Classification in Bone Scintigraphy Images Using Convolutional Neural Networks

Johnny Dang

Master's thesis
2016:E24

Lund University

Faculty of Engineering
Centre for Mathematical Sciences
Mathematics

Centrum Scientiarum Mathematicarum

Classification in Bone Scintigraphy Images Using Convolutional Neural Networks

Johnny Dang

June 2016

Master's Thesis
Faculty of Engineering
Centre for Mathematical Sciences
Mathematics

Supervisor 1: Karl Sjöstrand, Exini Diagnostics AB, Lund, Sweden
Supervisor 2: Niels Christian Overgaard, Mathematical Imaging Group, Lund University
Examiner: Kalle Åström, Mathematical Imaging Group, Lund University

Abstract

The main goal of this thesis is to design and implement a convolutional neural network (CNN) to classify whether hotspots in bone scintigraphy images represent cancer metastases, caused by prostate cancer, or some other physiological process. A side task was included, where another CNN was designed and trained to classify whether a bone scintigraphy image image is acquired from the front of the patient (anterior) or from the back (posterior). The CNNs were implemented using the deep learning frameworks Deeplearning4J and Keras, and were trained on labelled data sets provided by Exini Diagnostics AB, Lund, Sweden.

The performance of the CNNs were evaluated using various metrics commonly used in machine learning contexts, for instance accuracy and Receiver Operating Characteristics. The final trained CNN used for the classification of hotspots reached an accuracy of 89% on a test set consisting of images that had not been used for training. The corresponding CNN for the side task reached an accuracy of 99%. These results indicate that CNNs can work well for both classification problems.

Acknowledgements

I want to thank Exini Diagnostics AB, Lund, Sweden, for giving me this opportunity to do my masters thesis at Exini. I also thank my supervisors Karl Sjöstrand and Niels Christian Overgaard for all the help and advice they have given me throughout the thesis work. I am very thankful that I was introduced to the field of neural networks and deep learning by Karl Sjöstrand.

Contents

Chapter 1

Introduction

The ability for us to detect and recognize objects with such ease is a fantastic wonder that we seldom reflect over. When we see a dog, we can effortlessly recognize it as a dog instead of a bird, despite never having seen that dog before or even that breed. This ability to generalize from previous experience is something we take for granted, but it is rarely as simple as we think.

Our brain is a biological supercomputer that has been fine-tuned and trained for several million years. This has given the brain ample time to adapt to the signals we get from our senses. The information that is sent from our senses is processed in the brain by the almost hundred billions of neurons that are connected to each other through even more connections in tremendously complex ways. It is not that it is easy to recognize objects, rather, we humans are extraordinarily good at making sense of the information that we get from our senses. It is just that this is mostly done unconsciously and therefore we do not get to appreciate the skills that our brain is capable of, and that are incredibly hard for a computer to learn. Take for example the task of recognizing a car as part of an autonomous driving system. What computers see is just the numeric pixel values in images taken by a camera. Therefore when writing a program we can not use simple descriptions like "a metal box on four wheels" as that has no real meaning to a computer. Trying to make rules that precise the meaning of such simple descriptions will quickly lead to a mess of exceptions and special cases that would never lead anywhere.

Artificial neural networks (ANNs) represent a type of machine learning algorithm that attempts to mimic how the human brain processes information. Simple ANNs have been shown to be able to compute arithmetic and logical functions already in the late 1940's [34], which for a while drove research

to try to improve upon it. This resulted in several variants of the initial network, e.g. Hebbian learning [20] and the perceptron [39]. Researchers at that time believed that they were close to creating artificial intelligence and started to hype the potential of ANNs. As we can see today, the big promises were never fulfilled, and as new machine learning algorithms with seemingly better potential came up, the research on ANNs slowed down.

Even though ANNs were still being used and capable of acheiving state-of-the-art performance in several different areas, they were hard to train and the training usually took several weeks [49]. The breakthrough for ANNs came when new techniques for training networks were invented, coupled with increasing amounts of data and growing computer power made it possible to create deeper networks, also known as deep learning. Since its resurgence, deep learning has quickly become the state-of-the-art algorithm in many various fields, such as speech recognition [22], computer vision [4] and natural language processing [30]. Note that the terms ANN and neural network will be used interchangeably for variation in this report, as they are both common terms used throughout the machine learning community.

1.1 Medical Motivation

1.1.1 Prostate Cancer

Prostate cancer (PC) is the most common form of cancer in Sweden and constitutes a third of the cancer cases among men. In 2013, about 10,000 men were diagnosed with PC, which is an increase of almost 100% compared to 20 years ago [9]. This increase can partially be explained by the fact that screening using prostate-specific antigen, PSA, has become a lot more common.

PC may not always cause any serious harm for the patient. It usually grows slowly and is initially confined to he prostate gland, though there are more aggressive kinds that spread quickly. When diagnosed with PC, the best treatment may not always be to go through a surgical procedure, as the complications afterwards may overweight the benefits, especially for elder people. In some cases, it is better to use medical treatment and adopt a wait-and-see approach, as the tumour may not pose any significant threat [35]. Something that is always true is that the earlier the tumour is discovered, the better the chances for a cure to the disease or at least a symptom free

life where the cause of death is not related to PC [9].

1.1.2 Bone Metastases

When cancer cells from one or more primary tumours start to spread through the circulatory system, they may give rise to secondary tumours, commonly known as cancer metastases. Bone cancer is quite unusual as a primary tumour [6], but other forms of cancer that are in the later stages have a high risk of developing bone metastases. Studies show that primary tumours in the breasts, prostate, and lungs are the most common causes of bone metastases [11]. Most cases of patients with advanced PC will develop bone metastases that are, so far, incurable [33]. Even though the bone metastases are rarely the main cause of death of a patient, they lead to lower quality of life [10] and are strongly correlated with poor prognosis [43].

In our bodies bone tissue is continuously being built, broken down and rebuilt again in a process that is called bone remodelling. This process is mainly supported by two different cell types called osteoblasts and osteoclasts. The osteoblasts are responsible for creating bone tissue, while the osteoclasts do the opposite, i.e. they break down bone tissue. The balance between these two cell types are upheld in healthy bone tissue. Bone metastases disrupt this balance by either causing an abnormal level of bone construction, these metastases are called osteoblastic, or stop the reconstruction of bone tissue all together, these metastases are called osteolytic. Because of the bone overgrowth caused by the abnormal levels of bone construction, patients often suffer from pain, fractures and spinal-cord compression [33].

Studies have shown that bone metastases caused by PC are predominately of the osteoblastic kind [26]. The fact that metastases from PC are most commonly found in the form of bone metastases has led to a lot of research and development of diagnosis methods focused on identifying areas in the bone with abnormal levels of bone growth.

Some kinds of medical treatment for prostate cancer have a degrading effect on the bones, which further increases the risk of various bone injuries. This can be partially helped by using other medicines, though they have their own side effects, or by additionally use medicine to treat the degrading effect on the bones [42].

1.1.3 Bone Scintigraphy

Bone scintigraphy, also known as Bone Scan, is an image modality in which a patient is injected with a radioactive substance. This substance flows through the body and attaches to osteoblastic sites in the bone. After a short period of waiting to ensure that the substance has circulated through the whole body, the patient is placed inside a device that has two gamma cameras attached. These gamma cameras are sensitive to the radiation emitted by the injected radioactive substance. One of the cameras are placed in from of the patient, and the other is placed behind. So the resulting two images will show the anterior and posterior of the patient, see Figure 1.1.

Because the radioactive substance attaches to osteoblastic sites in the bone, these places will show up in the Bone Scan as hotspots with high intensities. Although these osteoblastic sites may have been incurred by a cancer metastasis, there are also many other types of bone injuries that activate osteoblastic activity and may therefore also give rise to hotspots in the Bone Scan. It is no easy task to differentiate the hotspots incurred by cancer metastases from other hotspots.

Compared to other image modalities that can be used to diagnose bone metastases, the Bone scan is cheap and is also the most widely available in hospitals. Together with the fact that the Bone Scan has high sensitivity has made this technique the most common way of discovering and diagnosing bone metastases [2].

1.1.4 Bone Scan Index

After a patient has gone through a Bone Scan, it is up to a doctor to interpret and evaluate the results. The evaluation of the Bone Scan is a very complex process and to a certain degree subjective. This leads to large variance between hospitals, doctors [40] and even the same doctor evaluating at different occasions [28]. The Bone Scan Index (BSI) was therefore developed as a reproducible quantitative measure of tumour burden on the bones, to counter the large variance and also utilize the fact that Bone Scans are so widely available [28].

BSI is defined as the percentage of total skeletal mass taken up by bone metastases. This is calculated from a Bone Scan by first classifying the hotspots as bone metastases or something else. The percentage of the anatomical region of the skeleton that is covered by hotspots classified as bone metas-

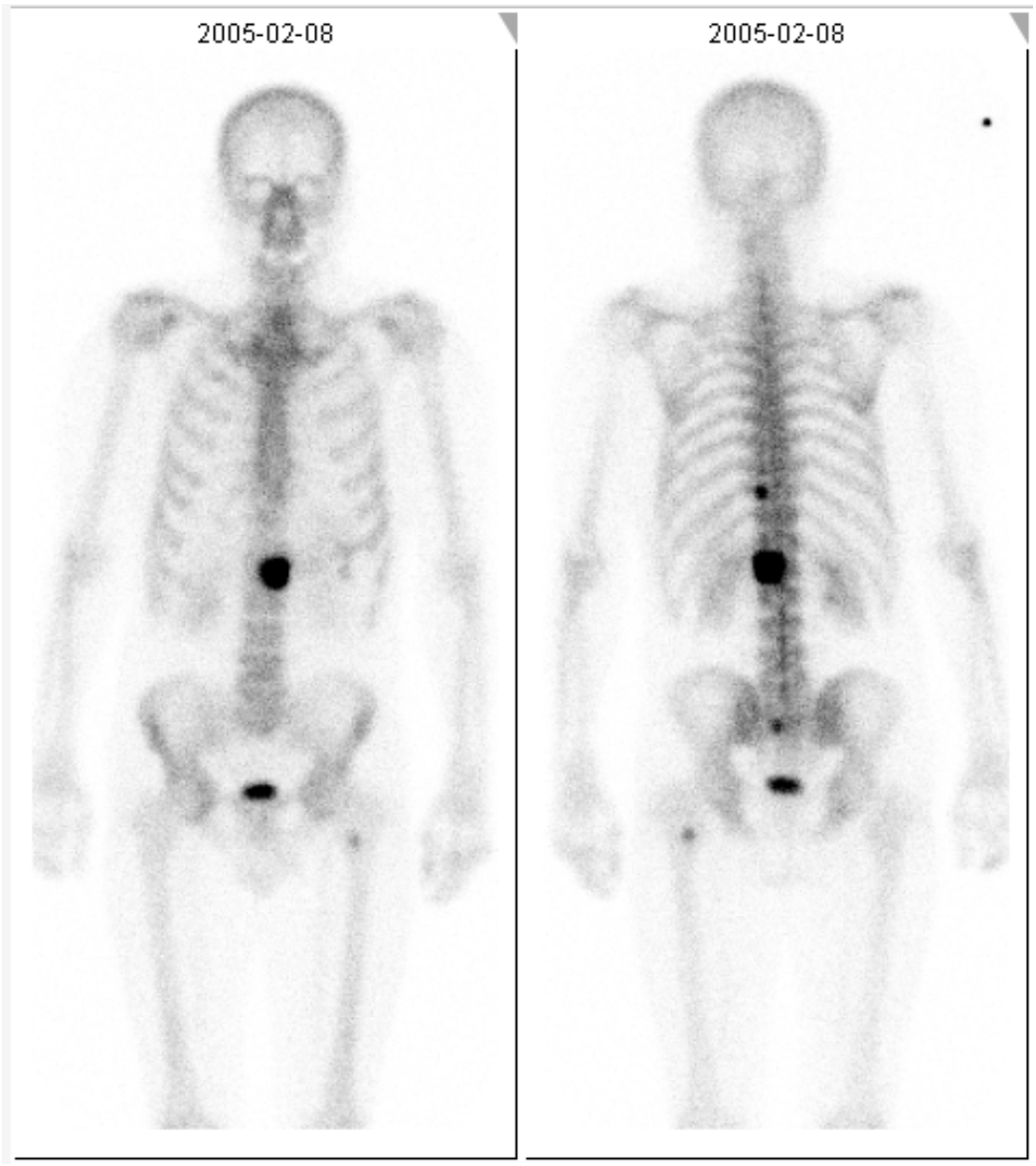


Figure 1.1: An example the resulting images acquired from a Bone Scan performed on a patient with some hotspots. Note that Bone Scans always give two images, one from the front (anterior) and one from the back (posterior) of the patient.

tases is calculated. Each region is then weighted by a coefficient that reflects the regional proportion of total skeletal mass. All the involvement values of the bone metastases are then added to get the BSI, which is a value between 0 and 100, where 100 represents that 100% of the entire skeleton is covered by bone metastases [28].

Studies have shown that BSI can be used as a prognostic tool in advanced prostate cancer. It may also become an important tool for risk classification and evaluation of treatments [15]. Even with these results, the use of BSI has not been widely adopted in hospitals and has been mostly limited to the hospital where the method was originally developed, at Memorial Sloan Kettering Cancer Center in New York [27]. This is largely because the method is done manually and is therefore time consuming and also suffers from being subjective. Therefore it is important to develop programs that can automatically calculate the BSI quickly while also being accurate at the same time. This is to increase the usage of BSI as a prognostic tool when diagnosing PC.

1.2 Related Works

Currently there exist a commercially available software package, called EXINI bone™, that is capable of detecting, segmenting and classifying hotspots and also calculate the BSI for whole-body Bone Scans. This is done by first segmenting the entire skeleton from the background in both the anterior and posterior views. A shape model based on a mean shape of several normal whole-body scans is then fitted to the skeleton using an image analysis algorithm called Morphon registration. The skeleton is divided up into several different skeletal regions, e.g. skull and ribs) and the hotspots in each of the regions are identified as regions with intensities that are sharply higher than its surrounding. Around 20 to 30 different features describing each hotspot are then calculated and used as input to an ensemble of ANNs that classify each hotspot. These ANNs have been trained on tens of thousands of hotspots that have already been labelled by experienced nuclear medicine physicians [47]. The BSI is then calculated as explained in Section 1.1.4.

1.3 Aim of the Thesis

The main goal of this thesis is to design and implement a type of artificial neural network called Convolutional Neural Network (CNN) to classify hotspots in Bone Scans. The focus of the thesis will be on the classification problem, and the identification and segmentation problems will not be considered. The data set will be provided by Exini Diagnostics AB, in the form of image patches of already found hotspots. Because of the time frame, only hotspots found in the spine will be used to train the CNN, as these are considered to be the easiest to classify.

We will investigate and apply various techniques available to CNNs to improve the performance of our CNN. The performance will be evaluated using common machine learning performance metrics, e.g. F_1 score, ROC curve and accuracy. The performance of our CNN will be compared with an ordinary ANN with similar properties as the ones used in Exini bone™ described in Section 1.2. To make the comparison fair, the ANN will be trained on the same data set using only features that can be collected from the image patches.

An additional side task is included in the thesis. The task is to train another CNN on Bone Scans to classify whether the image is an anterior or posterior view of the patient. It is considered to be easier compared to the main task, so the student will start tackling this task first. This is to let the student familiarize with the deep learning library Deeplearning4J (DL4J) [45] and the programming language Scala [38]. It is also seen as an evaluation of the performance and capabilities of DL4J for the purpose of using it in production. If the framework is deemed to not live up to expectations, the main task will be solved using another deep learning framework.

Chapter 2

Methodology

2.1 Machine Learning

2.1.1 Introduction

Ever since computers were invented, we have always fantasized about creating artificial intelligence, i.e. a machine capable of intelligent behaviour. One of the most defining features of intelligent behaviour is the ability to learn; to improve with experience. Even though we are able to do this since birth, the actual process of learning is still something we have not yet fully understood. The field of machine learning explores the construction of algorithms and programs that make computers capable of learning the underlying structure in a data set. This is done with the purpose of solving a problem without the need of using explicit descriptions in the program designed by a human being. This does not mean that the goal of the field of machine learning is specifically to create artificial intelligence, though the two fields may overlap. Instead, machine learning is used as a tool for data analysis and for building models of problems that can adapt and improve with increasing amounts of data.

A formal definition of the class of problems that is covered by machine learning is given by Tom M. Mitchell,

> *A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E. [36]*

This is intentionally a very broad definition of learning for the purpose of including tasks that we would conventionally call "learning" in everyday language. In this report the main learning task T is the classification of cancer hotspots in bone scintigraphy images caused by metastases from prostate cancer. The experience E will come from studying labelled cancer hotspots in bone scintigraphy images. The performance P will then be measured as the percent of hotspots that are correctly classified.

To motivate the use of learning algorithms, we will first take a look at non-learning programs, also known as expert systems. Expert systems are usually written with predetermined rules made by the programmer. This works very well in cases where there are not too many different situations that the programmer has to take into account when writing the program. As an example, we can take a basic calculator, which is often one of the first programs a novice writes. The various arithmetic functions in the calculator all have their own specific rules of how they should function that are easily implemented. The erroneous use cases, such as trying to divide by zero or multiplying non numbers, can also be predicted with relative ease. In this kind of program there is no need to use any machine learning algorithms, as every part of the program can covered by the programmer.

If we instead look at a more complicated problem, e.g. face recognition, we quickly run into problems when trying to write an expert system to solve it. Without bothering with the problem of having to find the face in a picture, there are still problems of how to differentiate one face from another. Just trying to code a representation of a face that computers can understand, i.e. describing what an eye or a mouth looks like, is in no way a trivial problem. We would then have to write rules in advance that should be flexible enough to differentiate various faces, while still not mistaking similar faces. This is something that is not possible for the programmer to fully take into account as there are too many variations of faces.

Unlike expert systems, machine learning uses a class of algorithms that have a data-driven approach. This means that when we use a machine learning algorithm, we train a model using data to tell it what a good answer to the problem is. As long as we have a data set of faces as a training set, chose a fitting learning algorithm and can select good features as input to the learning algorithm, e.g. the area of the eyes and mouth, the model may learn to recognize faces on its own. This way, we will not have to know what exactly to look at when differentiating faces, as the model has learnt the rules by itself by looking at the data. There are many parameters that may affect

the performance of the model, such as the choice of learning algorithm, what features to extract and how much data that is needed. Note that we will be using the terms train and fit interchangeably, as they are both ubiquitous in the machine learning community.

Because deep learning is just one of many types of learning algorithms, many of the concepts used in conjunction with deep learning are either shared or a variation of concepts used in other machine learning techniques. We will therefore introduce the various concepts from machine learning in this section by looking at the various parts that make up a basic machine learning work flow, see Figure 2.1.

The first step in most machine learning work flows is to analyse the given task and decide what kind of learning algorithm that will be used. Section 2.1.2 therefore introduces three broad classes that machine learning algorithms are usually separated into. The next step in the work flow is then to collect the data, where there are some requirements and important aspects of the data collection one has to think about, which will be presented in Section 2.1.3. The data then has to be split into two different sets, to make it possible for us to first train the model on the first set and then also evaluate the performance of it using the second set. The reasoning for this and some notes in case of unbalanced data will be discussed in 2.1.4.

Once we have the structured data, there are some preprocessing techniques that may speed up the training process and/or improve the performance of the trained model, which will be discussed in Section 2.1.5. We will then start introducing the training phase in Section 2.1.6 where some important concepts for training a model will be presented. Gradient descent, which is an optimization algorithm used to train many machine learning models, including neural networks, will also be presented in this section. The last part of the work flow is then to evaluate the performance of the trained model, and some common performance metrics in machine learning will be introduced in Section 2.1.7.

2.1.2 Types of learning

There are many ways to categorize learning algorithms, but in the machine learning community, the standard has been to divide them depending on what kind of data sets they use. The three main classes that are used are supervised learning, unsupervised learning and reinforcement learning. Note that a learning algorithm may be fitted to work under different classes, and

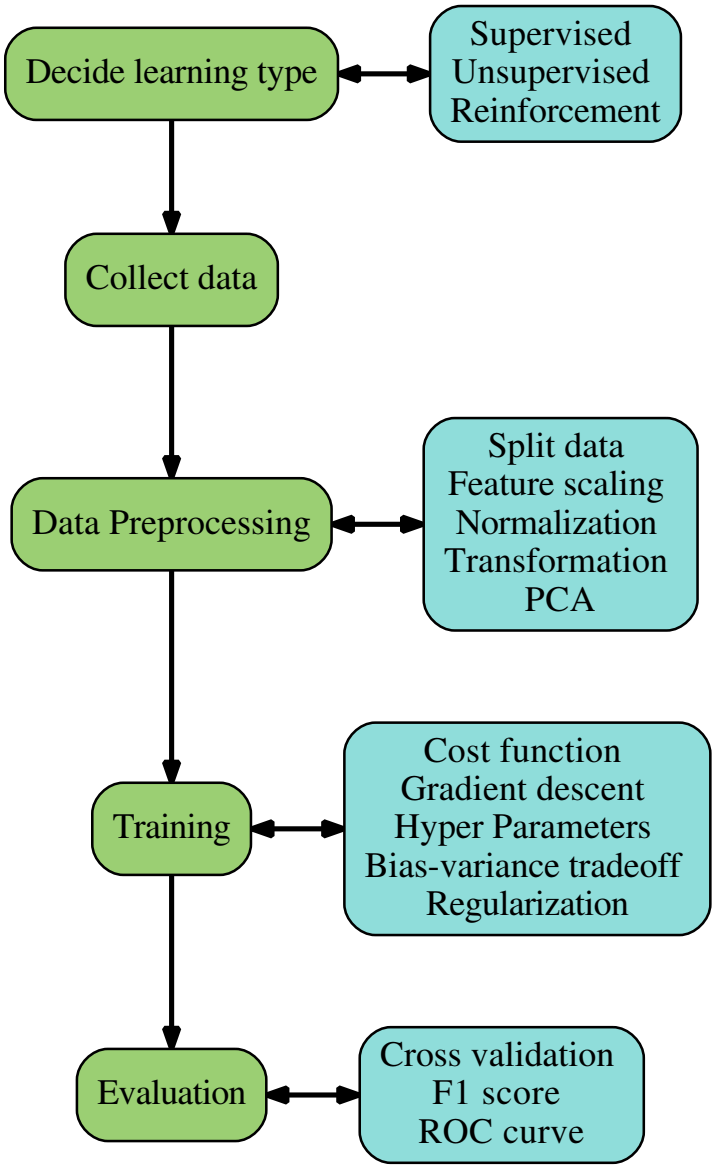


Figure 2.1: A basic machine learning work flow.

that some algorithms may fall in between two classes, such as semisupervised learning. These exceptions will not be discussed in this report.

Supervised learning

In supervised learning, in addition to the input in our data set, there is also an associated output to each example. A data set of N examples can then be seen as a collection of N input-output pairs $(\mathbf{x}_i, y_i), i \in (1, \ldots, N)$, where $\mathbf{x}_i \in \mathbf{R}^m$ is one input data point with m features and $y_i \in \mathbf{R}$ is the output. When training a model using supervised learning, the model tries to infer the underlying rules that explain the connection between the inputs and their corresponding output. In a mathematical sense, this can be seen as the model trying to learn the transfer function $h : \mathbf{R}^m \to \mathbf{R}$ which connects the input to the output as $h(\mathbf{x}_i) = y_i$ for $i = 1, \ldots, N$. It can do this by building a guess of how the function h looks like and then predict the answer. The answer will most likely not be correct, but by calculating how erroneous it was, it can then update its guess and redo the process. So when using supervised learning, we assume that when the model can approximate the transfer function well over a sufficiently large part of the data set, it will also approximate the transfer function well over yet unobserved examples. This can be seen a special case of the discussion in Section 2.1.3.

The two most common machine learning problems associated with supervised learning are *classification* and *regression*. Both of these problems are similar in nature, with the difference being what kind of outputs that are associated with each example. In the case of classification, the transfer function h will be discrete and this output can be seen as a class label. When training a model for a classification problem, one way of visualizing the result of the model is to draw the decision boundary together with examples in a graph. This decision boundary can be seen as the line that the model uses to classify new unobserved examples. Something that is important to have in mind when working with classification problems is that it is insufficient to only have examples of the object we want to classify, so called positive examples. It is just as important to have negative example that do not belong to that class.

A simple example of a classification problem could be whether a student is admitted to a specific school given the results from two different exams. In this case, $m = 2$ and there would be two different labels, $y_i = 1$ if admitted and $y_i = 0$ otherwise. Previous admittance results can be used as training

data. Figure 2.2 shows a synthetic example of the example task, where the points in the graph are used to train a logistic regression model.

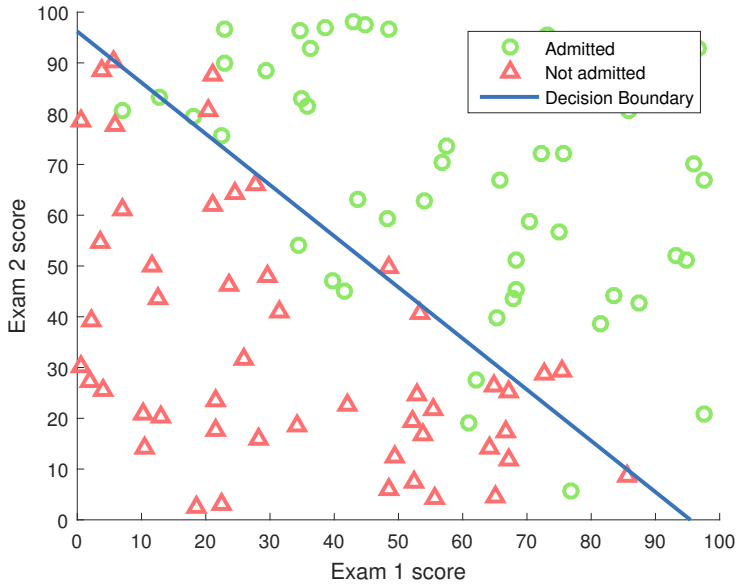


Figure 2.2: An example of a classification problem. The task is to classify whether a student gets admitted to a school given the results from two exams. The points in the graph are the training data used to train a logistic regression model, which resulted in the decision boundary in the graph.

Going over to the other type of problems in supervised learning, regression, where h will be real-valued instead. In this case, they do not function as labels, but instead it is a value from an continuous function f that we are trying to approximate using a machine learning algorithm given the collected data. This function f could be almost anything that outputs continuous values, e.g. the price of a car, that could then be approximated using features that we would think relates to the function, such as how old the car is, the top speed of the car and so on. A synthetic example based on the car example using only the top speed of the car as feature can be seen in Figure 2.3.

When training a model using supervised learning, there are often a problem of choosing which features to use. Not using enough features could lead

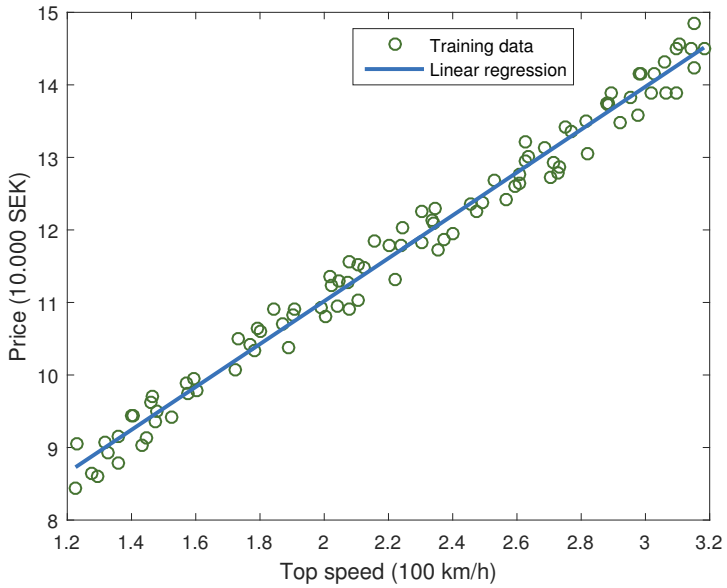


Figure 2.3: An example of a regression problem. The task is to predict the price of a car given its top speed.

to the model not able to reach the performance level wanted, as the model will not have enough information to derive the relation between input and associated output. It often comes up as a problem caused by poor understanding of the task, like in the example in Figure 2.3, where we can be sure that in a real world situation, the top speed would never be enough to predict the price of a car.

When looking at the case of too many features, there are two different types of problems that may come up. Using features that are not related to the task at hand could lead to the model finding some correlation between the unnecessary features and the output in specifically in our training data by coincidence. The model could then give good results for our training set, but when we test it on unobserved examples, it would likely give wrong answers. Using features that are redundant, such as the same features in different units of measurement, would likely not lower the performance of the model, but it may increase the training time, as the model would have to learn to filter the redundant information in the features. This will be further discussed in Section 2.1.6.

Unsupervised Learning

In unsupervised learning, the machine learning algorithm is only given the input data x with no associated output. In this case, there is no clear goal for the learning algorithm to work against. This may come up in many different situations where either there may not exist a clear answer or we just do not know the answer in advance. Unsupervised learning is therefore commonly used to group examples into different classes, i.e. clustering. The examples that are similar to each other are grouped together according to the underlying structure the model finds in the data. An example of an clustering algorithm, K-means, used on an unlabelled data set can be seen in Figure 2.4. Note that the learning algorithm by itself cannot chose the number of groups to split the data into, so this is a hyper parameter, which will be further explained in Section 2.1.6.

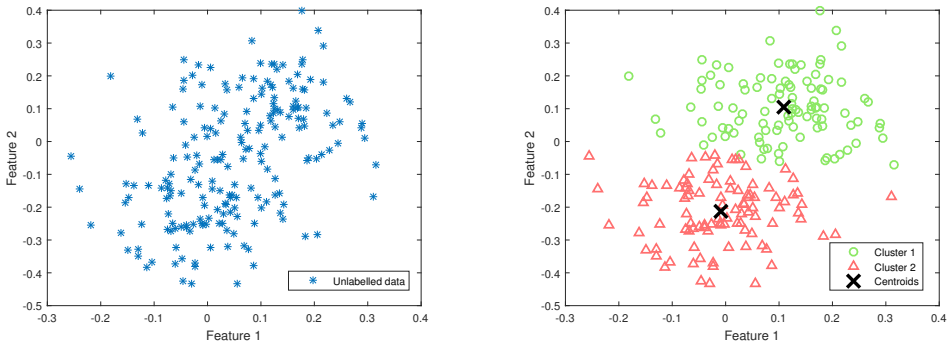


Figure 2.4: An example of an unsupervised learning algorithm, K-means, used on a synthesised unlabelled data set with no clear grouping.

Reinforcement learning

Reinforcement learning is very different from the other two classes in how it works. It is inspired by behavioural psychology, most notably the "carrot and stick" approach. Instead of supplying the machine learning algorithm with data, we make it interact with a dynamic environment where it has to perform a task to reach a certain goal. The model may not even have been told how or what it should do to achieve this goal. Instead the model is rewarded with positive or negative reinforcement after it has performed an

action. The model will then have to learn what kind of actions and how to do them to get the most positive rewards while avoiding the negative ones. An example would be learning to play a game against an opponent, e.g. chess. In this case, the learning algorithm would move the chess pieces in order to win the game, and there would be positive rewards for winning games and negative rewards for losing games and moving the chess pieces in a way that is not allowed in the game.

2.1.3 Data collection

One of the most central aspects for the performance of a trained model is the amount and quality of data that is used as input. One could even say that the trained machine learning model is only as good as the data used to train it. Therefore it is important to think about what kind of data that is needed for the specific task and a good way of collecting it. Just gathering a large amount of data and using it in its raw form may not give the best results when training a model.

We must first make an assumption in this report on the data that is collected, for theoretical purposes. This is to corroborate the concept of a data-driven approach, i.e. that training a model with learning algorithms using observed examples as data is possible. The assumption is that the examples in the data set can be seen as a collection of independent and identically distributed (iid) random variables taken from some distribution P_X. This means that the occurrence of one example does not affect the occurrence of other examples and that all the examples come from the same population/source. This does not mean that all the examples have the same probability of being observed. For example, the sequence of coloured balls taken with replacement from a jar, filled with 10 red and 2 blue balls, is iid.

It is important to note that when we talk about the data, we actually mean the collection of examples, where each example is an realization of the process that we are trying to analyse. Each example is in turn made up of data points that may be collected at different occasions, though they are usually obtained at the same time. This distinction is important because the assumptions above are put on each example, but the features in each example will most likely be correlated with each other. For instance, if we wanted to collect a data set of tree image, each image would then be a realization of the underlying process that in some sense define what trees look like. The pixels in the images would then be the data points within each example, and

these would be correlated with each other.

Going back to the assumptions, the important part is that the examples are identically distributed, as this simplifies the mathematical proofs for machine learning, and is in some cases what enables learning algorithms to work. The reason for this is because training a model on a specific problem can be seen as it trying to learn the underlying structure of P_X. That means that the structure that it has learnt from one set of examples can be used to help predicting the answer to new unseen examples from the same distribution. This will not necessarily hold for other distributions.

In practice, because the assumption of independence is quite strong, it is often sufficient to assume that the examples are exchangeable. This means that the collections of examples that we have gathered from the larger population of all possible examples have the same probability as any other combination of examples. This property can be seen as a formalization of the notion of "predicting the future based on past experience" [31], which is enough to ensure that we can actually fit a model.

Because of the assumption made on the data, we can now explain what the machine learning algorithm is actually learning. As mentioned earlier, our goal can be seen as having the model learn the underlying structure of the distribution P_X for all possible examples. We can see a distribution as having two main properties, which are the mean and the variance, that in some sense loosely define it. So when we draw an example from a distribution, we get an example that is the combination of some mean example, which has all the important properties of the population, together with a collection of variations that make each example different. Therefore, when we train a model, it is essentially trying to learn to replicate these two properties, though this is in general a massive simplification. The main issue when we train our model on our collected data is that it is actually learning the structure of the distribution $P_{X_{collected}}$ associated with our data. But through our assumptions, we can ensure that it is possible to make predictions about unseen examples drawn from P_X by using the experience gained from our data. The number and the quality of the data collected decide how similar the distributions P_X and $P_{X_{collected}}$ will be, and will therefore affect how well the trained model will perform. In this case the quality of the data represent how well it can represent the population which P_X is based on. In classification, this means that data should contain sufficient number of data points from each of the relevant classes.

As an example, imagine we had collected a data set that only consisted

of images of chihuahuas. It would not be suitable to only use it to train a model for dog recognition, as it would not represent the larger population of all possible dogs, and the trained model would only classify chihuahuas as dogs. But if we instead wanted to use it to train a model to recognize chihuahuas in pictures, then it could be a quite useful.

2.1.4 Structuring the data

Before doing any transformations on the raw data, it is important that we split the data into two different sets. One will be labelled the training set, which will be used to train our model, while the other will be labelled as the test set to evaluate the performance of the trained model. The reason for this is that we use $P_{X_{collected}}$ as a way of approximating P_X when training our model. The data used to train the model can then not be used to evaluate the performance of the model, as the model has already seen the examples and will have a higher chance of predicting the correct answer. These results will not reflect the actual performance of the model on new unseen examples. The information in the training set has in some sense been "used up" when fitting the model, so we can therefore not draw any additional conclusions using this data set.

An analogy to this problem would be a student doing a test. The study material will be the training set that the student uses to learn about the subject. When doing a test to evaluate the performance of the student, we would naturally not use the exercise questions from the study material as test questions, as this would not reflect how well the student has understood the subject. The obvious solution to this would be to just use new test questions, but in our case, it may not always be viable or even possible to acquire more data, so instead we split the data. The test set will then act as the new test questions that remain unseen while training the model. Note that when splitting the data, we assume that both sets will function as approximations to the original distribution. It is therefore necessary that the split is done in a "fair" way, such as randomly assign each example to a specific set.

When solving a classification problem, it is common that one or more classes are underrepresented in the data set. The may be due to various reasons, e.g. the underrepresented classes occur more seldom compared to the other classes, like patients with rare diseases versus healthy patients, or they may be more expensive to sample. This skewness in the data set may give rise to two different kinds of problems, depending on if the unbalance is

in the training set or in the test set.

The first kind comes up during training, in which the model will be presented with less examples from the underrepresented class that may lead to the model not being able to generalize what it has learnt to unobserved examples. In this case the two approaches that will be used in this thesis, which are also the most common, are oversampling and undersampling. In oversampling, examples in the the underrepresented classes are duplicated, either by random sampling or by some sort of systematic sampling. Undersampling can be seen as the opposite of oversampling, as it works by instead throwing away examples from the overrepresented class. In both approaches, the process is repeated until the unbalanced has been fixed, though equalizing the unbalance all the way may not always give the best results. Note that in some cases, this skewness in the data set may not cause any trouble when training a model, so it is often a good idea to start training the model without taking the unbalance into account.

The second kind of problem that may come with unbalanced data, arises when evaluating the performance of the model on the test set. Most of the time when evaluating the performance, the accuracy is a common metric used to compare the results with other models. But accuracy is not very usable when it comes to unbalanced data sets. This problem and ways to work around it will be further discussed in Section 2.1.7.

2.1.5 Data preprocessing

When working with machine learning algorithms, there are many reasons that the data used as input to the learning algorithm may need some kind of preprocessing, to either improve the performance or speed up the model fitting.

A common situation is that the features in the data may vary in scale and in some cases have differences on several magnitudes, e.g. the age of a person versus their income. The first step is therefore often to standardize the features in the examples. A common standardization method is to rescale each feature into a common range, often chosen to be either $[0, 1]$ or $[-1, 1]$, though the exact boundaries can vary as long as they are around the same scale. In some cases, it is better to normalize the data to have zero mean and unit variance, i.e. whitening. The choice of standardization method is very problem dependent and it is often best to try several different ones.

One of the reasons for standardizing the features is to make the training

process more numerically stable. The numerical instability is caused by the finite precision computers can use to represent numbers and calculations. As each computation causes a small error, which then propagates through the training process, and grows until it starts affecting the results and diverges from the true results. Another reason that feature scaling is used is to ensure that all the features are treated equally during the training process, which will be explained in Section 2.1.6.

Another type of preprocessing that is often employed in machine learning pipelines are dimensionality reduction. This means that we let an algorithm reduce the number of features used in each example. A common algorithm for this is Principal Component Analysis (PCA). For more information about PCA, see [41].

It is important to note that the parameters for the standardization method and any preprocessing algorithms should only be calculated on the training set. These calculated parameters should then be used in the same way to standardize and preprocess and new unobserved examples, like the test set and when used in production. The same reasoning used in Section 2.1.4 applies here, i.e. we do not want the test set to affect the model during training. In addition to this, it is also important to ensure that the data are in the same "state" as the training data was in when used to train the model.

2.1.6 Training phase

A model of a learning algorithm can be seen as a function $h(\mathbf{x}; W)$, where $\mathbf{x}$ is features from one example, and W is the weights that parametrize the function. Because this report is focused on a classification problem, i.e. a type of supervised learning, we will also have an output y that tell us the correct answer for each example $\mathbf{x}$. The weights are changed in order to fit the model to the given fixed input data. This is what it essentially means when we say that we train a model of a learning algorithm, and it is this process that is called learning in machine learning context.

There are many different kinds of machine learning algorithms and they differ in what kind of function $h(\mathbf{x}; W)$ they use and how the weights W are used to parametrize the function. One example of a common machine learning algorithm is linear regression,

$$h(\mathbf{x}; W) = \mathbf{x} \cdot W = w_0 x_0 + w_1 x_1 + ... + w_m x_m$$

where we have the function $h(\mathbf{x}; W)$ that takes the vector x, which has the

$m + 1$ features, $x_0, \ldots, x_m$, and is parametrized by W, which has $m + 1$ weights $w_0, \ldots, w_m$. Note that in linear regression x_0 is often predetermined to always be 1, as to make the weight w_0 work as the intercept term.

Loss function

When training a model with supervised learning methods, we know the true answer corresponding to each example. The model then learns by predicting an answer to the input and if the answer is wrong, it changes its parameters so as to hopefully give the right answer the next time. Before going into the process of updating the parameters to fit the new example, it is important that we have a measurement of how "wrong" the answer was, as this will be functioning as our goal when training the model. This is handled by a loss function

$$L(X, Y; W) = \frac{1}{N} \sum_{i=1}^{N} l(\mathbf{x}_i, y_i; W)$$

where $l(\mathbf{x}_i, y_i; W)$ is the cost function, N is the total number of examples in the training set and

$$X = \begin{pmatrix} \mathbf{x}_1^T \\ \vdots \\ \mathbf{x}_N^T \end{pmatrix} \in \mathbf{R}^{N \times m}, \quad Y = \begin{pmatrix} y_1 \\ \vdots \\ y_N \end{pmatrix} \in \mathbf{R}^N$$

which hold the m features of each example and the corresponding label in each row. The function $l(\mathbf{x}_i, y_i; W)$ is non negative and takes high values when the model makes a big mistake on the i-th example and vice versa. The function $L(X, Y; W)$ is consequently also a non negative, but takes high values when the model is doing a poor job on the whole training set.

When talking about various loss functions, what actually differentiates them from each other is the cost function used on each example. One loss function commonly used in statistics for regression problems, is the least squares loss function (also called mean squared error or sum of squares of errors), which has the form

$$L(X, Y; W) = \frac{1}{N} \sum_{i=1}^{N} l(\mathbf{x}_i, y_i; W) = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{2} (h(\mathbf{x}_i; W) - y_i)^2$$

where $h(\mathbf{x}_i; W)$, as mentioned above, is the prediction made by the model on the i-th example and y_i is the corresponding true answer.

Optimization

Because the loss function is a measure of how well the model is doing on the problem, our goal when training the model is to minimize this function. So the problem of learning can be seen as an optimization problem where, given the input data X and ground truth Y, we want to fit the model parameters W such as to minimize the loss function. This minimization problem in the case of ANNs is non convex and does not have any closed form solutions. One can see the optimization problem as a landscape which is filled with hills and valleys. This landscape is the space spanned by all the possible settings the weights may be in, where the height of each point corresponds to the value of the loss function. The goal of the problem is then to find the lowest position in the landscape, i.e. the configuration of weights that gives the lowest loss.

As there is no closed form solution, the solution to the optimization problem cannot be found in just one step. Instead we have to use an iterative optimization algorithm that steps through the parameter landscape in a direction that will hopefully lead to a minimum given enough steps are taken. Iterative methods work by calculating which direction to go in in each step and updating the parameters accordingly. This is repeated until a satisfying results is reached, i.e. the algorithm diverges or time runs out for the training process.

Because the problem is non convex, when we have found a minimum, we can not be sure if we have found a global minimum or if it is maybe just one of many local minima that the optimization algorithm may get stuck in. Therefore, in bigger problems where this training process becomes a time consuming task, it is rarely feasible to find the global minimum, so one often has to settle for a local minimum. Although there is usually no way to determine if we have actually found a minimum and what kind of minimum it is. For that reason it is important that there is some other way to evaluate the performance of the model, as mentioned earlier and will be further presented in Section 2.1.7.

Gradient descent

As mentioned earlier, each iterative optimization algorithm works by calculating a direction for the weights to change in as to reduce the value of the loss function $L(X; W)$. Recall that the loss function $L(X, Y; W) =$

$\frac{1}{N}\sum_{i=1}^{N} l(\mathbf{x}_i, y_i; W)$ is a function of all input data X and labels Y that is parametrized by the weights W. This means that in our optimization problem it is the weights that get changed as the input data is already fixed. One of the most common optimization algorithms used in machine learning is called gradient descent, which is a first order optimization algorithm. This means that gradient descent uses the information of the function value $L(X, Y; W)$ together with the gradient of the function

$$\nabla_W L(X, Y; W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W l(\mathbf{x}_i, y_i; W)$$

to calculate the direction. We will be using ∇ to denote the gradient ∇_W, as this is assumed to be the case throughout the report.

If we call the starting point in the configuration of the weights to be W_0, then we know that the gradient $\nabla L(X, Y; W_0)$ shows in which direction that the function increases the fastest in a close neighbourhood around W_0. This also means that the negative gradient $-\nabla L(X, Y; W_0)$ shows the direction of the steepest decrease around W_0. It then follows that if we make a step to W_1 as

$$W_1 = W_0 - \eta \nabla L(X, Y; W_0)$$

with η, which is the step length we use to move in the negative gradient direction. This step length is better known as the learning rate in the machine learning community. If we chose the learning rate to be small enough we will have

$$L(X, Y; W_0) > L(X, Y; W_1)$$

If this stepping process is repeated enough times, we will hopefully reach a minimum, though the value of η has a big effect on whether it will converge, and if so, how fast. If η is too large, the algorithm may never land in a minimum and will most likely diverge, but if it is too small, it may take a long time for it to reach a minimum.

Note that the gradient descent algorithm calculates the gradient of the loss function by calculating the average gradient of the cost function of each example in the training set before it can update the parameters once. Therefore if the training set is large, like 10^5 examples, it would mean that for every step that is taken, the optimization algorithm would have to calculate 10^5 gradients. Although this is the way to calculate the true gradient, it is time-consuming, especially if we have redundant data in the training set. For this reason, there is a stochastic approximation of the gradient descent,

called stochastic gradient descent (SGD), where we instead approximate the true gradient as

$$\nabla L(X, Y; W) = \frac{1}{N} \sum_{i=1}^{N} \nabla l(\mathbf{x}_i, y_i; W) \approx \nabla l(\mathbf{x}_j, y_j; W)$$

where the approximated gradient is calculated from just a random example $j \in \{1, 2, \ldots, N\}$ which means that we only have to calculate one gradient each time we take a step in the optimization algorithm. So when the ordinary gradient descent has evaluated all N gradients to take one step, SGD can take N steps. This leads to a significant speed up in the training process, though that does not mean that SGD is N times faster, as there more than just gradient evaluations in the algorithm. The negative side of SGD is that without the averaging over examples, we get very noisy gradients that may not always approximate the true gradient very well.

It is common to use a mix of these two methods, which is called mini-batch gradient descent. In this method we split the training data into mini-batches of k examples each (except maybe the last mini-batch which gets the rest), where $1 < k < N$, and calculate the average gradient for each mini batch. This method would have the advantages of, i.e. faster stepping and still be a good approximation of the true gradient.

Hyper parameters

Looking more closely on η, we see that it is a parameter in gradient descent that does not get changed by the training, on the contrary, it affects the training. The parameter η has to be chosen in advance of the training process, which in a statistical way can be seen as a parameter of a prior distribution which encompasses prior knowledge. These kinds of parameters are called hyper parameters, as a way to distinguish them from the model parameters that are fitted during training. An analogy to this would be a football team where the training of the players can be seen as the usual parameter optimization, while the way to train them and the schedule for training can be seen as hyper parameters.

Another example of a hyper parameter would be the number of features to use to build a model in linear regression. This does not directly change the training process, but instead changes the constitution of the model. In this example, linear regression pretty much only has the number of features as

hyper parameter, together with the learning rate η for the gradient descent. But if we switch over to the learning algorithms called neural networks, which we will have a closer look at in Section 2.2, they may have upwards of one hundred different hyper parameters.

Hyper parameter optimization

Because the hyper parameters have to be chosen before we start the training process and that they either affect the training or the actual model, it is important to pick "good" hyper parameters. This is somewhat alike the optimization problem associated with training, but is in this case called hyper parameter optimization. The question is then how to know which values that are actually good and how to find them.

We start by assuming that we have already found hyper parameters that we think are good. The problem is then how to measure how good the hyper parameters actually are, which mostly means to measure how good our model is. Therefore it is natural to use the same way we evaluate the model as way to evaluate the hyper parameters, that is to use the test accuracy. Note that the accuracy may not always be a good evaluation measure, which will be further explained in Section 2.1.7.

Recall the discussion from Section 2.1.4, where we argued for the splitting of our data set into training set and test set. The reason for the split was to ensure that we would not "use" up the information contained in the test set, since that would invalidate the test accuracy. The same reasons apply when we want to do our hyper parameters optimization, in which we update the hyper parameters depending on the test accuracy. In this case, the solution is to have a third data set which we do not use to train our model, but only to evaluate the performance of our hyper parameter optimization through the accuracy. This third data set is called validation set, and can be seen as a fake test set used to tune the hyper parameters. So when we have reached a satisfactory performance, the validation set can be seen as having been expended in a somewhat same way as the training set. We always want to use the test set only once at the end of the whole training process, including the hyper parameter optimization, as to ensure that the results will stay unbiased. As a continuation of the student analogy made in Section 2.1.4, we could see the validation set as a form of equivalent to studying old exams. So the student would start out with learning the materials in a course before studying old exams as a way to prepare for the real exam. If the student has

problems with any subject in the old exams, the student would then go back to the course material to train. The final test for the student would then be a new exam.

After having found a way to determine if a set of hyper parameters is good, we still have to come up with a method to find them. In this case, we cannot use gradient descent, because we have no clear function from hyper parameters to accuracy that we can calculate the gradient from, as this function would involve the training process itself. It is therefore seen as a black-box function that can receive an input and generate an output, but we have no idea of its shape. Because of the lack of information, the most common way to optimize the hyper parameters is to just use grid search.

Grid search is a very simple optimization algorithm in which we do an exhaustive search through a specified subset of the hyper parameters. This means that we select a set of reasonable values for each hyper parameter and then evaluate the performance of each combination of the hyper parameters. The "optimal" hyper parameters are then the set that gave the best results. Because each test of a set of hyper parameters is independent from the others, it makes grid search easy to parallelise. A problem with grid search is that it requires some knowledge about what kind of values that are reasonable for each hyper parameter. It also suffer from the curse of dimensionality, which means that the number of possible combination of hyper parameters grows exponentially with increasing number of hyper parameters. So if we have 2 hyper parameters with 10 different values, we have 100 different combinations, but if we just double the number of hyper parameters with the same number of values, we suddenly have 10.000 different combinations.

Since the use of grid search performs an exhaustive search through the set of potential hyper parameters, several other alternatives have been proposed that should give better performance. One of them is random search, in which instead of selecting a set of reasonable values, we just randomly sample values from a predetermined interval for each hyper parameter. It has been shown that this method will have the same or better performance than grid search [8]. In this thesis, a mix of random search and grid search will be used when doing hyper parameter optimization.

Underfitting and overfitting

When using machine learning algorithms, there is in most cases some hyper parameters that affect the complexity of the model. An example of this

would be how many features to use in linear regression, see Figure 2.5.

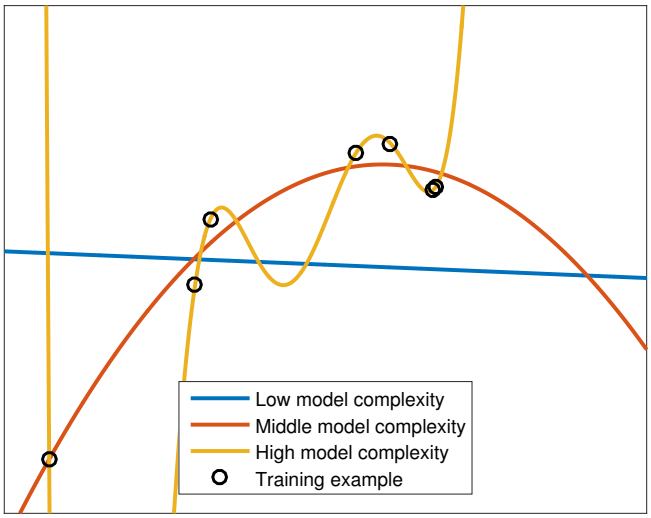


Figure 2.5: Example using linear regression to show how model complexity decides how well the model can fit to the training data. Increasing model complexity leads to better fitting on the training examples.

The training duration can also be seen as a contributing factor to the model complexity, as the longer the model gets to train, the more complex function it can represent, to a certain degree. Model complexity in some sense defines how capable the model is able to fit the underlying distribution of the data. This means that as the model complexity increases, the model will be able to replicate increasingly complicated functions. The problem is that the examples in the training data are just samples from the larger population of all possible examples, i.e. the global population. The training data can therefore only be seen as an approximation of the global population, with its own mean, variation and inherent noise within the training set that may not have to be the same as the global population. So, as mentioned in Section 2.1.3, when training the model, it learns the distribution of our training set as an attempt to generalize to the real distribution. If we once again bring back the student analogy, this situation would correspond to a student studying

the course material as a way to learn the actual subject. The course material will not be able to bring up every question and problem of the subject, but the student is expected to generalize the knowledge acquired from the course material when encountering new problems. A parallel to the inherent noise would be that each course book would have its own set of errors and mistakes, which the student will have to learn to avoid when generalizing.

An important concept in machine learning often associated with this problem is the bias-variance tradeoff. In our case, it is about the relationship between the model complexity and the ability for the model to generalize to unobserved examples. This relationship can be summarised in a curve with the error (which in many cases mean the loss function) on the y-axis and model complexity on the x-axis, see Figure 2.6.

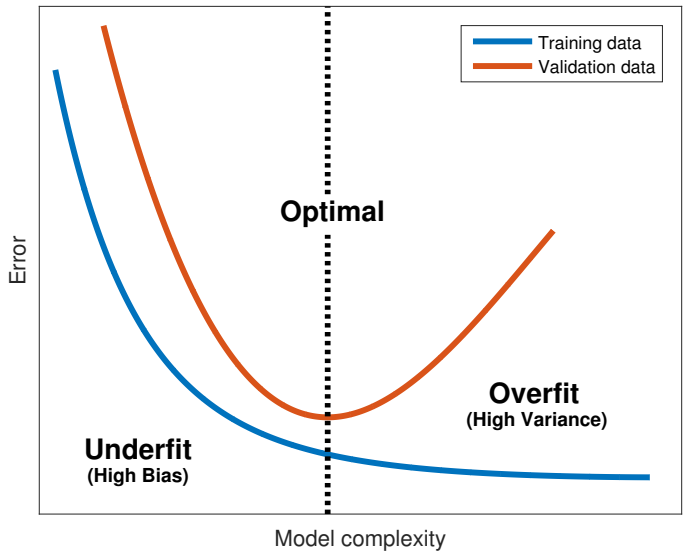


Figure 2.6: Simplified sketch of the relationship between the model complexity and the error the model makes on the training set (blue) and on the validation set (red). Too low model complexity leads to high bias, recognised by the high error on both training and validation set. Too high model complexity leads to high variance, recognised by the low error on the training set, but high error on the validation set.

If the model has too low model complexity, it will not even be able to learn the parameters of the training set distribution. This consequently means that it will not perform well when predicting the answers on the training set, not to mention the validation set. In the student analogy, this would correspond to a student not having studied the course material well enough. In this case, the model is said to be underfitting the data, also known as having high bias, because the expected prediction will be far from the correct answer. A sure sign of underfitting, as can be seen in Figure 2.6, is high error on both training and validation set. In the student analogy, this would correspond to only studying one chapter in the whole book. Although the student would be able to answer some questions correctly, the student is expected to be wrong most of the time.

As the model complexity increases, the model will start to learn more and more of the distribution of the training set and thereby decrease the error. It will therefore have increasingly better performance on both the training set and the validation set. But there will come a point where increasing the model complexity leads will start to worsen the performance on the validation set. This is the optimal amount of model complexity for the current set of hyper parameters, as the model has learnt everything it can learn from the training data that is useful. After this point, increasing model complexity will only lead to the model also fitting the inherent noise in the training data. This noise is something that exist in almost every measurement and has no correlation with the example itself. So therefore as the model learns the uncorrelated noise unique to the training data, the error on the training data will keep on decreasing, but the error on the validation will begin to increase. This is what is called overfitting, also known as high variance, because the random noise the model has learnt to take into account lead to a high variance in the answers that the model will give on unobserved examples. In the student case, this would correspond to the student also learning the mistakes that were made in the course material.

As an additional example, we look back at Figure 2.5, where we see that higher model complexity give better fitting on the training data. But if we test the models on new examples generated from the same distribution, see Figure 2.7, we see that in this case the model with the highest complexity was not able to generalize to new examples.

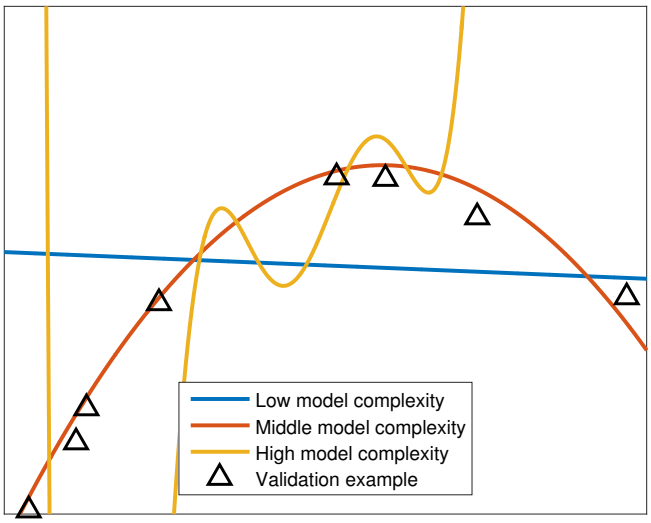


Figure 2.7: The same models as the ones in Figure 2.5, used on new examples generated from the same distribution as the training data. Here we can see that models with too high model complexity may not necessarily generalize to new examples.

Regularization

After the discussion in the previous section, we have come to understand that it may not always be beneficial to make the model as complex as possible. Even though they may be able to perfectly predict the answers to the training examples, i.e. attain zero loss, these models tend to not generalize very well to new examples. We know that the number of features used from each example is a hyper parameter that is bound to the model complexity, so one way to decrease the complexity could be to try to throw away some features. Assuming that there is no redundant information, we often do not want to throw away any features, as we rarely know if it actually could be valuable information for the problem. Another case which will often also lead to overfitting is in the case when the number of training examples are few in comparison to the number of parameters used in the model. This case will be especially important when we get to the artificial neural networks later in

the report.

Another aspect of the problem of overfitting is that a model with too high complexity has the ability to learn to depend too much on certain features that may be unique for just the training set. An example of this would be if we had a training set that consisted of images of cars, where many of the cars have four doors. Without bothering with how to represent it, when we train a model with too high model complexity, it may start to depend on the fact that many of the cars have four doors. This is obviously not true, as there are cars that have only two doors too. A way to solve this problem is to use a class of techniques called regularization. They do this by making the model prefer simpler models and penalizes complex models that have too much dependence on individual features in the data set. In another perspective, the use of regularization can be seen as our way of introducing prior knowledge about our desired optimal model into the training process. In this case, regularization induces the model to avoid high complexity.

There are many kinds of regularizations that fit to different kinds of algorithms, and some techniques that are not pure regularization techniques may have some regularizing effect. The most common ways to introduce regularization into our training process, is to add a regularization penalty to our loss function, that is

$$L(X, Y; W) = \frac{1}{N} \sum_{i=1}^{N} l(\mathbf{x}_i, y_i; W) + \lambda R(W)$$

where λ is a hyper parameter to control the impact of the regularization term $R(W)$. Notice that the regularization term only depends on the weights, and not the data itself.

The most common regularization techniques are the L_1-regularization and L_2-regularization, with their own effect on the training process. L_1-regularization, with the form

$$R(W) = \sum_i |w_i|$$

where the sum is over all the weights, will have the effect of making the weight matrix W sparse during optimization, which means that it introduces zeros to ensure that the model will depend on fewer features [18]. L_2-regularization, with the form

$$R(W) = \sum_i w_i^2$$

i.e. the squared sum of all the weights, will instead have the making the model pick weights that are smaller and diffuse the dependence over more features. So the final model will more likely have weak dependence over many features instead of strong dependence on a few features [18]. Note that it is possible to combine several regularization techniques, which would then have combination of the regularizing effects depending on the hyper parameter that controls each technique.

Something to have in mind during the training process when adding regularization terms to the loss function is that it is no longer possible to have zero loss. That would mean that all the weights are zero and at the same time always give the correct answer, which is most unlikely.

2.1.7 Performance measures

After the model has been trained using the training set and the hyper parameters have been optimized, the only remaining step is to evaluate the performance. This is something that is done at the absolute end of the machine learning process, where we then evaluate the performance of our best model by letting it predict the answer on the still not used test set.

When working with classification problems, each example has a numeric label $y = i \in (0, 1, ..., k-1)$, which means that it belongs to class i out of the k classes. If we only consider a binary classification problem, i.e. where there are only two classes, the possible numeric labels are either 0 or 1. It is common in binary classification problems that we want to know whether an example belongs to a certain class or not. The examples that belong to the class are usually labelled positive with the numeric label $y = 1$, and the rest are labelled negative with the numeric label $y = 0$.

So the goal of our model will then be to predict the numeric label. But the models often have a continuous output value $a \in [0, 1]$ that we have to threshold in order to acquire a label $\hat{y}$. The threshold value is commonly chosen as 0.5, i.e. an output $a \geq 0.5$ will result in a label $\hat{y} = 1$, but can be chosen be any value between 0 and 1 depending how we value finding the two different classes.

After having decided the threshold value, we can start talking about the four possible outcomes from the binary classifier. If we have $y = 1$ and $\hat{y} = 1$, then it is called true positive (TP). However if $y = 0$ and $\hat{y} = 1$, it is called false positive (FP). We have a matching case for true negative (TN) and false negative (FN). From these possible outcomes we can calculate the accuracy

of the model as

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} = \frac{\#Correct}{\#Examples}$$

Note that we the labels as the number of examples with the specific outcome. So TP in the equation is the number of examples with the outcome true positive.

The problem with accuracy, is that it does not take into account if the tested data set is unbalanced, i.e. one class having many more examples than the other. Take for example if we have a data set where we have 990 negative examples but only 10 positive examples. A good way to get high accuracy would then be to just let our model classify every example as negative, as that would give an accuracy of 99%. But that would not be considered to be a good classifier, as there are no actual dependence on the specific example. This may be an extreme case, but there are many situations in which it is hard to collect more examples, e.g. patients with a certain rare disease.

One way to solve this problem is to use another measurement. The most common measure that to some extent solves it is the F_1 score. To calculate the F_1 score, we first have to calculate two other values. The first one is the true positive rate (TPR), also known as *sensitivity* or *recall*, which is calculated as

$$TPR = \frac{TP}{TP + FN}$$

This value, which is bounded to the interval $[0, 1]$, can be seen as a measure of how well the classifier finds the positive examples. The second value is the positive predictive value (PPV), also known as *precision*, which is calculated as

$$PPV = \frac{TP}{TP + FP}$$

The PPV can be seen as a measure of how well the classifier can classify examples as positive. Lastly the F_1 score is then calculated as

$$F_1 = 2 \cdot \frac{TPR \cdot PPV}{TPR + PPV}$$

The F_1 score is the harmonic mean of TPR and PPV, and can be interpreted as a weighted average of the precision and recall. From the definition, we can see that is bounded in the interval $[0, 1]$, where 1 corresponds to the best value. It works much better with unbalanced data sets, though it is skewed

towards the positive examples, as both the TPR and PPV are calculated with the positive examples in focus. If we redo the previous example of 990 negatives and 10 positives, but measure the performance with F_1 score instead, we get a value of 0.

Both the accuracy and the F_1 score is dependent on the threshold value that we use to classify each example. As mentioned before, this threshold is commonly chosen as 0.5, but is a hyper parameter that can be chosen to be anything between 0 and 1. This hyper parameter is different from many other hyper parameters in that it does not affect the training or shape of the model at all, but instead affects the performance measurements. The choice of threshold affects the chance of finding examples of one class, which leads to less risk of missing them, but also increases the risk of missing the other class. This is something that is more important to some fields, e.g. in health care. When screening patients for cancer, one could argue that it is more important that we find all cases of cancer at the risk of maybe diagnosing healthy patients as having cancer, which would correspond to lowering the threshold. If we instead increased the threshold, this would correspond to requiring more certainty in our prediction before diagnosing a patient with cancer, at the risk of missing patients that actually have cancer. In this example, both cases have their arguments, but it's hard to know what may be better. So for this type of problem, one may use something called Receiver Operating Characteristic, or a ROC curve, see Figure 2.8 for an example.

It is created by calculating the TPR and the false positive rate (FPR), also known as $1 - \text{specificity}$,

$$FPR = \frac{FP}{TN + FP}$$

for several values of thresholds and plotting each resulting values of TPR and FPR in a graph. The FPR value can be seen as a measurement of how often it falsely classifies an examples as positive when it is actually negative. So having a low value of FPR means that the classifier rarely start false alarms of the positive class.

So all together, the ROC curve can be used as a way of finding optimal values of the threshold that achieves the desired TPR value while keeping the FPR value low. In Figure 2.8, we see an example of a random classifier, which follows the diagonal line as it will give on average the same number of correctly classified examples as incorrect ones. The figure also shows a good classifier which achieves a high TPR value while still keeping the FPR value

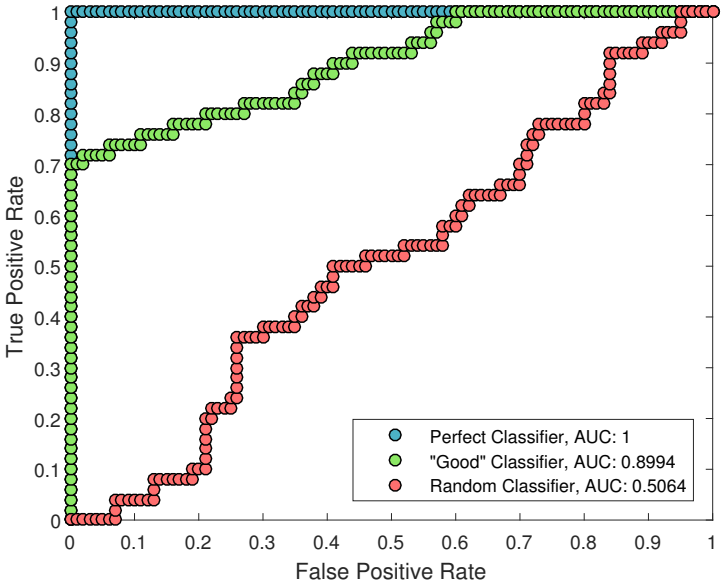


Figure 2.8: Example of three different ROC curves. The blue one is an example of a perfect classifier where each example is correctly classified no matter the threshold, except the at 0 and 1. The red one is an example of a random classifier, in which given an example, will randomly classify it as positive or negative. The green one is an example of a good classifier where it achieves good TPR while still having low FPR.

low.

When working with machine learning algorithms, it is often desirable to have one single measure of the performance, as it makes the comparison between two models easier. In this aspect, the ROC curve can be a bit unwieldy to handle, as it is hard to compare two different ROC curves [14]. For this, there is a measurement that can be derived from a ROC curve, called the area under the curve, also known as AUC or AUROC. It is given by calculating the area under the ROC curve, hence the name. It can be seen as the expectation of the classifier to rank a randomly chosen positive example higher than a random negative example.

2.2 Artificial Neural Network

2.2.1 Introduction

As mentioned in the introduction of this report, artificial neural networks (ANNs) represent a type of machine learning algorithm that were introduced in 1943 [34]. At first, the goal was to model the biological neural system in our brain, but the research later split into two different fields. One of them focuses on understanding the actual biological process in the brain, while the other focuses on the practical application of the ANNs, which is the focus of this thesis.

In the beginning the focus was on only using one artificial neuron to compute the answer to a problem, but if we compare it to our brain, we have billions of neurons that do all the processing. Therefore additional neurons were soon added to the single neuron to create an ANN as to replicate the structure in our brain. These early networks were so called one layer networks, with one input layer and one output layer, which results in one set of weights that the network could use. The reason they are called one layer networks will be explained in Section 2.2.4.

The one layer networks were trained using simple algorithms, such as just weighting the error between the true answer and the predicted answer and updating the the weights depending on if the predicted answer was correct. Using this method, one could only train one layer of weights, which explains why they were only using one layer networks, as the input layer has no associated weights and therefore no need to be trained. Even though this seriously limited the capabilities of the ANN, they were still very useful at that time. Several other ways to train networks were proposed, where there were some that made it possible to train more layers, they were not very effective and would often not converge to good values. As other machine learning algorithms were invented that had better performance given the available computer power and amount of data at that time, neural networks soon fell out of favour.

In 1974, a method that would enable training of ANNs with multiple layers would be introduced in [48], though it was not until more than a decade later, in 1986, that this method was popularized through [7]. This method is called backpropagation, a short hand for “backward propagation of errors”, and is still used to train virtually every feedforward ANN today. Backpropagation enabled ANNs to be trained more efficiently, which led to a

resurgence of research within the field of neural networks. But even though the networks could be trained faster, it also had its problems that had to be solved. This problem had its source in the increasing number of layers in the ANNs, which led to a new limit on the structure that could be used when building neural networks. And once again the networks soon fell out of favour as other machine learning algorithms were invented, this time the spotlight went to e.g. Support Vector Machine and Random Forests.

It took another decade for ANNs to once again reach the spotlight and gain the attention of a broader machine learning community. This time it was with a semi-supervised neural network, called Deep Belief Nets [23], which proved that neural networks indeed can be trained well as long as the weights are initialized in a clever way rather than randomly. Using this new way of training networks, the inventors were able to beat state-of-the-art performance record in numerous areas. In addition to showing good results, they also rebranded the field of neural networks into what is today widely known as deep learning [1], as the name neural networks had negative connotations at that time. This time the following is stronger than ever and deep learning algorithms are used in increasingly more varied fields, where they achieve previous state-of-the-art performances and often exceed them. But this revolution in the machine learning field is not only because of the many algorithms that have and that are still being invented. A big part of the uprising is due to the increasing computer power and the explosion of data, also called the rise of Big Data, that has come with the era of Internet.

The coming sections will function as an introduction to the field of ANNs, where we will go through the various parts of an ANN and some important associated concepts. Even though the ANNs used today are moving increasingly farther away from the original inspiration, we will start in Section 2.2.2 by looking at how the artificial neuron approximates the biological neuron as a mathematical model. This is to give an better understanding of the shape of the artificial neuron, and may also be used as a possible explanation of why it works. In the following section, Section 2.2.3, we present a more thorough look on the different parts that make up an artificial neuron; how they are used and what function they have for the computation capabilities of a neuron.

The next step after having built one neuron is to connect several of them together to create a neural network, After having built one neuron, we can connect several of them together to create a neural network, which has been shown to be able to be able to approximate any function, i.e. they work

as universal approximators [25]. Some important concepts associated with neural networks will be presented in Section 2.2.4. But even though the most basic neural networks can work as universal approximators, it is when you actually build deeper networks with more layers of neurons that enables the state-of-the-art performance we see today. This concept is known as deep learning and will also be covered in Section 2.2.4.

In Section 2.2.5, we will continue to build upon the training concepts presented in Section 2.1.6. This section will cover several important additions to the standard training phase that are often used in ANNs. The important method that enabled the training of neural networks with multiple layers, i.e. backpropagation, will also be presented. Afterwards a short introduction to regularization in ANNs together with a regularization technique unique to neural networks, called dropout. The section, and also the introduction of ANNs, will then be concluded by introducing some important concepts and quantities to track while training neural networks.

2.2.2 Biological Connection

The biological super computer that we all have in our head, i.e. the brain, is made up of about 86 billion neurons [3]. These neurons are the basic computational units that process all the information acquired from our senses, which is similar to how the processors in our computers are made up of transistors that process all the data. All these neurons are connected to each other through about $10^{14} - 10^{15}$ synaptic connections in a complex structure that we still do not fully understand [16].

There are many different kinds of neurons in our brain that have their own special properties, but we only consider the most basic neuron with the structure that is shared between all the various kinds of neurons. In Figure 2.9, we can see a cartoon drawing of the basic biological neuron on which we will base our mathematical model, the artificial neuron, which can be seen in Figure 2.10.

The main parts of a biological neuron is the cell body that has a nucleus, small branches, called dendrites, and one bigger branch, called the axon. Signals from other neurons are passed to the cell body through sites on the dendrites called synapses, which are represented in the mathematical model as the input data, e.g. x_1, x_2, x_3, to the artificial neuron. The signals are either strengthened or weakened in the dendrites or synapses before being processed by the neuron, which correspond to the input data being weighted by the

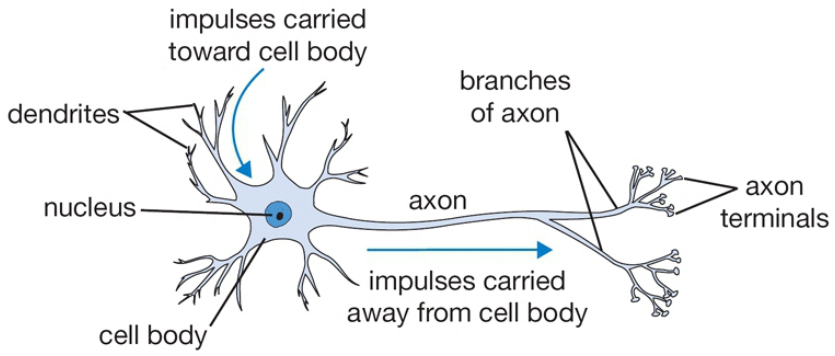


Figure 2.9: A cartoon drawing of a basic biological neuron with the important parts labelled[1].

associated weights in our artificial neuron. As mentioned in the machine learning section, what this weighting process does is to put emphasize on specific signals while ignoring others. So when the signals are then summed in the neuron, regardless if it is biological or artificial, it tries to detect a specific pattern in the signals.

For the biological neuron, if the specific pattern is strong enough in the signals, the neuron will transform the signal through a complex chemical process. The signal is then sent through the axon which is connected with dendrites of other neurons and the whole signal processing is repeated. But if the pattern is not found in the signal, nothing more will happen and no signal will be transferred to the other neurons. In the case of the artificial neuron, to mirror the complex transformation of the signal, an additional bias term b is added to the sum of signals before being processed using a non-linear function f called the activation function. Traditionally the sigmoid function

$$f_{sigmoid}(x) = \frac{1}{1 + e^{-x}} \tag{2.1}$$

has been used as f to replicate the all-or-nothing property of the biological neuron, though it has mostly been replaced by other functions, which will be further discussed in Section 2.2.3. The resulting value z from the activation function is sent to the other connected neurons where the whole process is repeated. Note that the mathematical model is a very coarse reproduction of the biological neuron, where many important aspects have been ignored.

[1]Source: http://www.wpclipart.com/medical/anatomy/cells/neuron/neuron.png.html

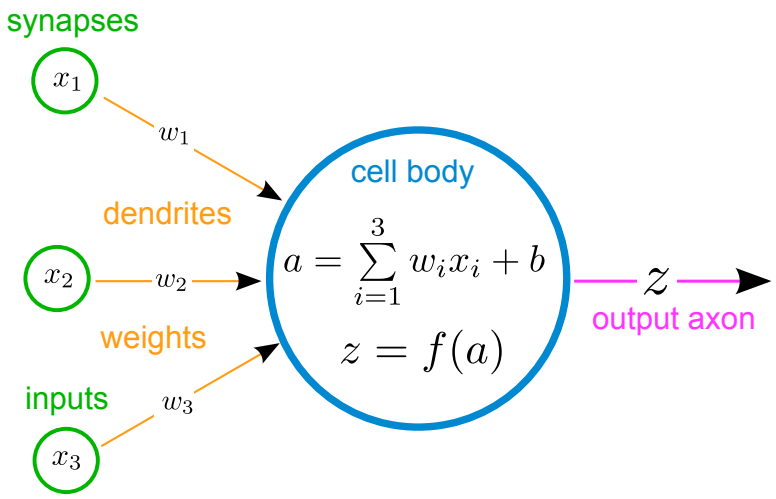


Figure 2.10: A mathematical model of a biological neuron.

This includes the various types of neurons, and simplifications of the signal processing done in the synapses and dendrites [29].

What we described above was the procedure in which a neuron processes signals to send a resulting signal, but this requires that the neuron has already learnt what patterns to look for. A big part of the foundation which is used to understand how the brain, and thereby the neurons, learn in a physiological sense is based on the findings in [21]. In this book, the author put forth the idea that knowledge and learning occurs in the brain primarily through the formation and change of synapses between neurons, which is stated in the book as

> *When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased.*

This means that as the signals from neuron A keep exciting the neuron B to also send signals, the connection between them will be getting stronger. Because this is what enables the brain to learn with experience, as long as we can replicate this process well enough in our artificial neurons, it should be possible for ANNs to learn too. This is what we try to do when we train an ANN, which will be further discussed in Section 2.2.5.

2.2.3 Building Blocks of an Artificial Neuron

An artificial neuron can be seen as a transfer function from the input x to an output z, see Figure 2.10. This transfer function can be divided into two separate steps, where the first step is to multiply each input x_i with its associated weight w_i and then sum them up together with a bias b to acquire the intermediate value a. The weights can be seen as representing how much each input affects the output of the neuron, either negatively or positively. As an example, assuming the input has been rescaled into the $[0, 1]$ interval, if a neuron had the weights $w_1 = 10, w_2 = 1, w_3 = -10$, then that would mean that the presence of x_1 affects the output positively much more than x_2. x_3 on the other hand, would instead inhibit the intermediate signal a. Before discussing the role of the bias b, we first take a look at the second step of the transfer function, which is the activation function.

The activation function was introduced in Section 2.2.2 as a way to replicate the complex transformation of the input signal after it had been weighted and summed in the biological neuron. But another reason that we use an activation function is to augment the output of the neuron with a non-linear function. Without the activation function, the transfer function of the neuron would only consist of linear parts. There would be no reason to connect several neurons together as the linear function of each neuron could just be replaced with one neuron, like how several transformation matrices can be replaced with one transformation matrix.

As mentioned before, traditionally the sigmoid function, see Equation 2.1, has been used as the default activation function. This is because it mimics the biological neuron all-or-nothing property by either outputting the vales 0 or 1, with some intermediate values between to make it differentiable. But it has fallen out of favour in the ANN community because of a problem that arises when training the neuron. To understand this problem we start by looking at the shape of the sigmoid function in Figure 2.11.

The sigmoid function saturates at the ends with the values 0 and 1 respectively. This means that for values outside the small region around 0 the derivative will be small, which can be confirmed when looking at the derivative of the sigmoid function

$$f'_{sigmoid}(x) = f_{sigmoid}(x)(1 - f_{sigmoid}(x))$$

and can be seen plotted in Figure 2.12.

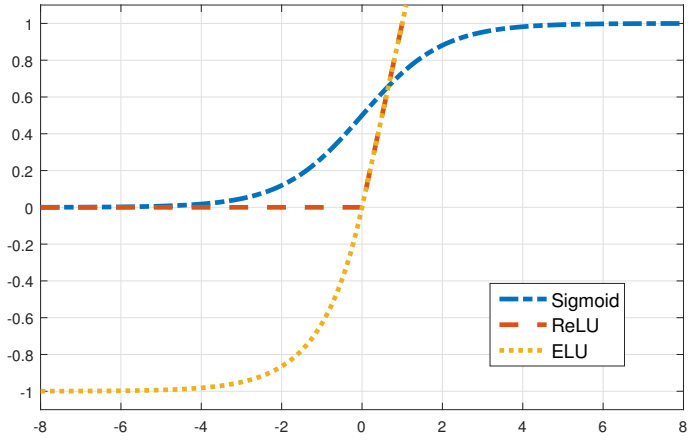


Figure 2.11: The activation functions sigmoid, ReLU and ELU with $\alpha = 1$ plotted in the same graph.

Another way to see it, is to note that the sigmoid function maps the all the real numbers onto a "small" interval of $[0, 1]$ in a non-linear way. This non-linear compression results in large regions where changes in the input give rise to small changes in the output. As an extreme example, if we input the value 1000 into the sigmoid function, the output will practically be equal to 1, but if we change the input into 20, the output will be around 0.99999999794, which is pretty much equal to 1. From this example we can see that the derivative in these regions are extremely small. So when we try to find the optimal weights for the neuron using the gradient descent algorithm given a loss function, see Section 2.1.6, we have the problem that if the output is at one of the ends, the gradient will be almost zero. This means that each update of the weights would change almost nothing, i.e. for each new step, we move almost nowhere in the optimization landscape. This problem is commonly known as the saturated gradient problem. Increasing the learning rate η would not help either, as this would lead to problems when the input is close to 0.

A solution to the saturated gradient problem is to use another activation function that does not compress the input. The most common activation function used today is the Rectified Linear Unit (ReLU) [37], which has the

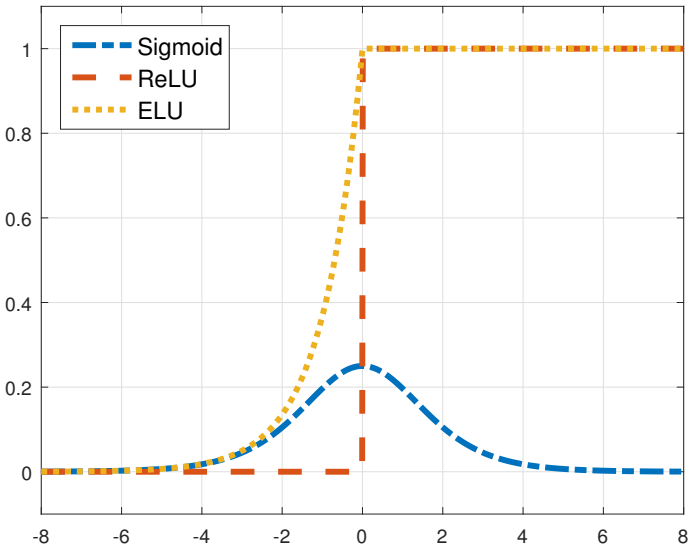


Figure 2.12: The derivative of the activation functions sigmoid, ReLU and ELU with $\alpha = 1$ plotted in the same graph.

form and derivative

$$f_{ReLU}(x) = \max(0, x), \quad f'_{ReLU}(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$$

These have been plotted in Figure 2.11 and 2.12, respectively. This activation function is very different from the sigmoid function, in that ReLU simply thresholds the input at 0. Although the ReLU function is technically not differentiable in $x = 0$, this is rarely a problem as the output from the first step is seldom exactly 0 and for the rare cases, the derivative is just set at 1. There are several benefits to using ReLU as activation function. The main benefit is that the ReLU does not suffer from the saturated gradient problem as it is always either 0 or 1. Although there is a problem if the weights are initialized or updated such that the intermediate value a always become negative, the gradient will always be 0 and will never get updated. These neurons are therefore considered "dead", and in a trained neural network, there may be a significant number that have died and are just wasting space. But in practice, this is considered to be a minor problem, and the use of

ReLUs have been shown to greatly accelerate the training of a neural network compared to when using the sigmoid function, in some cases by a factor 6 [32].

Another activation function, which can be seen as an extension of the ReLU, is the Exponential Linear Unit (ELU). The ELU has the form and derivative

$$f_{ELU}(x) = \begin{cases} \alpha(e^x - 1) & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}, \quad f'_{ELU}(x) = \begin{cases} f_{ELU}(x) + \alpha & \text{if } x \leq 0 \\ 1 & \text{if } x > 0 \end{cases}$$

and can be seen in Figure 2.11 and Figure 2.12, respectively. As can be seen, the ELU adds a function for the cases where $x < 0$, which means that it to some extent solves the problem of dying neurons caused by the ReLU. There are also other benefits to using ELU which lead to faster training, though these are too technical for this report and interested readers may read [12].

Going back to the linear part of the transfer function, i.e. the first step, we can now explain what role the bias term b has. It has no direct effect on the individual inputs, but is instead used to shift the intermediate value a into a suitable range before processing it through the activation function. Another way to see it is that the bias term changes the "break point" for the activation function. As an example, take the ReLU where we have the split at 0, this means that the weighted inputs have to sum up to over 0, i.e.

$$\sum_i w_i x_i > 0$$

to make the neuron send a signal, otherwise only a 0 is sent, which is pretty much equivalent to no signal. But if we add the bias term b, we get

$$\sum_i w_i x_i + b > 0 \quad \Leftrightarrow \quad \sum_i w_i x_i > -b$$

i.e. the sum of weighted inputs has to overcome the negative bias. This means that when we add a positive bias, we are essentially moving the graphs in Figure 2.11 and Figure 2.12 to the left, while a negative bias does the opposite. Note that when training a neural network, not only are the weights optimized, but also the bias term.

2.2.4 Network of Neurons

Since one single neuron working by itself is only capable of doing simple calculations, it is only able to solve simple problems. One way to increase

the computational power is to connect the neurons together and create a neural network, which are sometimes called MultiLayer Perceptrons (MLP) for historical reasons. Since a neuron can be seen as a computational unit with input and output connections, neural networks are often visualized as a graph where each node represents one neuron and the edges represent the weights, see Figure 2.13.

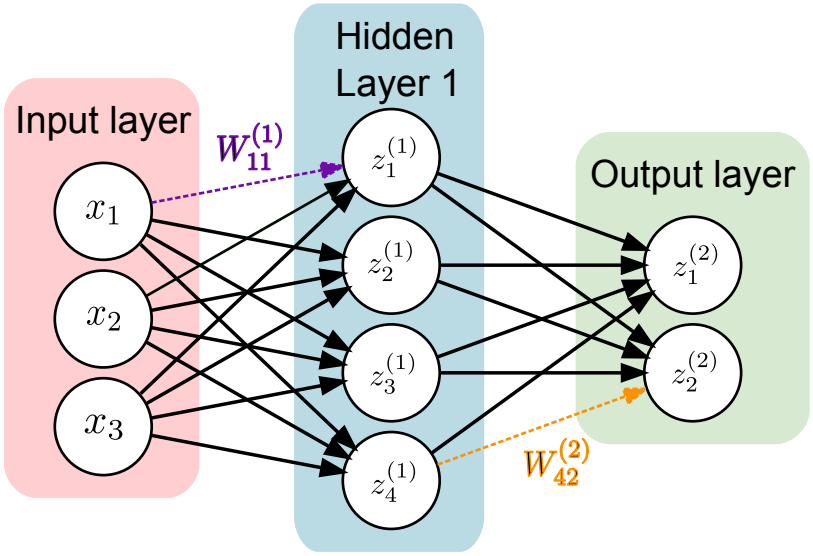


Figure 2.13: Example of a two layer neural network with some weights marked to show an example of the notation. The variable in each node represents the output of the node.

The most common kind of ANN is feedforward networks, which mean that they can be represented as acyclic graphs, which in turn means that all the edges go in the same direction and no cycles are allowed in the graph. In these feedforward networks, the input data is propagated forward from the input neurons to the output neurons with no neurons processing the same data more than once. There is another type of networks that allow for cyclic graphs, where the neurons may process the same signal multiple times. These networks are called recurrent neural networks, which are often used when the problem is connected to time in some way, e.g. time-series analysis or video analysis. The main focus of the thesis will be on feedforward networks, as the image classification problem does not have any dependence on previous

inputs or any time dependence.

Layer-wise configuration

Neural networks are often organized into distinct layers of neurons, and depending on the type of layer, the neurons in one layer will connect differently to the neurons in the next layer. When all the neurons in one layer is connected to all the neurons in the previous layer, that layer is called a fully connected layer. The first layer in a neural networks is called the input layer as this is the layer from which the data enters the network. The nodes in this layer are not really considered to be neurons as they have no associated weights and do not do any computations; instead they just represent each feature in the data example. Therefore the number of neurons in the input layer is determined by the number of features that is used in each data example.

The following layers after the input layer, except the last layer, i.e. the middle layers, are called hidden layers. There are many different possible reasons why the middle layers are called hidden layers. One possible reason is that when using a trained neural network, the outputs from these layers are not known as they are "hidden" in between the first layer and the last layer. These hidden layers hold the standard neurons that we have presented in earlier sections. After the hidden layers, we have the last layer which is called the output layer.

Just as in the hidden layers, the output layer uses neurons too, though we usually do not use any activation functions, which can also be seen as using the identity function $f(x) = x$ as the activation function. This is because the we usually want to interpret the output without it being processed through a non-linear function. For instance, the scores from the output can be interpreted as class scores in a classification problem where the example is given the class with the highest score. But in some cases, to improve the interpretation in classification problems, the output is processed with a softmax function during prediction. The softmax function, which is a vector-valued function, with the form

$$f_{Softmax}(\mathbf{z}) = \frac{(e^{z_1}, e^{z_2}, \ldots, e^{z_n})}{\sum_i e^{z_i}}, \quad \mathbf{z} \in \mathbf{R}^n$$

and has the property that it squashes each value in the vector $\mathbf{z}$ into the interval $[0, 1]$ such that the sum of all the output values is equal to 1. It

can be seen as a generalization of the sigmoid function, which squashed one value into the $[0, 1]$ range, to a higher dimension. In this way, each value can be seen as a probability that the associated class is correct according to the networks, or in other words, how sure the network is that the given example belongs to each class. Note that a standard convention when counting the number of layers in a neural network is to not count the input layer, as it does not do any computations. So a two layer network means a network with one input layer, one hidden layer and one output layer.

It has been shown that a two layer ANN are universal approximators. This means that for any continuous function $f(x)$ and some $\epsilon > 0$, there exists a two layer neural network with transfer function $g(x)$ such that $\forall x, |f(x) - g(x)| < \epsilon$. In other words, the neural network can approximate any continuous function.

When setting up a neural network, the size of the input layer and the output layers are already determined. This is because the input layer is given by the number of features in each example, while the output layer is determined by the number of classes. Therefore the structure of the hidden layers, such as the number of layers and the number of neurons in each layer, is what counts as hyper parameters that have to be optimized when setting up a neural network. It is important to note that, just as in any other hyper parameter optimization, changing the structure of the network means that the whole network has to be retrained from scratch.

To get a better understanding of how the computation works in an ANN, we will show the how the calculations are done to get the output $z_2^{(2)}$ in Figure 2.13. But before we can start the computation, we have to explain the notation used. The vector $\mathbf{x} = (x_1, \ldots, x_m)$ represents one example which consists of the m features, where $m = 3$ in this case. The hidden layer will use the activation function f_1 to get the vector $\mathbf{z}^{(1)} = (z_1^{(1)}, \ldots, z_{n_1}^{(1)})$ as output where $n_1 = 4$ is the number of neurons in the first hidden layer. The output layer will use the activation function f_2 to get the vector $\mathbf{z}^{(2)} = (z_1^{(2)}, \ldots, z_{n_{output}}^{(2)})$ as output where $n_2 = 2$ is the number of neurons in the output layer. The input layer is connected to the hidden layer through the weight matrix $W^{(1)} \in \mathbf{R}^{m \times n_1}$ and the hidden layer is in turn connected to the output layer through the weight matrix $W^{(2)} \in \mathbf{R}^{n_1 \times n_2}$. With these notations, we can now calculate the output $z^{(2)}$ in two steps. First we calculate the output

from the hidden layer

$$z_1^{(1)} = f_1\left(\sum_{i=1}^{3} W_{i1}^{(1)} x_i + b_1\right)$$
$$z_2^{(1)} = f_1\left(\sum_{i=1}^{3} W_{i2}^{(1)} x_i + b_2\right)$$
$$z_3^{(1)} = f_1\left(\sum_{i=1}^{3} W_{i3}^{(1)} x_i + b_3\right)$$
$$z_4^{(1)} = f_1\left(\sum_{i=1}^{3} W_{i4}^{(1)} x_i + b_4\right)$$

and from which we calculate

$$z_1^{(2)} = f_2\left(\sum_{i=1}^{4} W_{i1}^{(2)} z_i^{(1)} + b_2\right) = f_2\left(\sum_{i=1}^{4} W_{i1}^{(2)} f_1\left(\sum_{j=1}^{3} W_{j1}^{(1)} x_j + b_1\right) + b_2\right)$$
$$z_2^{(2)} = f_2\left(\sum_{i=1}^{4} W_{i2}^{(2)} z_i^{(1)} + b_2\right)$$

These equations can be shortened if we use the vector and matrix notations

$$\mathbf{z}^{(1)} = f_1(\mathbf{x} \cdot W^{(1)}) \quad \Rightarrow \quad \mathbf{z}^{(2)} = f_2(\mathbf{z}^{(1)} \cdot W^{(2)})$$

assuming that the activation functions f_1 and f_2 works on the matrices element-wise.

To calculate the number of parameters that can be trained in a neural network, we just have to calculate the number of elements in the weights and add the number of neurons in each hidden layer and output layer. In the ANN in Figure 2.13, we have $3 \cdot 4 + 4 = 16$ parameters for the hidden layer and $4 \cdot 2 + 2 = 10$ parameters for the output layer, which results in a total of 26 parameters in the whole network. Note that this is a very simple network with only a few neurons in two layers. Although it is not common, some neural networks have hundreds of input features and a hidden layer that has about 1000 neurons. The more common situation is that the networks have more hidden layers, which leads to the field of deep learning.

Deep Learning

As mentioned earlier, the name deep learning was at the beginning mostly just a rebranding of the field of ANNs, though nowadays it is considered by the machine learning community to be an actual subfield of ANNs. Deep learning is defined as the use of neural networks that has more than one hidden layer. These networks are called deep networks, while the ones with only one or no hidden layers are considered to be shallow networks. But as it has already been shown that 2 layer neural networks can work as universal approximators, there is seemingly no reason to go any deeper. But one has to keep in mind that it is just a theoretical proof which makes assumptions that are unreasonable in practice, e.g. it require "astronomical number of terms" [25]. So even though deep networks theoretically have the same representational power as the 2 layer networks, they have empirically been shown to perform better with the same number of neurons [5].

The reason that deep networks are more effective than shallow networks is that the deeper structure enables the networks to learn higher levels of structures in the data; they can learn hierarchical feature representations of the data. This means that instead of only learning features that are combinations of the input data, deeper networks allow the next layer to learn features that are a combination of the previous features. As an analogy, when we read, we begin by first understanding what each word means, which can be seen as the first hidden layer processing the data. We then base our understanding of these words to combine them into sentences, like how the next hidden layer create features by combining the features from the previous layer. This structure then continues as we combine the sentences into paragraphs, which in turn are combined into articles or books and so on. In the end, we can say that we have understood what we have read, which can be seen as the output from the neural network. A shallow network would then correspond to us trying to understand a book by just looking at all the possible combinations of the words within the book. It is theoretically possible, but extremely impractical.

2.2.5 Training

Before we can start training our neural network, we have to initialize the weights to some starting values. One may think that this has little impact on the training, as our optimization algorithms will step away from the starting

point anyway, but it has been shown to affect both the speed and performance of the training [24]. According to [17], a good way to initialize the weights is to sample the weights from a zero-mean unit variance Gaussian distribution and multiply each weight with

$$\sqrt{\frac{2}{n_{in} + n_{out}}}$$

where n_{in} and n_{out} are the number of neurons in the previous and next layer, respectively.

When training a neural network, the most common optimization algorithm is gradient descent, which has been covered in Section 2.1.6. To be able to use gradient descent, we first have to decide on a cost function $l(\mathbf{x}_i, y_i; W)$ to measure how well the network is performing on the current example. In classification problems, the cost function is often chosen to be a combination of the softmax function and the cross-entropy function

$$l(\mathbf{x}_i, y_i; W) = -\log\left(\frac{e^{z_{y_i}}}{\sum_j e^{z_j}}\right)$$

where z_{y_i} is the output of from the y_i-th neuron in the output layer and the sum in the denominator is over all the outputs from the output layer.

We will also make an addition to the gradient descent algorithm that is often used when training neural networks, called momentum update. This addition is inspired by the physical perspective of the optimization problem, i.e. the optimization landscape described in 2.1.6. It is explained by the fact that the landscape is made of hills and valleys, where our initialization of the settings can be seen as putting a ball somewhere in the landscape. As we train the network, the ball will roll down the hill into a valley, though the speed may sometimes be too high and result in the ball rolling out of the valley. This is partially solved by adding something akin to friction into the system. This is replicated in the gradient descent algorithm by adding a percentage of the previous gradients, which is done by changing the update step into

$$\begin{aligned} v_t &= \mu v_{t-1} - \eta \nabla L(X, Y; W_{t-1}) \\ W_t &= W_{t-1} + v_t \end{aligned}$$

where $v_0 = 0$ and μ is a hyper parameter often called momentum that is between 0 and 1 that says how much of the of the old gradients that should

be saved. Notice that even though the addition is called momentum update, as mentioned before, it is more like friction.

Backpropagation

As mentioned in Section 2.2.1, it was the backpropagation algorithm that enabled the training on multilayer neural networks. Although it had its problems, many of the problems have been solved and is therefore widely used to train neural networks together with the gradient descent algorithm. Note that we will present an outline of how the backpropagation works with focus on the weights, but the whole algorithm with all the details is quite involved and will not contribute much to the purpose of this report.

The backpropagation algorithm is used in many different fields outside of training neural networks, though in those cases it is more commonly known as reverse-mode differentiation [19]. It is used as a way of calculating the gradients of an expression through recursive application of the chain rule. Chain rule is a formula for computing the derivative of the composition of two or more functions. For instance if $f(x)$ and $g(x)$ are scalar functions, then the chain rule shows that the composition of the functions, $F(x) = f(g(x))$, can be calculated as

$$F'(x) = f'(g(x)) \cdot g'(x)$$

If we do the substitutions $y = g(x)$ and $z = F(x)$, then the chain rule can be expressed as an product of derivatives

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$

We can now start by working through the backpropagation algorithm and to make it easier to understand, we will do the backpropagation on the network in Figure 2.13. To simplify the notation, we will write the gradient using the operator $\frac{\partial}{\partial W}$ instead and also skip the W in the loss function. Our goal is to find the gradients

$$\frac{\partial L(X, Y)}{\partial W^{(1)}} \tag{2.2}$$

$$\frac{\partial L(X, Y)}{\partial W^{(2)}} \tag{2.3}$$

to be able to use gradient descent to update our weights. Recall that the matrix $X \in \mathbf{R}^{N \times m}$ holds one example of m features in every row with N being

the number of examples in each batch. Y is a N length column vector with the corresponding class label in each row. We will begin by first calculating the gradient in (2.3) as it is "closest" to the loss function and therefore easier to calculate. Assume that we have already propagated the input data and therefore acquired a value for the loss using the value $z^{(2)}$ from the output layer

$$L(X, Y) = L(z^{(2)}) = L$$

We can therefore apply the chain rule to (2.3) as

$$\frac{\partial L(z^{(2)})}{\partial W^{(2)}} = \frac{\partial L(z^{(2)})}{\partial z^{(2)}} \cdot \frac{\partial z^{(2)}}{\partial W^{(2)}}$$

We can calculate the first factor as long as we have chosen a differentiable cost function. To calculate the second factor, recall that $z^{(2)} = f_2(a^{(2)})$, where f_2 is the identity function, together with the intermediate value $a^{(2)} = z^{(1)}W^{(2)}$. This lets us rewrite the second term as

$$\frac{\partial z^{(2)}}{\partial W^{(2)}} = \frac{\partial f_2(a^{(2)})}{\partial a^{(2)}} \cdot \frac{\partial a^{(2)}}{\partial W^{(2)}} = \frac{\partial a^{(2)}}{\partial a^{(2)}} \cdot z^{(1)} = z^{(1)}$$

which now means that we have an expression for the gradient in (2.3) to update the weights $W^{(2)}$.

To calculate the gradient in (2.2) we start out in the same way as the previous gradient, which results in

$$\frac{\partial L(z^{(2)})}{\partial W^{(1)}} = \frac{\partial L(z^{(2)})}{\partial z^{(2)}} \cdot \frac{\partial a^{(2)}}{\partial W^{(1)}}$$

We then just have to continue using the chain rule

$$\frac{\partial a^{(2)}}{\partial W^{(1)}} = \frac{\partial a^{(2)}}{\partial z^{(1)}} \cdot \frac{\partial z^{(1)}}{\partial W^{(1)}}$$

Once again using $a^{(2)} = z^{(1)}W^{(2)}$ will give us $W^{(2)}$ as the first term, while using $z^{(1)} = f_1(a^{(1)})$, where f_1 is the activation function of the hidden layer, will give us

$$\frac{\partial z^{(1)}}{\partial W^{(1)}} = \frac{\partial f(a^{(1)})}{\partial a^{(1)}} \cdot \frac{\partial a^{(1)}}{\partial W^{(1)}} = \frac{\partial f(a^{(1)})}{\partial a^{(1)}} \cdot X$$

We now finally have expressions we can use to calculate the gradients in (2.2) and (2.3) after having forward propagated one batch of examples.

$$\frac{\partial L(X, Y)}{\partial W^{(1)}} = X \cdot W^{(2)} \cdot \frac{\partial f(a^{(1)})}{\partial a^{(1)}} \cdot \frac{\partial L(z^{(2)})}{\partial z^{(2)}}$$

$$\frac{\partial L(X,Y)}{\partial W^{(2)}} = z^{(1)} \cdot \frac{\partial L(z^{(2)})}{\partial z^{(2)}}$$

Note that in this example we have chosen to not bother with the dimensions of the various vectors and matrices. (The interested reader may derive the correct positioning of the matrices and vectors by going through the example once again while keeping in mind the dimensions.)

One way to interpret the backpropagation algorithm aside from the standard interpretation of the derivative, is that the error that was made in the output is split and assigned to the output neurons. This error is then split once again and assigned to the previous layer of neurons and so on; the error is backpropagated. The gradient descent then uses the gradients that are calculated from the errors to update the weights. Weights that have contributed a lot to the mistake are changed more than the weights with low contribution.

Regularization techniques

Because neural networks have many parameters, in some cases billions of them, they are prone to overfitting as the number of examples seldom match the amount of parameters. It is therefore important to use regularization techniques to make sure that the ANN will be able to generalize well on unseen data.

The two regularization techniques, L_1 and L_2, that are presented in Section 2.1.6, are also commonly used in neural networks. But there are also some other regularization techniques that are unique to ANNs. The most common one is the dropout technique, which is a simple technique to execute, but has been shown to be an effective way of preventing overtraining [44]. It works during the training phase by only letting a neuron be active with some probability p, otherwise the neuron is "dropped", which essentially means that the associated weights are set to zero. This ensures that each neurons in one layer may not learn to depend too much on the signals from a specific neuron in the previous layer. When using the networks for actual predictions, the output from each neuron is then scaled with the corresponding p as to ensure that next layer of neurons get signals with about the same magnitude as during training. The probability p is a hyper parameter that is set individually for each layer, though in most cases the same p, often $p = 0.5$, is used in each layer. Note that it is uncommon to use dropout on the input layer, as this would result in too large information loss.

It is not fully understood why the dropout technique works so well, but the theory is that the network in each training step can be seen as a sort of "new" sub-network that is trained separately from the whole network. So when using the whole network to predict the answer to an example, it can be seen as an ensemble of networks in one network.

Training notes

In many other machine learning algorithms, one only has to iterate over the training set once to acquire a trained model. Neural networks differ in this regard as they usually require more than one iteration over the whole training set, where one iteration over the training set is referred to as one epoch. As previously discussed in Section 2.1.6, as a neural network is trained, it will learn the underlying structure of the training set as an approximation to the underlying structure of the real population. If the neural network is trained for too many epochs, it will also start to learn the inherent noise specific for the training set, which will lead to overfitting. This phenomenon is similar to the concept of model complexity, and the number of epochs is therefore often seen as a hyper parameter related to model complexity. The reason that we have to train ANNs for more than one epoch is due to the fact that the gradient descent algorithm may not have had enough time to converge to a minimum.

During training there are some metrics that are useful to track as a way to ascertain the performance of the current choice of hyper parameters. The most useful metric is to plot the loss during training as a function of the training time, which can either be in number of gradient descent steps or the number of epochs. Using this plot, we can see how the training is going, which is closely connected to the choice of learning rate η. Too low learning rates results in long training times to get good performance, while high learning rates hinder the network to reach a minimum and in some cases even lead to increasing loss. We have drawn sketches of what the various cases may look like, see Figure 2.14.

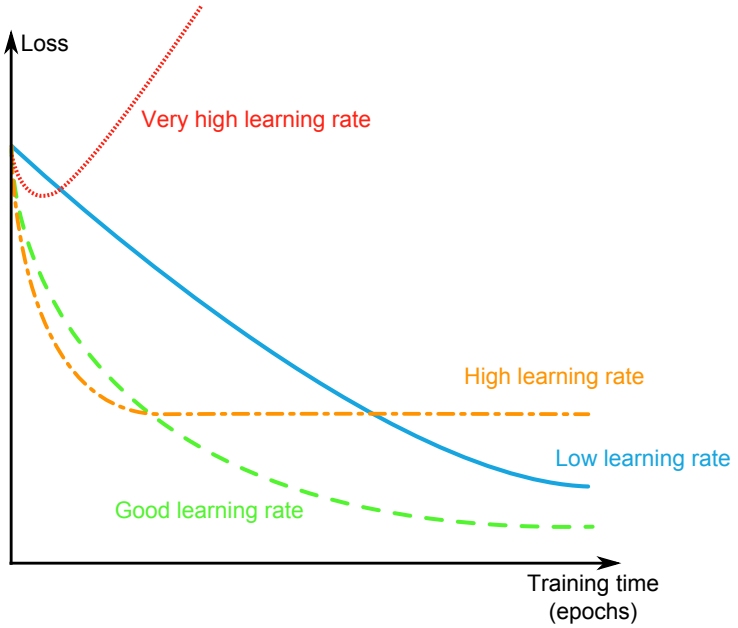


Figure 2.14: Some general sketches of what the plot of loss as a function of training time may look like with different choices of learning rate.

2.3 Convolutional Neural Network

2.3.1 Motivation

Ordinary ANNs work well when working with unorganized data, that is data that have features where there is no local structure to arrange them in, as these tend to be relatively few in number. But as soon as we start working with data that have a spatial grid structure, such as images, it very quickly gets unmanageable to use ordinary ANNs. As an example, say that we have RGB-colour images that have the size of 100×100 pixels. This would mean that the number of input nodes required to hold all the pixel values in one image would be $100 \cdot 100 \cdot 3 = 30,000$. This means that for each neuron in the first hidden layer, we would need $30,000$ weights to encompass the whole input layer, and we would most likely need many more neurons for the network to be able to analyse the image. Additionally, 100×100 images are not even considered to be big when working with ordinary photographs. This shows that ANNs do not scale well when used directly on images. One

possible way to solve it would be to preprocess and extract relevant features from each image and use those features to train the ANN instead of feeding the whole image into the network. The question is then how to choose, find and then extract good features from each image in a reproducible manner. This problem is also encountered in other machine learning algorithms, and almost always require expertise within the domain. So for instance, to extract good features from an X-ray image for classification of bone fractures, one has to consult a doctor about what kind of features that differentiate bone fractures from other possible complications. One then has to write a feature extractor to acquire these features before being able to train a machine learning algorithm. This process, called feature engineering, is usually a time-consuming process.

As a way to make ANNs scale better with images, we will be using a special type of ANNs called Convolutional Neural Networks (CNNs). What distinguishes CNNs from ordinary ANNs is that they use two extra kinds of layers in addition to the fully connected layer from ordinary ANNs. With the use of these two layers, the CNN is able to work directly on the data without having to use any feature engineering, as the CNN is able to learn what features it should look for by itself.

CNNs are most commonly used on images which have a 3D grid structure (if we count the colour channel as a third dimension), but also work with any other kind of data that has a grid structure, regardless of dimensionality. They for instance been successfully used for natural language processing tasks [13], which has 2D structure, and for video analysis [46], which can be seen as having a 4D structure. But in this thesis we will focus on describing CNNs in the context of images as input.

2.3.2 Convolutional Layer

The convolutional layer is the main building block in CNNs, which is what has given the CNN its name. In many ways it is similar to the fully connected layer, such as being made up of neurons that have trainable weights and usage of an activation functions to add non-linearity to the outputs. But at the same time, there are some fundamental differences between them that make the convolutional layer much more effective when used on data with a grid structure, most notably images. We will in the following sections go through the main features of the convolutional layer that distinguishes it from the fully connected layer.

Local connectivity

One of the main differences between a fully connected layer and a convolutional layer is in how their neurons are connected to the ones in the previous layer. For a fully connected layer, each neuron is naively connected to all of the neurons in the previous layer. This works well, and may be necessary, when working with data that have no spatial organization. But the pixels in natural images, i.e. images of objects in the real world, have an important grid structure where there is a spatial correlation between neighbours of pixels. For instance, if we know the colour of all the pixels around a specific pixel, that pixel is highly probable to have the same colour. The group of pixels then form natural hierarchical features, which can be seen as a group of pixels forming an edge, which in turn forms into a shape of an object when combined with other groups. This is what we are trying to exploit when we build CNNs; to have the network learn simple features in the first layer which are then combined to create more complex features. There is no need for a neuron to know all the pixel values, as the features in an image are often confined to a local region.

In a convolutional layer each neuron is instead made to only connect to a specific set of spatially local neurons in the previous layer. This means that each neuron is made to specialize on a small field of the input. The corresponding weights for the neuron are combined into what is called filters or kernels, which will be further described in the following sections. The reach of the neuron, i.e. the filter size, is usually called the receptive field and is a hyper parameter that is often chosen to be an odd value, which will further explained in a following section.

Spatial arrangement

Previously when working with ordinary ANNs, we have visualized the arrangement of the neurons in each layer as columns, see Figure 2.13, which naturally would have column vectors as output. But the convolutional layer works with data that have a grid structure, so therefore it is advantageous to also arrange the neurons in a similar grid configuration. Images can be seen as 3D matrices with the size $h \times w \times a$ where h and w are the height and width and a is the number of colour channels of the image. We also define our arrangement of the neurons in a similar way, where each layer of neurons is considered to be the ones on the same depth. This means that the output

of the convolutional layer, and therefore the shape of the neurons in the layer, will be a three dimensional matrix of size $m \times n \times d$. Here it is important to note that all the neurons along each depth column will be connected to the same local region. But even though we have the local connectivity property, that says that each neuron only connects to a local surface of the input volume, each neuron will still be connected with the whole depth of the input volume. This means that each neuron in the convolutional layer will have a three dimensional weight matrix with the size $m_f \times n_f \times c$ where (m_f, n_f) is the receptive field and c is the depth of the input volume to the layer.

The size of the output volume of a convolutional layer, and therefore the shape of neurons in the layer, depends on several hyper parameters. The first hyper parameter is the depth that decides the number of layers in in the output volume. The second hyper parameter is the receptive field (m_f, n_f), which is technically two different parameters, though they are in most cases chosen to be the same. The third hyper parameter is the stride, which decides the separation between each local region that the neurons are connected to. So a stride of 1 means that the local region moves with one unit for each neuron. The last hyper parameter is called zero-padding. In some cases the padding can be a real valued integer greater or equal to 0, but we choose to only see it as a boolean value that affects if the input volume should be padded with zeros along the first two dimensions. Without zero-padding, the neurons will not be able to reach all the way out to the border of the input volume as parts of the filter will fall outside. The output volume will therefore be spatially smaller than the input volume. To ensure that the output volume has the same height and width as the input volume one can add zeroes to the border of the input volume when doing the forward pass to ensure that the filter can reach the borders too.

Parameter sharing

Another important concept that distinguishes the convolutional layer from the fully connected layer is the concept of parameter sharing. It relies on the simple fact that if one of the neurons have acquired a set of weights that can compute an useful feature at some spatial position in the input, then it should also be useful to compute this feature at other positions. From this logic we make all the neurons on the same depth slice in the layer to use the same set of weights. Notice that because all of the neurons in one depth slice use the same weights, the output from these neurons can

be computed as a convolution of the weights with the input volume, hence the name convolutional layer. This is also the reason that the weights are commonly referred to as filters or kernels.

Using parameter sharing means that each neuron in the same layer will have the same filter to use to search after the same feature in the input volume, though at different spatial positions. This sharing of filters leads to a dramatic decrease in number of weights per neuron, but also has the additional positive effect of making the convolutional layer translation invariant.

2.3.3 Pooling layer

The pooling layer is a layer that does not have any weights to fit to the data. Instead it downsamples the input volume by using a filter with the max operation on each layer of the input volume. The filter is often chosen to be 2×2 with a stride of two, which results in halving the height and width of the input volume, see Figure 2.15. This means that 75% of the input volume will be discarded, which would seemingly lead to a major loss of data and therefore performance. But this is not the case, as it is argued that because of concepts used in convolutional layers, we have input volumes where each value in some sense encode the degree of which a certain feature was found in the previous input volume. Because each feature has been searched for in the whole previous input volume, as long as the feature was found, i.e. a high value in the current input volume, it should not matter too much where it is found within the filter size of the max pooling.

2	3	2	5
2	7	1	1
6	6	3	0
2	4	1	2

(a) Before max pooling

7	5
6	3

(b) After max pooling

Figure 2.15: Example of max pooling using a 2x2 filter.

The reason that max pooling is used in CNNs is to reduce the number of parameters and computations needed in the network. This will also have the effect of controlling overfitting, as fewer parameters means lower model complexity.

2.4 Data Material

All the data that is used in this thesis has been provided by Exini Diagnostics AB, Lund, Sweden. The data has been collected from 2164 patients over the period of January 1999 to March 2010. The selection criteria was prostate cancer patients who had undergone whole-body scintigraphy, because of suspected bone metastatic disease. The radiopharmaceutical 600 MBq Tc-99m methylene diphosphonate (MDP) was used together with the dual detector gamma camera Maxxus, General Electric.

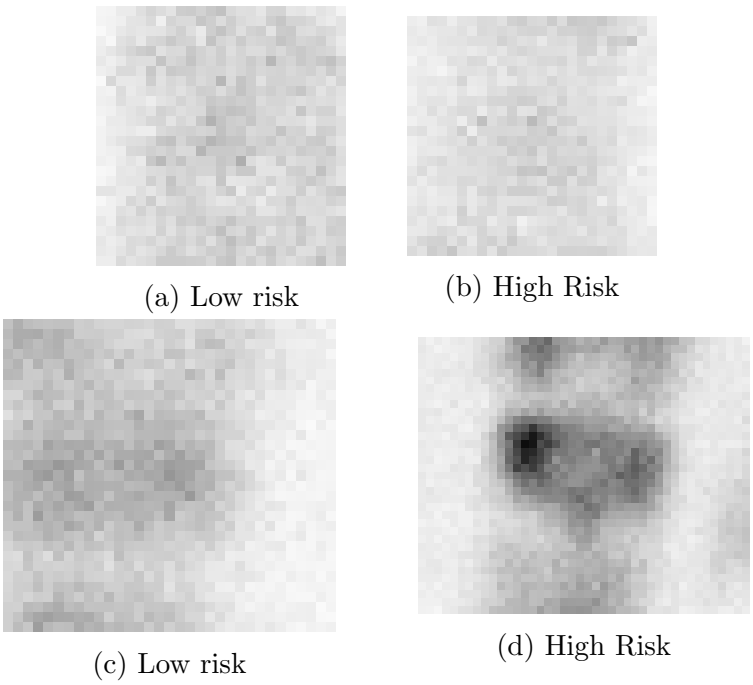
2.4.1 Main Task

Recall that the main task was to classify whether hotspots in Bone Scans have a high or low risk of being a cancer metastasis from prostate cancer, see Section 1.3 for more information. The hotspots were segmented, cropped and collected from Bone Scans using programs developed at Exini. We chose to only focus on hotspots that came from the lower spine, because we believed that these hotspots would be the easiest to classify. This is because the hotspots in the lower spine should have small size with low variance and have a high degree of symmetry. This turned out to be partially true when we started analysing the data set. As can be seen in Figure 2.16, the patches are extremely vague and it is hard to tell one class apart from the other.

The data set was divided into two different classes. Note that due to some quirks in how the data was loaded into the CNN in Keras, the "high risk"class became the negative class while the "low risk" class became the positive class. This is opposite to how the classes are commonly assigned, where the presence of an illness is assigned to be the positive class, but this switch does not affect the interpretation of the results in any significant way.

The data set consisted of a total of 10428 examples where 3170 of the examples belonged to the "high risk" class, while the remaining 7258 examples belonged to the "low risk" class. The data set took in total 89.1 MB of hard drive space.

It is important to note, as mentioned in Section 1.1.4, that the process of determining the label manually is a difficult task that is often to a certain degree subjective. Therefore there is some variability in the ground truth labels as this specific set reflect the opinion of the few experts that were consulted when creating this data set. For this reason, the goal may not be to reach 100% accuracy with our classifier, as that would just mean that the



(a) Low risk (b) High Risk

(c) Low risk (d) High Risk

Figure 2.16: Examples of patches of hotspots paired with their corresponding true label. Notice that the images have been approximately scaled according to their real image size.

classifier has learned to make the same judgement as the experts behind the labels. This does not mean that it is meaningless to train a classifier, as the majority of examples are still agreed by many to be correctly labelled. It is only the few hard cases that are debated. It is hard to know which ones from just looking at the data set that have been received, but it is something to keep in mind when training the model.

2.4.2 Side Task

Recall that the side task was to classify whether the image is an anterior or posterior view of the patient, see Section 1.3 for more information.

The data acquired was originally in DICOM files, a commonly used protocol for storage and communication in hospitals. The image data was extracted from these DICOM files to create new images in PNG-format instead. For some examples of what the images looked like, see Figure 1.1. The Bone

Scan images had the size 1024×256, and showed most of the body of the patient, though there were still quite a lot of variation on how much of the body that was visible in each image. In some images the whole body was visible, while others had various degrees of the legs missing. This is due to the fact that different hospitals have different protocols when doing Bone Scans.

The data set was divided into two different classes, where the "anterior" class was chosen to be the negative class, while the "posterior" was consequently chosen to be the positive class. It consisted of a total of 3838 examples where 1914 examples belong to the "anterior" class, while the remaining 1915 examples belong to the "posterior" class. The data set took in total 336 MB of hard drive space.

The classifier is expected to reach high accuracy, as the problem is considered to be simple for a human.

2.5 Software

2.5.1 Scala

Scala, which is an acronym for "Scalable Language", is a high level general purpose programming language. It was developed at the École Polytechnique Fédérale de Lausanne (EPFL) by Martin Odersky and was publicly released in 2004 [38]. The language has full support for both object-oriented and functional programming and is designed to be a flexible while still being concise. Scala source code is compiled to Java bytecode, and therefore runs on the Java virtual machine. This allows Scala to be freely mixed with Java and enables usage of the libraries of both languages. In this thesis, we used Scala version 2.10, mostly because of compability reasons.

2.5.2 Deeplearning4J

Deeplearning4J (DL4J) is an open source deep learning library written in Java and used with the Java virtual machine. This means that languages that work on the Java virtual machine is also able to use the framework, which is the main reason that it was chosen for this thesis work. It includes implementations of several different types of ANNs, including deep belief nets and CNNs. It is powered by its own open source numerical computing

library called ND4J. In this thesis, we used version 0.4-rc3.8 of DL4J, as it was the latest released version.

2.5.3 Python

Python is an open source high level general programming language. It was developed at Centrum Wiskunde & Informatica by Guido van Rossum and was publicly released in 1990. Today the language is widely used in many different fields because of the large standard library and third-party packages that are freely available. It was publicly released in 1990 and is today used within many different fields. In this thesis, we used Python v.3.4.4.

2.5.4 Keras

Keras is an open source deep learning library written in Python that is capable of running on top of either TensorFlow or Theano with GPU support. Both TensorFlow and Theano are also deep learning libraries, though Keras works as an abstraction layer over these libraries. Keras focuses on being easy to use by being modular and minimalistic. In this thesis, we have chosen to use the Theano backend as TensorFlow is only supported on UNIX systems. In this thesis, we used Keras v.1.0.2, as it was the latest released version.

Chapter 3

Implementation Details

In this chapter we will describe the various preprocessing methods used on the data, and the setup of the hyper parameters for the final CNNs. Both the data sets were split into training set, validation set and test set using the same ratios: $\frac{4}{10}$, $\frac{3}{10}$ and $\frac{3}{10}$, respectively.

3.1 Main task

The data set was split into training set, validation set and test set, where each had 4171, 3128, 3129 hotspot images respectively.

Because the image patches were in different sizes and ANNs require constant input size, we needed to use a method to standardize the size. We found that the smallest image patches were 23×23 and decided to crop each patch into 20×20 images. As a way of to avoid throwing away too much information from the bigger image patches, we cropped 5 images from each hotspots regardless of the size. 4 of the images came from the corners and the last one was cropped from the centre of each image. This cropping technique increased the size of the data set to 203 MB. We then implemented a CNN in Keras that could take in 5 image patches. Because of the slight unbalance between the two classes in the training data, undersampling was used, as it gave the best combination of faster training and good performance.

The shape of the final model can be seen in Figure 3.1. Notice that the network has five branches with one input in each branch, which correspond to the five patches cropped from each hotspot. This is one of several reasons that Keras was used to implement this network, as DL4J, at the time of

training the network, still had no implementation that allowed for this type of solution.

The following hyper parameters were used

- Batch size: 32
- Learning rate: $\eta = 3 \cdot 10^{-3}$
- Momentum: $\mu = 0$
- L_2 regularization: $\lambda_{L_2} = 1 \cdot 10^{-5}$
- Activation function: ELU, with $\alpha = 1$
- Dropout rate: $p = 0.5$
- Number of epochs trained: 40

3.2 Side task

At the beginning of the thesis work, this task was considered to be the easiest, which led to this task being worked on first. This is the reason that this task was chosen for evaluating the DL4J framework.

The data set was split into training set, validation set and test set, where each had 1535, 1151, 1152 Bone Scan images respectively.

Due to the fact all of the Bone Scan images had the size 1024×256, even with the use of CNNs, each new training process would take a huge amount of time. This would be infeasible for the thesis work, as we still need to iterate over many different configurations and settings of hyper parameters that each need a new iteration of the training process. We therefore had to decrease the size of the images before we could send them as input to the CNN. At first we went with cropping the head in each image using some simple image analysis algorithms. The reason we chose the head is that for a human it is easy to see if the image shows the front or back of the head. But we soon realised that it was not enough to crop only the head, as there were examples in the data set where the patient was looking to one side, which makes it impossible to to know the direction of the image just by looking at the head. The cropping was adjusted to include the upper torso too,

which was enough to successfully train a CNN with high accuracy. With the cropping, the total size of the data set fell down to only 15.1 MB.

Because the pixel values in the images was in the scale $[0, 255]$, we had to standardize the values. Because there were often outlier values in the image, i.e. pixels with higher values than the others, we decided to not use a zero mean and unit variance normalization. Instead we used what we called "K-highest-pixels" normalization, which means that we took the median value of the K highest pixels values and divided all pixels in the image with this value. This normalization method has the advantage that it only uses the information in the current image to standardize the values, as the outlier values in each image can vary a lot. To further squash the pixel values, we used element-wise square root on each image, which affect the pixels with high values more than the one with low values.

The shape of the final model can be seen in Figure 3.2. It has the following hyper parameters

- Batch size: 128
- Learning rate: $\eta = 8.77 \cdot 10^{-4}$
- Momentum: $\mu = 0.83$
- L_2 regularization: $\lambda_{L_2} = 1.37 \cdot 10^{-3}$
- Activation function: ELU, with $\alpha = 1$
- Dropout rate: $p = 0.5$
- Number of epochs trained: 8

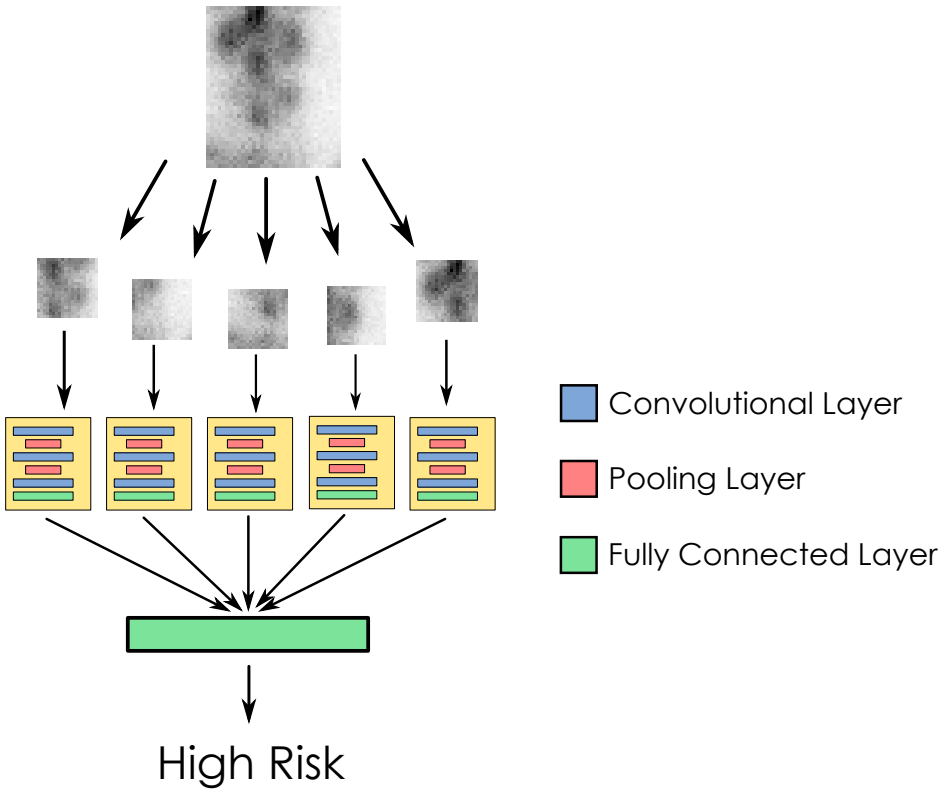


Figure 3.1: Illustration of the final model used in the main task. Each hotspot is cropped at each corner and the centre. These cropped images are sent to separate input nodes with their own trained layers. The output from each branch is then concatenated and processed by a final fully connected layer.

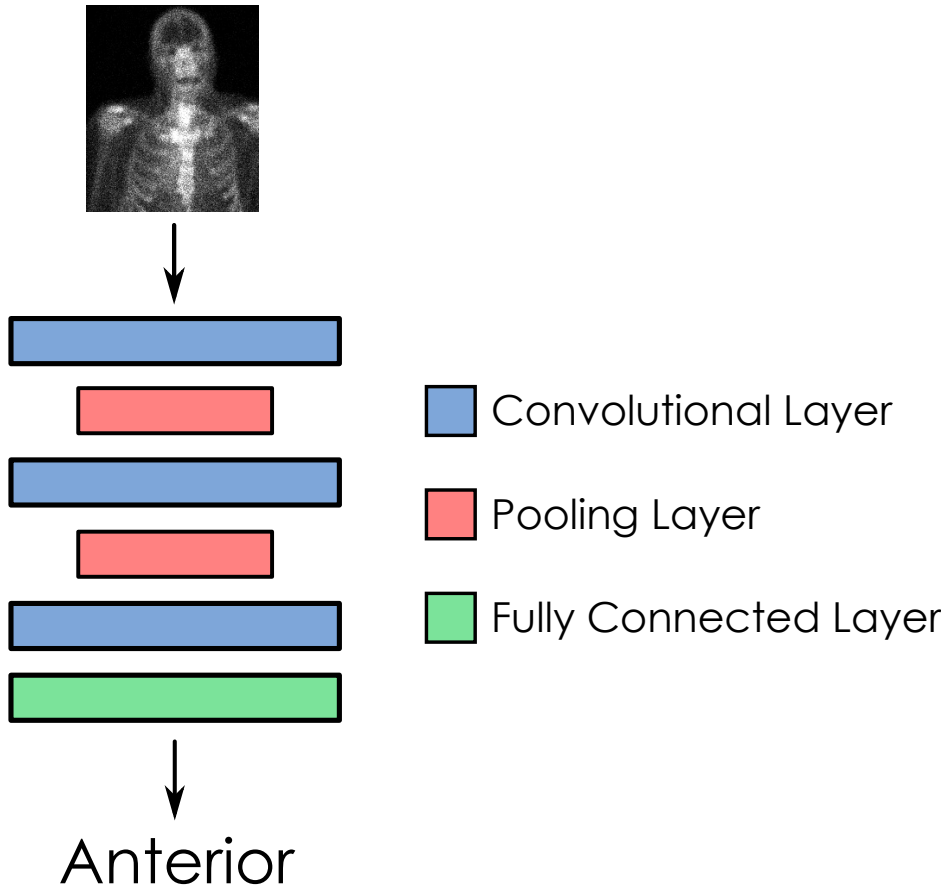


Figure 3.2: Illustration of the final model used in the side task.

Chapter 4

Results

As mentioned in Section 1.3, we will present the results in the form of classifier accuracy, F_1 score, ROC curve and AUC. We also include TPR and TNR as it gives some additional insight into the performance of the classifier. Almost all of these metrics have been described in Section 2.1.7. The only one not described is TNR, acronym for true negative rate, which corresponds to the TPR but for the negative cases. We decided to use the standard threshold of 0.5 when calculating the numerical metrics for both tasks, as we have no knowledge on what the emphasize in the classifier.

4.1 Main Task

We have gathered all the numerical metrics for both the validation set and the test set in Table 4.1. We have also included results from an ordinary shallow ANN that was only trained on handcrafted features that can be extracted from within each image, such as the area of the hotspots and the shape of it. The results from the shallow ANN is only based on the test set. As an side note for evaluating the performance the of the Keras framework, each iteration over an epoch took on average about 19 seconds when training the network.

The ROC curves from our CNN and the shallow ANN can be seen in Figure 4.1 and Figure 4.2 respectively. Note that the difference in style of the ROC curve is caused by the usage of different plotting tools.

Table 4.1: The numerical performance metrics of our trained CNN on both the validation set and test set for the main task. Results from an ordinary shallow ANN has also been included as comparison. 0.5 was used as threshold value when calculating these results.

	Validation set	Test set	Shallow ANN
Accuracy	0.872	0.890	0.880
F_1 score	0.915	0.919	0.826
TPR	0.983	0.981	0.945
TNR	0.610	0.649	0.733
AUC	0.959	0.955	0.922

Table 4.2: The numerical performance metrics of our trained CNN on both the validation set and test set for the side task. 0.5 was used as threshold value when calculating these results.

	Validation set	Test set
Accuracy	0.993	0.990
F_1 score	0.993	0.990
TPR	0.995	0.995
TNR	0.992	0.984
AUC	0.999	0.998

4.2 Side Task

We have gathered all the numerical metrics for the validation set and the test set in Table 4.2. The corresponding ROC curve can be found in Figure 4.3.

As an side note for evaluating the performance the of the Deeplearning4J framework, each iteration over an epoch took on average about 22 minutes when training the network. During the time that the majority of the thesis work was being done, Deeplearning4J had an upcoming update that would greatly improve performance for training and include better support for GPUs (graphical processing units). This update was delayed numerous times which resulted in that we did not have time to use it for the thesis.

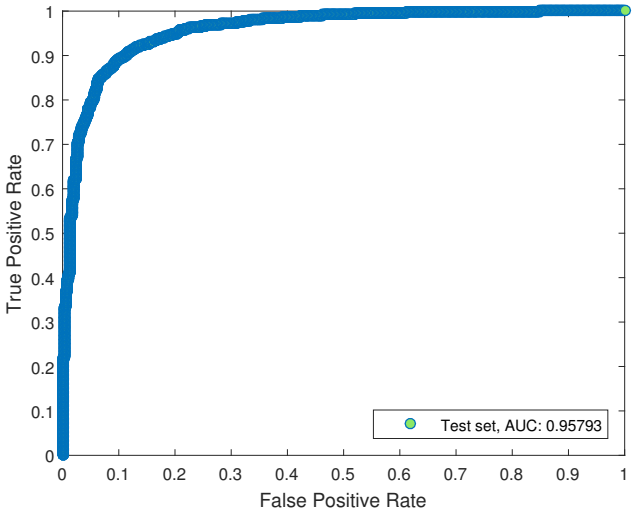


Figure 4.1: ROC curve from our hotspots classifier using CNNs on the test set.

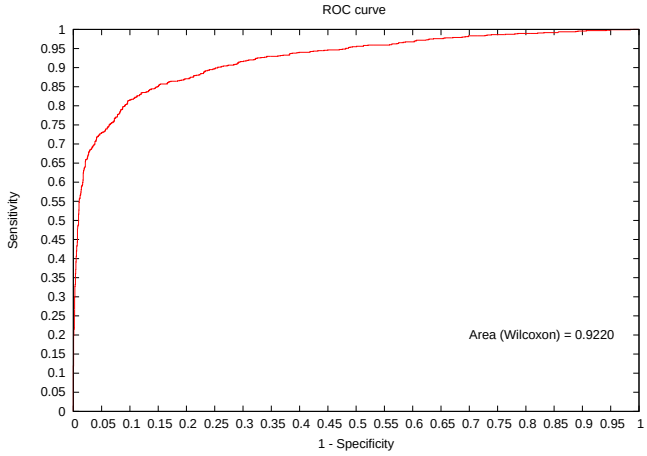


Figure 4.2: ROC curve from the hotspots classifier using a shallow ANN on the test set. The terms sensitivity and "1-specificity" are just other names on TPR and FPR, respectively.

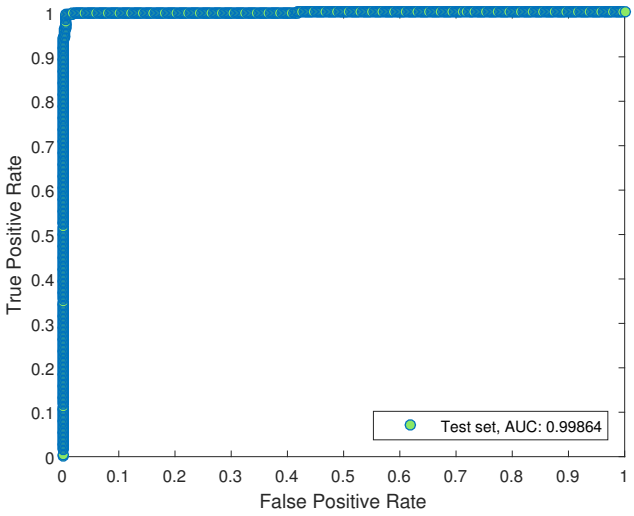


Figure 4.3: ROC curve from our anterior-posterior classifier on the test set.

Chapter 5

Discussion

We have successfully implemented two CNNs where one is able to distinguish between anterior and posterior in Bone scans, while the other CNN can classify whether a hotspot has a high risk of indicating cancer metastases or not. The first CNN was implemented in DL4J and the second one was implemented in Keras. The reason for moving over to Python and the deep learning framework Keras was because at the time of training networks for the two tasks, DL4J was just too slow. When comparing the two frameworks in terms of training speed, DL4J took on average 22 minutes per training epoch, while Keras only took 19 seconds. This difference could be due to fact that the networks were two different types of data sets, but looking at the sizes of the data sets after their respective cropping techniques, 15 versus 203 MB, Keras should have been slower.

Another problem with DL4J is that they do not have the same functionality as Keras has, which means that our current network for the main task can not be created in DL4J. Although DL4J was deemed to be unfit for production, it is actively being updated with new features that include support for GPUs and more flexible networks. The new updates have not been tested and may have given DL4J the performance boost needed for it to be usable in production.

Looking at the results for the main task, they can be considered to be quite good if we take into account the short discussion in Section 2.4.1. One possible way to incorporate opinions from several experts into one classifier would be to first collect a data set where each example would have several labels, one from each expert. A new network would then be trained for each of the experts' labels and these trained networks would then be put together

into an ensemble. When testing on an unobserved example, the answer from each network would then be collected and depending on this last classifier step, a final prediction would be made. This last classifier step could be to just average the answers and pick the class with the highest score, but we theorize that that the best way would be to use the answer with the highest score as the correct label. This would correspond to deciding the label of an hotspot by letting the expert that is the most sure decide.

Looking at the results for the side task, the results were like what we expected them to be, almost perfect. But something that could further increase the performance of the classifier would be to take into account that each Bone Scan results in two images for each patient. These image pairs are always the same, which would mean that when classifying the two images, if one is classified as anterior, the other images cannot be anterior too. We can therefore take the answer with the highest score as being the correct one and let the other image take the remaining class. This essentially means that we increase the performance of the network by letting it classify both images and only assign a class label to one of them.

Bibliography

[1] K. Allen. ***How a Toronto professor's research revolutionized artificial intelligence***. In: *Toronto Star* (2015). URL: https://www.thestar.com/news/world/2015/04/17/how-a-toronto-professors-research-revolutionized-artificial-intelligence.html (visited on 06/01/2016).

[2] I. Arnold et al. ***The Bone Scan***. In: *Seminars in Nuclear Medicine* 42.1 (2012), pp. 11–26. DOI: 10.1053/j.semnuclmed.2011.07.005.

[3] FA. Azevedo et al. ***Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain***. In: *The Journal of comparative neurology* 5 (2009). DOI: 10.1002/cne.21974.

[4] R. Benenson. ***What is the class of this image? Discover the current state of the art in objects classification***. URL: https://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html (visited on 04/20/2016).

[5] Y. Bengio. ***Learning Deep Architectures for AI***. In: *Found. Trends Mach. Learn.* 2 (2009). DOI: 10.1561/2200000006.

[6] O. Bergman et al. ***Cancer i siffror 2013***. In: (2013).

[7] J. Bergstra and Y. Bengio. ***Learning representations by back-propagating errors***. In: *Nature* 323 (1986).

[8] J. Bergstra and Y. Bengio. ***Random Search for Hyper-Parameter Optimization***. In: *Journal of Machine Learning Research* 13 (2012), pp. 281–305.

[9] Landstingens och regionernas nationella samverkansgrupp inom cancersjukvården. ***Prostatacancer. Årsrapport från Nationella prostatacancerregistret***. In: (2013).

[10] BI. Carlin and GL. Andriole. ***Pathogenesis of Osteoblastic Bone Metastases From Prostate Cancer***. In: *Cancer* 88.12 Suppl. (2000), pp. 2989–94. DOI: 10.1002/1097-0142.

[11] Onkologiskt centrum. ***Vårdprogram för vuxna patienter med skelettmetastaser***. In: (2003).

[12] DA. Clevert et al. ***Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)***. In: *Computing Reasearch Repository* (2015).

[13] R. Collobert et al. ***Natural Language Processing (almost) from Scratch***. In: *Computing Reasearch Repository* (2011).

[14] ER. DeLong et al. ***Comparing the areas under two or more correlated receiver operating characteristic curves: a nonparametric approach***. In: *Biometrics* 44 (1088), pp. 837–45.

[15] ER. Dennis et al. ***Bone Scan Index: A Quantitative Treatment Response Biomarker for Castration-Resistant Metastatic Prostate Cancer***. In: *Journal of Clinical Oncology* 30 (5 2012), pp. 519–24.

[16] DA. Drachman. ***Do we have brain to spare?*** In: *Neurology* 64 (2005). DOI: 10.1212/01.WNL.0000166914.38327.BB.

[17] X. Glorot and Y. Bengio. ***Understanding the difficulty of training deep feedforward neural networks***. In: *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics* (2010).

[18] I. Goodfellow et al. "Deep Learning". Book in preparation for MIT Press. 2016. URL: http://www.deeplearningbook.org.

[19] A. Griewank. ***Who Invented the Reverse Mode of Differentiation?*** In: *Documenta Mathematica* Extra Volume (2012).

[20] DO. Hebb. ***Organization of Behavior. A Neuropsychological Theory***. In: (1949).

[21] DO. Hebb. ***The organization of behavior. A neuropsychological theory***. 1st ed. Lawrence Erlbaum Associates, Inc., Publishers, 1949.

[22] G. Hinton et al. ***Deep Neural Networks for Acoustic Modeling in Speech Recognition***. In: *Signal Processing Magazine* (2012).

[23] GE. Hinton et al. ***A fast learning algorithm for deep belief nets***. In: *Neural Computation* 18 (7 2006), pp. 1527–54.

[24] GE. Hinton. ***Training Products of Experts by Minimizing Contrastive Divergence***. In: *Neural Computations* 8 (2002).

[25] K. Hornik et al. ***Multilayer feedforward networks are universal approximators***. In: *Neural Networks* 2 (5 1989), pp. 356–66. DOI: 10.1561/2200000006.

[26] T. Ibrahim, E. Flamini, et al. ***Pathogenesis of Osteoblastic Bone Metastases From Prostate Cancer***. In: *Cancer* 116.6 (2010), pp. 1406–18. DOI: 10.1002/cncr.24896.

[27] M. Imbraco et al. ***A new parameter for measuring metastatic bone involvement by prostate cancer: the Bone Scan Index***. In: *Clinical Cancer Research* 4 (1998), pp. 1765–72.

[28] M. Imbriaco et al. ***A new parameter for measuring metastatic bone involvement by prostate cancer: the Bone Scan Index***. In: *Clinical Cancer Research* 4 (1998), pp. 1765–72.

[29] ER. Kandel. ***Principles of neural science***. 5th ed. McGraw Hill, 2013.

[30] Y. Kim et al. ***Character-Aware Neural Language Models***. In: *Computing Research Repository* abs/1508.06615 (2015).

[31] S. Kotz and NL. Johnson. ***Breakthroughs in Statistics: Foundations and Basic Theory***. Springer Science & Business Media, 2012.

[32] A. Krizhevsky et al. ***Imagenet classification with deep convolutional neural networks***. In: *Advances in Neural Information Processing Systems 25* (2012), pp. 1106–14.

[33] CJ. Logothetis and S-H. Lin. ***Osteoblasts in Prostate Cancer Metastasis to Bone***. In: *Nature Reviews Cancer* 5.1 (2005), pp. 21–28. ISSN: 1474-175X. DOI: 10.1038/nrc1528.

[34] WS. McCulloch et al. ***A Logical Calculus of the Ideas Immanent in Nervous Activity***. In: *Bulletin of Mathematical Biophysics* 5 (1943).

[35] KT. McVary, CG. Roehrborn, et al. ***Update on AUA guideline on the management of benign prostatic hyperplasia***. In: *The Journal of Urology* 185.5 (2011), pp. 1793–1803. DOI: 10.1016/j.juro.2011.01.074.

[36] TM. Mitchell. ***Machine Learning***. 1st ed. McGraw-Hill, Inc., 1997.

[37] V. Nair and GE. Hinton. ***Rectified Linear Units Improve Restricted Boltzmann Machines***. In: *Proceedings of the 27th International Conference on Machine Learning* (2010).

[38] M. Odersky et al. ***An Overview of the Scala Programming Language. Second Edition***. In: *Computing Research Repository* (2006).

[39] F. Rosenblatt. ***The perceptron: a probabilistic model for information storage and organization in the brain***. In: *Psychological Review* 65 (6 1958).

[40] M. Sadik et al. ***Quality of planar whole-body bone scan interpretations–a nationwide survey***. In: *European Journal of Nuclear Medicine and Molecular Imaging* 35.8 (2013), pp. 1464–72. ISSN: 1619-7070. DOI: 10.1007/s00259-008-0721-5.

[41] Jonathon Shlens. ***A Tutorial on Principal Component Analysis***. In: *CoRR* abs/1404.1100 (2014).

[42] MR. Smith, J. Eastham, et al. ***Randomized Controlled Trial of Zoledronic Acid to Prevent Bone Loss in Men Receiving Androgen Deprivation Therapy for Nonmetastatic Prostate Cancer***. In: *The Journal of Urology* 169.6 (2003), pp. 2008–12. DOI: 10.1097/01.ju.0000063820.94994.95.

[43] MS. Soloway, SW. Hardeman, et al. ***Stratification of patients with metastatic prostate cancer based on extent of disease on initial bone scan***. In: *Cancer* 61.1 (1988), pp. 195–202. DOI: 10.1002/1097-0142.

[44] N. Srivastava et al. ***Dropout: A Simple Way to Prevent Neural Networks from Overfitting***. In: *Journal of Machine Learning Research* 15 (2014).

[45] Deeplearning4j Development Team. ***Deeplearning4j: Open-source distributed deep learning for the JVM***. URL: http://deeplearning4j.org (visited on 05/29/2016).

[46] D. Tran et al. ***Learning Spatiotemporal Features with 3D Convolutional Networks***. In: *Computing Reasearch Repository* (2014).

[47] D. Ulmert et al. ***A Novel Automated Platform for Quantifying the Extent of Skeletal Tumour Involvement in Prostate Cancer Patients Using the Bone Scan Index***. In: *Clinical Cancer Research* 62 (1 2012), pp. 78–84.

[48] PJ. Werbos. ***Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences. PhD thesis***. In: (1974).

[49] B. Widrow and MA. Lehr. ***30 years of adaptive neural networks: perceptron, Madaline, and backpropagation***. In: *Proceedings of the IEEE* (1990), pp. 1415–42. DOI: 10.1109/5.58323.

Master's Theses in Mathematical Sciences 2016:E24
ISSN 1404-6342

LUTFMA-3297-2016

Mathematics
Centre for Mathematical Sciences
Lund University
Box 118, SE-221 00 Lund, Sweden

http://www.maths.lth.se/