partial z_j^{[l]}} = \sum_k \frac{\partial L}{\partial a_k^{[l]}} \cdot \frac{\partial a_k^{[l]}}{\partial z_j^{[l]}} =^{1)} \sum_k \frac{\partial L}{\partial a_k^{[l]}} \cdot \delta_{kj} \cdot \frac{dg^{[l]}\left(z_j^{[l]}\right)}{dz} = \frac{\partial L}{\partial a_j^{[l]}} \cdot \frac{dg^{[l]}\left(z_j^{[l]}\right)}{dz}$$

At the equal sign $=^{1)}$ we introduced the Kronecker delta δ_{kj} which is equal to one if $k = j$ and zero else. It is a consequence of the fact, that the activation function $g^{[l]}(z)$ is applied element wise to the logits $z_j^{[l]}$. Therefore, the derivative of $a_k^{[l]} = g^{[l]}(z_k^{[l]})$ with respect to $z_j^{[l]}$ only gives a contribution for $k = j$. Thus, we obtain:

$$\boxed{\frac{\partial L}{\partial z_j^{[l]}} = \frac{\partial L}{\partial a_j^{[l]}} \cdot \frac{dg^{[l]}\left(z_j^{[l]}\right)}{dz}}$$

6.3.3.2 Determination of $\frac{\partial L}{\partial \mathbf{W}^{[l]}}$

$$\frac{\partial L}{\partial W_{ki}^{[l]}} = \sum_j \frac{\partial L}{\partial z_j^{[l]}} \cdot \frac{\partial z_j^{[l]}}{\partial W_{ki}^{[l]}} =^{1)} \sum_j \frac{\partial L}{\partial z_j^{[l]}} \cdot \delta_{kj} \cdot a_i^{[l-1]} = \frac{\partial L}{\partial z_k^{[l]}} \cdot a_i^{[l-1]}$$

At the equal sign $=^{1)}$ we again introduced the Kronecker delta δ_{kj} because only the logit to index $j = k$ will depend on $W_{ki}^{[l]}$. Furthermore, only the i-th component of the activation of the previous layer $a_i^{[l-1]}$ survives the derivative of $W_{ki}^{[l]}$. Thus, we obtain:

$$\boxed{\frac{\partial L}{\partial W_{ki}^{[l]}} = \frac{\partial L}{\partial z_k^{[l]}} \cdot a_i^{[l-1]}}$$

6.3.3.3 Determination of $\frac{\partial L}{\partial \mathbf{b}^{[l]}}$

This follows closely the derivation of the precious chapter 6.3.3.2:

$$\frac{\partial L}{\partial b_k^{[l]}} = \sum_j \frac{\partial L}{\partial z_j^{[l]}} \cdot \frac{\partial z_j^{[l]}}{\partial b_k^{[l]}} =^{1)} \sum_j \frac{\partial L}{\partial z_j^{[l]}} \cdot \delta_{kj} \cdot 1 = \frac{\partial L}{\partial z_k^{[l]}}$$

Thus, we obtain:

$$\boxed{\frac{\partial L}{\partial b_k^{[l]}} = \frac{\partial L}{\partial z_k^{[l]}}}$$

6.3.3.4 Determination of $\frac{\partial L}{\partial \mathbf{a}^{[l-1]}}$

$$\frac{\partial L}{\partial a_k^{[l-1]}} = \sum_j \frac{\partial L}{\partial z_j^{[l]}} \cdot \frac{\partial z_j^{[l]}}{\partial a_k^{[l-1]}} =^{1)} \sum_j \frac{\partial L}{\partial z_j^{[l]}} \cdot W_{jk}$$

At the equal sign $=^{1)}$ we used the fact that all logits $z_j^{[l]}$ depend on the activation of the previous layer $a_k^{[l-1]}$ but only the weight W_{jk} to column index k survives the derivative with respect to $a_k^{[l-1]}$.
Thus, we obtain:

$$\boxed{\frac{\partial L}{\partial a_k^{[l-1]}} = \sum_j \frac{\partial L}{\partial z_j^{[l]}} \cdot W_{jk}}$$

We will now rewrite these formulas in vector and matrix notation which will give a compact representation of the backpropagation scheme.

6.3.4 General Formulation of Backpropagation in Matrix Notation

The general view is represented in the following Figure 114.

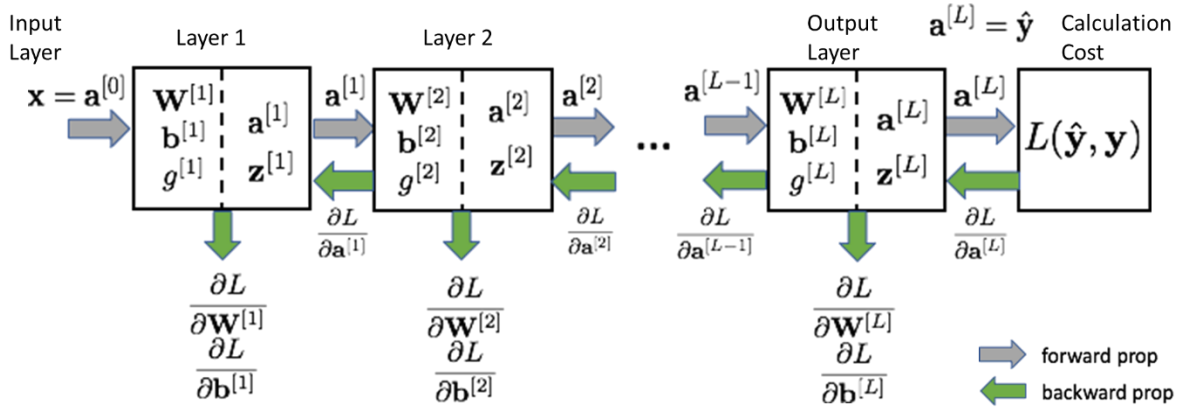


Figure 114: *Overview of the backpropagation scheme for an MLP.*

We now rewrite the equations from the previous sections in matrix notations:

$$\frac{\partial L}{\partial \mathbf{z}^{[l]}} = \frac{\partial L}{\partial \mathbf{a}^{[l]}} * \frac{dg^{[l]}(\mathbf{z}^{[l]})}{dz}$$
$$\frac{\partial L}{\partial \mathbf{W}^{[l]}} = \frac{\partial L}{\partial \mathbf{z}^{[l]}} \cdot \left(\mathbf{a}^{[l-1]}\right)^T$$
$$\frac{\partial L}{\partial \mathbf{b}^{[l]}} = \frac{\partial L}{\partial \mathbf{z}^{[l]}}$$
$$\frac{\partial L}{\partial \mathbf{a}^{[l-1]}} = \left(\mathbf{W}^{[l]}\right)^T \cdot \frac{\partial L}{\partial \mathbf{z}^{[l]}}$$

Equation 8

The following points should be noted:

- The operation in the first equation denoted by $*$ is an *elementwise* multiplication. Both $\frac{\partial L}{\partial \mathbf{a}^{[l]}}$ and $\frac{\partial g^{[l]}(\mathbf{z}^{[l]})}{\partial z}$ are vectors of size n^l x 1. Thus, their *elementwise* product will also give a vector of the same size.
- In the second equation a matrix product between $\frac{\partial L}{\partial \mathbf{z}^{[l]}}$ and the transpose of $\mathbf{a}^{[l-1]}$ (note operator $(..)^T$) is required. Here $\frac{\partial L}{\partial \mathbf{z}^{[l]}}$ is a vector of size n^l x 1 and the transpose of $\mathbf{a}^{[l-1]}$ is a vector of size 1 x n^{l-1}. Thus, the product will have the dimension n^l x n^{l-1}.
- For the last equation the matrix product of the transpose of $\mathbf{W}^{[l]}$ (dimension n^{l-1} x n^l) and $\frac{\partial L}{\partial \mathbf{z}^{[l]}}$ (dimension n^l x 1) is calculated with a result of dimension n^{l-1} x 1.

These formulas are correct for a fully connected layer of an MLP. For the final Softmax layer the first equation reads different. Here we only consider one term of the full sum of the cost function (with the extension to the full sum done in a further step):

$$\frac{\partial L}{\partial z_j^{[L]}} = -\frac{\partial}{\partial z_j^{[l]}} \log\left[\frac{\exp z_y^{[L]}}{\sum_{k=0}^{K-1} \exp z_k^{[L]}}\right] = -\frac{\partial}{\partial z_j^{[l]}}\left[z_y^{[L]} - \log \sum_{k=0}^{K-1} \exp z_k^{[L]}\right] = -\left[\delta_{j,y} - \frac{\exp z_j^{[L]}}{\sum_{k=0}^{K-1} \exp z_k^{[L]}}\right]$$
$$= -\left[\delta_{j,y} - \frac{\exp z_j^{[L]}}{\sum_{k=0}^{K-1} \exp z_k^{[L]}}\right] = -\left[\delta_{j,y} - \hat{y}_j\right]$$

Here, y denotes the correct label for the given sample $\mathbf{x}$ being the input to the first layer and $\hat{y}_j$ is the prediction of the model. Thus, we obtain for the Softmax layer:

$$\frac{\partial L}{\partial z_j^{[L]}} = \hat{y}_j - \delta_{j,y}$$

Or in vector notation:

$$\frac{\partial L}{\partial \mathbf{z}^{[L]}} = \hat{\mathbf{y}} - \mathbf{y}$$

Here both the prediction $\hat{\mathbf{y}}$ and the true label $\mathbf{y}$ are vectors of size n^L x 1, with n^L being the number of Softmax outputs. Furthermore, the true label $\mathbf{y}$ is a one hot vector with entries equal to zero except for the correct label where it is a one.

6.3.5 Formulation of Backpropagation for full Batch

In Equation 6 we already formulated the forward pass for a batch of m samples. We now extend the formulation of the backpropagation also for the case that the cost function is an average over m samples:

$$J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L\left(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)}\right)$$

This extension can be done in a similar way as was done for Equation 6. From there we can observe that the matrix equations for the forward pass are simply an extension to activations $\mathbf{A}^{[l]}$ and logits $\mathbf{Z}^{[l]}$ being matrices with m i.e., number of samples columns:

$$\mathbf{A}^{[l]} = \begin{pmatrix} \vdots & \vdots & & \vdots \\ \mathbf{a}^{[l](1)} & \mathbf{a}^{[l](2)} & \cdots & \mathbf{a}^{[l](m)} \\ \vdots & \vdots & & \vdots \end{pmatrix} \qquad \mathbf{Z}^{[l]} = \begin{pmatrix} \vdots & \vdots & & \vdots \\ \mathbf{z}^{[l](1)} & \mathbf{z}^{[l](2)} & \cdots & \mathbf{z}^{[l](m)} \\ \vdots & \vdots & & \vdots \end{pmatrix}$$

The formulation for backpropagation equations is similar in the sense that we will backpropagate each column $\mathbf{a}^{[l](i)}$ $(i = 1..m)$ of the activation matrix $\mathbf{A}^{[l]}$ of layer l in parallel. However, for the determination of $\frac{\partial L}{\partial \mathbf{W}^{[l]}}$ and $\frac{\partial L}{\partial \mathbf{b}^{[l]}}$ we will have to apply the averaging over the m samples. We will see that in the following.

We now introduce the derivative $\frac{\partial L}{\partial \mathbf{A}^{[l]}}$ of the cost function L with respect to the activation matrix $\mathbf{A}^{[l]}$.

The latter consists of the columns $\mathbf{a}^{[l](i)}$ $(i = 1..m)$ and therefore $\frac{\partial L}{\partial \mathbf{A}^{[l]}}$ will also be a matrix with m columns. Each column simply represents the partial derivatives of the cost with respect to the activations in the l-th layer, evaluated for the i-th sample:

$$\frac{\partial L}{\partial \mathbf{A}^{[l]}} = \begin{pmatrix} \vdots & \vdots & & \vdots \\ \frac{\partial L}{\partial \mathbf{a}^{[l](1)}} & \frac{\partial L}{\partial \mathbf{a}^{[l](2)}} & \cdots & \frac{\partial L}{\partial \mathbf{a}^{[l](m)}} \\ \vdots & \vdots & & \vdots \end{pmatrix}$$

Now we can extend the formulation of the backpropagation (Equation 8) to a batch of m samples.

We start with the derivative of the cost with respect to the matrix of logit entries $\frac{\partial L}{\partial \mathbf{Z}^{[l]}}$, which is nothing but an extension of the Equation 8 now performed for each of the $i = 1..m$ columns independently:

$$\frac{\partial L}{\partial \mathbf{Z}^{[l]}} = \frac{\partial L}{\partial \mathbf{A}^{[l]}} * \frac{dg^{[l]}\left(\mathbf{Z}^{[l]}\right)}{dz} = \begin{pmatrix} \vdots & \vdots & & \vdots \\ \frac{\partial L}{\partial \mathbf{z}^{[l](1)}} & \frac{\partial L}{\partial \mathbf{z}^{[l](2)}} & \cdots & \frac{\partial L}{\partial \mathbf{z}^{[l](m)}} \\ \vdots & \vdots & & \vdots \end{pmatrix} =$$

$$= \begin{pmatrix} \vdots & \vdots & & \vdots \\ \frac{\partial L}{\partial \mathbf{a}^{[l](1)}} & \frac{\partial L}{\partial \mathbf{a}^{[l](2)}} & \cdots & \frac{\partial L}{\partial \mathbf{a}^{[l](m)}} \\ \vdots & \vdots & & \vdots \end{pmatrix} * \begin{pmatrix} \vdots & \vdots & & \vdots \\ \frac{dg^{[l]}(\mathbf{z}^{[l](1)})}{dz} & \frac{dg^{[l]}(\mathbf{z}^{[l](2)})}{dz} & \cdots & \frac{dg^{[l]}(\mathbf{z}^{[l](m)})}{dz} \\ \vdots & \vdots & & \vdots \end{pmatrix}$$

Here again the operator $*$ is the element-wise multiplication for the full matrices of dimensions $n^l \text{ x } m$ each.
For the derivative of the cost with respect to the weight matrix $\frac{\partial L}{\partial \mathbf{W}^{[l]}}$ we will – as mentioned above – apply the average over all m training samples. This will read as:

$$\frac{\partial L}{\partial \mathbf{W}^{[l]}} = \frac{1}{m} \cdot \left[\frac{\partial L}{\partial \mathbf{z}^{[l](1)}} \cdot \left(\mathbf{a}^{[l-1](1)}\right)^{\boldsymbol{T}} + \frac{\partial L}{\partial \mathbf{z}^{[l](2)}} \cdot \left(\mathbf{a}^{[l-1](2)}\right)^{\boldsymbol{T}} + \cdots + \frac{\partial L}{\partial \mathbf{z}^{[l](m)}} \cdot \left(\mathbf{a}^{[l-1](m)}\right)^{\boldsymbol{T}} \right]$$

Please note that each of the m summands, corresponding to the i-th training sample, is a full $n^l \text{ x } n^{l-1}$ matrix thus the same is true for the average. Based on the basic rules of matrix multiplication, this can be written in a more compact form:

$$\frac{\partial L}{\partial \mathbf{W}^{[l]}} = \frac{1}{m} \cdot \frac{\partial L}{\partial \mathbf{Z}^{[l]}} \cdot \left(\mathbf{A}^{[l-1]}\right)^{\boldsymbol{T}} = \frac{1}{m} \cdot \begin{pmatrix} \vdots & \vdots & & \vdots \\ \frac{\partial L}{\partial \mathbf{z}^{[l](1)}} & \frac{\partial L}{\partial \mathbf{z}^{[l](2)}} & \cdots & \frac{\partial L}{\partial \mathbf{z}^{[l](m)}} \\ \vdots & \vdots & & \vdots \end{pmatrix} \cdot \begin{pmatrix} \dots & \mathbf{a}^{[l-1](1)} & \dots \\ \dots & \mathbf{a}^{[l-1](2)} & \dots \\ & \vdots & \\ \dots & \mathbf{a}^{[l-1](m)} & \dots \end{pmatrix}$$

Note that the transpose of the matrix $\mathbf{A}^{[l-1]}$ to the right is formed by turning the original columns vectors $\mathbf{a}^{[l-1](1)}$ to rows. Furthermore, the product between $\frac{\partial L}{\partial \mathbf{Z}^{[l]}}$ and the transpose of $\mathbf{A}^{[l-1]}$ is a standard matrix multiplication.
Having developed the more complicated case of the derivative of the cost with respect to the weight vector $\frac{\partial L}{\partial \mathbf{W}^{[l]}}$ it is now straight forward to formulate the rule for the bias vector $\frac{\partial L}{\partial \mathbf{b}^{[l]}}$. Again, we apply the average over all m training samples:

$$\frac{\partial L}{\partial \mathbf{b}^{[l]}} = \frac{1}{m} \cdot \left[\frac{\partial L}{\partial \mathbf{z}^{[l](1)}} + \frac{\partial L}{\partial \mathbf{z}^{[l](2)}} + \cdots + \frac{\partial L}{\partial \mathbf{z}^{[l](m)}} \right]$$

In the following the average aver the m training samples is expressed in Python notation using the sum-operator and – in addition – as a matrix multiplication using a single column vector of size $m \text{ x } 1$:

$$\frac{\partial L}{\partial \mathbf{b}^{[l]}} = \frac{1}{m} \cdot \text{sum}\left(\frac{\partial L}{\partial \mathbf{Z}^{[l]}}, \text{axis} = 1 \right) = \frac{1}{m} \cdot \frac{\partial L}{\partial \mathbf{Z}^{[l]}} \cdot \begin{pmatrix} \vdots \\ \mathbf{1} \\ \vdots \end{pmatrix} = \frac{1}{m} \cdot \begin{pmatrix} \vdots & \vdots & & \vdots \\ \frac{\partial L}{\partial \mathbf{z}^{[l](1)}} & \frac{\partial L}{\partial \mathbf{z}^{[l](2)}} & \cdots & \frac{\partial L}{\partial \mathbf{z}^{[l](m)}} \\ \vdots & \vdots & & \vdots \end{pmatrix} \cdot \begin{pmatrix} \vdots \\ \mathbf{1} \\ \vdots \end{pmatrix}$$

Finally, the propagation of the error to the next lower layer is formulated as follows:

$$\frac{\partial L}{\partial \mathbf{A}^{[l-1]}} = \left(\mathbf{W}^{[l]}\right)^{\boldsymbol{T}} \cdot \frac{\partial L}{\partial \mathbf{Z}^{[l]}} = \begin{pmatrix} \dots & \mathbf{w}^{[l](1)} & \dots \\ \dots & \mathbf{w}^{[l](2)} & \dots \\ & \vdots & \\ \dots & \mathbf{w}^{[l](m)} & \dots \end{pmatrix} \cdot \begin{pmatrix} \vdots & \vdots & & \vdots \\ \frac{\partial L}{\partial \mathbf{z}^{[l](1)}} & \frac{\partial L}{\partial \mathbf{z}^{[l](2)}} & \cdots & \frac{\partial L}{\partial \mathbf{z}^{[l](m)}} \\ \vdots & \vdots & & \vdots \end{pmatrix}$$

We can summarize the component wise formulations from above in compact matrix notation:

$$\frac{\partial L}{\partial \mathbf{Z}^{[l]}} = \frac{\partial L}{\partial \mathbf{A}^{[l]}} * \frac{dg^{[l]}(\mathbf{Z}^{[l]})}{dz}$$

$$\frac{\partial L}{\partial \mathbf{W}^{[l]}} = \frac{1}{m} \cdot \frac{\partial L}{\partial \mathbf{Z}^{[l]}} \cdot \left(\mathbf{A}^{[l-1]}\right)^{\boldsymbol{T}}$$

$$\frac{\partial L}{\partial \mathbf{b}^{[l]}} = \frac{1}{m} \cdot \frac{\partial L}{\partial \mathbf{Z}^{[l]}} \cdot \begin{pmatrix} \vdots \\ \mathbf{1} \\ \vdots \end{pmatrix}$$

$$\frac{\partial L}{\partial \mathbf{A}^{[l-1]}} = \left(\mathbf{W}^{[l]}\right)^{T} \cdot \frac{\partial L}{\partial \mathbf{Z}^{[l]}}$$

Equation 9

This is completed by the equation for $\frac{\partial L}{\partial \mathbf{Z}^{[l]}}$ for the last i.e., the Softmax layer:

$$\frac{\partial L}{\partial \mathbf{Z}^{[L]}} = \hat{\mathbf{Y}} - \mathbf{Y}$$

The matrix dimensions of the above quantities are as follows:

$$\frac{\partial L}{\partial \mathbf{Z}^{[l]}}, \frac{\partial L}{\partial \mathbf{A}^{[l]}}, \frac{dg^{[l]}\left(\mathbf{Z}^{[l]}\right)}{dz}: \; n_l \text{ x } m$$

$$\mathbf{A}^{[l-1]}, \frac{\partial L}{\partial \mathbf{A}^{[l-1]}}: \; n_{l-1} \text{ x } m$$

$$\frac{\partial L}{\partial \mathbf{W}^{[l]}}: \; n_l \text{ x } n_{l-1}$$

$$\frac{\partial L}{\partial \mathbf{b}^{[l]}}: \; n_l \text{ x } 1$$

$$\hat{\mathbf{Y}}, \mathbf{Y}: \; n_L \text{ x } m$$

The formulas given above (Equation 9) represent the implementation of the backpropagation for an MLP including two types of layers, the fully-connected and the Softmax layer. For other layer types e.g., convolutional layers of a CNN or RNN the backpropagation formulas are derived very similarly – by backpropagating through the computational graph. For CNN, the computational structure turns out to be similar, for RNNs, we will also have terms resulting from back propagation through time.

6.3.5.1 Numerical Gradient Check

As the above calculation are error prone it is helpful – for debugging purposes – to check numerically whether the calculation of the gradient of the loss function with respect to the weights and the biases is correct. This is performed as follows. Assume we want to check the derivative of the cost function with respect to the weight $W_{kj}^{[l]}$ of the layer l: Therefore, we determine the following quantity:

$$\frac{\partial L}{\partial W_{kj}^{[l]}} = \lim_{\varepsilon \to 0} \frac{L\left(\dots, W_{kj}^{[l]} + \varepsilon, \dots\right) - L(\dots, W_{kj}^{[l]}, \dots)}{\varepsilon}$$

We can approximate the true value of the derivative using a small but finite ε_0:

$$\frac{\partial L}{\partial W_{kj}^{[l]}} \approx \frac{L\left(\dots, W_{kj}^{[l]} + \varepsilon_0, \dots\right) - L(\dots, W_{kj}^{[l]}, \dots)}{\varepsilon_0}$$

7 Improved Strategies for Training Deep Networks

In the previous chapter we formulated the backpropagation algorithm for an MLP with arbitrary number of fully connected layers. The foundations for this were laid by Rumelhart back in the 1980s [10]. It may be surprising, therefore, that the ground-breaking progress obtained with deep NN took more than two more decades. In chapter 1.2.2 we have already presented the main drivers of DL, namely sheer quantitative progress in the form of more training data and improved computational power on the one hand, and algorithmic improvement on the other. Here we want to address the latter issue. Indeed, attempts to train deep neural networks until 2010 were not successful due to various problems:

- Vanishing and exploding gradients:
  When propagating the gradient of the cost function backwards through the network layers in each steps a multiplication with the gradient of the activation function $\frac{dg^{[l]}}{dz}$ and the layer weights $\left(\mathbf{W}^{[l]}\right)^{\boldsymbol{T}}$ is performed (Equation 9). This iterative multiplication may either lead to vanishing gradients e.g., when using the sigmoid activation functions with gradient close to zeros in the tails, or to exploding gradients when the weights are not properly initialized.

- Insufficient performance of the optimization schemes:
  While standard gradient descent methods (batch, mini-batch or stochastic) are efficient for shallow networks, they turn out to be much less efficient for deep networks with complex cost surfaces (Figure 49).

- Overfitting:
  The deeper the network the larger the number of parameters to optimize and the size of the training set may not increase accordingly. Therefore, efficient strategies to avoid overfitting are required.

We will address these issues step by step in the following sections.

7.1 Vanishing and Exploding Gradients

As mentioned in the introduction, general difficulties in training deep neural networks were observed until 2010, given the techniques available until then. When propagating the gradient of the cost function backwards through the network layers either vanishing or exploding or gradient were observed. More generally, deep NNs suffer from unstable gradients as different layers may learn at widely different speeds. Important insights into the reasons for these problems as well as algorithmic solutions have been obtained since then with important contributions from X. Glorot, J. Bengio [11] and S. Ioffe, C. Szegedy [12], which we will discuss in the following[74]. We will first list the various interrelated problems that can occur and then discuss the strategies that have been developed to address them.

7.1.1 Saturation of Activations

X. Glorot and J. Bengio [11] studied the behaviour of the activation values $\mathbf{A}^{[l]}$ i.e., the output of the activation functions, for a NN with 4 layers. Figure 115 shows one of their findings using the sigmoid activation function. The mean value of the activations $\mathbf{A}^{[l]}$ and their respective standard deviations (vertical bars) are shown for the 4 layers, where index $l = 1$ corresponds to the first hidden layer. The x-axis represents the number of training epochs. As expected, the values of activations observed (y-axis) lie in the interval $[0,1]$ corresponding to the output range of the sigmoid function. One observes, that the highest i.e., 4th layer saturates very early during the training phase at the value 0, which – according to the authors – considerably slows down the training. It only escapes the saturation region after training epoch ~100.

[74] See also the discussions in http://neuralnetworksanddeeplearning.com/chap5.html.

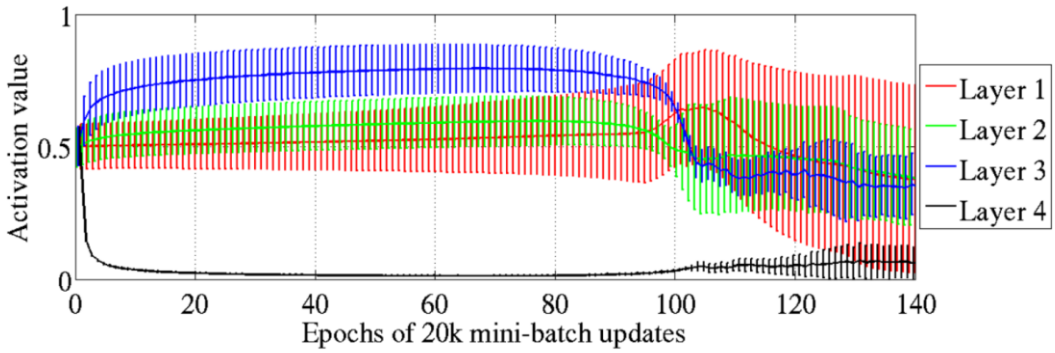


Figure 115: *The Problem of saturating activation caused by the sigmoid activation function [11].*

Such a saturation of the activation will directly cause vanishing gradients, as illustrated in Figure 116 and Figure 117, using the sigmoid activation function. In Figure 116 the application of the sigmoid function (top left) to a hypothetic set of logits z_i (bottom), which follow a normal distribution with mean $\mu_z = 0$ and standard deviation $\sigma_Z = 1$, is represented in the top right graph. For such a moderate standard deviation, the mapping will not lead to a saturation of the activations i.e., their values remain well distributed within the interval $[0,1]$.

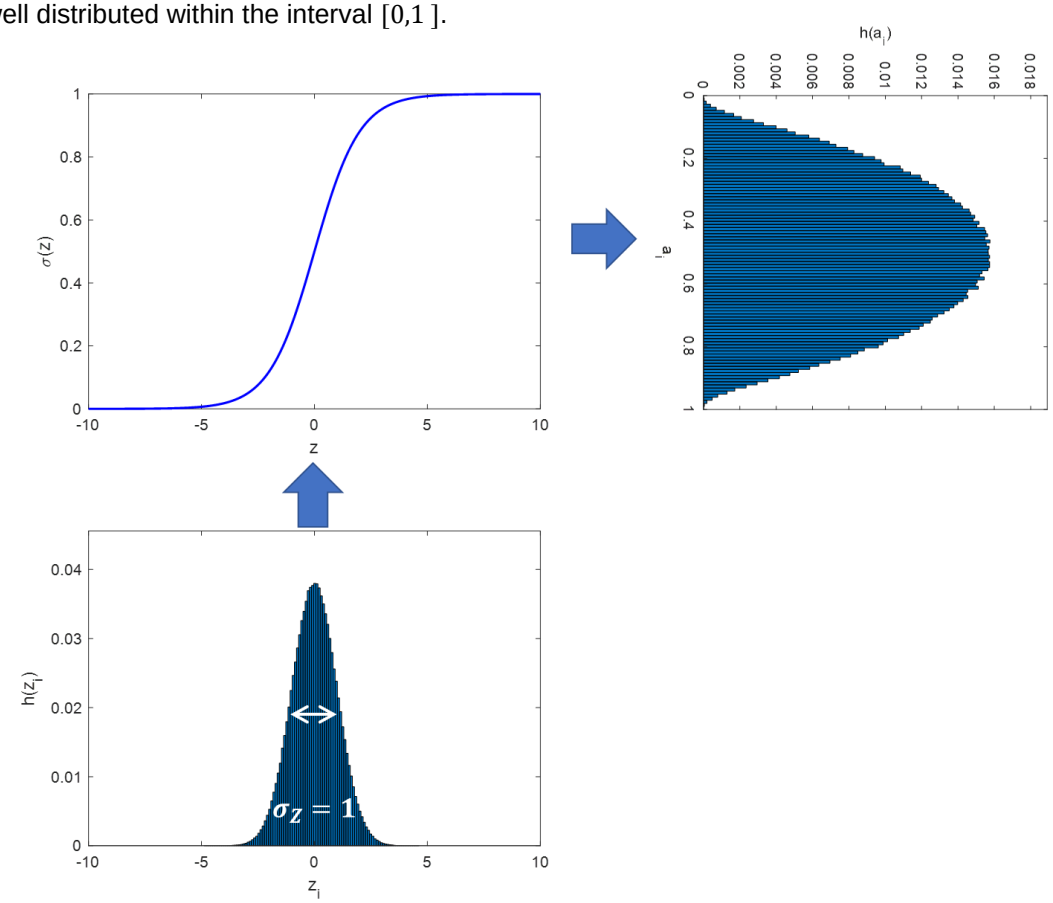


Figure 116: *If the logits (bottom) – the input to the sigmoid activation function (top left) – have a moderate standard deviation ($\sigma_Z = 1$) the saturation regions of the sigmoid function are reached with low probability (top right).*

However, if we increase the standard deviation of the logits to $\sigma_Z = 3$, as shown in Figure 117, the application of the sigmoid function to z_i leads to a strong saturation of the activations with a large part of the distribution pushed to the saturation regions zero and one (top right). Yet, these saturated activations will – in a successive backpropagation step – lead to gradients of the activation function $\frac{dg^{[l]}}{dz}$ which are essentially zero i.e., to the cited vanishing gradient problem.

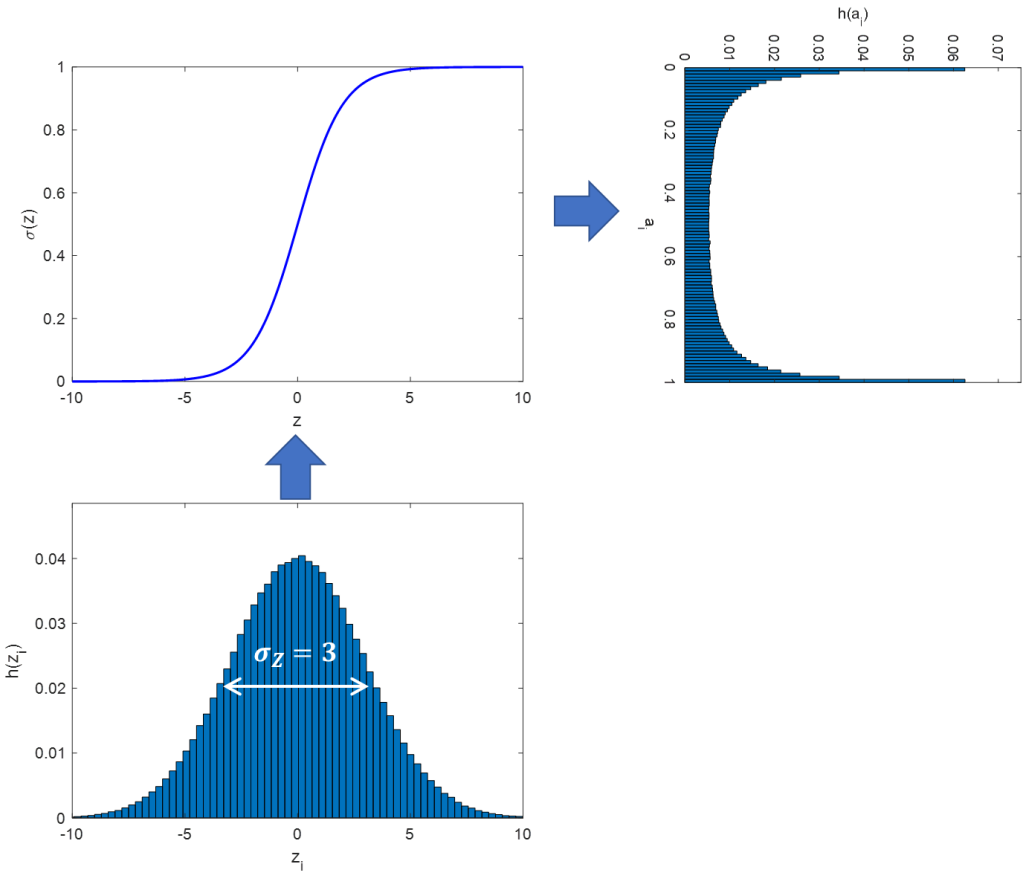


Figure 117: *If the logits (bottom) – the input to the sigmoid activation function (top left) – have a high standard deviation ($\sigma_z = 3$) the saturation regions of the sigmoid function are reached with high probability (top right).*

A further problem highlighted by this discussion is the fact that the sigmoid function introduces a systematic bias of $+0.5$ in each layer, which can also push activations into saturation at 1. Thus X. Glorot and J. Bengio [11] found that the Hyperbolic Tangent (Table 4), which is symmetric around zero, behaves better than the sigmoid function with respect to this issue. This raises another important point or question which is the correct choice of the activation function (c.f. chapter 7.1.6).

7.1.2 Changing Variance of Activations and Gradients

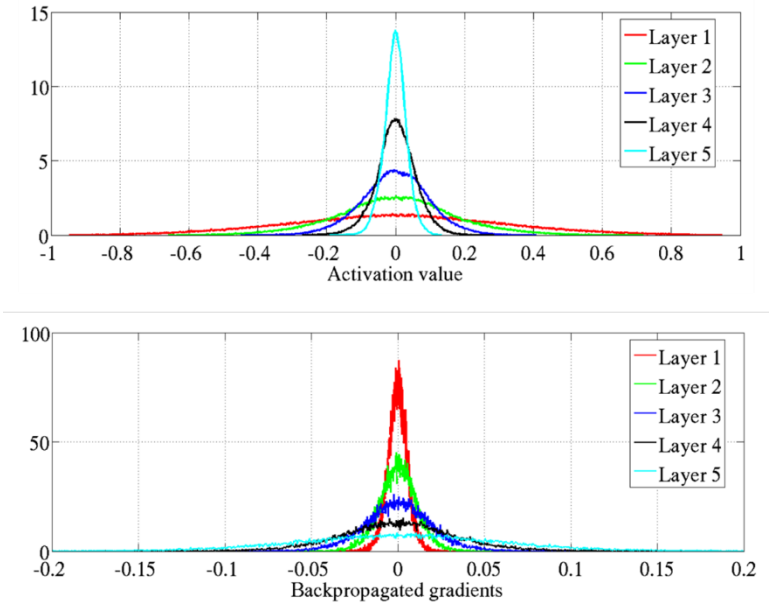


Figure 118: *The variance of both activations (top) and gradients (bottom) change over the layers [11].*

A problem related to the discussions of the previous section 7.1.1 is the change of the variance of both the activations $\mathbf{A}^{[l]}$ in the forward pass and of the gradients $\frac{\partial L}{\partial \mathbf{A}^{[l]}}$ during the backpropagation for different layer indices l. In fact, X. Glorot and J. Bengio [11] found that the variance of the activations $\mathbf{A}^{[l]}$ in the forward pass decrease with increasing layer index l (Figure 118, top). Similar behaviour was found for the variance of the gradients $\frac{\partial L}{\partial \mathbf{A}^{[l]}}$ which were found to decrease with *decreasing* layer index l during the backpropagation (Figure 118, bottom). In fact, these observations were made right at the beginning of the training procedure and are related to the initialization of the network parameters (weights and biases) which must be controlled correctly (c.f. chapter 7.1.4).

7.1.3 Multiplicative Structure of Backpropagation

Finally, the multiplicative structure of the backpropagation (Equation 9) when applied successively to a DL architecture may be the cause of unstable gradients[74]. We consider a "toy" MLP with L identical layers, each having only one neuron and the sigmoid activation function. It is easy to show by application of Equation 9 that the derivate of the cost function with respect to the weights $w^{[l]}$ and biases $b^{[l]}$ of layer l are given by the following expression:

$$\begin{aligned} \frac{\partial L}{\partial w^{[l]}} &= a^{[l-1]} \cdot g'(z^{[l]}) \cdot \left(\prod_{k=l+1}^{L} w^{[k]} g'(z^{[k]})\right) \cdot \frac{\partial L}{\partial a^{[L]}} \\ \frac{\partial L}{\partial b^{[l]}} &= g'(z^{[l]}) \cdot \left(\prod_{k=l+1}^{L} w^{[k]} g'(z^{[k]})\right) \cdot \frac{\partial L}{\partial a^{[L]}} \end{aligned}$$

Product Term

Figure 119: *Successive application of the backpropagation rule leads to numerically unstable terms.*

If the number of layers L increases the product term may become unstable leading both to low i.e., vanishing, or high i.e., exploding gradients. E.g., for the sigmoid activation $g(z) = \sigma(z)$ function, we can express the derivative as:

$$g'(z) = \sigma'(z) = \sigma(z) \cdot (1 - \sigma(z))$$

It follows immediately that the absolute value of $\sigma'(z)$ is less than 1/4. Thus, by successive application of the backpropagation rule, we will decrease the gradient values systematically by a factor of 1/4, which may lead to vanishing gradients. On the other hand, if the weights $w^{[l]}$ are large the product term may increase and exploding gradients may results.

As already mentioned, the problems illustrated in the previous sections 7.1.1 till 7.1.3 limited the success of training deeper NN architectures till the 2010. We will now discuss the solutions proposed in [11] and [12], which where the basis for breakthrough in DL training:

- Proper parameter initialisation:
  The change of variance and derivatives observed as a function of the layer index at the beginning of the training procedure (Figure 118) can be handled with a proper initialization of the weights so that the logits do not grow too large in magnitude.

- Batch Normalisation:
  To further assure proper weight scaling also when training continues the so-called Batch Normalisation scheme will be introduced.

- Non-saturating Activation Functions:
  Finally the use of activation functions that do not saturate will also be beneficial for the training of DL architectures.

- Gradient Clipping:
  Finally, application of clipping of the gradients i.e., manually limiting their absolute values, stabilises the training of deep NNs.

7.1.4 Xavier and He Initialization of Network Parameters

We recall that X. Glorot and J. Bengio [11] studied the change of the variance for the activations $\mathbf{A}^{[l]}$ in

the forward pass and for the gradients $\frac{\partial L}{\partial \mathbf{A}^{[l]}}$ during the backpropagation as a function of the layer index l. The behaviour shown in Figure 118 was observed at the beginning of the training procedure and the authors concluded that this was related to a poor initialization of the network parameters. Figure 118 was obtained with the following initialization scheme (considered as state of the art in 2010[75]):

$$\mathbf{W}^{[l]} \sim \mathrm{U}\left[-\frac{1}{\sqrt{n_{l-1}}}, \frac{1}{\sqrt{n_{l-1}}}\right]$$
$$\mathbf{b}^{[l]} = 0$$

Equation 10

Here $\mathrm{U}[-\mathrm{a}, \mathrm{a}]$ is a uniform distribution in the interval $[-\mathrm{a}, \mathrm{a}]$ and n_{l-1} is the size – i.e. number of output neurons – of layer index $l-1$. The considerations behind this initialization scheme are the following. We recall Equation 7 for the update of the layer l given its weights $W_{ji}^{[l]}$ and biases $b_j^{[l]}$. Here the sum extends over n_{l-1} i.e., the size of layer index $l-1$.

$$a_j^{[l]} = g^{[l]}\left(z_j^{[l]}\right) = g^{[l]}\left(\sum_i W_{ji}^{[l]} \cdot a_i^{[l-1]} + b_j^{[l]}\right)$$

Assuming that we are still in the linear regime of the activation function $g^{[l]}(z) \sim z$ e.g., the centre of the sigmoid function, we can evaluate the variance of the activation:

$$\mathrm{Var}\left(a_j^{[l]}\right) \sim \mathrm{Var}\left(\sum_i W_{ji}^{[l]} \cdot a_i^{[l-1]} + b_j^{[l]}\right) =^{(1)} \mathrm{Var}\left(\sum_i W_{ji}^{[l]} \cdot a_i^{[l-1]}\right) =^{(2)} \sum_i \mathrm{Var}\left(W_{ji}^{[l]}\right) \cdot \mathrm{Var}\left(a_i^{[l-1]}\right)$$

In step $=^{(1)}$ we used that the biases are initialized to zero and for $=^{(2)}$ the independence of the weights $W_{ji}^{[l]}$ and the activations $a_i^{[l-1]}$. If we now assume that both the weights $W_{ji}^{[l]}$ and the activations $a_i^{[l-1]}$ have all the same variance (denoted by $\mathrm{Var}(W^{[l]})$ and $\mathrm{Var}(a^{[l-1]})$ respectively) we can further conclude:

$$\mathrm{Var}\left(a_j^{[l]}\right) \sim \sum_i \mathrm{Var}\left(W_{ji}^{[l]}\right) \cdot \mathrm{Var}\left(a_i^{[l-1]}\right) = \underbrace{n_{l-1} \cdot \mathrm{Var}\left(W^{[l]}\right)}_{(*)} \cdot \mathrm{Var}\left(a^{[l-1]}\right)$$

Thus, we see, that under the applied assumptions the variance of the activations stays constant from layer $l-1$ to l if the term $(*)$ is of order one. In contrast to the scheme from Equation 10, this would require:

$$\mathbf{W}^{[l]} \sim \mathrm{U}\left[-\sqrt{\frac{3}{n_{l-1}}}, \sqrt{\frac{3}{n_{l-1}}}\right]$$

Equation 11

The reason for the additional factor $\sqrt{3}$ is because a uniform distribution $\mathrm{U}[-1,1]$ has the variance $1/3$. This explains the decrease of the activations with increasing layer index l in Figure 118[76].
X. Glorot and J. Bengio [11] pushed their analysis further to understand the change in the variance of the gradients $\frac{\partial L}{\partial \mathbf{A}^{[l]}}$ (Figure 118). When considering Equation 9 for the backpropagation we see that it is governed by a similar equation as for the forward pass:

$$\frac{\partial L}{\partial \mathbf{A}^{[l-1]}} = \left(\mathbf{W}^{[l]}\right)^{\boldsymbol{T}} \cdot \frac{\partial L}{\partial \mathbf{Z}^{[l]}}$$

The main difference to the considerations that led to Equation 11 is that the weight matrix is applied in

[75] Sometimes the initialization schemes $\mathbf{W}^{[l]} \sim \mathrm{U}[-1,1]$ or $\mathbf{W}^{[l]} \sim \mathrm{N}[0,1]$ (i.e., Gaussian normal distribution with zero mean and unit variance) are cited, which are very unfavourable for the variance of the activations and quickly lead to saturation. However, in [11] these were not considered.
[76] In fact, analysis of the standard deviations of the activations in Figure 118 for the different layers reveals that they behave indeed like $\mathrm{Var}\left(a^{[l]}\right) \approx k \cdot \mathrm{Var}\left(a^{[l-1]}\right)$ with $k = 1/1.8 \approx 1/\sqrt{3}$ as expected from Equation 11.

transposed form i.e., that rows and columns are exchanged. Thus, one can conclude that the variance of the activations stays constant from layer l to $l-1$ during the backpropagation if the following initialization scheme is applied:

$$\mathbf{W}^{[l]} \sim \mathrm{U}\left[-\sqrt{\frac{3}{n_l}}, \sqrt{\frac{3}{n_l}}\right]$$

This condition and Equation 11 cannot be fulfilled simultaneously except for the case that the weight matrix $\mathbf{W}^{[l]}$ is quadratic i.e., $n_{l-1} = n_l$. However, X. Glorot and J. Bengio [11] proposed a good compromise between the forward pass and the backpropagation which turns out to work well in practice:

$$\mathbf{W}^{[l]} \sim \mathrm{U}\left[-\sqrt{\frac{6}{n_{l-1}+n_l}}, \sqrt{\frac{6}{n_{l-1}+n_l}}\right]$$

This formula just takes the average of the number of inputs n_{l-1} and number of outputs n_l of layer l. Using this initialization scheme X. Glorot and J. Bengio [11] demonstrate that the variances of both activations and gradients stay constant over all layers (Figure 120).

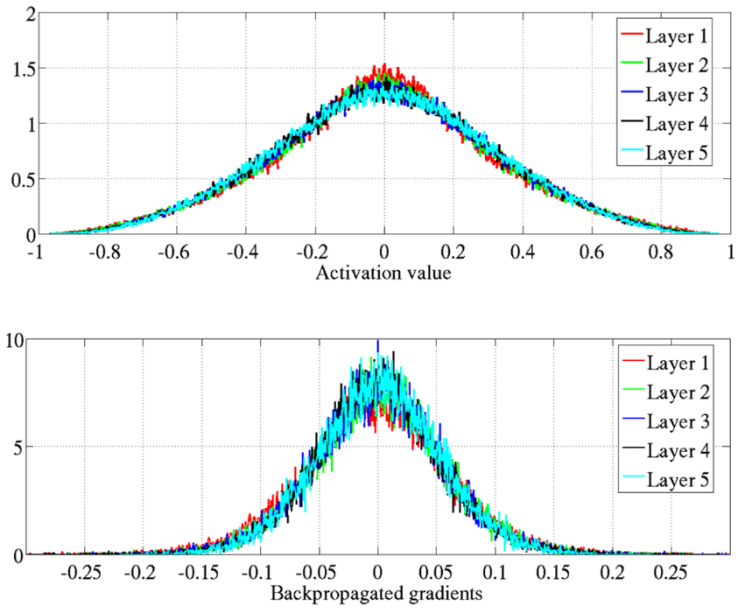


Figure 120: *After proper parameter initialization the variances of both activations and derivatives stay constant over the different layers at the beginning of the training procedure (c.f. Figure 118) [11].*

Extension to parameter initialization using a normal distribution and other activation functions lead to the following general set of parameter initialization schemes:

	Activation function	Uniform distribution [−r, r]	Normal distribution
Xavier	Logistic	$r = \sqrt{\frac{6}{n_{\text{inputs}}+n_{\text{outputs}}}}$	$\sigma = \sqrt{\frac{2}{n_{\text{inputs}}+n_{\text{outputs}}}}$
	Hyperbolic tangent	$r = 4\sqrt{\frac{6}{n_{\text{inputs}}+n_{\text{outputs}}}}$	$\sigma = 4\sqrt{\frac{2}{n_{\text{inputs}}+n_{\text{outputs}}}}$
He	ReLU (and its variants)	$r = \sqrt{2}\sqrt{\frac{6}{n_{\text{inputs}}+n_{\text{outputs}}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{inputs}}+n_{\text{outputs}}}}$

Figure 121: *Different parameter initialization schemes for the weight matrices of a NN.*

Note that the “Xavier” initialization is frequently called “Glorot”, depending on whether the first name or surname of “Xavier Glorot” is stressed.

Application of these schemes in the different DL frameworks is usually straight forward. The following Figure 122 shows its use in the tensorflow/keras API.

```python
#function to create a 'sequential' models with list of dense layers
def define_model(hidden_neurons, tensorboard_output, init = 'Glorot'):
    """
    create a sequential models of dense layers with the given list of hidden plus output (=10) neurons
    Arguments:
    hidden_neurons -- list of hidden neurons including output (e.g. [
    tensorboard_output -- index of layer with activations to be writt
    init -- type of initialsation (default 'Glorot')
    """

    if init == 'Glorot':
        initializer = tf.keras.initializers.GlorotNormal()
        print('Glorot initialiser')
    else:
        initializer = tf.keras.initializers.RandomNormal(mean=0., stddev=1.)
        print('N(0,1) initialiser')

    layers = []
    layers.append(tf.keras.layers.Flatten(input_shape=(28,28)))
    for i0 in range(len(hidden_neurons)-1):
        layers.append(tf.keras.layers.Dense(hidden_neurons[i0], activation='sigmoid', kernel_initializer=initializer))
        if i0 == tensorboard_output:
            layers.append(tf.keras.layers.Lambda(custom_summary))

    layers.append(tf.keras.layers.Dense(hidden_neurons[-1], activation="softmax", name="softmax"))
    model = tf.keras.models.Sequential(layers)

    print(model.summary())

    return model
```

pass initializer object to constructor of layer:
- **kernel_initializer** (for weights)
- **bias_initialiser** (for bias)

or a string-identifier for an initializer-method

Figure 122: *Using Xavier (or Glorot) activation in tensorflow/keras API.*

7.1.5 Batch Normalization

While the weight initialization schemes in Figure 121 work well at the beginning of the training phase there is no guarantee that the activations remain well scaled and the gradients well behaved while the training goes on. Especially for deep architectures small changes in activations may accumulate from layer to layer and push the activations into the saturation regions. Thus, a behaviour like the one shown in Figure 115 may still result.
S. Ioffe and C. Szegedy [12] analysed this problem and proposed a scheme called Batch Normalization to reduce the "*Internal Covariate Shift*". With this expression they designate the change in the distributions of the internal nodes of a deep network while training. Their main idea is to normalize the logits of each layer such that they have zero mean and unit variance. The normalization is done over each mini batch individually. However, such a brute force scaling, which pushes all logits into the linear regime of the activation function[77], would affect the representational capacity of the model (c.f. chapter 3.8). Therefore, an additional scaling and shift of the normalized logits is applied using two additional network parameters (which will be called $\boldsymbol{\gamma}^{[l]}$ and $\boldsymbol{\beta}^{[l]}$ respectively) that must be optimized during training.

In detail their approach works as follows[78]:
We consider the layer to index l with the matrix of logits $\mathbf{Z}_{\{r\}}^{[l]}$ being the input to the activation function. We use the additional index $\{r\}$ here to indicate the index of the current batch of the training samples. The matrix $\mathbf{Z}_{\{r\}}^{[l]}$ consists of column vectors $\mathbf{z}_{\{r\}}^{[l](i)}$ $(i = 1..m)$ where m denotes the size of the mini-batch. Each vector $\mathbf{z}_{\{r\}}^{[l](i)}$ has n_l entries corresponding to the number of output neurons of layer l.

$$\mathbf{Z}_{\{r\}}^{[l]} = \begin{pmatrix} \vdots & \vdots & & \vdots \\ \mathbf{z}_{\{r\}}^{[l](1)} & \mathbf{z}_{\{r\}}^{[l](2)} & \dots & \mathbf{z}_{\{r\}}^{[l](m)} \\ \vdots & \vdots & & \vdots \end{pmatrix}$$

To apply batch normalization, we calculate the average $\boldsymbol{\mu}_{\{r\}}^{[l]}$ and standard deviation $\boldsymbol{\sigma}_{\{r\}}^{[l]}$ over the m

[77] We consider the use of the sigmoid function for which $\sigma(z) \approx z$ for $z \approx 0$.
[78] For the notation see also chapter 6.3.5 on the formulation of the backpropagation for a full batch.

column vectors $\mathbf{z}_{\{r\}}^{[l](i)}$ of the mini-batch according to:

$$\boldsymbol{\mu}_{\{r\}}^{[l]} = \frac{1}{m}\sum_{i=1}^{m} \mathbf{z}_{\{r\}}^{[l](i)}$$

$$\boldsymbol{\sigma}_{\{r\}}^{[l]} = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(\mathbf{z}_{\{r\}}^{[l](i)} - \boldsymbol{\mu}_{\{r\}}^{[l]}\right)^2}$$

Now, the actual normalization of the logit matrix is as follows:

$$\hat{\mathbf{Z}}_{\{r\}}^{[l]} = \frac{\mathbf{Z}_{\{r\}}^{[l]} - \boldsymbol{\mu}_{\{r\}}^{[l]}}{\boldsymbol{\sigma}_{\{r\}}^{[l]} + \epsilon}$$

The following points should be noted:

- The average $\boldsymbol{\mu}_{\{r\}}^{[l]}$ is a column vector. I.e., the minus sign in the numerator is to be understood as a broadcast over all columns of the matrix $\mathbf{Z}_{\{r\}}^{[l]}$.
- The standard deviation $\boldsymbol{\sigma}_{\{r\}}^{[l]}$ is a column vector. I.e., the division is applied component wise and as a broadcast over all columns of the matrix $\mathbf{Z}_{\{r\}}^{[l]}$.
- The term ϵ in the denominator is for numerical stability in case that the standard deviation $\boldsymbol{\sigma}_{\{r\}}^{[l]}$ has zero entries. E.g., in tensorflow/keras API the default value used is $\epsilon = .001$.

As already mentioned above the use of the normalized logit $\hat{\mathbf{Z}}_{\{r\}}^{[l]}$ would limit the representational capacity of the NN because each logit is centred at zero and therefore close to the linear regime of the activation function. Therefore, two addition parameter vectors are introduced that rescale the logits according to:

$$\tilde{\mathbf{Z}}_{\{r\}}^{[l]} = \boldsymbol{\gamma}^{[l]} \cdot \hat{\mathbf{Z}}_{\{r\}}^{[l]} + \boldsymbol{\beta}^{[l]}$$

The values $\tilde{\mathbf{Z}}_{\{r\}}^{[l]}$ are now used as new input to the activation function.
The following points should be noted:

- Both $\boldsymbol{\gamma}^{[l]}$ and $\boldsymbol{\beta}^{[l]}$ are column vectors of size n_l and both the multiplication and addition are applied component wise and as broadcast over all columns of $\mathbf{Z}_{\{r\}}^{[l]}$.
- For the choice $\boldsymbol{\gamma}^{[l]} = \boldsymbol{\sigma}_{\{r\}}^{[l]}$ and $\boldsymbol{\beta}^{[l]} = \boldsymbol{\mu}_{\{r\}}^{[l]}$ we recover the original logit values because we simply undo the scaling operation. Thus, introducing the scaling with parameters $\boldsymbol{\gamma}^{[l]}$ and $\boldsymbol{\beta}^{[l]}$ will recover the original representational capacity of the NN.

The scaling parameters $\boldsymbol{\gamma}^{[l]}$ and $\boldsymbol{\beta}^{[l]}$ now seem to undo the normalization obtained with $\boldsymbol{\sigma}_{\{r\}}^{[l]}$ and $\boldsymbol{\mu}_{\{r\}}^{[l]}$. However, the main difference is that $\boldsymbol{\gamma}^{[l]}$ and $\boldsymbol{\beta}^{[l]}$ are *optimized* during the training process such that the model effectively learns the best scaling of the logits. It turns out that the batch normalization is extremely efficient with respect to various aspects:

- It reduces the vanishing gradient problem to a point that even saturating activation functions could be used like sigmoid and hyperbolic tangent.
- It reduces the sensitivity to proper weight initialization (c.f. chapter 7.1.4).

- It allows much higher learning rates which speeds up the learning process[79].
- It considerably improved the accuracy obtained on various classification tasks using state of the art training algorithms.
- It has a regularization effect i.e., it avoids overfitting (c.f. chapter 7.3).

Finally, the following points should be noted:

- When using batch normalization, the use of the bias when updating the logits (e.g. Equation 6) is redundant and can be eliminated:

$$\mathbf{Z}^{[l]} = \mathbf{W}^{[l]} \cdot \mathbf{A}^{[l-1]} + \cancel{\mathbf{b}^{[l]}}$$

- The mean values $\boldsymbol{\mu}_{\{r\}}^{[l]}$ and standard deviations $\boldsymbol{\sigma}_{\{r\}}^{[l]}$ are calculated during the training for each batch $\{r\}$ separately. For testing and the productive phase typically, an exponentially weighted average is used with a parameter β according to:

$$\boldsymbol{\mu}^{[l]} = (1-\beta) \cdot \boldsymbol{\mu}_{\{r\}}^{[l]} + \beta \cdot \boldsymbol{\mu}^{[l]}$$
$$\boldsymbol{\sigma}^{[l]} = (1-\beta) \cdot \boldsymbol{\sigma}_{\{r\}}^{[l]} + \beta \cdot \boldsymbol{\sigma}^{[l]}$$

- Batch normalisation has one inconvenience that it changes the backprop equations. Thus, on the one hand the derivatives for the new parameters $\boldsymbol{\gamma}^{[l]}$ and $\boldsymbol{\beta}^{[l]}$ must be added. On the other hand, the derivatives with respect to the normalization parameters $\boldsymbol{\mu}_{\{r\}}^{[l]}$ and $\boldsymbol{\sigma}_{\{r\}}^{[l]}$ must be considered[80].

- Because batch normalization is applied to the logits the layer definition in a DL frameworks should in principle follow the scheme[81]:
  - Dense layer with linear activation, no bias required
  - Batch normalization
  - Activation function

7.1.6 Non-saturating Activation Functions

Until 2010 the sigmoid activation function was considered as the optimal choice for NNs also since biological systems roughly apply sigmoidal activation. However, one of the insights of X. Glorot and Y. Bengio [11] was that the problem of vanishing gradients is in part related to the use of saturating activation functions as illustrated in Figure 116 and Figure 117. Since then, it was found that other activation functions behave much better in deep NNs. In Table 4 all relevant activation functions were given. The following main points should be considered:

- S-shaped activation functions that flatten out at larger z-magnitudes should be avoided. In case an S-shaped activation is used (e.g. in LSTM) the hyperbolic tangent should be preferred over the sigmoid function because the former is symmetric around zero and does not introduce a bias.

- Rectified Linear Unit (ReLU) is a very common choice of a non-saturating activation function due to its efficient calculation. However, it suffers from the so-called "*dying units' problem*": During training, if a neuron's weights may get updated such that the weighted sum of the neuron's inputs is negative, its output will be zero. Since the gradient at $z < 0$ is constant zero, no weight update will occur for this neuron and the neuron is likely to stay "dead".

[79] In the paper [12] 14 times fewer learning steps at even better accuracy are reported.
[80] For a detailed derivation of the corresponding formulas see e.g.: https://chrisyeh96.github.io/2017/08/28/deriving-batchnorm-backprop.html
[81] Nevertheless, many researchers argue that it is just as good or even better to place Batch Normalization after the activation layer.

- The leaky ReLU does not suffer from the dying units' problem and should be preferred over simple ReLU. However, leaky ReLU introduces an additional hyperparameter (the slope α for $z < 0$) which must be determined by hyper-parameter tuning or by setting a good default value[82].

- The Exponential Linear Unit turns out to be the optimal choice but at the cost of higher inference time. Nevertheless, the authors could show in [14] that "*ELUs lead not only to faster learning, but also to significantly better generalization performance than ReLUs on networks with more than 5 layers*".

Choosing a desired activation function is usually straight forward in DL frameworks as shown for the tensorflow/keras API in the following Figure 123.

```
model = tf.keras.models.Sequential(
            [tf.keras.layers.Dense(100, activation='sigmoid', name="hidden1"),
             tf.keras.layers.Dense(1, activation='linear', name="output")]
        )
```

Figure 123: *Choosing an activation function using the tensorflow/keras API.*

7.1.7 Gradient Clipping

[83]Deep networks often have extremely steep regions resembling cliffs because of the composite structure of the operations where weights get multiplied (c.f. chapter 7.1.3). At the cliffs, the gradient can be become very large (Figure 124 ,[1]). Therefore, it is beneficial to limit the values of the gradient by application of clipping. This is illustrated in Figure 124. On the left side the case without gradient clipping is show. GD moves along the red arrow towards the minimum, which is supposed to be in the small ravine just below the cliff. If the learning rate is still somewhat high a GD step towards the cliff may lead to a very high gradient from the cliff face and pushing the update up the cliff along the yellow arrow and finally ending up far away from the minimum. By application of gradient clipping (right) a more moderate reaction to cliff results with faster convergence to the minimum is expected.

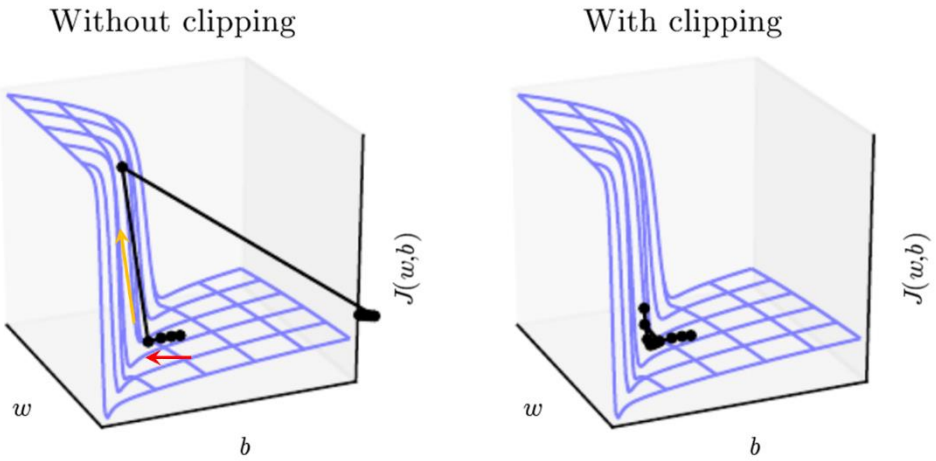


Figure 124: *In case of very steep regions of the cost function, gradient clipping should be applied (details see text).*

7.2 Advanced Optimizers

In chapters 3.4 and 3.5 we introduced the Vanilla Gradient Descent algorithms (BGD, MBGD, SGD) for the minimization of the cost function. It turns out that for the training of deep NNs these concepts may lead to slow learning in particular in flat regions. We recall that the cost functions in deep NNs are

[82] See [13] for a comparative evaluation of the different ReLUs.
[83] This topic is especially relevant for RNNs.

non-convex. Thus, multiple local minima, saddle points and flat regions may occur. Interestingly non-global minima are *not* considered as a problem. The reasons therefore are:

- Typical network structures have high number of symmetries (e.g., MLP with respect to the permutations of the units in a layer). An according number of global minima must exist.
- Other non-global minima are considered to be very rare. At a given local minimum, cost function needs to grow in all directions, a condition hard to meet in high dimensions.
- BGD and SGD can help to escape from local minima.

However major problems come from saddle points or flat regions where learning can get very slow and therefore faster optimizers are required. We will present the most common approaches first and then illustrate their performance using a set of animations.

7.2.1 Momentum Optimization

An intuitive idea for the momentum optimization is the following:
"Imagine a ball rolling down a gentle slope on a smooth surface. It starts out slowly but quickly picks up momentum until it eventually reaches terminal velocity (if there is some friction or air resistance).(…) In contrast, regular Gradient Descent will simply take small regular steps down the slope, so it will take much more time to reach the bottom" [9].
Momentum optimization computes an exponentially decaying sum (parameter β) of past gradients and moves in the direction of this sum:

$$\mathbf{m} \leftarrow \beta \cdot \mathbf{m} + \alpha \cdot \nabla_\theta J$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \mathbf{m}$$

A value of $\beta = 0.9$ is typically chosen (tensorflow/keras API) and $\mathbf{m}$ is initialized to zero.
To understand this updating scheme, we consider the gradient $\nabla_\theta J$ to be constant and determine the value of the momentum $\mathbf{m}$ after each update step t:

$$\begin{aligned}
t = 1: &\quad \alpha \cdot \nabla_\theta J \\
t = 2: &\quad \beta \cdot \alpha \cdot \nabla_\theta J + \alpha \cdot \nabla_\theta J \\
t = 3: &\quad \beta^2 \cdot \alpha \cdot \nabla_\theta J + \beta \cdot \alpha \cdot \nabla_\theta J + \alpha \cdot \nabla_\theta J \\
&\ldots \\
t = n: &\quad \beta^{n-1} \cdot \alpha \cdot \nabla_\theta J + \beta^{n-2} \cdot \alpha \cdot \nabla_\theta J + \cdots + \beta \cdot \alpha \cdot \nabla_\theta J + \alpha \cdot \nabla_\theta J = \frac{1-\beta^n}{1-\beta} \cdot \nabla_\theta J
\end{aligned}$$

Thus, if n grows large the value of β^n will tend to zero and the momentum will reach the value:

$$\mathbf{m} = \frac{1}{1-\beta} \cdot \nabla_\theta J$$

Thus, for a choice of $\beta = 0.9$ the momentum will reach 10 times the value of $\nabla_\theta J$ i.e., vanilla GD.

7.2.1.1 Nesterov Accelerated Gradient Optimization

An additional improvement of the momentum optimization as proposed by Y. Nesterov is:

$$\mathbf{m} \leftarrow \beta \cdot \mathbf{m} + \alpha \cdot \nabla_\theta J(\boldsymbol{\theta} - \beta \cdot \mathbf{m})$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \mathbf{m}$$

The idea is to evaluate the gradient not at the current parameter position $\boldsymbol{\theta}$ but slightly ahead in the direction of the momentum at position $\boldsymbol{\theta} - \beta \cdot \mathbf{m}$. The idea behind is illustrated in Figure 125 below.

Because the moment will in general point into the right direction it will be slightly more accurate to use the gradient measured a bit further in that direction.

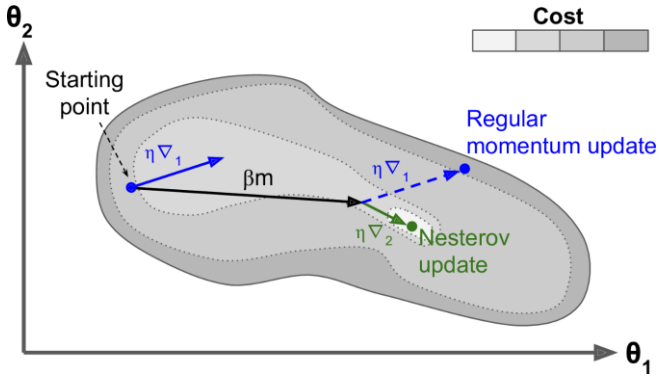


Figure 125: *Illustration of Nesterov update scheme (details see text)*[84].

7.2.2 AdaGrad Optimization

One of the problems frequently slowing down the convergence is the so-called elongated bowl problem. This occurs when the parameters θ_i have very different scales as illustrated in Figure 126 for two parameters θ_1 and θ_2. Due to the different scales the cost function is very flat in direction of θ_1 and very steep in direction of θ_2. Vanilla GD will first move down along the steepest slope i.e., in direction of θ_2 and then move slowly along the bottom of the valley towards the global minimum. It would be nice if the algorithm could detect this type of topology early on and correct its direction to point more towards the global optimum right from the beginning.

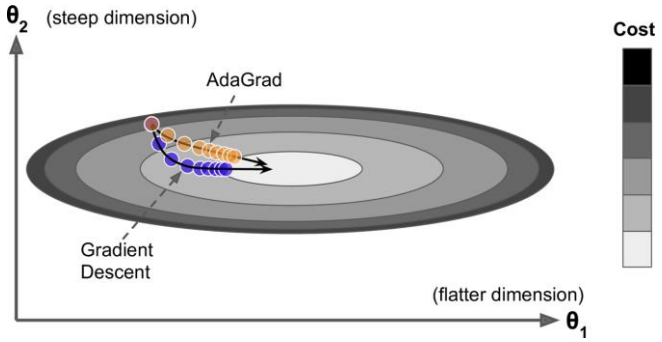


Figure 126: *The elongated bowl problem where vanilla GD may take a long time to converge while AdaGrad would speed up the convergence*[84].

The AdaGrad algorithm achieves this by scaling down the gradient vector along the steepest dimensions. The updating scheme is:

$$\mathbf{s} \leftarrow \mathbf{s} + \nabla_\theta J \cdot \nabla_\theta J$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \frac{\alpha}{\sqrt{\mathbf{s} + \epsilon}} \cdot \nabla_\theta J$$

It is important to note that all operations (multiplications, division, square root) are to be taken component wise.
In the first line the updating scheme for the vector $\mathbf{s}$ is given. It accumulates the sum of the squares of *all* past gradients of the cost function $\nabla_\theta J$ (square of $\nabla_\theta J$ to be taken component wise). If a partial derivative of the cost function $\nabla_{\theta_i} J$ with respect to the parameter θ_i is large, the corresponding component of the vector $\mathbf{s}$ denoted by s_i will grow larger and larger in each update step. In the actual parameter

[84] https://medium.com/mlearning-ai/deep-learning-optimizers-4c13d0799b4d

update – given in the second line – a component wise scaling of the gradients $\nabla_\theta J$ by $\sqrt{\mathbf{s}+\epsilon}$ is applied[85]. Thus, AdaGrad leads to a continuous decay of the learning rate, but it does so faster for steep dimensions – i.e., for larger components s_i of the vector $\mathbf{s}$ – than for dimensions with gentler slopes. This is called an adaptive learning rate. It helps to point the resulting updates more directly towards the global optimum (Figure 126). One additional benefit is that it requires much less tuning of the learning rate hyperparameter α (c.f. chapter 7.2.6).
However, AdaGrad frequently stops too early when training neural networks because the learning rate gets scaled down too much and the algorithm ends up stopping entirely before reaching the global optimum. So even though DL frameworks provide AdaGrad optimizer, it should rather not be used to train deep DL. Here we only presented it to prepare the next optimizer scheme RMSProp.

7.2.3 RMSProp Optimization

The problem with AdaGrad i.e., that it slows down too fast and ends up never converging to the global optimum, can be fixed by accumulating only the gradients from recent iterations (as opposed to all the gradients since the beginning of training). This is done by using an exponentially decaying average in addition to AdaGrad:

$$\mathbf{s} \leftarrow \beta \cdot \mathbf{s} + (1-\beta) \cdot \nabla_\theta J \cdot \nabla_\theta J$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \frac{\alpha}{\sqrt{\mathbf{s}+\epsilon}} \cdot \nabla_\theta J$$

Again, all operations (multiplications, division, square root) are to be taken component wise. A typical default value for the decay parameter is $\beta = 0.9$ (tensorflow/keras API).

7.2.4 Adam Optimization

Adam optimization combines Momentum and RMS Prop and is currently considered as the *de facto standard*. The full updating scheme is as follows:

$$\mathbf{m} \leftarrow \beta_1 \cdot \mathbf{m} + (1-\beta_1) \cdot \nabla_\theta J$$

$$\mathbf{s} \leftarrow \beta_2 \cdot \mathbf{s} + (1-\beta_2) \cdot \nabla_\theta J \cdot \nabla_\theta J$$

$$\hat{\mathbf{m}} = \frac{\mathbf{m}}{1-\beta_1}$$

$$\hat{\mathbf{s}} = \frac{\mathbf{s}}{1-\beta_2}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \frac{\alpha}{\sqrt{\hat{\mathbf{s}}+\epsilon}} \cdot \hat{\mathbf{m}}$$

Typical choices of parameter values are (tensorflow/keras API) $\beta_1 = 0.9$ and $\beta_2 = 0.999$. Looking at the lines 1, 2, and 5 of the update scheme shows the close similarity of Adam to both Momentum and RMSProp optimization. The only difference is that line 1 computes an exponentially decaying average rather than an exponentially decaying sum, but these are equivalent except for a constant factor (the decaying average is just $1-\beta_1$ times the decaying sum). Steps 3 and 4 are somewhat of technical detail: since $\mathbf{m}$ and $\mathbf{s}$ are initialized to zero, they will be biased towards zero at the beginning of training, so these two steps will help boost $\mathbf{m}$ and $\mathbf{s}$ at the beginning of training

7.2.5 Comparison of the Optimizers

The following figures illustrate the behaviour the different optimizers under different topologies that are known to pose problems for vanilla GD. The illustrations are for a cost function dependent on two parameters. The original figures are animations and can be found here[86]. The correspondences of the naming convention of the figures to our definitions are:

[85] Here the term $\epsilon \approx 10^{-8}$ is again for numerical stability in case that one of the components of $\mathbf{s}$ is zero.

- Gradient Descent (GD)
- Momentum Optimization
- Nesterov Accelerated Gradient Optimization
- AdaGrad Optimization
- Adam Optimization
- RMSProp Optimization

7.2.5.1 Motion along the steep axis of a saddle Point

The start of the optimizers is close to a saddle point with direction along the steeper of the two axes.

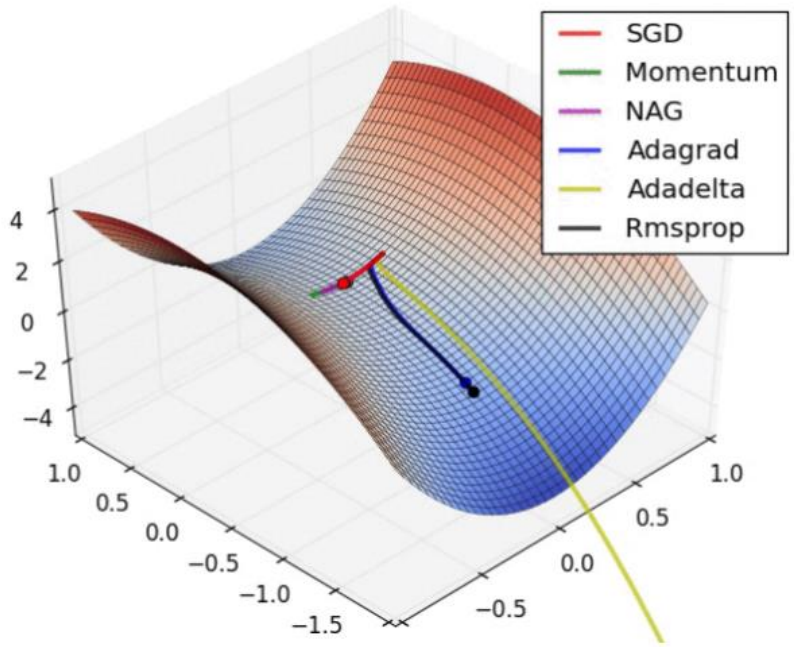


Figure 127: *Different optimizers on a saddle point (details see text)*[86].

- Optimizers that do not apply scaling based on gradient information struggle to break symmetry here:
  - GD just gets stuck on the ridge.
  - Both Momentum and Nesterov Optimization exhibits many oscillations along the ridge until they build up velocity in the perpendicular direction.
- All other optimizers that do apply scaling based on gradient information (AdaGrad, Adam and RMSProp) quickly break symmetry and begin to descent towards the optimum.

7.2.5.2 Motion in an elongated bowl topology

The geometry in Figure 128 resembles an elongated bowl with very steep and very flat parts of the cost function.

- GD behaves as expected (c.f. Figure 126) i.e., it starts off rather fast along the steep part but then slows down and fails to converge when it gets stuck on the flat bottom of the bowl.
- Momentum based techniques (Momentum and Nesterov Optimization) acquire – due to the

[86] See animations on https://imgur.com/a/Hqolp .

large initial gradient – a high velocity at the beginning which makes them shoot off and bouncing around.

- Optimizers that apply scaling based on gradient information (AdaGrad, Adam and RMSProp) proceed more like accelerated SGD and handle large gradients with more stability. Nevertheless, AdaGrad almost goes unstable – like momentum-based techniques – due to the high initial velocity.

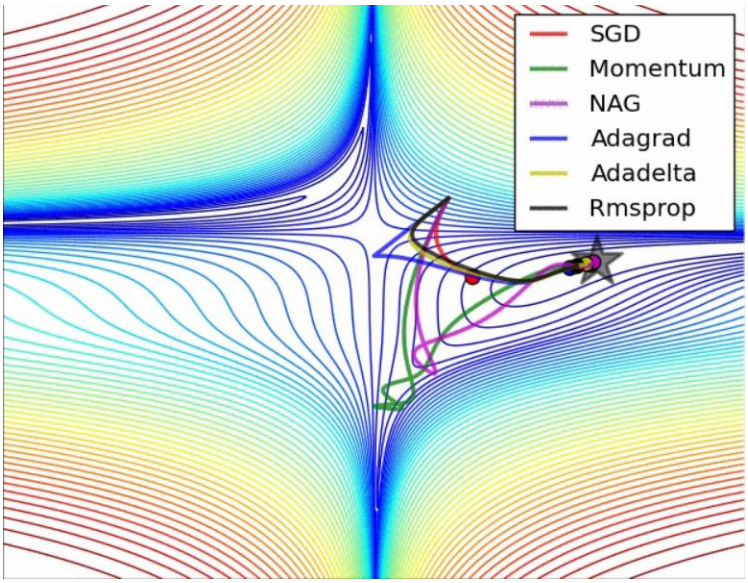


Figure 128: *Different optimizers in an elongated bowl like topoglogy[86].*

7.2.5.3 Motion along the flat axis of a saddle Point

The start of the optimizers is close to a saddle point with a direction along the flatter of the two axes.

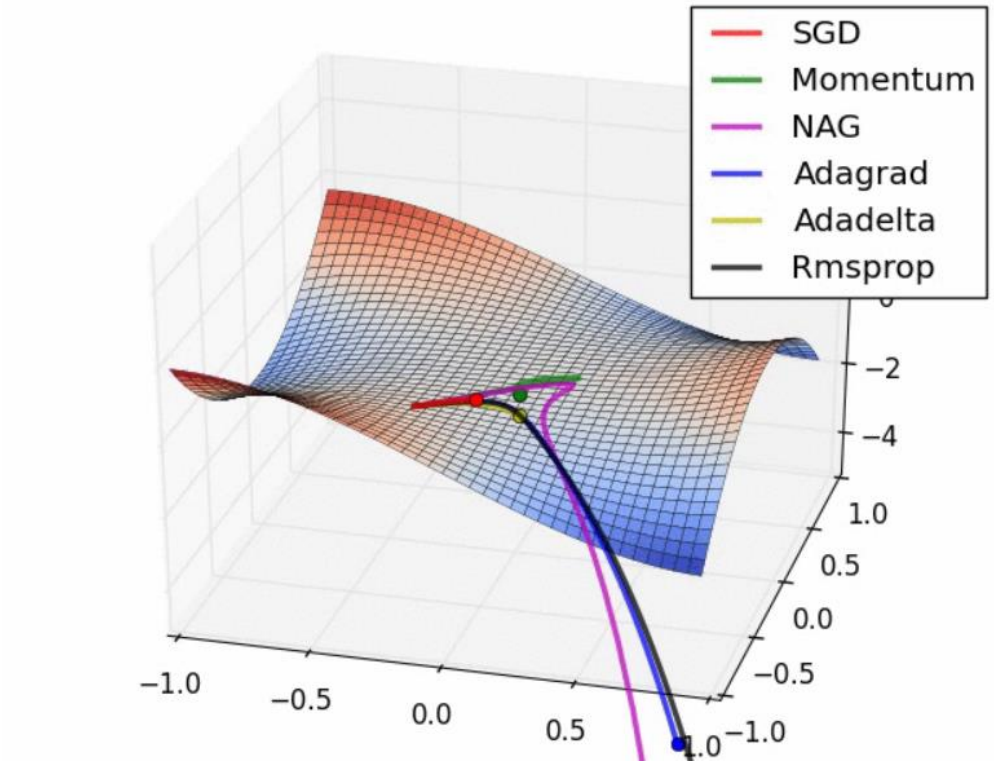


Figure 129: *Different optimizers on a saddle point[86].*

- Due to the flat region GD has problems to start off and to converge.
- Momentum based techniques (Momentum and Nesterov Optimization) acquire a high velocity and explore around, almost taking a different path.
- Optimizers that apply scaling based on gradient information (AdaGrad, Adam and RMSProp) proceed like accelerated SGD and handle the situation well.

7.2.6 Learning Rate Scheduling

The choice of a good learning rate may be difficult. If it is chosen too hight, training may diverge. If it is set too low training may eventually converge to the optimum, but it may take a long time. Therefore, learning rate scheduling i.e., the continuous reduction of the learning rate during training, is important, even in the case that adaptive learning rate optimizers like AdaGrad, Adam and RMSProp are chosen.

7.2.6.1 Predetermined piecewise constant learning rate

The learning rate is decreased according to a predefined schedule like:

Epoch	Learning Rate
1-100	0.1
101-200	0.01
201-300	0.001
...	...

Although this solution can work well, it often requires fiddling around to determine the right learning rates and epochs when to use them.

7.2.6.2 Performance scheduling

The validation error is measured every N steps. When the validation error stops dropping, the learning rate is reduced by a factor of λ.

7.2.6.3 Exponential scheduling

The learning rate is set to a function of the iteration number t e.g., according to:

$$\alpha(t) = \alpha_0 \cdot 10^{-t/T}$$

Here the learning rate will drop – starting from an initial value of α_0 – by a factor of 10 every T steps. While this works well in practice it requires the tuning of the hyperparameters α_0 and T.

7.2.6.4 Power scheduling

The learning rate is set to a function of the iteration number t e.g., according to:

$$\alpha(t) = \alpha_0 \cdot \left(1 + \frac{t}{T}\right)^{-c}$$

The idea is like exponential scheduling, but the learning rate will drop much more slowly. The hyperparameter c is typically set to 1. Again, this requires the tuning of the hyperparameters α_0 and T.

7.3 Regularisation

Deep NN typically have tens of thousands of parameters, sometimes even millions. The high number of parameters allows a large representational capacity, but this great flexibility also means that it is prone to overfitting the training set. In the following, we will present different so-called regularisation techniques that will allow to control the problem of overfitting. The following techniques will be presented:

- Weight Penalty
  These represent constraints on parameters (e.g., length of parameter vector, number of parameters) to give preference to simple models.

- Dropout
  A method to randomly drop (neutralise) neurons during training to make the solution less dependent on individual neurons.

- Early Stopping
  A method that stops training at the minimum of the cost function on the validation set.

- Data Augmentation[87]
  Methods that generate more training data with additional characteristics (e.g. symmetries) which the solution should have.

7.3.1 Weight Penalty

The loss function is modified to give preference to smaller or fewer weights – by adding a suitable penalty term. This is done as follows:

$$J(\boldsymbol{\theta}) = J_0(\boldsymbol{\theta}) + \lambda \cdot \Omega(\mathbf{W}) \quad (\lambda \geq 0)$$

Here the parameters $\boldsymbol{\theta} = (W_{ji}^{[l]}, b_j^{[l]})$ represent the set of weights and biases in the network with layer index $[l]$. $J_0(\boldsymbol{\theta})$ is the original cost function, $\lambda \geq 0$ is a regularization parameter and $\Omega(\mathbf{W})$ is a penalty term that favours models with smaller (or fewer) weights. Typically, the biases are not included into the penalty term. Two forms of penalties are popular:

- L_1-Regularization
- L_2-Regularization

$$\Omega(\mathbf{W}) = \|\mathbf{W}\|_1 = \sum_{j,i,l} \left| W_{ji}^{[l]} \right|$$

$$\Omega(\mathbf{W}) = \frac{1}{2} \cdot \|\mathbf{W}\|_2 = \frac{1}{2} \cdot \sum_{j,i,l} \left(W_{ji}^{[l]} \right)^2$$

When applying the optimisation with gradient descent, the derivatives of the loss are modified by the penalty term in a direction to make the loss and the penalty term smaller. The gradient of the additional penalty term can be determined in straight forward manner allowing an intuitive interpretation of the weight penalty schemes. We will do this in the following two sections.

7.3.1.1 Gradient Descent with L_2-Regularization

We determine the gradient of the cost function:

$$\nabla J(\boldsymbol{\theta}) = \nabla \left(J_0(\boldsymbol{\theta}) + \lambda \cdot \frac{1}{2} \cdot \|\mathbf{W}\|_2 \right) = \nabla J_0(\boldsymbol{\theta}) + \lambda \cdot \mathbf{W}$$

Application of the GD update rule gives:

$$\mathbf{W} \leftarrow \mathbf{W} - \alpha \cdot \nabla J(\boldsymbol{\theta}) = \mathbf{W} - \alpha \cdot (\nabla J_0(\boldsymbol{\theta}) + \lambda \cdot \mathbf{W})$$

or:

$$\mathbf{W} \leftarrow \underbrace{(1 - \alpha \cdot \lambda)}_{(1)} \cdot \mathbf{W} - \underbrace{\alpha \cdot \nabla J_0(\boldsymbol{\theta})}_{(2)}$$

The term (2) represents the original term while (1) corresponds to a new term, which modifies the learning rule to multiplicatively shrink the weight vector by a constant factor before performing the

[87] This topic will be treated in the second part of the course in the context of CNNs.

usual gradient update.

7.3.1.2 Gradient Descent with L_1-Regularization

We determine the gradient of the cost function:

$$\nabla J(\boldsymbol{\theta}) = \nabla(J_0(\boldsymbol{\theta}) + \lambda \cdot \|\mathbf{W}\|_1) = \nabla J_0(\boldsymbol{\theta}) + \lambda \cdot \operatorname{sign}(\mathbf{W})$$

Here the sign-function is applied elementwise to the weights $\mathbf{W}$.
The following Figure 130 illustrates the application of the L_1-regularization. For linear regression problems the addition of the L_1 penalty term is also referred to as LASSO regression. The ellipses show the contours of $J_0(\boldsymbol{\theta})$ i.e., the cost function without regularization. The regularization term $\sum_{j,i,l}\left|W_{ji}^{[l]}\right|$ will read like $|\beta_1| + |\beta_2|$ for the illustrated case of two parameters β_i. The contours of constant penalty take the form of a diamond. Application of the L_1-regularization will now push the original minimum (blue) towards the β_2 axis (red) thus eliminating the parameter β_1. In fact, L_1-regularization automatically performs a feature selection and leads to a sparser solution (as compared to no or L_2-regularization).

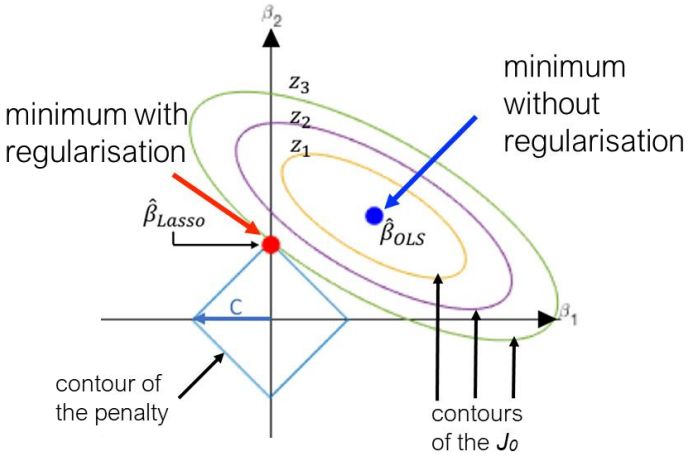


Figure 130: *Illustration of the L_1-regularization (details see text).*

7.3.2 Dropout

This is the most popular regularisation technique for deep neural networks because it is highly successful. Even state-of-the-art neural networks got a 1–2% accuracy boost simply by adding dropout. it was proposed by G. E. Hinton in 2012 [15] and further detailed in a paper by Nitish Srivastava et al. [16]. The Algorithm works as follows (Figure 131):

- At each training step, each neuron (including input neurons, but excluding output neurons) has a probability p (so-called dropout rate) of being ignored during this step.
- The hyperparameter p is typically set to 50% for hidden, and 20% for input units.
- For testing or in production, neurons don't get dropped, but weights or outputs are corrected by the so-called keep probability $1 - p$ (to correct for the higher number of connections as compared to the training phase).

It might appear quite surprising that Dropout works for the regularization of NN. To put it in the words of A. Géron [9]:
"*Would a company perform better if its employees were told to toss a coin every morning to decide whether or not to go to work? Well, who knows; perhaps it would! The company would obviously be forced to adapt its organization; it could not rely on any single person to fill in the coffee machine or perform any other critical tasks, so this expertise would have to be spread across several people. Employees would have to learn to cooperate with many of their co-workers, not just a handful of them.*

The company would become much more resilient. If one person quit, it wouldn't make much of a difference."

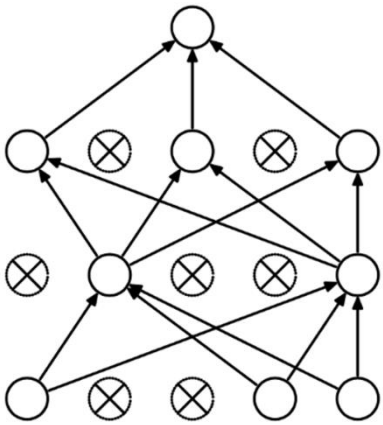


Figure 131: *Illustration of Dropout in a neural network.*

The implications of applying Dropout on NN are:

- Simpler models with less units are considered during training.
- Models learn to be less dependent on single units and units cannot co-adapt with their neighbouring units; they must be as useful as possible on their own.
- A more robust network is obtained that generalises better.

Another way to understand the power of Dropout is to consider the training process as an ensemble average over many sub-nets. If we consider the number of neurons for which Dropout may be applied to be N we can form 2^N subnets, because each neuron can be dropped or not independently. After 10'000 training steps essentially an ensemble average over 10'000 different – even though not independent – sub-networks is formed.

The following points concerning the implementation of Dropout should be noted:

- It is computationally very cheap:
  - It requires only drawing for each unit a random binary number leading to $O(n)$ additional computations per update step (mini-batch)
  - It requires only $O(n)$ additional memory to store these binary numbers for the back-propagation.
  - No additional cost occur at test time or in production.

- It is very versatile:
  - It does not significantly limit the type of model or training procedure that can be used. Thus, it is applicable to MLP, CNN or RNNs using any optimizer.
  - It can be combined with other regularisation techniques.

- It nevertheless reduces the representational capacity:
  - As a regularisation technique, dropout reduces the effective capacity of a model. Possibly, the capacity of model needs to be increased. Training of these larger models have an impact on performance.
  - For very large datasets, regularisation implies little reduction in generalisation error. In these cases, the computational cost of using dropout and larger models may outweigh the benefit of regularisation.

7.3.3 Early Stopping

The concept of early stopping is related to the analysis of the training curves in particular the observation of the training and the validation or test error (c.f. chapter 4.1). For early stopping the training is stopped when the validation or test error begins to increase again while training error still decreases.

Figure 132: *lustration of early stopping to be applied when the validation or test error begins to increase again while the training error still decreases.*

The application of early stopping works as follows:

- Run the optimisation algorithm to train the model - simultaneously compute the validation set error.
- Store a copy of the model parameters if the validation set error improves.
- Iterate until validation set error stops improving (e.g. has not improved for a given number of steps).
- Return the parameters where the smallest validation set error is observed.

While the concept of early stopping seems intuitively sound it is not immediately clear why it is a regularization method. The following points try to illustrate this.

- The training time (number of training steps/epochs) can be considered as hyper-parameter.
- It controls the *effective capacity* of the model by determining the number of steps it can take to fit the training set.
    - It restricts the volume in parameter space to be searched – it can only move a limited number of steps T away from the initial parameters.
    - It is in fact equivalent to L_2-regularisation in the case of a linear model with a quadratic error function [1].

Furthermore, the following points should be noted:

- It is efficient and non-intrusive:
    - The cost relates to repeated calculations of the validation set error and keeping a copy of the recent model parameters. The first can easily be parallelised.
    - Almost no change to the code needed.

- It is easy to combine with other regularisation techniques.
    - Typically, the best generalisation does not occur at a local minimum of the training objective.

8 Annex

8.1 Unbalanced Datasets

In practice many classification problems are facing the problem of unbalanced classes. Examples may be:

- Fraud prediction: Frauds are less frequent than genuine transactions[88]
- Natural disaster predictions: Disasters are much less frequent than normal events
- Pathology detection: Few examples of healthy patient at training time while healthy patient are way more frequent at inference time

We already discussed the problem of unbalanced datasets in the context of the confusion matrix and the use of accuracy as performance measure in chapter 5.1. Here we want to deepen our insight on this issue.

8.1.1 Bayesian approach on Classification

From mathematical statistics we know Bayes rule which tells us to elect as winning category the one having the largest *a posteriori* probability. Doing so we are guaranteed to maximise the accuracy. This can be formalized as follows:
Suppose we have a feature $\mathbf{x}$ e.g., our MNIST images and we want to perform a classification into classes C_k e.g., classes of digits. We may perform a binary classification problem with e.g., digit '5' against all other digits or – using a Softmax output layer – perform a multiclass classification in all ten digits at a time. In the former case the dataset will be highly unbalanced because only $\frac{1}{10}$ of all data represent the digit '5' and $\frac{9}{10}$ the rest. In the latter case the dataset would be well balanced with all ten classes being equally well represented according to $\frac{1}{10}$. We represent this so-called *a priori* probability i.e., the probability of the occurrence of class k by $P(C_k)$. E.g., in the former case we would have:

$$P(C_5) = \frac{1}{10} \quad ; \quad P(C_{\neg 5}) = \frac{1}{10}$$

Here the symbol $\neg 5$ represents the class of 'not 5'. The rule of Bayes now states that the a posteriori probability $P(C_k|\mathbf{x})$ for occurrence of the class C_k given the observation $\mathbf{x}$ can be calculated by:

$$P(C_k|\mathbf{x}) = \frac{p(\mathbf{x}|C_k) \cdot P(C_k)}{p(\mathbf{x})}$$

Equation 12

Here $p(\mathbf{x}|C_k)$ is the so-called likelihood i.e., the probability to observe $\mathbf{x}$ given the class C_k. Finally, $p(\mathbf{x})$ is the so-called evidence i.e., the overall probability to observe $\mathbf{x}$ independent of any class. We will illustrate the application of this formula using several examples with increasing complexity.

8.1.2 Example 1 – Discrete Observations

We consider a very simple example of two types of food trucks, called C_{red} and C_{blue} according to their respective colour (Figure 133). Both trucks transport oranges (🟠) and apples (🟢). We take randomly a fruit out of the box and observe the type $\mathbf{x}$ being orange **o** or apple **a** and want to decide whether we are in red or a blue truck. We can make a prediction because the trucks transport fixed ratios of oranges and apples. These ratios define the likelihoods $p(\mathbf{x}|C_{red})$ according to Bayes rule:

$$p(\mathbf{o}|C_{red}) = \frac{6}{8} \qquad p(\mathbf{a}|C_{red}) = \frac{2}{8}$$

[88] You may want to have a look at the data on kaggle with credit card fraud data. Only 0.172% of the data are true positives: https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud

$$p(\mathbf{o}|C_{blue}) = \frac{3}{8} \qquad p(\mathbf{a}|C_{blue}) = \frac{5}{8}$$

Thus, $p(\mathbf{o}|C_{red})$ represents the probability of finding an orange in case we are in a red truck and $p(\mathbf{a}|C_{red}) = 1 - p(\mathbf{o}|C_{red})$ is the probability of finding an apple.

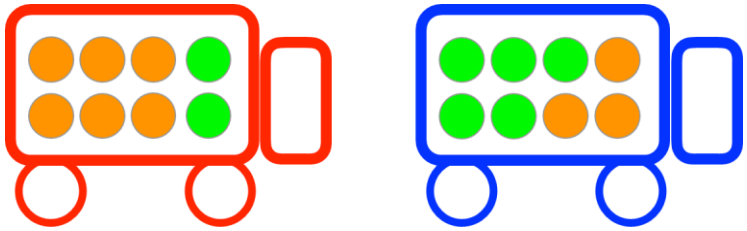


Figure 133: *Example of food trucks for application of Bayes rule.*

If we want to decide whether we are in a red or blue truck having chosen some fruit $\mathbf{x}$ and observed its type, we must – according to Bayes rule (Equation 12) –maximize the a posteriori probability of the observation. Assume that $\mathbf{x}$ is an orange $\mathbf{o}$. Bayes rule tells us:

$$p(C_{red}|\mathbf{o}) = \frac{p(\mathbf{o}|C_{red}) \cdot p(C_{red})}{p(\mathbf{o})}$$

The identical formula is true for the blue truck. It states that the a posteriori probability $p(C_{red}|\mathbf{o})$ of being in a red truck having observed an orange is given by the product of the likelihood $p(\mathbf{o}|C_{blue})$ and the a priori probability $p(C_{red})$ of red cars. The division by the evidence of finding an orange $p(\mathbf{o})$, which does depend on the class, will not change the maximization:

$$p(C_{red}|\mathbf{o}) \lessgtr p(C_{blue}|\mathbf{o}) \quad \Leftrightarrow \quad \frac{p(\mathbf{o}|C_{red}) \cdot p(C_{red})}{p(\mathbf{o})} \lessgtr \frac{p(\mathbf{o}|C_{blue}) \cdot p(C_{blue})}{p(\mathbf{o})}$$

$$\Leftrightarrow \quad p(\mathbf{o}|C_{red}) \cdot p(C_{red}) \lessgtr p(\mathbf{o}|C_{blue}) \cdot p(C_{blue})$$

If the probabilities for red and blue trucks – i.e., the a priori values $p(C_{red})$ and $p(C_{blue})$ – are equal, we would decide on being in a red car under the observation of an orange, because the likelihood of oranges is larger for red cars:

$$p(C_{red}) = p(C_{blue}) \quad \Rightarrow \quad p(\mathbf{o}|C_{red}) \cdot p(C_{red}) > p(\mathbf{o}|C_{blue}) \cdot p(C_{blue})$$

$$\Rightarrow \quad p(C_{red}|\mathbf{o}) > p(C_{blue}|\mathbf{o})$$

However, if the a priori values $p(C_{red})$ and $p(C_{blue})$ are not equal the situation may be different. Lets assume the following case:

$$p(C_{red}) = \frac{1}{5} < p(C_{blue}) = \frac{4}{5} \quad \Rightarrow \quad p(\mathbf{o}|C_{red}) \cdot p(C_{red}) = \frac{6}{8} \cdot \frac{1}{5} <= \frac{3}{8} \cdot \frac{4}{5} = p(\mathbf{o}|C_{blue}) \cdot p(C_{blue})$$

$$\Rightarrow \quad p(C_{red}|\mathbf{o}) < p(C_{blue}|\mathbf{o})$$

Thus, in case that red trucks are much less frequent we would decide on being in a blue truck due to the high a priori probability of the latter.

8.1.3 Example 2 – Continuous Observations

In this example we want to set up a classification of women vs. men based on the length of hair. While this is certainly not a very performant classifier it is nevertheless well suited to illustrate the use of Equation 12.
We observe the length of hair of a set of women C_w and men C_m (c.f. Figure 134). Our sample set

comprises $N_w = 12$ women and $N_m = 20$ men. Thus, we can determine the a priori probability of the two classes:

$$P(C_w) = \frac{12}{32} = \frac{3}{8} \quad ; \quad P(C_m) = \frac{5}{8}$$

The likelihood in Equation 12 is usually obtained by modelling the probability for observing the feature $\mathbf{x}$ given the respective class C_k. E.g., in our example we could measure the hair length of all women and men, draw two histograms and approximate these using a simple model e.g., a Gaussian distribution (Figure 135). We are still missing the evidence $p(\mathbf{x})$ to apply Equation 12. However, because the evidence does not depend on the classes it will not influence the maximation of the right-hand-side of Equation 12 and can therefore be ignored.

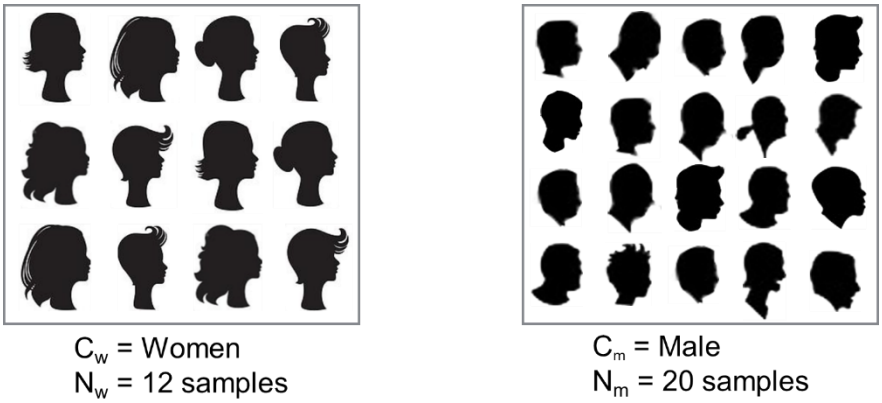


Figure 134: *A set of 12 women and 20 men are used to decide among these two classes based on the hair length.*

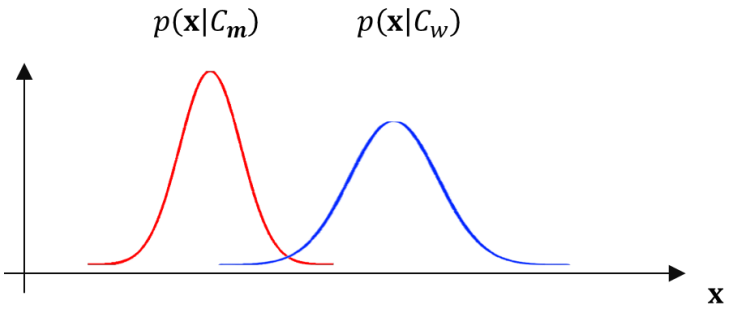


Figure 135: *The likelihood could be obtained by modelling the average hair length of men ($\boldsymbol{C_m}$) and women ($\boldsymbol{C_w}$).*

8.1.4 Strongly unbalanced set – Medical Test

In a final example we want to discuss again the case of a strongly unbalanced data set. We consider the case of a medical test T applied to a randomly chosen person within the population to detect a given disease. Obviously, we do not know in advance whether he or she is healthy or ill. The test outcome may be positive (T) or negative ($\neg T$) and we want to decide between the classes C_i and C_h denoting respectively ill and healthy persons. As for any binary classification problem we have four possible outcomes which will form the confusion table below (Table 7):

1. True Positive (TP):
   The test outcome is positive (T) and the person is ill (C_i)

2. True Negative (TN):
   The test outcome is negative ($\neg T$) and the person is healthy (C_h)

3. False Negative (FN):
   The test outcome is negative ($\neg T$) and the person is ill (C_i)

4. False positive (FP):
   The test outcome is positive (T) and the person is healthy (C_h)

We now assume the following (hypothetical) probabilities for the different outcomes[89]:

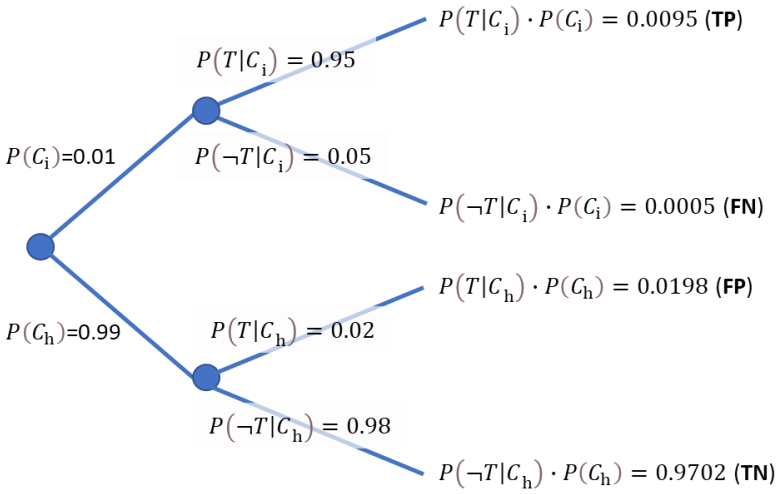


Figure 136: *Decision tree for a medical test (details c.f. text).*

We assume that 1% of the population has the given decease (a priori probability $P(C_i) = 1 - P(C_h)$) leading to a highly unbalanced data set[90]. Furthermore, we assume the following likelihood probabilities (c.f. Equation 12)[91]:

- positive test outcome for class C_i:
$$P(T|C_i) = 0.95$$
- negative test outcome for class C_i:
$$P(\neg T|C_i) = 0.05$$
- positive test outcome for class C_h:
$$P(T|C_h) = 0.02$$
- negative test outcome for class C_h:
$$P(\neg T|C_h) = 0.98$$

Based on these values the probabilities given on the right-hand side of Table 7 denoting the TP rate, FP rate, etc. can be determined and the full confusion table be evaluated (Table 7):

	T	$\neg T$	
C_i	0.0095	0.0005	0.01
C_h	0.0198	0.9702	0.99
	0.0293	0.9707	1

Table 7: *Confusion table for the medical test (details c.f. text).*

For application of the test, we assume that a test is applied to a person chosen randomly within the population i.e., we do not know whether he or she is healthy or ill. The test outcome may be positive (T) or negative ($\neg T$). The following two probabilities are now relevant to evaluation the performance of the test:

Probability, that a person is ill, in case the test outcome is positive:

[89] It is convenient to represent the different cases in form of a decision tree.
[90] Typically, this is much worse in medical tests as diseases might be much less frequent then 1%.
[91] These could be determined based on a test series with healthy and ill persons.

$$P(C_i|T) =^{(1)} \frac{P(T|C_i)\cdot P(C_i)}{p(T)} =^{(2)} \frac{P(T|C_i)\cdot P(C_i)}{P(T|C_\mathrm{i})\cdot P(C_\mathrm{i}) + P(T|C_\mathrm{h})\cdot P(C_\mathrm{h})} = \frac{0.0095}{0.0095 + 0.0198} = 0.32$$

At the equal sign $=^{(1)}$ we applied Bayes rule (Equation 12) for $=^{(2)}$ we added the two possibilities for a positive test outcome $p(T)$, one in the upper branch of the decision tree for an ill person and the other one in the lower branch for a healthy person (Figure 136).

Probability, that a person is healthy, in case the test outcome is negative:

$$P(C_2|\neg T) = \frac{P(\neg T|C_2)\cdot P(C_2)}{p(\neg T)} = \frac{P(\neg T|C_2)\cdot P(C_2)}{P(\neg T|C_2)\cdot P(C_2) + P(\neg T|C_1)\cdot P(C_1)} = \frac{0.9702}{0.9702 + 0.0005} = 0.9995$$

Similar transformations as above were applied to obtain the result.
Furthermore, we determine precision, recall and accuracy:

Recall: $P(T|C_1) = 0.95$

Precision: $P(C_1|T) = 0.32$

Accuracy: 0.9797

The following two probabilities are now relevant to evaluation the performance of the test:
shall be evaluated in terms of its performance in deciding denoting patients with or without a certain decease (i.e., ill vs. healthy).
For an ill person the test outcome should be positive (T) and negative for a healthy one ($\neg T$).

Thus, the tests reveal that T gives the correct outcome for an ill person with 95% ($P(T|C_1)$) and 98% for a healthy person ($P(\neg T|C_2)$).
The decision tree in Figure 136 furthermore allows to evaluate the combined probabilities e.g., that a given person is ill *and* that the test outcome is positive ($P(T|C_1)\cdot P(C_1) = 0.0095$). And similar for the other three cases.

8.2 Acknowledgment

These lecture notes are to a large extend based on the lecture slides prepared by J. Hennebert and M. Melchior for the module TSM-DeLearn held in Zurich. We are grateful to them for the possibility to use them.

8.3 Reference

[1] Ian Goodfellow, Yoshua Bengio, Aaron Courville, “Deep Learning”, 2015, (Weblink: https://www.deeplearningbook.org)
[2] S. Legg; M. Hutter (2007). “Universal Intelligence: A Definition of Machine Intelligence”. Minds and Machines. 17 (4): 391–444 (Weblink: https://ui.adsabs.harvard.edu/abs/2007arXiv0712.3329L).
[3] Suwajanakorn, Supasorn, Steven M. Seitz, and Ira Kemelmacher-Shlizerman. “Synthesizing obama: learning lip sync from audio.” ACM Transactions on Graphics (ToG) 36.4 (2017): 1-13 (Weblink: https://dl.acm.org/doi/pdf/10.1145/3072959.3073640).
[4] K. Murphy, “Machine Learning – A Probabilistic Perspective”, 2012.
[5] T. Mitchell, “Machine Learning”, 1997.
[6] DL-notations.pdf
[7] Cybenko, George. "Approximation by superpositions of a sigmoidal function." Mathematics of control, signals and systems 2.4 (1989): 303-314. (Weblink: https://link.springer.com/content/pdf/10.1007/BF02551274.pdf)
[8] K. Murphy, “Machine Learning – A Probabilistic Perspective”, 2012.
[9] A. Géron, “Hands-On Machine Learning with Scikit-Learn and TensorFlow”, 2017.
[10] Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors." nature 323.6088 (1986): 533-536.

(Weblink: https://ui.adsabs.harvard.edu/abs/1986Natur.323..533R)
[11] Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." Proceedings of the thirteenth international conference on artificial intelligence and statistics. JMLR Workshop and Conference Proceedings, 2010.
(Weblink http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf)
[12] Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." International conference on machine learning. PMLR, 2015.
(Weblink http://proceedings.mlr.press/v37/ioffe15.pdf)
[13] Xu, Bing, et al. "Empirical evaluation of rectified activations in convolutional network." arXiv preprint arXiv:1505.00853 (2015).
(Weblink https://arxiv.org/pdf/1505.00853.pdf?ref=https://githubhelp.com)
[14] Clevert, Djork-Arné, Thomas Unterthiner, and Sepp Hochreiter. "Fast and accurate deep network learning by exponential linear units (elus)." arXiv preprint arXiv:1511.07289 (2015).
(Weblink https://arxiv.org/abs/1511.07289)
[15] Hinton, Geoffrey E., et al. "Improving neural networks by preventing co-adaptation of feature detectors." arXiv preprint arXiv:1207.0580 (2012).
(Weblink https://arxiv.org/pdf/1207.0580.pdf)
[16] Srivastava, Nitish, et al. "Dropout: a simple way to prevent neural networks from overfitting." The journal of machine learning research 15.1 (2014): 1929-1958.
(Weblink https://www.jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf)

8.4 List of Abbreviations

AI	Artificial Intelligence
DL	Deep Learning
NN	Neural Network
ML	Machine Learning
CV	Computer Vision
OCR	Optical Character Recognition
NLP	Natural Language Processing
CNN	Convolutional Neural Network
RNN	Recurrent Neural Network
ASR	Automatic Speech Recognition
GPU	Graphics Processing Unit
ANN	Artificial Neural Networks
LTU	Linear Threshold Unit
SVN	Support Vector Machine
MLP	Multi-Layer Perceptron
GD	Gradient Descent
MSE	Mean Square Error
CE	Cross Entropy
MLE	Maximum Likelihood Estimator
BGD	Batch Gradient Descent
MBGD	Mini Batch Gradient Descent
SGD	Stochastic Gradient Descent
TP,FP,TN,FN	True Positive, False Positive, True Negative, False Negative