

# 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 2024

## Table of Content

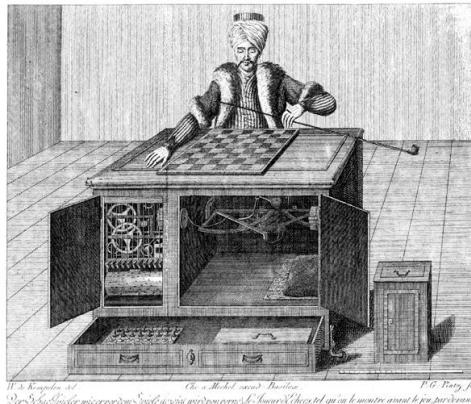
<b>1</b>	<b>Introduction to Artificial Intelligence and Deep Learning .....</b>	<b>5</b>
<b>1.1</b>	<b>Successful Applications of Machine Learning and Deep Learning.....</b>	<b>5</b>
1.1.1	Computer Vision .....	6
1.1.2	Natural Language Processing .....	8
<b>1.2</b>	<b>Definition of Deep Learning .....</b>	<b>12</b>
1.2.1	Definition of Machine Learning .....	12
1.2.2	Definition of Deep Learning .....	17
<b>2</b>	<b>First Neural Networks – a Historical Perspective .....</b>	<b>21</b>
<b>2.1</b>	<b>Biological Neural Systems .....</b>	<b>21</b>
<b>2.2</b>	<b>Artificial Neurons .....</b>	<b>22</b>
2.2.1	McCulloch-Pitts Neuron .....	22
2.2.2	Rosenblatt's Perceptron .....	24
<b>2.3</b>	<b>Artificial Neural Networks .....</b>	<b>28</b>
2.3.1	A first Neural Network: Single Layer LTUs .....	28
2.3.2	Multi-Layer Perceptron .....	29
<b>3</b>	<b>Learning and Optimisation.....</b>	<b>31</b>
<b>3.1</b>	<b>MNIST Dataset .....</b>	<b>32</b>
<b>3.2</b>	<b>Data Preparation .....</b>	<b>33</b>
3.2.1	Split of data in Training and Test Set .....	33
3.2.2	Data Normalisation – Scaling and Centring.....	33
<b>3.3</b>	<b>Generalised Perceptron .....</b>	<b>36</b>
3.3.1	Sigmoid Activation Function .....	36
<b>3.4</b>	<b>Gradient Descent .....</b>	<b>37</b>
3.4.1	Mean Squared Error Cost Function .....	37
3.4.2	General formulation of Gradient Descent .....	37
3.4.3	GD Update for MSE Cost of generalised Perceptron .....	39
3.4.4	Cross Entropy Cost Function.....	43
3.4.5	GD Update for CE Cost of generalised Perceptron.....	45
3.4.6	Differences between MSE and CE Cost GD Update .....	46
3.4.7	Summary of Results on binary Classification .....	47
<b>3.5</b>	<b>Performance Measures .....</b>	<b>49</b>
3.5.1	Confusion Matrix .....	49
3.5.2	Confusion Table.....	50
3.5.3	Precision-Recall- and ROC-Curves .....	53
<b>3.6</b>	<b>Bias and Variance of Model, Overfitting and Model Selection .....</b>	<b>54</b>
3.6.1	Bias and Variance.....	54
3.6.2	Model Selection Process .....	61
<b>3.7</b>	<b>Extension of Gradient Descent – Stochastic and Mini Batch GD .....</b>	<b>65</b>
3.7.1	Stochastic Gradient Descent .....	65
3.7.2	Mini-Batch Gradient Descent.....	68
<b>3.8</b>	<b>Multi-Class Classification and Softmax Activation .....</b>	<b>70</b>
3.8.1	GD Update with Softmax .....	72
<b>4</b>	<b>Deep Neural Networks .....</b>	<b>76</b>
<b>4.1</b>	<b>Curse of Dimensionality .....</b>	<b>76</b>
<b>4.2</b>	<b>A note about activation functions .....</b>	<b>79</b>
<b>4.3</b>	<b>Representational Capacity of the Multi-Layer Perceptron.....</b>	<b>81</b>
4.3.1	Decision Boundaries produced by MLPs.....	81
4.3.2	Universal Approximation Theorem .....	84

<b>4.4 Computational Graph .....</b>	<b>88</b>
<b>4.5 Backpropagation.....</b>	<b>92</b>
4.5.1 An illustrative Example .....	92
4.5.2 Backprop through a Single MLP Layer .....	94
4.5.3 General Formulation of Backpropagation in Matrix Notation.....	96
4.5.4 Formulation of Backpropagation for full Batch.....	97
<b>4.6 Results for MLP with one Hidden Layer .....</b>	<b>100</b>
<b>5 Improved Strategies for Training Deep Networks .....</b>	<b>105</b>
<b>5.1 Vanishing and Exploding Gradients .....</b>	<b>105</b>
5.1.1 Saturation of Activations .....	105
5.1.2 Changing Variance of Activations and Gradients .....	108
5.1.3 Multiplicative Structure of Backpropagation .....	109
5.1.4 Xavier and He Initialization of Network Parameters .....	111
5.1.5 Batch Normalization.....	113
5.1.6 Non-saturating Activation Functions.....	115
5.1.7 Gradient Clipping .....	116
<b>5.2 Advanced Optimizers .....</b>	<b>116</b>
5.2.1 Momentum Optimization .....	117
5.2.2 AdaGrad Optimization .....	118
5.2.3 RMSProp Optimization .....	119
5.2.4 Adam Optimization .....	119
5.2.5 Comparison of the Optimizers .....	119
5.2.6 Learning Rate Scheduling .....	122
<b>5.3 Regularisation .....</b>	<b>122</b>
5.3.1 Weight Penalty.....	123
5.3.2 Dropout .....	126
5.3.3 Early Stopping.....	128
<b>6 Deep Learning Frameworks .....</b>	<b>130</b>
<b>6.1 Low-level view on PyTorch and Tensorflow .....</b>	<b>132</b>
6.1.1 PyTorch.....	132
6.1.2 TensorFlow .....	133
<b>6.2 Introduction to PyTorch .....</b>	<b>133</b>
6.2.1 nn-Library and Sequential Container .....	133
6.2.2 DataSet and DataLoader .....	135
6.2.3 TensorBoard for Result Visualization .....	139
6.2.4 Summary: DL-model using PyTorch.....	144
<b>6.3 Introduction to Keras API for TensorFlow .....</b>	<b>146</b>
<b>7 Convolutional Neural Networks.....</b>	<b>150</b>
<b>7.1 Visual Object Classification with MLP .....</b>	<b>150</b>
<b>7.2 Convolutions in Image Processing .....</b>	<b>153</b>
<b>7.3 General Architecture of CNNS.....</b>	<b>156</b>
7.3.1 Convolutional Layer .....	156
7.3.2 Pooling Layer .....	163
7.3.3 Overall CNN Architecture .....	164
<b>8 Annex .....</b>	<b>170</b>
<b>8.1 Backpropagation for Batch Normalization Layer .....</b>	<b>170</b>
8.1.1 Formulas for a single feature .....	171
8.1.2 Formulas for entire feature vector.....	173
<b>8.2 Unbalanced Datasets.....</b>	<b>174</b>
8.2.1 Bayesian approach on Classification .....	174
8.2.2 Example 1 – Discrete Observations .....	175
8.2.3 Example 2 – Continuous Observations .....	176

8.2.4	Strongly unbalanced set – Medical Test.....	176
<b>8.3</b>	<b>Acknowledgment .....</b>	<b>178</b>
<b>8.4</b>	<b>Reference.....</b>	<b>178</b>
<b>8.5</b>	<b>List of Abbreviations .....</b>	<b>180</b>

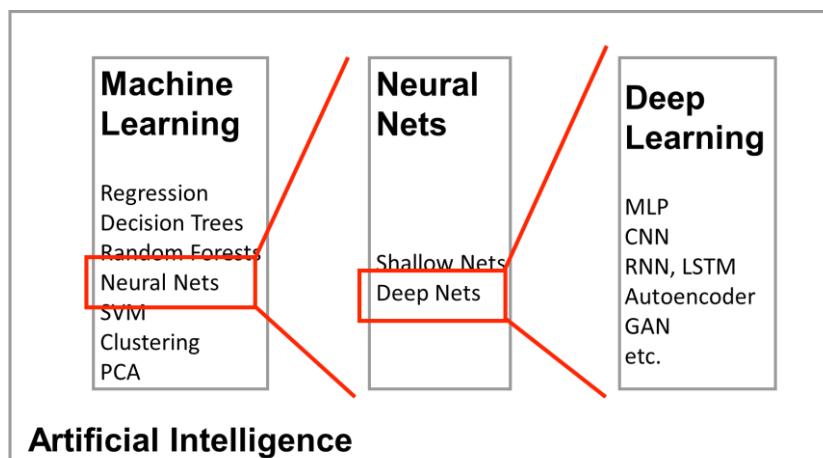
## 1 Introduction to Artificial Intelligence and Deep Learning

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



**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<sup>3</sup>. 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 a 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

<sup>4</sup>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<sup>5</sup>.

<sup>1</sup> While we certainly have an intuitive idea of “intelligence” its formal definition is far from being obvious [2].

<sup>2</sup> [https://en.wikipedia.org/wiki/Mechanical\\_Turk](https://en.wikipedia.org/wiki/Mechanical_Turk).

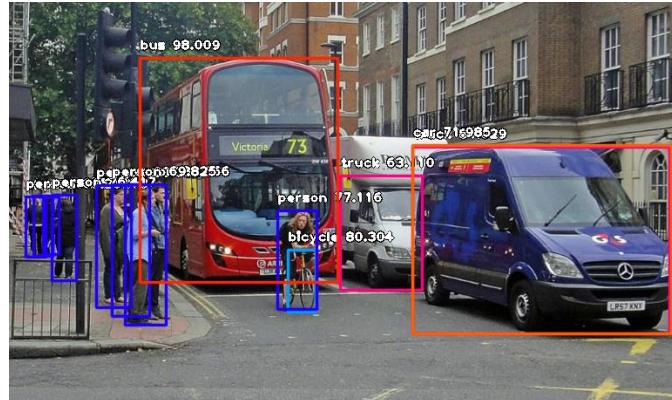
<sup>3</sup> We will give some historical reference in chapter 2.

<sup>4</sup> Due to the current rapid development of AI, any summary of AI applications is inevitably outdated.

<sup>5</sup> 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<sup>6</sup>.

#### 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<sup>7</sup> 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<sup>8</sup>.

<sup>6</sup> <https://towardsdatascience.com/object-detection-with-10-lines-of-code-d6cb4d86f606>

<sup>7</sup> <http://yann.lecun.com/exdb/mnist/>

<sup>8</sup> <https://github.com/akshaypunwatkar/Image-captioning-on-flickerdata>

### 1.1.1.3 Self-Driving cars



**The Autonomous-Car Chaos of the 2004 Darpa Grand Challenge**

The self-driving vehicles smashed, burned, flipped, and tipped. But the ambitious race through the Mojave launched an industry.

**Figure 6:** Article on the “Chaos” of the 2004 Darpa Grand Challenge<sup>9</sup>.

While the 2004 Darpa Grand Challenge<sup>10</sup>, 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<sup>11</sup>.

### 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<sup>12</sup>.

<sup>9</sup> <https://www.wired.com/story/autonomous-car-chaos-2004-darpa-grand-challenge/>

<sup>10</sup> [https://en.wikipedia.org/wiki/DARPA\\_Grand\\_Challenge\\_\(2004\)](https://en.wikipedia.org/wiki/DARPA_Grand_Challenge_(2004))

<sup>11</sup> [https://youtu.be/\\_EoOvVkJMo](https://youtu.be/_EoOvVkJMo)

<sup>12</sup> <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<sup>13</sup>. 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<sup>14</sup>.



**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<sup>15,16</sup>. 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).

<sup>13</sup> [https://www.youtube.com/watch?v=MVBe6\\_o4cMI](https://www.youtube.com/watch?v=MVBe6_o4cMI)

<sup>14</sup> <https://youtu.be/UOEIH2l9z7c>

<sup>15</sup> <https://translate.google.com>

<sup>16</sup> <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 permuted:

Gmäess eneir Sutide eneir elgnihcesn Uvinisterät ist es nchit witihcg, in wlecehr Rneflogheie die Bsta-chuebn in eneim Wrot snid, das ezniige was wcthig ist, ist, dass der estre und der leztte Bstabchue an der ritihcegn Pstoion snid. Der Rset knan ein ttoaelr Bsinöldn sien, tedztorm knan man ihn onhe Pe-moblre lseen. Das ist so, wiel wir nciht jeedn Bstachuebn enzelin leesn, snderon das Wrot als gse-a-tems.

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<sup>17</sup>.

### 1.1.2.4 Large Language Models

In the recent years Large Language Models (LLM) like ChatGPT<sup>18</sup> have generated a great deal of interest. These models often based on the so-called transformer architecture [4] have the ability to achieve general-purpose language understanding and generation. The interaction with LLM-based chatbots seem to be very “human-like”. The following example given also illustrates, that the results are continuously improving. While the first answer – from mid-2023 – to a fairly complex problem from basic Physics was rather nonsense, the second – beginning of 2024 – was completely correct.

<sup>17</sup> <https://stackabuse.com/text-classification-with-python-and-scikit-learn/>

<sup>18</sup> <https://chat.openai.com>

ChatGPT 3.5:

Question asked:

KL

**You**

- Consider the following problem. A wagon of mass 10 kilograms rolls frictionlessly on a plane at the speed of 1 metre per second. The trolley is open at the top and completely tight. Since it is raining, the rainwater collects in the trolley. At the beginning, the trolley is completely empty. After 100 seconds, the speed of the trolley has halved. At what rate (kilograms per second) does the rainwater accumulate in the trolley?

First (wrong) answer:

**ChatGPT**

The rate at which the rainwater accumulates in the trolley can be found by considering the conservation of momentum principle. The principle states that the total momentum of an isolated system remains constant unless acted upon by an external force.

In this case, the trolley is the isolated system, and the rainwater is the external force. At the beginning, the trolley has a momentum of  $10 \text{ kg} \cdot 1 \text{ m/s} = 10 \text{ kg m/s}$ . After 100 seconds, the trolley has a momentum of  $10 \text{ kg} \cdot 0.5 \text{ m/s} = 5 \text{ kg m/s}$ . Since the total momentum of the system must remain constant, the change in momentum of the trolley must be equal to the momentum of the rainwater that has accumulated in the trolley.

Therefore, the rate at which the rainwater accumulates in the trolley is  $(5 \text{ kg m/s} - 10 \text{ kg m/s})/100 \text{ s} = -0.5 \text{ kg/s}$ . This means that on average, the trolley is losing 0.5 kilograms of rainwater per second.

Second (fully correct) answer:

**ChatGPT**

To solve this problem, we can apply the principle of conservation of momentum. Initially, the momentum of the system (wagon + rainwater) is equal to the momentum after the rainwater starts accumulating in the trolley.

Let's denote:

- $m_w$  as the mass of the wagon (10 kilograms)
- $v_w$  as the initial velocity of the wagon (1 meter per second)
- $v_t$  as the velocity of the trolley after 100 seconds
- $\Delta m$  as the mass of the rainwater collected in the trolley during the 100 seconds

The initial momentum of the system is the momentum of the wagon:

$$m_w \cdot v_w$$

After the rainwater starts accumulating, the momentum of the system is the momentum of the wagon plus the momentum of the rainwater:

$$(m_w + \Delta m) \cdot v_t$$

According to the problem, the velocity of the trolley halves after 100 seconds. So,  $v_t = \frac{1}{2}v_w = \frac{1}{2} \cdot 1 = 0.5 \text{ m/s}$ .

Using the conservation of momentum, we equate the initial and final momenta:

$$m_w \cdot v_w = (m_w + \Delta m) \cdot v_t$$

$$10 \cdot 1 = (10 + \Delta m) \cdot 0.5$$

Solve for  $\Delta m$ :

$$10 = 5 + 0.5\Delta m$$

$$5 = 0.5\Delta m$$

$$\Delta m = 10 \text{ kg}$$

So, during the 100 seconds, 10 kilograms of rainwater accumulate in the trolley.

To find the rate at which the rainwater accumulates, we divide the total mass of rainwater by the time it took to accumulate:

$$\text{Rate of accumulation} = \frac{\Delta m}{\text{time}}$$

$$\text{Rate of accumulation} = \frac{10 \text{ kg}}{100 \text{ s}}$$

$$\text{Rate of accumulation} = 0.1 \text{ kg/s}$$

Therefore, the rainwater accumulates in the trolley at a rate of 0.1 kilograms per second.

### 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<sup>19</sup>. 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

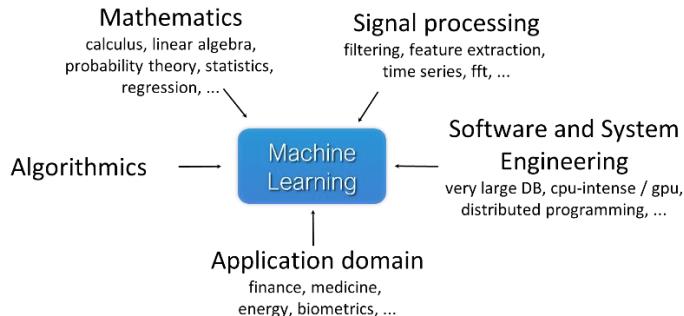
- What could be the problem with Go?

<sup>19</sup> <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.

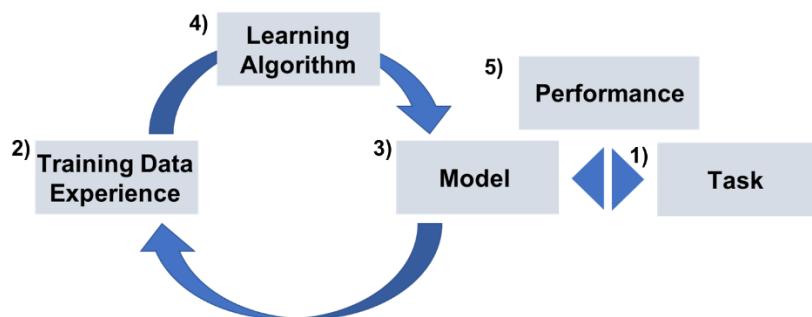
Machine Learning is at the convergence of different fields (Figure 12). Like the term *machine intelligence*<sup>1</sup> there is no unique definition of Machine Learning and different versions are used. Three examples are cited below:

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 [8].

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 [9].

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.

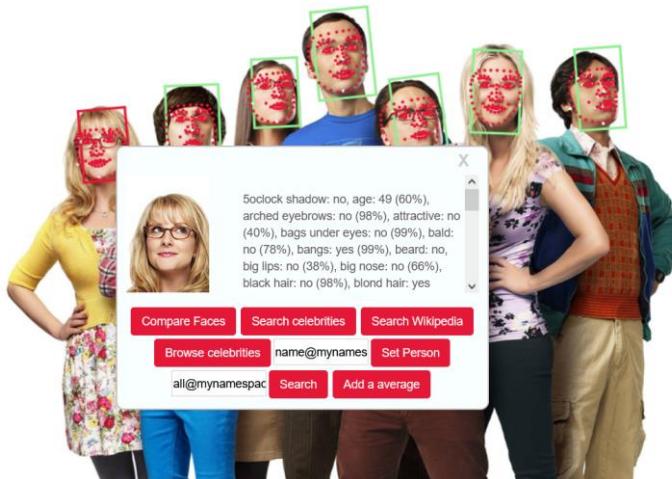


**Figure 14:** The task will be the recognition of different faces in an image.

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<sup>20</sup> is used and the results as shown in the following Figure 15 are obtained.



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

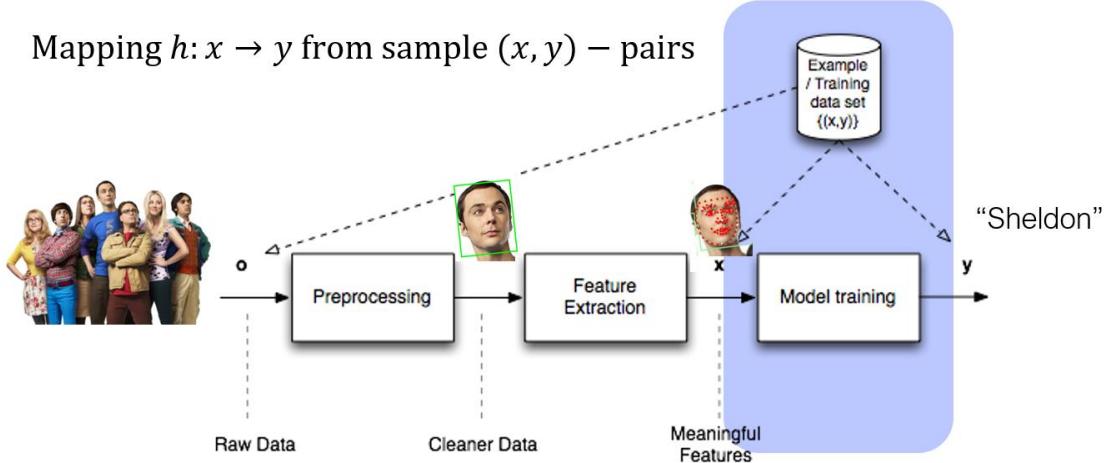
<sup>20</sup> <https://www.betafaceapi.com/demo.html>

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<sup>21</sup>.
- 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  $\mathbf{x}$  from the raw observation data  $\mathbf{o}$  and learn a **mapping** from inputs  $\mathbf{x}$  to outputs  $\mathbf{y}$  given a set of example data pairs  $(\mathbf{x}, \mathbf{y})$  called the **training set**.



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

In our case the features  $\mathbf{x}$  could be the information coded in the red dots and the output  $\mathbf{y}$  is the name of the respective person. The training set consists of a certain number of feature sets  $\mathbf{x}$  extracted for all seven faces. This learning paradigm is called Supervised Learning because the match between the features  $\mathbf{x}$  and the output  $\mathbf{y}$  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”).



<sup>21</sup> 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.



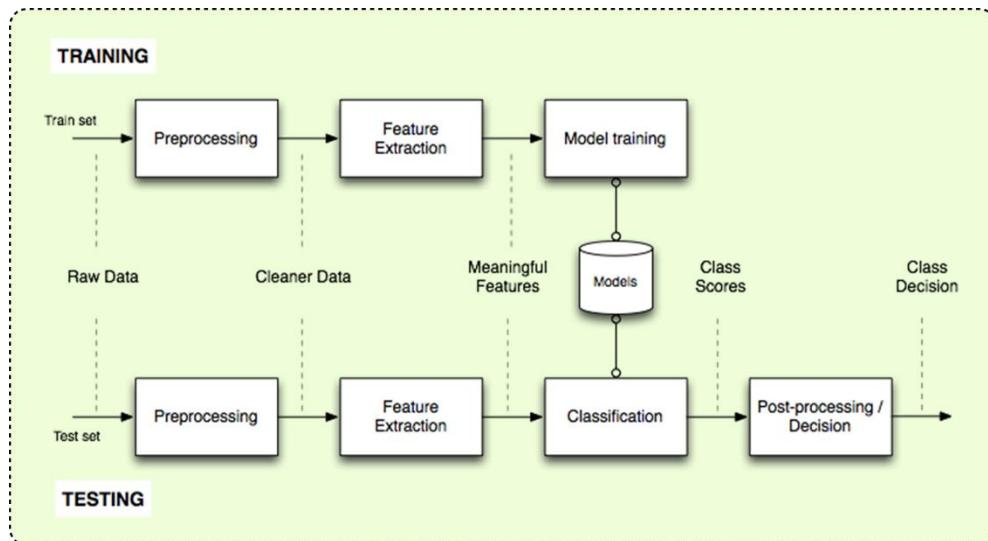
3) Optimal choice of features

It is a priori unclear, which features  $\mathbf{x}$  extracted from the raw face data  $\mathbf{o}$  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 additional 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  $\mathbf{x}$  and the output  $\mathbf{y}$  is given, the goal of unsupervised learning is to discover patterns or structures in the input  $\mathbf{x}$  automatically i.e., from unlabelled data.

### Unsupervised Learning:

The goal of the ML algorithm is to discover interesting **structures** from inputs  $\mathbf{x}$  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.



Market Segmentation

Astronomical and Solar Data Analysis



Social Network Analysis

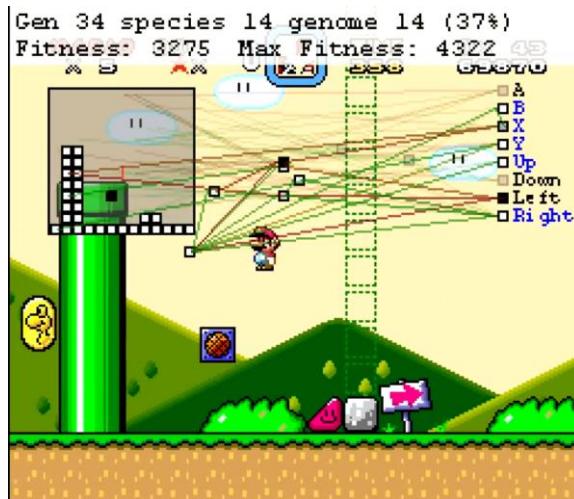
**Figure 18:** Typical fields of application for unsupervised ML approaches.

## 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<sup>22</sup>.

<sup>22</sup> <https://www.youtube.com/watch?v=qv6UVQ0F44>

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<sup>23</sup>.

Exercise:

In groups of 2, do the following:

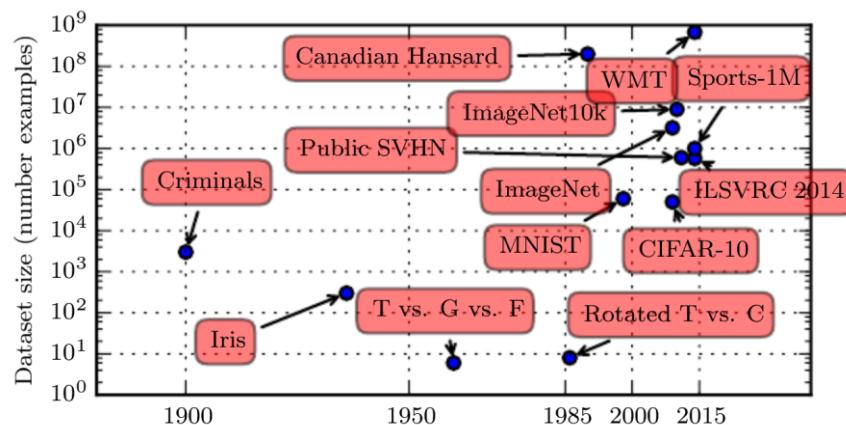
- Watch (parts of) the video with explanation on how computers learn a game, here Super Mario: <https://www.youtube.com/watch?v=qv6UVQ0F44>
- From the video and through discussions try to figure out how the main principles of reinforcement learning work.
- Be prepared to explain these principles in simple 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<sup>24</sup>.



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

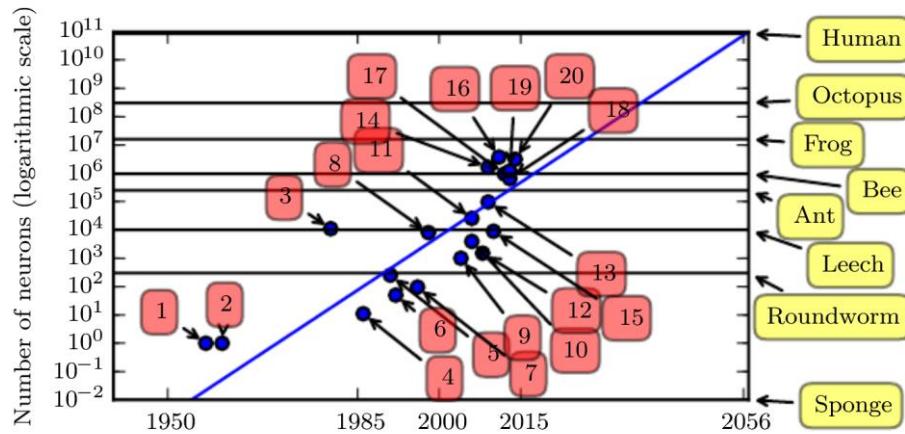
<sup>23</sup> <https://youtu.be/rVlhMGQqDkY>, <https://youtu.be/tf7IEVTDjng>

<sup>24</sup> [https://en.wikipedia.org/wiki/Moore%27s\\_law](https://en.wikipedia.org/wiki/Moore%27s_law)

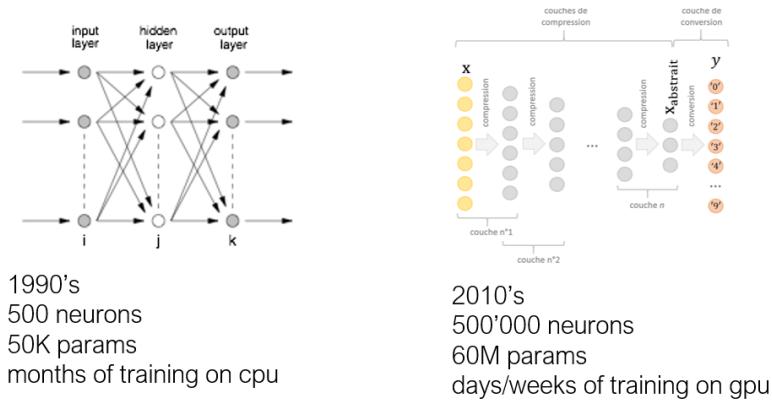
We already stressed the importance of sufficiently large datasets in the previous chapter (c.f. point 2) and this becomes even more relevant for the training of deep ML architectures.

### 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<sup>25</sup>. Since the introduction of hidden units, artificial neural networks have doubled in size roughly every 2.4 years [1].

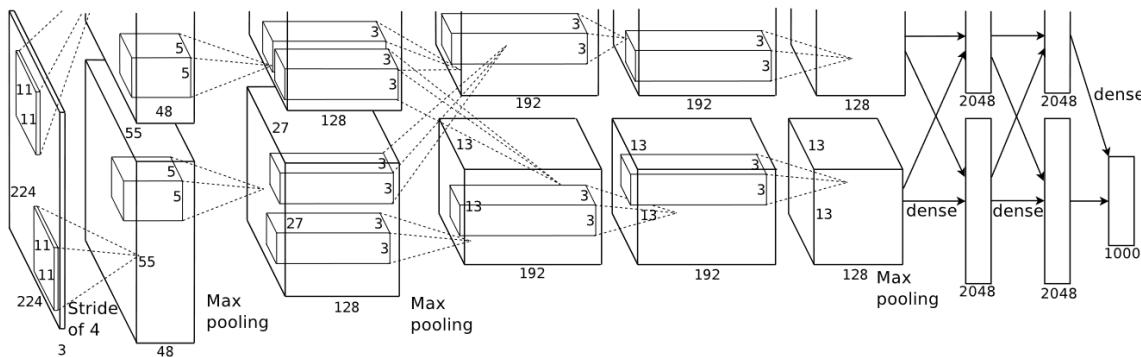


**Figure 23:** Evolution of NN-size and training time from the 1990s till now.

### Availability of new algorithms

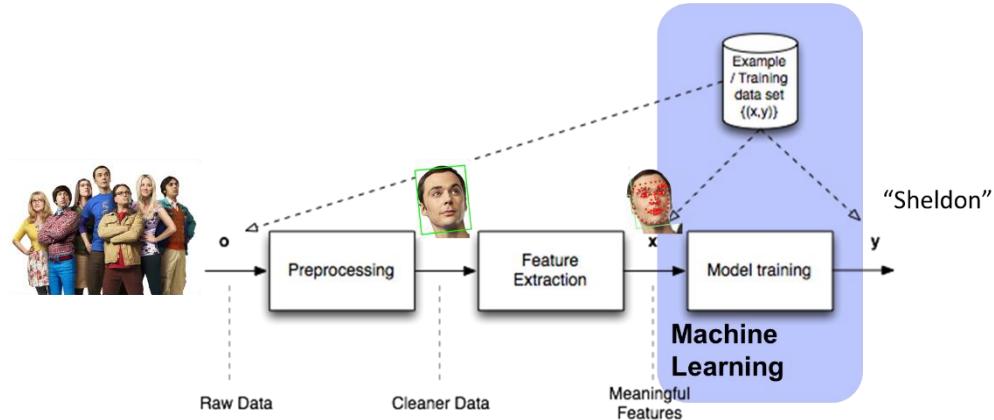
In addition to the increase in dataset size and processing power, ML algorithms have also improved, a topic we will discuss in the following chapters, especially in section 5. As one cornerstone we will cite here already the work by Krizhevsky et al. [5]. Based on a deep CNN architecture (Figure 24) they could push the top-5 error rate on the ILSVRC (ImageNet Large Scale Visual Recognition Challenge) from 26.1% down to 15.3% in 2012. While the CNN-architecture was already developed end of the 1980s [7] it was really their application within a deep architecture that allowed for such a considerable performance gain.

<sup>25</sup> For a legend of the numbers c.f. [1], part 1 Introduction, page 23.

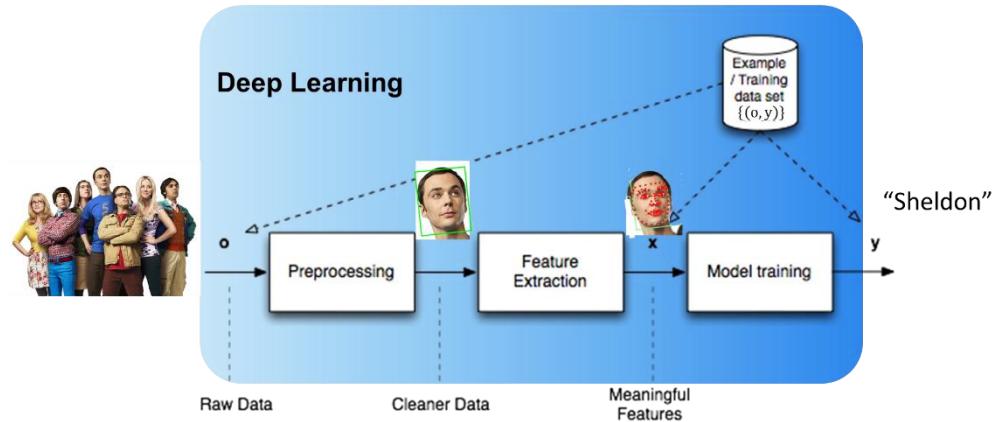


**Figure 24:** Architecture of the CNN used in [5] to push the top-5 error rate on the ILSVRC-challenge in 2012 [6] from 26.1% down to 15.3%.

Mapping  $h: x \rightarrow y$  from sample  $(x, y)$  – pairs



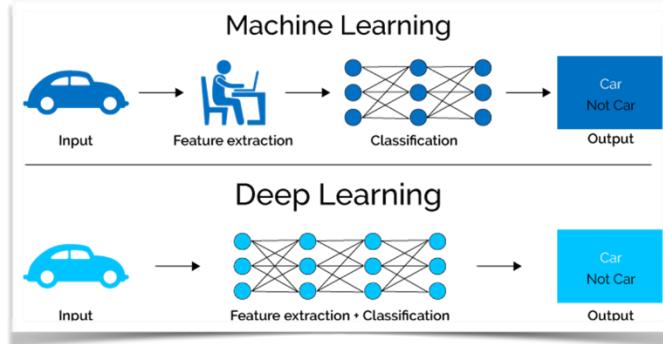
Mapping  $h: o \rightarrow y$  from sample  $(o, y)$  – pairs



**Figure 25:** The DL approach (bottom) can work directly on the raw input data  $o$  and automate the Preprocessing and Feature Extraction/Optimization step, which are both manual for the ML approach (top).

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 25 (top) the ML approach as discussed in the previous chapter is shown. It involves the manual labelling of the faces in the im-

ages (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  $\mathbf{x}$  for the ML algorithm. The corresponding DL approach is shown in bottom part of the Figure 25. Deep ML architecture can work directly on the raw input data  $\mathbf{o}$ , 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 26).



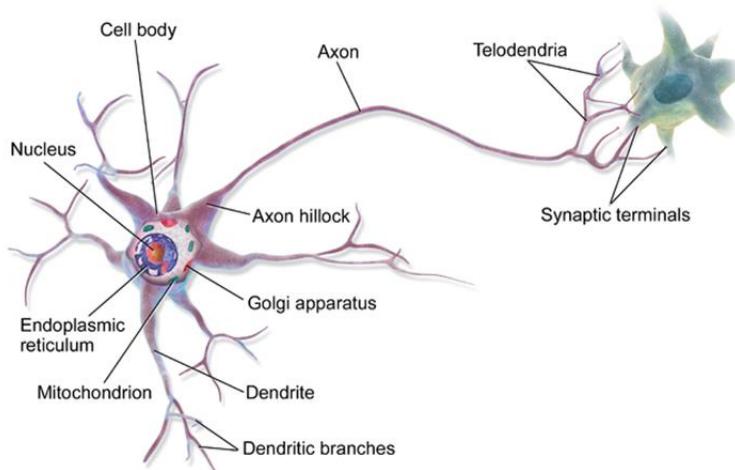
**Figure 26:** DL (bottom) allows to skip the step of the hand-engineered features in ML (top) which will considerably save time and improve the result.

## 2 First Neural Networks – a Historical Perspective

### 2.1 Biological Neural Systems

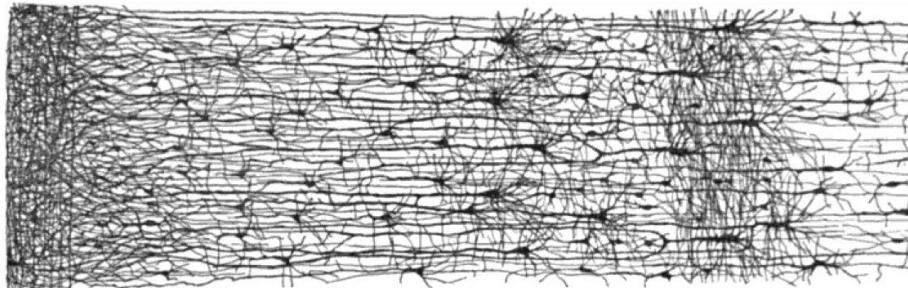
Neuroscience i.e., the scientific study of biological neurons and the nervous system in general was an important driver for the early developments of the field of artificial intelligence. Historically, in Neural Network Science the Neuro-Scientific Perspective was prevalent with the goal to understand 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<sup>26</sup>. 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 27). 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 27:** Representation of a biological neuron (details see text)<sup>27</sup>.

Due to the mutual connections of the neurons, they form a complex neural network (Figure 28).



**Figure 28:** Cross section of the visual cortex showing neurons connected to a network<sup>28</sup>.

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

<sup>26</sup> A more recent initiative in that sense is the Human Brain Project of EC: <https://www.humanbrainproject.eu/en/>

<sup>27</sup> <https://en.wikipedia.org/wiki/Neuron>

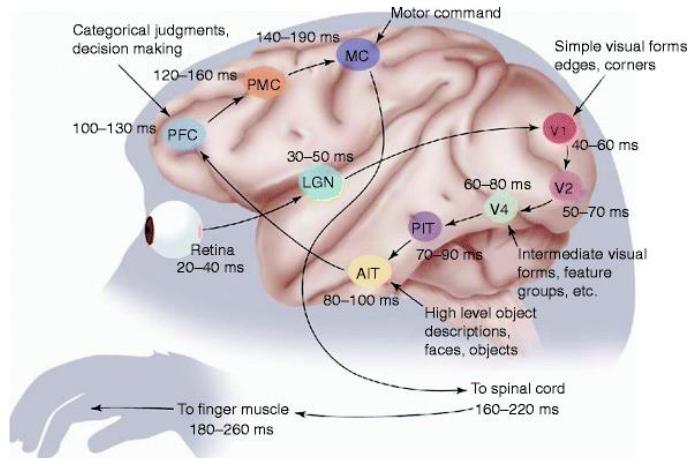
<sup>28</sup> [https://en.wikipedia.org/wiki/Cerebral\\_cortex](https://en.wikipedia.org/wiki/Cerebral_cortex)

- 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 29: 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 29:** 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

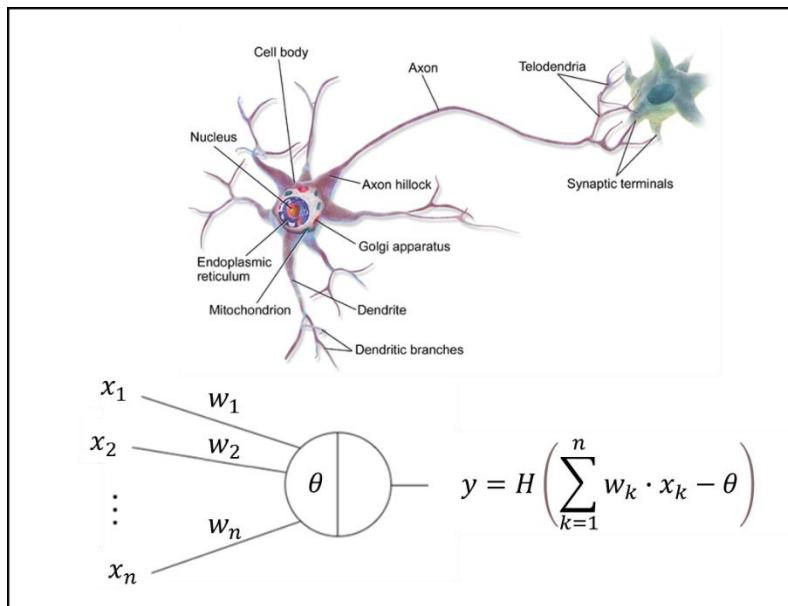
The first theoretical model for the activation of a neuron was developed in 1943 by Warren McCulloch and Walter Pitts (Figure 30, bottom). The idea was inspired by the functioning of biological neurons as discussed in the previous chapter. I.e., each neuron receives input signals from its dendrites and 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  $\theta$ , the neuron will fire, sending a spike along its axon. For the McCulloch-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:

$$\begin{aligned} x_k &= 0,1 \\ w_k &= \pm 1 \quad (+1 \text{ excitatory or } -1 \text{ inhibitory}) \end{aligned}$$



**Figure 30:** The McCulloch-Pitts Neuron (bottom) was the first model for the activation of biological neurons (top) (details c.f. text).

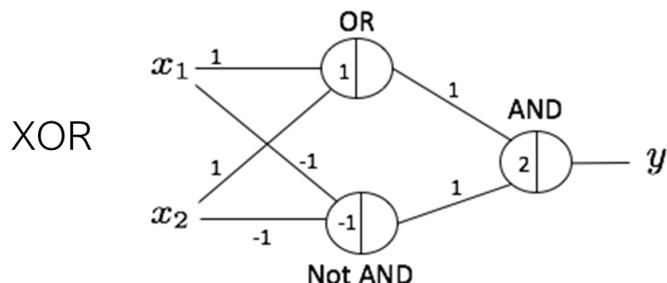
Despite the simple nature of the McCulloch-Pitts neuron its computational power is quite strong. McCulloch and Pitts showed that, in principle, any logical or arithmetic function can be computed with networks of such neurons<sup>29</sup>. Three examples for simple logical functions are given below:

$$\text{AND: } y = H(x_1 + x_2 - 2)$$

$$\text{OR: } y = H(x_1 + x_2 - 1)$$

$$\text{XOR: } y = H(H(x_1 + x_2 - 1) + H(1 - x_1 - x_2) - 2)$$

In Figure 31 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 31:** 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  $\theta$ .

#### 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

<sup>29</sup> <https://link.springer.com/article/10.1007%2FBF02478259>

While the McCulloch-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<sup>30</sup> 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<sup>31</sup> of the second ...".

In brief: "*Cells that fire together wire together.*"

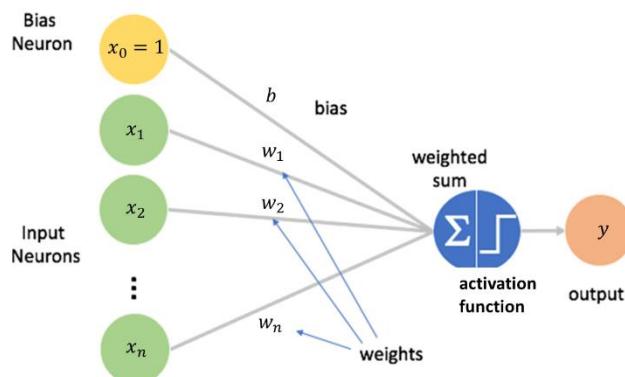
While this learning rule is not proven from a biological point of view it is nevertheless considered as the first rule for "self-organised learning". However, when taken as a principle for ANNs to adjust connection weights this does not provide a stable learning procedure.

## 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 McCulloch-Pitts neurons but allows for arbitrary real inputs, weights and biases i.e.:

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

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



**Figure 32:** Graphical representation of the single Perceptron also referred to as Linear Threshold Unit.

In Figure 32 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  $\Sigma$ -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?

<sup>30</sup> [https://en.wikipedia.org/wiki/Hebbian\\_theory](https://en.wikipedia.org/wiki/Hebbian_theory)

<sup>31</sup> Cell body.

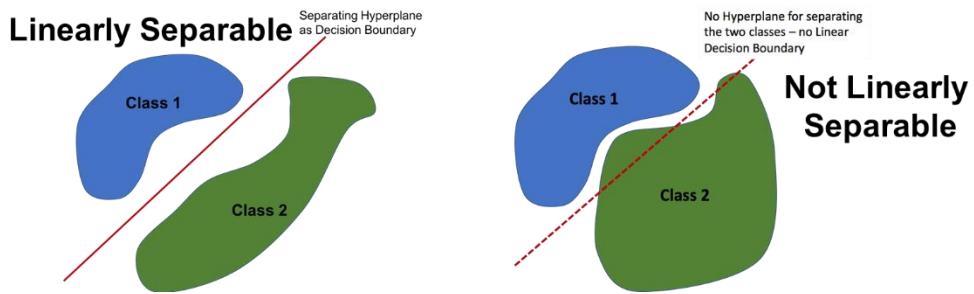
The set of points<sup>32</sup>  $H_{w,b}$  defined as<sup>33</sup>

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

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

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

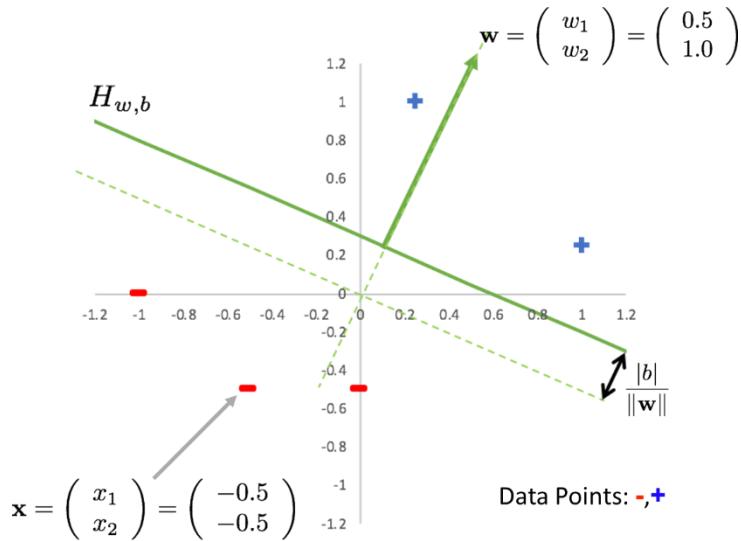
For any point  $\mathbf{x}$  not lying on the hyperplane the result of  $\mathbf{w} \cdot \mathbf{x} + b$  will be greater or smaller than zero, depending on which side of the hyperplane the point  $\mathbf{x}$  is lying. Thus, the Perceptron is suited for **linearly separable binary classification**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



**Figure 33:** The single Perceptron is suited for linearly separable binary classification problems.

A simple example is given in Figure 34, with the values:

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



**Figure 34:** Illustration of the decision boundary defined by a single Perceptron.

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

<sup>32</sup> The indices  $w, b$  express the dependencies upon the weights and the bias.

<sup>33</sup> 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  $\{(\mathbf{x}^{(i)}, y^{(i)}) \mid i = 1, \dots, N\}$ , 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:

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

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

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 35 the Perceptron Learning Algorithm is illustrated by applying one update step.

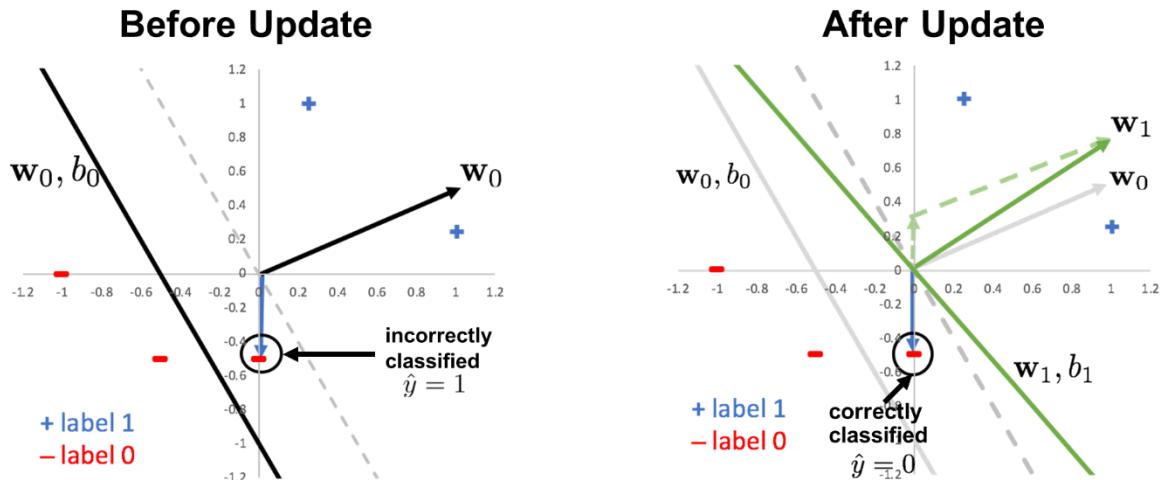
On the left-hand side, the situation before the update is shown. The values of the weights and bias are:

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

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:  $w_1 = \begin{pmatrix} 1 \\ 0.5 \end{pmatrix} - \alpha \begin{pmatrix} 0 \\ -0.5 \end{pmatrix} = \begin{pmatrix} 1 \\ 0.75 \end{pmatrix}$
- Update rule for bias:  $b_1 = b_0 - \alpha = 0$

The new hyperplane (green) on the right-hand side of Figure 35 results, which represents a perfect separation of the two pointsets. We see in fact, that the weight update tilts the boundary counter clockwise 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 35:** 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.

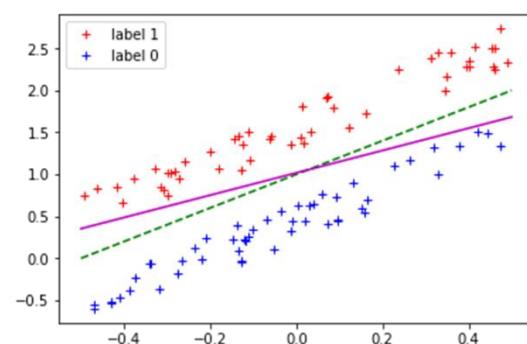
The following theorem can be proved:

#### 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.

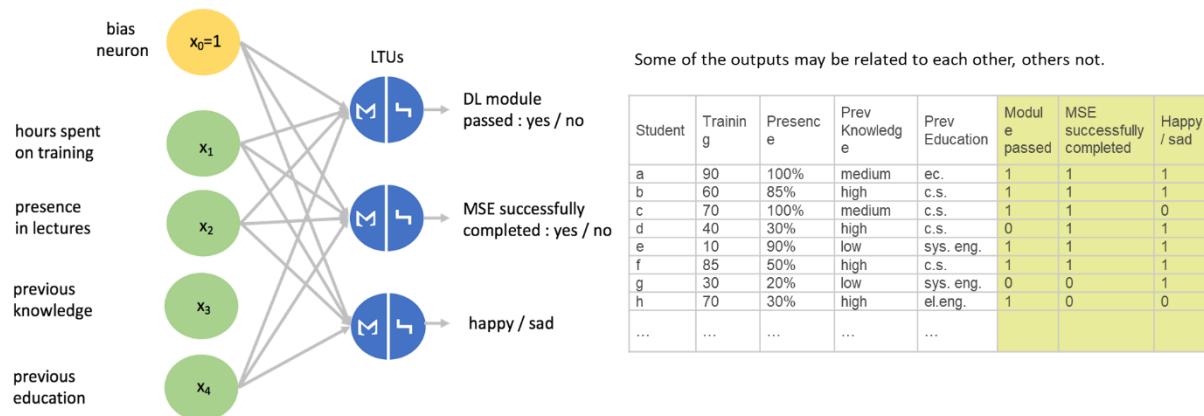


## 2.3 Artificial Neural Networks

### 2.3.1 A first Neural Network: Single Layer LTUs

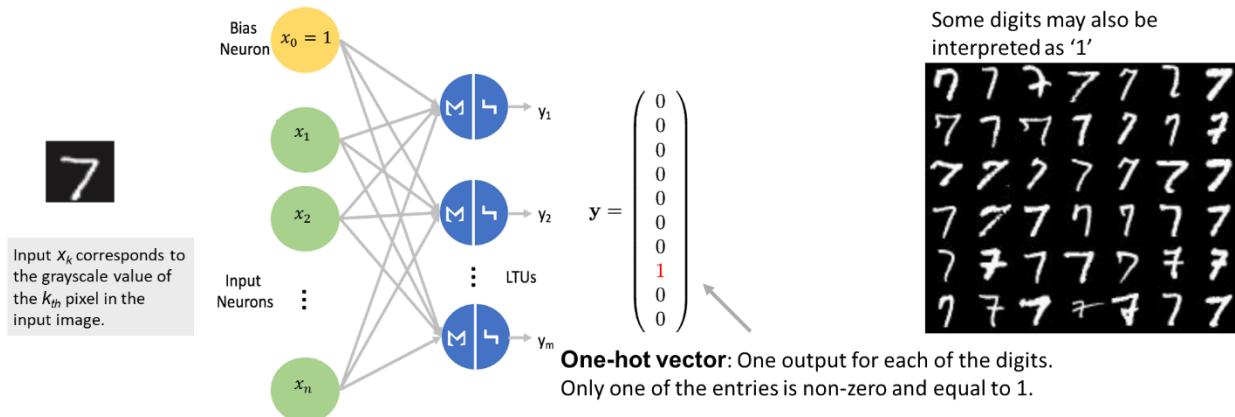
Combining many Perceptron units will build a so-called neural network. Figure 36 shows an example of a single layer network with just one output layer, represented by the blue nodes. The green nodes are the input neurons and each of them is connected to each of the output neurons forming a so-called *fully connected layer*.

From an application point of view, the network in Figure 36 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.



**Figure 36:** 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 37. Based on the MNIST-dataset<sup>7</sup> the classification of handwritten digits is performed with a single layer LTU. The input vector consists of the grayscale values of the  $28 \times 28$ -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 36, 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 37:** 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 37?
- 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

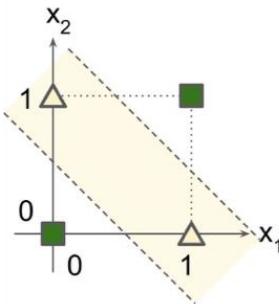
The representational capacity of the single Perceptron is given by a hyperplane separating the feature space in two parts and therefore allowing to classify linearly separable binary set. A single layer L TU, which consists of  $m$  output neurons, represents an extension to  $m$  hyperplanes and therefore 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.

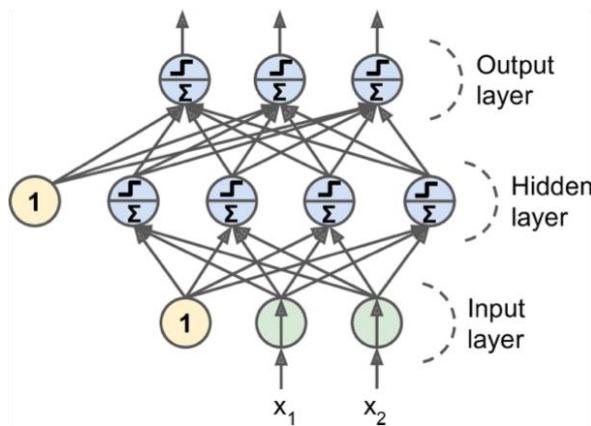
Graphically the XOR problem can be represented as a binary classification task with two sets as shown in the following Figure 38 with the green squares and yellow triangles. It is clear, that these two pointsets cannot be separated by a single layer L TU which immediately represents a considerable limitation 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<sup>34</sup>.



**Figure 38:** 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 39). 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 39:** An MLP is composed an input layer, one or several hidden layers and an output layer.

<sup>34</sup> 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 40). 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.

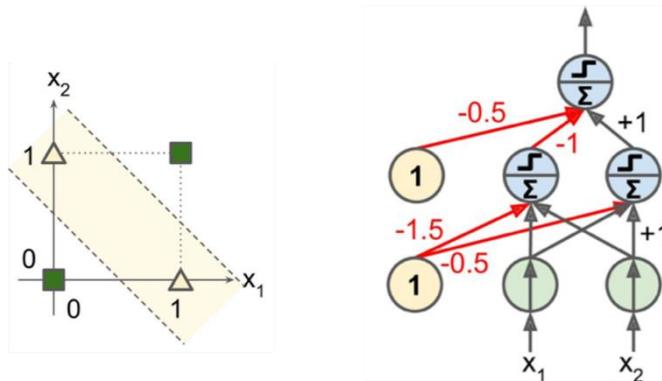
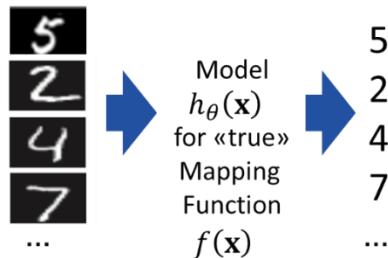


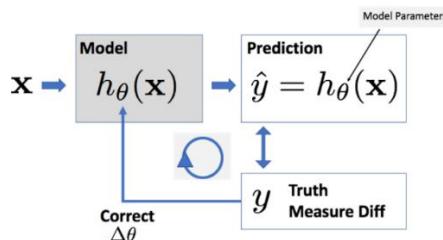
Figure 40: Solution of the XOR-problem based on a MLP with one hidden layer.

### 3 Learning and Optimisation

In this chapter we will formulate the machine learning task (Figure 43) in formal mathematical language<sup>35</sup>. The exemplary task we want to solve is to automatically predict the correct digit from a handwritten image (Figure 41). 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  $\theta$  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 42. 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  $\theta$  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.



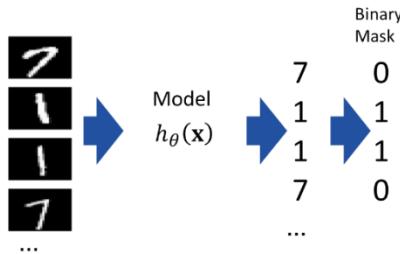
**Figure 41:** We want to solve a classification task i.e., find the correct digits (right) to the input images (left).



**Figure 42:** Formalization of the learning procedure required to optimize our model  $h_\theta(\mathbf{x})$  (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 32, 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 43:** To simplify the problem we start with a binary classification task based on a set of two digits only.

<sup>35</sup> Before reading further it might be worth having a look at the document summarizing the conventions used for the mathematical notations [10].

We will nevertheless not tackle right away the classification task described in Figure 41 with 10 different classes but start with the simpler binary classification problem based on two digits only (Figure 43). As dataset we will use the MNIST digits which we will quickly introduce in the following section.

### 3.1 MNIST Dataset

MNIST<sup>7</sup> 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. MNIST data is available for download via reference given here<sup>7</sup> or various Python frameworks (scikit-learn, PyTorch, keras, tensor flow).



**Figure 44:** A subset of the 70'000 MNIST patches containing handwritten (American English<sup>36</sup>) digits.

In addition to the MINST digit data we will use the Fashion-MNIST dataset [11], which bears some similarities with the original MINST digit set, as it has the same image size (28 x 28-pixel), the same number of categories (10) and the same number of images (70'000). Fashion-MNIST is based on the assortment on Zalando’s website, where every fashion product is presented with a set of pictures shot by professional photographers. The Fashion-MNIST pictures are a reduced version of the front look thumbnail of each article used.



**Figure 45:** A subset of the 70'000 Fashion-MNIST patches [11] containing thumbnails of articles from Zalando’s webpage.

#### Exercise:

Using the provided iPython notebooks perform the following tasks:

- Start using the iPython notebook `1.2-generalised_perceptron-stud.ipynb`. Download the MNIST and Fashion-MNIST data using cell [2].

<sup>36</sup> Note that a 1 is written using a single vertical bar only.

- Verify in cell [3] that the size of the MNIST data is as indicated above.
- Familiarize yourself with the plot functions `plot_img()` and `plot_tiles()`, cells [4] and [5] respectively and visualize some sample images

## 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

After the learning procedure we want to test our machine learning model. 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<sup>37</sup>:

- 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.

### 3.2.2 Data Normalisation – Scaling and Centring

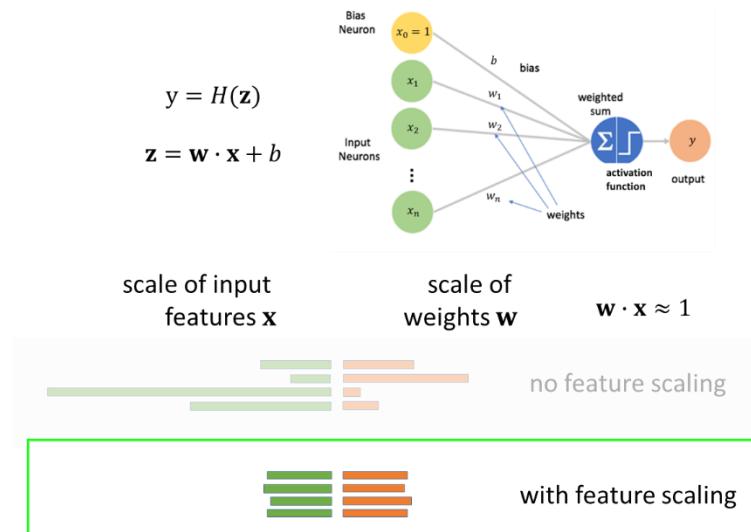
Data normalization is the process of scaling and centring the data:

- 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 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.

---

<sup>37</sup> Splitting the dataset into three parts (with an additional validation set) will be treated later.

### Illustration of Scaling of input data:



**Figure 46:** 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<sup>38</sup> 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 may contain the following features:

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

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 46. The activation function  $H(\mathbf{z})$ <sup>39</sup> 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 46). 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

<sup>40</sup>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:

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

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]:

<sup>38</sup>This may also apply to the output data  $y$  e.g., in case we have a regression task.

<sup>39</sup>In Figure 46 the Heaviside function is used but the same would be true for other types of activation functions (c.f. chapter 4.2).

<sup>40</sup>For the notation convention refer to [10].

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

```
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.  1.  1.  1.]
 [ 0.  2. 14.  5. 10.  1.  1.  1.]
 [ 0.  0.  6. 13. 10.  1.  1.  1.]
```

**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 47:** Min-Max Rescaling applied to (a single) MNIST light<sup>41</sup> image.

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  $x^{(i)}$  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 47, 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):

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

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

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

<sup>41</sup> MNIST light is a lightweight version of the MNIST digit dataset with a size of 8 x 8-pixel and 16 gray values only.

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

Exercise:

Using the provided iPython notebook perform the following tasks:

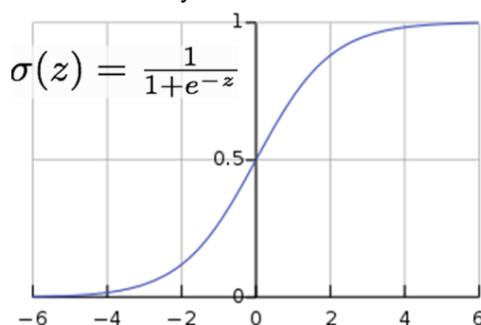
- Continue using the iPython notebook `1.2-generalised_perceptron-stud.ipynb`. Complete cell [8] (function `prepare_data()`) such that the training and test-set split works.
- Then add min-max-Rescaling and Normalisation option in `prepare_data()`. Verify that your implementation is correct.

### 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 32) 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 48<sup>42</sup>. 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<sup>43</sup>.



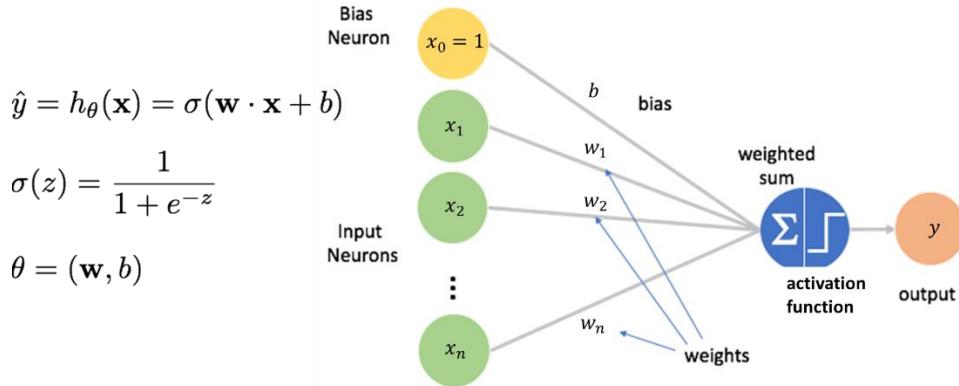
**Figure 48:** The sigmoid activation function is a smooth generalisation of the Heaviside step function.

Now, based on the sigmoid function we can generalise the Perceptron to the version shown in Figure 49 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:

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

<sup>42</sup> Later, in chapter 4.2 we will present all activation functions commonly used for ML models.

<sup>43</sup> Properties of the sigmoid function will be studied in the practical work exercises.



**Figure 49:** The generalised perceptron using the sigmoid activation function.

As for the Rosenblatt Perceptron we require to optimize the parameters  $\theta = (\mathbf{w}, b)$  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  $\theta = (\mathbf{w}, b)$  such that the predictions  $\hat{y}^{(i)} = h_\theta(\mathbf{x}^{(i)})$  of our model are “close” to the true outcomes  $y^{(i)}$ . This 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  $\hat{y}^{(i)} = h_\theta(\mathbf{x}^{(i)})$ :

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

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 42) can be simply formulated by determining of the minimum of the cost function  $J_{MSE}(\theta)$  with respect to the parameters  $\theta$ . 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  $J(\theta)$  based on the GD schemes works as follows:

1. Start with some initial value  $\theta_0$  for the parameter vector  $\theta$ .  
(e.g., random values or all 0)
2. Iteratively update the parameter vector  $\theta$  by
  - a. Computing the gradient of the cost function at the current position  $\theta_t$ :  $\nabla_\theta J(\theta_t)$
  - b. Step in the negative gradient direction according to:  

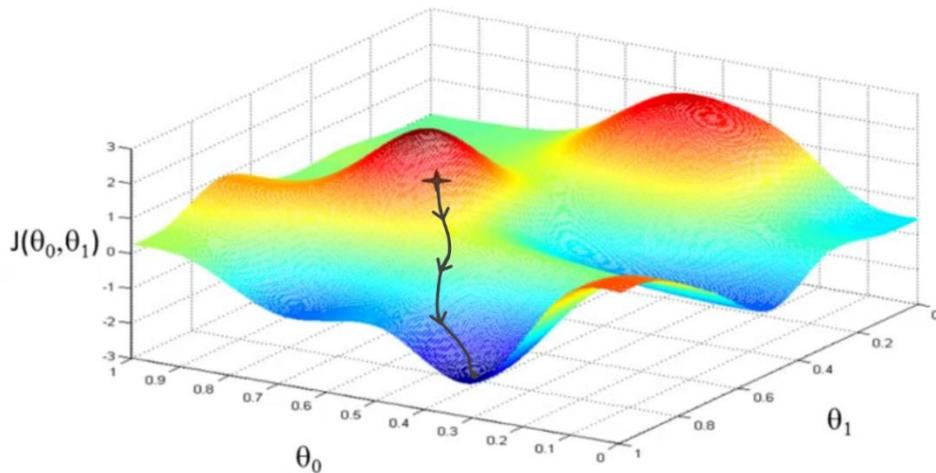
$$\theta_{t+1} = \theta_t - \alpha \cdot \nabla_\theta J(\theta_t) \quad (\alpha \text{ is the learning rate})$$
3. Stop when the change in parameter vector update step ( $\theta_t \rightarrow \theta_{t+1}$ ) is “small”

Figure 50 illustrates GD using a cost function  $J(\theta)$  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 from multidimensional analysis, the derivative

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

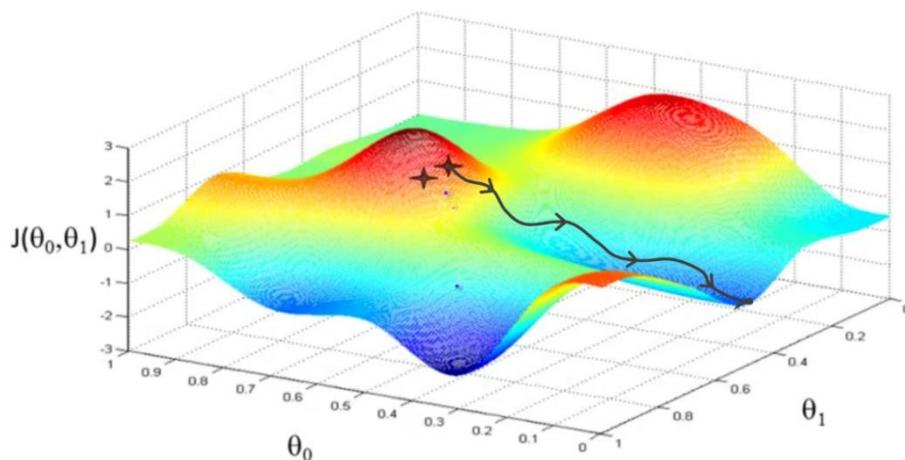
is a 2D vector pointing in direction of the strongest descent of the function  $J(\boldsymbol{\theta}_t)$ . Thus, when starting at the point indicated by the star, the GD algorithm will lead successively along the black trajectory to the indicated local minimum, which however is not necessarily the global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



**Figure 50:** 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 51).



**Figure 51:** 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 49).

### 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

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

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

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

Here, the parameter components  $\theta_k$  can be either a weight component  $w_k$  or the bias  $b$ . We also recall that prediction of the model is given by  $\hat{y}^{(i)} = h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}) = \sigma(\mathbf{w} \cdot \mathbf{x}^{(i)} + b)$ .

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:

$$\begin{aligned} \frac{\partial}{\partial \theta_k} (h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}) - y^{(i)})^2 &= 2 \cdot (h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}) - y^{(i)}) \cdot \frac{\partial}{\partial \theta_k} h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}) =^{(1)} \\ &= 2 \cdot (h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}) - y^{(i)}) \cdot (1 - h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)})) \cdot h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}) \cdot \frac{\partial}{\partial \theta_k} (\mathbf{w} \cdot \mathbf{x}^{(i)} + b) \end{aligned}$$

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  $\theta_k$  is a weight component  $w_k$  or the bias  $b$ :

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

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

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

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

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:

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

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:

$$\begin{aligned} &(\hat{y}^{(i)} - y^{(i)}) \cdot \mathbf{x}^{(i)} \\ &(\hat{y}^{(i)} - y^{(i)}) \end{aligned}$$

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<sup>44</sup>.

#### Exercise:

Using the provided iPython notebook perform the following tasks:

- Continue using the iPython notebook `1.2-generalised_perceptron-stud.ipynb`. Complete the class `NeuralNetwork` in cell [9]:
  - Implement the sigmoid function in method `activation_function`.
  - Implement the MSE cost function in method `cost_funct`.
  - Implement the derivatives of the MSE cost function with respect to parameter vector  $\boldsymbol{\theta} = (\mathbf{w}, b)$  (`self.grad_w`, `self.grad_b`) in method `back_propagate`.
- Then run the training in cell [12] and verify that your implementation is correct.

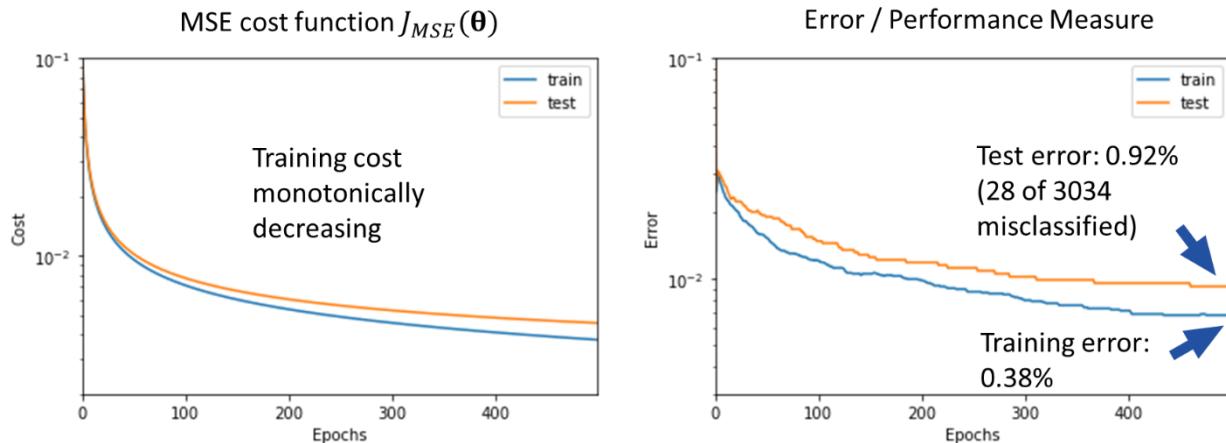
#### 3.4.3.1 Binary Classification for generalised Perceptron using GD and MSE Cost

We will perform binary classification (c.f. Figure 43) of digits 1 and 7<sup>45</sup>, which are represented with a total of 7877 and 7293 in MNIST, respectively. We will apply the following settings:

- The generalised Perceptron (Figure 49), 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.
- A learning rate  $\alpha = 0.5$  is used.
- A total of 500 GD optimisation steps (so-called *epochs*) are applied.

<sup>44</sup> The corresponding iPython notebooks will be part of the practical work.

<sup>45</sup> We will see later that the final error of the prediction may vary considerably depending on the choice of digits.



**Figure 52:** 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 52 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 52 shows the fraction of misclassified samples (error rate) as performance measure. Its definition is given in the Figure 53 below. We use the value of 0.5 as threshold to decide whether a digit was classified as 1 or 7 (c.f. Figure 49).

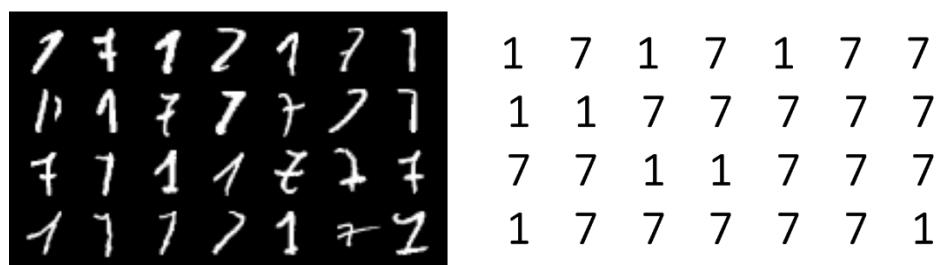
Fraction of misclassified samples as **performance measure**:

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

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

( $m_{\text{test}} = \# \text{ test samples}$ )

**Figure 53:** The fraction of misclassified samples (with respect to the total training or testing set size) is used as performance measure.



**Figure 54:** The 28 misclassified digits from the testing set with their respective ground truth given to the right.

Figure 54 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 55 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<sup>46</sup>. 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<sup>47</sup>. However, this is only half of the story as is illustrated in Figure 56.

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.

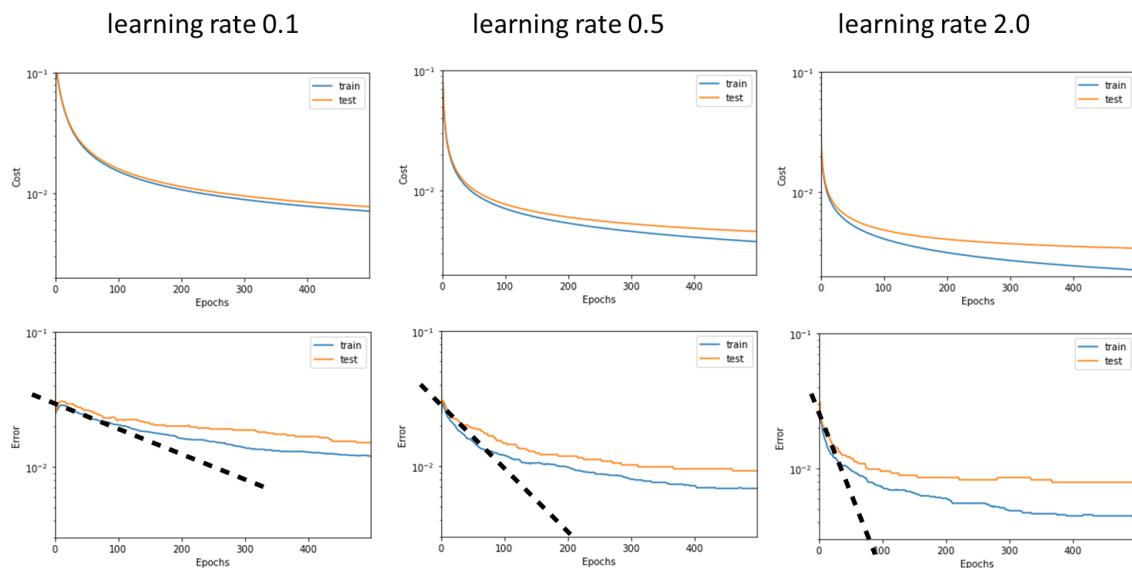


Figure 55: Effect of different learning rates on cost (top) and error rate (bottom).

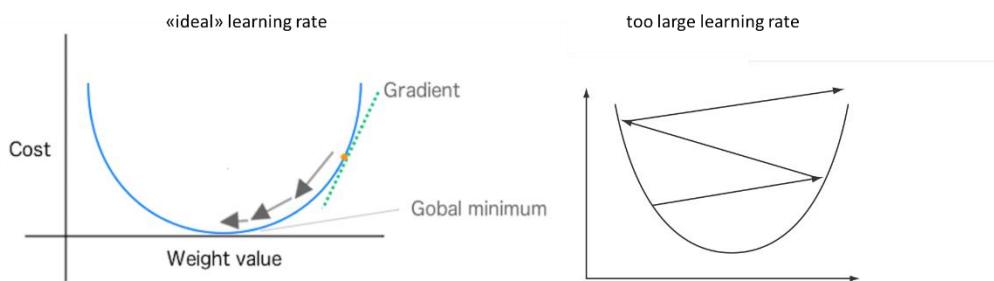
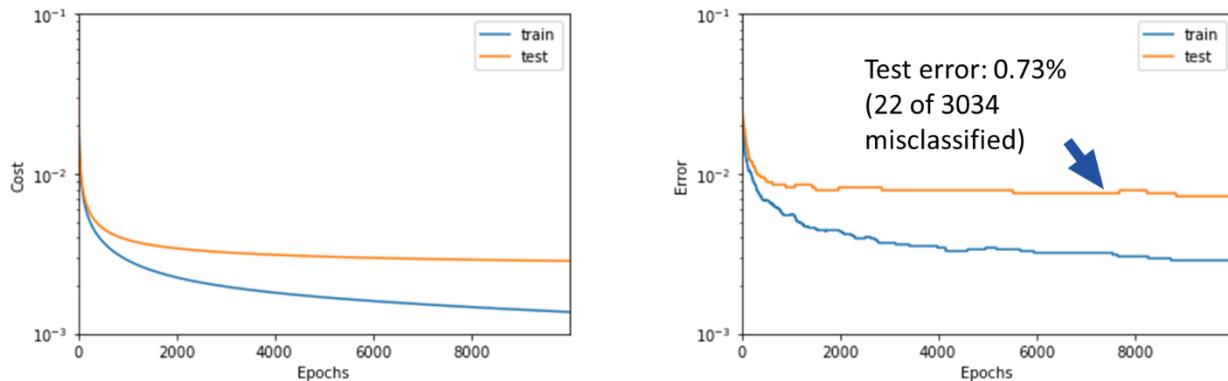


Figure 56: Illustration of a «good» choice of learning rate (left), where GD moves smoothly to the minimum. A too large learning rate leads to GD oscillating around the minimum (right).

<sup>46</sup> 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.

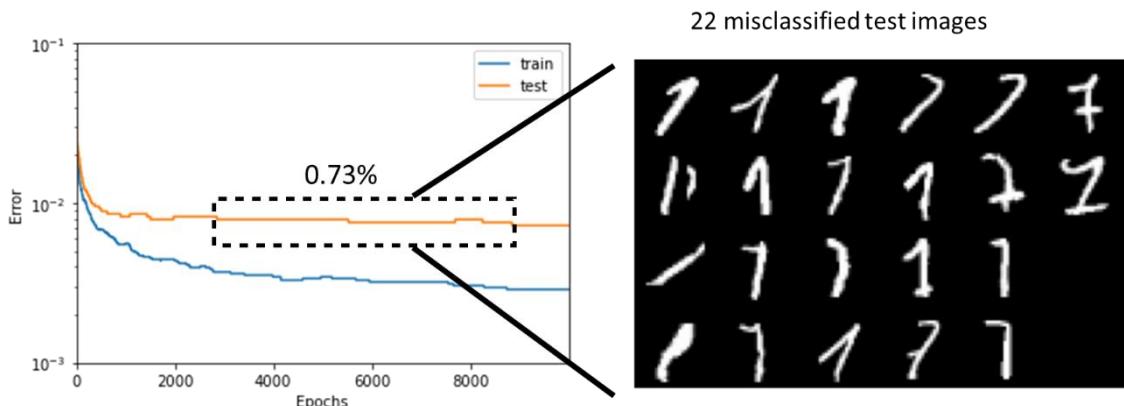
<sup>47</sup> It should be noted that the learning rates used here for the illustration purpose are atypically large and usually considerably smaller values are used.

### Increasing number of epochs



**Figure 57:** Cost function and error rate for 10'000 epochs. After some 1000 epochs the test error rate will not decrease any further.

In Figure 57 and Figure 58 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 58:** Like Figure 57 above. The test error rate will not decrease after some 1000 epochs corresponding essentially always to the 22 misclassified images shown.

Thus, in summary we see that both the choice of the learning rate  $\alpha$  and of the number of training epochs is far from 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<sup>48</sup> 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  $\theta$  denoted by the subscript. Our goal now is to optimize this estimator by tuning the parameter vector  $\theta$ . The maximum likelihood estimator now states to select those parameter values  $\theta_{MLE}$  that make the observed data most probable (*most likely*) under that estimator. In mathematical terms this reads:

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

Here the expression  $p_\theta(y|\mathbf{x})$  denotes the probability for the entire training data. On the right-hand-side this is 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<sup>49</sup> is taken leading to the final formulation of the maximum likelihood estimator:

$$\theta_{MLE} = \operatorname{argmax}_\theta \sum_{i=1}^m \log p_\theta(y^{(i)}|\mathbf{x}^{(i)}) = \operatorname{argmin}_\theta \left[ - \sum_{i=1}^m \log p_\theta(y^{(i)}|\mathbf{x}^{(i)}) \right]$$

CE Cost

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  $y^{(i)} = 0$ <sup>50</sup>:

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

<sup>48</sup> 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.

<sup>49</sup> As the logarithm function is monotonically increasing this will not influence the result.

<sup>50</sup> For a Softmax-layer output, which we will study in chapter 3.8, 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:

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

**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

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

with respect to the components  $\theta_k$  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):

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

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:

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

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

$$\frac{1}{2m} \sum_{i=1}^m (\dots)$$

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 from chapter 2.2.2.2:

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

#### Exercise:

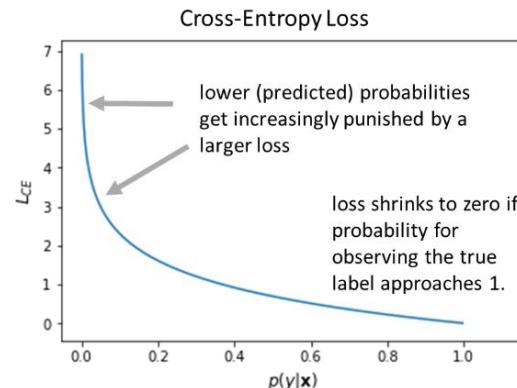
Using the provided iPython notebook perform the following tasks:

- Continue using the iPython notebook `1.2-generalised_perceptron-stud.ipynb`. Complete the class `NeuralNetwork` in cell [9]:
  - Implement the CE cost function in method `cost_funct.`
  - Implement the derivatives of the CE cost function with respect to parameter vector  $\theta = (\mathbf{w}, b)$  (`self.grad_w`, `self.grad_b`) in method `back_propagate`.
- Then run the training in cell [12] and verify that your implementation is correct.

### 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.

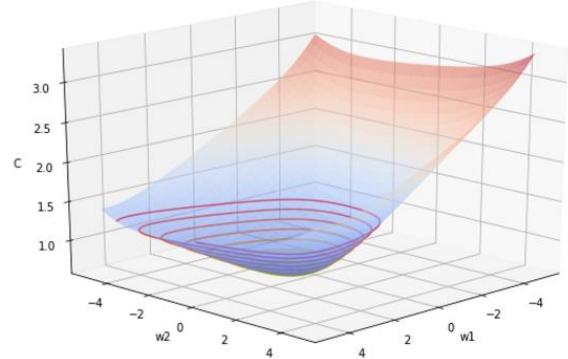
In the figure to the right the behaviour of the function  $\log p_\theta(y|\mathbf{x})$  i.e., the CE cost is shown as a function of the probability  $p_\theta(y|\mathbf{x})$ . It is meant to give some intuition on why the CE Loss has superior performance as compared to the MSE Loss. The major point 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.

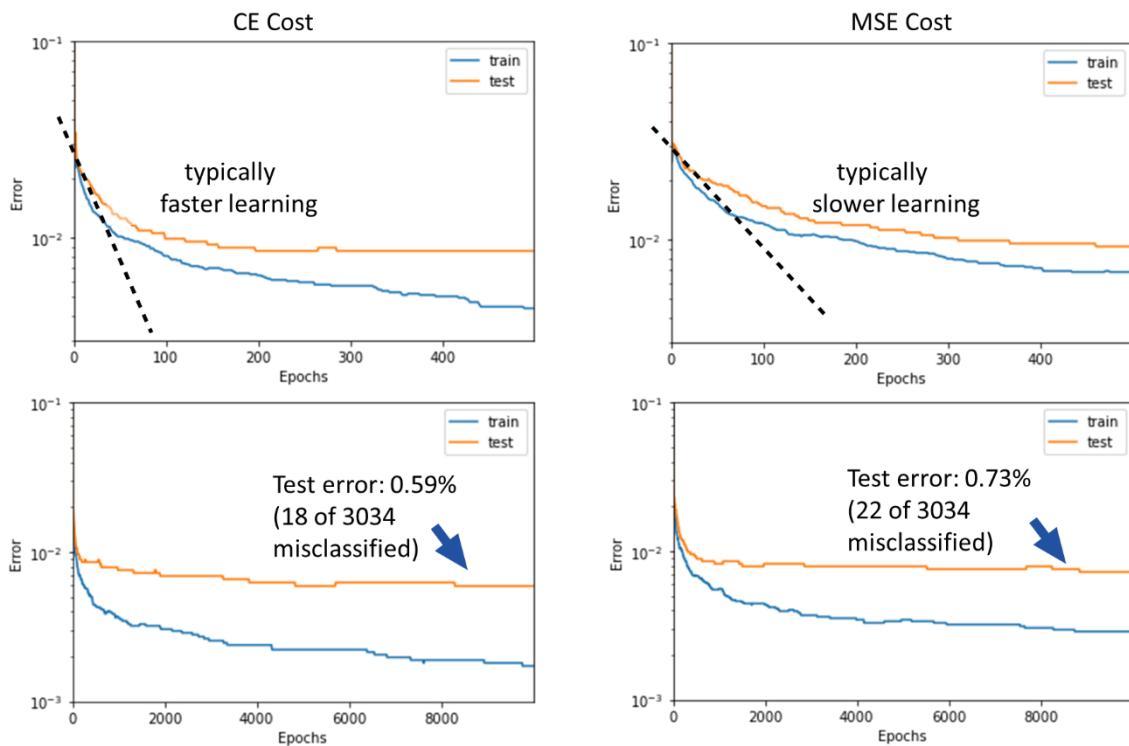
The CE cost function for the generalised perceptron is a convex function i.e., bowl-shaped as illustrated to the right<sup>51</sup>.

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.



In the Figure 59 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 59:** 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  $\alpha = 0.5$

<sup>51</sup> It should be noted that for more complex architectures e.g., MLPs this is not the case.

- 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.

	$\leftarrow j \rightarrow$									
	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 51 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  $z \sim N(0,1)$ <sup>52</sup>, a property which is beneficial for deep architectures and which we will study in chapter 5.1.

<sup>52</sup> We use the independence of  $w_k$  and  $x_k$  leading to  $\text{var}(w_k \cdot x_k) = \text{var}(w_k) \cdot \text{var}(x_k)$  (with zero mean).

- 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 60 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.



Figure 60: Set of 100 misclassified digits for a binary (5,3) classification on the MNIST data set.

## 3.5 Performance Measures

### 3.5.1 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 61, the orange box indicates that 63 samples of total 221 of actual class **b** (8+131+63+19) are predicted as class **c**.

		predicted class				$\Sigma$
		<b>a</b>	<b>b</b>	<b>c</b>	<b>d</b>	
<b>a</b>	<b>a</b>	120	21	7	8	156
	<b>b</b>	8	131	63	19	221
<b>c</b>	<b>c</b>	12	30	80	11	133
	<b>d</b>	1	11	8	40	60
$\Sigma$		141	193	158	78	570

Figure 61: Illustration of a confusion matrix which presents the number of the actual class labels versus the predicted ones.

Figure 62 shows a further example from the MNIST dataset<sup>53</sup>. 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'.

actual	predicted	0	1	2	3	4	5	6	7	8	9
0		[[[1336,	0,	2,	3,	5,	7,	13,	3,	9,	2],
1		[ 1,	1575,	11,	7,	1,	7,	2,	3,	23,	2],
2		[ 8,	9,	1290,	23,	15,	5,	16,	19,	43,	5],
3		[ 8,	4,	31,	1258,	2,	41,	3,	23,	47,	14],
4		[ 2,	8,	10,	1,	1244,	1,	8,	7,	21,	26],
5		[ 12,	9,	4,	40,	22,	1091,	33,	11,	65,	10],
6		[ 9,	1,	6,	0,	18,	13,	1275,	0,	9,	0],
7		[ 5,	4,	10,	7,	16,	1,	1,	1365,	5,	30],
8		[ 9,	19,	11,	22,	11,	22,	10,	5,	227,	15],
9		[ 6,	10,	2,	19,	60,	8,	1,	53,	20,	1194]]]

Actual class '9',  
predicted as '4'
Actual class '5',  
predicted as '8'

Figure 62: 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%.

$$\text{accuracy} = \frac{\sum \text{diagonal elements}}{\# \text{samples}}$$

#### Error Rate:

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

$$\text{error rate} = 1 - \text{accuracy}$$

#### 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.

### 3.5.2 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

<sup>53</sup> Currently we are limited to binary classifications tasks but once we completed chapter 3.8 we will be able to create such multi-class confusion tables.

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

**Figure 63:** Illustration of a confusion table i.e., a confusion matrix for a binary classification problem.

Figure 63 illustrates an example. Because this concept is often used in the context of hypothesis testing the terms “Positive” 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.

It is also frequent to use rates i.e., “True Positive Rate”, where the normalization is always done with respect to the true values. Thus, one finds the following dependencies:

$$\begin{aligned}
 \text{True Positive Rate} &= \frac{TP}{TP+FN} \\
 \text{True Negative Rate} &= \frac{TN}{TN+FP} \\
 \text{False Positive Rate} &= \frac{FP}{TN+FP} = 1 - \text{True Negative Rate} \\
 \text{False Negative Rate} &= \frac{FN}{TP+FN} = 1 - \text{True Positive Rate}
 \end{aligned}$$

To develop the different performance measures, we will transform the confusion matrix from Figure 62 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 64 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 64:** Confusion table obtained from Figure 62 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:  
 $(1'091 + 12'598) / 14'000 = 97.8\%$ .

$$\text{class accuracy} = \frac{TP + TN}{N}$$

		Predicted		Total
		P	N	
Actual	P	1091	206	1297
	N	105	12598	12703
Total	1196	12804	14000	

#### Per Class Sensitivity (or Recall or True Positive Rate):

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:

$$1091 / (1091 + 206) = 84.1\%$$

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

		Predicted		Total
		P	N	
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:

$$1091 / (1091 + 105) = 91.2\%$$

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

		Predicted		Total
		P	N	
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:

$$1091 / (1091 + (206+105)/2) = 87.5\%$$

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

		Predicted		Total
		P	N	
Actual	P	1091	206	1297
	N	105	12598	12703
Total	1196	12804	14000	

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.

### 3.5.3 Precision-Recall- and ROC-Curves

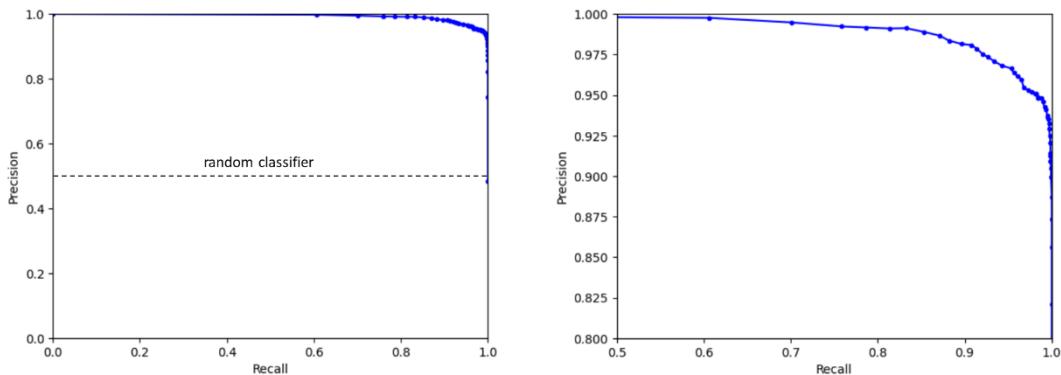
Depending on the specific classification task it might be interesting to “tune” the classifier in order to have a higher precision – with necessarily lower recall – or vice versa. Therefore we change the value of the threshold used to determine – based on the output of the sigmoid function – which kind of class we predict (c.f. recall from chapter 3.3.1):

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

Thus we replace the above decision rule by:

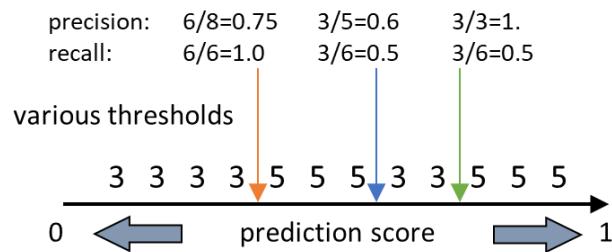
$$\text{yes: } \hat{y} = h_{\theta}(\mathbf{x}) \geq \text{threshold} \quad \text{no: } \hat{y} = h_{\theta}(\mathbf{x}) < \text{threshold}$$

Here *threshold* is a value that may run in the interval from<sup>54</sup> ]0,1[ and depending on it we may determine a corresponding confusion table with individual precision and recall values. From these we can construct the so-called precision-recall-curve (“PR-curve”), which is of fundamental importance to evaluate the performance of a classifier. Because only if precision and recall are *simultaneously* high a classifier provides a good performance. Figure 65 shows an example for a binary classification problem with FashionMNIST data (classes T-shirt [0] and trouser [1])<sup>55</sup>. Obviously, the higher the curve (precision) for all values (recall) the better. Therefore, the area-under-curve (AUC) can be used as a global measure for a classifier, because it makes a statement on the full performance, not only for a single point or value. Also note the PR-curve of a purely random classifier, which has a constant precision of  $\frac{1}{2}$ .



**Figure 65:** Precision-Recall curve (right zoomed – notice the axes) of a binary classification problem with FashionMNIST data (classes T-shirt [0] and trouser [1]).

It is interesting to note that, that the recall will always increase monotonously if the *threshold* is decreased. However, the precision must not necessarily behave monotonously as illustrated in the following Figure 66.



**Figure 66:** While the recall will always increase monotonously as a function of the *threshold* the precision must

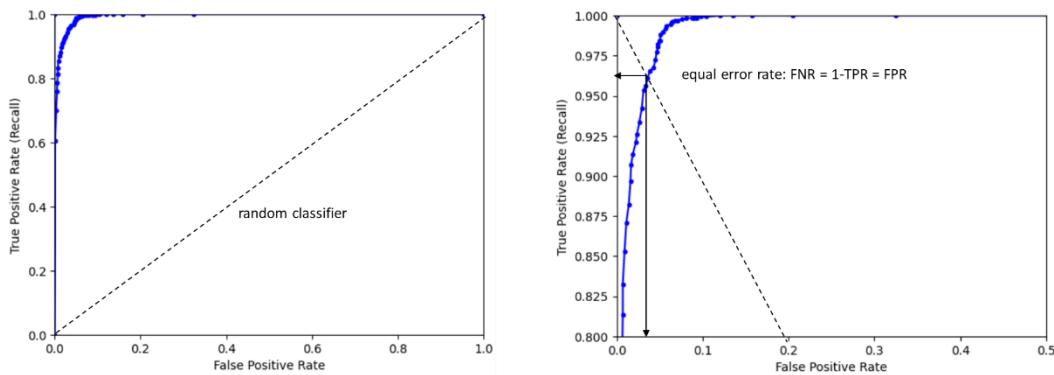
<sup>54</sup> The choice of the boundaries 0 or 1 do not really make sense because then the decision would be trivial.

<sup>55</sup> Training with MSE-cost function for 200 epochs with  $\alpha = 0.5$ .

not.

In addition to the PR-curve the so-called Receiver-Operation-Characteristics (ROC-curve) is used. There the True Positive Rate is shown as a function of the False Positive Rate (Figure 67). Again, the AUC is used as global measure for the performance of the classifier in question. In addition, the so-called equal error rate (EER) is a frequently used (global) measure. This is the point, where the False Positive Rate is equal to the False Negative Rate ( $= 1 - \text{True Positive Rate}$ ). Thus, this point can be considered as a classifier without bias for positive or negative values.

The problem with the ROC-curve is that frequently the number of True Negatives is unknown. This is e.g. true for object detection problems in images, where the positions of certain object categories are predicted in larger images. There, the True Negatives cannot be defined in a meaningful way, because any position in the image could be a potential TN-position. Therefore, the PR-curve has meanwhile gained a higher relevance in ML.



**Figure 67:** ROC-curve (right zoomed – notice the axes) of a binary classification problem with FashionMNIST data (classes T-shirt [0] and trouser [1]) (details see text).

### Exercise:

Using the provided iPython notebook perform the following tasks:

- Use `2.1.precision_recall_curve_stud.ipynb`. Complete the functions `plot_pr_curve` and `plot_conf_matrix` in cells [14] and [15] respectively and reproduce the curves shown in the Figure 65 and Figure 67.
- The code has to be added in the sections

```
### START YOUR CODE ###
...
### END YOUR CODE ###
```

## 3.6 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.

### 3.6.1 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 41) 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  $\theta$ ) 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  $\theta$ . We will indicate this dependency by the additional subscript  $D$  i.e.,  $h_{\theta,D}(\mathbf{x})$ <sup>56</sup>. 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<sup>57</sup> 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<sup>58</sup>:

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

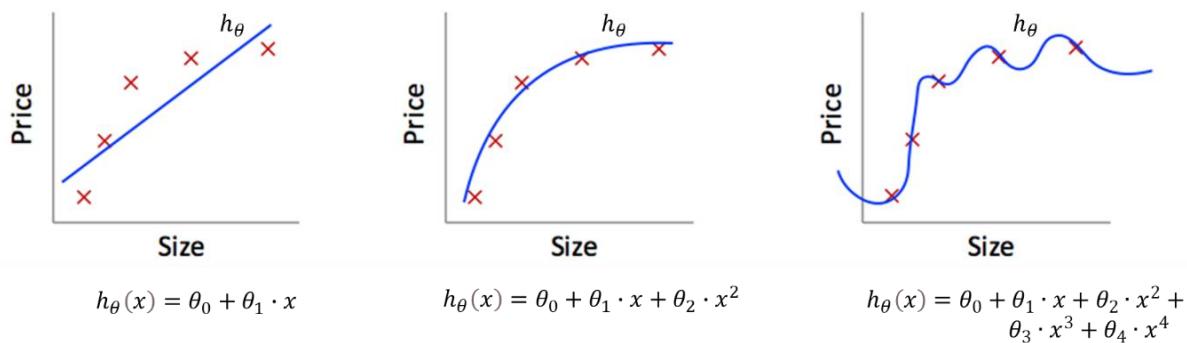
Here  $\mathbf{E}[h_{\theta,D}]$  means the expectation (or average) of  $h_{\theta,D}(\mathbf{x})$  over the training sample set  $D$ . Thus, the bias measures how close the model is on the average to the true mapping.

- 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  $\mathbf{E}[h_{\theta,D}]$ :

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

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

The concept of varying bias is illustrated in Figure 68 and Figure 69 where we use a regression and a classification example. Figure 68 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  $h_\theta$  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

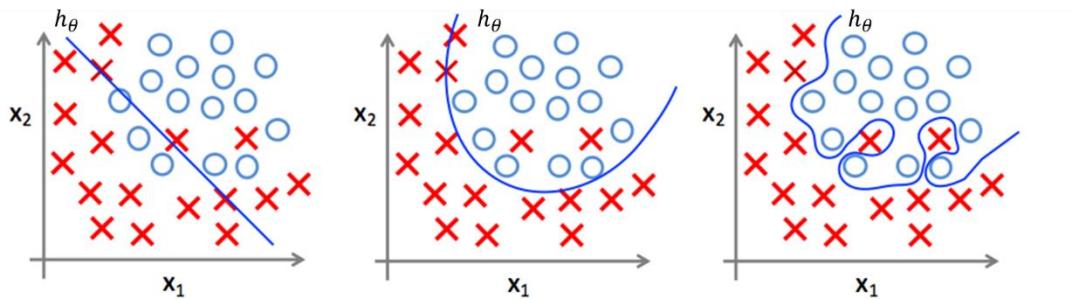
<sup>56</sup> We will use this notation only in this section.

<sup>57</sup> This is frequently abbreviated as i.i.d.

<sup>58</sup> We skip the dependency on the input vector  $\mathbf{x}$  for the sake of clarity.

longer corresponds to the underlying structure of the data i.e., the mapping  $f(x)$ . It is expected that changing (i.e. resampling) the data set will lead to a strong variation of the model  $h_\theta$  i.e., to a high variance. We will call this “overfitting”.

Figure 69 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 69:** 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 <sup>59</sup>Occam’s razor. It states:

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

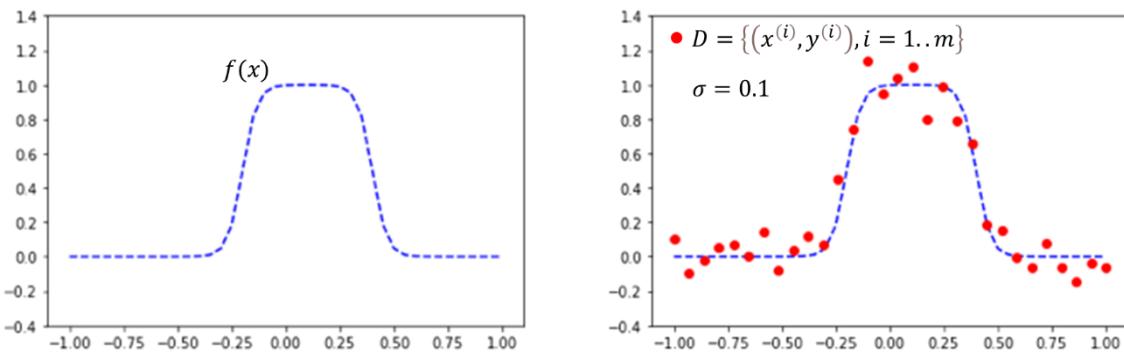
We will now illustrate the concepts of bias and variance introduced above using a simple toy model.

### 3.6.1.1 Illustrative Example

The Figure 70 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<sup>60,61</sup>:

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

Equation 3

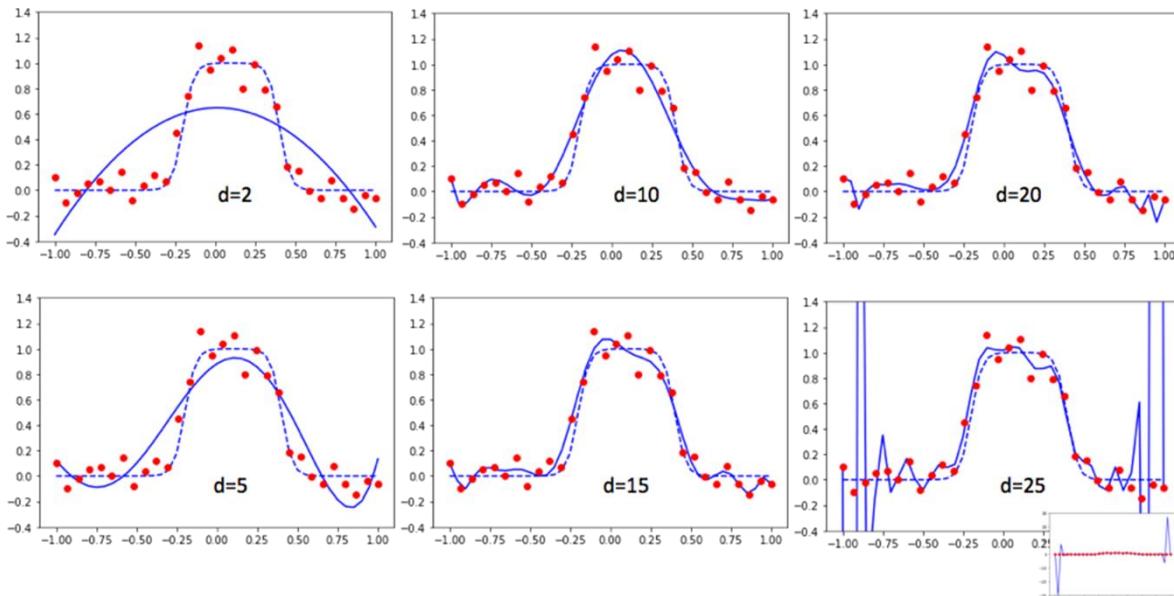


**Figure 70:** Toy model used to illustrate the concepts of bias and variance.

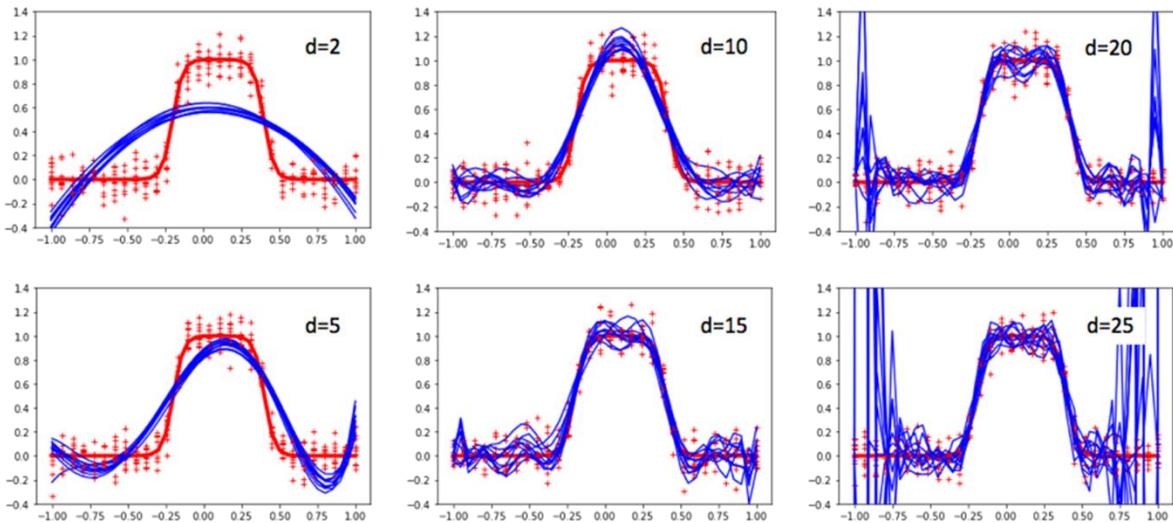
<sup>59</sup> [https://en.wikipedia.org/wiki/William\\_of\\_Ockham](https://en.wikipedia.org/wiki/William_of_Ockham)

<sup>60</sup>  $N(0,1)$  is a Gaussian random variable with zero mean and unit variance.

<sup>61</sup> 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 71:** Increasing the capacity of the model (polynomial degree  $d$ ) will decrease its bias.



**Figure 72:** 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:

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

Figure 71 shows a set of models obtained after optimizing the MSE cost for increasing polynomial degrees  $d = 2, \dots, 25$ . 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 72). 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 73 (left) shows the bias and variance of the different models, evaluated according to the formulas given at the beginning of this chapter 3.6.1, as a function of the model capacity  $d$ . It proves our qualitative observation that bias decreases and the variance increases with increasing capacity.

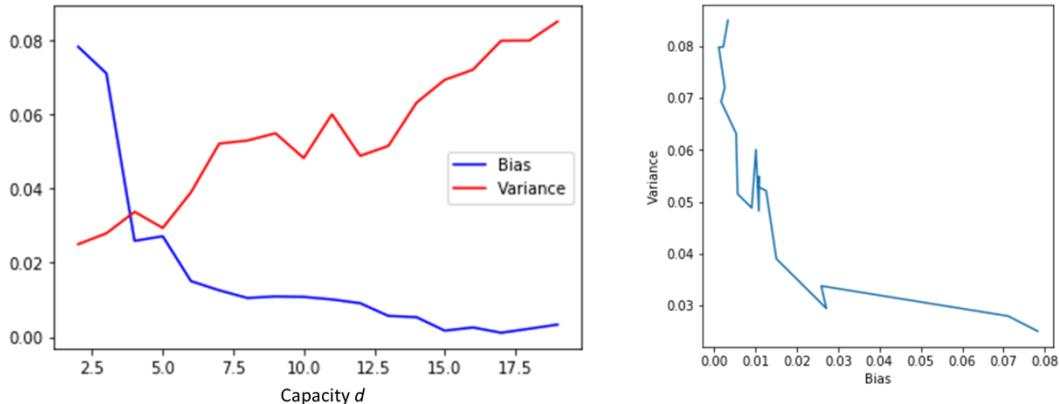


Figure 73: Dependency of bias and variance as a function of the model capacity (polynomial degree  $d$ ).

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.

### 3.6.1.2 Bias-Variance Trade-off

We recall the formulas for the bias

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

and the variance

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

introduced at the beginning of this chapter 3.6.1, 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:

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

For MSE cost function<sup>62</sup> 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:

$$\begin{aligned} \mathbf{E}[(h_{\theta,D} - f)^2] &= \mathbf{E}[(h_{\theta,D} - \mathbf{E}[h_{\theta,D}] + \mathbf{E}[h_{\theta,D}] - f)^2] = \\ \mathbf{E}[(h_{\theta,D} - \mathbf{E}[h_{\theta,D}])^2] &+ \mathbf{E}[(\mathbf{E}[h_{\theta,D}] - f)^2] + 2 \cdot \underbrace{\mathbf{E}[(h_{\theta,D} - \mathbf{E}[h_{\theta,D}]) \cdot (\mathbf{E}[h_{\theta,D}] - f)]}_0 = \\ \text{Var}(h_\theta) &+ \text{bias}(h_\theta)^2 \end{aligned}$$

Thus, we find a compact relationship known as:

#### Bias Variance Trade-off:

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

We added (manually) the irreducible error variance  $\sigma^2$ , which corresponds to the sampling noise (c.f. Equation 3)<sup>63</sup>. This important relation states that the generalization or test error<sup>64</sup> is the sum of two

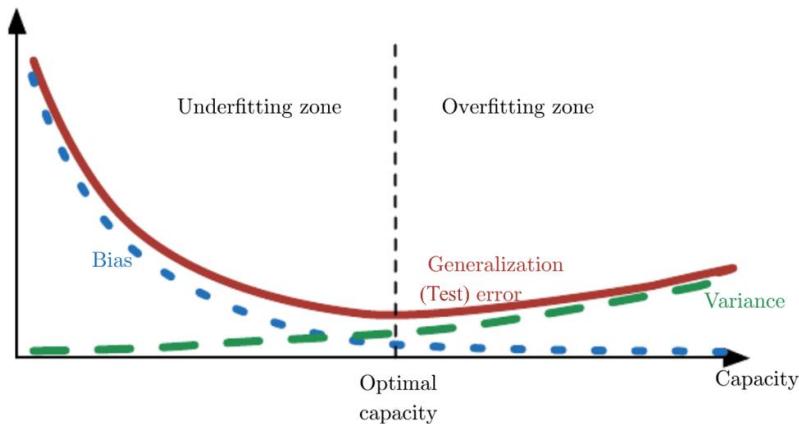
<sup>62</sup> While these considerations are strictly true only for MSE cost (regression problems) they are usually applied also for to CE cost i.e., classification problems. Note nevertheless the explications in section 6.4.4.3 of [14].

<sup>63</sup> The full derivation i.e., including  $\sigma$  can be found here:

[https://en.wikipedia.org/wiki/Bias%E2%80%93variance\\_tradeoff](https://en.wikipedia.org/wiki/Bias%E2%80%93variance_tradeoff)

<sup>64</sup> We use the independent test data set to estimate the generalization error.

terms the bias and the variance<sup>65</sup>. 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 74). 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 74:** The generalization or test error (if measured using MSE) is the sum of bias and variance.

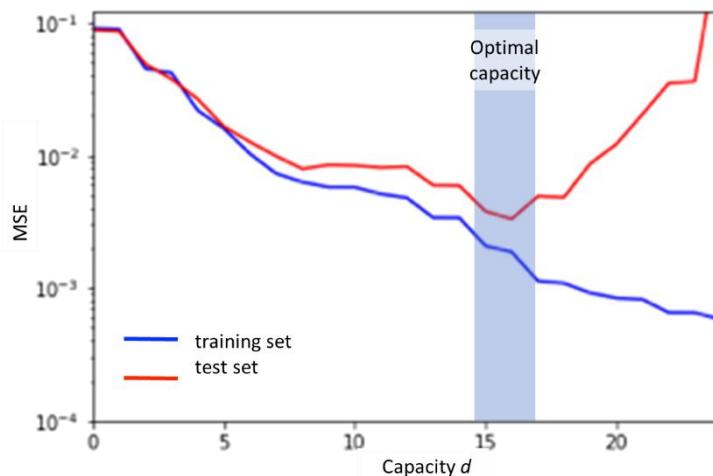
Analysing the MSE curves for the training and the test data set as function of the model capacity  $d$  of our toy model of chapter 3.6.1.1 also reveals the same behaviour (Figure 75). I.e., the test error shows a pronounced minimum in the centre region of the capacity corresponding to the optimal trade-off 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.



**Figure 75:** The MSE curves for training and test set of our toy model (chapter 3.6.1.1) reveal the region of optimal

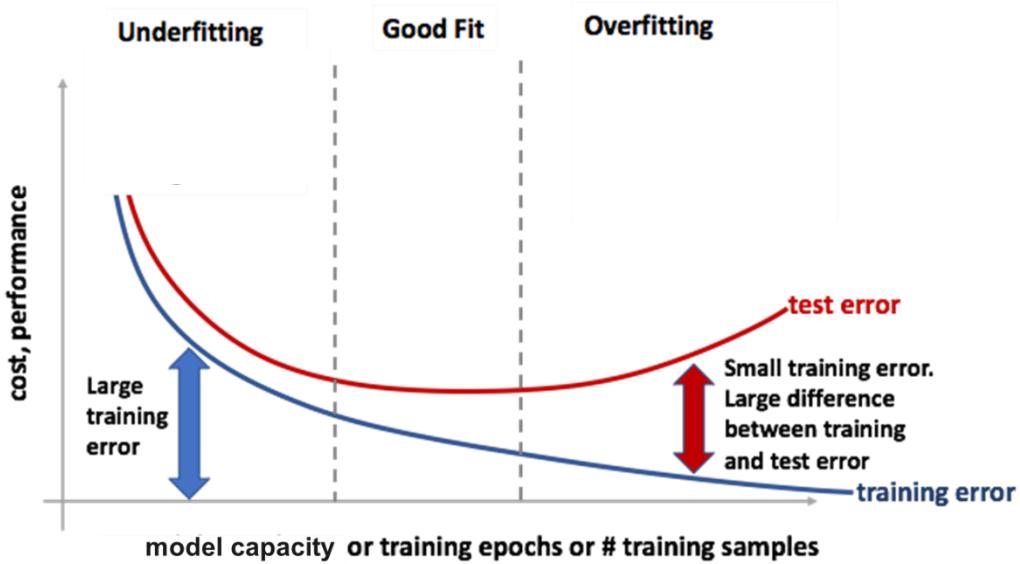
<sup>65</sup> In addition, we have the irreducible error  $\sigma^2$  but which can only be influenced by improving the data sampling process.

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 75 can be generalized as represented in Figure 76. 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 76:** General representation of the possibility to analyse overfitting.

As illustrated in Figure 76 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.

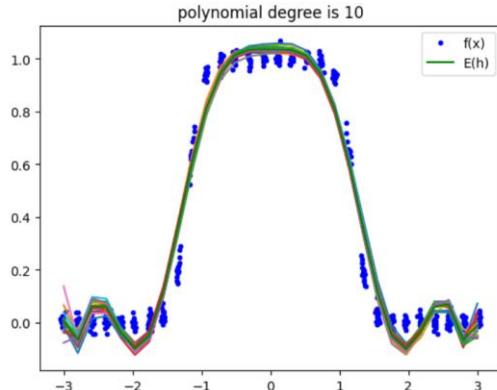
In the next chapter – after the following exercise – we will present suitable strategies to selection the optimal model.

#### Exercise:

We will reproduce the results of the above figures using the provided iPython notebook:

- Use `2.2.bias_variance_tradeoff_stud.ipynb`. Work yourself through the notebook to understand the main concepts:
  - Cell [2]: creation of the "groundtruth" function  $f(x)$  (c.f. Figure 70)
  - Cell [3]: plotting of the function  $f(x)$
  - Cell [4]: The fit routine (you must not understand the details)
  - Cell [5] Application of the fit routine.
- Do some trials in cell [5] using different polynomial degrees for the fitting. You may also play with the noise `sig` applied to the data and the number of data points (`num`) and the number of fits (`num_trials`). Compare the results to Figure 71 and Figure 72.  
E.g. for a polynomial of degree 10 with the "default" settings (as given in the notebook), the

result looks as follows:



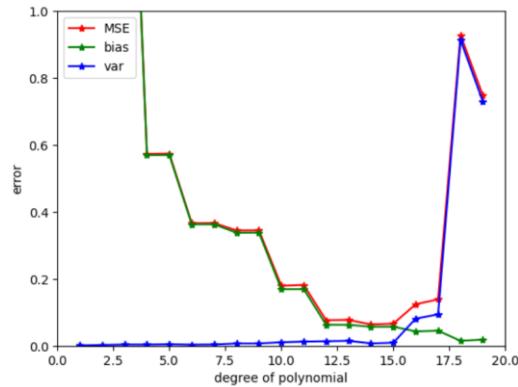
- Now complete the code at the bottom of cell [4] in the sections

```
### START YOUR CODE ###  
...  
### END YOUR CODE ###
```

You are supposed to determine the three quantities (c.f. 3.6.2.2):

- MSE (mean squared error)
- bias
- variance

If this is done correctly you can prove the bias-variance trade-off theorem in cell [6]. The result should look similar to this:



### 3.6.2 Model Selection Process

From the discussion in the previous paragraph in particular Figure 76 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 3. 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 determined by optimising the cost on the training set.

Examples of hyperparameters are:

- Learning rate  $\alpha$
- Batch size  $b$
- Activation function (Table 6)
- 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 3.6.1.1)
- ...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.

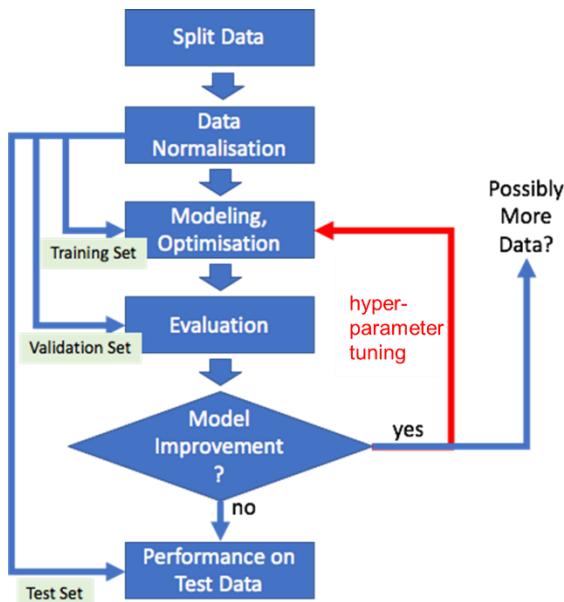
Training Set	Subset of trustable <sup>1)</sup> data used to train the model (choose the parameters that lead to a small cost). Should not be used for evaluating the model.	60%	98%
Validation Set	Subset of trustable data used to select models e.g., by selecting the best hyperparameters of the model.	20%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

**Table 3:** The available data set is split in three parts, training, validation, and test set.

<sup>1)</sup>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 ( $\geq 10^6$ ) datasets (3<sup>rd</sup> and 4<sup>th</sup> column).



**Figure 77:** 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.

Based on these three data sets we can now formulate the standard process for Model Selection in its general form (Figure 77) which works as follows:

- We split the data according to Table 3 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  $h_\theta$  that depends on a set of model parameters  $\theta$  and additional hyperparameters.
- The model parameters  $\theta$  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 76).

- If model improvements seem possible the hyperparameters (e.g., learning rate  $\alpha$ ) 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.

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.

### 3.6.2.1 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 78 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 determined giving an idea on how confident the values are. The model leading to the best performance, among the  $k$  validation folds possible, is retained.

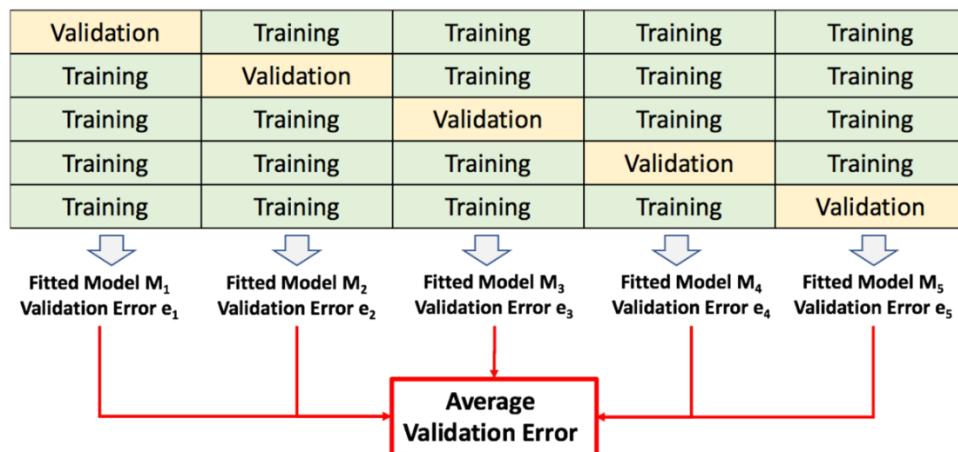
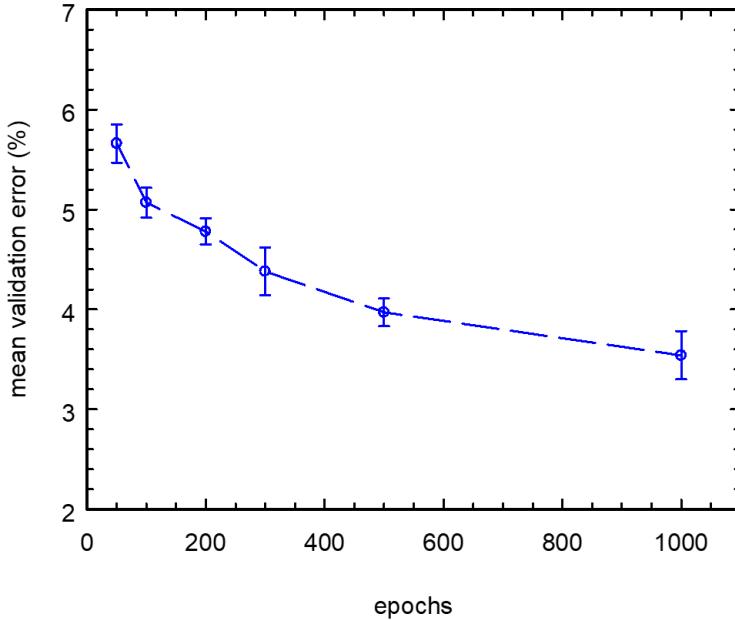


Figure 78: The idea of cross validation using  $k = 5$  folds.

The following <sup>66</sup>Figure 79 shows a simple example of hyperparameter tuning using the Generalized Perceptron on the FashionMNIST data. The x-axis corresponds to the number of epochs chosen. Cost function was MSE and the learning rate  $\alpha = 0.5$ . The y-axis shows the mean validation error – over the k-folds using  $k = 5$  – including the error bars. The bars are an estimator for the generalization error.

<sup>66</sup> This figure will be part of practical work 02.

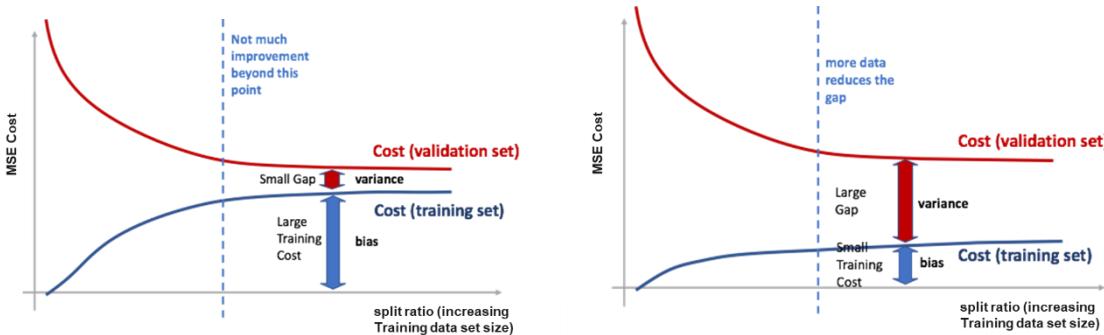


**Figure 79:** Example of hyperparameter tuning (#epochs) using the Generalized Perceptron on the FashionMNIST data (MSE cost function , learning rate  $\alpha = 0.5$ ).

### 3.6.2.2 Selecting a Split Ratio

In Table 3 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 80:** Behaviour of training and validation cost as a function of the split ratio<sup>67</sup>.

This is illustrated for two cases in Figure 80<sup>68</sup>. 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 high variance i.e., the overfitting case is represented. As for Figure 76 this shows that the detailed observation and study of such *learning curves* provides precious information on the status of the model training process.

<sup>67</sup> C.f. [http://www.holohouse.org/mlclass/17\\_Large\\_Scale\\_Machine\\_Learning.html](http://www.holohouse.org/mlclass/17_Large_Scale_Machine_Learning.html)

<sup>68</sup> C.f. also a similar discussion in [1], Figure 5.4, p.114.

### 3.7 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”)<sup>69</sup>:

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

Equation 4

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.

#### 3.7.1 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:

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:
 
$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \cdot (h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}) - y^{(i)}) \cdot \mathbf{x}^{(i)}$$

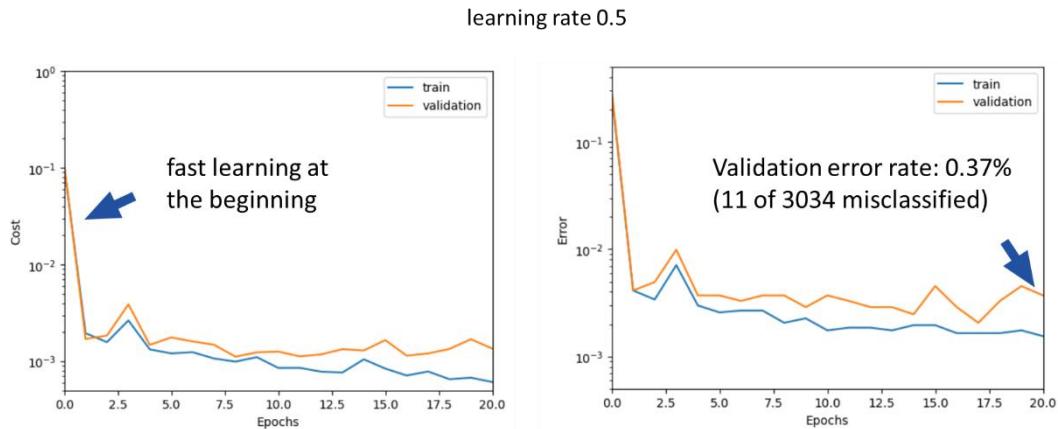
$$b_{t+1} = b_t - \alpha \cdot (h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}) - y^{(i)})$$
3. Stop when the change in parameter vector update step ( $\boldsymbol{\theta}_t \rightarrow \boldsymbol{\theta}_{t+1}$ ) is small.

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.

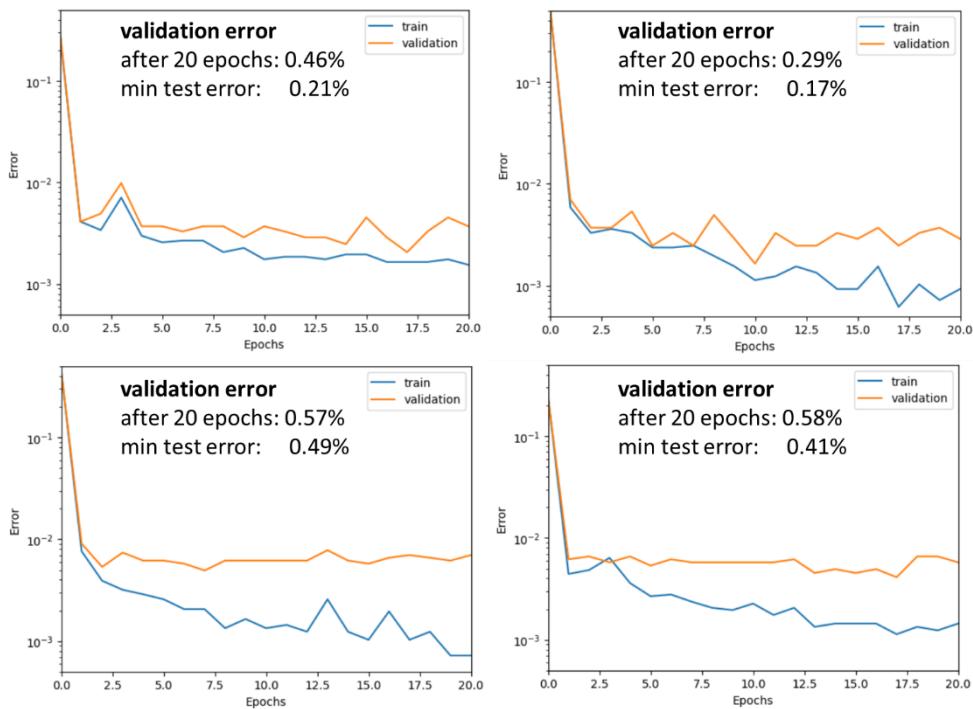
#### Results for Binary Classification on MNIST Dataset

Figure 81 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 (total number of ‘1’ and ‘7’) update steps of one sample each have been performed.

<sup>69</sup> 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”.



**Figure 81:** Results of SGD for CE cost and error rate for binary classification of digits (1,7) on MNIST data set.



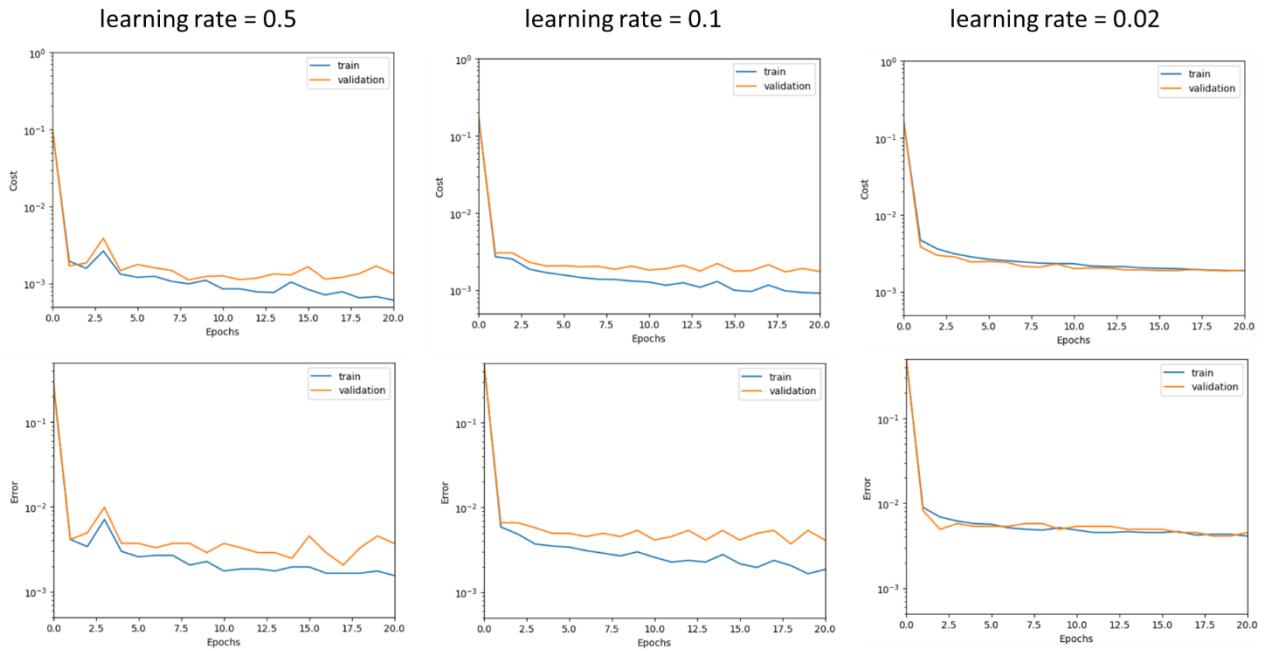
**Figure 82:** Comparison of error rates for different runs using SGD (details see text).

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 observe a fast learning for the first few epochs at the beginning of the optimization. When comparing with Batch GD (Figure 59) we observe 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<sup>70</sup>.

When performing several runs even with identical parameter settings (Figure 82) 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 83 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.

<sup>70</sup> This is particularly true when smoothed learning curves are computed (as e.g. done in the frameworks, c.f. chapter 6) because this requires the evaluation of the entire sum over all m samples both for cost function and error rate.



**Figure 83:** Reducing the learning rate for SGD will reduce the fluctuations of both cost function and error rates.

#### Exercise:

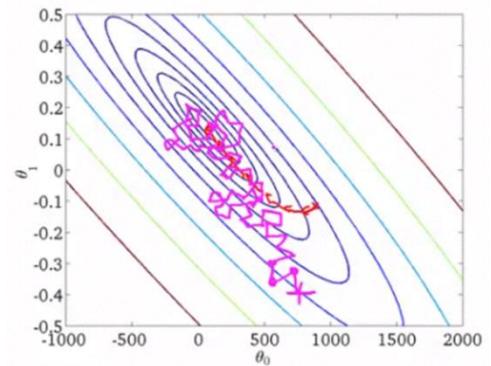
In groups of two discuss the following questions:

- 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  $\alpha$  (Figure 83)?

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

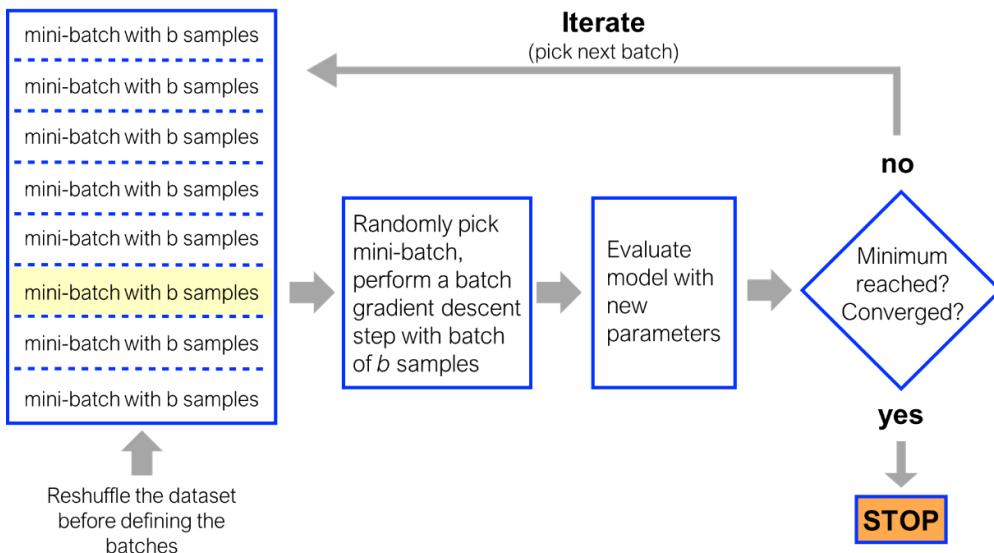
- SGD needs less epochs since parameters are updated for each training sample.
- It can handle very large sets of data and is great for learning on huge datasets that do not fit in memory (“out-of-core learning”).
- It allows for *incremental* learning (“online learning”) i.e., on-the-fly adjustment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

#### Disadvantage

- SGD cannot easily be parallelised.

### 3.7.2 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 84. 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 4) only over the samples contained in the batch. One epoch consists of one loop over all batches.

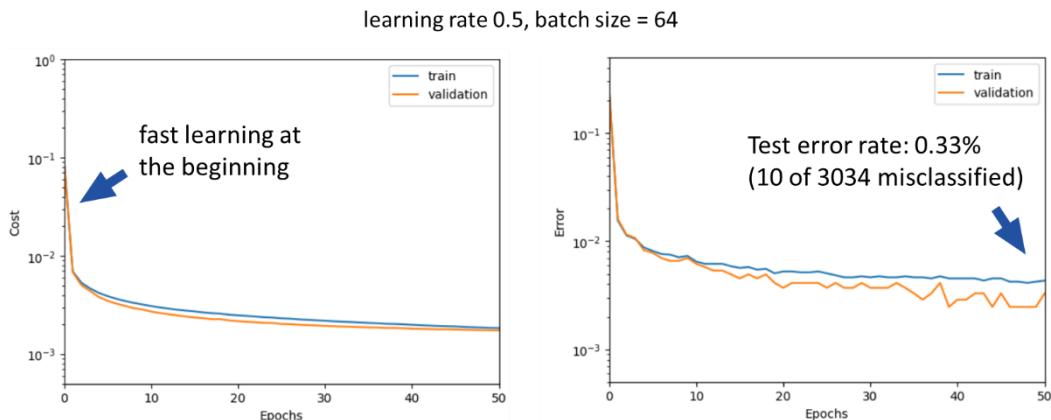


**Figure 84:** Schematic presentation of MBGD (details see text).

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  $\theta_0$  for the parameter vector  $\theta_0 = (\mathbf{w}_0, b_0)$ .  
(e.g., random values or all 0)
2. Iteratively update the parameter vector  $\theta = (\mathbf{w}, b)$  by
  - a. Selecting one mini batch with indices  $i_1, i_2, i_3, \dots$  i.e.,  $(\mathbf{x}^{(i_1)}, y^{(i_1)}), (\mathbf{x}^{(i_2)}, y^{(i_2)}), (\mathbf{x}^{(i_3)}, y^{(i_3)}), \dots$  randomly
  - b. Step in the negative gradient direction according to:
 
$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \cdot \frac{1}{b} \sum_{j=1}^b (h_\theta(\mathbf{x}^{(i_j)}) - y^{(i_j)}) \cdot \mathbf{x}^{(i_j)}$$

$$b_{t+1} = b_t - \alpha \cdot \frac{1}{b} \sum_{j=1}^b (h_\theta(\mathbf{x}^{(i_j)}) - y^{(i_j)})$$
3. Stop when the change in parameter vector update step ( $\theta_t \rightarrow \theta_{t+1}$ ) is small



**Figure 85:** Results of CE cost and Error rate for binary classification of digits (1,7) on MNIST data set.

The Figure 85 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 observe 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 ~0.33% was reached (10 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”.

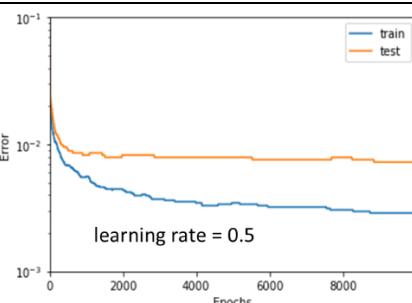
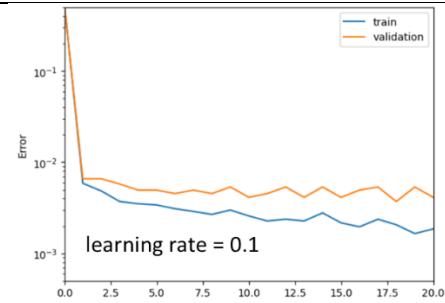
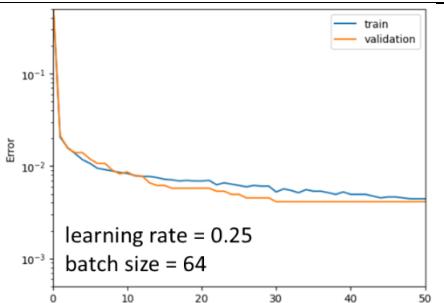
#### Advantages

- 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 4 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 4:** Summary of the differences between BGD, SGD, and MBGD. Note the difference in learning rate.

#### Exercise:

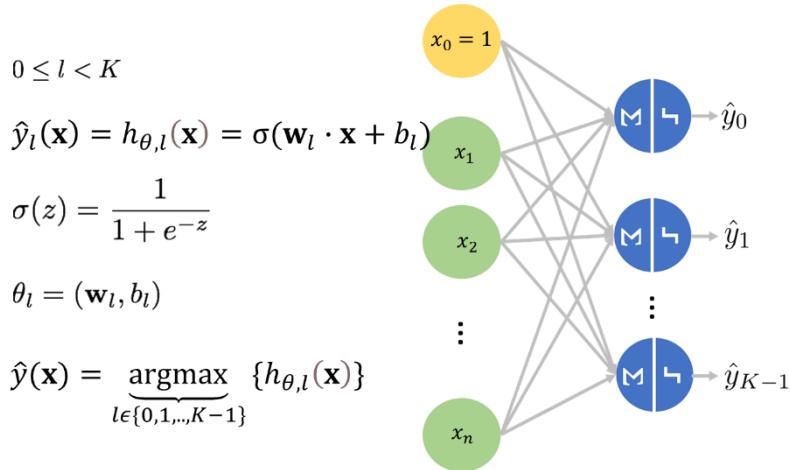
Using the provided iPython notebook perform the following tasks:

- Use `3.0.sgd_mbgd_bgd_binary_stud.ipynb` to familiarise yourself with the differences between BGD, MBGD and SGD.
- Study the class `MiniBatches` in cell [9]. It provides the possibility to split the full training set into a number of equally sizes batches of size `batch_size`. Note that the set is always shuffled randomly before the batches are defined.
- Now observe the usage of the class `MiniBatches` in the `optimise` method of class `NeuralNetwork`. Be sure to understand the concept of an epoch for MGD and SGD: it represents the loop over all batches i.e., the full training set.
- Move on to cell [13] (Sample execution of Neural Network) and perform the training with the given parameters:
  - training scheme: BGD
  - epochs: 400
  - learning rate: 0.5
 Start changing these parameters to implement MBGD and SGD and try to reproduce some of the findings from Table 4

### 3.8 Multi-Class Classification and Softmax Activation

So far, we restricted our models to binary classification as represented in Figure 49. We now want to generalise our task to  $K$  independent classes as shown in Figure 86 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  $\theta_l = (\mathbf{w}_l, b_l)$ . 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  $0 \leq l < K$ .

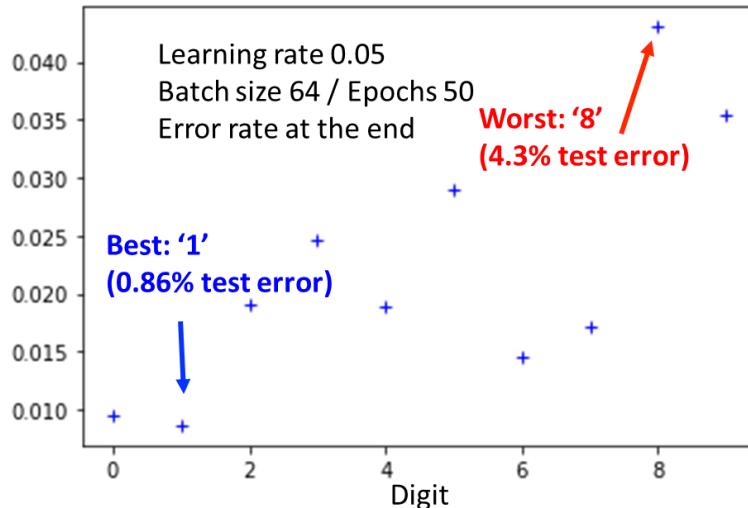


**Figure 86:** Generalisation of the classification task to  $K$  independent classes (details see text).

Figure 87 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

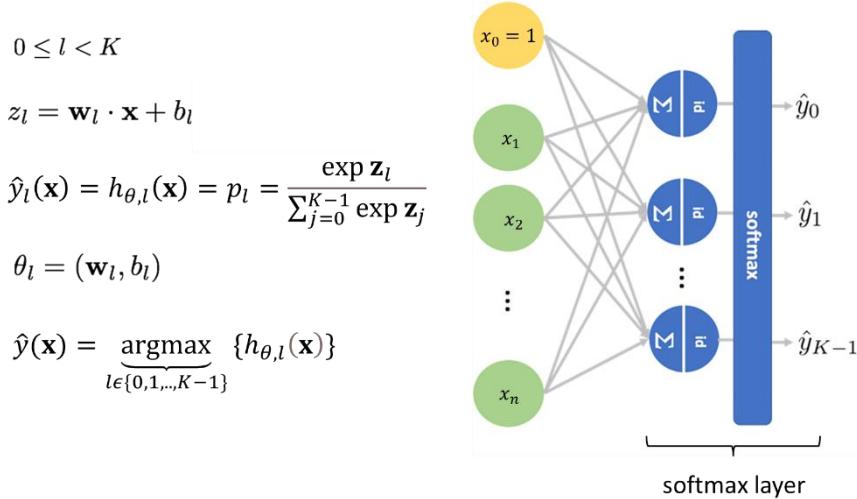
$$\operatorname{argmax}_l (h_{\theta_l}(\mathbf{x}))$$

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 87:** 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 86 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 88).



**Figure 88:** 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 88) and normalize the  $K$  outputs (given by the Logits  $z_l = \mathbf{w}_l \cdot \mathbf{x} + b_l$ )<sup>71</sup> 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:

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

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:

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

### 3.8.1 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  $\theta_k$  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:

<sup>71</sup> 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  $\mathbf{x}$ .

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

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  $\mathbf{w}_l$ :

$$\frac{\partial}{\partial \mathbf{w}_l} z_j = \frac{\partial}{\partial \mathbf{w}_l} [\mathbf{w}_j \cdot \mathbf{x}^{(i)} + b_j] = \mathbf{x}^{(i)}$$

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:

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

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

And the corresponding GD update steps:

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

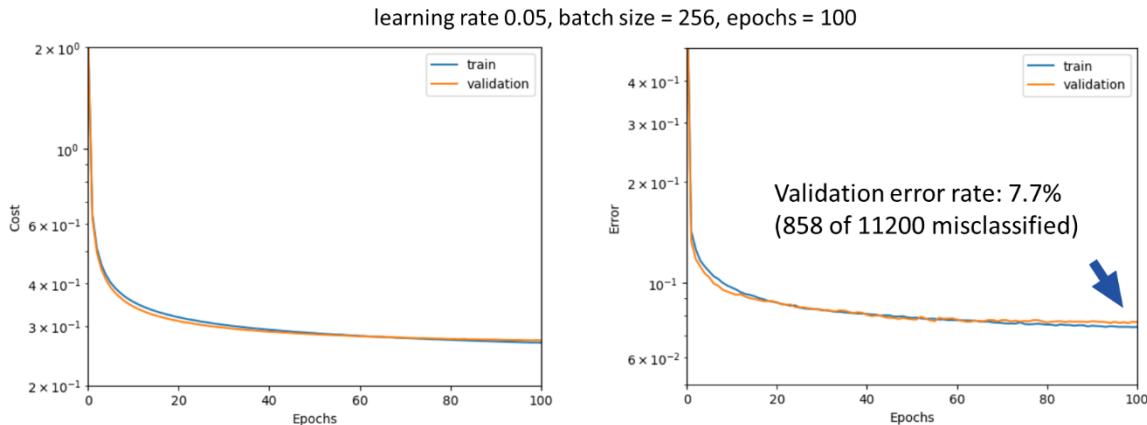
$$b_l \leftarrow b_l - \alpha \cdot \nabla_{b_l} J_{CE}(\mathbf{w}, b)$$

### 3.8.1.1 Results for MNIST Dataset

In Figure 89 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 validation error rate is 7.7%.

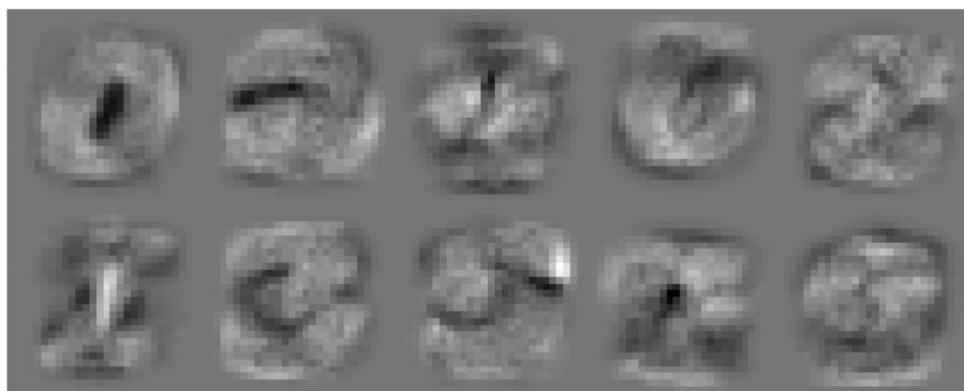


**Figure 89:** Multi-class classification using MBGD on MNIST dataset using a Softmax layer.

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

**Figure 90:** Set of 100 misclassified MNIST digits, in the centre the correct, to the right the false labels.

The Figure 90 shows 100 of the 1123 misclassified images with the correct (middle) and wrong (right) labels given. Finally, in Figure 91 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 91. Apparently, the weight vectors have learned some of the structure of the respective digit.



**Figure 91:** The weight vectors  $\mathbf{w}_l$  ( $l = 0..9$ ) (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%<sup>7</sup>. To achieve this performance, larger

models with higher representational capacity are needed including (much) more model parameters. This will be the topic of the following chapter.

## 4 Deep Neural Networks

Based on the work done in the previous chapters we can now generalize our consideration on multi-layer architectures. A deep feedforward network, also called feedforward neural network, or multi-layer Perceptron (MLP), which is simply a stack of single layer LTUs, is the quintessential deep learning model. 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.

In Figure 92 the simplest MLP-architecture is shown with just one additional layer – between the input and output layer. These intermediate layers are called hidden layers and are characterized by their respective number of neurons. The represented hidden layer is of fully connected type<sup>72</sup> 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 (●-●).

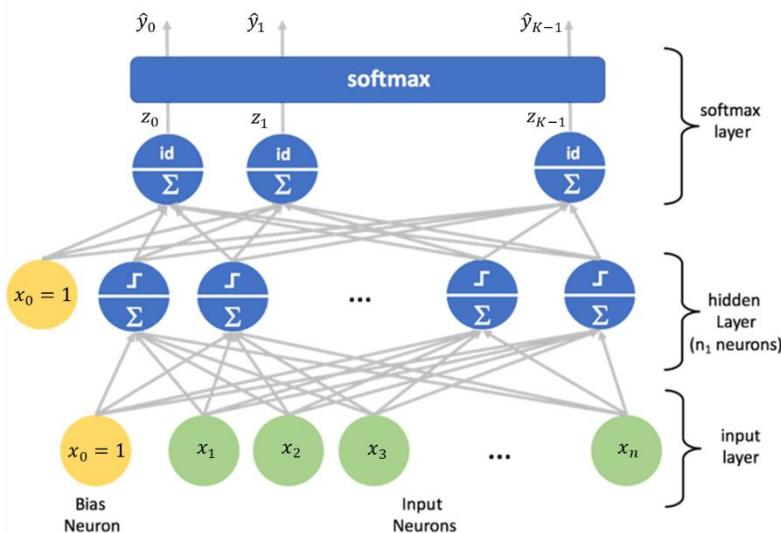


Figure 92: A Multi-Layer Perceptron with one hidden layer.

In the present chapter we will extend this architecture to an arbitrary number of hidden layers and including the formulation of the gradient decent update rule. This will bring us to the so-called back-propagation algorithm. But before that, we deal with three other topics. We will first discuss a common problem of all Machine Learning Algorithms (section 4.1), the so-called *Curse of Dimensionality*. Then we will introduce the set of relevant activation function (section 4.2) and finally discuss the representational capacity of the MLP from a theoretical point of view (4.3).

### 4.1 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 68, Figure 69). 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

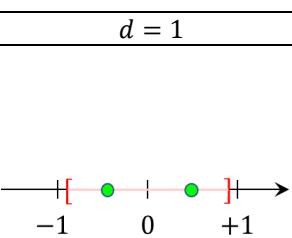
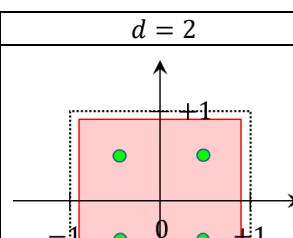
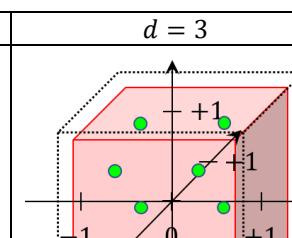
<sup>72</sup> We will introduce other hidden layer types in chapter 6, when we discuss the CNN-architecture.

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<sup>73</sup>:

“... 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 5.

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

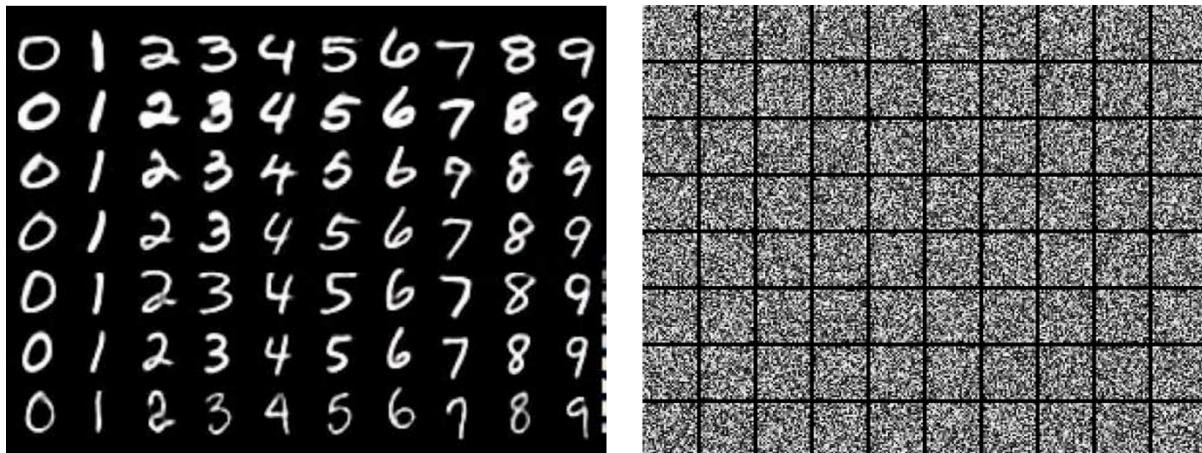
**Table 5:** Illustration of the effect of increasing dimension on different parameters (details see text).

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  $2/\sqrt{d}$  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

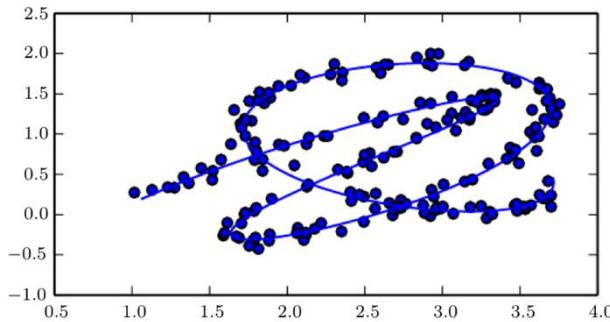
<sup>73</sup> [https://en.wikipedia.org/wiki/Curse\\_of\\_dimensionality](https://en.wikipedia.org/wiki/Curse_of_dimensionality)

way allow to apply the local smoothness assumption.



**Figure 93:** 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 93, right). This situation is typically illustrated as in Figure 94, 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 sub-manifold, but not in the original input space.



**Figure 94:** 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:

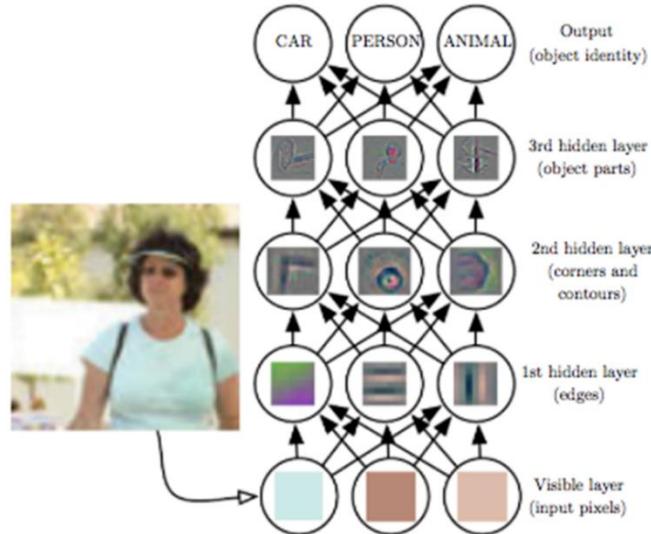
- We assume that the data was generated by the composition of factors or features, potentially at multiple levels in a hierarchy.
- Learning then involves discovering a set of underlying factors of variation that can be described in terms of other, simpler underlying factors (arranged in a hierarchy).

One prominent example cited frequently in that context are convolutional neural networks (Figure 95). 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<sup>74</sup>.

<sup>74</sup> C.f. also the discussion on biological NNs in chapter 2.1, in particular Figure 29.

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 95: CNN extract features at different levels of hierarchy.**

## 4.2 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 3.8). 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.

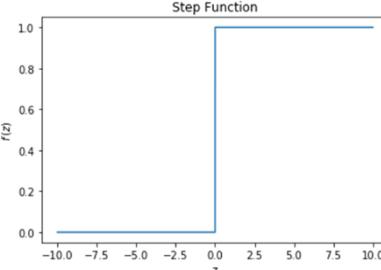
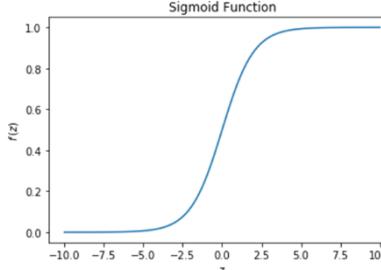
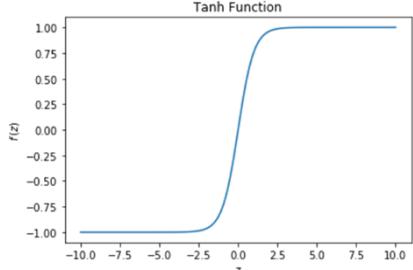
$$h(\mathbf{x}) = f(z) = f(\mathbf{w} \cdot \mathbf{x} + b)$$

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 88). 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.

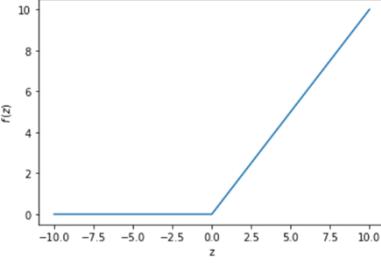
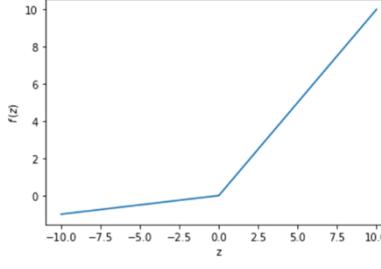
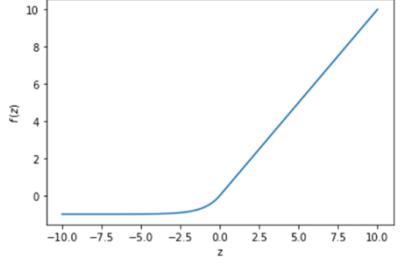
The following Table 6 gives an overview of commonly used activation function and their major characteristics.

Heaviside	Sigmoid	Hyperbolic Tangent
 <p>Step Function</p>	 <p>Sigmoid Function</p>	 <p>Tanh Function</p>
$f(z) = \begin{cases} 1 & (z \geq 0) \\ 0 & (z < 0) \end{cases}$	$f(z) = \frac{1}{1+\exp(-z)}$	$f(z) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}$

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
 <p>ReLU</p>	 <p>LeakyReLU</p>	 <p>ELU</p>
$f(z) = \max(0, z)$	$f(z) = \max(\alpha z, z)$	$f(z) = \begin{cases} \alpha (\exp(z) - 1) & (z < 0) \\ z & (z \geq 0) \end{cases}$

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 6: Overview of all activation functions commonly used in NN architectures.

### 4.3 Representational Capacity of the Multi-Layer Perceptron

Before we start to look in detail on the MLP-architecture we want to discuss, which kind of problems we can solve with a given type of architecture. In other words, we would like to know the representational capacity of a given architecture, depending on the number of hidden layers and the corresponding neurons. It turns out, that a MLP with one hidden layer, already has a very high representational capacity which is represented by the “Universal Approximation Theorem” formulated below. While this theorem is expressed as a regression theorem, we can apply it also to a classification task if we formulate it as such. Nevertheless, to develop some intuition on the representational capacity of an MLP with respect to classification problems we will first consider the possible decision boundaries (in an arbitrary features space), that can be produced by MLPs using – for simplicity – the Heaviside activation function.

#### 4.3.1 Decision Boundaries produced by MLPs

It turns out, that an MLP with two hidden layers can produce any arbitrary decision boundary. With one single hidden layer decision boundaries surrounding any convex region are possible. However, it should be noted that this is not the most general case. In Figure 96 an illustration is given for a two-dimensional space summarizing these results. We will now have a look at the three individual cases.

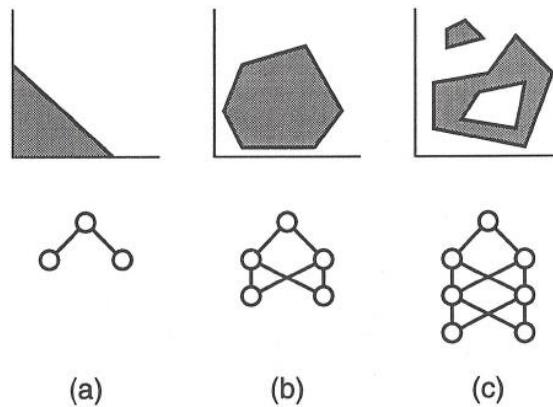
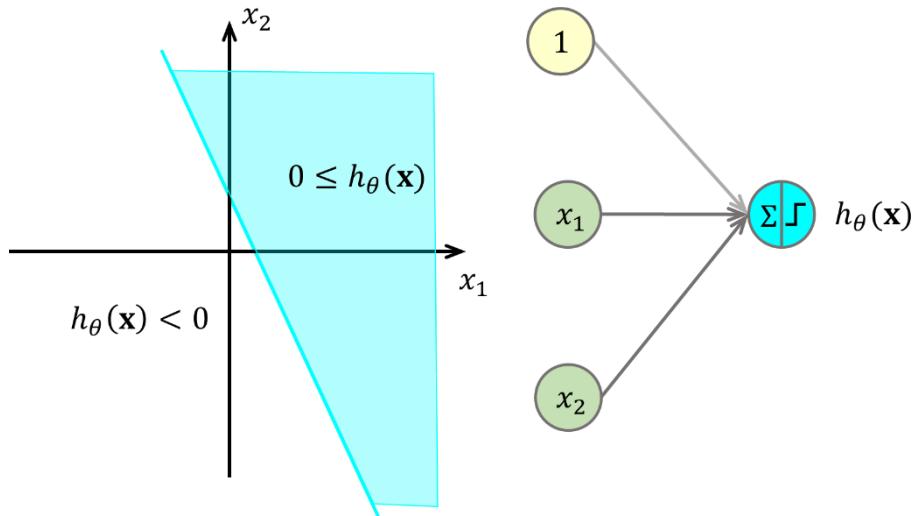


Figure 96: The number of hidden layers – (a)= 0, (b) = 1, (c) = 2 – determines the capacity of the model to form decision boundaries [12] (details c.f. text).

#### 4.3.1.1 Decision Boundary for Zero hidden Layer – Perceptron

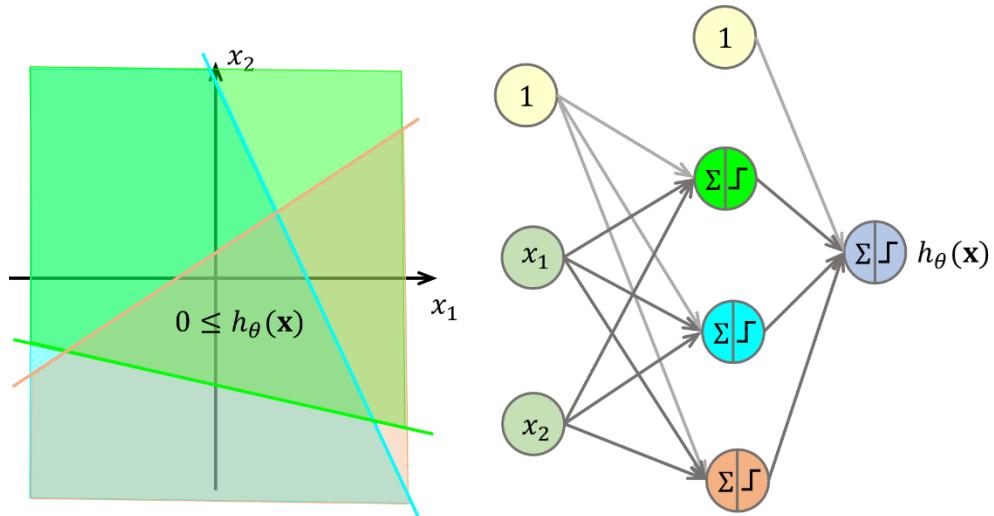
We already know from the discussion on the Perceptron (section 3.3) that its decision boundary is a simple hyperplane in the respective feature space. This can be considered as the case with zero hidden layers. We recall this case because it is the basis for the understanding of the two following, more complex cases.



**Figure 97:** For the Perceptron the decision boundary is a hyperplane.

#### 4.3.1.2 Decision Boundary for One hidden Layer

Now we consider the case of one hidden layer as represented in Figure 98 below. Each neuron of the hidden layer will define one decision boundary, represented by its respective colour. If the bias of the output neuron is set to  $-3$  the output  $h_\theta(\mathbf{x})$  will only be equal or larger than zero if all three hidden neurons are equal to one. This corresponds to the triangle defined by the three decision boundaries, which is the intersection of the three corresponding decision regions.

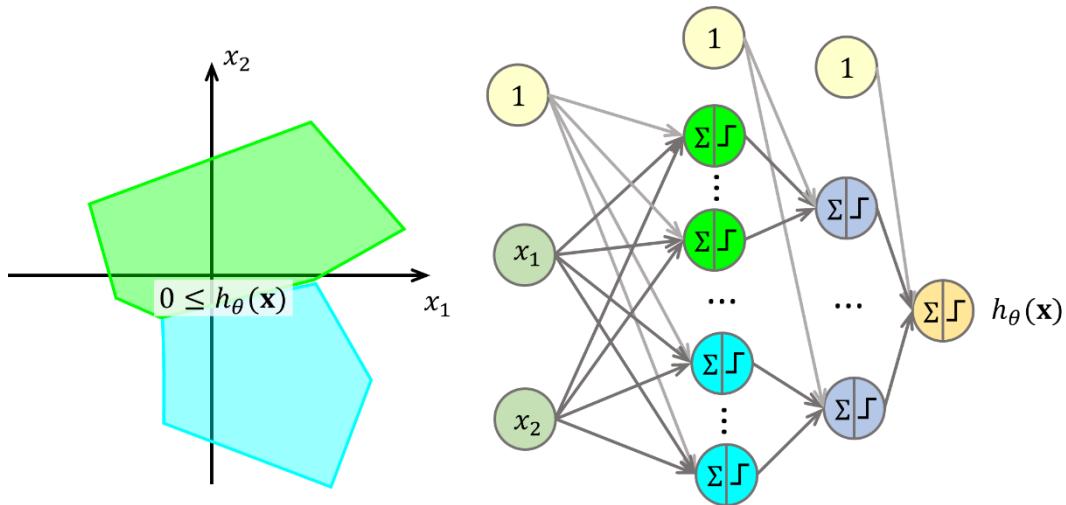


**Figure 98:** For one hidden layer decision boundaries surrounding any arbitrary convex region can be produced (details c.f. text).

#### 4.3.1.3 Decision Boundary for Two hidden Layer

Finally, we consider the case of two hidden layers as represented in Figure 99 below. We group the neurons of the hidden layer in groups, represented by the different colours. Each group – the vertical dots indicate addition neurons being part of that group – will, together with the corresponding neuron of the second hidden layer, define a decision boundary for a convex region. Just as explained in the

previous chapter. The horizontal dots indicate that there might be additional groups representing additional convex regions. Now, if the bias of the output neuron is set to  $-1$ ,  $h_\theta(\mathbf{x})$  will be equal or larger than zero if at least one region contributes. Thus, the overall decision region corresponds to the union of all convex regions and we can therefore form any arbitrary region.



**Figure 99:** For two hidden layers decision boundaries surrounding any arbitrary region can be produced (details c.f. text).

It is immediately obvious, that this kind of constructive prove is not very valuable for practical applications, because the decision boundaries would have to be known in advance. In addition, it makes no statement about the necessary number of neurons in the hidden layer. Nevertheless, it gives an intuition on the representational capacity of the MLP and will raise – later on – the question, why shallow neural networks obtain much smaller accuracy than deep ones.

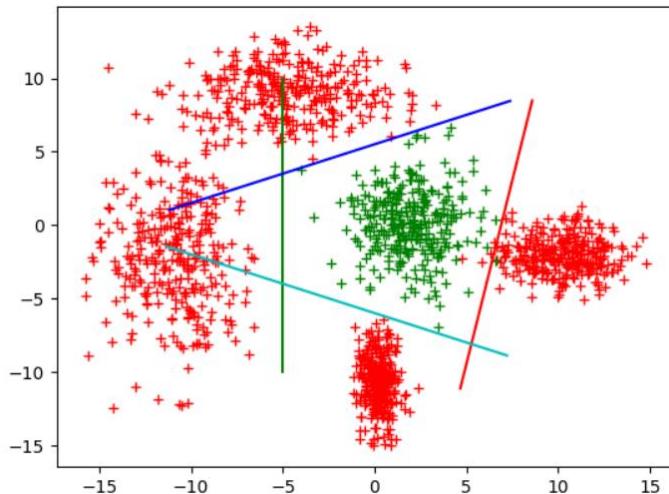
#### Exercise:

We will reproduce the results of the above figures using the provided iPython notebook:

- Use `3.1.decision_boundaries_stud.ipynb`. Work yourself through the notebook to understand the main concepts:
  - Cell [2]: helper functions only -> you can skip.
  - Cell [3]: creation of random point sets (“green”  $\leftrightarrow$  ‘1’ and “red”  $\leftrightarrow$  ‘0’)
  - Cell [4]: definition of class MultiLayerPerceptron; study the comments in the markdown cell above.
  - Cell [5]: Here you are supposed to work. The lists of weights and biases are defined here. An example for two decision boundaries is given.
  - Cell [6]: graphical representation of the result. The top graph shows the ground truth colours of the points sets (-> the goal) the bottom the prediction of the MLP.
  - Cell [7]: a further graphical representation of the decision boundary supposed to help identifying wrong choices in cell [5].
- The goal is to choose additional decision boundaries – corresponding to neurons in the hidden layer – and to optimise their position such that the “green” class is well separated from the “red” one. This is done by completion of the lists of weights and biases defined in cell [5]. The colours of the decision lines – as shown in the output of cell [6] – correspond to the neurons in the hidden layer according to the following order:

Neuron index:	0	1	2	3
Colour:	red	green	blue	cyan

- With some optimisation the result will look like in the following graph.



### 4.3.2 Universal Approximation Theorem

We will now formulate the universal approximation theorem, which is applicable to regression problems. Nevertheless, we can apply it to classification problems. 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 41). The model  $h_\theta(\mathbf{x})$  represents a family of functions that depend on the parameters  $\theta$  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*<sup>75</sup>. A very general statement now proves [13] 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<sup>76</sup>  $f: \mathbb{R}^{n_x} \rightarrow \mathbb{R}^{n_y}$  with arbitrary accuracy<sup>77</sup> – 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?

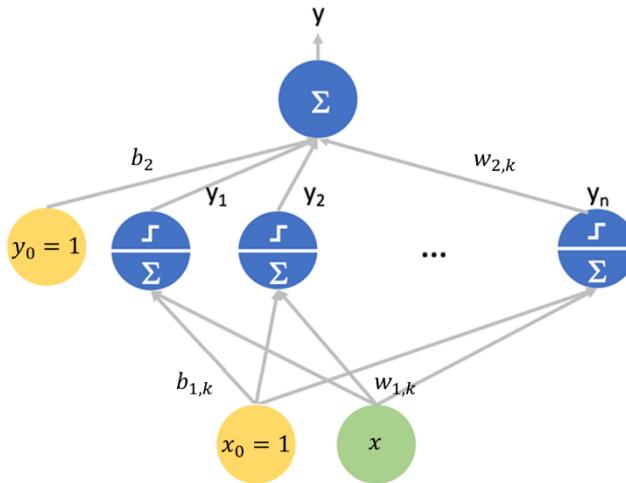
We will now try to develop some intuition on the idea behind the Universal Approximation Theorem by “proving” it for the case of a 1-dimensional function.

<sup>75</sup> This can be considered as a formal definition of this term, which we used so far only “intuitively”.

<sup>76</sup> E.g., any continuous function on compact support.

<sup>77</sup> Accuracy quantified with suitable distance measure (e.g., L<sup>2</sup>-distance ~ integrated mean-square distance).

#### 4.3.2.1 Function Approximation with Sigmoids in 1D



**Figure 100:** Network architecture used to illustrate the Universal Approximation Theorem in 1D.

<sup>78</sup>We will illustrate the Universal Approximation Theorem with a real valued function in one dimension i.e.,  $f: \mathbb{R}^1 \rightarrow \mathbb{R}^1, f(x) = y$ . The model  $h_\theta(x)$  is given in Figure 100 having the following characteristics:

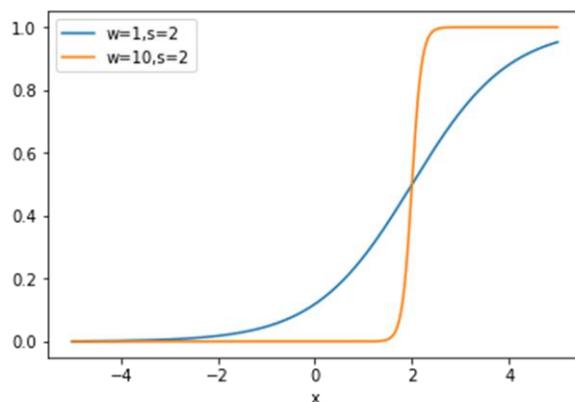
- One-dimensional real-valued input  $x$ .
- One hidden layer with  $n$  neurons and sigmoid activation function, weights  $w_{1,k}$  and biases  $b_{1,k}$ .
- Linear output layer with weights  $w_{2,k}$  and bias  $b_2$ .
- The linear output layer leads to a linear combination of sigmoid functions:

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

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:

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

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 101 below.



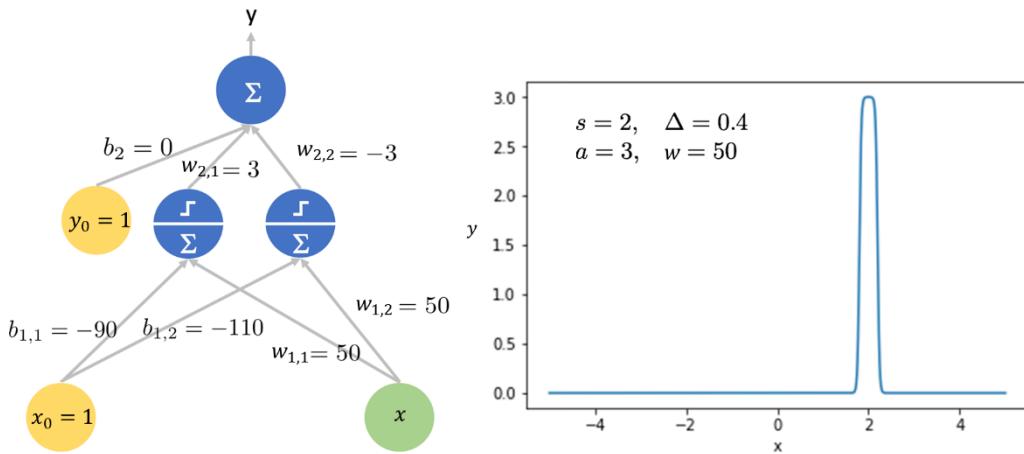
**Figure 101:** Illustration of the effect of the parameter  $w$  on the slope of the step.

<sup>78</sup> The following was inspired by <http://neuralnetworksanddeeplearning.com/chap4.html>.

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

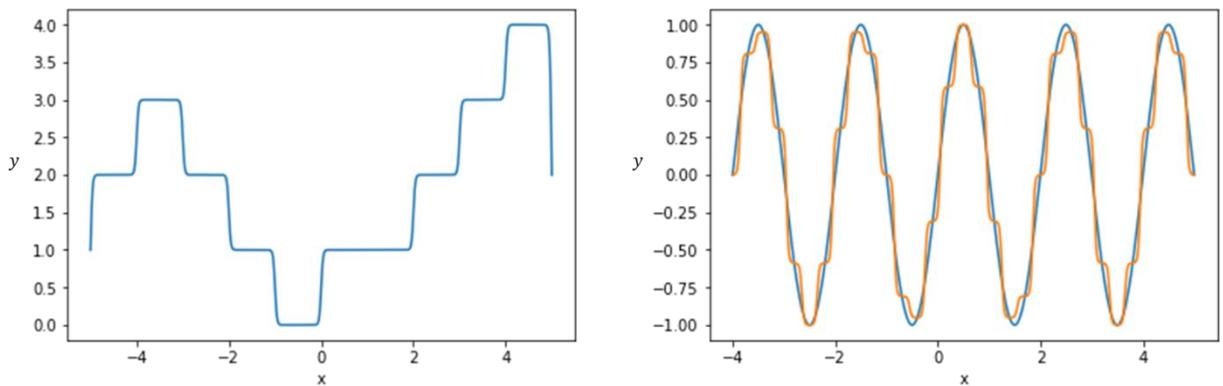
$$a \cdot [\sigma(w \cdot (x - s + \Delta/2)) - \sigma(w \cdot (x - s - \Delta/2))]$$

This will create a peak as shown in Figure 102 (right). The value of  $a$  just scales the final peak height (being 3 in the example). The left-hand side of Figure 102 shows the corresponding model.



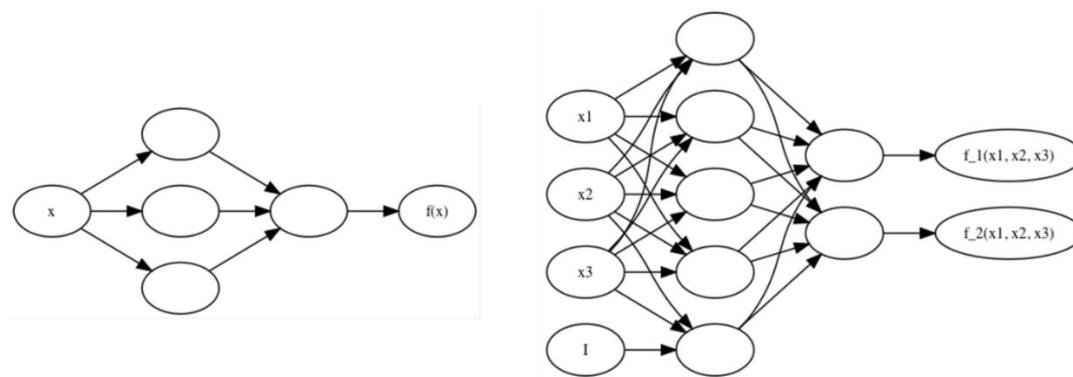
**Figure 102:** Combination of two “steep” step functions allows to construct a peak.

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 103.



**Figure 103:** 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 104 comparing the network structure for the one-dimensional case (left) with the general situation.

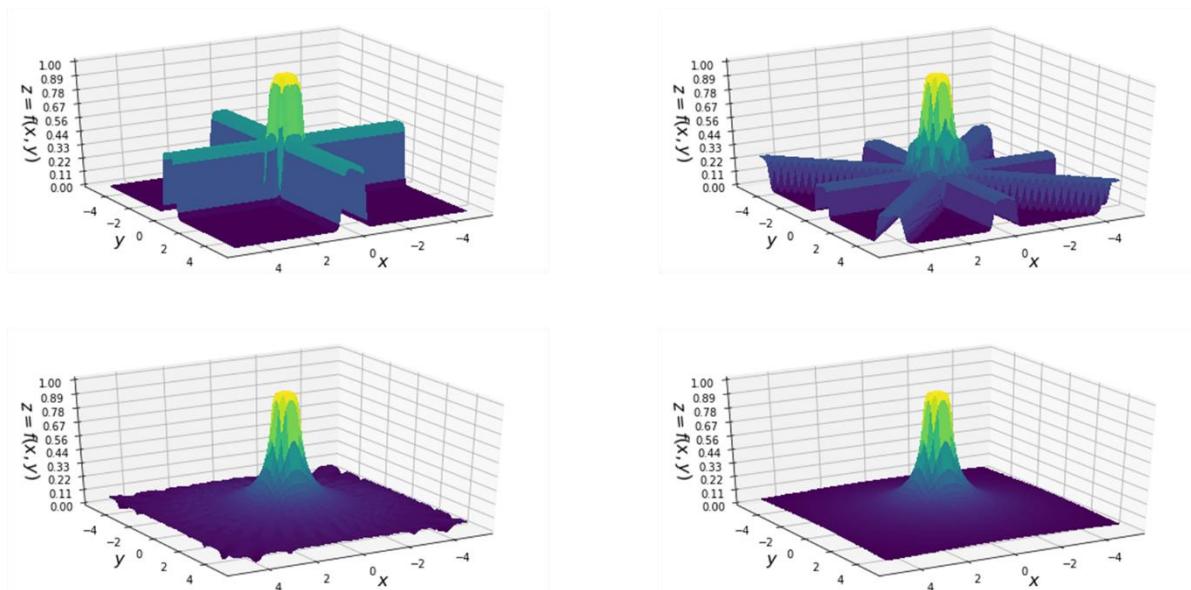


**Figure 104:** A network for a function  $f: \mathbb{R}^1 \rightarrow \mathbb{R}^1$  (left) is easily extend to the general case  $: \mathbb{R}^3 \rightarrow \mathbb{R}^2$  (right).

Figure 105 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 103 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 105:** Illustration of the creation of 2D peak functions with the help of 1D ridges.

## 4.4 Computational Graph

We will now address the problem of formalizing the architecture of the MLP to automate 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<sup>79</sup> could be represented as follows (c.f. Figure 106):

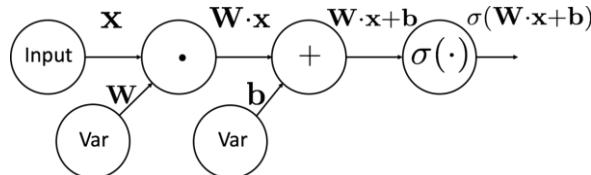
$$g(\mathbf{x}; \mathbf{W}; \mathbf{b}) = \sigma(\mathbf{W} \cdot \mathbf{x} + \mathbf{b})$$

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 106) corresponding to the following variables with given dimensions<sup>80</sup>:

$$\begin{aligned}\mathbf{x}: & n_x \times 1 \\ \mathbf{W}: & n_1 \times n_x \\ \mathbf{b}: & n_1 \times 1\end{aligned}$$

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  $\sigma(\dots)$ .

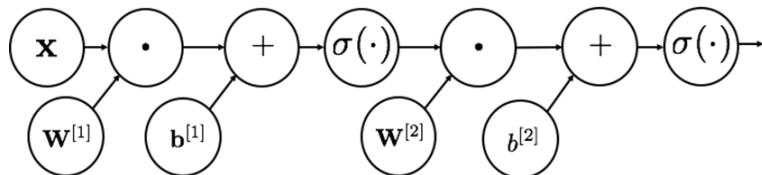


**Figure 106:** A single MLP layer represented as a computational graph.

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  $n_x \times m$ .

Starting with the next higher complexity of a two-layer architecture we obtain<sup>81</sup>:

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



**Figure 107:** Two-layer MLP network represented as graph.

For the respective dimensions we have:

<sup>79</sup> For the moment we will limit ourselves to the so-called "dense" and "Softmax" – for the output – layer types. Other types will be introduced in subsequent chapters.

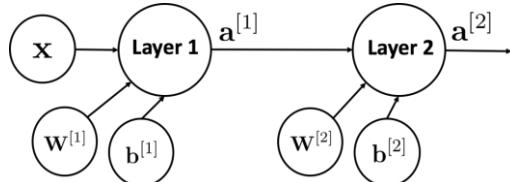
<sup>80</sup> We will eventually formulate all update rules in matrix-notation, so it is essential to be familiar with matrix calculus.

<sup>81</sup> We start numbering the first layer with  $\mathbf{W}^{[1]}$  using index [1] because we refer to it as the "first" hidden layer. The numbering in the programming language (e.g. numpy) will usually start with index 0.

$$\begin{aligned} \mathbf{x}: & n_x \times 1 \\ \mathbf{W}^{[1]}: & n_1 \times n_x \\ \mathbf{b}^{[1]}: & n_1 \times 1 \\ \mathbf{W}^{[2]}: & n_2 \times n_1 \\ \mathbf{b}^{[2]}: & n_2 \times 1 \end{aligned}$$

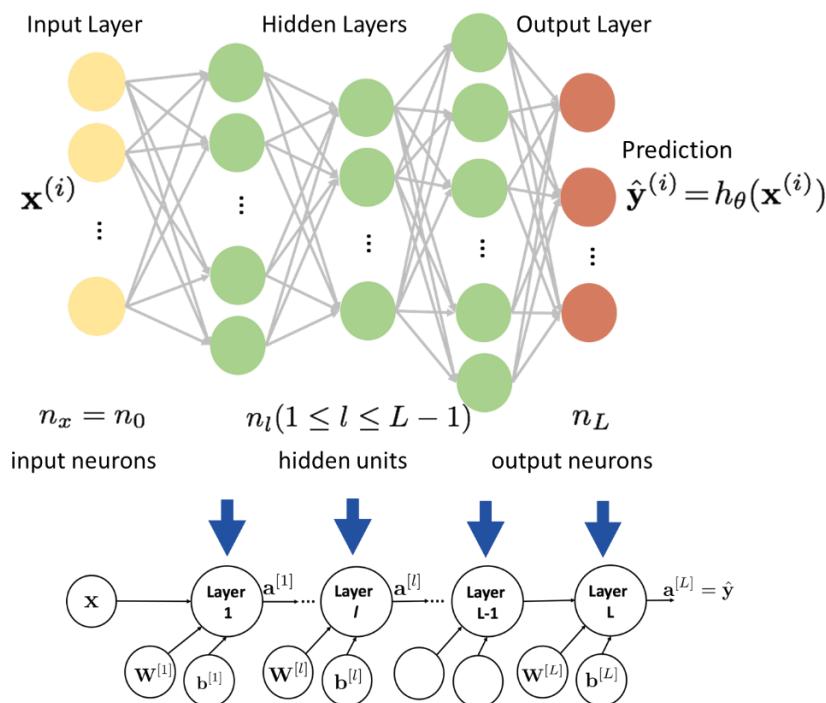
We will however choose a more compact notation convention by collapsing the affine transformation and the application of the activation function into a single node:

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



**Figure 108:** Compact notation with a single node for the operation  $\mathbf{a}^{[l]} = \sigma(\mathbf{W}^{[l]} \cdot \mathbf{x} + \mathbf{b}^{[l]})$ .

It is now straightforward to represent a multi-layer Perceptron with arbitrary number of hidden layers as a computational graph (Figure 109). The input vectors  $\mathbf{x}^{(i)}$  have the size  $n_x \times 1$ . 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  $n_l \times n_{l-1}$ , 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  $\mathbf{a}^{[l]}$ .



**Figure 109:** Representation of a MLP as a computational graph.

For a single layer of an MLP the notation, including the detailed vector and matrix indices, is shown in Figure 110. The dimensions are as follows:

**a**<sup>[l-1]</sup>: n<sub>l-1</sub> x 1  
**W**<sup>[l]</sup>: n<sub>l</sub> x n<sub>l-1</sub>  
**b**<sup>[l]</sup>: n<sub>l</sub> x 1  
**z**<sup>[l]</sup>: n<sub>l</sub> x 1  
**a**<sup>[l]</sup>: n<sub>l</sub> x 1

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

$$\begin{aligned} \mathbf{z}^{[l]} &= \mathbf{W}^{[l]} \cdot \mathbf{a}^{[l-1]} + \mathbf{b}^{[l]} \\ \mathbf{a}^{[l]} &= \sigma^{[l]}(\mathbf{z}^{[l]}) \end{aligned}$$

Equation 5

Here  $\mathbf{z}^{[l]}$  represents the so-called logit i.e., the input to the activation function, which is denoted by<sup>82</sup>  $\sigma^{[l]}$ .

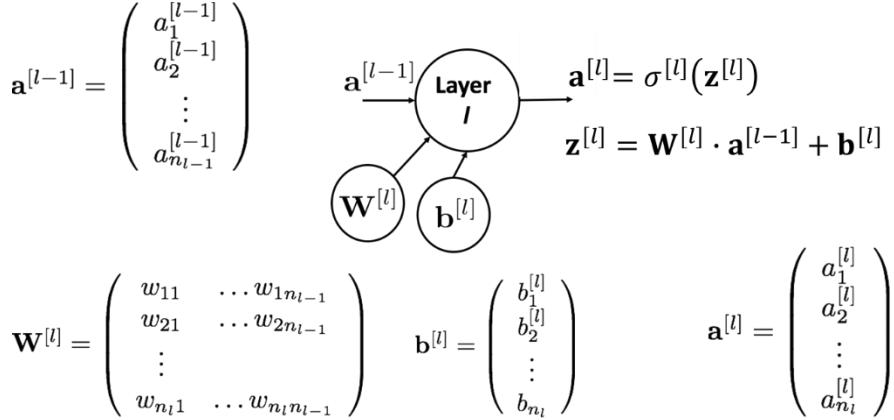


Figure 110: Notation for single layer in MLP with indices explicitly given.

Using this notation the forward pass of our general MLP can be represented as in the following Figure 111:

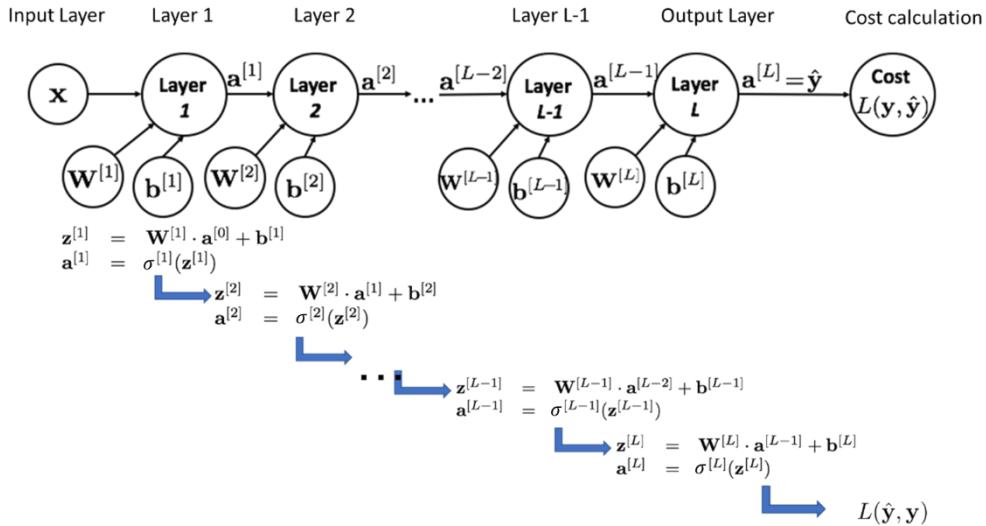


Figure 111: Forward propagation for an MLP with arbitrary number of hidden layers.

### Exercise:

We will illustrate the forward propagation with a simply “toy” mlp:

- Use 4.1.compact-mlp.ipynb. Work yourself through the following cells of the notebook to understand the main concepts:
  - Cell [2]: helper functions only -> you can skip.
  - Cell [3]: creation of class NeuralNetwork (it is meant for illustration purposes not for

<sup>82</sup> We will continue to represent the activation function using the symbol  $\sigma$ . This must not necessarily be the sigmoid function can is a place holder for any choice out of Table 6.

- application).
- Cell [4]: creation of an instance of class `NeuralNetwork` and propagation of single input vector  $\mathbf{x}$ .
  - Observe in detail the sizes of the input vector, the weights and biases and the corresponding activations. Make sure to understand the matrix notation. It will be essential for the understanding of the following!

Figure 111 represents in principle the most general case of an MLP. However, the forward (and the backward step coming next) is usually performed simultaneously on a full mini batch of  $m$  input vectors:

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

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

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

$$\mathbf{Z}^{[l]} = \mathbf{W}^{[l]} \cdot \mathbf{A}^{[l-1]} + \mathbf{b}^{[l]}$$

$$\mathbf{A}^{[l]} = \sigma^{[l]}(\mathbf{Z}^{[l]})$$

**Equation 6**

The dimensions are as follows:

$$\begin{aligned} \mathbf{A}^{[l-1]} &: n_{l-1} \times m \\ \mathbf{W}^{[l]} &: n_l \times n_{l-1} \\ \mathbf{b}^{[l]} &: n_l \times 1 \\ \mathbf{Z}^{[l]} &: n_l \times m \\ \mathbf{A}^{[l]} &: n_l \times m \end{aligned}$$

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  $m$ .

#### Exercise:

We will illustrate the extension to the processing of a full batch using or “toy” mlp:

- Use again `4.1.compact-mlp.ipynb`.
- Cell [5]: extend this cell by defining an input consisting of a batch of size 2 (or even larger if you like) i.e.:

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

look for:

```
### START YOUR CODE ###
...
### END YOUR CODE ###
```

- Replace the processing using this matrix as input to the propagation step.
- Again, observe in detail the sizes of the input matrix, the weights and biases and the corresponding activations.

## 4.5 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 [16]. We recall that the goal is to minimise the cost function  $J(\Theta)$  (quadratic for regression problems, cross-entropy for classification)

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

Equation 7

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

$$\Theta = (\mathbf{W}^{[1]}, \mathbf{b}^{[1]}, \dots, \mathbf{W}^{[L]}, \mathbf{b}^{[L]})$$

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:

$$\begin{aligned}\mathbf{W}^{[l]} &\leftarrow \mathbf{W}^{[l]} - \alpha \cdot \frac{\partial J(\Theta)}{\partial \mathbf{W}^{[l]}} \\ \mathbf{b}^{[l]} &\leftarrow \mathbf{b}^{[l]} - \alpha \cdot \frac{\partial J(\Theta)}{\partial \mathbf{b}^{[l]}}\end{aligned}$$

I.e., we require the derivatives of the cost function  $J(\Theta)$  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.

### 4.5.1 An illustrative Example

We start with a simple example, which allows to verify all calculations in obvious manner. We define the function following function,

$$a = (w \cdot x + b)^2$$

and represent it in form of a computational graph:

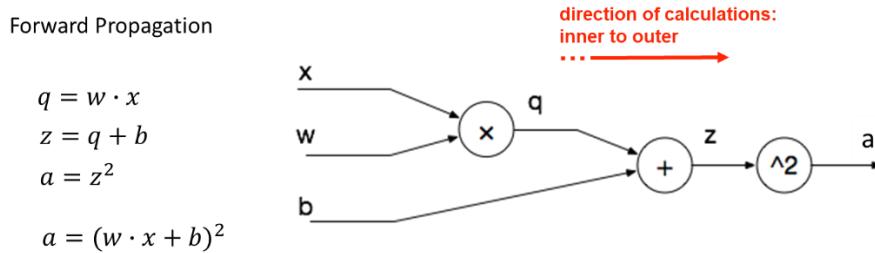
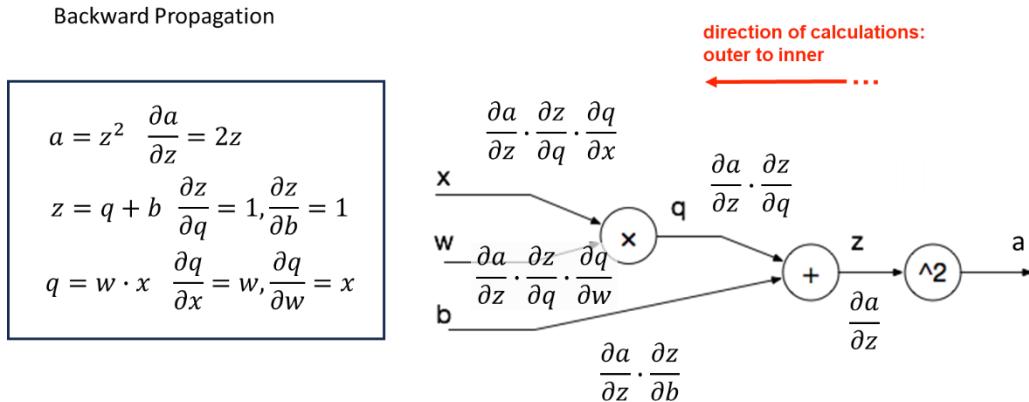


Figure 112: Forward path for the function  $a = (w \cdot x + b)^2$ .

We want to determine the derivatives of the result  $a$  with respect to  $b$ ,  $w$  and  $x$ . From the basic rules of differential calculus, it follows that the successive use of the chain rule will allow to do this in a simple manner by backpropagating through the graph from right to left. This is shown in the following Figure 113.



**Figure 113:** By applying successively the chain rule the derivatives of result  $a$  with respect to  $b$ ,  $w$  and  $x$  can be determined.

Thus, we start with the node at the extreme right, which represents the square function  $a = z^2$ . The derivative of  $a$  with respect to  $z$  is simply given by:

$$\frac{\partial a}{\partial z} = 2z$$

Now we want to determine the derivative of  $a$  with respect to  $b$  (lower branch of the backpropagation path after the '+' node). By application of the chain rule this is simply given by:

$$\frac{\partial a}{\partial b} = \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial b} = 2z \cdot 1$$

Now we continue with the upper branch of the backpropagation path after the '+' node to determine the derivative of  $a$  with respect to  $q$ , which we require as an intermediate result for the derivatives with respect to  $x$  and  $w$ . Again, by application of the chain rule this is simply given by:

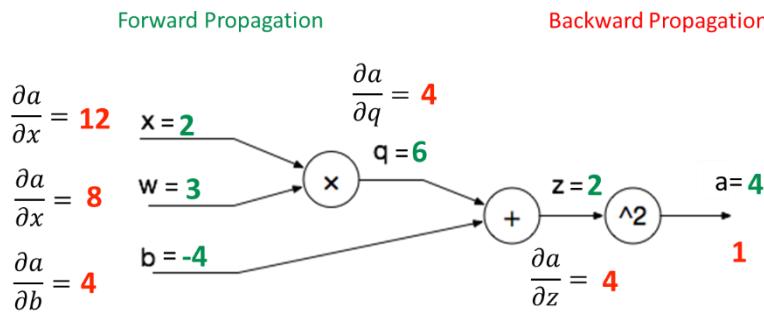
$$\frac{\partial a}{\partial q} = \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial q} = 2z \cdot 1$$

Now we can finish with the derivatives of  $a$  with respect to  $x$  and  $w$ , which are given respectively by:

$$\frac{\partial a}{\partial x} = \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial q} \cdot \frac{\partial q}{\partial x} = 2z \cdot 1 \cdot w$$

$$\frac{\partial a}{\partial w} = \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial q} \cdot \frac{\partial q}{\partial w} = 2z \cdot 1 \cdot x$$

The following gives some numerical values both for the forward path (green) and the backpropagation. The value of 1 in the backward propagation all to left is used as starting point for the successive multiplication of the derivative during the backpropagation.

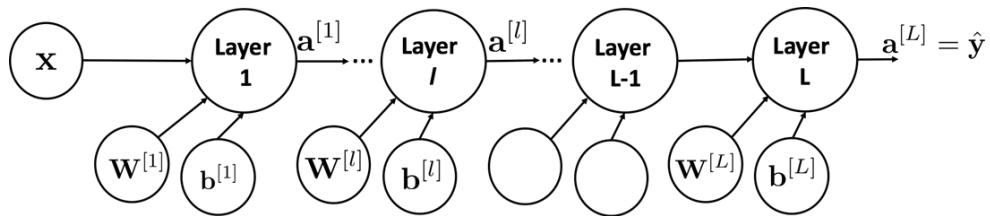


**Figure 114:** Forward and backward path with numerical values (details c.f. text).

Exercise:

Verify the values of the forward and backward path and make sure that you understand the application of the chain rule during the backpropagation.

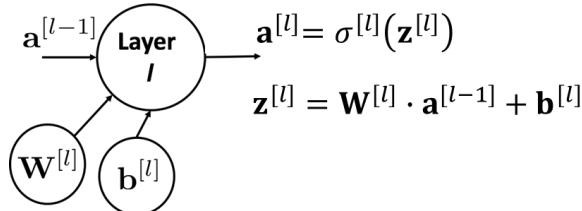
From the considerations above we see that we only require the derivates for each individual node function and can then apply – in the backward path – the chain rule differential calculus. We recall (Figure 115) that our MLP is made up of identical layers. Therefore, it will be sufficient to develop the derivatives for a single MLP layer. We will do this in three steps. In the next chapter 4.5.2 we will derive the formulas for the derivatives using notation with indices. Because the implementation will be done using matrices, we will reformulate the derivative calculation in matrix notation in chapter 4.5.3, which will turn out to be very compact. Both for chapter 4.5.2 and chapter 4.5.3 we will only use one single input vector i.e. the sum over the cost function will contain only one single term (Equation 7). Because the cost is a sum – thus a linear function – it will be straight forward, to extend the considerations to an input consisting of a full batch of a given size. This will be done in chapter 4.5.4.



**Figure 115:** Because our MLP is build up from layers with identical shape, we only require the backpropage rules for this layer.

### 4.5.2 Backprop through a Single MLP Layer

We recall the formula for the forward pass for a single MLP layer (Figure 116).



**Figure 116:** Forward pass for a single MLP layer.

If that we have the quantity

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

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

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

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  $\sigma^{[l]}$  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  $b_j^{[l]}$ :

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

**Equation 8**

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

#### 4.5.2.1 Determination of $\frac{\partial L}{\partial z_j^{[l]}}$

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

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

$$\boxed{\frac{\partial L}{\partial z_j^{[l]}} = \frac{\partial L}{\partial a_j^{[l]}} \cdot \frac{d\sigma^{[l]}(z_j^{[l]})}{dz}}$$

#### 4.5.2.2 Determination of $\frac{\partial L}{\partial w_{ki}^{[l]}}$

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

At the equal sign  $=^1$  we again introduced the Kronecker delta  $\delta_{kj}$  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:

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

#### 4.5.2.3 Determination of $\frac{\partial L}{\partial b_k^{[l]}}$

This follows closely the derivation of the previous chapter 4.5.2.2:

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

Thus, we obtain:

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

#### 4.5.2.4 Determination of $\frac{\partial L}{\partial a_k^{[l-1]}}$

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

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.

#### 4.5.3 General Formulation of Backpropagation in Matrix Notation

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

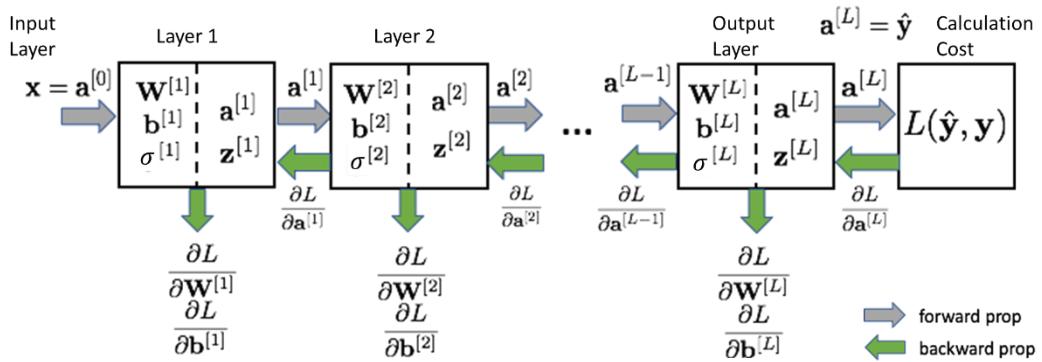
$$\begin{aligned}\frac{\partial L}{\partial \mathbf{z}^{[l]}} &= \frac{\partial L}{\partial \mathbf{a}^{[l]}} * \frac{d\sigma^{[l]}(\mathbf{z}^{[l]})}{dz} \\ \frac{\partial L}{\partial \mathbf{W}^{[l]}} &= \frac{\partial L}{\partial \mathbf{z}^{[l]}} \cdot (\mathbf{a}^{[l-1]})^T \\ \frac{\partial L}{\partial \mathbf{b}^{[l]}} &= \frac{\partial L}{\partial \mathbf{z}^{[l]}} \\ \frac{\partial L}{\partial \mathbf{a}^{[l-1]}} &= (\mathbf{W}^{[l]})^T \cdot \frac{\partial L}{\partial \mathbf{z}^{[l]}}\end{aligned}$$

**Equation 9**

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  $\frac{d\sigma^{[l]}(\mathbf{z}^{[l]})}{dz}$  are vectors of size  $n^l \times 1$ . 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  $n^l \times 1$  and the transpose of  $\mathbf{a}^{[l-1]}$  is a vector of size  $1 \times n^{l-1}$ . Thus, the product will have the dimension  $n^l \times n^{l-1}$ .
- For the last equation the matrix product of the transpose of  $\mathbf{W}^{[l]}$  (dimension  $n^{l-1} \times n^l$ ) and  $\frac{\partial L}{\partial \mathbf{z}^{[l]}}$  (dimension  $n^l \times 1$ ) is calculated with a result of dimension  $n^{l-1} \times 1$ .

Because the formulas apply to all layers – except the final softmax layer, which we will treat below – we can now simply apply successively these scheme during the backward propagation, as illustrated in the following Figure 117.



**Figure 117:** Overview of the backpropagation scheme for an MLP.

These formulas above are correct for a dense (or fully connected) layer of an MLP. For the final Softmax layer the first equation reads different. To determine the correct version, we will first consider one term of the full sum of the cost function (with the extension to the full sum done in a further step):

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

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

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:

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

Or in vector notation:

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

Here both the prediction  $\hat{\mathbf{y}}$  and the true label  $\mathbf{y}$  are vectors of size  $n^L \times 1$ , 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.

#### Exercise:

We will now implement the backpropagation for our “toy” mlp:

- Use again 4.1.compact-mlp.ipynb.
  - Cell [6]: Implement the backpropagation step formulas (Equation 9). Note that the terms `dg2_dz` and `dg1_dz` represent the derivatives of the activation functions  $\frac{d\sigma^{[l]}(\mathbf{z}^{[l]})}{d\mathbf{z}}$  – respectively for the second and first layer – taken at the corresponding value of the logit ( $\mathbf{z}^{[2]}$  and  $\mathbf{z}^{[1]}$ ). Furthermore, we do not use the Softmax layer with CE cost (but a simplified MSE-loss) its derivative – i.e. that start value – is simply given by:

$$\text{dL_da2} = \mathbf{a2}$$

Look for:

```
### START YOUR CODE ###
...
### END YOUR CODE ###
```

and implement your solution.

- Verification of the implementation:  
Once you have completed your implementation move to cell [8]. This is a numerical gradient check. You may have a quick look forward to chapter 4.5.4.1. If your implementation is correct the difference between the numerical values und your implementation, which are calculated at the end of this cell, should be of the order to  $\sim 10^{-8}$ .

#### 4.5.4 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:

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

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:

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

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<sup>83</sup>:

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

Now we can extend the formulation of the backpropagation (Equation 9) 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 9 now performed for each of the  $i = 1..m$  columns independently:

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

Here again the operator  $*$  is the *element-wise* multiplication for the full matrices of dimensions  $n^l \times m$  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:

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

Please note that each of the  $m$  summands, corresponding to the  $i$ -th training sample, is a full  $n^l \times n^{l-1}$  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:

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

Note that the transpose of the matrix  $\mathbf{A}^{[l-1]}$  to the right is formed by turning the original column 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.

<sup>83</sup> This is a simple consequence of the fact that the cost is a (linear) sum of independent results corresponding to the  $m$  different samples in our batch.

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 over 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  $m \times 1$ :

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

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

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

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

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

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

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

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

**Equation 10**

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:

$$\frac{\partial L}{\partial \mathbf{Z}^{[l]}}, \frac{\partial L}{\partial \mathbf{A}^{[l]}}, \frac{dg^{[l]}(\mathbf{Z}^{[l]})}{dz}: n_l \times m$$

$$\mathbf{A}^{[l-1]}, \frac{\partial L}{\partial \mathbf{A}^{[l-1]}}, \frac{\partial L}{\partial \mathbf{Z}^{[l]}}: n_{l-1} \times m$$

$$\frac{\partial L}{\partial \mathbf{W}^{[l]}}, \frac{\partial L}{\partial \mathbf{b}^{[l]}}, \frac{\partial L}{\partial \mathbf{A}^{[l-1]}}: n_l \times n_{l-1}$$

$$\frac{\partial L}{\partial \mathbf{b}^{[l]}}, \frac{\partial L}{\partial \mathbf{A}^{[l-1]}}, \frac{\partial L}{\partial \mathbf{Z}^{[l]}}: n_l \times 1$$

$$\hat{\mathbf{Y}}, \mathbf{Y}: n_L \times m$$

The formulas above (Equation 10) are for dense and the Softmax layer only. Corresponding formulas for other layer types (batch normalisation, convolutional, dropout layers) will be discussed in the next chapters<sup>84</sup>.

### Exercise:

We will now implement the backpropagation for a full batch for our “toy” mlp:

- Use again 4.1.compact-mlp.ipynb.
  - Cell [7]: Implement the backpropagation step formulas (Equation 10). Note that the terms `dg2_dz` and `dg1_dz` represent the derivatives of the activation functions  $\frac{d\sigma^{[l]}(z^{[l]})}{dz}$  – respectively for the second and first layer – taken at the corresponding value of the logit ( $z^{[2]}$  and  $z^{[1]}$ ). Furthermore, we do not use the Softmax layer with CE cost (but a simplified MSE-loss) its derivative – i.e. that start value – is simply given by:

$$dL_da2 = a2/m$$

Here  $m$  is the batch size.

Implement your solution in the sections marked with:

```
### START YOUR CODE ###
...
### END YOUR CODE ###
```

- Verification of the implementation:  
 Once you have completed your implementation move to cell [8]. This is a numerical gradient check. You may have a quick look forward to chapter 4.5.4.1. *Important:* Change the input vector `x_n` to the implementation of a full batch input `x` from cell [7]. If your implementation is correct the difference between the numerical values und your implementation, which are calculated at the end of this cell, should be of the order to  $\sim 10^{-8}$ .

#### 4.5.4.1 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:

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

We can approximate the true value of the derivative using a small but finite  $\varepsilon_0$ :

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

## 4.6 Results for MLP with one Hidden Layer

We present results of a Multi-Layer Perceptron with one hidden layer with the indicated number of neurons. Figure 118 to Figure 124 present the corresponding training curves as function of different hyperparameters learning rates, batch sizes and number of hidden neurons. Each curve was trained for 100 epochs. The curves are obtained with the solution of Practical Work 04.

---

<sup>84</sup> One important point to notice when implementing the above formulas in Python is that in ML frameworks the first matrix- (or in general tensor-) dimension represents the sample index (within the batch) and *not* the feature index, as we used for the matrix formulation here. For elementwise “`*`” multiplications this is not of importance, for matrix multiplications ‘`@`’ the two factors can just be switched i.e.,  $dL \cdot A^T$  can be replaced by  $A^T \cdot dL$  !

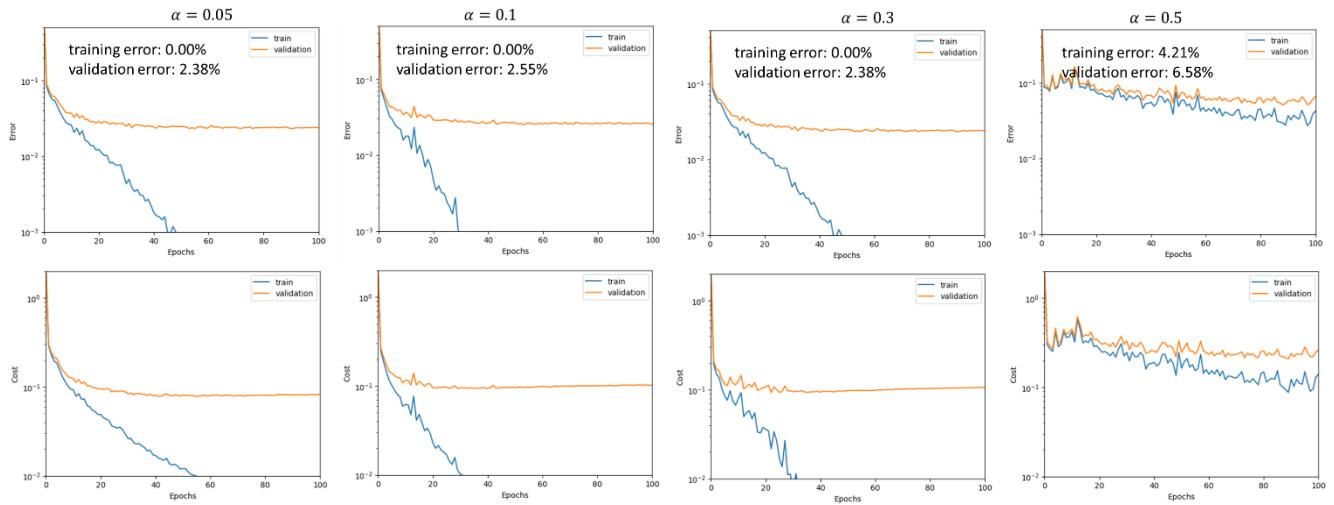


Figure 118: Results for MLP model with single hidden layer (Figure 92) with varying learning rate.

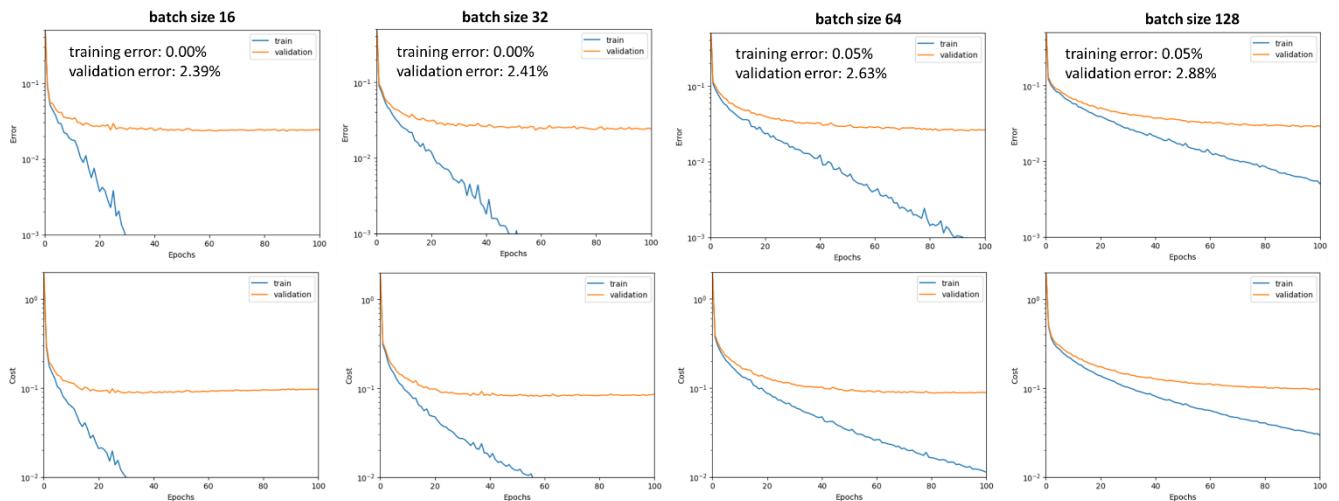


Figure 119: Results for MLP model with single hidden layer (Figure 92) with varying batch size.

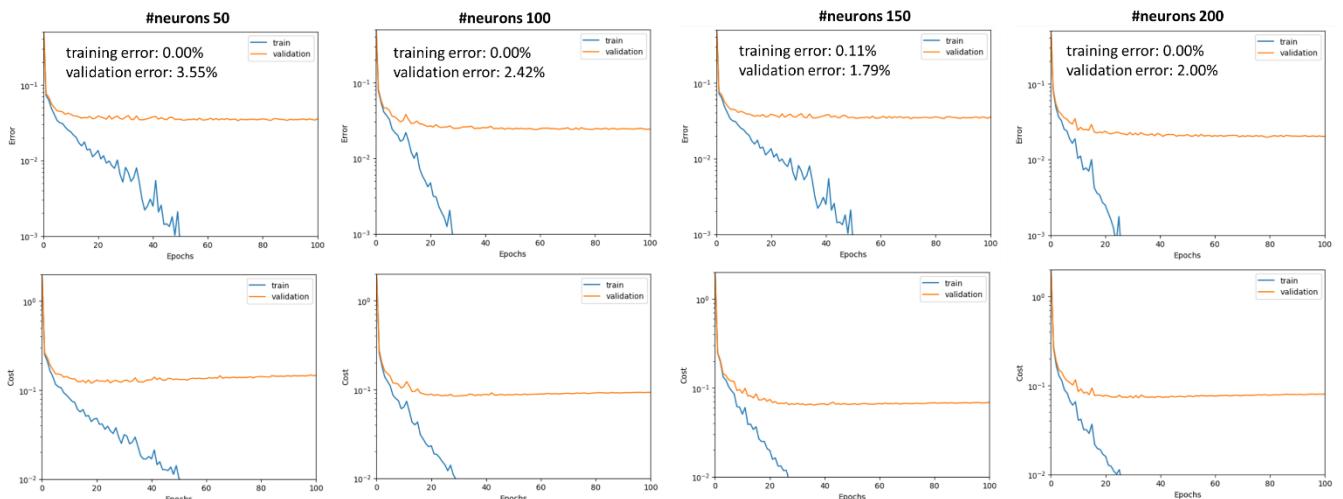
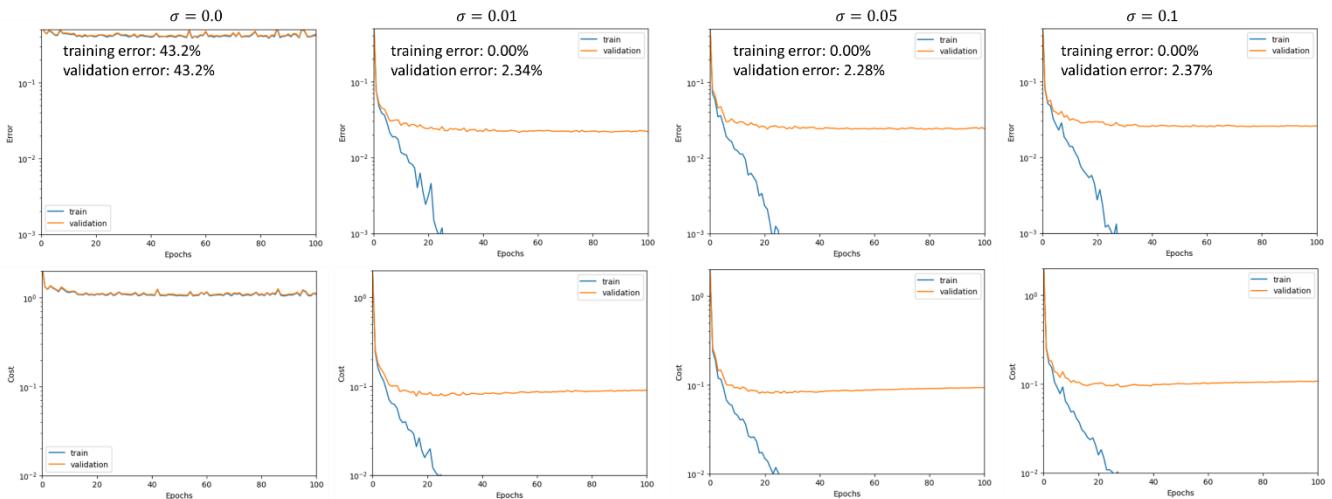


Figure 120: Results for MLP model with single hidden layer (Figure 92) with varying number of hidden neurons.

The following points can be noted:

- For all curves a considerably overfitting occurs after a few epochs: the training error and cost tend to practically zero, while the validation curves start to saturate.
- Learning rates around a value of  $\alpha \approx 0.1$  are a good choice. The highest learning rate  $\alpha = 0.5$  shows bad convergence (Figure 118).
- A learning rate of 16 leads to the fastest convergence (Figure 118).
- Increasing the number of hidden neurons increases the representational capacity of the network and allows for smaller error rates (Figure 120).

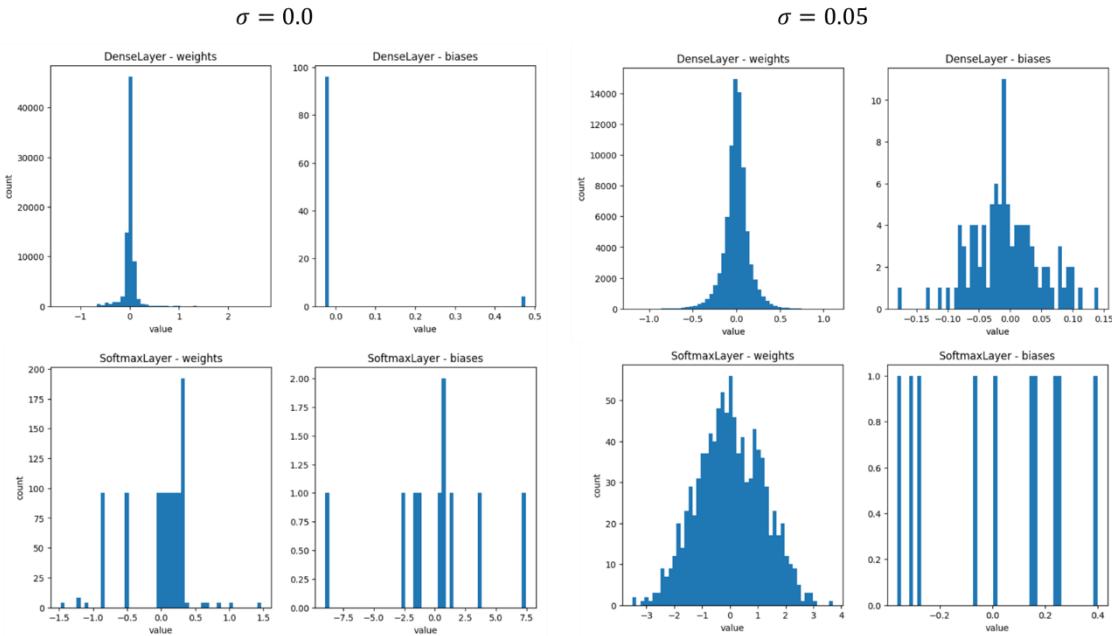
We want to investigate one further hyperparameter which is the standard deviation  $\sigma$  used to initialize the weight and bias values. When looking at the curves in Figure 121 it turns out that with a value of  $\sigma = 0.0$  the algorithm will not converge. We recall that for the generalized Perceptron the CE cost function is convex such that the choice of the initial parameters is of – almost – no importance for the convergence. For a MLP this is different. If all weights and biases are initialized to zero, the propagation pass will lead to a value of zero for all logits  $z^{[l]}$  (c.f. Equation 5) and 0.5 for the activations (zero of the sigmoid function). During the backward path it will be hard for the weights to leave a value of zero, even for larger number of epochs. For all other values  $\sigma \in \{0.01, 0.05, 0.1\}$  almost identical learning curves are obtained. The reason for the problem with  $\sigma = 0.0$  becomes evident, when the distribution of the weight and bias values are analysed at the end of the training phase. In Figure 122 the histogram of the weights and biases are shown (top for the hidden layer, bottom for the softmax layer). If weights and biases are initialized to zero (left) they will remain mainly close to or equal to zero, even after 100 training epochs. A random initialization (right) with  $\sigma=0.05$  leads to a Gaussian-like distribution of the weights around zero. While a value of  $\sigma = 0.0$  is obviously a bad choice, and could have been ruled out right from the beginning, this analysis already shows that care has to be taken when initializing the parameters of a MLP, a problem that will be discussed in depth in chapter 5.1.



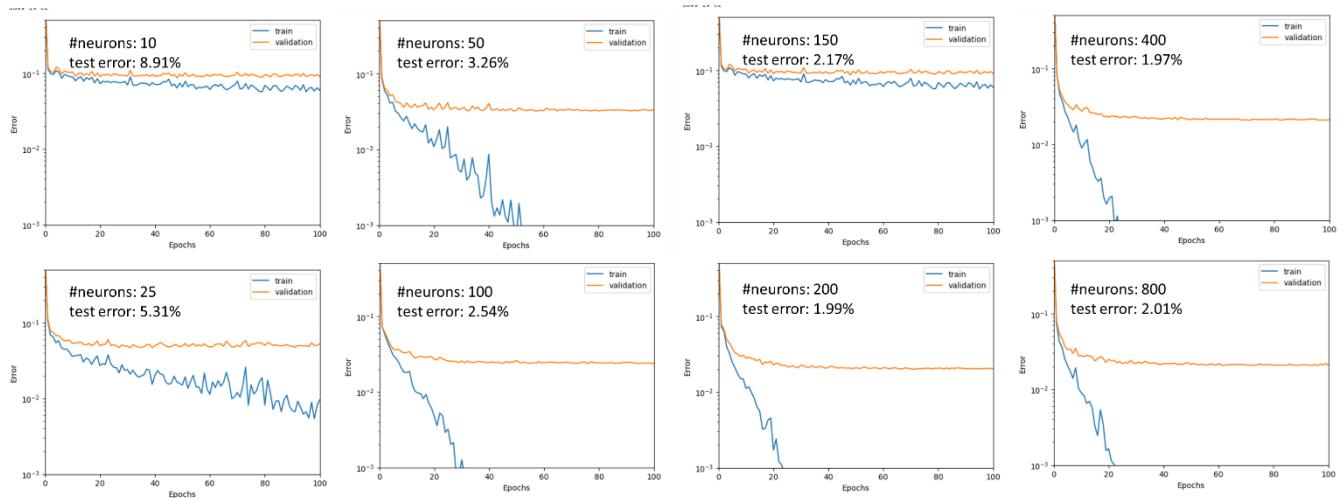
**Figure 121:** Results for MLP model with single hidden layer (Figure 92) with varying standard deviation ( $\sigma$ ) used to initialize the weight values.

Before we close this chapter, we want to further exploit the representational capacity of a MLP with one hidden layer. According to the universal approximation theorem (chapter 4.3.2) with a sufficient number of hidden neurons any function can be approximated. Therefore, the number of hidden neurons was increased even more (than in Figure 120) and the behaviour of the test error rate was observed. The batch size was chosen to 16 (presumably best value from Figure 119) and the learning rate to 0.1 (Figure 118). The results – an extension to Figure 120 – are shown in Figure 123. The behaviour of the test error rate as a function of the number of epochs is plotted in Figure 124. Apparently, the increase in number of hidden neurons reduces the test error rate from ~9.0% down to ~

2.0%. However, the test error rate reaches a plateau at around 200 neurons with a final value of ~2.0% and no further decrease can be observed. In addition, the strong overfitting apparently does not allow the algorithm to improve on the test set.



**Figure 122:** If weights and biases are initialized to zero (left) they will remain mainly close to or equal to zero, even after a considerable number of training step. A random initialization (right) with  $\sigma = 0.05$  shows a Gaussian-like distribution of the weights around zero.



**Figure 123:** Results for MLP model with single hidden layer (Figure 92) with varying number of hidden neurons (extension of Figure 123).

Thus, from our first study on MLPs with 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? 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.

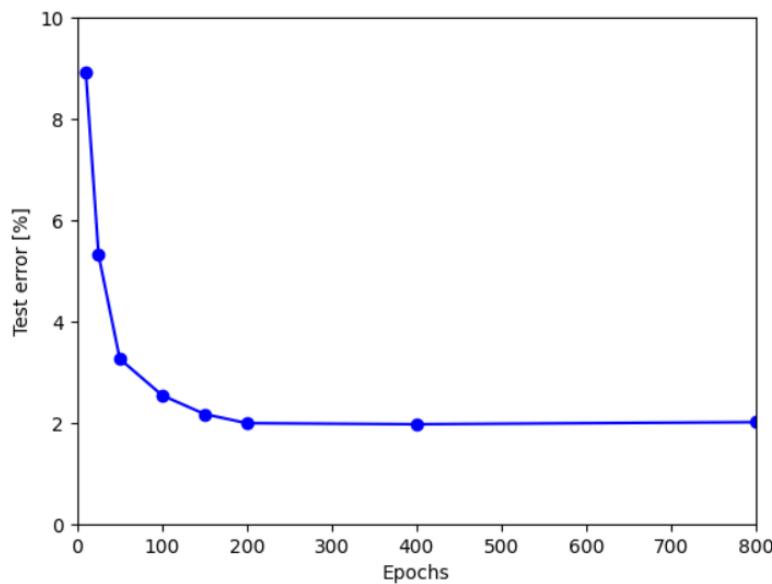


Figure 124: Test error rate for MLP model with single hidden layer (Figure 92) over number of hidden neurons.

## 5 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 [16]. 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  $(\mathbf{W}^{[l]})^T$  is performed (Equation 10). 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 50).
- 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.

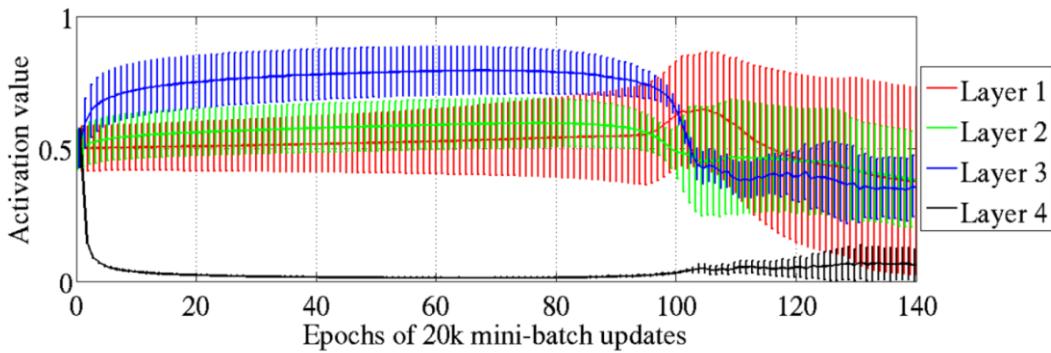
### 5.1 Vanishing and Exploding Gradients

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 [17] and S. Ioffe, C. Szegedy [18], which we will discuss in the following<sup>85</sup>. We will first list the various interrelated problems that can occur and then discuss the strategies that have been developed to address them.

#### 5.1.1 Saturation of Activations

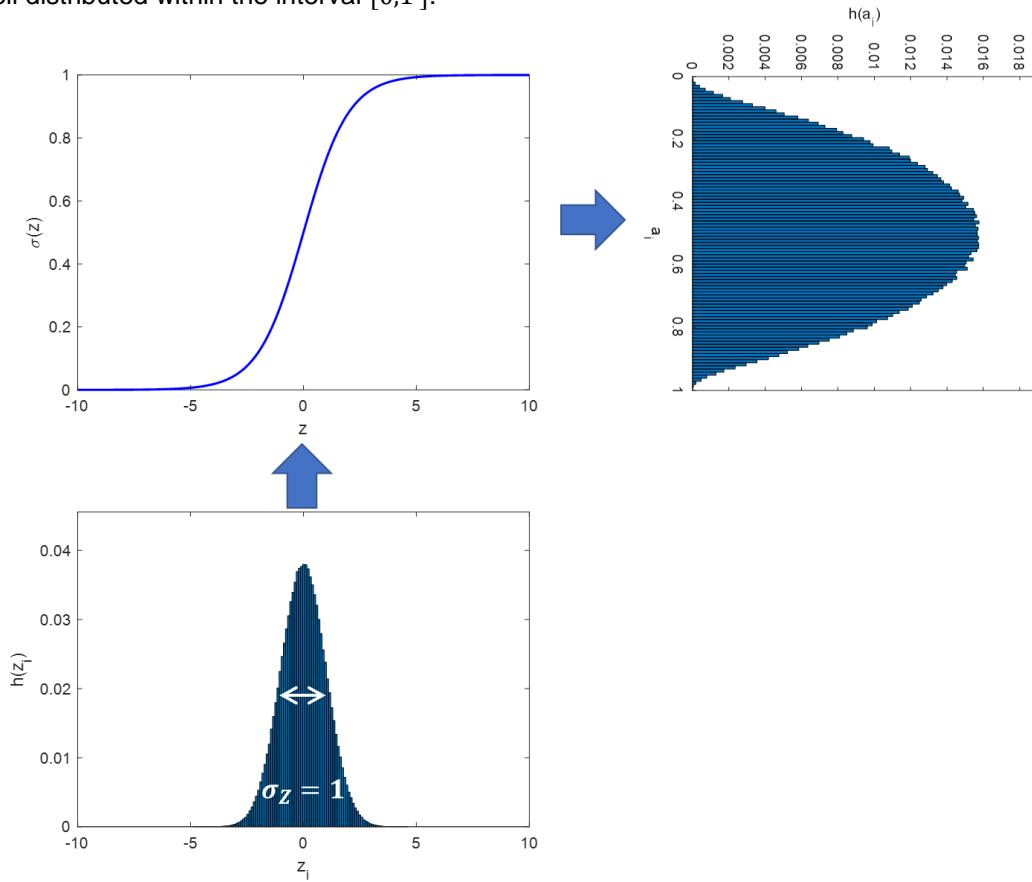
X. Glorot and J. Bengio [17] studied the behaviour of the activation values  $\mathbf{A}^{[l]}$  i.e., the output of the activation functions, for a NN with 4 layers. Figure 125 shows one of their findings using the sigmoid activation function. The mean value of the activations  $\mathbf{A}^{[l]}$  and 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., 4<sup>th</sup> 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.

<sup>85</sup> See also the discussions in <http://neuralnetworksanddeeplearning.com/chap5.html>.



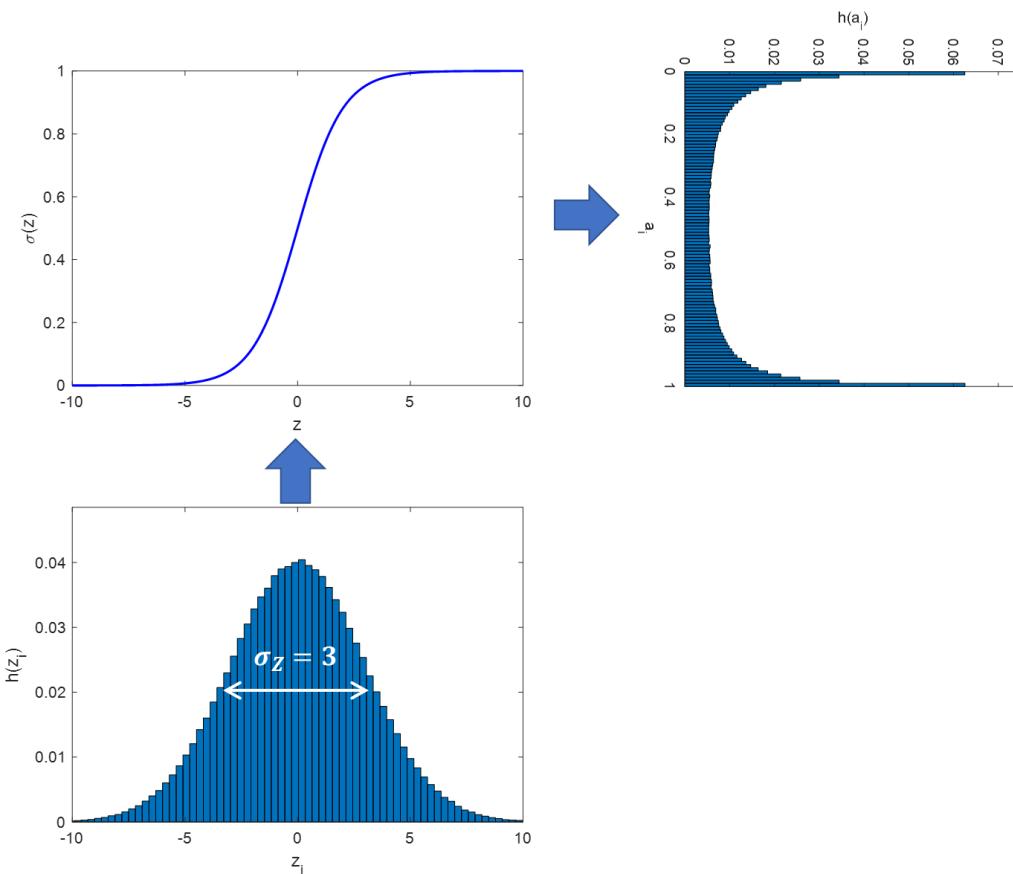
**Figure 125:** The Problem of saturating activation caused by the sigmoid activation function [17].

Such a saturation of the activation will directly cause vanishing gradients, as illustrated in Figure 126 and Figure 127, using the sigmoid activation function. In Figure 126 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 126:** 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 127, 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.



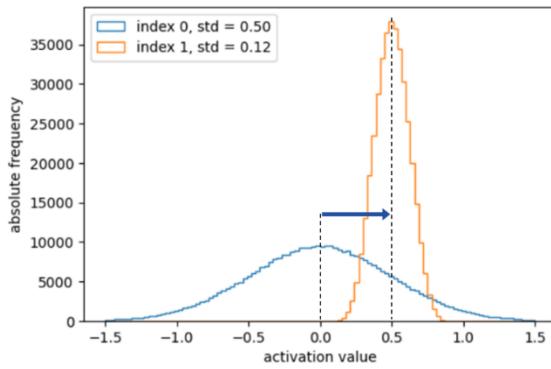
**Figure 127:** If the logits (bottom) – the input to the sigmoid activation function (top left) – have a high standard deviation ( $\sigma_z = 3$ ) the saturation regions of the sigmoid function are reached with high probability (top right).

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 [17] found that the Hyperbolic Tangent (Table 6), 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 5.1.6).

#### Exercise:

We want to illustrate the saturation behaviour of activation functions (Figure 127) using a simple model:

- Study 5.0.saturation\_activation\_stud.ipynb. It is fairly simple:
  - Cell [2]: helper functions -> you can skip this for the moment.
  - Cell [3]: Creation of a set of random samples (logits  $z$ ) – with Gaussian distribution of given standard deviation `stand_dev`. These will be used as input to the activation function `sig`. The resulting activations are denoted by  $a$ . In addition, the derivative of the activation function at the respective positions  $z$  are calculated (`dsig(z)`). The function `plot_histogram()` allows to plot the histogram of the given list of values  $[z, a]$  i.e., the logits and the activations.
- Choose a moderate standard deviation `stand_dev` (c.f. Figure 126) and reproduce the behaviour of the activation histogram. Then increase the standard deviation `stand_dev` (c.f. Figure 127) and observe the behaviour of the activations. In addition, observe the histogram of the derivatives and verify that for high standard deviation – i.e. considerable saturation of the activations – a large fraction of the derivatives is equal to zero.
- In addition to the saturation effect illustrated in Figure 127 the sigmoid function has a further disadvantage, which is the systematic bias shift of +0.5 introduced to the logits:



A simple alternative to the sigmoid activation function is the hyperbolic tangent

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

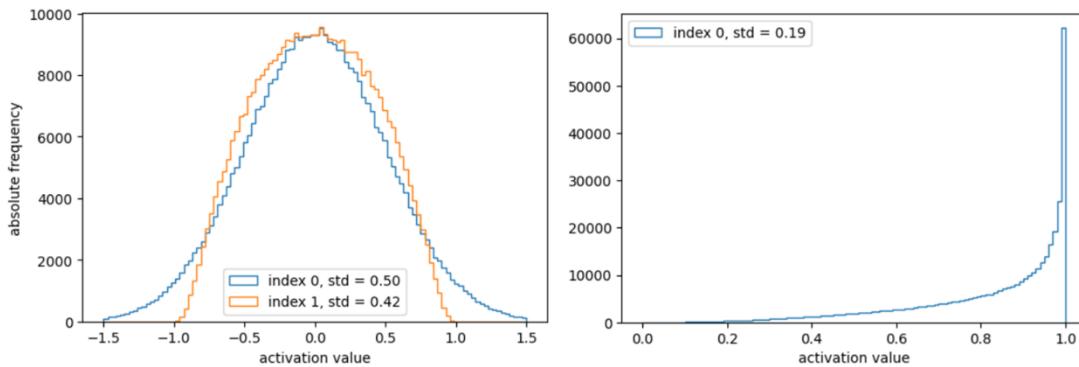
with its derivative:

$$\frac{d \tanh(z)}{dz} = \frac{4}{(e^z + e^{-z})^2}$$

You may use the illustration at the very bottom of the iPython notebook to illustrate these two functions in comparison with sigmoid function and its derivative.

Now implement in cell [2] the missing derivative of the hyperbolic tangent (`def dtanh(z):`) and then replace the sigmoid activation function – and derivative – in cell [3] by the hyperbolic tangent.

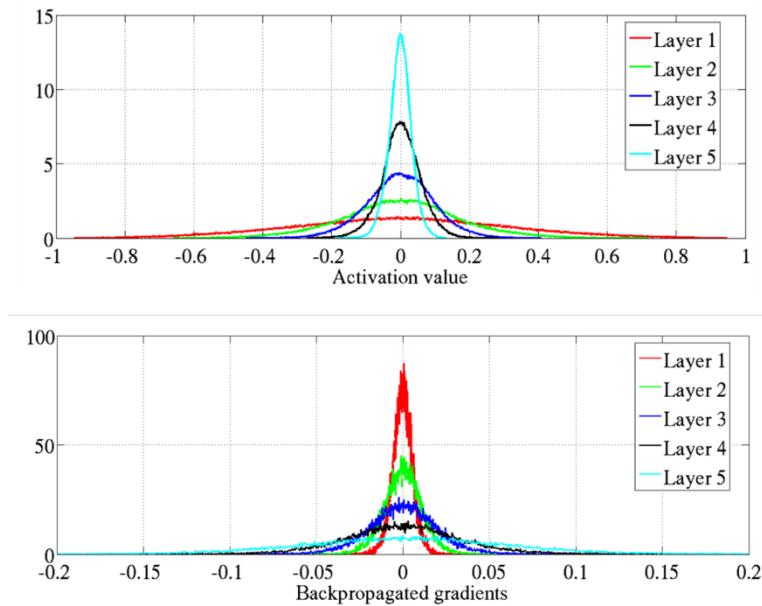
Choose a moderate standard deviation `stand_dev` and perform the forward and backward pass. Observe the histograms of the activations and derivatives and analyse the change with respect to the results of the sigmoid function. The results for `stand_dev = 0.5` should look like the following graphs:



Discuss the possible consequences for the training.

### 5.1.2 Changing Variance of Activations and Gradients

A problem related to the discussions of the previous section 5.1.1 is the change of the variance of both the activations  $A^{[l]}$  in the forward pass and of the gradients  $\frac{\partial L}{\partial A^{[l]}}$  during the backpropagation for different layer indices  $l$ . In fact, X. Glorot and J. Bengio [17] found that the variance of the activations  $A^{[l]}$  in the forward pass decrease with increasing layer index  $l$  (Figure 128, top). Similar behaviour was found for the variance of the gradients  $\frac{\partial L}{\partial A^{[l]}}$  which were found to decrease with *decreasing* layer index  $l$  during the backpropagation (Figure 128, 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 5.1.4).



**Figure 128:** The variance of both activations (top) and gradients (bottom) change over the layers [17].

#### Exercise:

We want to understand the reason for the behaviour of the activation functions and gradients using a simple model. This will prepare the discussion in chapter 5.1.4 about the correct weight initialisation:

- Study 5.1.activation\_variance\_stud.ipynb:
  - Cell [2]: helper functions -> you can skip this.
  - Cell [3]: Definition of the – simplified<sup>86</sup> – class `DenseLayer` used by the `MLP` class.
  - Cell [4]: Definition of class `NeuralNetwork`, a simplified `MLP` with only a series of dense layers i.e., without `Softmax` layer. The constructor receives the list of hidden neurons, the standard deviation for the initialisation of the weights (biases are all zero!) and the activation function. Due to the discussion in the exercise at the end of chapter 5.1.1 we will use the hyperbolic tangent (as in [17]).
  - Cell [5]: A “deep” (4 hidden layers) `MLP` is instantiated and a set of random samples with given standard deviation are prepared. Note the choice of ‘`tanh`’ as activation function.
  - Cell [6]:
    - One single forward and backward path with a graphical representation of the activations and gradients for each layer.
- Execute the cells [5] and [6] and verify the correspondence of the graphical output of cell [6] with Figure 128. Be sure to understand the ordering of the layers in the two graphs and the fact that the standard deviations of *both* the activations *and* gradients *decrease* in the forward and backward path respectively. Note that the ratio of the standard deviations of two successive layers is shown in the legend and is close to be three. We will understand this behaviour in chapter 5.1.4.

### 5.1.3 Multiplicative Structure of Backpropagation

Finally, the multiplicative structure of the backpropagation (Equation 10) when applied successively to a DL architecture may be the cause of unstable gradients<sup>85</sup>. 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 10 that the derivate of the cost function with respect to the weights  $w^{[l]}$  and biases  $b^{[l]}$

<sup>86</sup> Bias equal to zero, backpropagation only partially implemented.

of layer  $l$  are given by the following expression:

$$\begin{aligned}\frac{\partial L}{\partial w^{[l]}} &= a^{[l-1]} \cdot g'(z^{[l]}) \\ \frac{\partial L}{\partial b^{[l]}} &= g'(z^{[l]})\end{aligned}$$

$\left( \prod_{k=l+1}^L w^{[k]} g'(z^{[k]}) \right) \cdot \frac{\partial L}{\partial a^{[L]}}$

$\left( \prod_{k=l+1}^L w^{[k]} g'(z^{[k]}) \right) \cdot \frac{\partial L}{\partial a^{[L]}}$

**Product Term**

**Figure 129:** 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 result.

As already mentioned, the problems illustrated in the previous sections 5.1.1 till 5.1.3 limited the success of training deeper NN architectures till the 2010. We will now discuss the solutions proposed in [17] and [18], which were 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 128) 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.

#### Exercise:

We want to illustrate the decreasing nature of the gradients in the backward path when using the sigmoid activation function:

- Use again the iPython notebook 5.0.saturation\_activation\_stud.ipynb. Now continue to study the next cell:
  - Cell [4]: Like Cell [3] but with number of `layer` successive application of the activation function. In addition, the product of the derivatives of the activation function for each “layer” is calculated. Note that currently the sigmoid activation function is chosen.
- Execute cell [4] and observe the behaviour of the activations and derivatives when increasing the number of layers from one, which corresponds to the result of cell [3], successively (`layers = 1, 2, 3, ...`). You will observe that indeed the gradients move all to zero with increasing layer number. In addition, the activation will all accumulate around a value of around 0.659. This is the so-called fixed-point of the sigmoid function and the observed behaviour is a consequence of fixed-point theorem of Banach<sup>87</sup>.

<sup>87</sup> [https://en.wikipedia.org/wiki/Banach\\_fixed-point\\_theorem](https://en.wikipedia.org/wiki/Banach_fixed-point_theorem)

- Now change the activation to the hyperbolic tangent in cell [4] and observe the improvement.

### 5.1.4 Xavier and He Initialization of Network Parameters

We recall that X. Glorot and J. Bengio [17] studied the change of the variance for the activations  $A^{[l]}$  in the forward pass and for the gradients  $\frac{\partial L}{\partial A^{[l]}}$  during the backpropagation as a function of the layer index  $l$ . The behaviour shown in Figure 128 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 128 was obtained with the following initialization scheme (considered as state of the art in 2010<sup>88</sup>):

$$\mathbf{W}^{[l]} \sim U\left[-\frac{1}{\sqrt{n_{l-1}}}, \frac{1}{\sqrt{n_{l-1}}}\right]$$

$$\mathbf{b}^{[l]} = 0$$

Equation 11

Here  $U[-a, a]$  is a uniform distribution in the interval  $[-a, a]$  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 8 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  $l - 1$ .

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

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:

$$\text{Var}(a_j^{[l]}) \sim \text{Var}\left(\sum_i W_{ji}^{[l]} \cdot a_i^{[l-1]} + b_j^{[l]}\right) =^{(1)} \text{Var}\left(\sum_i W_{ji}^{[l]} \cdot a_i^{[l-1]}\right) =^{(2)} \sum_i \text{Var}(W_{ji}^{[l]}) \cdot \text{Var}(a_i^{[l-1]})$$

In step  $=^{(1)}$  we used that the biases are initialized to zero and for  $=^{(2)}$  the following identity, which holds for independent random variables  $x$  and  $y$ <sup>89</sup>:

$$\text{Var}(x \cdot y) = \text{Var}(x) \cdot \text{Var}(y) + [\text{E}(x)]^2 \cdot \text{Var}(y) + \text{Var}(x) \cdot [\text{E}(y)]^2$$

As the weights  $W_{ji}^{[l]}$  and the activations  $a_i^{[l-1]}$  are independent and in addition have zero mean (i.e. expectation) the identity  $=^{(2)}$  holds<sup>90</sup>.

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  $\text{Var}(W^{[l]})$  and  $\text{Var}(a^{[l-1]})$  respectively) we can further conclude:

$$\text{Var}(a_j^{[l]}) \sim \sum_i \text{Var}(W_{ji}^{[l]}) \cdot \text{Var}(a_i^{[l-1]}) = \underbrace{n_{l-1} \cdot \text{Var}(W^{[l]}) \cdot \text{Var}(a^{[l-1]})}_{(*)}$$

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 11, this would require:

$$\mathbf{W}^{[l]} \sim U\left[-\sqrt{\frac{3}{n_{l-1}}}, \sqrt{\frac{3}{n_{l-1}}}\right]$$

<sup>88</sup>Sometimes the initialization schemes  $\mathbf{W}^{[l]} \sim U[-1,1]$  or  $\mathbf{W}^{[l]} \sim N[0,1]$  (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 [17] these were not considered.

<sup>89</sup>Here  $\text{E}(x)$  denotes the expectation of  $x$ .

<sup>90</sup>This explains, why the shift of the activations by +0.5 from the sigmoid activation function is a problem.

## Equation 12

The reason for the additional factor  $\sqrt{3}$  is because a uniform distribution  $U[-1,1]$  has the variance  $1/3$ . This explains the decrease of the activations with increasing layer index  $l$  in Figure 128<sup>91</sup>.

X. Glorot and J. Bengio [17] pushed their analysis further to understand the change in the variance of the gradients  $\frac{\partial L}{\partial \mathbf{A}^{[l]}}$  (Figure 128). When considering Equation 10 for the backpropagation we see that it is governed by a similar equation as for the forward pass:

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

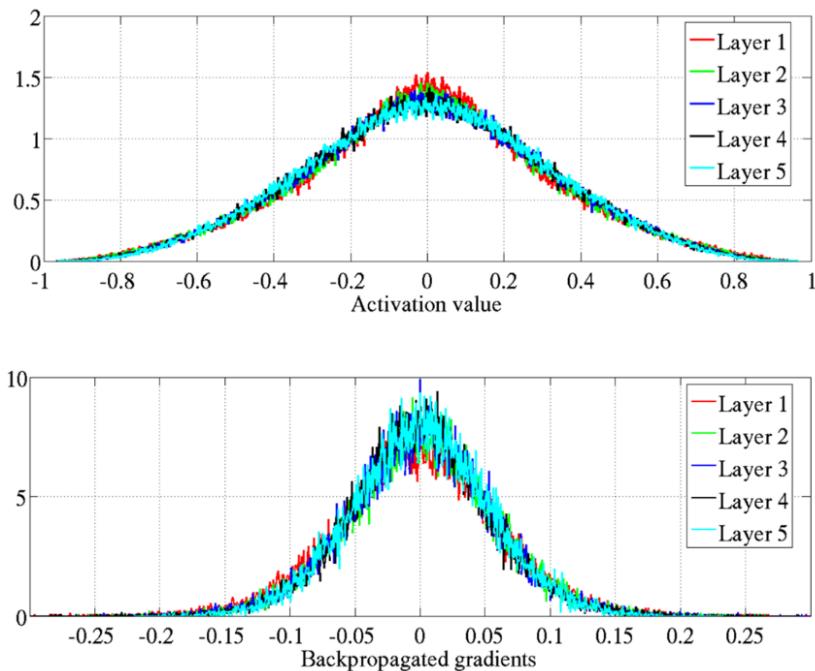
The main difference to the considerations that led to Equation 12 is that the weight matrix is applied in transposed form i.e., that rows and columns are exchanged. Thus, one can conclude that the variance of the activations stays constant from layer  $l$  to  $l - 1$  during the backpropagation if the following initialization scheme is applied:

$$\mathbf{W}^{[l]} \sim U \left[ -\sqrt{\frac{3}{n_l}}, \sqrt{\frac{3}{n_l}} \right]$$

This condition and Equation 12 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 [17] proposed a good compromise between the forward pass and the backpropagation which turns out to work well in practice:

$$\mathbf{W}^{[l]} \sim U \left[ -\sqrt{\frac{6}{n_{l-1} + n_l}}, \sqrt{\frac{6}{n_{l-1} + n_l}} \right]$$

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 [17] demonstrate that the variances of both activations and gradients stay constant over all layers (Figure 130).



**Figure 130:** 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 128) [17].

<sup>91</sup> In fact, analysis of the standard deviations of the activations in Figure 128 for the different layers reveals that they behave indeed like  $\text{Var}(a^{[l]}) \approx k \cdot \text{Var}(a^{[l-1]})$  with  $k = 1/1.8 \approx 1/\sqrt{3}$  as expected from Equation 12.

Extension to parameter initialization using a normal distribution and other activation functions lead to the following general set of parameter initialization schemes:

	<b>Activation function</b>	<b>Uniform distribution [-r, r]</b>	<b>Normal distribution</b>
Xavier	Sigmoid	$4 \cdot \sqrt{\frac{6}{n_{l-1} + n_l}}$	$4 \cdot \sqrt{\frac{2}{n_{l-1} + n_l}}$
	Hyperbolic tangent	$\sqrt{\frac{6}{n_{l-1} + n_l}}$	$\sqrt{\frac{2}{n_{l-1} + n_l}}$
He	ReLU (and its variants)	$\sqrt{2} \cdot \sqrt{\frac{6}{n_{l-1} + n_l}}$	$\sqrt{2} \cdot \sqrt{\frac{2}{n_{l-1} + n_l}}$

**Figure 131:** 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. We will apply these schemes in the different DL frameworks in chapter 6.

#### Exercise:

Using the iPython notebook `5.1.activation_variance_stud.ipynb` we can now reproduce these findings by scheme X. Glorot and J. Bengio [17]:

- In the instantiation of the `NeuralNetwork` class in cell [5] now correct the initialisation of the weights with the correct schema from Figure 131:

```
random_std = 1*np.sqrt(1/num_feat)
```

Then execute the propagation step and observe the improvement. You may have to adjust the plot range in the histogram functions. You may further try the rectified linear unit (`relu`) activation function.

### 5.1.5 Batch Normalization

While the weight initialization schemes in Figure 131 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 125 may still result.

S. Ioffe and C. Szegedy [18] 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<sup>92</sup>, would affect the representational capacity of the model (c.f. chapter 4.2). Therefore, an additional scaling and shift of the normalized logits is applied using two additional network parameters (which will be called  $\gamma^{[l]}$  and  $\beta^{[l]}$  respectively) that must be optimized during training.

In detail their approach works as follows<sup>93</sup>:

We consider the layer to index  $l$  with the matrix of logits  $Z_{\{r\}}^{[l]}$  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  $Z_{\{r\}}^{[l]}$  consists of column vectors  $z_{\{r\}}^{[l](i)}$  ( $i = 1..m$ ) where  $m$  denotes the size of the mini-batch. Each vector  $z_{\{r\}}^{[l](i)}$  has  $n_l$  entries corresponding to the number of output neurons of layer  $l$ .

<sup>92</sup> We consider the use of the sigmoid function for which  $\sigma(z) \approx z$  for  $z \approx 0$ .

<sup>93</sup> For the notation see also chapter 4.5.4 on the formulation of the backpropagation for a full batch.

$$\mathbf{Z}_{\{r\}}^{[l]} = \begin{pmatrix} \vdots & \vdots & \vdots \\ \mathbf{z}_{\{r\}}^{[l](1)} & \mathbf{z}_{\{r\}}^{[l](2)} & \dots & \mathbf{z}_{\{r\}}^{[l](m)} \\ \vdots & \vdots & & \vdots \end{pmatrix}$$

To apply batch normalization, we calculate the average  $\mu_{\{r\}}^{[l]}$  and standard deviation  $\sigma_{\{r\}}^{[l]}$  over the  $m$  column vectors  $\mathbf{z}_{\{r\}}^{[l](i)}$  of the mini-batch according to:

$$\mu_{\{r\}}^{[l]} = \frac{1}{m} \sum_{i=1}^m \mathbf{z}_{\{r\}}^{[l](i)}$$

$$\sigma_{\{r\}}^{[l]} = \sqrt{\frac{1}{m} \sum_{i=1}^m (\mathbf{z}_{\{r\}}^{[l](i)} - \mu_{\{r\}}^{[l]})^2}$$

Now, the actual normalization of the logit matrix is as follows:

$$\hat{\mathbf{Z}}_{\{r\}}^{[l]} = \frac{\mathbf{Z}_{\{r\}}^{[l]} - \mu_{\{r\}}^{[l]}}{\sigma_{\{r\}}^{[l]} + \epsilon}$$

The following points should be noted:

- The average  $\mu_{\{r\}}^{[l]}$  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  $\sigma_{\{r\}}^{[l]}$  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  $\epsilon$  in the denominator is for numerical stability in case that the standard deviation  $\sigma_{\{r\}}^{[l]}$  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:

$$\tilde{\mathbf{Z}}_{\{r\}}^{[l]} = \gamma^{[l]} \cdot \hat{\mathbf{Z}}_{\{r\}}^{[l]} + \beta^{[l]}$$

The values  $\tilde{\mathbf{Z}}_{\{r\}}^{[l]}$  are now used as new input to the activation function.

The following points should be noted:

- Both  $\gamma^{[l]}$  and  $\beta^{[l]}$  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  $\gamma^{[l]} = \sigma_{\{r\}}^{[l]}$  and  $\beta^{[l]} = \mu_{\{r\}}^{[l]}$  we recover the original logit values because we simply undo the scaling operation. Thus, introducing the scaling with parameters  $\gamma^{[l]}$  and  $\beta^{[l]}$  will recover the original representational capacity of the NN.

The scaling parameters  $\gamma^{[l]}$  and  $\beta^{[l]}$  now seem to undo the normalization obtained with  $\sigma_{\{r\}}^{[l]}$  and  $\mu_{\{r\}}^{[l]}$ . However, the main difference is that  $\gamma^{[l]}$  and  $\beta^{[l]}$  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 5.1.4).
- It allows much higher learning rates which speeds up the learning process<sup>94</sup>.
- 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 5.3).

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:

$$\mathbf{Z}^{[l]} = \mathbf{W}^{[l]} \cdot \mathbf{A}^{[l-1]} + \mathbf{b}^{[l]}$$

- The mean values  $\mu_{\{r\}}^{[l]}$  and standard deviations  $\sigma_{\{r\}}^{[l]}$  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  $\beta$  according to:

$$\begin{aligned}\mu^{[l]} &= (1 - \beta) \cdot \mu_{\{r\}}^{[l]} + \beta \cdot \mu^{[l]} \\ \sigma^{[l]} &= (1 - \beta) \cdot \sigma_{\{r\}}^{[l]} + \beta \cdot \sigma^{[l]}\end{aligned}$$

- Batch normalisation has one inconvenience that it changes the backprop equations. Thus, on the one hand the derivatives for the new parameters  $\gamma^{[l]}$  and  $\beta^{[l]}$  must be added. On the other hand, the derivatives with respect to the normalization parameters  $\mu_{\{r\}}^{[l]}$  and  $\sigma_{\{r\}}^{[l]}$  must be considered<sup>95</sup>.
- Because batch normalization is applied to the logits the layer definition in a DL frameworks should in principle follow the scheme<sup>96</sup>:
  - Dense layer with linear activation, no bias required
  - Batch normalization
  - Activation function

### 5.1.6 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 [17] was that the problem of vanishing gradients is in part related to the use of saturating activation functions as illustrated in Figure 126 and Figure 127. Since then, it was found that other activation functions behave much better in deep NNs. In Table 6 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

<sup>94</sup> In the paper [18] 14 times fewer learning steps at even better accuracy are reported.

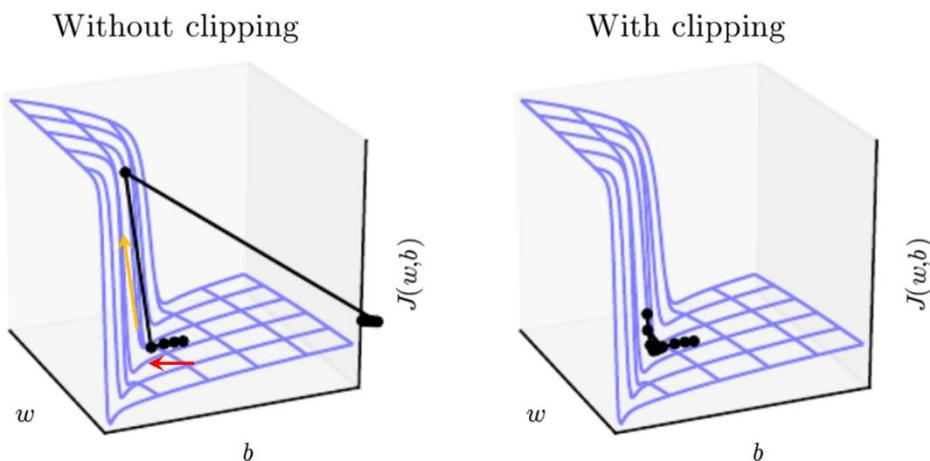
<sup>95</sup> For the derivation of the corresponding formulas see appendix chapter 8.1.

<sup>96</sup> Nevertheless, many researchers argue that it is just as good or even better to place batch normalization after the activation layer.

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”.

- 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  $\alpha$  for  $z < 0$ ) which must be determined by hyper-parameter tuning or by setting a good default value<sup>97</sup>.
- 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 [20] that “*ELUs lead not only to faster learning, but also to significantly better generalization performance than ReLUs on networks with more than 5 layers*”.

### 5.1.7 Gradient Clipping



**Figure 132:** In case of very steep regions of the cost function, gradient clipping should be applied (details see text).

<sup>98</sup>Deep networks often have extremely steep regions resembling cliffs because of the composite structure of the operations where weights get multiplied (c.f. chapter 5.1.3). At the cliffs, the gradient can be become very large (Figure 132, [1]). Therefore, it is beneficial to limit the values of the gradient by application of clipping. This is illustrated in Figure 132. 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.

## 5.2 Advanced Optimizers

In chapters 3.4 and 3.7 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 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:

<sup>97</sup> See [19] for a comparative evaluation of the different ReLUs.

<sup>98</sup> This topic is especially relevant for RNNs.

- 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.

### 5.2.1 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"* [15].

Momentum optimization computes an exponentially decaying sum (parameter  $\beta$ ) of past gradients and moves in the direction of this sum:

$$\begin{aligned}\mathbf{m} &\leftarrow \beta \cdot \mathbf{m} + \alpha \cdot \nabla_{\theta} J \\ \theta &\leftarrow \theta - \mathbf{m}\end{aligned}$$

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  $t$ :

$$\begin{aligned}t = 1: \quad & \alpha \cdot \nabla_{\theta} J \\ t = 2: \quad & \beta \cdot \alpha \cdot \nabla_{\theta} J + \alpha \cdot \nabla_{\theta} J \\ t = 3: \quad & \beta^2 \cdot \alpha \cdot \nabla_{\theta} J + \beta \cdot \alpha \cdot \nabla_{\theta} J + \alpha \cdot \nabla_{\theta} J \\ \dots \\ t = n: \quad & \beta^{n-1} \cdot \alpha \cdot \nabla_{\theta} J + \beta^{n-2} \cdot \alpha \cdot \nabla_{\theta} J + \dots + \beta \cdot \alpha \cdot \nabla_{\theta} J + \alpha \cdot \nabla_{\theta} J = \frac{1 - \beta^n}{1 - \beta} \cdot \nabla_{\theta} J\end{aligned}$$

Thus, if  $n$  grows large the value of  $\beta^n$  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.

#### 5.2.1.1 Nesterov Accelerated Gradient Optimization

An additional improvement of the momentum optimization as proposed by Y. Nesterov is:

$$\begin{aligned}\mathbf{m} &\leftarrow \beta \cdot \mathbf{m} + \alpha \cdot \nabla_{\theta} J(\theta - \beta \cdot \mathbf{m}) \\ \theta &\leftarrow \theta - \mathbf{m}\end{aligned}$$

The idea is to evaluate the gradient not at the current parameter position  $\theta$  but slightly ahead in the direction of the momentum at position  $\theta - \beta \cdot \mathbf{m}$ . The idea behind is illustrated in Figure 133 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.

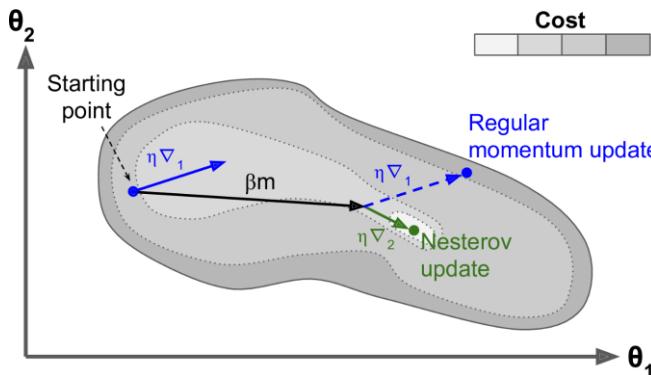


Figure 133: Illustration of Nesterov update scheme (details see text)<sup>99</sup>.

## 5.2.2 AdaGrad Optimization

One of the problems frequently slowing down the convergence is the so-called elongated bowl problem. This occurs when the parameters  $\theta_i$  have very different scales as illustrated in Figure 134 for two parameters  $\theta_1$  and  $\theta_2$ . Due to the different scales the cost function is very flat in direction of  $\theta_1$  and very steep in direction of  $\theta_2$ . Vanilla GD will first move down along the steepest slope i.e., in direction of  $\theta_2$  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.

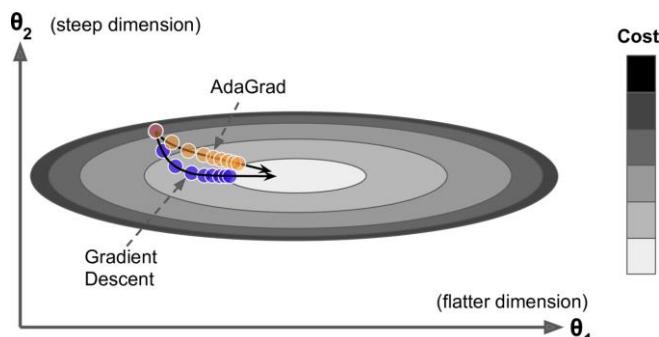


Figure 134: The elongated bowl problem where vanilla GD may take a long time to converge while AdaGrad would speed up the convergence<sup>99</sup>.

The AdaGrad algorithm achieves this by scaling down the gradient vector along the steepest dimensions. The updating scheme is:

$$\begin{aligned} \mathbf{s} &\leftarrow \mathbf{s} + \nabla_{\theta} J \cdot \nabla_{\theta} J \\ \boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} - \frac{\alpha}{\sqrt{\mathbf{s} + \epsilon}} \cdot \nabla_{\theta} J \end{aligned}$$

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  $\theta_i$  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 update – given in the second line – a component wise scaling of the gradients  $\nabla_{\theta} J$  by  $\sqrt{\mathbf{s} + \epsilon}$  is applied<sup>100</sup>. 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

<sup>99</sup> <https://medium.com/ml-learning-ai/deep-learning-optimizers-4c13d0799b4d>

<sup>100</sup> Here the term  $\epsilon \approx 10^{-8}$  is again for numerical stability in case that one of the components of  $\mathbf{s}$  is zero.

the global optimum (Figure 134). One additional benefit is that it requires much less tuning of the learning rate hyperparameter  $\alpha$  (c.f. chapter 5.2.6).

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.

### 5.2.3 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:

$$\begin{aligned}\mathbf{s} &\leftarrow \beta \cdot \mathbf{s} + (1 - \beta) \cdot \nabla_{\theta} J \cdot \nabla_{\theta} J \\ \boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} - \frac{\alpha}{\sqrt{\mathbf{s} + \epsilon}} \cdot \nabla_{\theta} J\end{aligned}$$

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).

### 5.2.4 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:

$$\begin{aligned}\mathbf{m} &\leftarrow \beta_1 \cdot \mathbf{m} + (1 - \beta_1) \cdot \nabla_{\theta} J \\ \mathbf{s} &\leftarrow \beta_2 \cdot \mathbf{s} + (1 - \beta_2) \cdot \nabla_{\theta} J \cdot \nabla_{\theta} J \\ \hat{\mathbf{m}} &= \frac{\mathbf{m}}{1 - \beta_1} \\ \hat{\mathbf{s}} &= \frac{\mathbf{s}}{1 - \beta_2} \\ \boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} - \frac{\alpha}{\sqrt{\hat{\mathbf{s}}} + \epsilon} \cdot \hat{\mathbf{m}}\end{aligned}$$

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

### 5.2.5 Comparison of the Optimizers

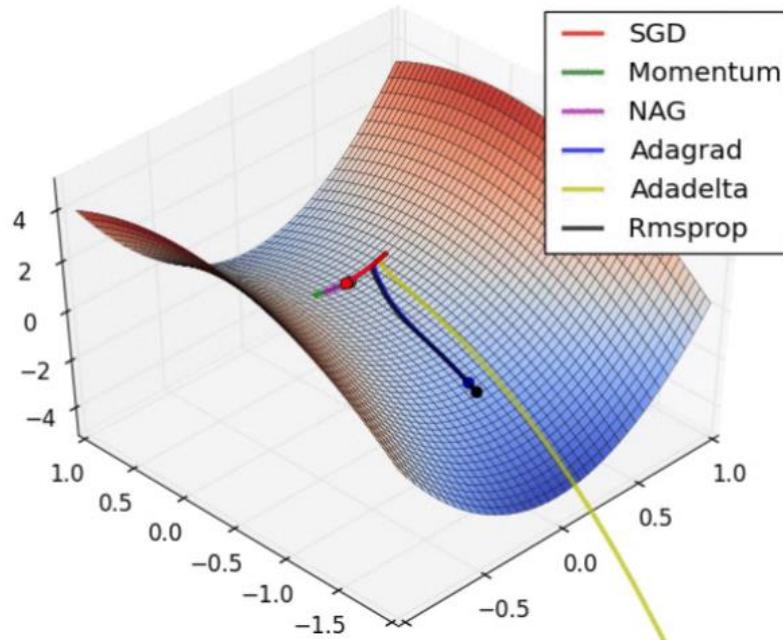
The following figures illustrate the behaviour the different optimizers under different topologies that are known to pose problems for vanilla GD. The illustrations are for a cost function dependent on two parameters. The original figures are animations and can be found here<sup>101</sup>. The correspondences of the naming convention of the figures to our definitions are:

- — Gradient Descent (GD)
- — Momentum Optimization
- — Nesterov Accelerated Gradient Optimization
- — AdaGrad Optimization
- — Adam Optimization

- RMSProp Optimization

### 5.2.5.1 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 135:** Different optimizers on a saddle point (details see text)<sup>101</sup>.

- 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.

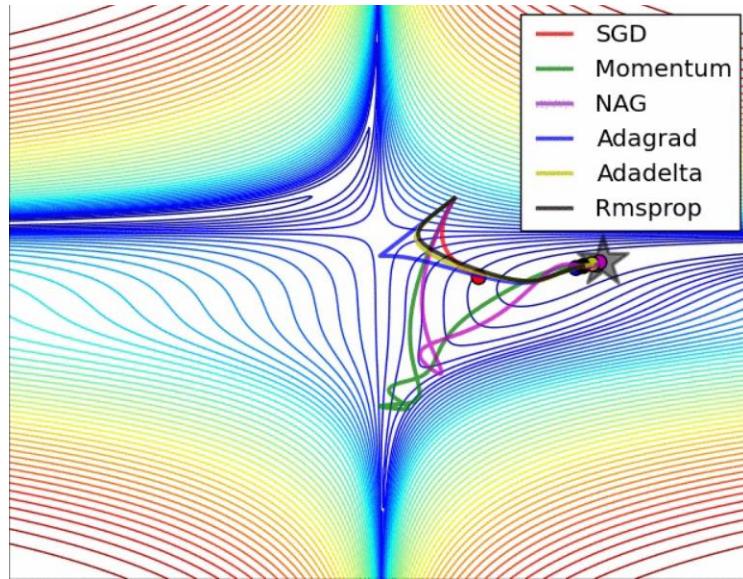
### 5.2.5.2 Motion in an elongated bowl topology

The geometry in Figure 136 resembles an elongated bowl with very steep and very flat parts of the cost function.

- GD behaves as expected (c.f. Figure 134) 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 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. Neverthe-

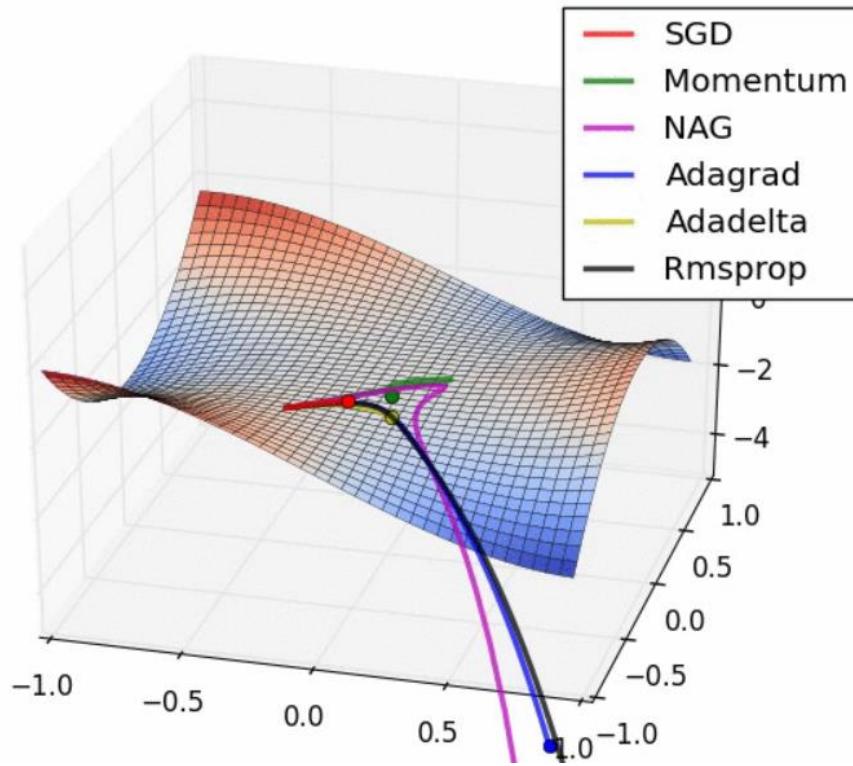
<sup>101</sup> See animations on <https://imgur.com/a/Hqolp> .

less, AdaGrad almost goes unstable – like momentum-based techniques – due to the high initial velocity.



**Figure 136:** Different optimizers in an elongated bowl like topoglogy<sup>101</sup>.

#### 5.2.5.3 Motion along the flat axis of a saddle Point



**Figure 137:** Different optimizers on a saddle point<sup>101</sup>.

The start of the optimizers is close to a saddle point with a direction along the flatter of the two axes.

- 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.

## 5.2.6 Learning Rate Scheduling

The choice of a good learning rate may be difficult. If it is chosen too high, 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.

### 5.2.6.1 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.

### 5.2.6.2 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  $\lambda$ .

### 5.2.6.3 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  $\alpha_0$  – by a factor of 10 every  $T$  steps. While this works well in practice it requires the tuning of the hyperparameters  $\alpha_0$  and  $T$ .

### 5.2.6.4 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  $\alpha_0$  and  $T$ .

## 5.3 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<sup>102</sup>  
Methods that generate more training data with additional characteristics (e.g. symmetries) which the solution should have.

### 5.3.1 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<sup>1</sup>-Regularisation
- L<sup>2</sup>-Regularisation

$$\Omega(\mathbf{W}) = \|\mathbf{W}\|_1 = \sum_{j,i,l} |W_{ji}^{[l]}|$$

$$\Omega(\mathbf{W}) = \frac{1}{2} \cdot \|\mathbf{W}\|_2^2 = \frac{1}{2} \cdot \sum_{j,i,l} (W_{ji}^{[l]})^2$$

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.

#### 5.3.1.1 Gradient Descent with L<sup>2</sup>-Regularisation

We determine the gradient of the cost function:

$$\nabla J(\boldsymbol{\theta}) = \nabla \left( J_0(\boldsymbol{\theta}) + \lambda \cdot \frac{1}{2} \cdot \|\mathbf{W}\|_2^2 \right) = \nabla J_0(\boldsymbol{\theta}) + \lambda \cdot \mathbf{W}$$

Application of the GD update rule gives:

$$\mathbf{W} \leftarrow \mathbf{W} - \alpha \cdot \nabla J(\boldsymbol{\theta}) = \mathbf{W} - \alpha \cdot (\nabla J_0(\boldsymbol{\theta}) + \lambda \cdot \mathbf{W})$$

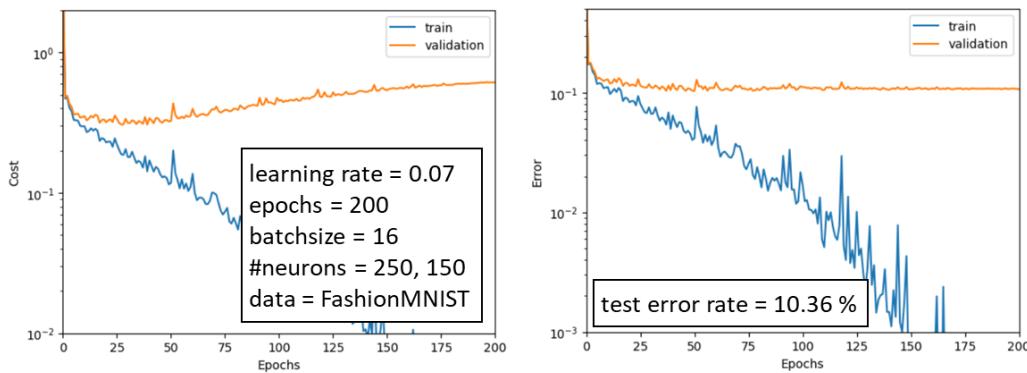
or:

$$\mathbf{W} \leftarrow \underbrace{(1 - \alpha \cdot \lambda)}_{(1)} \cdot \mathbf{W} - \underbrace{\alpha \cdot \nabla J_0(\boldsymbol{\theta})}_{(2)}$$

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 usual gradient update.

The following figures illustrate the application of the L<sup>2</sup> on an MLP (result of PW-05).

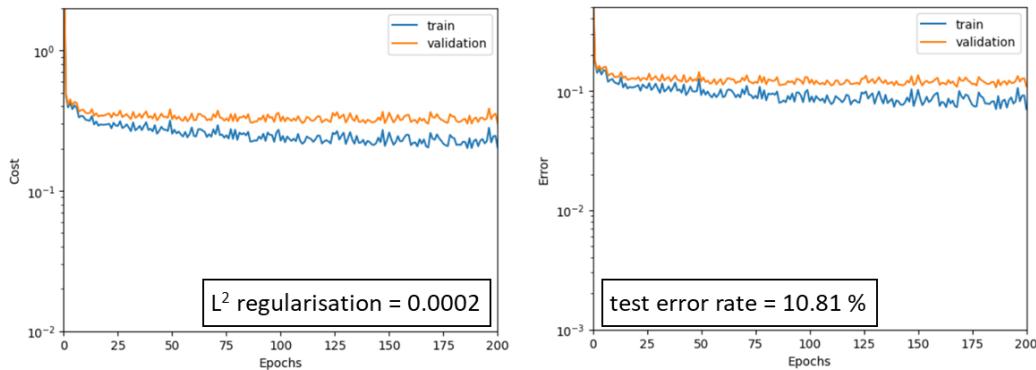
<sup>102</sup> This topic will be treated in the second part of the course in the context of CNNs.



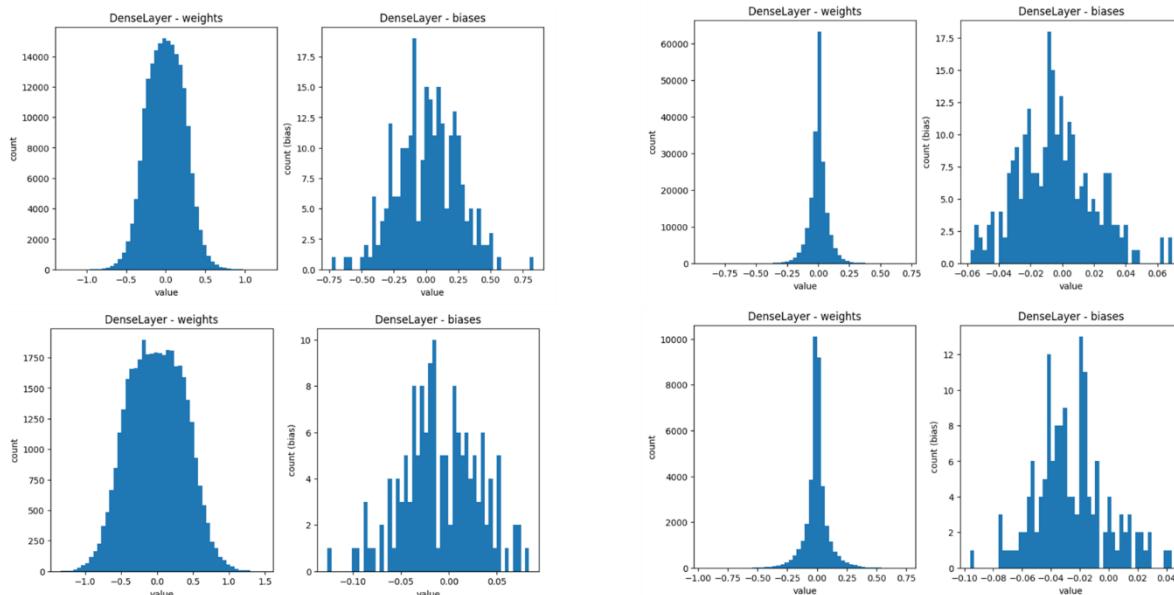
**Figure 138:** Cost and error for a MLP with two hidden layers of 250 and 150 neurons respectively. The cost shows clearly that we are overfitting after epoch ~30.

Exercise:

In groups of 2, discuss why the overfitting is clearly visible in the cost but not in the error learning curve.



**Figure 139:** Cost and error for a MLP with two hidden layers of 250 and 150 neurons respectively. Applying  $L^2$  regularisation removes the overfitting.



**Figure 140:** Weights and biases of the two dense layers for the MLP of the two previous figures. (Left) without and (right) with  $L^2$ -regularisation. With regularisation the weights and biases are considerably rescaled to smaller values with regularisation (note the different axis ranges).

Figure 138 shows the learning curves for cost and error of an MLP with two hidden layers of 250 and 150 neurons respectively<sup>103</sup>. The cost clearly shows overfitting after epoch ~30. Applying L<sup>2</sup>-regularisation removes the overfitting (Figure 139). The rescaling of the weights and biases for the two hidden layers is illustrated in Figure 140, left without right with regularisation. Note the different x-range of the plots.

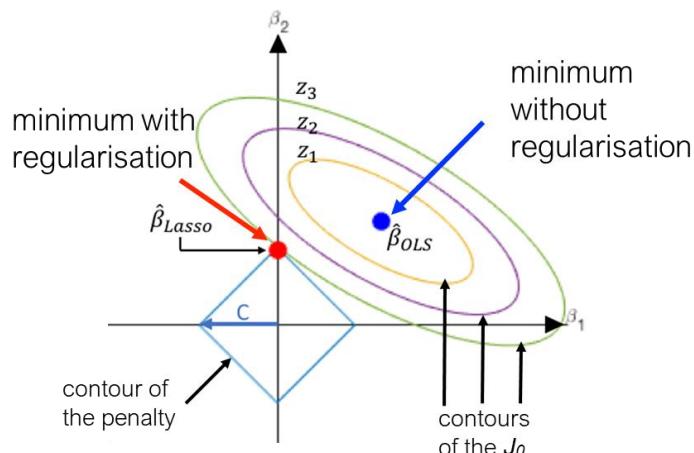
### 5.3.1.2 Gradient Descent with L<sub>1</sub>-Regularisation

We determine the gradient of the cost function:

$$\nabla J(\boldsymbol{\theta}) = \nabla J_0(\boldsymbol{\theta}) + \lambda \cdot \|\mathbf{W}\|_1 = \nabla J_0(\boldsymbol{\theta}) + \lambda \cdot \text{sign}(\mathbf{W})$$

Here the sign-function is applied elementwise to the weights  $\mathbf{W}$ .

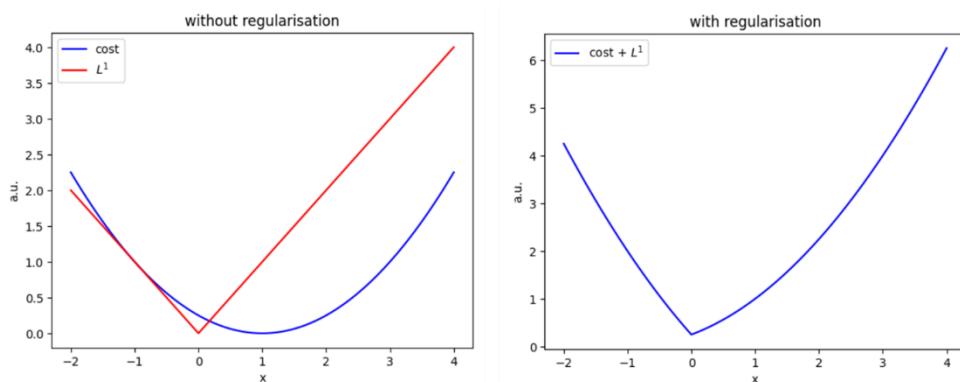
The following Figure 141 illustrates the application of the L<sub>1</sub>-regularization. For linear regression problems the addition of the L<sub>1</sub> penalty term is also referred to as LASSO regression. The ellipses show the contours of  $J_0(\boldsymbol{\theta})$  i.e., the cost function without regularization. The regularization term  $\sum_{j,i,l} |W_{ji}^{[l]}|$  will read like  $|\beta_1| + |\beta_2|$  for the illustrated case of two parameters  $\beta_i$ . The contours of constant penalty take the form of a diamond. Application of the L<sub>1</sub>-regularization will now push the original minimum (blue) towards the  $\beta_2$  axis (red) thus eliminating the parameter  $\beta_1$ . In fact, L<sub>1</sub>-regularization automatically performs a feature selection and leads to a sparser solution (as compared to no or L<sub>2</sub>-regularization).



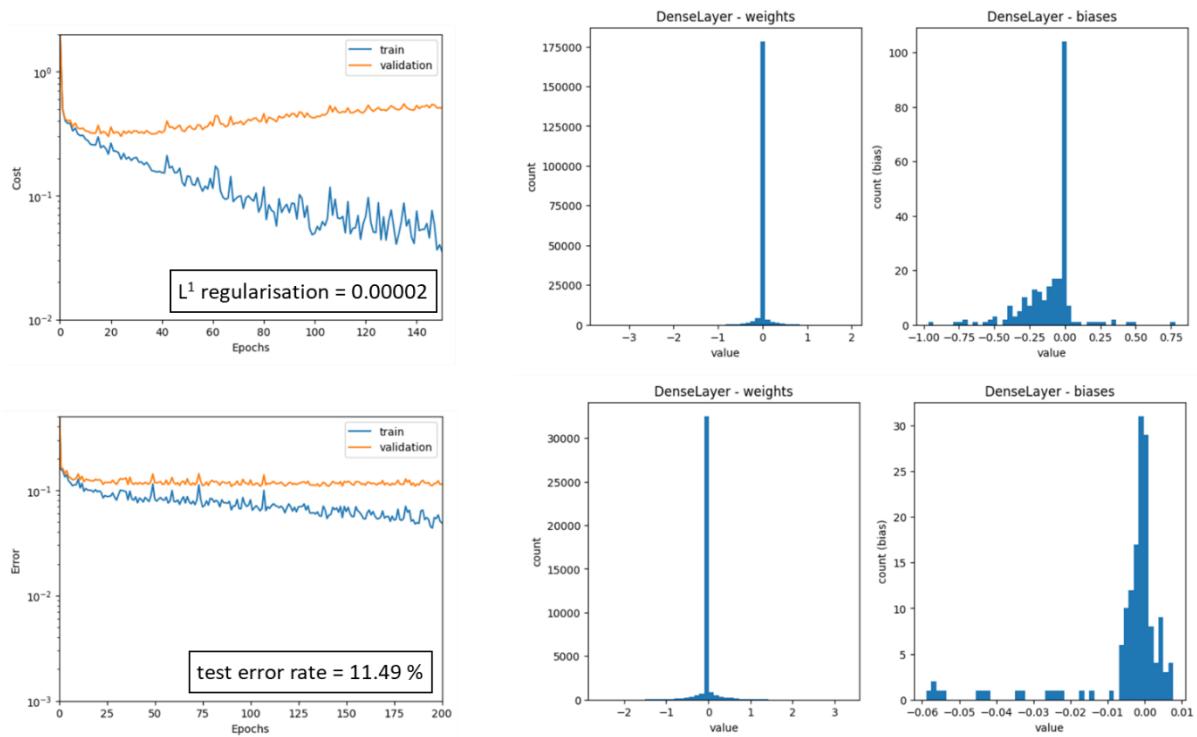
**Figure 141:** Illustration of the L<sub>1</sub>-regularization (details see text).

#### Exercise:

The iPython notebook 5.2.L1\_regularisation\_stud.ipynb illustrates the effect of an L<sub>1</sub>-regularisation term on a simple quadratic cost function – as in Figure 141 – however for the 1D case, which is easier to understand. Study the notebook and make sure to understand the following graph:



<sup>103</sup> Deeper MLP tend to overfit more.



**Figure 142:** Results corresponding to the settings of Figure 138 but with  $L^1$ -regularisation. From the parameter histograms (the two hidden layers) it is obvious that a considerable number of weight and bias values were “pushed” to zero.

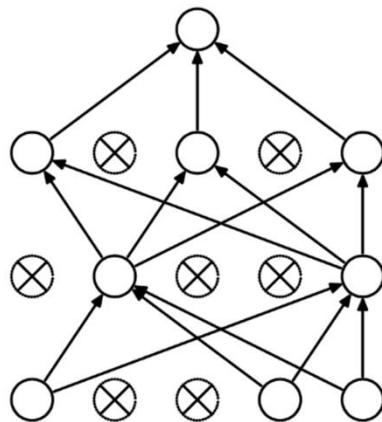
### 5.3.2 Dropout

This is the most popular regularization 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 [21] and further detailed in a paper by Nitish Srivastava et al. [22]. The algorithm works as follows (Figure 143):

- 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 [15]:

*“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.”*

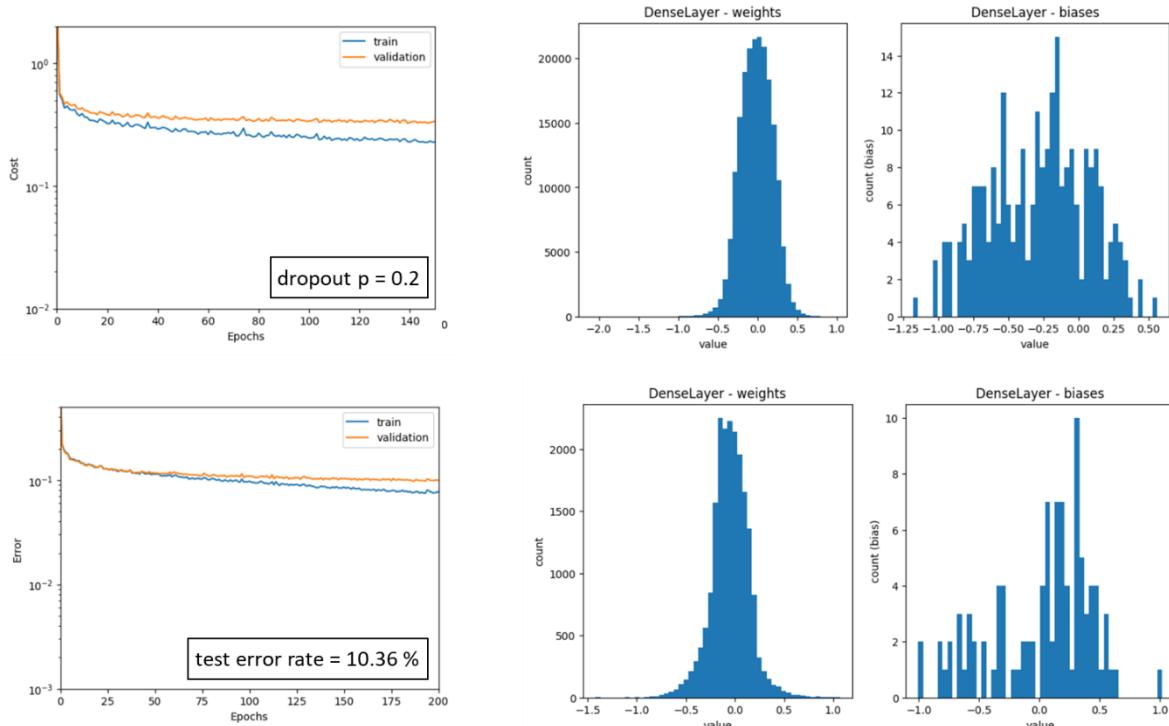


**Figure 143:** Illustration of Dropout in a neural network.

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.



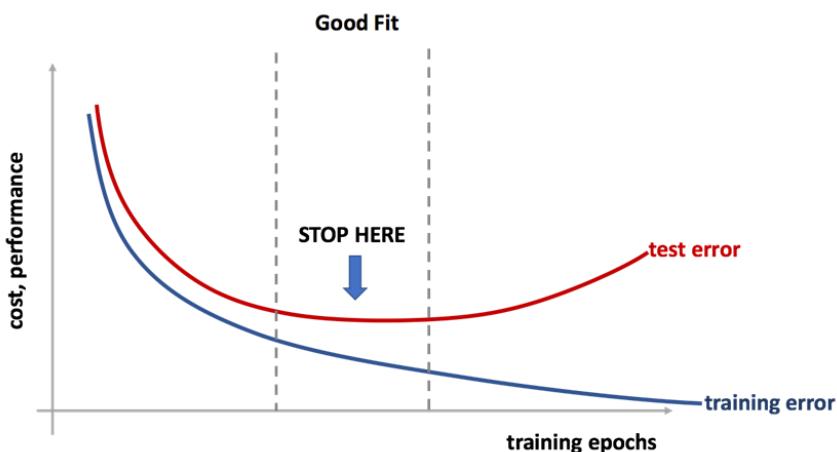
**Figure 144:** Results corresponding to the settings of Figure 138 but with dropout regularisation (result of PW-05). Overfitting is efficiently reduced, and a certain rescaling of the weights is visible.

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.

### 5.3.3 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 3.6.1). For early stopping the training is stopped when the validation or test error begins to increase again while training error still decreases.



**Figure 145:** Illustration 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.

## 6 Deep Learning Frameworks

Following the content of the last chapters we managed to program a full MLP architecture including backpropagation with an arbitrary number of hidden layers (and neurons), with flexibility concerning the choice of the activation function, with the possibility to add regularisation and with different Gradient Descent schemes. Nevertheless, we realized that the effort therefore was continuously increasing and the processing time on a state-of-the-art computer – even for a relatively small (in terms of resolution) image set like (Fashion-)MNIST – was considerable.

In chapter 1.2.2 we already discussed the main drivers of DL and in particular the quantitative progress in form of training data. With increasing data sets the demand with respect to computational power and efficiency was also increasing and the scientist were moving away from generic multi-purpose processing frameworks – like numpy – to frameworks that were highly optimised for the task of training and evaluating deep machine learning models. In particular the possibility to accelerate the training through highly parallelizing the task on a GPU made such frameworks almost a necessity. Meanwhile there is almost a plethora of frameworks available, Figure 146 giving a non-exhaustive overview with the comparison made according to whether the framework was developed in academia or in industry<sup>104</sup>. From all the frameworks two emerged as the most prominent ones which are PyTorch<sup>105</sup> and TensorFlow<sup>106</sup>. In Figure 147 the number of publications in the field of DL has been analysed and the frequency of the applied framework was established. It shows that the importance of TensorFlow – at least in academia – is continuously decreasing and PyTorch is emerging as the main framework for DL-architecture implementation. However, one should keep in mind that this represents to a large extend only the picture of academia and that some of the main breakthroughs in DL were obtained by companies e.g., Deepmind<sup>107</sup> with the Go-program AlphaGo [23] and the program AlphaFold, which is capable of predicting the structure of around 200 million proteins [24].

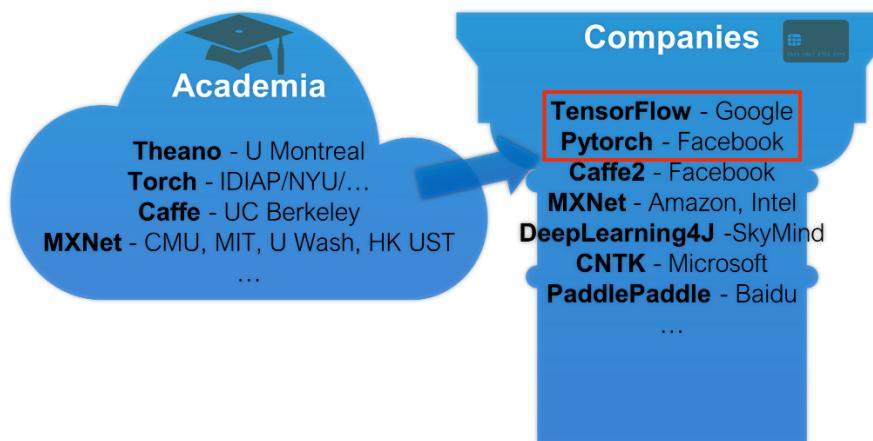


Figure 146: Meanwhile there is almost a «zoo» of frameworks for DL purposes available.

Here we will focus on these two frameworks i.e., PyTorch and TensorFlow and in addition introduce a high-level API for TensorFlow called Keras<sup>108</sup>. It is part of TensorFlow since version 2 and provides a very convenient abstraction level for defining DL-models using TensorFlow.

In Figure 148 the main differences between TensorFlow (left) and PyTorch (right) are summarised. Both frameworks were developed by big internet companies – having access to large amounts of data – which are Google and Meta (ex. Facebook) respectively. Furthermore, both frameworks took their roots in frameworks developed previously in academia. Theano, which is the basis for TensorFlow emerged from the university of Montreal, precisely speaking the Quebec AI Institute (originally Montreal Institute for Learning Algorithms)<sup>109</sup>, which was founded by J. Bengio (c.f. [1][17]). PyTorch is

<sup>104</sup> See also [https://en.wikipedia.org/wiki/Comparison\\_of\\_deep\\_learning\\_software](https://en.wikipedia.org/wiki/Comparison_of_deep_learning_software) for a larger overview.

<sup>105</sup> <https://pytorch.org/>

<sup>106</sup> <https://www.tensorflow.org/>

<sup>107</sup> <https://de.wikipedia.org/wiki/DeepMind>

<sup>108</sup> [https://www.tensorflow.org/api\\_docs/python/tf/keras](https://www.tensorflow.org/api_docs/python/tf/keras)

<sup>109</sup> <https://mila.quebec/en/>

based on Torch, which was originally developed by Idiap Research Institute at EPFL<sup>110</sup>. As we will see, PyTorch bears some close similarities with Python and – in terms of treatment of tensors – with numpy. This makes its use fairly intuitive for researchers familiar with Python and numpy. PyTorch has a dynamic strategy for the computational graph (i.e. the model), which is also referred to as “eager mode”. This means that operations are evaluated immediately. TensorFlow – in particular version 1 which is now no longer used – uses a static strategy, where the graph is composed beforehand and then executed in a so-called session. This has several advantages like execution speed and simpler portability. With the current version 2 of TensorFlow eager execution is also possible so that the two frameworks here are becoming similar.

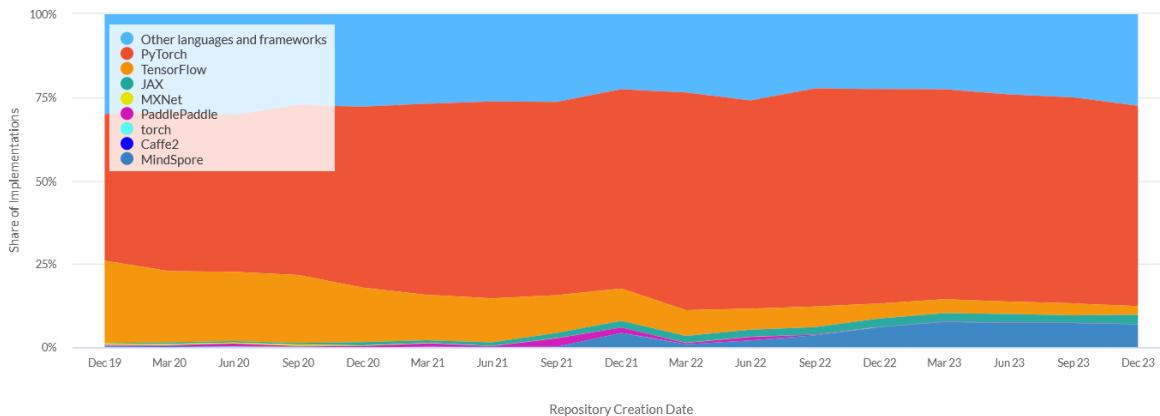


Figure 147: Paper implementations grouped by the DL framework used for the implementation<sup>111</sup>.

- |   |  |
|---|--|
| <ul style="list-style-type: none"> <li>From Google</li> <li>Based on Theano</li> <li>Initially based on a <b>static</b> computational graph strategy <ul style="list-style-type: none"> <li>e.g. faster on CNN, more cumbersome on variable input length such as in RNN</li> <li>From version 2.0: based on an “eager” compilation of the graph (a bit as in Pytorch), also more “Pythonic”</li> </ul> </li> <li>Steeper learning curve, people usually use high level APIs (Keras)</li> <li>Bigger community</li> <li>Seems more accepted to go to production in the industry</li> </ul> | <ul style="list-style-type: none"> <li>From (Facebook) Meta</li> <li>Based on Torch</li> <li><b>Dynamic</b> computational graph strategy <ul style="list-style-type: none"> <li>e.g. faster on RNN and slower on static architectures</li> <li>Can handle variable sequence length in RNN</li> </ul> </li> <li>Closer to python code, more “pythonic”, feel more “Python native”</li> <li>Younger, smaller community</li> <li>Better to do rapid prototyping, seems to be well accepted in the research community</li> </ul> |
|---|--|

Figure 148: Main differences between TensorFlow (left) and PyTorch (right).

To become familiar with the frameworks we will use the following approach:

As PyTorch and in particular the nn-library with the Sequential container is very similar to the way we designed our MLP, we will use it as a guide to develop an increasingly abstract view of handling DL-models. First, we will introduce the nn-library with the Sequential container of PyTorch, which allows to create and line up a large class of standard layer types (fully connected, Dropout, Convolutional, Pooling, ...). Then we will introduce the Dataset and DataLoader classes to handle reading and transformation of training data. Finally, we will learn about Tensorboard, a versatile tool to visualize intermediate and final results of the training procedure. Once we reached this stage, we will finally introduce the Keras API for TensorFlow, which provides the most abstract and compact approach to DL-model programming.

But in a first step we will learn some basic features both about PyTorch and TensorFlow to get an intuition about the low-level structure of these frameworks.

<sup>110</sup> <https://www.idiap.ch/en>

<sup>111</sup> <https://paperswithcode.com/trends>

## 6.1 Low-level view on PyTorch and Tensorflow

### 6.1.1 PyTorch

The best way to approach PyTorch – or the frameworks in general – is through “learning by doing” which will be the guideline for the hole chapter.

Exercise:

Using the iPython notebook `6.0.intro-pytorch_stud.ipynb` study the main similarities of PyTorch with numpy in particular concerning the handling of tensors i.e., multi-dimensional matrices:

- Study the cells [1] to [7] which deal with the conversion of tensors between the two frameworks and the creation of tensors from scratch in PyTorch. In addition the matrix multiplication is treated in PyTorch which is identical to numpy.
- Starting from cell [8] the main – and most important – difference between PyTorch and numpy tensors is discussed, the so-called `autograd` functionality. In fact, PyTorch can automatically determine the derivative of a scalar quantity – i.e. “`cost`” in cell [8] – with respect to any tensor (`w` and `b` in cell [8]) used in the derivation of this quantity. This requires, that a corresponding flag is activated during the definition of the tensors `w` and `b`:

```
W = torch.randn([2,3], dtype=torch.double, requires_grad=True)
b = torch.randn([2,1], dtype=torch.double, requires_grad=True)
```

Once the value of `cost` depending on the values of tensors `W` and `b` established

```
a = W@x + b
cost = a.T@a
```

the `backward`-operator is called on `cost`.

This operator determines recursively – c.f. section 4.5.1 – the derivatives of all tensors with the `requires_grad` flag equal to true. The respective gradients are then available in the field `grad` for each tensor, e.g.:

```
W.grad
b.grad
```

- It is important to notice, that the `grad` member of each tensor is not cleared automatically before a call to `backward()`. Thus, successive calls will accumulate the result as illustrated in cell [9]. In order to clear the `grad` member it must be set to `None` (c.f. cell [10]).
- A further problem may arise once you want to use the values of tensors with the `requires_grad` flag equal to true for further calculation, because they are restricted in their use. Nevertheless, you can always use the decorator

```
with torch.no_grad()
```

in order to suppress the gradient calculation (c.f. cell [11]).

- Finally we verify that the automatic calculation of the gradients by PyTorch corresponds to the correct analytic (cell [12]) and numerical<sup>112</sup> (cell [13]) result.

---

<sup>112</sup> C.f. see section 4.5.4.1.

### 6.1.2 TensorFlow

Again we want to discuss a simple example to investigate the low level functioning of TensorFlow.

Exercise:

Using the iPython notebook `6.1.intro-tensorflow_stud.ipynb` to study the similarities and differences of Tensorflow and PyTorch:

- Study the cells [2] where a – very simple – TensorFlow graph is created. In fact the decorator `tf.function` compiles a function into a callable TensorFlow graph. Here the same cost function (as in the previous exercise using `6.0.intro-pytorch_stud.ipynb`) based on `w` and `b` is determined and its gradients calculated. This represents the static nature of the graphs already discussed in relation to Figure 148. The advantage of the precompiled static graph execution is that it enables portability outside Python and tends to offer better performance<sup>113</sup>.
- Execute cell [2] to compile the graph and then use it in cell [3]. Finally cell [4] allows again to verify the gradient calculation. Note that the values of `w` and `b` are random and the results change with each call to cell [3].

It is obvious just from the few lines of code that TensorFlow uses a very different syntax than Python/numpy and also PyTorch. Thus, it requires a certain knowledge ramp-up to get familiar with it and to program more complex architectures. Therefore, we limit our presentation of the low-level code to these few lines and will – after introducing PyTorch – immediately switch to the high level Keras API, which will appear – after a closer look on PyTorch – rather intuitive.

## 6.2 Introduction to PyTorch

We will start with our MLP implementation in Python/numpy and step by step replace our code using PyTorch functionality. This will allow us to get a deeper understanding of the respective PyTorch functionality.

### 6.2.1 nn-Library and Sequential Container.

The nn-library<sup>114</sup> provides the possibility to set up all standard layer types just in a single call. Then we will organise them in the so-called sequential container<sup>115</sup>, which currently fulfils all our needs in terms of model complexity. PyTorch also allows to organise layers in a more flexible way using the `nn.Module` class where the `forward` method – corresponding to our `propagate` step – has to be defined by the user<sup>116</sup>. Because in the MLP the data flows linearly from the input to the output layer we can limit ourselves here to the sequential container. As before we will discuss the main points along an iPython sample script.

Exercise:

Use the iPython notebook `6.2.mlp_pytorch_1_stud.ipynb` to study the layers provided by the nn-library of PyTorch:

- Study the upper part of the notebook until the definition of the class `MiniBatches` which is copy of the solution to the PW-05 exercise `mlp_regularizer.ipynb`. Below that, follows immediately the definition of the class `NeuralNetwork`, because the individual layers (Dropout, Dense, Softmax) are now provided by the nn-library of PyTorch.
- Analyse the constructor of the class `NeuralNetwork`, and observe the following points (c.f. Figure 149):
  - Definition of the sequential container

<sup>113</sup> See also the explanation in the TensorFlow documentation [https://www.tensorflow.org/guide/intro\\_to\\_graphs](https://www.tensorflow.org/guide/intro_to_graphs)

<sup>114</sup> <https://pytorch.org/docs/stable/nn.html>

<sup>115</sup> <https://pytorch.org/docs/stable/generated/torch.nn.Sequential.html#torch.nn.Sequential>

<sup>116</sup> <https://pytorch.org/docs/stable/generated/torch.nn.Module.html>

- ```
        self.model = torch.nn.Sequential()


  - Sequential addition of modules in the container:  

self.model.add_module(...)
  - Separation of dense layer in calculation of logit ( $z = W \cdot x + b$ ) and activation function:  

torch.nn.Sigmoid()
  - The same is true for the final softmax layer with the following activation function definition:  

torch.nn.Softmax(dim=1)
  - The cost function – cross entropy loss – is simply a method of the class torch.nn:  

torch.nn.CrossEntropyLoss(...)
```

The diagram shows the Python code for the `NeuralNetwork` constructor with various annotations:

- `self.model = torch.nn.Sequential()` is labeled "define the sequential container".
- `#first construct dense layers` is labeled "add a new layer to the container".
- `for i0 in range(len(list_num_neurons)-2):` is labeled "calculation of logit  $z = (W \cdot x + b)$ ".
- `self.model.add_module('dense' + str(i0), torch.nn.Linear(list_num_neurons[i0], list_num_neurons[i0+1]))` is labeled "define the activation function".
- `self.model.add_module('act' + str(i0), torch.nn.Sigmoid())` is labeled "define the cost function".
- `#finally add softmax Layer` is labeled "define the cost function".
- `self.model.add_module('dense' + str(i0+1), torch.nn.Linear(list_num_neurons[-2], list_num_neurons[-1]))` is labeled "define the cost function".
- `self.model.add_module('act' + str(i0+1), torch.nn.Softmax(dim=1))` is labeled "define the cost function".
- `#define the cost function` is labeled "define the cost function".
- `self.cost_fn = torch.nn.CrossEntropyLoss(reduction='mean')` is labeled "define the cost function".

Figure 149: Constructor of class `NeuralNetwork`.

- Now we move to the method `optimise()` of our class `NeuralNetwork` (c.f. Figure 150). We can identify the main steps as before (i.e. in our numpy MLP), which are
  - the forward path
  - the backpropagation
  - and the gradient descent step.
 We nevertheless can identify an additional step which is the determination of the cost function, which is executed before the backpropagation step. The reason for that is the way PyTorch determines the gradients of the model parameters. We recall from section 6.1.1 that the gradients are determined from the scalar function `cost` by calling the `backward()` method. This is done in our method `back_propagate()` (c.f. Figure 152). Note that we also set the gradients to zero in the method `back_propagate()`.

The diagram shows the Python code for the `optimise()` method with annotations:

- `for ib in range(batches.number_of_batches()):` is labeled "forward path".
- `batch = batches.next()` is labeled "forward path".
- `#do prediction` is labeled "forward path".
- `y_pred = self.propagate(batch['x_batch'])` is labeled "forward path".
- `#determine the Loss` is labeled "cost is required for autograd functionality".
- `cost = self.cost_func(y_pred, batch['y_batch'])` is labeled "define the cost function".
- `#determine the error` is labeled "define the cost function".
- `self.back_propagate(cost)` is labeled "backpropagation".
- `#do the correction step` is labeled "gradient descent step".
- `self.gradient_descend(alpha)` is labeled "gradient descent step".
- `#calculate the error` is labeled "gradient descent step".

Figure 150: The method `optimise()` of our class `NeuralNetwork`.

The implementation of the propagate step (c.f. Figure 151) is straight forward because the model simply receives the input `x` (batch of images) and propagates them through the full sequential set of layers.

Finally observe the `gradient_descent()` method in Figure 153. The parameters of the model are available via an iterator and the gradient descent step can be performed for all parameters in one loop. It should be noted that the autograd functionality has to be deactivated in order to perform this step.

```
def propagate(self, x):
    """
    calculates the function ε
    """
    y_pred = self.model(x) ← forward pass
    return y_pred
```

**Figure 151:** The method `propagate()` of our class `NeuralNetwork`.

```
def back_propagate(self, cost):
    """
    calculates the backpropagation
    this function must be performed
    """
    #set gradient values to zero
    self.model.zero_grad() ← set gradients to zero
    cost.backward() ← determine derivatives
```

**Figure 152:** The method `back_propagate()` of our class `NeuralNetwork`.

```
def gradient_descend(self, alpha):
    """
    does the gradient descend based on result
    """
    with torch.no_grad(): ← deactivate gradient calculation
        for param in self.model.parameters():
            param -= alpha * param.grad ← iterate over parameter
  and do GD step
```

**Figure 153:** The method `gradient_descent()` of our class `NeuralNetwork`.

- Execute the script and perform one – or several – training runs. As an exercise you may replace the sigmoid activation in the dense layer with the ReLU function (c.f. Table 6) and repeat the training. You should observe a gain in accuracy.

In summary we realise that the nn-library of PyTorch provides a means to implement complex DL-architecture in just a few lines code due to the availability of essentially all relevant layer types<sup>114</sup>, which can be stacked in the sequential container<sup>115</sup> to a full model, and the possibility to determine automatically the gradients of the loss function using the autograd functionality. We will later in the final version (section 6.2.3) also replace the gradient descent step (c.f. Figure 153) with an optimiser provided by PyTorch and which provides the full set of methods discussed in chapter 5.2. But before that we want to understand how PyTorch reads and transforms the training data.

## 6.2.2 DataSet and DataLoader

The `DataSet` class<sup>117</sup> provides a generic way to prepare and sample datasets for the training step. In the notation of our numpy MLP it will take over the role of the function `prepare_data()` (data preparation) and the class `MiniBatches` (sampling). It will require as input a `Dataset` class, which is mainly a container for the data itself. PyTorch provides already a large choice of build-in datasets<sup>118</sup> but one can also define custom – i.e. own – datasets. The MNIST and FashionMNIST data we used so far are part of the available PyTorch models and are of the type `Dataset`. The `Dataset`

<sup>117</sup> <https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader>

<sup>118</sup> E.g. image datasets: <https://pytorch.org/vision/stable/datasets.html>

and DataLoader concepts allow to keep the code for maintaining the data set and the model – together with the training process – well separated (see also the following link for a concise introduction to these concepts<sup>119</sup>). Therefore, a further tool is used which is the transformation and – in a later step – the augmentation of the data. These are actually part of the torchvision sub-library of the PyTorch project<sup>120</sup> (as are the available datasets) and are passed through the transform argument to the Dataset class. torchvision provides a large set of available transformations and we will use type conversion and data normalisation below. We will now introduce these concepts using the following

#### Exercise:

Use the iPython notebook `6.3.mlp_pytorch_2_stud.ipynb` to study the Dataset and DataLoader classes as well as the transform concept of PyTorch:

- Study the cell [2] where the transform concept is introduced. We have to put it there because it is required for the definition of the dataset in the cell underneath. Our data transformation is named `my_transform` the name to be given to the dataset definition. It is a composition of three individual transformations which are:
  - `ToImage`: Conversion to a PyTorch tensor from type PIL image
  - `ToDtype`: Type conversion from `uint8` to `float32`
  - `Normalize`: Normalisation of data (here Min-Max Normalisation)

```

      name to be used in the Dataset definition
# import shortcut (just for here)
from torchvision.transforms import v2
my_transform = v2.Compose([
    v2.ToImage(), # Convert to tensor, data are PIL images
    v2.ToDtype(torch.float32, scale=False), # convert to float; optionally normalize data
    # (if True choose mean=[0.5], std=[0.5] below (why?))
    v2.Normalize(mean=[128.], std=[128.]), # the set of transforms to be applied
])

```

- Moving now to the – well known – cell [3] you realise a small difference, which is the replacement of the transform argument which was changed to the name of our custom transform.

```

data_set = 'FashionMNIST'

if data_set == 'MNIST':
    training_data = torchvision.datasets.MNIST(
        root='../week1/data',
        train=True,
        download=True,
        transform=my_transform) # name from transform definition
)

test_data = torchvision.datasets.MNIST()

```

- Now move on to cell [4] where we check that the predefined PyTorch datasets are of the type Dataset. To be precise they realise the interface of the class Dataset. As we will see below, this requires – apart from the `__init__` method which is obvious – the methods `__len__` and `__getitem__` to be implemented.

```

from torch.utils.data import Dataset

print(isinstance(training_data, Dataset)) # training_data is of type Dataset
print(hasattr(training_data, '__len__')) # Dataset requires these two methods
print(hasattr(training_data, '__getitem__'))

```

<sup>119</sup> [https://pytorch.org/tutorials/beginner/basics/data\\_tutorial.html](https://pytorch.org/tutorials/beginner/basics/data_tutorial.html)

<sup>120</sup> <https://pytorch.org/vision/stable/transforms.html#start-here>

- Cell[5] just illustrates that the `training_data` instance has the `transform` method or attribute:

```
print(hasattr(training_data, 'transform'))
```

- The cells [6] and [7] now introduce the `DataLoader` concept. The constructor the class `DataLoader` receives as input the instance of the `Dataset` class (`training_data`) we prepared beforehand. In addition, further training parameters as the batch size and sampling strategy (random shuffling or not) can be chosen. Refer to the PyTorch documentation<sup>117</sup> for a complete list of arguments.

Once the `DataLoader` set up an iterator is defined and used to get the data batch-wise, including the corresponding labels.

When executing the cell note the size of the returned tensor which is:

```
torch.Size([256, 1, 28, 28])
```

This differs from our numpy implementation where we used ([256, 784]). PyTorch leaves the original image dimension (28x28) because in other than dense network types – e.g. convolutional neural networks – this information is required. In addition, a further dimension – here of size 1 – occurs, which may represent e.g. the colour information of the image. As we were only dealing with black and white images so far, we did not require this information.

```
# dataloaders (we use original set (training/test_data); own data has to realize the abst
train_loader = torch.utils.data.DataLoader(training_data, batch_size=256, shuffle=True)
    receives as argument the Dataset
    ↑
    parameters used for training
#setup iterator
data_iterator = iter(train_loader) ← Set up an iterator
    ↑
#fetch first batch of images (and corresponding labels)
images, labels = next(data_iterator) ← get one batch of images (and labels)
    ↑
#note that the images are a 4-dim tensor; first dimension is image index in batch, second
print(images.shape)
print(labels.shape)
#print first ten labels -> change 'shuffle=True' to 'shuffle=False' to check that order w
print(labels[:10])
```

- In cell [8] we will now verify the application of the transformations during the iteration step over the data. It is *very important* to notice that the transformations are done on the fly by the `DataLoader` when iterating over the data and are not performed on the original data, as you can see when executing the cell [8].

```
#original data
print('type of original data: %r' % training_data.data.dtype) ← access to original data
print('min value of original data: %r' % torch.min(training_data.data).item())
print('max value of original data: %r' % torch.max(training_data.data).item())

#loaded data
print('\n type of loaded data: %r' % images.dtype) ← access to transformed data
print('min value of loaded data: %r' % torch.min(images).item())
print('max value of loaded data: %r' % torch.max(images).item())
#why is upper value not 1?
```

- In principle we are now ready to use the PyTorch `DataLoader` for the training – and testing – process and replace our own implementation using the function `prepare_data()` (doing the transformation) and the class `MiniBatches` (sampling and iteration).

But before that we do a test (cell [9]) of the DataLoader by looping over the entire dataset. We also measure the time for this operation:

```
#setup Loop over all batchs
data_iterator = iter(train_loader)

start=time.time() ← measure execution time
#note shape of last batch
print('shape of image batches:')
for batch_iter in data_iterator: ← iteration over full dataset
    print(batch_iter[0].numpy().shape)

end=time.time()
print('time for transforming entire batch: %2.1f s' % (end-start))
```

When executing the cell [9] one realises that the loop requires – depending obviously on the CPU power of the computer used – a time of the order of 10 seconds or more. Intuitively this appears to be a lot e.g. compared to our own numpy implementation. The problem lies in the fact that the PyTorch DataLoader applies the transformation on the fly when iterating over the data. For large dataset – and especially for data augmentation – this perfectly makes sense, and the overhead is justified. However here it would considerably slow down the execution of the training process and we will therefore implement our own Dataset class. Sooner or later, one is anyway obliged to do this using an own i.e. custom dataset, so it is a good exercise.

- Study the implementation of the class `MyDataset` in cell [10]. It inherits the class `Dataset` and therefore has to implement – apart from the `__init__` method which is obvious the two methods:  
`__len__`: returns the number of elements of the dataset  
`__getitem__`: returns a tuple containing one data element (a PyTorch tensor) and the corresponding label

```
class MyDataset(Dataset): ← inherit the class Dataset
    """Owns dataset."""

    def __init__(self, dataset, classes = torch.tensor([0,
        """])
        Arguments:
            dataset -- a tuple with the [images, labels] of the
            classes -- list of classes to use for training (at
            min one normalization -- whether to do min max normaliz.
```

When you study the implementation you will see that we mainly “recycled” our function `prepare_data()` in the constructor the class, where we make a copy of the original data and apply all transformations once and forever. Note that this is only possible because we are dealing here with small datasets – in terms of image resolution – like MNIST and FashionMNIST. For larger image resolutions only the PyTorch way – i.e. transforming the images on the fly during the iteration access – is an option.

- Now execute cells [11] and [12] to test the implementation of `MyDataset` and to verify that the execution time of the loop over the full dataset is now barely noticeable on a state-of-the-art computer.
- We are ready now to finalise our implementation i.e., the replacement of the function `prepare_data()` and the class `MiniBatches` by the PyTorch DataLoader including our own class `MyDataset`.  
When moving down in the script you will observe that for plotting the images tiles the data is accessed using the DataLoader.  
Move further on to the implementation of the class `NeuralNetwork` where the main differences were applied.

In the Sequential container we added a new layer of type Flatten. This is necessary because we left – for reasons of compatibility – the dataset organised according to the PyTorch size convention i.e., ( $\text{[#b, 1, 28, 28]}$ ). Here  $\text{#b}$  denotes the batch size. The flatten-layer just stretches the data between the indicated dimensions and with the present configuration we recover the size ( $\text{[#b, 784]}$ ) required as input to the dense layer.

```

self.model = torch.nn.Sequential()
#now we require a flatten tensor
self.model.add_module('flatten', torch.nn.Flatten(start_dim=1, end_dim=-1))
#first construct dense Layers
for i0 in range(len(list_num_neurons)-2):
    self.model.add_module('dense' + str(i0), torch.nn.Linear(list_num_neurons[i0], list_num_neurons[i0+1]))
    self.model.add_module('act' + str(i0), torch.nn.ReLU())

```

Furthermore in the `optimise` method we now use the `DataLoader` – based on our own Dataset class `MyDataset` – to access the training and validation (in method `append_result`) data.

```

# dataloader for training image
train_loader = torch.utils.data.DataLoader(data['train'], batch_size=batch_size, shuffle=True)

# save results before 1st step
if self.epoch_counter == 0:
    res_data = self.append_result()

for i0 in range(0, epochs):
    #measure time for one epoch
    start=time.time()
    #setup Loop over all batches
    data_iterator = iter(train_loader)
    for batch_iter in data_iterator:
        #do prediction
        y_pred = self.propagate(batch_iter[0])
        #determine the loss
        cost = self.cost_func(y_pred, batch_iter[1])
        #determine the error

```

- You can now execute the corresponding cell to start the training. You may also observe the way the data is prepared to be handed over to the `optimise` method of the `NeuralNetwork` class. Note that the number of (object-) classes was reduced to speed up processing.

Apart from the optimiser – topic, which we will also address in the next section – we now replaced our complete original numpy implementation for accessing and transforming the training data as well as the setup and training of the MLP by the PyTorch framework.

The only larger code sections that remain concern the presentation of the results during and after the training. We will present the tensorboard library in the next chapter as a flexible and versatile tool to perform this task.

### 6.2.3 TensorBoard for Result Visualization

Tensorboard<sup>121</sup> is actually part of the TensorFlow framework but – due to its open source character – it can also be used by other frameworks e.g. by PyTorch<sup>122</sup>: “*TensorBoard is a visualization toolkit for machine learning experimentation. TensorBoard allows tracking and visualizing metrics such as loss and accuracy, visualizing the model graph, viewing histograms, displaying images and much more*”. PyTorch uses TensorBoard via the class `SummaryWriter`<sup>123</sup>, which provides methods for sending virtually any kind of data to TensorBoard for visualisation.

For the following it is rather important to understand the big picture of the architecture of the software

<sup>121</sup> <https://www.tensorflow.org/tensorboard>

<sup>122</sup> [https://pytorch.org/tutorials/recipes/recipes/tensorboard\\_with\\_pytorch.html](https://pytorch.org/tutorials/recipes/recipes/tensorboard_with_pytorch.html)

<sup>123</sup> <https://pytorch.org/docs/stable/tensorboard.html#torch.utils.tensorboard.writer.SummaryWriter>

components involved when using TensorBoard as shown in the following Figure 154:

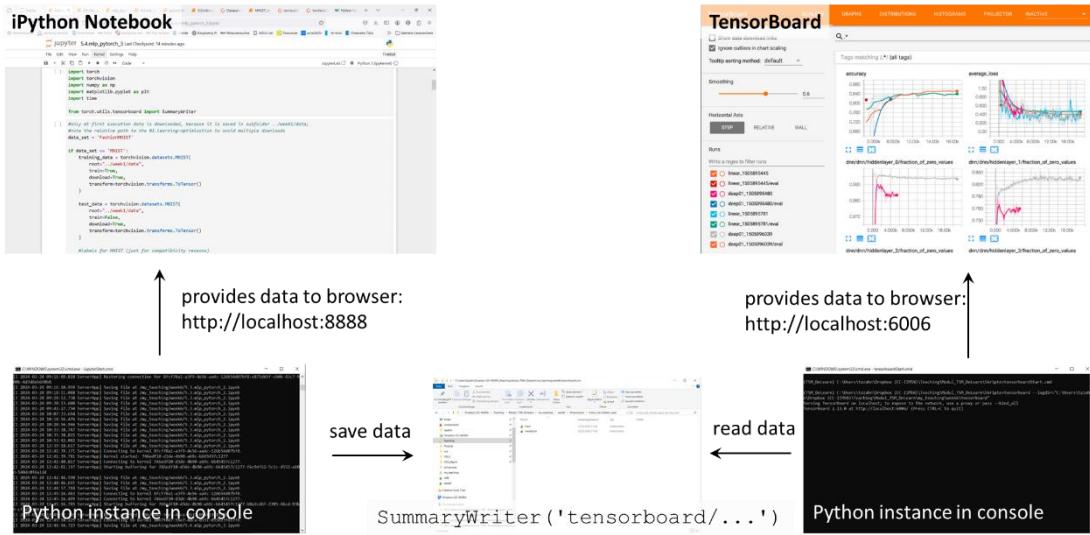


Figure 154: Overview of the SW-architecture for the use of TensorBoard (details see text).

The left side of the diagram should already be familiar for us with the Python instance running – in an Anaconda environment – as console application. The web application Jupyter Notebook serves as programming interface for the Python application. The browser connects to the webserver in the Jupyter Notebook via the local port 8888<sup>124</sup>.

TensorBoard (right side) works the same way. I.e. it is a console application running inside an Anaconda environment and it provides the data to a browser – where the visualisation is done – thought the local port 6006. However, we have the problem that our Python program (left) and TensorBoard (right) must exchange data. To be precise the data to be visualised must be sent from our application to TensorBoard. This is done in an *asynchronous* way through a file exchange. In fact, when creating the class `SummaryWriter` in our program using the PyTorch framework a folder path is given where all subsequent calls to the `SummaryWriter` instance will be saved in a TensorBoard compatible format. Thus, TensorBoard has no information on whether or when information is stored. It simply polls from time to time, whether data is available in the configured directory. Therefore, the TensorBoard instance must know the directory name. *It is given as argument (use the absolute path) when starting TensorBoard*, and which is done the following way<sup>125</sup>:

```
tensorboard --logdir="C:\ABSOLUT_PATH\tensorboard"
```

To put it in other words, the folder names given in the constructor of the class `SummaryWriter` in our Python program and the `logdir` argument for the start of TensorBoard *must* be identical. Otherwise TensorBoard cannot know, where to search for data. On the other hand, our notebook (or Python program) and TensorBoard work completely independent of each other, which can also be an advantage. Therefore, for the moment there is no need to start TensorBoard yet and we can first investigate the code changes required for the output of the data to be visualised.

#### Exercise:

Use the iPython notebook `6.4.mlp_pytorch_3_stud.ipynb` to study how the class `SummaryWriter` is used to output data for visualisation in TensorBoard:

- The first cells [1] to [3] should be familiar and you can move down to cell [4] where the class `SummaryWriter` is instantiated with the call:

```
writer = SummaryWriter('tensorboard/fashion_mnist_experiment')
```

<sup>124</sup> Note that this port changes if you use more than one Jupyter Notebook instances on the same machine.

<sup>125</sup> This is the syntax (in particular the double quotes for the absolute path to the folder) for Windows operating systems. For other OS types consult the respective documentation.

Note that this path must not be absolute because the browser will create the output directory relative to the current working directory. Once you execute the line the folder will be created. Check its existence in the file explorer.

### Set output directory for tensorboard

This folder is relative to the working path on the hard disk

```
writer = SummaryWriter('tensorboard/fashion_mnist_experiment')
```

- Now observe the cell [5] where we send a first set of data to TensorBoard – to be precise we simply save it in the output folder, where TensorBoard will pick it up. We collect a set of 20 images and add the missing colour channel. Then we use an utility of the `torchvision` library – similar to our function `plot_tiles()` – to organise these images as tiles of a single image. Finally, we save the data through the `SummaryWriter` instance to the output folder

```
# get some random training images (add map dimension at position 1)
my_images = torch.unsqueeze(training_data.data[:20],1) ← read a set of 20 images and add colour channel

# create grid of images
img_grid = torchvision.utils.make_grid(my_images) ← organise the images as tiles (similar to our function plot_tiles())

# write to tensorboard
writer.add_image('a_set_of_fashion_mnist_images', img_grid) ← write output to tensorboard
```

- Now it is the moment to start TensorBoard in an Anaconda environment from the console:

```
tensorboard --logdir="C:\ABSOLUT_PATH\tensorboard"
```

Make sure to replace the string `ABSOLUT_PATH` with the correct string for your computer environment. Also observe the output on the console after starting TensorBoard to verify the port (here 6006).

Now open a new browser window and paste the url shown in the console output into the address bar:

```
http://localhost:6006/
```

The TensorBoard windows as shown in Figure 155 should appear in the browser. At the beginning the images are not shown, because TensorBoard has to read them from the output folder of the `SummaryWriter`. This update is only done periodically but can be forced manually with the circular arrow in the upper right corner of the window. Then – magically – the images written by the `SummaryWriter` instance will appear. We will in the following write more data, also of other type. The orange bar at the top of the window allows to select the respective data types.

- Now that we understand and have set up TensorBoard we can finish to investigate the iPython notebook. We start with the method `append_result()` of the class `NeuralNetwork`. Here we calculate – at the end of each epoch – the current costs and errors of the training and validation

data. In addition we send them to TensorBoard using the call:

```
writer.add_scalars(...)
```

Note the **s**, because there exists a similar call without **s** for one single scalar. The present call here a dictionary of scalar values which will all be shown by TensorBoard in the same graph together for comparison. The first line is for the cost and the second for the error of the training and validation data sets.

```
self.result_data = torch.cat((self.result_data, res_data), 0)

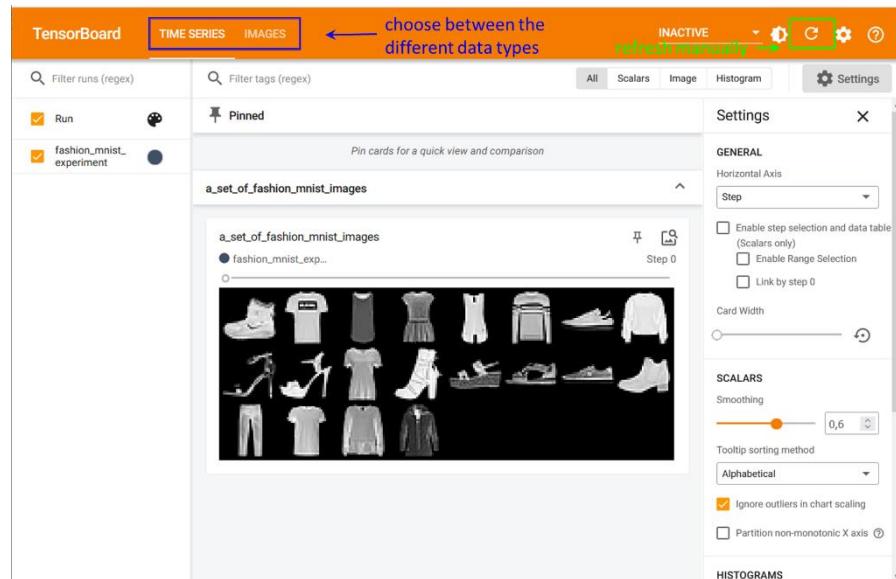
#send data to tensorboard
writer.add_scalars('loss', {'train': res_data[0, 0].item(), \
                           'validate': res_data[0, 2].item()}, self.epoch_counter)
writer.add_scalars('error', {'train': res_data[0, 1].item(), \
                           'validate': res_data[0, 3].item()}, self.epoch_counter)

#increase epoch counter here (used for plot routines below)
self.epoch_counter += 1
```

add a dictionary of scalars to the output folder

← train and validation cost

← train and validation error



**Figure 155:** The TensorBoard window as it appears in the browser with a set of images visualised.

- We move on further to the cell named “Sample execution of Neural Network”. In the present script we separated the instantiation of the class `NeuralNetwork` and the execution of the training. This is because we want to inspect the network graph in TensorBoard before the training. For complex graph architecture this is a convenient way to verify its correctness. To do so, you have to execute the cell named “Send the graph to tensorboard” (you can also execute the cell below with the embedding data). Once you refresh the TensorBoard window in the browser, the icon for the graph (blue rectangle in Figure 156) should appear. Select it and use the information on Figure 156 as a starting point to investigate the model graph architecture.

If you sent the data for the embedding to TensorBoard you can also inspect it. If the projector icon does not appear in the TensorBoard window (blue rectangle in Figure 157) you may have to close the TensorBoard console application and restart it again (you can keep the browser window open). The projector allows to embed high dimensional data into the 3D space. The idea is to get an intuition on the closeness of different data categories and there are different algorithms for the projection (green rectangle in Figure 157) PCA<sup>126</sup> and t-sne<sup>127</sup> being the

<sup>126</sup> [https://en.wikipedia.org/wiki/Principal\\_component\\_analysis](https://en.wikipedia.org/wiki/Principal_component_analysis)

<sup>127</sup> [https://en.wikipedia.org/wiki/T-distributed\\_stochastic\\_neighbor\\_embedding](https://en.wikipedia.org/wiki/T-distributed_stochastic_neighbor_embedding)

most popular ones. Keep in mind that our 3D-intuition is completely wrong in high dimensional spaces (c.f. section 4.1) but it is nevertheless an interesting tool.

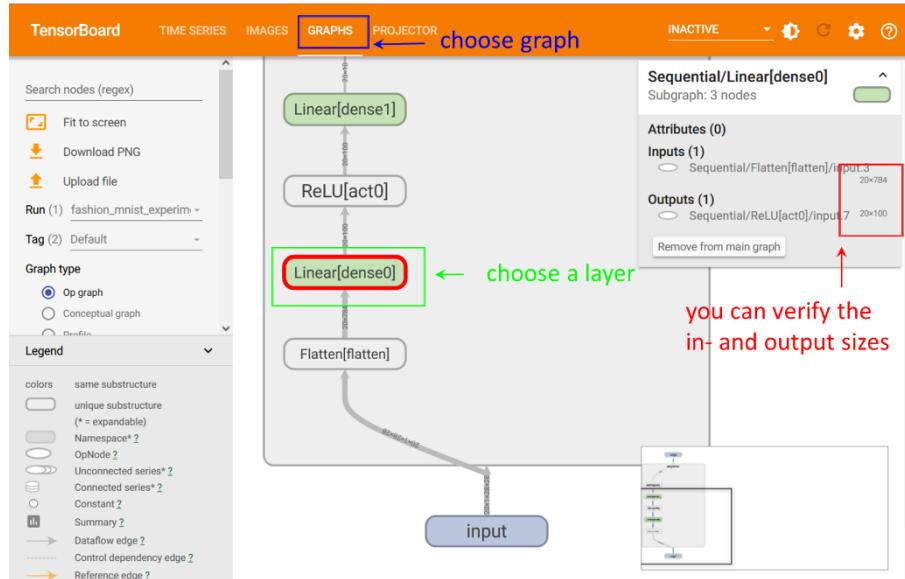


Figure 156: Presentation of the model graph in TensorBoard.

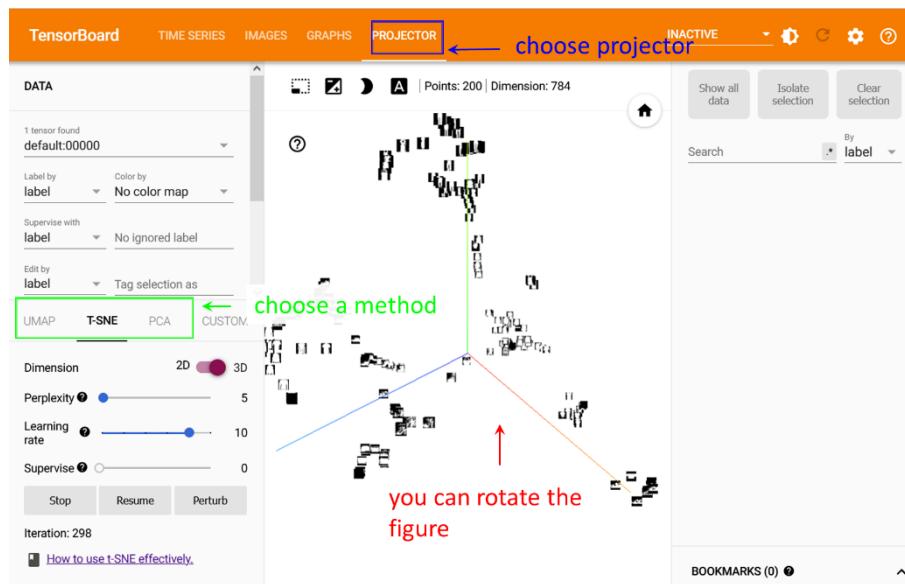
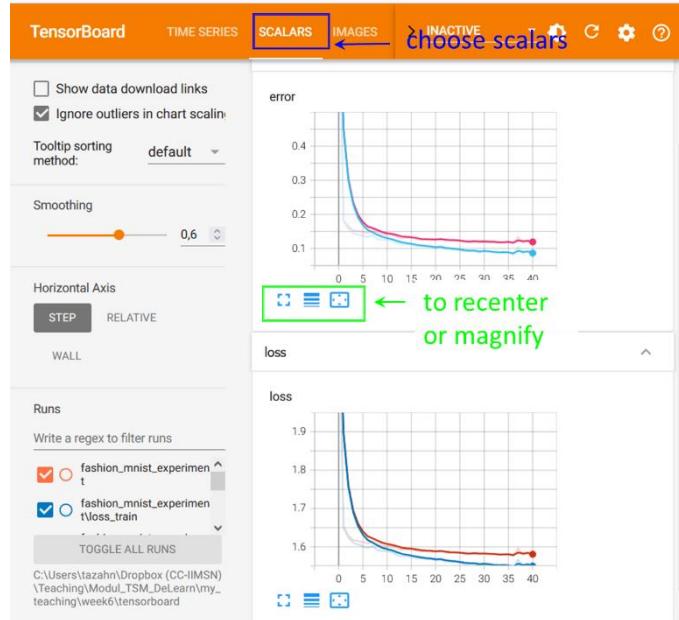


Figure 157: Embedding of a set of FashionMNIST data onto 3D space.

- Now start the training in the iPython notebook and switch to TensorBoard. Choose the scalar category on the top of the window (blue rectangle in Figure 158) and observe the presentation of the training curves. These represent the data we send in the method `append_result()` of the class `NeuralNetwork` to the output folder. The data in TensorBoard is updated periodically or you can force it by using the manual button. It is obvious that this represents a convenient tool to visualise the progress of the training especially for long runs. In addition, as the data is saved to file, it can be reviewed later and/or used for comparison of different runs.

TensorBoard has a plethora of further options, some of which we will see in chapter 6.3 where we will use TensorBoard together with TensorFlow. As TensorBoard is integral part of TensorFlow its integration is implicitly given and more options for visualisation are immediately available.



**Figure 158:** Training curves (cost and error) as shown in TensorBoard.

- To finish the exercise – and the introduction to PyTorch – we will again have a look at the class `NeuralNetwork` in the code. In addition to the use of TensorBoard we introduced a further change with respect to the previous iPython notebook (`6.3.mlp_pytorch_2_stud.ipynb`) which is the use of the PyTorch inbuild optimiser. This is done at the top of the method `optimise()`. We left the actual gradient step in the method `gradient_descend()`, which is the single call to:

```
self.optimizer.step()
```

This will perform the GD step using the configured optimiser for all model parameters. The optimiser will take care of everything.

```
method
NeuralNetwork::gradient_descend()

def gradient_descend(self, alpha):
    """
    does the gradient descend based on results from last back_pro
    """
    with torch.no_grad():
        self.optimizer.step() ← GD step for all model parameters
```

```
method
NeuralNetwork::optimise()
    self.vaca = vaca
    #we define the optimiser
    self.optimizer = torch.optim.SGD(self.model.parameters(), lr=alpha, momentum=0.) ← selection and configuration of optimiser used
    #self.optimizer = torch.optim.Adam(self.model.parameters(), lr=alpha)
```

In summary we see that with a few lines of code we can build up, configure, feed, and train a complex DL-model using PyTorch with just a few lines of code. We will now summarize this in the following section.

#### 6.2.4 Summary: DL-model using PyTorch

In the previous sections we used our numpy model as starting point and guideline to replace – step by

step – our own implementation by PyTorch functionality. Here we will just summarise the relevant PyTorch calls – without any additional overhead – to illustrate the “big picture”:

1. Setup Dataset

Download (e.g. `training_data = torchvision.datasets.MNIST(...)`) an available dataset or create your own custom dataset (c.f. class `MyDataset()`).

2. Model definition

Set up the model (using `model = torch.nn.Sequential()`) and add the required layers using `model.add_module(...)`.

3. Cost function

Choose the cost function (CE or MSE) e.g. CE:

```
cost_fn = torch.nn.CrossEntropyLoss(reduction='mean')
```

4. Optimiser

Setup the optimiser e.g. SGD:

```
optimizer = torch.optim.SGD(...)
```

5. DataLoader

Setup the DataLoader using the downloaded or custom dataset (point 1):

```
train_loader = torch.utils.data.DataLoader(Dataset, batch_size=...)
```

6. Training

Perform the training which – for each epoch – is a loop over the full training set using the iterator of the dataset:

```
data_iterator = iter(train_loader)
for batch_iter in data_iterator:
    ...do sth....
```

The training is set up according to the following scheme:

a. Propagation or forward pass

Predict the outcome of the model using the current batch of data (first position of tuple `batch_iter`)

```
y_pred = self.model(batch_iter[0])
```

b. Determine the cost function

The cost function is required for the backpropagation step. Therefore we evaluate it:

```
cost = cost_fn(y_pred, batch_iter[1])
```

Note that the true outcome is in the second position of the tuple `batch_iter`.

c. Reset gradients

Before we calculate the gradients of all model parameters we have to clear the memory:

```
model.zero_grad()
```

As an alternative the optimiser can be used for this task:

```
optimizer.zero_grad()
```

d. Backpropagation

Using the autograd functionality of PyTorch for the cost function, we can determine the gradients of all model parameters:

```
cost.backward()
```

e. Gradient descend

Finally the gradient descend step is performed using the chosen optimiser by:

```
with torch.no_grad():
    optimizer.step()
```

## 7. Result visualisation

In case you want to visualise results using TensorBoard you must do the following additional steps before starting the training.

- a. Setup `SummaryWriter` class

Set up the writer instance for the (file) output of all data to be visualised:

```
writer = SummaryWriter('tensorboard/...')
```

- b. Setup output

Choose the data you want to visualise during the training and add a writer call for each of them. E.g. for the output of the training and validation loss:

```
writer.add_scalars('loss', {'train': ..., 'val': ...}, counter)
```

The dots ('...') have to be replaced by the numerical values.

- c. Start TensorBoard

Start Tensorboard from an Anaconda environment using the same path as defined in the `SummaryWriter` class:

```
tensorboard --logdir="PATH"
```

- d. Open Browser

Open browser and display the results using the url:

```
http://localhost:6006/
```

Having reached this kind of abstraction it will now – as a final step in this chapter – be relatively straightforward to become familiar with the Keras API for TensorFlow, which uses even a higher level of abstraction.

### 6.3 Introduction to Keras API for TensorFlow

In section 6.1.2 we already had a brief look at the low-level programming with TensorFlow. We saw that it provides mainly the same functionality as PyTorch but using a quite different syntax and programming paradigm (i.e. a static graph, c.f. Figure 148). During the first years after the first publishing of TensorFlow, programming with the low-level API was – due to the absence of a good alternative – the standard way. With the appearance of Keras<sup>128</sup>, a high-level interface (or API) appeared which became increasingly popular. In the beginning Keras had to be installed on top of TensorFlow but since the release of TensorFlow 1.4, it has been part of the TensorFlow Core API<sup>129</sup>. Since then, the use of the low-level TensorFlow API has become less frequent. Therefore, we will directly introduce the Keras API.

Keras is an open-source framework written in Python and first published in 2015. Originally it provided a uniform interface for various backends, including TensorFlow, Microsoft Cognitive Toolkit (formerly CNTK) and Theano. The aim of Keras was to make the use of these libraries as beginner and user-friendly as possible. After it became part of the TensorFlow Core API (since release 1.4), Keras still continued as a standalone library because originally it was not intended as an interface for TensorFlow alone, but as an interface for many libraries. However, with the release of Keras 2.4, multi-backend support was discontinued and since then, it refers directly to the implementation of TensorFlow version 2.

We will see that with the knowledge from the previous chapter, it will be quite easy to become familiar with Keras, especially with the Sequential model. As before we will use an iPython script as guideline to the introduction.

#### Exercise:

Use the iPython notebook `6.5.mlp_tensorflow_stud.ipynb` to study how the Keras API for TensorFlow is used:

- The first cells [1] – [8] will be very familiar because they contain the download and presentation of the data. Because TensorFlow works directly with numpy-array input we could leave

<sup>128</sup> The original Keras webpage: <https://keras.io>

<sup>129</sup> [https://www.tensorflow.org/api\\_docs/python/tf/keras](https://www.tensorflow.org/api_docs/python/tf/keras)

this part identical to our own numpy implementation. Thus, you can move on directly to cell [9] with the definition of the class `NeuralNetwork`.

- When examining the class `NeuralNetwork` one immediately observes that it is extremely “slim”. As already mentioned above this is in fact the intention of the Keras API to provide a very user-friendly approach to DL-frameworks.  
We start to study the method `__init__`, which is conceptually very similar to our PyTorch example:

- First we set up the Sequential model type:

```
self.model = tf.keras.Sequential()
```

- Then we successively add the different layers using the `model.add()` method. The first layer is of type flatten:

```
- tf.keras.layers.Flatten(input_shape=(1, 28, 28))
```

As for the PyTorch example, we use an input size of `(1, 28, 28)` to the network in order to be compatible in the future with more complex layer types (e.g. convolutional layers), which require the information on the image resolution. Furthermore – this way – we are compatible with colour image input.

Then a set of dense layers is added:

```
- tf.keras.layers.Dense(..)
```

Note that the dense layer receives as input only the size of the output, because the input size can be deduced from the layer configuration. The activation function type is given directly as argument to the dense layer constructor.

The final dense layer is configured to be a Softmax layer simply by choosing the appropriate activation function.

```
def __init__(self, list_num_neurons, alpha):
    """
    constructor

    Arguments:
    list_num_neurons -- list of layer sizes including in- and output layer
    alpha -- learning rate (required because optimiser must be known to compile model)
    """
    self.model = tf.keras.Sequential()           ← define the sequential model

    #we require a flatten tensor
    self.model.add(tf.keras.layers.Flatten(input_shape=(1, 28, 28))) ← Flatten layer
    add layers to the model
    for i0 in range(len(list_num_neurons)-1):
        self.model.add(tf.keras.layers.Dense(list_num_neurons[i0], activation='sigmoid')) ← Dense layer
    define activation functions
    #finally add softmax layer
    self.model.add(tf.keras.layers.Dense(list_num_neurons[-1], activation='softmax')) ← define activation functions

    print(self.model.summary())

    #choose the optimiser
    optimizer = tf.keras.optimizers.SGD(learning_rate=alpha) ← define the optimiser and
    the cost function

    Compile model
    #wrap the model to a tf.function (unless run_eagerly=True)
    self.model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])
```

- Now the optimiser can be defined: we choose standard gradient descend with configurable but fixed learning rate:

- optimizer = tf.keras.optimizers.SGD(learning\_rate=alpha)
- o As a final step the model is compiled i.e. it is wrapped into a tf.function (c.f. section 6.1.2), which is – as already mentioned before – one of the major difference with respect to PyTorch<sup>130</sup>. The compile method of the model receives different input arguments, which are the optimizer (defined before), the loss-function and the metric, which are evaluated by the model during training and testing.

```
self.model.compile(optimizer=optimizer,
                    loss='categorical_crossentropy',
                    metrics=['accuracy'])
```

This already completes the full setup of a MLP with arbitrary number of hidden layers. Obviously, there is a plethora of additional parameters and arguments that can be configured. Refer to the Keras API for TensorFlow documentation for further information<sup>129</sup>.

- The class NeuralNetwork has only one additional method – optimise – which is nothing but a wrapper for the call to model.fit, which performs the training. All relevant parameters are given as input arguments:

- o The data:

```
data['x_train'], data['y_train']
```

Data and labels are given as numpy arrays. Note that the labels are onehot vectors. The training and validation data split is done by the model itself according to the parameter validation\_split, which characterises the fraction of validation data.

- o The training hyperparameters e.g. batchsize and number of epochs are given as further argument.
- o Finally, the output type is specified. The parameter verbose specifies the granularity of the output during the training. The parameter callbacks can be used to send output to TensorBoard as we will see in the example below. Here we keep the output within the iPython notebook.

```
def optimise(self, data, epochs, valid_size=0.2, batch_size=0, debug=0, call_backs=None):
    """
    performs epochs number of gradient descend steps and appends result to output array

    Arguments:
    data -- dictionary with NORMALISED data
    epochs -- number of epochs
    valid_size -- fraction of data used for validation
    batch_size -- size of batches (1 = SGD, 1 < .. < n = mini-batch)
    debug -- output: 0 = silent, 1 = progress bar, 2 = one line per epoch
    call_backs -- use to send output to tensorboard
    """
    start the training
    #now start training (you can choose the train/validation set split)
    self.history = self.model.fit(data['x_train'], data['y_train'], ← training (+validation) data
                                  validation_split=valid_size, ← define validation split
                                  batch_size=batch_size, epochs=epochs, ← define training parameters
                                  callbacks=call_backs, verbose = debug) ← define output type
```

<sup>130</sup> Notice however, that with the flag run\_eagerly=True an eager mode – i.e. without precompiled graph as in PyTorch – can be forced.

Now everything is set up for training.

- Moving further to cells [10] and [11] we find our plot routines for the (here) accuracy (i.e. 1- error) and the loss. We adapted these functions to be compatible with the TensorFlow output.
- Now you can start the training in cell [12] (you may want to choose a number of classes less than 10 to speed up training) and observe the output. The cells [13] and [14] should also be familiar as they analyse the performance of the model on the testset ([13]) and visualise the weights of the first layer as image tiles ([14]).
- We already introduced the tool TensorBoard (section 6.2.3) to visualise results during and after the training. As TensorBoard is part of the TensorFlow API its integration is also straight forward. Move down to the last cell of the notebook ([15]) which is to a large extend a copy of cell [12], which we just used for the training. In addition, in cell [15] a tensorflow callback is set up, which is used as an argument for the call to `NNNet.optimise()` in the last line of the cell:

```
tensorboard = TensorBoard(
    log_dir='./tensorboard/' + name + '/',
    update_freq='epoch', write_graph=True,
    histogram_freq=1)
```

The callback receives as argument the output folder, which must then coincide with the folder used to start the TensorBoard instance (c.f. Figure 154 and exercise below). There are further parameters as the update frequency and the information to be sent.

```
#clear everything
tf.keras.backend.clear_session() ← clear history to free memory

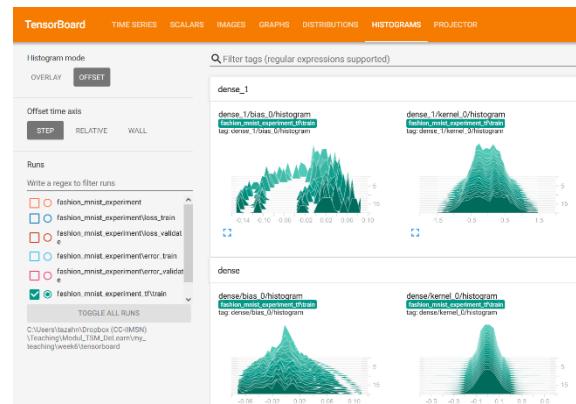
#you may have to delete the model to start from scratch
#del model

#output is written to the folder with name defined below
name = 'fashion_mnist_experiment_tf' ← define output folder

tensorboard = TensorBoard( ← define callback
    log_dir='./tensorboard/' + name + '/',
    update_freq='epoch', write_graph=True,
    histogram_freq=1) ← further parameters
  (c.f. documentation)

#choose the hyperparameters you want to use for training
epochs = 100
batchsize = 16
NNNet.optimise(data, epochs, valid_size=0.2, batch_size=batchsize, debug=0, callbacks=[tensorboard])
```

- Start TensorBoard and start the training through cell [15]. Open the browser and observe the output. You will notice that more output options are available automatically due to the embedding of Tensorboard into TensorFlow. E.g. the weight histograms are now available, and their evolution can be studied as a function of the training epoch.



## 7 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) emerged from the study of the visual cortex of mammals, which was found to be organised in hierarchical structures inspiring the CNN architecture (c.f. Figure 29). In a famous study [25] on the visual cortex of cats it was shown that many neurons in the visual cortex have only a small receptive field – partially overlapping – and that different neurons react to different stimuli e.g. to horizontal or vertical lines only. Moreover, it could be shown that certain neurons with larger fields of view react to more complex patterns that are combinations of the “low-level” neuron features. These studies inspired the so-called neocognitron [26], which gradually evolved in what we now call CNN. An important further milestone was the publication of LeNet-5 by Yann LeCun et al. [27] with application of CNN to the recognition of handwritten digits.

Nowadays CNNs are mainly used for visual classification and detection tasks but may be applied to any kind of 2D – or even 3D – data type. Here we will focus on the visual object classification e.g. like the recognition of the handwritten MNIST digits. Visual object detection – i.e. the localisation of visual object categories in images – is also possible with CNNs by applying certain changes or extensions to their architecture. But this topic will not be treated here.

CNNs introduce two new layer types which are convolutional and pooling layer. The convolutional layer realises the concept of a neuron with limited receptive field and in addition gives CNN the ability of translational invariance in the detection task. The pooling layer, which is a spatial decimation, introduces the concept of feature hierarchy.

Before we introduce the general architecture of the CNN, we will first recall some concepts of the MLP, which will allow us to understand its limitations in terms of the visual classification task and then have a quick look at the convolution-concept, as it is used in classical image processing.

### 7.1 Visual Object Classification with MLP

While computers play chess at human level since the 1997, image recognition at human level has only been possible since about a decade ago. The problem is that a given visual object category may be subject to considerable interclass variation, viewpoint changes, deformations, occlusions, ... (Figure 159). A classification task should be able to cope with all these changes. However, the MLP architecture has certain weaknesses that limit their performance on visual classification tasks.



**Figure 159:** The category «dog» may be subject to considerable interclass variation, viewpoint changes, deformations, occlusions. A classification or detection task should be able to cope with all these changes<sup>131</sup>.

<sup>131</sup> Selection of images from the [CIFAR-10](#) image set.

Exercise:

We use the iPython notebook `7.0.mlp_pytorch_cifar_stud.ipynb` to review some features of the MLP:

- The script is mainly a copy of the iPython notebook `6.4.mlp_pytorch_3_stud.ipynb` used in chapter 6.2.3. We just removed some of the output to TensorBoard. Here we will use it to review the fact, that the weights of the first layer learn the structure of the corresponding categories (c.f. Figure 91). Therefore, we will use the CIFAR-10 image set.
- Quickly review the structure of the notebook:
  - Cell [2] reads the CIFAR-10 image set.
  - Cell [3] contains helper functions for graphical output.
  - Cell [4] is our custom dataset discussed in chapter 6.2.2.
  - Cell [5] initialises the SummaryWriter for TensorBoard.
  - Cell [6] defines the Neural Network class using PyTorch.
- In cell [7] we configure a simple Generalised Perceptron i.e., without any hidden layer. This corresponds to the situation shown in Figure 91, where the number of weight vectors corresponds to the number of categories. We furthermore reduce the random initialisation of the weights as this will make the result look “nicer”.

```
#choose the hyperparameters you want to use for the initialisation
size_in = train_dataset[0][0].flatten().shape[0] #access to first image in torchSubset train_data
size_out = 10
list_num_neurons = [size_in, size_out];
NNNet = NeuralNetwork(list_num_neurons) ← no hidden layer (Generalised Perceptron)

#we reduce the weights to make result somewhat "nicer"
NNNet.model[1].weight.data = NNNet.model[1].weight.data/30 ← reduce random weight initialisation
```

- In cell [8] we trigger the training (only one epoch with small learning rate) and the cells [9] and [10] allows to visualise the weights reorganised as 32 x 32 image patches (Figure 160).
- Execute these cells and discuss the results with your neighbour. In particular have a look at the ‘two-headed’ horse and consider on how the algorithm behaves under the different transformations discussion in Figure 159.
- Now move back to cell [7] and configure a MLP with one single hidden layer and some 100 neurons:

```
#choose the hyperparameters you want to use for the initialisation
size_in = train_dataset[0][0].flatten().shape[0] #access to first image in torchSubset train_data
size_out = 10
list_num_neurons = [size_in, 100, size_out];
NNNet = NeuralNetwork(list_num_neurons) ← one hidden layer

#we reduce the weights to make result somewhat "nicer"
NNNet.model[1].weight.data = NNNet.model[1].weight.data/30 ← reduce random weight initialisation
```

Again trigger the training (use some 5 epochs at the same learning rate) and visualise the result of the weights of the first layer. Now it should look similar to Figure 161.



**Figure 160:** Weights of a Generalised Perceptron learn the structure of the corresponding object categories<sup>132</sup>.



**Figure 161:** Weight vectors of the first hidden layer of a MLP for classification of CIFAR10 data.

The images in Figure 160 and Figure 161 have been slightly “tuned” to show the desired effect but the observation is nevertheless clear. For the Generalised Perceptron in Figure 160 the classifier just learns some sort of average of all representations of a given category. This explains the “two-headed” horse. In the more complex case in Figure 161 different representations of a given category are learned by the classifier. In any case, two apparent problems arise:

1. The MLP learns the position of an object “by heart” or in other word each position of an object – in particular if the image is larger – would have to be learnt. The algorithm is not intrinsically translation invariant.
2. So far, we only used small image sizes of some 1000 pixels in total. Consider now the case of a state-of-the-art colour image of 400 x 400 pixel, which leads to 480'000 input values. An MLP with one hidden layer of “only<sup>133</sup>” 1000 neurons would already require 480 million parameters, which would become quickly intractable.

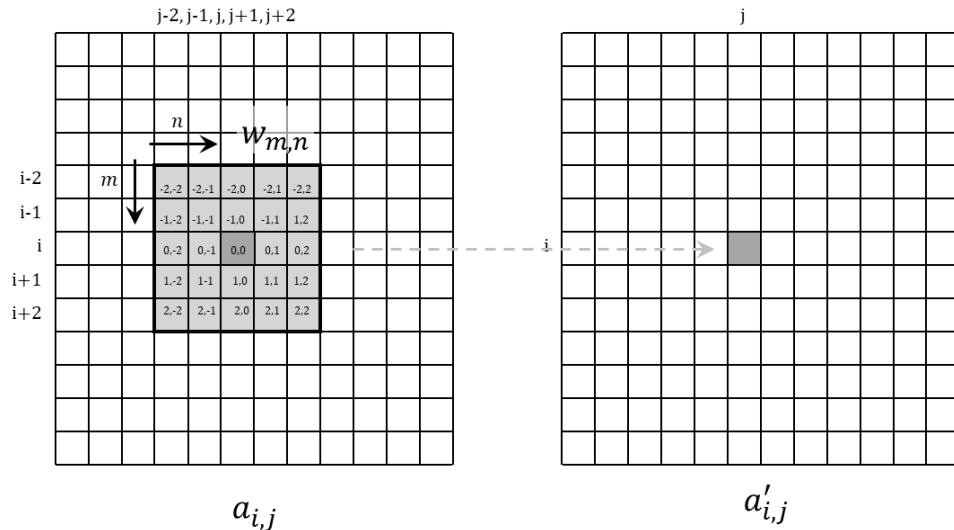
We will see in the following, how the idea of convolutional layers will allow to solve these two problems i.e., to introduce translational invariance and to considerably reduce the number of parameters. We will in a first step discuss the convolution from the point of view of classical images processing.

<sup>132</sup> Idea taken from [https://cs231n.stanford.edu/slides/2021/lecture\\_2.pdf](https://cs231n.stanford.edu/slides/2021/lecture_2.pdf)

<sup>133</sup> Considering the number of input values this would be fairly small in order to learn arbitrary positions, viewpoints, etc. for a larger class of object categories (c.f. Figure 161).

## 7.2 Convolutions in Image Processing

Convolutions are a well-known tool both in digital signal and in image processing. The two most important types are low-pass and high-pass filters. We will focus here on the latter type as these are most relevant for understanding convolutional layers.

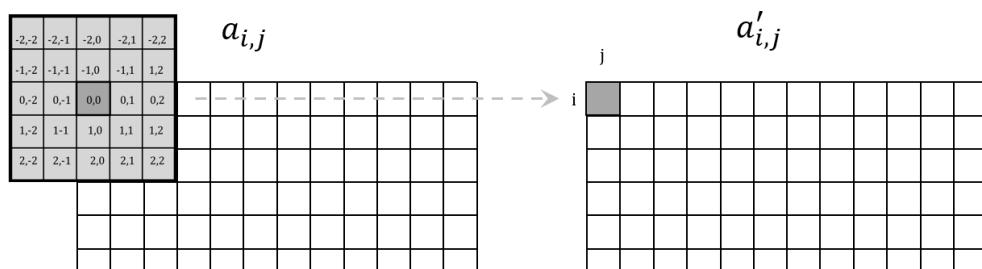


**Figure 162:** Convolution of an image  $a_{i,j}$  with a kernel (or mask)  $w_{m,n}$  (of size  $5 \times 5$ ) to obtain the result  $a'_{i,j}$ .

Figure 162 represents the convolution of an image<sup>134</sup>  $a_{i,j}$  with a kernel (or mask)  $w_{m,n}$  (of size  $5 \times 5$ ) to obtain the result  $a'_{i,j}$ . Therefore, all image pixel in the “receptive” region of the kernel ( $5 \times 5$ ) centred at image pixel  $a_{i,j}$  are multiplied with the respective kernel values  $w_{m,n}$  and their sum is assigned to  $a'_{i,j}$ . One can visualise this operation as sliding the kernel – e.g. row-wise – over the full image  $a_{i,j}$  to obtain  $a'_{i,j}$ . Mathematically, the operation for the convolution can be formulated as follows:

$$a'_{i,j} = \sum_{m,n} a_{i+m, j+n} \cdot w_{m,n} \quad \forall i, j$$

However, this immediately raises the question of the treatment of the border pixel. As shown in the following Figure 163, if we want to preserve the size of the original image  $a_{i,j}$  – and e.g. as shown in the figure want to treat pixel  $a_{0,0}$  – we have to extend (“pad”) the original image to provide the missing values to the kernel. Default padding is usually to extend the original image with zeros, but more robust versions exist like reflecting the image pixels at the border to avoid artefacts arising from discontinuities due to zero-padding.



**Figure 163:** To preserve the size of the original image  $a_{i,j}$ , padding is required at the borders.

For the application of convolutions in image processing typical kernel choices of size  $3 \times 3$  for edge detection (high-pass filter) in x- or y-direction are (left and right respectively)<sup>135</sup>:

<sup>134</sup> The output of the convolutions will be called activation (maps). Therefore we use the letter  $a$ .

<sup>135</sup> Note the sign-convention for the y-direction (minus-sign in top line of kernel) which is because the y-index increases from top to bottom.

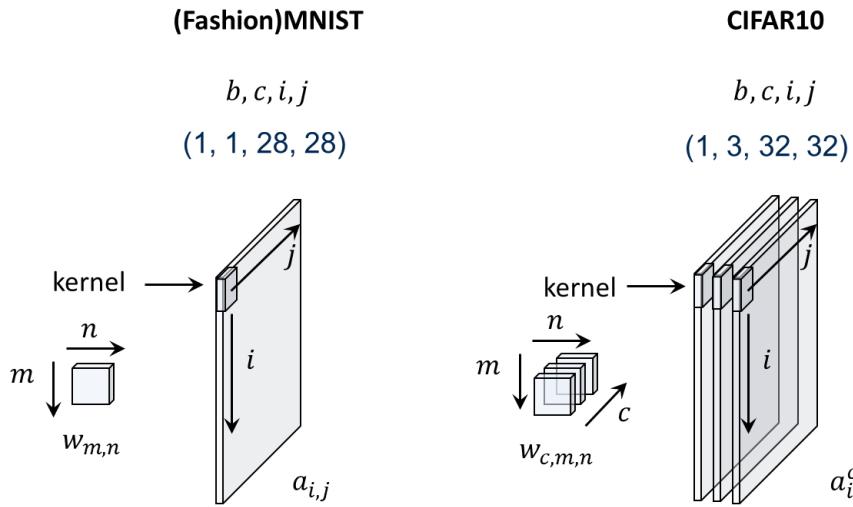
$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

While edge detection filters represent an approximation for the first derivative the second derivative can be approximated with the Laplace filter:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

The classic formulation of convolutions as summarised above usually deals only with single channel input images. This is the case illustrated in Figure 164 on the left-hand side, where a kernel  $w_{m,n}$  is applied on a MNIST or FashionMNIST image  $a_{i,j}$  of size 28 x 28 pixel.



**Figure 164:** (left) Classical convolution on a single channel image and extension (right) to three channel case (details c.f. text). Also note the 4D-indices ( $b, c, i, j$ ) of the image including  $b$  the index of the image in the batch.

For use in CNNs we will extend the definition of convolutions to multi-channel input images. This is represented for three channel i.e., colour images in Figure 164 on the right. Input is a CIFAR10 image with three colour channels  $a_{i,j}^c$  i.e.,  $c = 0, 1, 2$ . Now the kernel also must have three channels  $w_{c,m,n}$  where  $c = 0, 1, 2$ . Thus, the kernel now has three channels e.g. of size 3 x 3 (other types possible and frequent) and the values of the three channels can obviously be different. The formula for the convolution now reads:

$$a'_{i,j} = \sum_{c,m,n} a_{i+m,j+n}^c \cdot w_{c,m,n} \quad \forall i, j$$

Note that the result is again only a single channel image.

#### Exercise:

We use the iPython notebook `7.1.intro_convolution_stud.ipynb` to review the concept of convolutions in image processing for single channel images and extend it to multi-channel:

- Quickly review the structure of the notebook:
  - Cell [2] allows to choose between the (Fashion)MNIST (single channel, value 1) or CIFAR10 (multi-channel, value 3) image sets:

```
num_images_planes = 1.
```

This allows to study convolutions on single channel or multiple channel images. Note the size conversion of the images to a 4D-tensor.

```

original data types and shape
<class 'torch.Tensor'>
torch.Size([60000, 28, 28])
<class 'torch.Tensor'>

converted data types
torch.Size([60000, 1, 28, 28]) ← 4D tensor

```

We start with the single channel option.

- Cell [3] and [4] define the plot routines.
- Cell [5] now allows to apply the convolution.  
At the top you can choose the object category and further below the kernel type. The edge detection in x-direction is implemented. You have to complete the y-direction and the Laplace filter.

```

#choose a given class 0..9
category = 0 ← choose object category

x_sel = x[y == category]
img_grid = torchvision.utils.make_grid(x_sel[:40])
print(labels_map[category])

#here you can define the kernel type
kernel_tpye = 0 ← choose kernel type

if kernel_tpye == 0:
    #edge detection in x-direction
    kernel_1D = np.array([[[-1., 0., 1],[-2., 0., 2],[-1., 0., 1]]], dtype=np.float32)
elif kernel_tpye == 1:
    #edge detection in y-direction
    kernel_1D = np.array(['implement filter'], dtype=np.float32).T
elif kernel_tpye == 2: ← implement missing filter types
    #Laplace filter for edge detection (2nd derivative)
    kernel_1D = np.array(['implement filter'], dtype=np.float32)

print('kernel type:')
print(kernel_1D)

```

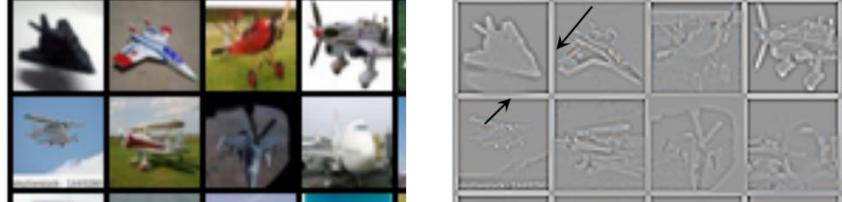
Visualise and interpret the result as shown in Figure 165.

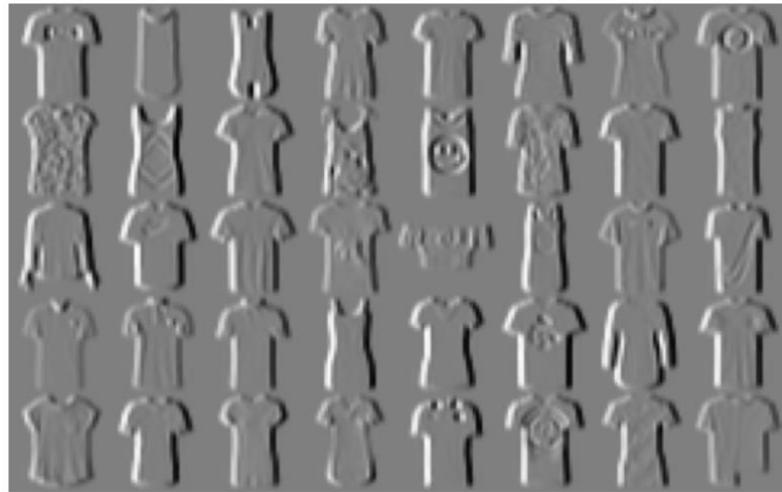
- Now move back to cell [2] and choose the CIFAR10 i.e., multi-channel images. Execute the convolutions and make sure that you understand the application of the 3-channel kernel on the CIFAR10 image as represented in Figure 164 (right). The result for the Laplace kernel should look like in Figure 166.

Note also the artefacts that appear due to the zero padding of the function

```
torchvision.utils.make_grid(..)
```

used to create the tile-images:





**Figure 165:** Edge-detection filter as an example for a convolution on a single channel image (FashionMNIST).



**Figure 166:** Laplace-filter as an example for a convolution on a multi-channel image (CIFAR10).

Equipped with this knowledge on convolutions we can now introduce the general architecture of CNNs.

### 7.3 General Architecture of CNNs

The key elements of the CNN are the convolutional and the pooling layer, which were inspired by the study on the visual cortex of cats [25]. In fact, the main building block of CNNs are – possibly several – convolutional layers (including a non-linear activation function) followed by a pooling layer. This block is then repeated several times until the spatial resolution is small enough to flatten the result to a single vector and to apply a standard MLP with dense layers as classifier.

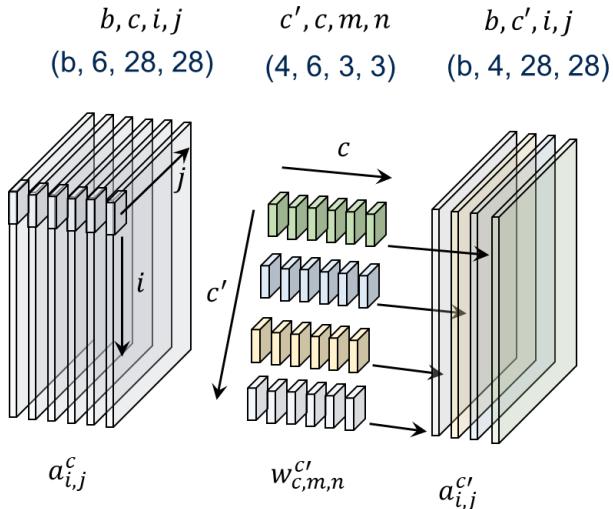
We will start with the convolutional layer and treat the pooling layer in the next section 7.3.2.

#### 7.3.1 Convolutional Layer

The illustration of the convolution in Figure 162 (c.f. also Figure 174) shows that each output neuron – here positioned at  $a'_{i,j}$  – has only a limited receptive field determined by the size of the kernel  $w_{m,n}$ . In addition, the same kernel is used at all positions or in other words, it is shared between all output neurons. Therefore, it requires much less parameters as compared to a dense layer of comparable size. Furthermore, the features – e.g. as shown in Figure 165 or Figure 166 – are detected everywhere in the image. Thus, the detection is intrinsically translation invariant. Therefore, convolutions allow to solve the two problems – stated at the end of section 7.1 – related to object classification using MLP with dense layers. However, it is immediately clear that only very simple features – like edges in Figure

165 – could be determined with a kernel of limited receptive field. This is where the pooling layer comes into play to reduce iteratively the size of the image which – in turn – increases the complexity of the detect features. This will be discussed in detail in the next section 7.3.2.

So far, we only applied a single kernel to the input images. However, a single kernel allows to detect only one typical feature type – e.g. as shown in Figure 165 or Figure 166 – but for the classification of a large number of object categories we will certainly require many different features. Therefore, we will extend the single kernel case shown in Figure 164 to multiple kernels. This is illustrated in the following Figure 167.



**Figure 167:** Extension of the convolution to a set of 4 kernels  $c' = 4$ .

As shown the input to the convolution operation is a multi-channel image  $a_{i,j}^c$  with a total of  $c = 6$ . We consider images of size  $28 \times 28 (i \times j)$ . The convolution now comprises a total of  $c' = 4$  kernels and we have to add an additional index to the kernel elements  $w_{c,m,n}^{c'}$ . Application of each of the  $c' = 4$  individual kernels will give as output a single channel image with the same output size (we assume padding). Thus, the output “image” – or activation map –  $a_{i,j}^{c'}$  will have a total of  $c' = 4$  channels.

The formula for the convolution now reads<sup>136</sup>:

$$a_{i,j}^{c'} = \sum_{c,m,n} a_{i+m,j+n}^c \cdot w_{c,m,n}^{c'} + b^{c'} \quad \forall c', i, j$$

**Equation 13**

Note that we added the bias  $b^{c'}$ , which – for each value of  $c'$  – is a single scalar value. Note further, that the convolution is applied to a batch consisting of  $b$  images. In Figure 168 we extended the above illustration to represent the case of  $b = 5$ .

So far, we limited the convolution operation to shift the kernel in each step – row or column – by one pixel. This is the so-called stride. It is also possible to extend the convolution operation to larger strides. Obviously, this will reduce the output size of the image. In Figure 169 this is illustrated for the case of a  $10 \times 10$  input image and a  $3 \times 3$  kernel size. If a “standard” stride  $S = 1$  (top) is applied, the kernel be applied to all pixels of the input image  $a_{i,j}$  and the output  $a'_{i,j}$  will have the same size (i.e.  $10 \times 10$ ). If we apply a stride of  $S = 2$  (bottom) the kernel will only be applied to every other pixel (grey) row- and column-wise. Therefore, the output will have a reduced size of  $5 \times 5$  for the illustrated numbers.

In general, the output size can be evaluated depending on the different parameters according to the following formulas:

$$i' = \frac{i - m + 2 \cdot p_i}{s_i} + 1 \qquad \qquad j' = \frac{j - n + 2 \cdot p_j}{s_j} + 1$$

<sup>136</sup> Please note that we changed our notation with respect to the discussion of the dense layer in section 4.5.2. E.g. we do not indicate the layer index to keep things as simple as possible.

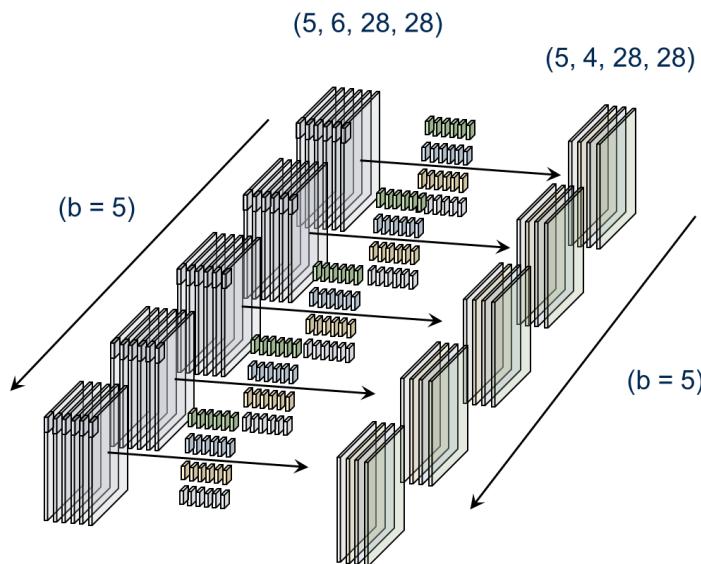
We apply integer division i.e., rounding to the lower value. Furthermore, the meaning of the variables are as follows:

- $i, i', j, j'$ : number of rows and columns of in- and output images, respectively<sup>137</sup>.
- $m, n$ : number of rows and columns of the kernel.
- $p_i, p_j$ : number of pixels padded in row and column direction, respectively.
- $S_i, S_j$ : stride in row and column direction, respectively.

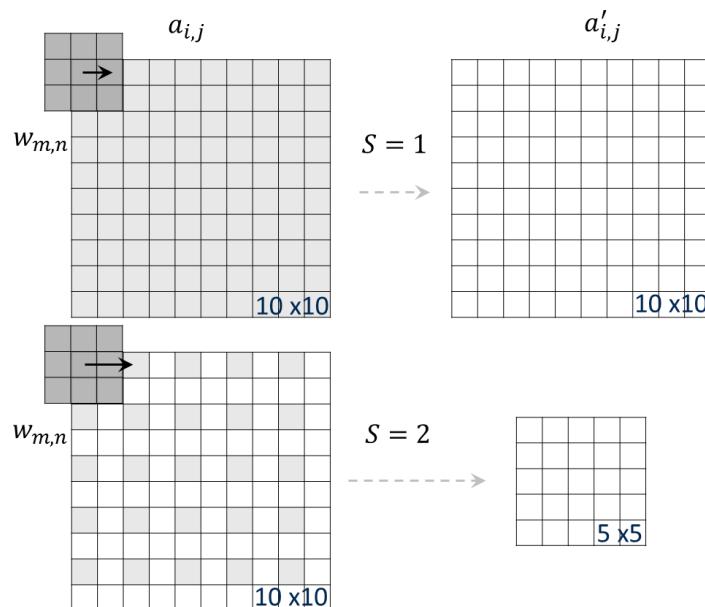
For the example given in Figure 169 application of the formula (row and column direction is identical) for the  $S = 2$  case leads to:

$$i' = \frac{10 - 3 + 2 \cdot 1}{2} + 1 = 5$$

This is as expected from the figure.



**Figure 168:** Application of a convolution (input channels  $c = 6$ , output channels  $c' = 4$ ) to a batch of  $b = 5$  images.



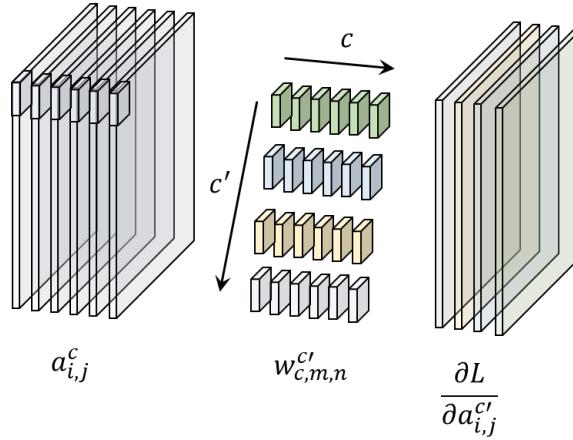
**Figure 169:** Application of different strides ( $S = 1$  top,  $S = 2$  bottom) leads to different output sizes (padding assumed).

<sup>137</sup> Note that for reason of simplicity we do not distinguish the index (e.g.  $i$ ) from the upper bound.

### 7.3.1.1 Backpropagation through convolutional Layer

While the derivation of the formulas for the backpropagation through a convolutional layer are straightforward, the handling of the indices is somewhat tedious. The results are nevertheless very instructive because they allow a very simple interpretation. In addition, we will create a link between the convolutional and the dense layer and will see, that the backpropagation formulas are consistent.

For the following, we consider the activation function to be an own layer so that we only determine the derivatives of the Loss  $L$  after  $b^{c'}$ ,  $w_{c,m,n}^{c'}$  and  $a_{i,j}^c$ . As usual, we assume that the derivatives of the Loss  $L$  with respect to the output activations  $a_{i,j}^{c'}$  are known. We illustrate the situation always using the same set of parameters as used in Figure 167 i.e., with  $c = 6$  input and  $c' = 4$  output channels (Figure 170).



**Figure 170:** Situation at the start of the backpropagation of the convolutional layer.  $\frac{\partial L}{\partial a_{i,j}^{c'}}$  are given and we require the derivatives after of the Loss  $L$  after  $b^{c'}$ ,  $w_{c,m,n}^{c'}$  and  $a_{i,j}^c$ .

For the following, we use the convention that the indices of the variable by which differentiation is made (e.g.  $b^{\tilde{c}'}$  in the next sections) are given a tilde to distinguish them from the indices of the variables that are differentiated.

#### 7.3.1.1.1 Determination of $\frac{\partial L}{\partial b^{\tilde{c}'}}$

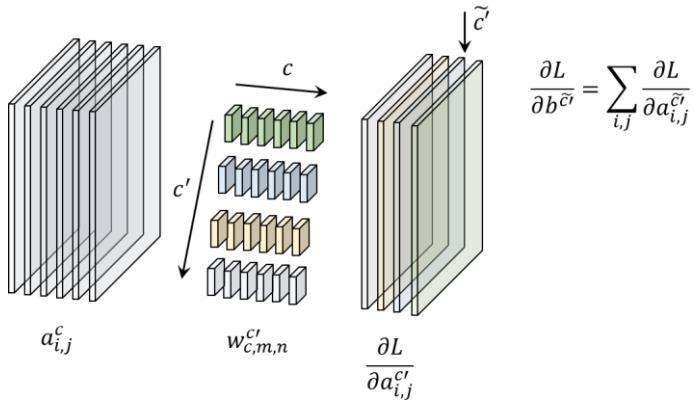
$$\begin{aligned} \frac{\partial L}{\partial b^{\tilde{c}'}} &=^1 \frac{\partial L(\dots, a_{i,j}^{c'}, \dots)}{\partial b^{\tilde{c}'}} =^2 \sum_{c',i,j} \frac{\partial L}{\partial a_{i,j}^{c'}} \cdot \frac{\partial a_{i,j}^{c'}}{\partial b^{\tilde{c}'}} =^3 \sum_{c',i,j} \frac{\partial L}{\partial a_{i,j}^{c'}} \cdot \underbrace{\frac{\partial}{\partial b^{\tilde{c}'}} \left[ \sum_{c,m,n} a_{i+m,j+n}^c \cdot w_{c,m,n}^{c'} + b^{c'} \right]}_{\delta_{c',c'}^{\tilde{c}'}} =^4 \\ &= \sum_{c',i,j} \frac{\partial L}{\partial a_{i,j}^{c'}} \cdot \delta_{c',c'}^{\tilde{c}'} = \sum_{i,j} \frac{\partial L}{\partial a_{i,j}^{\tilde{c}'}} \end{aligned}$$

$$\frac{\partial L}{\partial b^{\tilde{c}'}} = \sum_{i,j} \frac{\partial L}{\partial a_{i,j}^{\tilde{c}'}}$$

At the equal sign  $=^1$  we made the dependency of the loss function upon the output activations  $a_{i,j}^{c'}$  explicit. At  $=^2$  we expand  $\frac{\partial L}{\partial b^{\tilde{c}'}}$  into a sum over terms  $\frac{\partial L}{\partial a_{i,j}^{c'}}$ . At  $=^3$  we used the Equation 13. At  $=^4$  we used that fact that only  $b^{c'}$  depends on  $b^{\tilde{c}'}$  and that the result is equal to zero unless  $\tilde{c}' = c'$ .

Finally, we obtain that the derivative  $\frac{\partial L}{\partial b^{\tilde{c}'}}$  is just the sum  $\sum_{i,j} \frac{\partial L}{\partial a_{i,j}^{\tilde{c}'}}$  for the respective output index  $\tilde{c}'$ . In addition, the mean over the batch has to be performed<sup>138</sup> (Figure 171).

<sup>138</sup> We will not develop these formulas here explicitly as the extension is obvious and will not lead – in contrary to the dense layer case in section 4.5.2 – to a more compact formulation.



**Figure 171:** The derivative  $\frac{\partial L}{\partial b^{\tilde{c}'}}$  is just the sum over  $\sum_{i,j} \frac{\partial L}{\partial a_{i,j}^{\tilde{c}'}}$  for the respective output index  $c'$ . In addition, the mean over the batch has to be done.

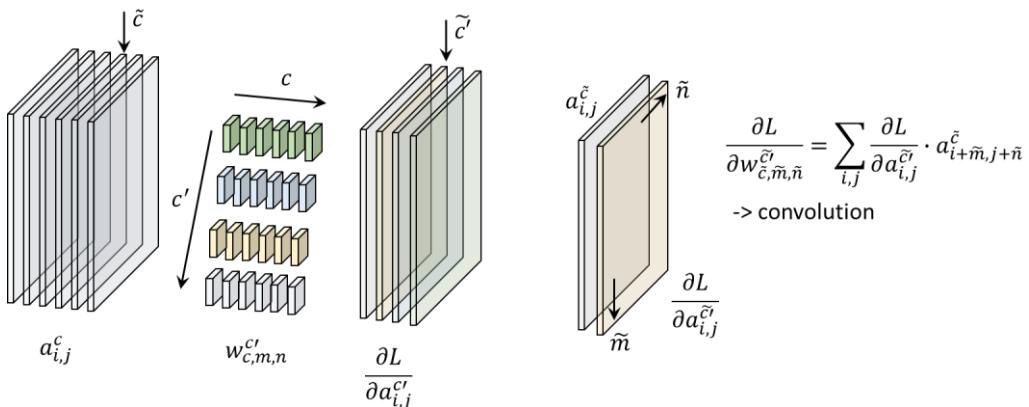
#### 7.3.1.1.2 Determination of $\frac{\partial L}{\partial w_{\tilde{c},\tilde{m},\tilde{n}}^{\tilde{c}'}}$

The derivation follows closely the previous section with the determination of  $\frac{\partial L}{\partial b^{\tilde{c}'}}$ :

$$\begin{aligned} \frac{\partial L}{\partial w_{\tilde{c},\tilde{m},\tilde{n}}^{\tilde{c}'}} &= \sum_{c',i,j} \frac{\partial L}{\partial a_{i,j}^{c'}} \cdot \frac{\partial a_{i,j}^{c'}}{\partial w_{\tilde{c},\tilde{m},\tilde{n}}^{\tilde{c}'}} = \sum_{c',i,j} \frac{\partial L}{\partial a_{i,j}^{c'}} \cdot \frac{\partial}{\partial w_{\tilde{c},\tilde{m},\tilde{n}}^{\tilde{c}'}} \left[ \sum_{c,m,n} a_{i+m,j+n}^c \cdot w_{c,m,n}^{c'} + b^{c'} \right] =^{1)} \\ &= \sum_{c',i,j} \frac{\partial L}{\partial a_{i,j}^{c'}} \cdot \sum_{c,m,n} a_{i+m,j+n}^c \cdot \delta_{\tilde{c},c'} \cdot \delta_{\tilde{m},m} \cdot \delta_{\tilde{n},n} = \sum_{i,j} \frac{\partial L}{\partial a_{i,j}^{\tilde{c}'}} \cdot a_{i+\tilde{m},j+\tilde{n}}^{\tilde{c}} \\ \boxed{\frac{\partial L}{\partial w_{\tilde{c},\tilde{m},\tilde{n}}^{\tilde{c}'}} = \sum_{i,j} \frac{\partial L}{\partial a_{i,j}^{\tilde{c}'}} \cdot a_{i+\tilde{m},j+\tilde{n}}^{\tilde{c}}} \end{aligned}$$

At the equal sign =<sup>1)</sup> we used the fact that the derivative with respect to  $w_{\tilde{c},\tilde{m},\tilde{n}}^{\tilde{c}'}$  is zero unless all corresponding indices are equal.

Interestingly, the result is just the convolution of the derivative of the output activations  $\frac{\partial L}{\partial a_{i,j}^{\tilde{c}'}}$  (output channel  $\tilde{c}'$ ) with the input channel  $a_{i,j}^{\tilde{c}}$  (input channel  $\tilde{c}$ ). In addition, the mean over the batch has to be done.



**Figure 172:** The derivative  $\frac{\partial L}{\partial w_{\tilde{c},\tilde{m},\tilde{n}}^{\tilde{c}'}}$  is just the convolution of the derivative of the output activation  $\frac{\partial L}{\partial a_{i,j}^{\tilde{c}'}}$  (output channel  $\tilde{c}'$ ) with the input channel  $a_{i,j}^{\tilde{c}}$  (input channel  $\tilde{c}$ ). In addition, the mean over the batch has to be done.

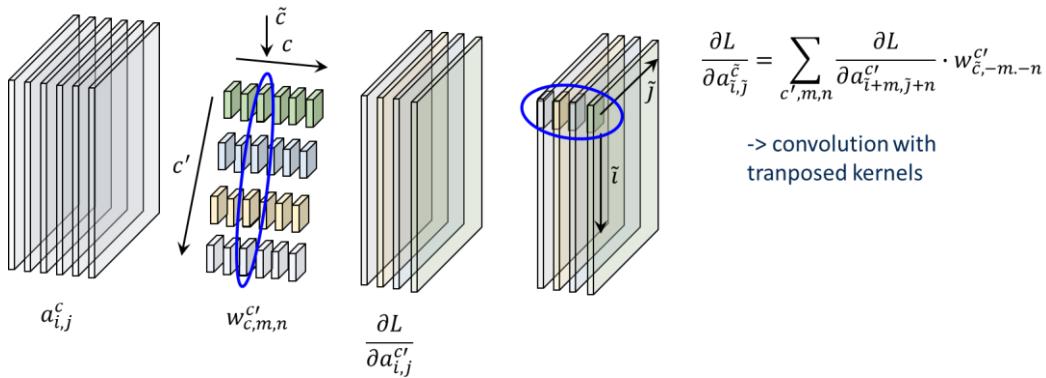
### 7.3.1.1.3 Determination of $\frac{\partial L}{\partial a_{i,j}^{\tilde{c}}}$

The derivation follows closely the previous sections:

$$\begin{aligned}\frac{\partial L}{\partial a_{i,j}^{\tilde{c}}} &= \sum_{c',i,j} \frac{\partial L}{\partial a_{i,j}^{c'}} \cdot \frac{\partial a_{i,j}^{c'}}{\partial a_{i,j}^{\tilde{c}}} = \sum_{c',i,j} \frac{\partial L}{\partial a_{i,j}^{c'}} \cdot \frac{\partial}{\partial a_{i,j}^{\tilde{c}}} \left[ \sum_{c,m,n} a_{i+m,j+n}^c \cdot w_{c,m,n}^{c'} + b^{c'} \right] = \\ \sum_{c',i,j} \frac{\partial L}{\partial a_{i,j}^{c'}} \cdot \sum_{c,m,n} \delta_{\tilde{c},c} \cdot \delta_{i+m,j+n} \cdot w_{c,m,n}^{c'} &= \sum_{c',m,n} \frac{\partial L}{\partial a_{i-m,j-n}^{c'}} \cdot w_{\tilde{c},m,n}^{c'} = \sum_{c',m,n} \frac{\partial L}{\partial a_{i+m,j+n}^{c'}} \cdot w_{\tilde{c},-m,-n}^{c'}\end{aligned}$$

$$\frac{\partial L}{\partial a_{i,j}^{\tilde{c}}} = \sum_{c',m,n} \frac{\partial L}{\partial a_{i+m,j+n}^{c'}} \cdot w_{\tilde{c},-m,-n}^{c'}$$

As for the case of  $\frac{\partial L}{\partial w_{\tilde{c},\tilde{m},\tilde{n}}^{c'}}$  we find a very intuitive result. The derivative  $\frac{\partial L}{\partial a_{i,j}^{\tilde{c}}}$  is just the convolution of the derivative of the output activation  $\frac{\partial L}{\partial a_{i,j}^{c'}}$  with the – transposed<sup>139</sup> – kernel values  $w_{\tilde{c},-m,-n}^{c'}$  corresponding to the input channel index  $\tilde{c}$  (blue ellipses). In addition, the mean over the batch has to be done.



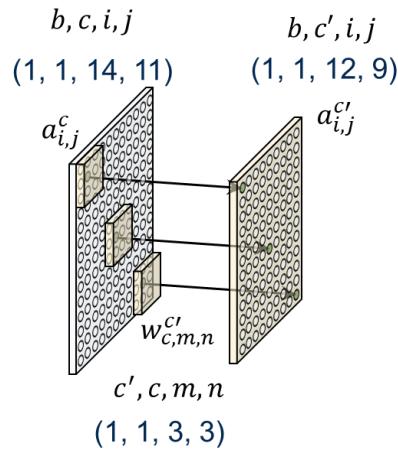
**Figure 173:** The derivative  $\frac{\partial L}{\partial a_{i,j}^{\tilde{c}}}$  is just the convolution of the derivatives of the output activations  $\frac{\partial L}{\partial a_{i,j}^{c'}}$  with the – transposed<sup>139</sup> – kernel values  $w_{\tilde{c},-m,-n}^{c'}$  corresponding to the input channel index  $\tilde{c}$  (blue ellipses). In addition, the mean over the batch has to be done.

### 7.3.1.2 Relation to dense Layer

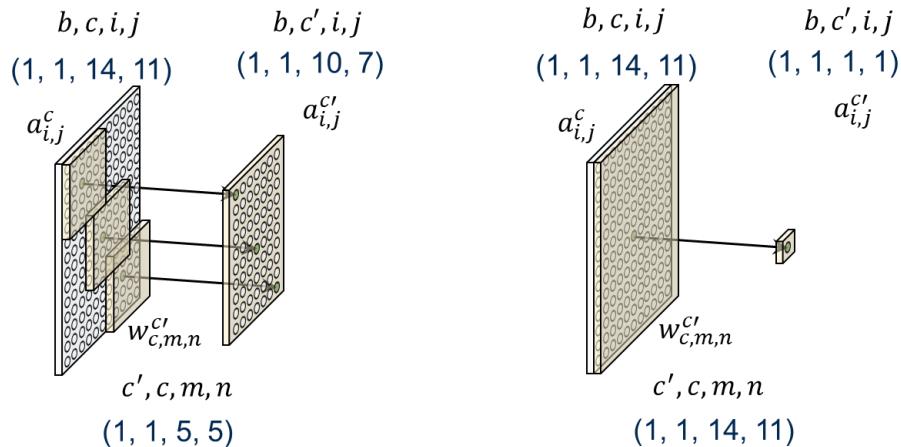
It is instructive to consider the relationship between the convolutional layer and the dense layer. Therefore, we consider a simple convolutional layer with only one input and output channel as represented in Figure 174. The kernel has the size  $3 \times 3$  and we do not apply padding. In addition, we made the input and output neurons – i.e. the pixel – explicit. The input channel has  $14 \times 11$  pixel (neurons) and – due to the absence of padding – the output channel only  $12 \times 9$ . Three corresponding in- and output neurons are highlighted with the corresponding kernel position. We see on the one hand the limited receptive field of the output neurons within the input image and that on the other that the spatial relationship between the in- and output neurons is preserved.

We now increase the kernel size to – first –  $7 \times 7$  and then to  $14 \times 11$  i.e. the size of the input channel (left and right in Figure 175 respectively). For kernel size  $7 \times 7$  the output channel is already reduced to a size of  $10 \times 9$ . If we finally increase the kernel size to  $14 \times 11$  the output channel will consist of only one single pixel (neuron) and – in addition – the spatial information will be completely lost because this output neurons receives the information of all input pixel.

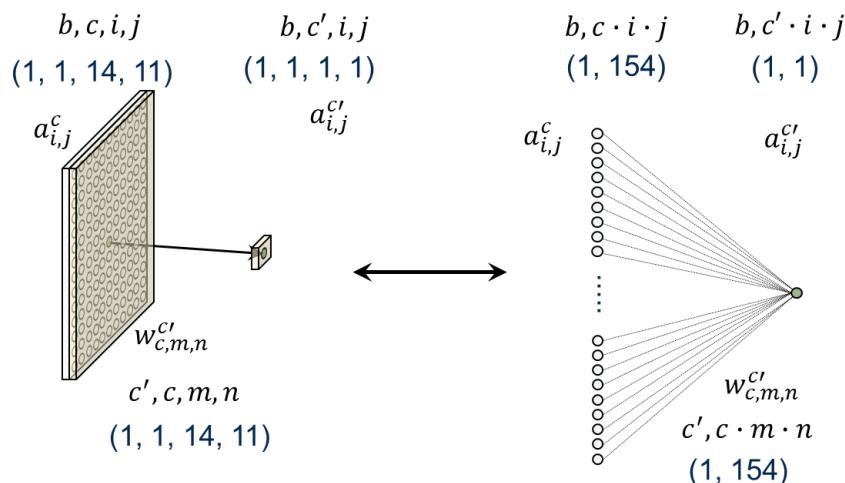
<sup>139</sup> This is expressed by the minus sign in front of the row and column indices i.e.  $-m, -n$ .



**Figure 174:** A simple convolutional layer with one input and output channel (details c.f. text).



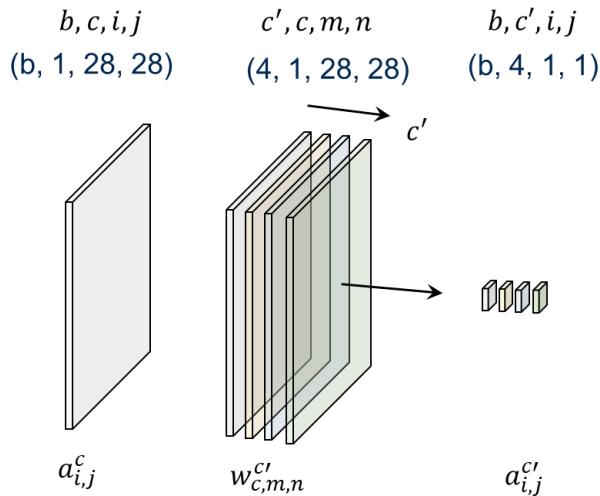
**Figure 175:** Increasing the size of the kernel will – without padding – decrease the size of the output activation (details c.f. text).



**Figure 176:** A «convolutional layer» with kernel size corresponding to the input image size (left) is equivalent to a dense layer with a single output neuron.

Because the spatial information is lost, there is no further need to represent the input pixel as a 2D-

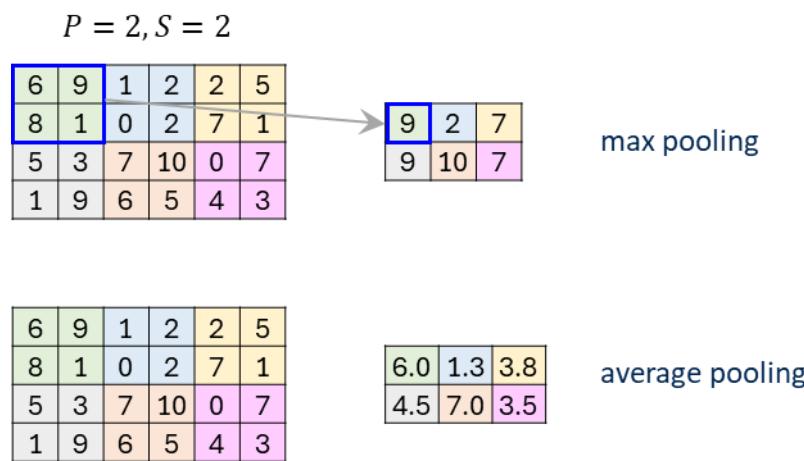
image array, but we can flatten them to a single input vector (Figure 176, right). This will give a total of  $1 \times 14 \times 11 = 154$  input pixel (neurons) with a single output neuron. Thus, we see that a convolutional layer with a kernel size equal to the input image – and without padding – can be considered as a dense layer with one output neuron (c.f. Figure 49). Additional output neurons would then correspond to additional kernels of the same size i.e.  $1 \times 154$ . Thus, e.g. application of a dense layer with four output neurons on a (Fashion)MNIST image represented as convolutional layer would look like illustrated in the following Figure 177. It is now also possible to identify the formulas for the backpropagation through the convolutional layer from section 7.3.1.1 with the corresponding formulas from Equation 9. We leave this as exercise for the reader.



**Figure 177:** Dense layer with four output neurons represented as convolutional layer.

### 7.3.2 Pooling Layer

In the previous section we illustrated (e.g. Figure 174) how the convolutional layer preserves the spatial information form the input image. However, the limited receptive field would allow to detect only relatively low-level features like edges, corners, etc. This is where the pooling layer comes into play to reduce iteratively the size of the image which – in turn – increases the complexity of the detect features. The idea is conceptually simple and is illustrated in the following Figure 178.



**Figure 178:** Pooling applied to a  $4 \times 6$ -pixel images patch. Max pooling top, average pooling bottom (average values rounded to 2 significant digits).

The choice of parameters represented is the most frequent one: a pooling size of  $P = 2$  with a stride of

$S = 2$ . This means that the pooling is performed over each sub-patch of size  $2 \times 2^{140}$  and the patch is always shifted by 2 pixel in row and column direction. For the pooling the most frequent option is max pooling, where the maximum value of the activations in the sub-patch is chosen (top in Figure 178). In addition, average pooling is a frequent option, where the average value of the activations in the patch is calculated (bottom in Figure 178).

The idea of max pooling is that the most prominent activation in the pooling region is selected and then passed to the next layer, which – as we will see in the next section – is usually a convolutional layer. Due to the image decimation the receptive field of the convolutional layer will now cover a larger part (twice as much for the settings in Figure 178) of the original image and can therefore detect features of higher complexity. This alternating sequence of convolutional and pooling layers is the main architectural paradigm of CNNs. It is continued to a point, where the features complexity is high enough such that a “standard” MLP classifier can be used for the object classification.

### 7.3.3 Overall CNN Architecture

As already mentioned above the main characteristics of CNN is the alternating sequence of convolutional and pooling layer followed – after a flattening – by a set of dense layers for the actual classification. In the following we will implement the architecture represented in Figure 179. The representation is for (Fashion)MNIST images i.e., input image size  $1 \times 28 \times 28$  but can be easily adjusted for CIFAR10 images with size  $3 \times 32 \times 32$ .

The first layer is a convolutional layer with 16 kernels of size  $5 \times 5$  applied at stride  $S = 1$ . We will always apply padding to preserve the original image size. The result will be an activation map with 16 channels i.e.,  $16 \times 28 \times 28$ . Then we apply a max pooling layer of pooling size  $P = 2$  with stride  $S = 2$ . We obtain output activations of size  $16 \times 14 \times 14$ . Now we apply a second convolutional layer with 32 kernels of size  $3 \times 3$  to obtain an output of size  $32 \times 14 \times 14$ , again followed by a max pooling layer ( $P = 2, S = 2$ ) to obtain the intermediate result of size  $32 \times 7 \times 7$ .

This is the point where we consider the features to be of sufficient complexity to apply a standard MLP classifier. Therefore, the output is flattened – to a vector of  $32 \times 7 \times 7 = 1568$  element – which is the input to the first dense layer with 200 neurons followed by the final softmax output with the 10 classes. We will see below that both PyTorch and TensorFlow offer convenient functionality to implement and parametrise these layers in a few lines of code.

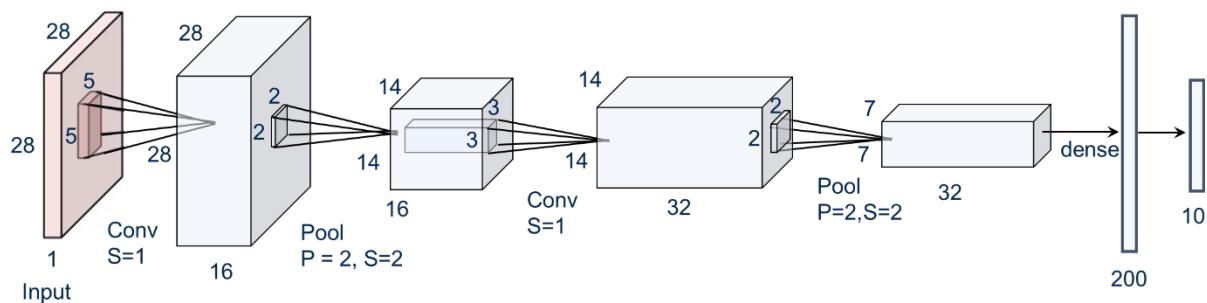


Figure 179: CNN-architecture that will be implemented in PyTorch and TensorFlow below (details c.f. text).

Several important improvements and/or extensions to this standard architecture have been proposed since the publication of Alexnet [5]<sup>141</sup>. Here we will only mention VGG16 [28] because its architecture extends the above scheme with a very common idea consisting in a stack of several consecutive convolutional layers before applying a pooling layer.

The following Figure 180 gives an overview of the VGG16 architecture, which consists of stacks of 2 or 3 consecutive convolutional layers followed by a pooling layer. The detailed layer sizes are summarised in Table 7. There, in addition the number of parameters per layer – only weights no biases – are given. Observe the ratio of parameters in the convolutional with respect to the dense layers. This is a consequence of the weight sharing of the neurons in the convolutional layer (c.f. Figure 174).

<sup>140</sup> Obviously anisotropic kernel size are also an option.

<sup>141</sup> See the book of Géron, chapter 13 for a summary of important CNN architectures [15].

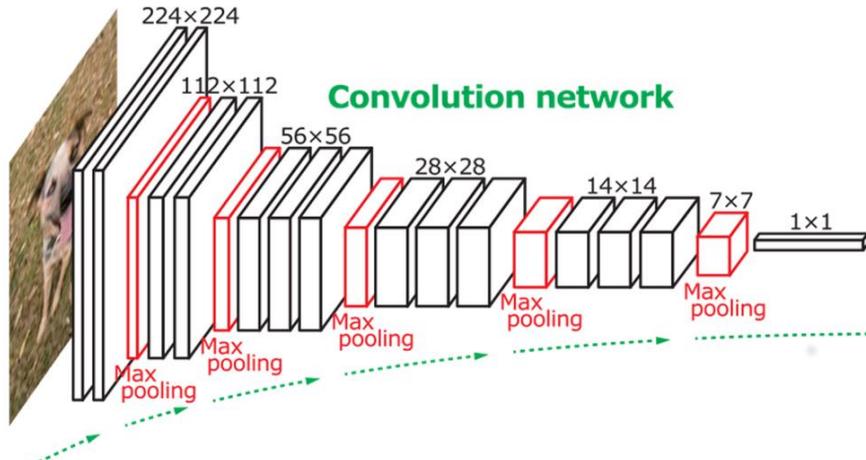


Figure 180: Architecture of the VGG16 CNN [28].

|                          |                                                            |
|--------------------------|------------------------------------------------------------|
| INPUT: [224x224x3]       | weights: 0                                                 |
| CONV3-64: [224x224x64]   | weights: $(3 \times 3 \times 3) \times 64 = 1,728$         |
| CONV3-64: [224x224x64]   | weights: $(3 \times 3 \times 64) \times 64 = 36,864$       |
| POOL2: [112x112x64]      | weights: 0                                                 |
| CONV3-128: [112x112x128] | weights: $(3 \times 3 \times 64) \times 128 = 73,7$        |
| CONV3-128: [112x112x128] | weights: $(3 \times 3 \times 128) \times 128 = 147,456$    |
| POOL2: [56x56x128]       | weights: 0                                                 |
| CONV3-256: [56x56x256]   | weights: $(3 \times 3 \times 128) \times 256 = 294,912$    |
| CONV3-256: [56x56x256]   | weights: $(3 \times 3 \times 256) \times 256 = 589,824$    |
| CONV3-256: [56x56x256]   | weights: $(3 \times 3 \times 256) \times 256 = 589,824$    |
| POOL2: [28x28x256]       | weights: 0                                                 |
| CONV3-512: [28x28x512]   | weights: $(3 \times 3 \times 256) \times 512 = 1,179,648$  |
| CONV3-512: [28x28x512]   | weights: $(3 \times 3 \times 512) \times 512 = 2,359,296$  |
| CONV3-512: [28x28x512]   | weights: $(3 \times 3 \times 512) \times 512 = 2,359,296$  |
| POOL2: [14x14x512]       | weights: 0                                                 |
| CONV3-512: [14x14x512]   | weights: $(3 \times 3 \times 512) \times 512 = 2,359,296$  |
| CONV3-512: [14x14x512]   | weights: $(3 \times 3 \times 512) \times 512 = 2,359,296$  |
| CONV3-512: [14x14x512]   | weights: $(3 \times 3 \times 512) \times 512 = 2,359,296$  |
| POOL2: [7x7x512]         | weights: 0                                                 |
| FC: [1x1x4096]           | weights: $7 \times 7 \times 512 \times 4096 = 102,760,448$ |
| FC: [1x1x4096]           | weights: $4096 \times 4096 = 16,777,216$                   |
| FC: [1x1x1000]           | weights: $4096 \times 1000 = 4,096,000$                    |

Table 7: Layer sizes for the VGG16 CNN architecture and corresponding number of parameters (only weights no biases).

#### Exercise:

We use the iPython notebook `7.2.cnn_pytorch_stud.ipynb` to implement the CNN architecture of Figure 179 in PyTorch:

- Quickly review the structure of the notebook:  
You will realize that it is mainly a copy of the notebook `7.0.mlp_pytorch_cifar_stud.ipynb`. One difference is that cell [2] allows to select the dataset to be either (Fashion)MNIST (1) or CIFAR10 (3):

```
num_images_planes = 1
```

This is to change the CNN input size from one to three channels. Start with single channel input.

- Execute the cells [1] to [5] (the latter for TensorBoard output<sup>142</sup>) and move to cell [6] ("Class

<sup>142</sup> Be sure to use the correct path when starting TensorBoard (c.f. section 6.2.3).

ConvNeuralNetwork"). In the constructor of the class you are supposed to add the required code lines in order to implement the CNN network from Figure 179. Look for the section:

```
### START YOUR CODE ###
```

```
### END YOUR CODE ###
```

The way to proceed will be similar to the exercise in section 6.2.1 by using the sequential container of PyTorch. Only the layer types will be different. In particular, the following layer types will be of interest (refer to the PyTorch documentation on the layer types for further details<sup>114</sup>):

- o `torch.nn.Conv2d(..)`

You have to specify the parameters for the

- `in_channels`: number of channels of input (`size_in`) or previous layer
- `out_channels`: number of kernels in layer e.g. 16 for the first conv layer
- `kernel_size`: e.g. 5 (corresponding to 5 x 5) for the first conv layer

In addition make sure to set padding correctly, to preserve the size:

- `padding='same'`

If we use padding we should use a padding option with a minimum of artefacts at the borders e.g. reflect:

```
padding_mode='reflect'
```

- o `torch.nn.ReLU()`

We already know the ReLU activation function from the exercise

```
7.0.mlp_pytorch_cifar_stud.ipynb.
```

- o `torch.nn.MaxPool2d(..)`

You have to specify the pooling size:

- `kernel_size`: We always choose a value of  $P = 2$

The stride is per default identical to the kernel\_size so can leave the default ("None").

- o `torch.nn.flatten(..)`

We already know the flatten layer and its parameters from the exercise

```
7.0.mlp_pytorch_cifar_stud.ipynb.
```

In addition you will have to implement the dense layers – including their activations – and the final Softmax layer, but both are already known from the exercise

```
7.0.mlp_pytorch_cifar_stud.ipynb.
```

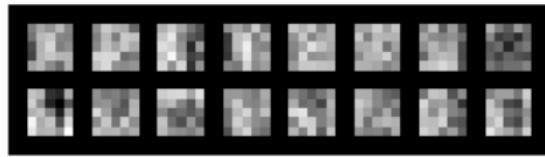
- Implement the full CNN and initialise it using cell [7]. Execute cell [8] to visualise the parameters of the different layers. Observe the ratio of parameters between the convolutional and dense layers and review the Table 7 with the VGG16 architecture and parameter counts.
- Now trigger the training of the CNN using the cell [9] and observe the curves on TensorBoard.
- Once the training finished move to cells [10] and below. These serve to visualize the weights and activations of the CNN.
  - o Cell [10] plots the weights of the first convolutional layer, with access given by:

```
CNN.model[0].weight
```

Make sure to understand the above syntax. The weights – shown as tile image – may look as follows<sup>143</sup>:

---

<sup>143</sup> Note that the weights are trained and therefore will differ each time you run the algorithm.

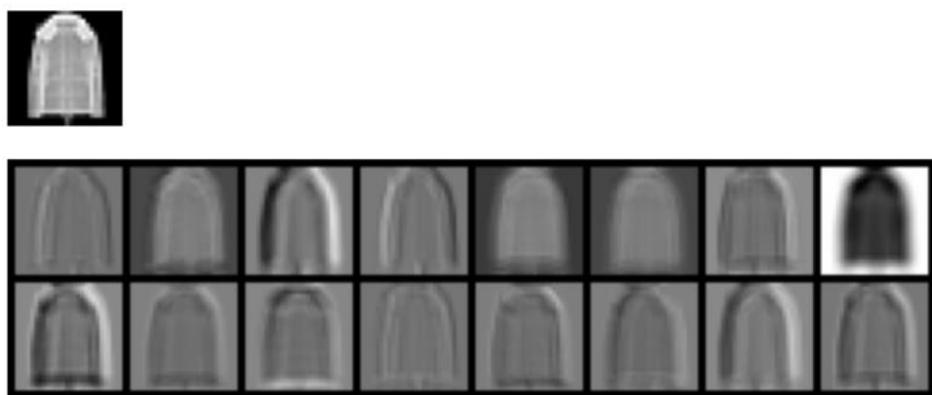


- Cell [11] just sets up the DataLoader to access images for the forward pass in order to visualise the activation maps. This is done through the following calls (cell [12]):

```
activ=CNN.model[0](img):
```

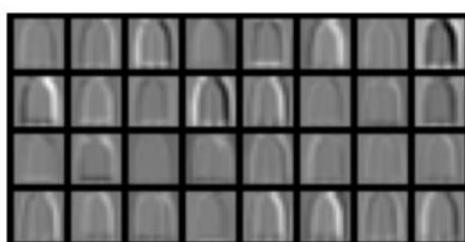
Application of the first conv layer to the image. The output are the 16 activation maps from the first layer (without activation function), with size [1, 16, 28, 28]. Make sure to understand the implementation and verify the output size.

An example for the original image and the activation maps are given below<sup>144</sup>:



```
for i0 in range(1,4):
    activ=CNN.model[i0](activ):
```

This loop applies successively layers 1 to 3 (activation function of 1<sup>st</sup> conv layer, 1<sup>st</sup> pooling layer, 2<sup>nd</sup> conv layer) and will therefore give the activation maps of the 2<sup>nd</sup> conv layer. Verify that their size is [1, 32, 14, 14]. Make sure to understand how the size is determined by the different layers. Below these activation maps are plotted for the input image shown above:

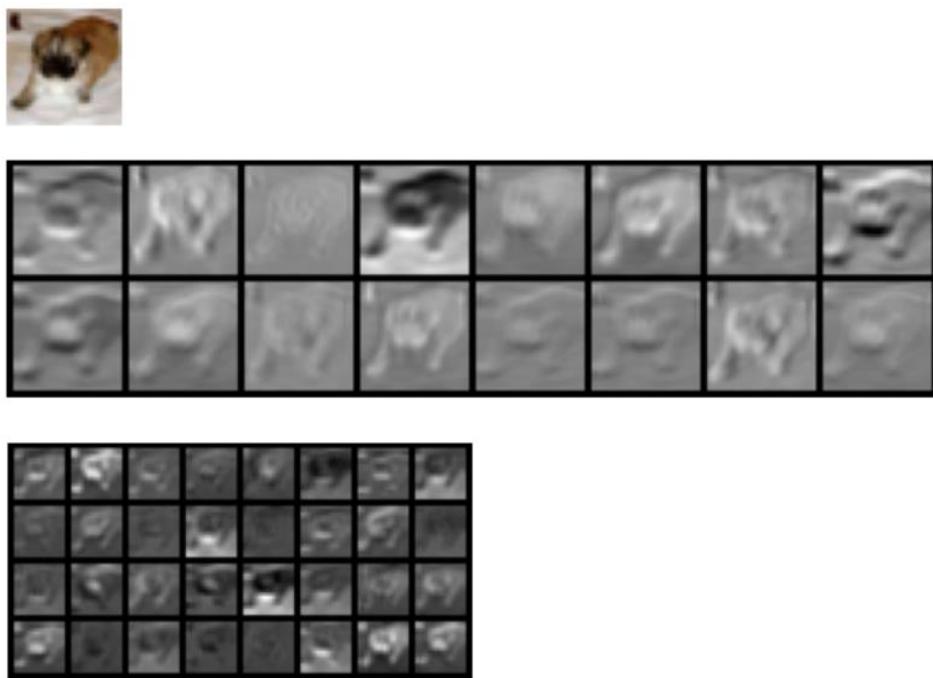


- Finally move back to cell [2] and choose the CIFAR10 images with three input channels. Adapt – if not already working – the CNN architecture and repeat the points from above. The corresponding images could look as follows:

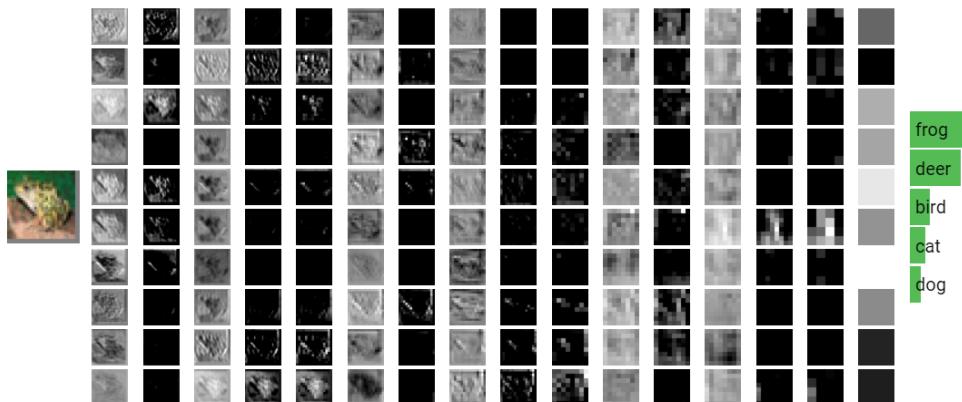



---

<sup>144</sup> Each call to the DataLoader iterator will give a different input image.



For comparison you may also have a look at the following webpage, where the activation maps are shown for a VGG16 architecture online in your Browser<sup>145</sup>:



**Figure 181:** Online determination of activation maps for a CNN architecture in the Browser<sup>145</sup>.

We will finally repeat the implementation of the CNN architecture of Figure 179 using the Keras API of TensorFlow.

Exercise:

We use the iPython notebook `7.3.cnn_tensorflow_stud.ipynb` to implement the CNN architecture of Figure 179 in TensorFlow:

- Quickly review the structure of the notebook. Its structure is very similar to the notebook `6.5.mlp_tensorflow_stud.ipynb`. One difference is that cell [2] allows to select the dataset to be either (Fashion)MNIST (1) or CIFAR10 (3):

```
num_images_planes = 1
```

<sup>145</sup> <https://cs231n.stanford.edu/>

This is to change the CNN input size from one to three channels. Start with single channel input.

- When executing cell [2] for the data selection observe the size of the data which is (for FashionMNIST)

(60000, 28, 28, 1)

In contrary to PyTorch, where the channel index is at the second position, TensorFlow allows to use "channels\_first" (<-> PyTorch) or "channels\_last" (as shown above) option. Default setting of TensorFlow is to use channels\_last<sup>146</sup>, which has the advantage that it is compatible with the representation of images in "classic" image processing. We will therefore keep this convention.

- Execute the cells [1] to [6] and move to cell [7] ("Class ConvNeuralNetwork"). In the constructor of the class, you are again supposed to add the required code lines in order to implement the CNN network from Figure 179. Look for the section:

```
### START YOUR CODE ###  
### END YOUR CODE ###
```

The syntax is similar to PyTorch. For details either refer to the TensorFlow documentation<sup>129</sup> or use the document "Keras\_Cheat\_Sheet\_gssmi8.pdf" on Moodle.

- Implement the CNN and run the training via cell [8]. Observe the output of

```
self.model.summary()
```

on the console, which summarises the CNN architecture including the number of parameters. The learning curves are visible on TensorBoard.

- Finally you may move back to cell [2] and choose the CIFAR10 images with three input channels. Adapt – if not already working – the CNN architecture and repeat the points from above.

---

<sup>146</sup> This difference between PyTorch and TensorFlow makes conversion of model architectures between the two frameworks sometimes tedious.

## 8 Annex

### 8.1 Backpropagation for Batch Normalization Layer

In the following we will derive the formulas for the backpropagation in case a batch normalization layer is used. In chapter 5.1.5 we introduced vanilla batch normalization with application to the logits. Nevertheless, here we will consider the batch normalization as an independent layer applied to the activations of the previous layer. The reason is twofold. First, we can derive the backpropagation formulas with focus on the batch normalization only<sup>147</sup>, which makes the approach and results clearer. Second, many researchers argue that it is just as good or even better to place Batch Normalization after the activation layer.

Two points are relevant for the derivation of the backpropagation formulas of a batch normalization layer:

- All calculation during the normalization process involves only values – for the full batch – of one single feature i.e., no “mixture” of different features occurs. Thus, we can simply derive the formulas for one single feature and then extend to the full feature set by elementwise calculations.
- In contrast to chapter 4.5, where we could first focus on a single sample (section 4.5.3) we must directly tackle the case for a full batch, as done in chapter 4.5.4.

We consider the layer to index  $l$  with the matrix of activations  $\mathbf{A}_{\{r\}}^{[l]}$  being the input to the batch normalization layer function. As in chapter 5.1.5 the additional index  $\{r\}$  indicates the index of the current batch of the training samples. The matrix  $\mathbf{A}_{\{r\}}^{[l]}$  consists of column vectors  $\mathbf{a}_{\{r\}}^{[l](i)}$  ( $i = 1..m$ ) where  $m$  denotes the size of the mini-batch. Each vector  $\mathbf{a}_{\{r\}}^{[l](i)}$  has  $n_l$  entries corresponding to the number of output neurons of layer  $l$ .

$$\mathbf{A}_{\{r\}}^{[l]} = \begin{pmatrix} \vdots & \vdots & \vdots \\ \mathbf{a}_{\{r\}}^{[l](1)} & \mathbf{a}_{\{r\}}^{[l](2)} & \dots & \mathbf{a}_{\{r\}}^{[l](m)} \\ \vdots & \vdots & & \vdots \end{pmatrix}$$

We recall the formulas for batch normalization, which require the average  $\mu_{\{r\}}^{[l]}$  and standard deviation  $\sigma_{\{r\}}^{[l]}$  over the  $m$  column vectors  $\mathbf{a}_{\{r\}}^{[l](i)}$  of the mini-batch according to:

$$\mu_{\{r\}}^{[l]} = \frac{1}{m} \sum_{i=1}^m \mathbf{a}_{\{r\}}^{[l](i)}$$

$$\sigma_{\{r\}}^{[l]} = \sqrt{\frac{1}{m} \sum_{i=1}^m (\mathbf{a}_{\{r\}}^{[l](i)} - \mu_{\{r\}}^{[l]})^2}$$

Now, the actual normalization of the activation matrix is as follows:

$$\hat{\mathbf{A}}_{\{r\}}^{[l]} = \frac{\mathbf{A}_{\{r\}}^{[l]} - \mu_{\{r\}}^{[l]}}{\sigma_{\{r\}}^{[l]} + \epsilon}$$

As already mentioned above all calculations are considered to be elementwise.

Finally, the values of  $\hat{\mathbf{A}}_{\{r\}}^{[l]}$  are rescaled according to,

$$\tilde{\mathbf{A}}_{\{r\}}^{[l]} = \gamma^{[l]} \cdot \hat{\mathbf{A}}_{\{r\}}^{[l]} + \beta^{[l]}$$

---

<sup>147</sup> From the considerations below it is nevertheless straight forward to derive the formulas for vanilla batch normalization.

where the scaling parameters  $\gamma^{[l]}$  and  $\beta^{[l]}$  are optimized during the training process. The values  $\tilde{A}_{\{r\}}^{[l]}$  are the output of the batch normalization layer.

For the derivation of the backpropagation formulas, we assume that the derivatives of the loss with respect to these outputs

$$\frac{\partial L}{\partial \tilde{A}^{[l]}}$$

are given and we require the following quantities:

- The derivatives with respect to the scaling parameters  $\gamma^{[l]}$  and  $\beta^{[l]}$ :

$$\frac{\partial L}{\partial \gamma^{[l]}}, \frac{\partial L}{\partial \beta^{[l]}}$$

- The derivatives with respect to the output of the previous layer i.e., the activations  $A_{\{r\}}^{[l]}$ :

$$\frac{\partial L}{\partial A^{[l]}}$$

For the following we will simplify the notation to keep thing clear:

We will now omit the indices  $\{r\}$  for the batch and  $[l]$  for the layer and only keep the index over the different samples in the batch ( $i$ ). In addition, we will suppress the vector notation (bold face), because – as already mentioned above – we can focus on a single feature. Then e.g., the input activations to our batch normalization layer will read  $a^{(i)}$  instead of  $a_{\{r\}}^{[l](i)}$ .

### 8.1.1 Formulas for a single feature

As mentioned, we will first derive the formulas for a single feature and then extend the notation to the full feature vectors.

#### 8.1.1.1 Derivatives with respect to the scaling parameter $\gamma^{[l]}$ and $\beta^{[l]}$

The derivatives with respect to the scaling parameter  $\gamma^{[l]}$  and  $\beta^{[l]}$  are straight forward to obtain. We only have to keep in mind that these values are always averages over the mini-batch<sup>148</sup>.

$$\begin{aligned}\frac{\partial L}{\partial \gamma} &= \frac{1}{m} \sum_{i=1}^m \frac{\partial L}{\partial \tilde{a}^{(i)}} \cdot \frac{\partial \tilde{a}^{(i)}}{\partial \gamma} = \frac{1}{m} \sum_{i=1}^m \frac{\partial L}{\partial \tilde{a}^{(i)}} \cdot \hat{a}^{(i)} \\ \frac{\partial L}{\partial \beta} &= \frac{1}{m} \sum_{i=1}^m \frac{\partial L}{\partial \tilde{a}^{(i)}} \cdot \frac{\partial \tilde{a}^{(i)}}{\partial \beta} = \frac{1}{m} \sum_{i=1}^m \frac{\partial L}{\partial \tilde{a}^{(i)}}\end{aligned}$$

Equation 14

#### 8.1.1.2 Derivatives with respect to the activations $\frac{\partial L}{\partial A^{[l]}}$ of the previous layer

The derivatives with respect to the activations of the previous layer are somewhat more tedious, because the average  $\mu_{\{r\}}^{[l]}$  and standard deviation  $\sigma_{\{r\}}^{[l]}$  of the mini-batch depend upon  $A^{[l]}$ . Nevertheless, the derivatives  $\frac{\partial L}{\partial A^{[l]}}$  can be obtained in a straightforward manner by applying the chain rule.

We will use as intermediate result for the following the identity:

$$\frac{\partial L}{\partial \hat{a}^{(i)}} = \sum_{k=1}^m \frac{\partial L}{\partial \tilde{a}^{(k)}} \cdot \frac{\partial \tilde{a}^{(k)}}{\partial \hat{a}^{(i)}} = \sum_{k=1}^m \frac{\partial L}{\partial \tilde{a}^{(k)}} \cdot \gamma \cdot \delta_{ik} = \frac{\partial L}{\partial \tilde{a}^{(i)}} \cdot \gamma$$

<sup>148</sup> Please note that  $\gamma$  (as well as  $\beta, \mu, \sigma$ ) do not require any index, because the indices  $\{r\}$  for the batch and  $[l]$  for the layer are omitted and  $\gamma$  is a property for all samples (index  $(i)$ ) of the batch.

Now we start with the following identity,

$$\frac{\partial L}{\partial a^{(i)}} = \sum_{k=1}^m \frac{\partial L}{\partial \hat{a}^{(k)}} \cdot \frac{\partial \hat{a}^{(k)}}{\partial a^{(i)}}$$

which – together with the previous formula for  $\frac{\partial L}{\partial \hat{a}^{(i)}}$  – “reduces” the problem in finding the derivatives  $\frac{\partial \hat{a}^{(k)}}{\partial a^{(i)}}$ :

$$\frac{\partial \hat{a}^{(k)}}{\partial a^{(i)}} = \frac{\partial}{\partial a^{(i)}} \left( \frac{a^{(k)} - \mu}{\sqrt{\sigma^2 + \epsilon}} \right)$$

Please note that for reasons of convenience we wrote the term  $(\sigma + \epsilon)$  as  $\sqrt{\sigma^2 + \epsilon}$ . While this is not correct from a rigorous mathematical point of view we can note that  $\epsilon$  is only a term to ensure numerical stability in case  $\sigma = 0$  and in all other cases we can consider  $\epsilon$  to be equal to zero.

Furthermore, recall that – making the dependencies of  $\mu$  and  $\sigma$  on the  $a^{(i)}$  explicit – in our notation we have:

$$\begin{aligned} \mu &= \frac{1}{m} \sum_{i=1}^m a^{(i)} \\ \sigma^2 &= \frac{1}{m} \sum_{i=1}^m (a^{(i)} - \mu)^2 \\ \hat{a}^{(i)} &= \frac{a^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}} \end{aligned}$$

Thus, we can continue the development of  $\frac{\partial \hat{a}^{(k)}}{\partial a^{(i)}}$ :

$$\frac{\partial \hat{a}^{(k)}}{\partial a^{(i)}} = \frac{\partial}{\partial a^{(i)}} \left( \frac{a^{(k)} - \mu}{\sqrt{\sigma^2 + \epsilon}} \right) = \frac{1}{\sqrt{\sigma^2 + \epsilon}} \cdot \underbrace{\frac{\partial}{\partial a^{(i)}} (a^{(k)} - \mu)}_{(1)} + (a^{(k)} - \mu) \cdot \underbrace{\frac{\partial}{\partial a^{(i)}} \left( \frac{1}{\sqrt{\sigma^2 + \epsilon}} \right)}_{(2)}$$

We now determine the two parts (1) and (2) separately:

(1):

$$\frac{\partial}{\partial a^{(i)}} (a^{(k)} - \mu) = \left( \delta_{ik} - \frac{\partial \mu}{\partial a^{(i)}} \right) =^{(*)} \left( \delta_{ik} - \frac{1}{m} \right)$$

The identity at  $=^{(*)}$  is a consequence of the fact that  $\mu$  depends linearly (factor  $\frac{1}{m}$ ) upon all  $a^{(k)}$ .

(2):

$$\frac{\partial}{\partial a^{(i)}} \left( \frac{1}{\sqrt{\sigma^2 + \epsilon}} \right) = -\frac{1}{2} \frac{1}{(\sqrt{\sigma^2 + \epsilon})^3} \cdot \frac{\partial}{\partial a^{(i)}} \sigma^2$$

And furthermore:

$$\frac{\partial}{\partial a^{(i)}} \sigma^2 = \frac{\partial}{\partial a^{(i)}} \left( \frac{1}{m} \sum_{k=1}^m (a^{(k)} - \mu)^2 \right) =^{(*)} \frac{\partial}{\partial a^{(i)}} \left( \frac{1}{m} \sum_{k=1}^m (a^{(k)})^2 - \mu^2 \right) = \frac{2}{m} \cdot (a^{(i)} - \mu)$$

Here, the identity at  $=^{(*)}$  is well known from statistics.

Now we put everything together to obtain in a first step  $\frac{\partial \hat{a}^{(k)}}{\partial a^{(i)}}$ :

$$\frac{\partial \hat{a}^{(k)}}{\partial a^{(i)}} = \frac{\partial}{\partial a^{(i)}} \left( \frac{a^{(k)} - \mu}{\sqrt{\sigma^2 + \epsilon}} \right) = \frac{1}{\sqrt{\sigma^2 + \epsilon}} \cdot \frac{\partial}{\partial a^{(i)}} (a^{(k)} - \mu) + (a^{(k)} - \mu) \cdot \frac{\partial}{\partial a^{(i)}} \left( \frac{1}{\sqrt{\sigma^2 + \epsilon}} \right) =$$

$$\begin{aligned}
 &= \frac{1}{\sqrt{\sigma^2 + \epsilon}} \cdot \left( \delta_{ik} - \frac{1}{m} \right) + (a^{(k)} - \mu) \cdot \frac{(-1)}{2} \frac{1}{(\sqrt{\sigma^2 + \epsilon})^3} \cdot \frac{2}{m} \cdot (a^{(i)} - \mu) = \\
 &= \frac{1}{\sqrt{\sigma^2 + \epsilon}} \cdot \left( \delta_{ik} - \frac{1}{m} - \frac{1}{m} \cdot \frac{(a^{(k)} - \mu)}{\sqrt{\sigma^2 + \epsilon}} \cdot \frac{(a^{(i)} - \mu)}{\sqrt{\sigma^2 + \epsilon}} \right) = \\
 &= \frac{1}{\sqrt{\sigma^2 + \epsilon}} \cdot \left( \delta_{ik} - \frac{1}{m} - \frac{1}{m} \cdot \hat{a}^{(k)} \cdot \hat{a}^{(i)} \right)
 \end{aligned}$$

Thus, despite the lengthy derivation the final formula remains rather simple:

$$\begin{aligned}
 \frac{\partial L}{\partial a^{(i)}} &= \sum_{k=1}^m \frac{\partial L}{\partial \hat{a}^{(k)}} \cdot \frac{\partial \hat{a}^{(k)}}{\partial a^{(i)}} = \sum_{k=1}^m \frac{\partial L}{\partial \hat{a}^{(k)}} \cdot \frac{1}{\sqrt{\sigma^2 + \epsilon}} \cdot \left( \delta_{ik} - \frac{1}{m} - \frac{1}{m} \cdot \hat{a}^{(k)} \cdot \hat{a}^{(i)} \right) \\
 \frac{\partial L}{\partial a^{(i)}} &= \frac{1}{\sqrt{\sigma^2 + \epsilon}} \cdot \left( \frac{\partial L}{\partial \hat{a}^{(i)}} - \frac{1}{m} \cdot \sum_{k=1}^m \frac{\partial L}{\partial \hat{a}^{(k)}} - \hat{a}^{(i)} \cdot \frac{1}{m} \cdot \sum_{k=1}^m \frac{\partial L}{\partial \hat{a}^{(k)}} \cdot \hat{a}^{(k)} \right)
 \end{aligned}$$

With the additional identity,

$$\frac{\partial L}{\partial \hat{a}^{(i)}} = \frac{\partial L}{\partial \tilde{a}^{(i)}} \cdot \gamma$$

and the formulas from Equation 14 we can write the result in a quite compact way:

$$\begin{aligned}
 \frac{\partial L}{\partial a^{(i)}} &= \frac{1}{\sqrt{\sigma^2 + \epsilon}} \cdot \left( \frac{\partial L}{\partial \hat{a}^{(i)}} - \frac{1}{m} \cdot \sum_{k=1}^m \frac{\partial L}{\partial \hat{a}^{(k)}} - \hat{a}^{(i)} \cdot \frac{1}{m} \cdot \sum_{k=1}^m \frac{\partial L}{\partial \hat{a}^{(k)}} \cdot \hat{a}^{(k)} \right) = \\
 &\quad \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}} \cdot \left( \frac{\partial L}{\partial \tilde{a}^{(i)}} - \frac{1}{m} \cdot \sum_{k=1}^m \frac{\partial L}{\partial \tilde{a}^{(k)}} - \hat{a}^{(i)} \cdot \frac{1}{m} \cdot \sum_{k=1}^m \frac{\partial L}{\partial \tilde{a}^{(k)}} \cdot \hat{a}^{(k)} \right)
 \end{aligned}$$

And we finally obtain:

$$\frac{\partial L}{\partial a^{(i)}} = \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}} \cdot \left( \frac{\partial L}{\partial \tilde{a}^{(i)}} - \frac{\partial L}{\partial \beta} - \hat{a}^{(i)} \cdot \frac{\partial L}{\partial \gamma} \right)$$

### 8.1.2 Formulas for entire feature vector

Extension of the formulas for the entire feature vector are now straight forward, as all feature components are independent. Thus, the formals from the previous chapter can be simply extended:

$$\begin{aligned}
 \frac{\partial L}{\partial \mathbf{y}^{[l]}} &= \frac{1}{m} \cdot \left( \frac{\partial L}{\partial \tilde{\mathbf{A}}_{\{r\}}^{[l]}} * \widehat{\mathbf{A}}_{\{r\}}^{[l]} \right) \cdot \begin{pmatrix} \vdots \\ \mathbf{1} \\ \vdots \end{pmatrix} \\
 \frac{\partial L}{\partial \boldsymbol{\beta}^{[l]}} &= \frac{1}{m} \cdot \frac{\partial L}{\partial \tilde{\mathbf{A}}_{\{r\}}^{[l]}} \cdot \begin{pmatrix} \vdots \\ \mathbf{1} \\ \vdots \end{pmatrix} \\
 \frac{\partial L}{\partial \mathbf{A}_{\{r\}}^{[l]}} &= \frac{1}{\sigma_{\{r\}}^{[l]} + \epsilon} * \mathbf{y}^{[l]} * \left( \frac{\partial L}{\partial \tilde{\mathbf{A}}_{\{r\}}^{[l]}} - \frac{\partial L}{\partial \boldsymbol{\beta}^{[l]}} - \widehat{\mathbf{A}}_{\{r\}}^{[l]} * \frac{\partial L}{\partial \mathbf{y}^{[l]}} \right)
 \end{aligned}$$

Equation 15

Please note the following:

- The vectors  $\begin{pmatrix} \vdots \\ \mathbf{1} \\ \vdots \end{pmatrix}$ , which is multiplied from the right in the formulas for  $\frac{\partial L}{\partial \mathbf{y}^{[l]}}$  and  $\frac{\partial L}{\partial \boldsymbol{\beta}^{[l]}}$  are just an

alternative way to express the summation over the columns of the matrices  $\hat{\mathbf{A}}_{\{r\}}^{[l]}$ ,  $\frac{\partial L}{\partial \hat{\mathbf{A}}_{\{r\}}^{[l]}}$ , etc. of size  $n^l \times m$ .

- All multiplications and the division by  $\sigma_{\{r\}}^{[l]}$  are component wise.
- $\sigma_{\{r\}}^{[l]}$  and  $\gamma^{[l]}$  being row vectors of dimension  $n^l \times 1$ , their elementwise product will also give a vector of the same size. Therefore, the product sign after  $\gamma^{[l]}$  is a broadcast over the full matrices  $\frac{\partial L}{\partial \hat{\mathbf{A}}_{\{r\}}^{[l]}}$  etc. in the parenthesis, which are of size  $n^l \times m$ .

The same applies for  $\frac{\partial L}{\partial \gamma^{[l]}}$  and  $\frac{\partial L}{\partial \beta^{[l]}}$ .

## 8.2 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<sup>149</sup>
- 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 3.5.1. Here we want to deepen our insight on this issue.

### 8.2.1 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:

$$P(C_5) = \frac{1}{10} ; P(C_{\neg 5}) = \frac{1}{10}$$

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 16

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.

<sup>149</sup> 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>

### 8.2.2 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 182). Both trucks transport oranges (orange circle) and apples (green circle). We take randomly a fruit out of the box and observe the type  $x$  being orange  $\text{o}$  or apple  $\text{a}$  and want to decide whether we are in a 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  $p(x|C_{red})$  according to Bayes rule:

$$p(\text{o}|C_{red}) = \frac{6}{8} \quad p(\text{a}|C_{red}) = \frac{2}{8}$$

$$p(\text{o}|C_{blue}) = \frac{3}{8} \quad p(\text{a}|C_{blue}) = \frac{5}{8}$$

Thus,  $p(\text{o}|C_{red})$  represents the probability of finding an orange in case we are in a red truck and  $p(\text{a}|C_{red}) = 1 - p(\text{o}|C_{red})$  is the probability of finding an apple.

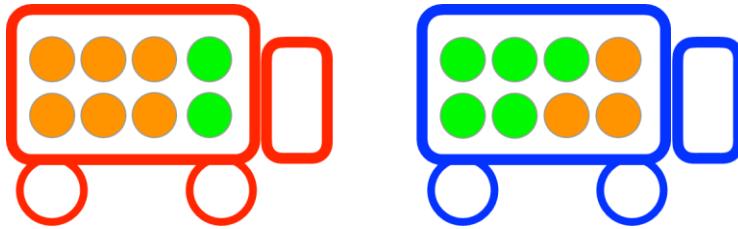


Figure 182: Example of food trucks for application of Bayes rule.

If we want to decide whether we are in a red or blue truck having chosen some fruit  $x$  and observed its type, we must – according to Bayes rule (Equation 16) – maximize the a posteriori probability of the observation. Assume that  $x$  is an orange  $\text{o}$ . Bayes rule tells us:

$$p(C_{red}|\text{o}) = \frac{p(\text{o}|C_{red}) \cdot p(C_{red})}{p(\text{o})}$$

The identical formula is true for the blue truck. It states that the a posteriori probability  $p(C_{red}|\text{o})$  of being in a red truck having observed an orange is given by the product of the likelihood  $p(\text{o}|C_{blue})$  and the a priori probability  $p(C_{red})$  of red cars. The division by the evidence of finding an orange  $p(\text{o})$ , which does depend on the class, will not change the maximization:

$$\begin{aligned} p(C_{red}|\text{o}) &\leq p(C_{blue}|\text{o}) \Leftrightarrow \frac{p(\text{o}|C_{red}) \cdot p(C_{red})}{p(\text{o})} \leq \frac{p(\text{o}|C_{blue}) \cdot p(C_{blue})}{p(\text{o})} \\ &\Leftrightarrow p(\text{o}|C_{red}) \cdot p(C_{red}) \leq p(\text{o}|C_{blue}) \cdot p(C_{blue}) \end{aligned}$$

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:

$$\begin{aligned} p(C_{red}) = p(C_{blue}) &\Rightarrow p(\text{o}|C_{red}) \cdot p(C_{red}) > p(\text{o}|C_{blue}) \cdot p(C_{blue}) \\ &\Rightarrow p(C_{red}|\text{o}) > p(C_{blue}|\text{o}) \end{aligned}$$

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:

$$\begin{aligned} p(C_{red}) = \frac{1}{5} < p(C_{blue}) = \frac{4}{5} &\Rightarrow p(\text{o}|C_{red}) \cdot p(C_{red}) = \frac{6}{8} \cdot \frac{1}{5} < = \frac{3}{8} \cdot \frac{4}{5} = p(\text{o}|C_{blue}) \cdot p(C_{blue}) \\ &\Rightarrow p(C_{red}|\text{o}) < p(C_{blue}|\text{o}) \end{aligned}$$

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.

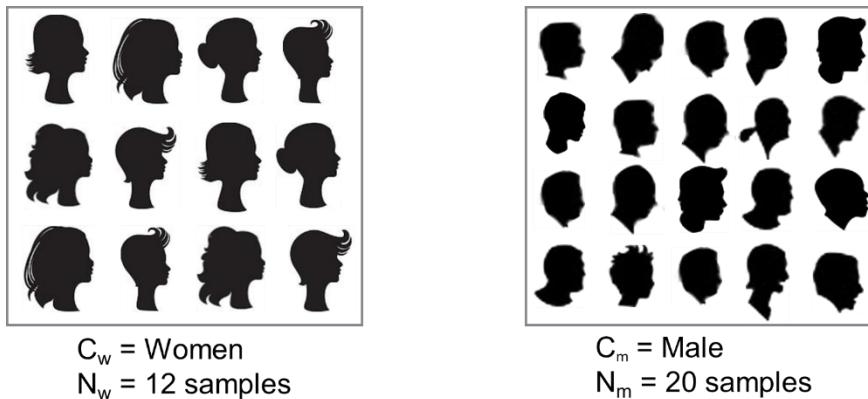
### 8.2.3 Example 2 – Continuous Observations

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 16.

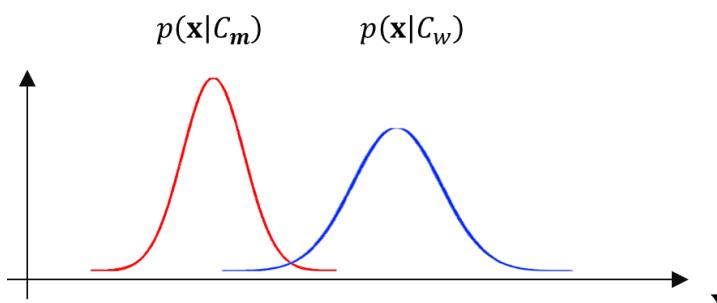
We observe the length of hair of a set of women  $C_w$  and men  $C_m$  (c.f. Figure 183). Our sample set comprises  $N_w = 12$  women and  $N_m = 20$  men. Thus, we can determine the a priori probability of the two classes:

$$P(C_w) = \frac{12}{32} = \frac{3}{8} ; P(C_m) = \frac{5}{8}$$

The likelihood in Equation 16 is usually obtained by modelling the probability for observing the feature  $x$  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 184). We are still missing the evidence  $p(x)$  to apply Equation 16. However, because the evidence does not depend on the classes it will not influence the maximization of the right-hand-side of Equation 16 and can therefore be ignored.



**Figure 183:** A set of 12 women and 20 men are used to decide among these two classes based on the hair length.



**Figure 184:** The likelihood could be obtained by modelling the average hair length of men ( $C_m$ ) and women ( $C_w$ ).

### 8.2.4 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 ( $-T$ ) 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 8):

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 ( $C_h$ )

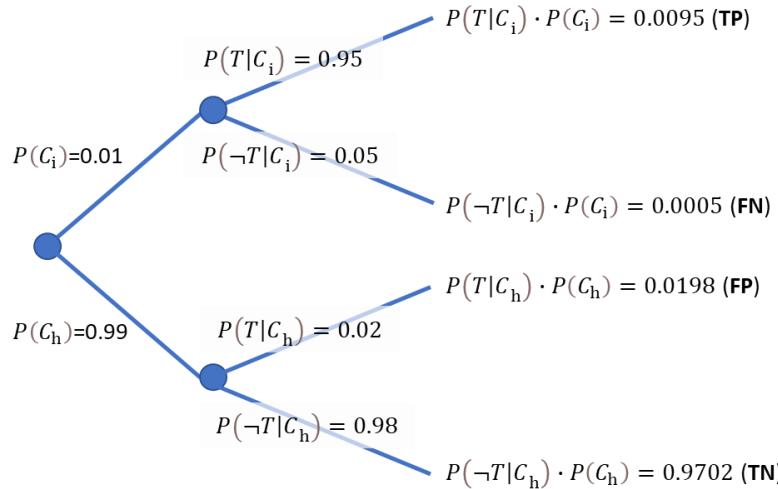
3. False Negative (FN):

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<sup>150</sup>:



**Figure 185:** 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<sup>151</sup>. Furthermore, we assume the following likelihood probabilities (c.f. Equation 16)<sup>152</sup>:

- 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 8 denoting the TP rate, FP rate, etc. can be determined and the full confusion table be evaluated (Table 8):

<sup>150</sup> It is convenient to represent the different cases in form of a decision tree.

<sup>151</sup> Typically, this is much worse in medical tests as diseases might be much less frequent then 1%.

<sup>152</sup> These could be determined based on a test series with healthy and ill persons.

|       | $T$    | $\neg T$ |      |
|-------|--------|----------|------|
| $C_i$ | 0.0095 | 0.0005   | 0.01 |
| $C_h$ | 0.0198 | 0.9702   | 0.99 |
|       | 0.0293 | 0.9707   | 1    |

**Table 8:** 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:

$$P(C_i|T) =^{(1)} \frac{P(T|C_i) \cdot P(C_i)}{p(T)} =^{(2)} \frac{P(T|C_i) \cdot P(C_i)}{P(T|C_i) \cdot P(C_i) + P(T|C_h) \cdot P(C_h)} = \frac{0.0095}{0.0095 + 0.0198} = 0.32$$

At the equal sign  $=^{(1)}$  we applied Bayes rule (Equation 16) 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 185).

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:

$$\text{Recall: } P(T|C_1) = 0.95$$

$$\text{Precision: } P(C_1|T) = 0.32$$

$$\text{Accuracy: } 0.9797$$

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 185 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.

### 8.3 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.

### 8.4 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] Suwananakorn, 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] Vaswani, Ashish, et al. "Attention is all you need." *Advances in neural information processing systems* 30 (2017).
- [5] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." *Advances in neural information processing systems* 25 (2012).
- [6] <https://image-net.org/challenges/LSVRC/index.php>
- [7] LeCun, Yann, et al. "Backpropagation applied to handwritten zip code recognition." Neural computation 1.4 (1989): 541-551.
- [8] K. Murphy, "Machine Learning – A Probabilistic Perspective", 2012.
- [9] T. Mitchell, "Machine Learning", 1997.
- [10] [DL-notations.pdf](#)
- [11] Xiao, Han, Kashif Rasul, and Roland Vollgraf. "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms." arXiv preprint arXiv:1708.07747 (2017).
- [12] Bishop, C. M. "Neural Networks for Pattern Recognition." Clarendon Press google schola 2 (1995): 223-228.
- [13] 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> )
- [14] K. Murphy, "Machine Learning – A Probabilistic Perspective", 2012.
- [15] A. Géron, "Hands-On Machine Learning with Scikit-Learn and TensorFlow", 2017.
- [16] 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> )
- [17] 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> )
- [18] 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> )
- [19] 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>)
- [20] 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> )
- [21] 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> )
- [22] 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> )
- [23] Silver, David, et al. "Mastering the game of go without human knowledge." nature 550.7676 (2017): 354-359.
- [24] Callaway, Ewen. "The entire protein universe': AI predicts shape of nearly every known protein." Nature 608.7921 (2022): 15-16.
- [25] Hubel, David H., and Torsten N. Wiesel. "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex." *The Journal of physiology* 160.1 (1962): 106.
- [26] Fukushima, Kunihiko. "Neocognitron: A hierarchical neural network capable of visual pattern recognition." *Neural networks* 1.2 (1988): 119-130.
- [27] LeCun, Yann, et al. "Gradient-based learning applied to document recognition." *Proceedings of the IEEE* 86.11 (1998): 2278-2324.
- [28] Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." *arXiv preprint arXiv:1409.1556* (2014).

## 8.5 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 |
| ILSVR       | ImageNet Large Scale Visual Recognition Challenge            |
| PR          | Precision-Recall                                             |
| ROC         | Receiver Operation Characteristics                           |
|             |                                                              |