NTNU – Trondheim
Norwegian University of
Science and Technology

Supervised Learning and its Development to Generative Adversarial Networks

Lone Marselia Werness Bekkeheien

December 2019

PROJECT THESIS

Department of Engineering Cybernetics

Norwegian University of Science and Technology

Supervisor: Anastasios Lekkas, ITK

Preface

This project thesis represents my work at the Norwegian University of Science and Technology(NTNU) as part of the study program Master of Science in Cybernetics and Robotics. This work has been done under the supervision of Anastasios Lekkas during the autumn semester of 2019, which has been a great inspiration! This thesis shows my gained knowledge about machine learning and how it can be used to generate data by utilizing generative adversarial networks (GANs). The work has been performed on a MacBook Air with a 1,6 GHz Intel Core i5 processor.

The theory and implementation in Section 3 is mainly based upon [1, 2]. For the implementations of basic ML methods like linear regression, logistic regression, SVM and decision trees, JupyterNotebook was utilized together with Python version 3.6.4. The data for the linear regression is retrieved from [3]. The implementation is mainly based upon the books [2, 1], and is inspired by an assignment in the subject *TDT4173 Machine Learning and Case Based Reasoning* at Norwegian University of Science and Technology. The libraries used in the implementation are the following:

- Pandas 0.25.3
- NumPy 1.17.3
- Matplotlib 3.1.1
- Scikit-Learn 0.21.3

The Pandas library version 0.25.3 is utilized to load the data, and the NumPy library version 1.17.3 is utilized to store the data in arrays and do manipulations on it. Scikit-Learn's library version 0.21.3 is utilized to do a train-test split on the data.

The main libraries utilized for the logistic regression, SVM and decision trees are exactly the same as for the linear regression above. The Scikit-Learn library is utilized to get the different models for the classification. The implementation is built on inspiration from [2], and the logistic regression also gets inspiration from an assignment in the subject *TDT4173 Machine Learning and Case Based Reasoning* at Norwegian University of Science and Technology .

Due to a lack of processing power running on the MacBook's processor in JupyterNotebook, the more advanced implementations like CNN and GAN were utilizing a GPU Google Colab server. Google Colab currently uses Python version 3.6.9 which was used for the CNN and GAN implementation. The Python version was found by importing the platform and printing its version in Google Colab.

The main inspiration for the implementation and theory in Section 4.2 was gotten from [2, 4, 5]. Libraries utilized in the CNN implementation are the following:

- TensorFlow 1.15.0
- Keras 2.2.4
- NumPy 1.17.4
- Matplotlib 3.1.2

TensorFlow version 1.15.0 was utilized in the CNN implementation in Section 4.2. TensorFlow is an end-to-end open source platform for ML [5]. TensorFlow provides multiple levels of abstraction, where the high-level Keras API was chosen for this implementation. The implemented Keras version in TensorFlow 1.15.0 is Keras 2.2.4, which is utilized in this implementation. A TensorFlow Keras sequential model is created by passing a list of layer instances to a sequential constructor [6]. This way of making a model was chosen because of intuitive reasons. TensorFlow and Keras are the main libraries for the CNN implementation, but two help libraries are needed as well. The first help library is NumPy version 1.17.4 which is utilized to store the data in arrays and to perform changes to the arrays. The library Matplotlib version 3.1.2 is utilized to visualize both the data and the results from the implementation. The MNIST- and CIFAR10 images, described in Section 3, are retrieved from the Keras library's datasets. The bar charts visualizing the data in Figure 4.1 and 4.2 is inspired from [7]. Inspiration for visualizing the different images in the datasets are also gotten from [8]. An inspiring discussion with Kerstin Bach at the Department of Computer science was also helpful when implementing the CNN.

The theory and implementation in Section 5 is mainly based on [9, 4, 10]. Libraries utilized in the GAN implementation are the following:

- TensorFlow 2.0.0
- Keras 2.2.4
- NumPy 1.17.4
- Matplotlib 3.1.2
- Python's Time library
- IPython
- Keras Models

At first TensorFlow version 1.15.0 was also used in the GAN implementation. To be able to iterate through the batches in the dataset with a for loop, TensorFlow's eager execution had to be enabled. By switching to TensorFlow 2.0.0, which will be default version in Google Colab in the feature, eager execution is enabled by default. Therefore TensorFlow version 2.0.0 is utilized in the GAN implementation. The libraries NumPy and Matplotlib are used for the same purposes as for the CNN implementation described above. Python's Time library is utilized to measure how much time each epoch takes to execute. IPhyton's Display library is used to display the performance of the GAN after each epoch as showed in Figure 5.3 and 5.4, which are inspired by [8].

Trondheim, 17-12-2019

Lone Marselia Werness Bekkeheien

Acknowledgment

Learning about generative adversarial networks with little, up to no knowledge in field of machine learning and neural networks in advanced has been challenging. I would like to thank my supervisor, Anastasios Lekkas, for pointing me in the right direction as well as motivating me with his engagement in the field. I would also like to thank Kerstin Bach at the Department of Computer science for a helpful discussion in the field of neural networks.

L.M.W.B

Summary

Machine learning was defined in 1959 by Arthur Samuel as the field of study that gives computers ability to learn without being explicitly programmed [2]. Basic machine learning methods like regression and decision trees have developed to more advanced methods. Neural networks was truly evolved by a collaboration between neuroscientists and computer scientists looking into what intelligence actually is [11]. Convolutional neural networks has been a game changer in the field of image recognition. Generative adversarial networks have been a further development in the field of image recognition, where the networks can learn an unknown data distribution function to generate new images.

In this thesis, implementation of basic machine learning methods, like linear- and logistic regression, decision trees and support vector machines, and convolutional neural networks have been performed. The results showed how support vector machines and decision trees worked better than logistic regression when the dataset include a lot of target variables and features. The linear regression method needed a linearly separable dataset, thus it could not be used on a non linearly seperable dataset without applying manipulations. For predictions on more advanced datasets, like the Mixed National Institute of Standards and Technology images a convolutional neural network was utilized. This implementation gave satisfactory results with a test accuracy of 98.6 percent.

The knowledge gained from these implementations have been utilized to implement generative adversarial networks to generate new data from an already known data distribution. Being familiar with neural networks was a great advance when implementing the generative adversarial networks. The research in different papers, and the theory study of generative adversarial networks also showed to pay off when small tweaks in the hyper parameters did drastically changes to the generated images quality. Based on the results, generative adversarial network shows great potential for future work in the master thesis.

Contents

Chapter 1

Introduction

This project's thesis concerns the generative adversarial networks (GANs) and its application to generate data. Section 1.1 presents the background and motivation for this project's thesis. Section 1.2 presents the main objectives and the approach that will be needed to reach the overall goal of the thesis. Finally, Section 1.3 will give an overview of the report's structure.

1.1 Background and Motivation

In 1959 Arthur Samuel gave the first definition for machine learning (ML), *machine learning is the field of study that gives computers ability to learn without being explicitly programmed* [2].
Today ML is everywhere. One of the first ML algorithm that helped making everyone's life easier was the spam filter for email. The spam filter can be developed with logistic regression for classification, where the train data is labeled "spam" or "not spam". From that information, ML algorithms extract patterns that help classify new incoming data as spam or not spam. If a spam filter is made without the use of ML, rules for what kind of emails that should be flagged as spam has to be written explicitly. Such a process would have been very time consuming.
Basic ML methods are for instance both linear- and logistic regression, decision trees and support vector machines (SVMs). All of these methods will be represented in this thesis.
Today there exists more advanced branches in ML, and deep learning (DL) is one of the them.

The field of DL makes it possible to solve more complex tasks better, like image recognition with state of the art performance for instance. Since 2012 DL has gained a lot of attention, where Krishevsky, Sutskever and Hinton managed to half the existing error rate on an ImageNet classification with deep convolutional neural networks (CNNs) in the Large Scale Visual Recognition Challenge(LSVRC) competition [12]. The concept of neural networks (NNs) truly occurred with a collaboration between computer scientists and neuroscientists looking at what intelligence actually is [11]. Artificial NNs (ANNs) are built on the logic of how the human brain works by areas of neurons communicating [11]. The field of study, DL, involves NN with many layers communicating, which needs a lot of data to be able to detect patterns and make predictions. The network constructs the world by picewise interpreting it in a nested hierarchy, where each interpretation learns by simpler interpretations. DL has brought an enormous amount of applications in different fields. In the health sector DL has been used to detect skin cancer from images for instance [13].

From DL another ML branch has been developed, namely the generative adversarial networks (GANs). It was first introduced by Ian Goodfellow and other researchers at the University of Montreal in 2014 [14]. GANs consist of two NNs, the generative and the discriminative, which work as adversarials. The aim of a generative network is to learn the true data distribution of the training data, and then use this distribution to generate new data that looks like it comes from the same distribution. The discriminator aims to predict which data comes from the training dataset and which comes from the generated dataset. Thus, the overall goal for a GAN network is to generate data that looks as if it has been generated by the same set of rules as the training data. To reach this goal, an unknown probabilistic distribution function that explains why some data are more likely to be found in the training dataset and others are not, is to be found.

The main task of this project thesis is to gain enough knowledge to be able to utilize GANs to generate data. To be able to do this sufficient experience and insight in the field of ML and DL is needed.

Data is becoming the most valuable asset in the world. In the mid-19th century the oil industry as we know it began extracting oil. Exploitation of oil is highly responsible for Norway's welfare today. Imagine if it was possible to make a data program that generates oil.

Companies like Google and Facebook own vast amounts of data which they can exploit for commercial purposes, this is the reason why these companies are so valuable. Since GANs can be used to generate data, it is actually possible to make a data program to generate the most valuable asset in the world!

GANs have been used to generate synthetic pieces of art, music, people and much more. In 2016 a start up company called Brud generated a fake Instagram profile under the name Lil Miquela [15]. She has over 1.6 million followers and in 2017 she even released an album. Brud is earning a lot of money on an artificial intelligence generated person. Another example is a portrait generated by a GAN that was sold for USD 432.000 [16]. Generative modeling is also important in further understanding of the human brain [10]. As shown above, the potential of GANs is massive as it can mimic any distribution of data. Since GANs were first introduced in 2014 [14] it has been researched extensively [17, 18, 19, 20, 21, 22], although there are still many application areas where their benefits are yet to explore.

1.2 Objectives and Approach

The main goal of this work is for the author to become familiar with the main approaches within supervised learning, and pave the way for implementing GANs in industry-relevant application during the MSc thesis period.

To be able to reach this goal, knowledge in the field of ML is needed. The limitations on the authors behalf is the lack of prior knowledge in this field. The approach to reach the goal will therefore be to start with basic theory of ML and implement basic algorithms. The main literature used to reach this objective will be [1, 2]. After the basic theory, knowledge of DL will be gained and applied by mainly using [2, 4]. In the last section, the knowledge gained on reaching the previous objectives will be used together with theory about GANs, mainly based on [5, 9, 4, 10], to implement GANs.

1.3 Structure of the Report

The rest of the report is structured as follows: Section 2 gives an introduction to the basic theory in ML. The next section will focus on basic ML algorithms like linear regression, logistic regression, support vector machine and decision trees. This section will include comparison and plots of the different methods. Section 4.2 will dig into NNs like deep feed forward- and convolutional networks. The last chapter will use the knowledge from the previous chapters to work on GANs. It will include discussion around a trained model for making artificial data.

Chapter 2

Introduction to Machine Learning

The representation in this section is mainly based on [2].

A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E.
—Tom Mitchell, 1997 [2].

2.1 What is Machine Learning?

The building block in ML is *concept learning*, which is when a machine is trained to learn a concept by given some predefined examples. For instance, humans differentiate a certain type of flower from all the flowers by a specific set of features among a large number of features. A feature can be the color of the flower, the length of the leaves and so on. This set of features that are used to differentiate flowers can be called a *concept*. Machines can also learn this concept to recognize a certain type of flower by training on examples to find a hypothesis that best fits this training data. According to the definition by Tom Mitchell, if the program learns the concept of the flowers from the training data, the performance will improve with the experience when the data comes from the same distribution [2]. For an algorithm to support concept learning it needs to have training data, the target concept and actual data objects to test the model. For the algorithm to be able to make predictions on the test data it must be assumed that the approximation of the target function for the training data will work as an approximation for the target function of the test data as well. This assumption is called the *fundamental assumption of inductive learning* [23]. The algorithm also requires an *inductive bias* to learn the concept [23]. Without the inductive bias, the algorithm is just serving as storage for the training data. The inductive bias is an a priori assumption regarding the identity of the target concept, and without it there is no rational basis for being able to classify unseen instances [23].

2.1.1 Categories

The ML algorithms can be categorized into three main categories. The first category is based on whether or not the correct answers to the classification problems are known.

Supervised learning algorithms have both the input and output data available in the training set, that is, the data is labelled [2]. Two typical types of supervised learning is classification and regression, which will be introduced in later sections. The email spam filter is a typical classification example, where the training data consists of different emails with corresponding labels classifying them as spam emails or not. A typical regression task is to predict the type of flower based on different kinds of features. The training data for this regression task will consist of sets of features for different flowers with their corresponding type. Linear regression, logistic regression, SVM, decision trees and NNs are supervised learning algorithms that will be written about in this paper.

Unsupervised learning algorithms operate on unlabeled training data [2]. Some important unsupervised learning algorithms are clustering and visualization and dimensionality reduction. The clustering algorithm is trying to group the input data in clusters based on similarity. For instance, you can use a clustering algorithm to detect what kind of groups are visiting an online newspaper, and at what times. This information can further be used to target the specific readers at specific times. Visualization algorithms can be used to input data and get a graphical representation as output. In unsupervised learning, the algorithms try to understand and learn from the data without the solution given in the training data, as for supervised learning. It is possible to make a hybrid of supervised- and unsupervised learning, which is called semi-supervised learning. Algorithms like this will have a data set consisting of a mix between unlabeled- and labeled data.

Reinforcement learning (RL) algorithms learn based on penalties/rewards received on the last action performed [2]. RL algorithms can also work in a changing environment, which is typically used for an algorithm that plays a game. The player is then called an agent, the game is the environment, and the player is selecting action based on a policy [2]. This policy is updated based on penalties/rewards from previous actions. The algorithm will play the certain game until the optimal policy is reached.

A ML algorithm can also be categorized based on if they learn "as they go" or not.

Online learning is when an algorithm learns online, which means that it can learn "as it goes" [2]. New data is utilized to train the algorithm and updates will happen while it is being used. An example of this is the e-mail spam filter. As the user receives new emails the spam filter can use them as train data to improve the filter, while it is flagging the emails as spam or not. A *learning rate* can be set according to how fast the user wants the algorithm to adapt to the new data. The drawback of this type of algorithms is that they might get poor input which will lead to unwanted performance. It is therefore important to closely watch the performance and react to abnormal input data.

Batch learning algorithms, in opposition to online learning algorithms, will learn offline [2]. The model will be trained using all available data, and will not learn anymore when it is deployed. If new data is available, the model needs to be trained with the whole data set, which means that updating the model might take a while. This is why an online learning algorithm is a better option for certain tasks that need to be updated frequently.

The last category for ML algorithms is based on if they work by comparing data points or detects patterns in the data to build a predictive model.

Instance based learning algorithms learns based on the input instances for training, then it generalizes to test data by similarity measure [2]. This means that if the algorithm learns that an email with the word "sale" is spam from the train data, it could also flag that a new email including "super sale" is spam depending on the similarity measure chosen.

Model based learning algorithms generalizes from a set of examples by searching for an optimal value for the model parameters [2]. This type of algorithms detects patterns in the training data and uses them to build a predictive model. To achieve this a cost function is utilized as well as a penalty for the model complexity.

2.2 Main Challenges

The main challenges of ML algorithms either concerns the algorithm itself or the data it uses [2]. The training data can be insufficient, non-representative, of poor quality or have irrelevant features. ML algorithms need a lot of data to be able to learn a concept, and the more complex the problem is, the more data is needed. Having representative training data means that it should be representative of the new data the algorithm aims generalizing to. *Sample bias* occurs when the training data does not represent the actual environment the model will be running in [2]. Data of poor quality typically has a substantial amount of outliers, errors and noise. It can therefore be an idea to clean up the training data by removing outliers before training the algorithm. It is also important that the training data contains enough relevant features for the problem to be solved, and few irrelevant ones.

Models suffering from poor performance on the test data can either suffer from overfitting or underfitting.

Overfitting is a phenomenon that it is important to be aware of. It happens when the algorithm takes the training data to literal, when it contains white noise or when it does not contain enough data to represent the actual problem. For instance, by looking at the graph plotted in the linear regression Section 3, it can be seen that a linear prediction function is utilized. A naive way of thinking could be, "why do not I use a prediction function that runs through every single point, this way I will have no error what so ever". If this is done there will be a perfect fit between the training data points given and the model. Why is it then desired to use the linear model as was done in the linear regression implementation in Section 3? The point is to predict output values based on input values, and this works great on the training data when the linear model is utilized. After the model is trained it has to be tested on the test data. The overfitted prediction function works perfect on the training dataset, but is not generalized enough to perform well on new data. The prediction function, which is the red line in Figure 3.2, is more general and will therefore give a better prediction for the test data. It can therefore be seen that it is not preferable to make a model that is tailored toward the training data. The model has to work for data it has never seen before, which is the whole point in learning. The risk of overfitting can be reduced by *regularization*, which is done by penalizing one of the parameters in the model so it becomes simpler [2]. It is not desirable to penalize the parameters too much because then the model will not fit the training data. The regularization is done through a *hyperparameter* in the algorithm before training the model. Tuning hyperparameters is an important part of ML algorithms.

Underfitting on the other hand is when the model learns from only some part of the training data. For instance, a model implemented to perform classification on the Canadian Institute For Advanced Research 10 (CIFAR10) dataset, described in 4.2, trains on training data that does not have images of airplanes and birds. When the trained model is being tested on a dataset including airplanes and birds it might not perform well, thus the model is underfitted. Underfitting can be fixed by for instance reducing the regularization hyperparameter, finding better features or finding a training dataset that is representative.

Cross-validation is a way to compare different ML methods and get a sense of how well they will work in practice. It can be performed by firstly splitting the data into four subsets for instance, and then taking the first three blocks for training and the last for testing. Secondly, the first block can be used for testing and the last three for training. This procedure continues until every block/subset of data has been used for testing and then average the results from each test. This is used to see how well the model generalizes, which also means if the model is overfitted or underfitted. Since the data is divided into four blocks in this example, it is called Four-Fold Cross Validation. However, the number of blocks/subset could be chosen arbitrarily. By using cross-validation different ML methods can be compared to see which model performs best on the test data.

Chapter 3

Theory and Implementation

The representation and implementation in this Section is mainly inspired by [1, 2].

Dataset

The dataset utilized for the implementation of the linear regression method is a set of 200 input-output vectors containing information about money spent on television advertisement and their generated sales.

The dataset used for the implementation of the SVM, logistic regression and decision trees is Python's iris data set, consisting of 150 flowers. This data set contains four features which are the petal- and sepal length and width of three different iris species called Setosa, Versicolor and Virginica. An example of the iris flowers are shown in Figure 3.1. The dataset is split into training- and test data using Python's Sickit-Learn *train-test-split*, where 80 percent of the data is used for training and 20 percent is used for testing. All methods are applied on the whole dataset, as well as a subset of the dataset.

The subset consists of the features sepal length and width and is used to classify whether or not the flower is of the species Setosa. This subset is used because the classes Septosa or not Septosa are linearly separable.

To represent the result of logistic regression, SVM and decision trees Python's Sickit-Learn *confusion matrix* will be utilized. It will be represented as shown in Table 3.1. Table 3.1 shows that

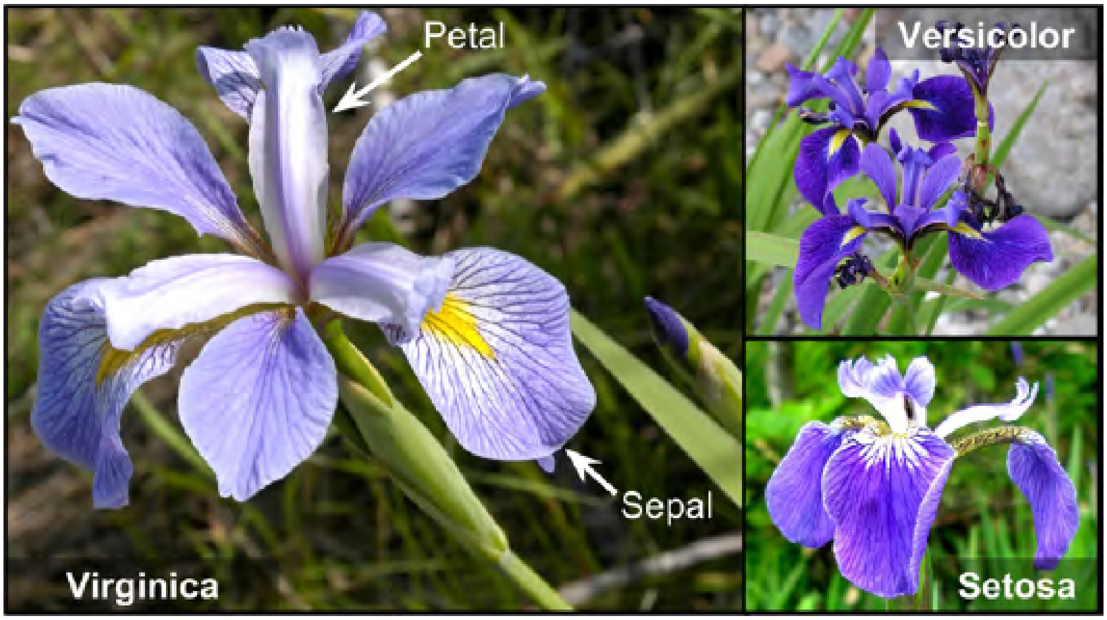


Figure 3.1: Flowers of three iris plant species [2].

		Predicted species		
		Setosa	Versicolor	Virginica
Actual species	Setosa	a	b	c
	Versicolor	d	e	f
	Virginica	g	h	i

Table 3.1: An example of a generic confusion matrix

a, e, i on the diagonal represents the correct predictions, while b, c, d, f, g, h represents incorrect predictions. Python's Scikit-Learn *accuracy score* will also be used to evaluate the performance.

3.1 Linear Regression

Linear regression is used when the goal is to predict continuous output variables, y, based on input variables represented by a vector $\mathbf{x}$. The aim is to use training data, consisting of $\mathbf{x}$ and its corresponding target values $\mathbf{t}$, to find weights that fit the model so that when new data is used the prediction of the output based on the input is quite accurate [1]. The model is based on linear combinations of functions of the input variables, known as a *basis functions* [1]. There are many possible choices for the basis functions, but in this section the focus will be on the simplest linear models for regression, the linear basis function models. This means that the functions of the input variables are linear, as shown below.

$$y(\mathbf{x},\mathbf{w}) = \sum_{j=0}^{M-1} w_j \phi_j(\mathbf{x}) = \mathbf{w}^T \phi(\mathbf{x}) \tag{3.1}$$

In (3.1) the basis function is represented by $\phi_j(\mathbf{x})$, and $j = 0$ represents the *bias* parameter in the model [1]. This means that the bias is integrated in the model, and to be able to do so the first element in the basis function, $\phi_0(\mathbf{x})$ is equal to 1, and w_0 is the bias. The equation represents a linear model because it is linear in $\mathbf{x}$. For the rest of this section the basis function vector $\phi(\mathbf{x})$ will be equal to the input vector $\mathbf{x}$, as shown in (3.2). There will also only be a single target variable, t.

$$y(\mathbf{x},\mathbf{w}) = \sum_{j=0}^{M-1} w_j x_j = \begin{bmatrix} w_0 & w_1 & \dots & w_{M-1} \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_{M-1} \end{bmatrix} = \mathbf{w}^T \mathbf{x} \tag{3.2}$$

After getting familiar with the model for linear regression, the next step is how to find the weights.

The aim is to select weights **w** so that the sum-of-squares error function, which is shown in (3.3) is minimized [1].

$$E_D(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\{t_n - \mathbf{w}^T\phi(\mathbf{x}_n)\}^2 \tag{3.3}$$

Minimizing this function is equivalent to maximizing a likelihood function under a conditional Gaussian noise distribution for a linear model [1]. Below the logarithm of this function is defined.

$$\ln p(\mathbf{t}|\mathbf{w},\beta) = \frac{N}{1}\ln\beta - \frac{N}{2}\ln(2\pi) - \beta E_D(\mathbf{w}) \tag{3.4}$$

In (3.4) the noise precision parameter is represented by β. To find the maximum of this function the gradient can be found and set to zero as shown below.

$$\nabla \ln p(\mathbf{t}|\mathbf{w},\beta) = \sum_{n=1}^{N}\{t_n - \mathbf{w}^T\phi(\mathbf{x}_n)\}\phi(\mathbf{x}_n)^T = 0 \tag{3.5}$$

Solving for **w** the weights can be,

$$\mathbf{w}_{ML} = (\phi^T\phi)^{-1}\phi^T\mathbf{t}. \tag{3.6}$$

The weights found in (3.6) can now be used in (3.1) to find the predicted output. This has been done in an implementation of the linear regression model, with the base function ϕ equal to **x**. As described in Section 3 the data is a set of 200 input-output vectors (x_i, y_i) divided into test data and training data. It contains information about money spent on television advertisement and their generated sales. The input vector **x** is added a one for every data point to be able to represent the bias. First, the weights are found by using the (3.6) on the training data. Then these weights were used in (3.1) along with the test data to predict the generated sales.

Figure 3.2 shows that there exist a linear relationship between sales and money spent on TV advertisement, and a straight line can be fitted to the data. This line is known as the *decision*

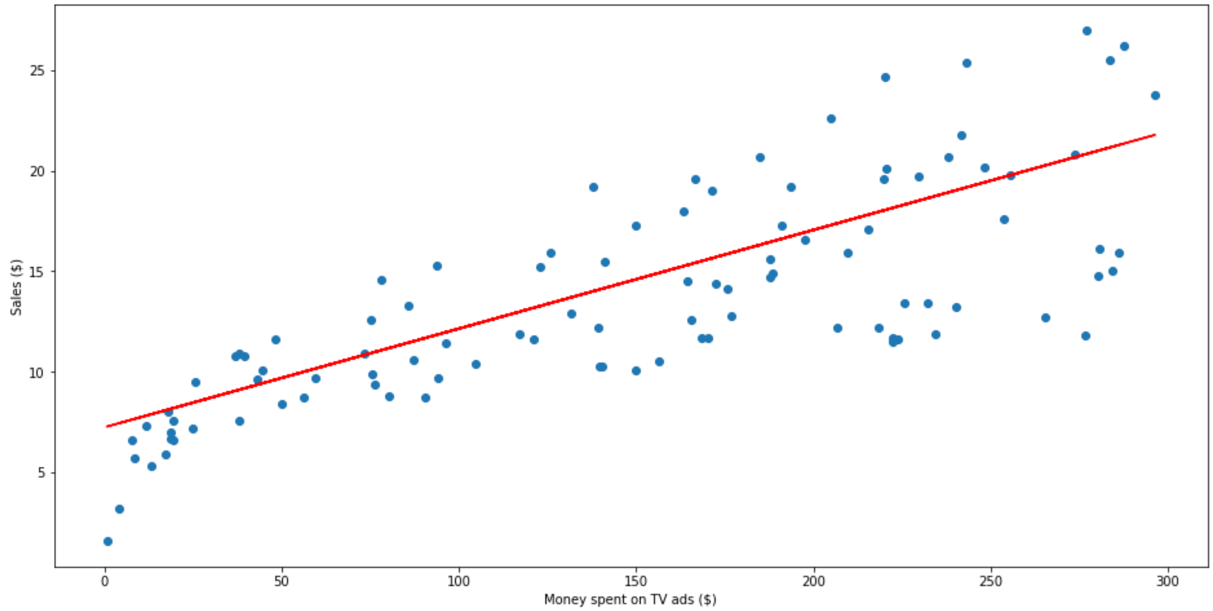


Figure 3.2: Simple linear regression model tested on the test data

boundary [1]. The figure also shows that the data points are quite spread, and therefore the sum-of-squares error is calculated to be 12.097 based on the test data. The sum-of-squares error for the training data is 9.341, which means that the model is not generalizing well. One reason for this might be the way the data was split. In Figure 3.2 the data was split in half to create a test- and training set. A general rule is to have 80 percent of the data in the training set and 20 percent as test data [2]. By using this split the sum-of-squares error for the training data is 9.700 and for the test data, it is 14.129, which is worse than it was for just dividing the data in half. This means that it is not known which chunk of the data is best for training and testing the model. Therefore the Python Scikit-Learn *train-test-split* function is better to split the data in 80 percent training data and 20 percent test data [2]. By doing this the error for the training data is 11.05 and for the test data, it is 8.41, which proves that the self-picked chunk of data was not the best. The result is shown in Figure 3.3. Cross-validation could be used to see how well the model is generalizing, and will flag problems like overfitting as explained in the previous Section 2.

One of the benefits of linear regression is how simple the model is. For someone with little knowledge in ML it is very simple to understand and use this method. It can be used to find a relationship between the two variables even if it does not fit the data. On the other hand not all data has this linear relationship that the method assumes. If most of the data is quite dense, and

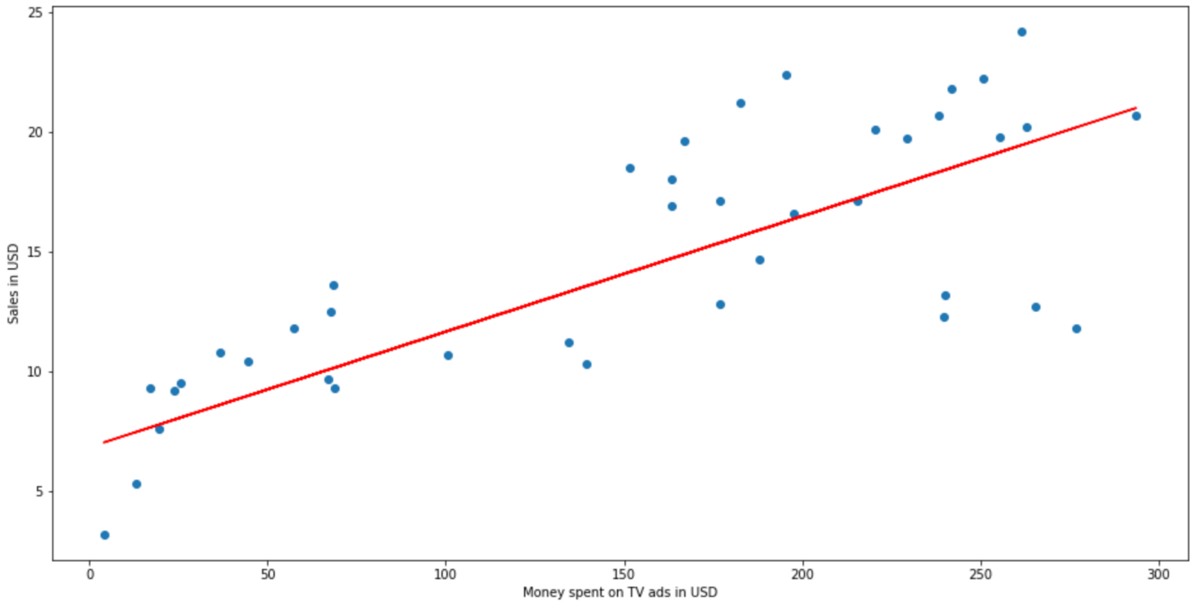


Figure 3.3: Simple linear regression with the scikit-learn split of data

there is one data point that is by itself it will influence the model coefficients significantly.

3.2 Logistic Regression

Logistic regression can be used to do binary classification. Looking at two-class classification, the model returns the probability that the input **x** belongs to the classification $y = 1$, C_1.The probability of the input belonging to class C_1 can be written as a logistic sigmoid function acting on ϕ

$$p(C_1|\phi) = y(\phi) = \sigma(\mathbf{w}^T\Phi). \tag{3.7}$$

The sigmoid function is represented by $\sigma(.)$, and is shown in Figure 3.4. The function is used to capture the probability. The logistic regression model is based on the same linear equation as for linear regression, shown in (3.1). The computation of the weights is a bit more complex for the logistic regression. It is done by starting with an initial set of weights, for instance setting the weights to zero. If the initial weights are zero, Figure 3.4 shows that the probability for the input x_i to belong in class $y = 1$ is equal to 0.5. The weights can be learned by either maximizing the likelihood function or minimizing the negative likelihood function. The likelihood function takes the form as shown below.

$$p(\mathbf{t}|\mathbf{w}) = \prod_{n=1}^{N} y_n^{t_n} \{1 - y_n\}^{1-t_n} \tag{3.8}$$

When the data sets are linearly separable the maximum likelihood function should not be used. The reason for this is that the solution of the maximum likelihood happens when the hyperplane, $\sigma = 0.5$, separates the two classes and **w**'s magnitude goes to infinity. This could potentially lead to over-fitting. Therefore the negative logarithm of the likelihood will be chosen, which can be represented as,

$$E(\mathbf{w}) = -\ln p(\mathbf{t}|\mathbf{w}) = -\sum_{n=1}^{N} \left\{ t_n \ln y_n + (1 - t_n) \ln(1 - y_n) \right\}, \tag{3.9}$$

where $y_n = \sigma(\mathbf{w}^T \phi_n)$. This function can also be called cross-entropy error function and will be used to see how good the model is.

To minimize (3.9), the gradient with respect to **w** will be performed, which results in

$$\nabla E(\mathbf{w}) = \sum_{n=e}^{N} (y_n - t_n)\phi_n. \tag{3.10}$$

The cross entropy error function can be minimized by the *Newton-Raphson* iterative optimization scheme, which is shown below.

$$\mathbf{w}^{(new)} = \mathbf{w}^{(old)} - \mathbf{H}^{-1} \nabla E(\mathbf{w}) \tag{3.11}$$

This means that for every iteration new weights will be generated based on the old ones. The matrix **H** represents the Hessian matrix, which is

$$\mathbf{H} = \nabla\nabla E(\mathbf{w}) = \Phi^T \mathbf{R} \Phi. \tag{3.12}$$

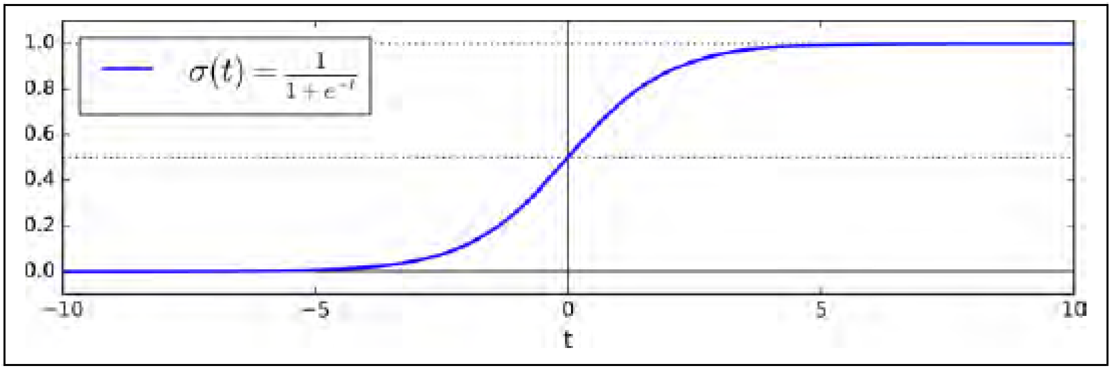


Figure 3.4: The logistic function represented by a sigmoid function [2]

		Predicted species		
		Setosa	Versicolor	Virginica
Actual species	Setosa	3	2	3
	Versicolor	4	2	3
	Virginica	2	6	5

Table 3.2: Confusion matrix for the logistic regression implementation

Where **R** is an $N \times N$ diagonal matrix with elements like

$$R_{nn} = y_n(1 - y_n). \tag{3.13}$$

Logistic regression have been implemented and trained on the data set described above in Section 3. When using the whole data set, Python's Sickit-Learn model for logistic regression is used. To represent the performance of the model a confusion matrix and accuracy score is used. After fitting the data to our model and making the predictions, the confusion matrix turned out to be the following.

The first row in a confusion matrix represents the actual label of the first target variable, while the first column represents the predicted label of the first target variable. The second row and column represent the second target variable and the same for the third. This means that the diagonal of the matrix represents the correct predictions, as described in Section 3. From the matrix in 3.2 it can be seen that the first element on the diagonal is 3, which means that three Setosas were correctly predicted. From the rest of the elements in the first row and column it can also be interpreted that $2+3+4+2 = 11$ Satosas were predicted incorrectly. The same

		Predicted species	
		Septosa	Not Septosae
Actual species	Septosa	9	0
	Not Septosa	0	10

Table 3.3: Confusion matrix for the logistic regression implementation on the sub dataset

interpretation can be done for the rest of the target variables, Versicolor and Virginica. It can see that the overall performance is quite bad. There exists more incorrect predictions than correct ones. This also explains why the accuracy score is 0.333. When trying logistic regression on only the subset of the data set, which is linearly separable, the performance is better. The confusion matrix is

which shows that all of the predictions are correct, and therefore the accuracy score is 1. By this it can be seen that the logistic regression performs better when there are fewer target variables and features, and the dataset is linearly separable.

3.3 Support Vector Machine

The SVM algorithm is kernel-based, which means that the kernel function only needs to evaluate a subset of the training data points to predict output based on new input. SVM can perform linear or nonlinear classification, regression and even outlier detection. In this paper the focus will be on SVM for classification to get a basic understanding. To find the model parameters a convex optimization problem has to be solved. This means that any local solution is also a global solution. Posterior probabilities are not the output for this method because SVM is a decision machine.

3.3.1 SVM for Classification

To explain how the SVM a two-class classification problem will be introduced. The linear model for this problem looks a lot like the one described previously for one-class classification in (3.2). The only difference is that the input representation, $\mathbf{x}$, needs to be replaced with a fixed feature space transformation $\phi(\mathbf{x})$. The equation is

$$y(\mathbf{x}) = \mathbf{w}^T \phi(\mathbf{x}) + b. \tag{3.14}$$

In (3.14) it can be seen that the bias b has been made explicit. The data set for training consist of input vectors $\mathbf{x}_1, ..., \mathbf{x}_N$ and their corresponding binary target values $t_1, ..., t_N$.

It will be assumed that the training data is linearly separable. A perceptron algorithm can be used to find a solution but is dependent on the chosen initial bias, weights and the order of the data points. This algorithm uses the *perceptron criterion* to find the weights $\mathbf{w}$. Roughly explained, this algorithm tries to minimize the quantity $-\mathbf{w}^T \phi(\mathbf{x}_n)\, t_n$ for all incorrectly classified patterns, which is called the error function E_p. To find the change in the weight vector $\mathbf{w}$ for each iteration the *stochastic gradient descent* (SGD) algorithm is applied to the error function defined above. The gradient descent was truly proposed in 1847 by M. Augustin Cauchy [24], and was not utilized in ML purposes until many years later [2]. Applying the SGD algorithm iteratively results in the following expression

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \Delta E_p(\mathbf{w}) = \mathbf{w}^{(\tau)} + \eta \phi_n t_n. \tag{3.15}$$

The aim of this (3.15) is to find the weights $\mathbf{w}$ and bias b that maximizes the margin. where the learning rate is described by η and the current iteration of the algorithm is represented by τ. It can be seen from (3.15) that the weight vector is only changed when the pattern is incorrectly classified. If there exists more than one solution that classifes the training data exactly, then the one which generalizes best should be chosen. SVM does this by using the smallest distance between the decision boundaries and any of the samples, which is called the *margin* concept. The decision boundary is found by using *statistical learning theory* to maximize the margin. This is done as shown below

$$\underset{\mathbf{w},x}{\arg\max} \left\{ \frac{1}{||\mathbf{w}||} \min_n [t_n(\mathbf{w}^T \phi(\mathbf{x}) + b)] \right\}. \tag{3.16}$$

Equation 3.16 is quite hard to solve and will therefore be simplified by utilizing some of its properties. First it can be noticed that there exists a gain κ for $\mathbf{w}$ and b such that the distance from any given point $\mathbf{x}_n$ to the decision surface stays unchanged. By utilizing this property the part that is aimed to be minimized in (3.16) can be constrained to one for the point that is closest to the surface, and greater than one for all the other points. To find the maximum of $\frac{1}{||\mathbf{w}||}$ the *quadratic programming* problem is used. This results in the following expression,

$$\underset{\mathbf{w},x}{\arg\min} \frac{1}{2}||\mathbf{w}||^2, \tag{3.17}$$

where the factor $\frac{1}{2}$ is just there for practical reasons. This is a constrained optimization problem that can be solved by the Lagrangian function,

$$L(\mathbf{w}, b, \mathbf{a}) = \frac{1}{2}||\mathbf{w}||^2 - \sum_{n=1}^{N} a_n\{t_n(\mathbf{w}^T\phi(\mathbf{x}_n) + b) - 1\}, \tag{3.18}$$

where $\mathbf{a} = (a_1, ...a_N)^T$ is the Lagrange multipliers. It can be seen from (3.19) that $\mathbf{w}$ and b is being minimized and $\mathbf{a}$ is maximized. The Lagrangian (3.19) is derived with respect to $\mathbf{w}$ and b, and these two equations are set equal to zero. Inserting the result in (3.19) the *dual representation* of the maximum margin problem is obtained,

$$\tilde{L}(\mathbf{a}) = \sum_{n=1}^{N} a_n - \frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{M} a_n a_m t_n t_m \kappa(\mathbf{x}_n, \mathbf{x}_m), \tag{3.19}$$

which is maximized with respect to $\mathbf{a}$. The kernel function is here defined as $\kappa(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x})^T\phi(\mathbf{x}')$. Reformulating the model using kernels makes the application of the maximum margin classifier efficient even to infinite feature spaces. From the Lagrangian constraints it is known that the kernel function is positive definite, because the Lagrangian function needs to be bounded below. To classify new data points the following equation can be found by using the partial derivative of the Lagrangian function with respect to $\mathbf{w}$ and (3.14) and the resulting function is

$$y(\mathbf{x}) = \sum_{n=1}^{N} a_n t_n \kappa(\mathbf{x}, \mathbf{x}_n) + b. \tag{3.20}$$

Since the Lagrangian optimization problem is utilized the *Karush-Kuhn-Tucker* (KKT) conditions have to be satisfied

$$a_n \geq 0 \tag{3.21}$$

$$t_n y(\mathbf{x}_n) - 1 \geq 0 \tag{3.22}$$

$$a_n \{t_n y(\mathbf{x}_n) - 1\} = 0. \tag{3.23}$$

From constraint (3.21) and (3.20) it can be seen that the data points with their corresponding a_n equal to zero will not affect the prediction of new data points. The data points with a corresponding a_n greater than zero are called $support vectors$. From constraint (3.23) the $t_n y(\mathbf{x}_n) = 1$ has to be true, and therefore the support vectors lie on the maximum margin hyperplanes in feature space. Once the model is trained only the support vectors are left and the rest of the data points can be discarded.

The quadratic problem is now solved and the only thing left is to find the bias b. This can be done by utilizing (3.20) and constraint (3.23) which gives,

$$b = \frac{1}{N_s} \sum_{n \in s} (t_n - \sum_{m \in s} a_m t_m \kappa(\mathbf{x}_n, \mathbf{x}_m)), \tag{3.24}$$

where N_s denotes the total amount of support vectors.

For the implementation of SVM Python's Iris data set is used which is explained more in detail in Section 3. First of all the linearly separable subset will be used, which means that a linear kernel can be used. Using a linear kernel is beneficial because the training with a linear kernel is faster than any other kernel. It is also only required to optimize one parameter when training, which is the regularisation parameter. With low regularization some errors could occur, but it might even be better for the model. The decision boundary is to be chosen, and it is desired to choose the one with the highest margin. This is because it classifies the two different groups in a better way. This is what the SVM tries to do, choose the decision boundary with the highest margin. The data points nearby the chosen boundary are the ones called support vectors, as explained

		Predicted species		
		Setosa	Versicolor	Virginica
Actual species	Setosa	8	0	0
	Versicolor	0	8	1
	Virginica	0	0	13

Table 3.4: Confusion matrix for the SVM implementation

in the theory section above. It is also chosen to focus on only two of the species because then the decision boundary is a straight line instead of a plane, which is done to compare the result with the classification gotten with logistic regression. Python's Scikit-Learn Linear Support Vector Classification (LinearSVC) is used, with the default regularization parameter which is equal to 1. The result is actually the same as in the confusion matrix for logistic regression 3.3. The maximum number of iterations to be run for the model is kept at the default of 1000 iterations. When using SVM on the whole dataset, the decision boundary is a hyperplane because it includes more than three target values. The LinearSVC is also used for this dataset. When using the default maximum number of iterations at 1000, the model did not manage to converge. This parameter was changed to 4000 maximum iterations to run. For the rest of the parameters the default values were kept. The result is shown below.

The matrix in 3.4 shows a good prediction where only one species was incorrect predicted, which means that the accuracy score is 0.967. By being able to choose the kernel in the SVM model it is possible to solve complex problems, but it is quite difficult to choose a "good" kernel function. Since 4000 iterations were needed when using the LinearSVC model on the whole dataset, another kernel could be better.

3.4 Decision Trees

In a decision tree the input space is recursively partitioned into cuboid regions aligned with the axes of the feature space. Each region is assigned a simple model. Only one of these models is responsible for making predictions at any given point. The model is chosen through the traversal of a binary tree. In this section the focus will be on classification trees.

3.4.1 Decision Trees for Classification

After dividing the input space into cuboids, one can classify new input by starting at the root node and then following a certain path based on criteria at each node. When arriving at the leaf node of the tree the specific class that this input belongs is found. This means that every leaf node represents one cuboid region, and this region will represent a class in a classification problem. Decision trees determine which class an input belongs to, not the probability that an input belongs to a class as for logistic regression described above. In classification, the model that is assigned to each region, is therefore a specific class. One of the reasons why decision trees are so popular is of because how intuitive the method is for humans. For instance, as will be shown below, when trying to classify if an iris flower is a certain species the sequence of binary decisions is taken by comparing the flower with the criteria. To be able to learn such a model, a few parameters have to be set. A greedy algorithm can be used to grow a tree. This way a large tree will be grown using a stopping criterion based on the number of targets for classification, and then prune the tree afterward. Roughly explained, the pruning is done by removing certain nodes by merging regions/classifications that correspond to each other. In the code example for classification below, a *Gini* criterion is used. The *Gini index* can be expressed as follows

$$Q_\tau(T) = \sum_{k=1}^{K} p_{\tau k}(1 - p_{\tau k}) \tag{3.25}$$

where $p_{\tau k}$ is the proportion of data points in region R_τ assigned to class k. τ represents a leaf node, and T is the total amount of leaf nodes. The cross-entropy error for classification problems are

$$Q_\tau(T) = \sum_{k=1}^{K} p_{\tau k} ln p_{\tau k}. \tag{3.26}$$

Equation 3.26 and 3.25 is used to measure the performance. It can be seen that both the Gini index and the cross-entropy error is zero for the proportion of data points in a region equal to 1 and 0. The equations are both maximum when this proportion is 0.5. This way the model tries to create regions that represent a class with a high proportion of data points. The general

		Predicted species		
		Setosa	Versicolor	Virginica
Actual species	Setosa	8	0	0
	Versicolor	0	8	1
	Virginica	0	0	13

Table 3.5: Confusion matrix for the decision tree implementation

optimal prediction for region R_τ is

$$y_\tau(T) = \frac{1}{N_\tau} \sum_{\mathbf{x}_n \in R_\tau} t_n. \tag{3.27}$$

Even though the tree structure is intuitive for humans,it is sensitive to small changes to the training data. Another drawback with the tree method is that the region splits are aligned with the axis which might not always be optimal when it comes to finding the decision boundary between two regions.

A decision tree is implemented to solve the same classification problem as in the previous sections, classify the iris species. The algorithm is used both on the whole data set with four features and three target values, and on a subset of it where there are only two features and classes which are linearly separable. Using the decision tree model on the subset gives the same result as for logistic regression 3.2. The result for the whole dataset is the same as for SVM and is shown below.

Different tree depths were tested. The best result is the one shown in the matrix above 3.5 with the tree depth of three. An advantage of this algorithm is that the visual representation of the tree is quite easy to understand, as can be seen in Figure 3.5 which represents an example , retrieved from [2], of a decision tree for the Iris dataset.

3.5 Comparison

As shown above, logistic regression, SVM and decision trees all work for classification problems. Logistic regression is dependent of linearly separable classes, and if this is not the case the data

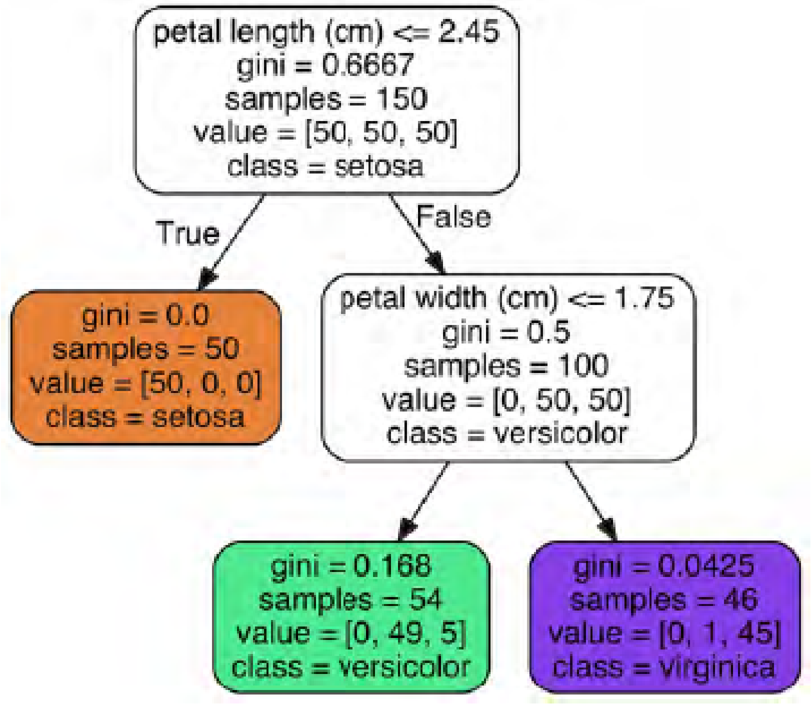


Figure 3.5: Decision tree example of the Iris dataset retrieved from [2].

needs to be transformed. SVM, on the other hand, projects the feature space into kernel space which makes the classes linearly separable. This means that it does not matter if the classes are linearly separable or not when using the SVM algorithm. A drawback with SVM is that it might be difficult to choose a "good" kernel. Decision trees will try to fit the decision boundary as well as it can be based on the depth of the tree. That is, if the classes are not linearly separable more depth is needed to make a decision boundary fit. It has also been showed that the logistic regression gives probabilities as output. With this method, the threshold can be adjusted to get different classification for the same model. For the subset used here the logistic regression works quite well, the problem comes when the data includes a lot of target variables and features. This is the reason why more advanced methods like decision trees and SVMs are needed. When a lot of data is available, NNs will in most cases perform better than decision trees and SVMs.

Chapter 4

Neural Networks

This section is mainly based on knowledge from [2, 4, 5].
The ANNs have truly evolved from a collaboration between neuroscientists and computer scientists looking into what intelligence is [11]. Tomaso Paggio is a professor at Massachusetts Institue of Technology and started looking at the problem of intelligence around 1990[11]. He was making computer vision algorithms for detecting faces and people in street scenes. He started collaborating with experimental neuroscientists about how the brain detects faces and people. Neuroscientists have known that the brain's visual system is built up by a hierarchy of areas since the 1960s [11]. Paggio tried to mimic the brain in his model for visual recognition. In Paggio's model the low levels recognize edges and lines and the higher ones could turn the edges and lines into object parts and then objects. The brain is built up by 100 billion neurons communicating [11]. The neurons in the brain are the ones doing the recognition, and from communicating it among each other the neurons in the highest level can detect the objects. The word "neural" in NNs comes from the neurons in the brain. The NN is a set of algorithms loosely modeled after the human brain.

An example of a NN can be one for recognizing a person in an image, as what Paggio looked into. The overall goal is to map the input which is an image with a person in it, to the output which is the recognized person.

In the previous Section 3, we looked at how a system can acquire knowledge by extracting patterns from raw data. We found a few features that we decided to extract for the task and provided them to the used ML algorithm. Sometimes these features might be hard to find. To make the algorithm recognize a face, you know that one of the features to look at is the nose. But it is quite hard to explain what a nose looks like. When designing the features you also need to separate the *factors of variation* that explain the observed data [4]. These factors can be looked at as abstractions that are needed to make sense of variability in the data. For instance, when the algorithm is trying to recognize the face from the picture, factors like the angle of the face and brightness of the sun are important. The pixels colors might look a bit different than they are by night for instance. Due to this we need to find the factors of variation that is important for the specified task and take these into consideration. The only problem is that these factors might be quite difficult to extract and therefore when looking at this as one mapping, the task is very complex. By dividing this mapping into smaller, nested mappings the task is less complex. This is exactly what Paggio did when he decided to build his vision recognition model based on the brain's hierarchy of areas. The mapping of input to output is done by processing the input by a set of functions, and then pass the output to the next layer. A layer, in a NN, represents the state of the computer's memory after executing another set of instructions in parallel [4].

The first layer's task can be to identify edges by comparing the brightness of neighboring pixels. The output from the first layer is given to the second layer, and the second layer can look for corners and extended contours. By using the second layers explanation of the image by corners and contours, the third layer can find specific collections of contours and corners which will result in entire parts of objects. The fourth layer can use the third layer's description of the image by object parts to recognize the different objects in the image. We have now solved the overall task of mapping the input image to the output as a recognized object by dividing the mapping into smaller, nested mappings. This is how NNs generally work by learning a concept at each layer and communicating it to the other layers.

The difference between DL and NNs is the "deep" part, which means that DL has more learned concepts or a greater amount of compositions than NNs [4]. The definition of how many learned concepts or compositions that are needed to be a deep NN is a bit vague. DL is a branch in ML which learns to represent the world by a nested hierarchy of concepts, where each concept is represented by simpler concepts [4]. This way DL achieves great power and flexibility. The rest of Section 4.2 will look at deep feedforward- and convolutional networks.

4.1 Deep Feedforward Networks

To explain what deep feedforward networks are, we will start by looking at what a Perceptron is. The Perceptron was truly intevented by Frank Rosenblatt in 1957 [25], and is one of the simplest ANN architectures. Rosenblatt's perceptron contributed to the first popularity wave of ANN [2]. The neurons in a Perceptron has numbered inputs with weights. A linear threshold unit (LTU) sums the weighted inputs and puts the result in a step function. This step function is typically a Heaviside function, where the output is dependent on the weighted sum of the inputs. Perceptrons are based on linear models, which means that they cannot learn XOR functionality for instance. The XOR learning inability, among other limitations the Perceptron has, is pointed out by Marvin Minsky and Papert Seymour in the book *Perceptrons: An Introduction to Computational Geometry* [26]. When flaws like the ones mentioned in Minsky- and Seymour's book where known, it backlashed against NN approach.

Some of the Perceptron's limitations, like learning the XOR functionality, can be fixed by introducing the Multi-Layer Perceptron (MLP). A MLP consists of stacked Perceptrons, and is also called feedforward NN. The network is called feedforward because the information flows from the input $\mathbf{x}$, through the intermediate layers with the computations used to define the approximated function f, and at the end the information goes to the output $\mathbf{y}$. The approximated function, f, is formed by each layer's sub function, where every layer's function uses the previous layer's function. The layers between the input and the output layers are called hidden layers because the input does not include a description of what each layer should do to create the output. The hidden layers contain hidden units, and the output of every unit in one layer is connected to the input of every unit in the next layer. Having this connection between the units means that the feedforward network has fully connected layers. The algorithm itself has to choose what each layer should be to find the best approximated function. The overall goal of a feedforward network is to approximate some function, f^*, to generate the most accurate prediction of the output based on the input.

$$y = f(\mathbf{x};\boldsymbol{\theta},\mathbf{w}) = \phi(\mathbf{x};\boldsymbol{\theta})^T\mathbf{w} \tag{4.1}$$

An example of a mathematical representation of the feedforward NN is showed in (4.1). The parameters $\boldsymbol{\theta}$ are used to learn the hidden layer's function, ϕ. $\mathbf{w}$ is mapping the learned function, $\phi(\mathbf{x})$, to the output, y. The hidden layers functions can be called *activation functions*. The activation function will compute the layer's values. In todays NNs, it is normally recommended to use the rectified linear unit (ReLU) as activation function. A ReLU computes a linear function of the inputs and outputs the result if it is positive, and 0 otherwise [4]. For the feedforward network to be able to learn, the gradients of complicated functions are needed. We call this *gradient-based learning*.

To train NNs, iterative, gradient-based optimizers that derive the cost function to a very low value are usually used [4]. That is, the training algorithm is based on using the gradient to descend the cost function for the feedforward NN. This means that the training of a NN is not very different from the training of the basic ML algorithms described in Section 3. Computing the gradient for a NN might be a bit more complicated than for the basic ML methods. An efficient way of computing the gradient is built on the mathematical chain-rule concept. This principle is called the *back-propagation* algorithm, which was presented in 1986 by David E. Rumelhart and other researchers [27]. After this algorithm was introduced, NNs gained popularity and had a peak in the 1990s. Today's feedforward NN has approximately the same back-propagation and approach to gradient descend as in the 1980s. The feedforward NN itself does not use the back-propagation algorithm, but the back-propagation algorithm uses the feedforward NN. The network is used to feed forward the values from input to output, and then the back-propagation algorithm will calculate the error and propagate it back to the previous layers. That is, the algorithm will go through the network in reverse to measure each layer's error contribution from each connection. The gradient descend is used after the back-propagation algorithm to adjust the weighted connections to reduce the overall error. The error that will propagate back is found by utilizing a cost function.

For NNs the cost function is usually defined as the cross-entropy between the training data and the model's predictions plus a regularization term, which has already been mentioned for basic ML algorithms in Section 2. The regularization term in the cost function is used to make the model generalize well. That is, to make the model not only perform well on the training data but also on new instances. From Section 2 we know that this means to avoid overfitting. The regularization term in the loss function therefore penalizes for large weights. One type of regularization that can be used by a broad family of models is called *dropout*. This technique to avoid overfitting will randomly ignore or dropout some hidden units in a given layer.

Saying that the NNs is trained using the cross-entropy error is equivalent to the negative log-likelihood, as discussed in Section 3. To be able to compensate for the error, the negative log-likelihood is minimized. The minimization can be done by using a gradient descend algorithm.

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{,\sim\hat{p}} \quad \log p_{model}(\mathbf{y}|\mathbf{x}) \tag{4.2}$$

(4.2) shows a general form of the cost function for a NN. The form of this cost function depends on the model.

We want to find the gradient, $\nabla J(\boldsymbol{\theta})$, of the cost function with respect to the parameters. The evaluation of the gradient is done by the back-propagation algorithm. Each layer has to change its weights according to a back-propagated error message from a later layer, and calculate an error message for the previous layer. The error is calculated using the gradient, and this is done efficiently with the back-propagation algorithm. Let us say that we have the input and output vectors $\mathbf{x} \in \mathbb{R}^m$ and $\mathbf{y} \in \mathbb{R}^n$ respectively, and g maps from $\mathbb{R}^m$ to $\mathbb{R}^n$. This means that g is the activation function defined for each hidden unit. A function f maps from $\mathbb{R}^n$ to $\mathbb{R}$, $\mathbf{y} = g(\mathbf{x})$ and $z = f(\mathbf{y})$. Then we can use the chain rule in the back-propagation algorithm to get the gradient as follows.

$$\nabla z = \frac{\partial \mathbf{y}}{\partial \mathbf{x}}^T \nabla z \tag{4.3}$$

Equation (4.3) shows that the gradient can be computed using the chain rule, which means that the gradient of a variable $\mathbf{x}$ is computed by multiplying the Jacobian matrix $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ by the gradient

∇z for each node in the graph [4]. This technique is used to find $\nabla J(\boldsymbol{\theta})$. Normally the back-propagation algorithm is applied to tensors rather than vectors, but the principle is the same as in (4.3) by doing some rearranging in the tensor before we run the algorithm.

The SGD is used to perform learning utilizing the gradient found by the back-propagation algorithm. The SGD algorithm finds an estimate of the gradient by finding the average gradient on a minibatch of m examples drawn independent and identical distributed from the data generating distribution [4]. The learning rate is a crucial part of the SGD, and it is gradually decreasing throughout the algorithm until a certain iteration is reached. The iterations after this iteration will have a constant learning rate.

Batch Normalization (BN) is usually used in the gradient descend algorithm to optimize the learning. BN addresses the problem that the distribution of each layer's input changes during training [2]. This happens because the parameters of the previous layers change. The BN operation is done before the activation function in each layer. The operation includes scaling and shifting of the layer's input. BN makes it possible for the NNs to have a larger learning rate, and make them less sensitive to weight initialization. Without using the BN, the exploding gradient could occur which can make the learning unstable. The gradient is found by looking at the difference between the predicted values and the actual values, which means that if the error is big the gradient will get big and could "explode".

We have now seen how the feedforward nerual network operates, and how it learns by using gradient-based learning. The deep feedforward network is a fully connceted network and therefore it has a lot of paramteres to tune for complex data. In the introduction of Chapter 4 we mentioned an example of recognizing a person in an image. This will work fine with deep feedforward NN if the image is small, but with larger images the network will break down [2]. To perform image recognition on larger images a specialized kind of deep feedforward networks called convolutional networks can be used.

4.2 Convolutional Networks

The human perception of differentiating objects seems effortless, but for a computer this task is extremely complex. The perception happens outside the human's consciousness, within specialized visual, auditory, and other sensory modules in our brains [2]. This type of NNs, which also goes under the name *convolutional neural networks* (CNNs), are specialized in processing data that has a known grid-like topology [4]. The CNNs are typically used for processing images, which has a 2D grid of pixels, and has occurred from studying the visual cortex of the brain. [2]. These networks performs better on image recognition for large images than the deep feedforward NN because it has partially connected layers. This way the CNN has less parameters to tune than the feedforward NN. The CNNs also uses convolution instead of matrix multiplication, as the deep feedforward NNs uses, in at least one of the layers. The neurons in the first convolutional layer is not connected to every pixel in the input image, instead it is just connected to neurons located in a small rectangle of the input image [2]. Every convolutional layer's neurons in the CNN is only connected to a small rectangle of the neurons in the previous layer. For a layer to have the same height and width as the previous layer, *zero padding* is used around the input.

The input of a convolution in ML is usually a multidimensional array of data. The kernel is usually a multidimensional array of parameters that the learning algorithm adapts. The kernel represents the neuron's weights, and can also be called a *filter*. A multidimensional array will from now on be called a *tensor*. We can represent the discrete convolution between a filter and an image as done below.

$$s(i,j) = (K * I)(i,j) = \sum_{m}\sum_{n} I(i-m, j-n)K(m,n) \tag{4.4}$$

In (4.4) the input is the image I, and the kernel is also two-dimensional and is represented by K. The convolution has a commutative property because the kernel in this example is flipped relative to the input. This is a property that is not needed for the implementation of NNs. Therefore we say convolution, but the networks are normally using *cross-correlation*, which is the same as convolution except it does not do the flip operation. In this paper the same convention will be used, and it will be specified if the kernel is flipped. A representation of the convolution without a flipped kernel, the cross-correlation, is shown in Figure 4.5.

$$s(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{4.5}$$

This convolution is used in the layers to recognize patterns like edges, object parts, full objects and so on. The kernel is convolved with the input in the convolutional layer, and the result, $s(i,j)$, is the output that is given to the next layer. A layer of neurons using the same filter gives a *feature map* where the pixels in the image that are similar to the filter are highlighted [2]. Multiple feature maps like this are stacked upon each other and compose a 3D convolutional layer. The layer does multiple convolutions simultaneously to its input, where each feature map has its weights and bias [2]. An image also consists of layers, which are called channels in the image, where a colored image has three layers. One layer for red, one for green and one for blue.
To understand how this discrete convolution works in practice, you can look at it as a matrix multiplication. Where the matrix have certain constraints depending on the input.

The interaction between input and output in CNNs is referred to as *sparse interactions* or *sparse weights*. Compared to traditional NNs where every input interacts with every output, convolutional networks have a small kernel that only occupies some of the important pixels. We see that there is an obvious improvement in efficiency.
Another difference from traditional NNs are the concept of *parameter sharing*, which means that each parameter can be used for more than one function in a model in convolutional networks. As mentioned earlier, traditional NNs use matrix multiplication between the input matrix and the weight matrix. This means that each element in the weight matrix is only multiplied with one of the input elements, we call this *tied weights*. This means that we need to learn a set of parameters for every location. With CNNs on the other hand, each parameter of the kernel is used by every input parameter, with a few expcetions. By using this method the network only needs to learn one set of parameters, which reduces the storage.
The CNNs also have a *equvariant representation*, which means that if the input changes, the output changes the same way. For example if we shift the input by time, the output will be the same as before but shifted the same amount of time as the input.

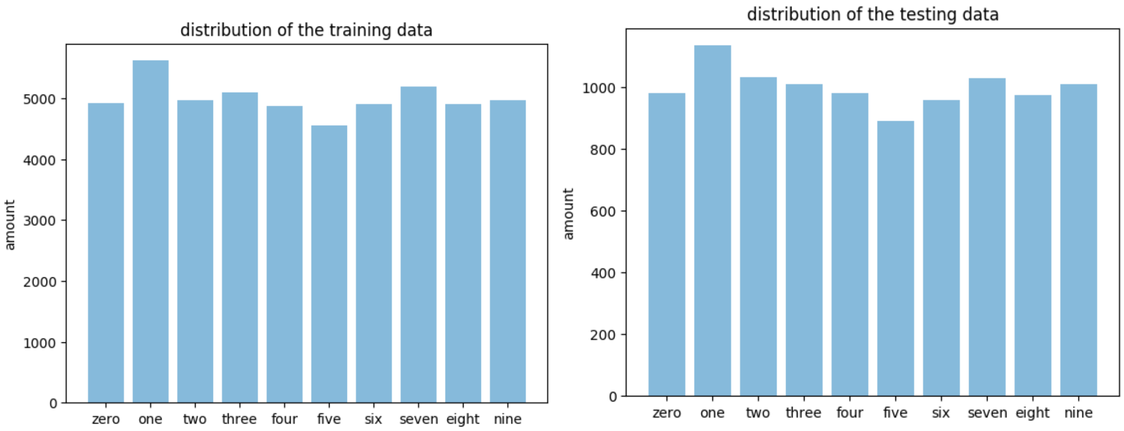
A convolutional layer typically consists of three stages. The first one does the convolutions in parallel and gives its output, a set of linear activation, to the next stage. Stage two feeds the linear activation into a nonlinear activation function, sometimes called the detector stage. In the last stage, a pooling function is used to for instance make the representation more robust against small translations in the input, invariance. This becomes handy when we want to check if a feature is present rather than exactly where it is. Pooling can also be used to reduce the size of the representation to speed up the computation and to make it possible to use input of different sizes. Different pooling functions exist to make the wanted output for the specific task. A typical pooling function can be max or mean. Say for instance the max pooling kernel is of size $2x2$, and the input is of size $4x4$. When the pooling kernel moves over each quadruple of pixels, it only keeps the maximum value. The pooling kernel that will be sent to the next layer will therefore consist of 4 pixel elements, where each represents the maximum of its quadruple in the input. 75 percent of the input is dropped in the output. The neurons in the pooling layer do not have any weights. The pooling layer's task is only to optimize the convolutional layer. A clustering algorithm can, for example, be used to dynamically pool features together.

Dataset

The first dataset that will be used to implement CNN is the Mixed National Institute of Standards and Technology (MNIST) dataset of handwritten digits. The MNIST dataset is distributed by Yann LeCunn's "THE MINST DATABASE of handwritten digits". This dataset consists of grey-scale images with handwritten digits. Grey-scale images have just one channel, compared to the colored images with three channels. Each image has a corresponding label of which digit is represented by the image. The dataset consists of 60000 pairs of handwritten digits and labels for training, and 10000 for testing. The training set is split into 9999 data set for validation of the model, and 50000 for training. One sample is left out to make sure that the validation- and training set do not have any similarities. This dataset will be used for image recognition classification, and the result is showed in the implementation section. Before implementing CNN and applying it to the MNIST dataset, the dataset needs to be analyzed.

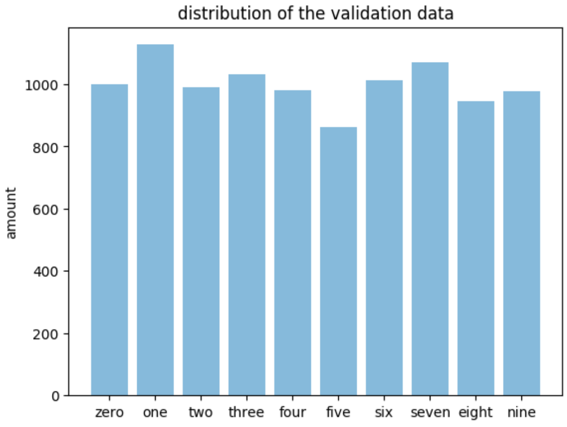
The matrix of an image consists of $28x28$ pixels, which means that the image is relatively small.

Figure 4.1 shows the data distribution for the train-, test- and validation data. All of the datasets have the most occurrences of the digit one, and it could look like the second most occurred digit is number seven. These two digits might be a bit hard to differentiate because of their similar appearance. The digits six and eight might also be a bit hard to differentiate because of their appearance. Since the distribution for all of the dataset is quite similar, digits that might be hard to differentiate should affect the datasets more or less equally. By looking at the different images in the datasets there is no obvious difference in quality when it comes to resolution and bad handwritten digits. If the resulting performance for the implementation acts strangely on the different datasets this could be investigated more carefully.



(a) Training data distribution

(b) Test data distribution

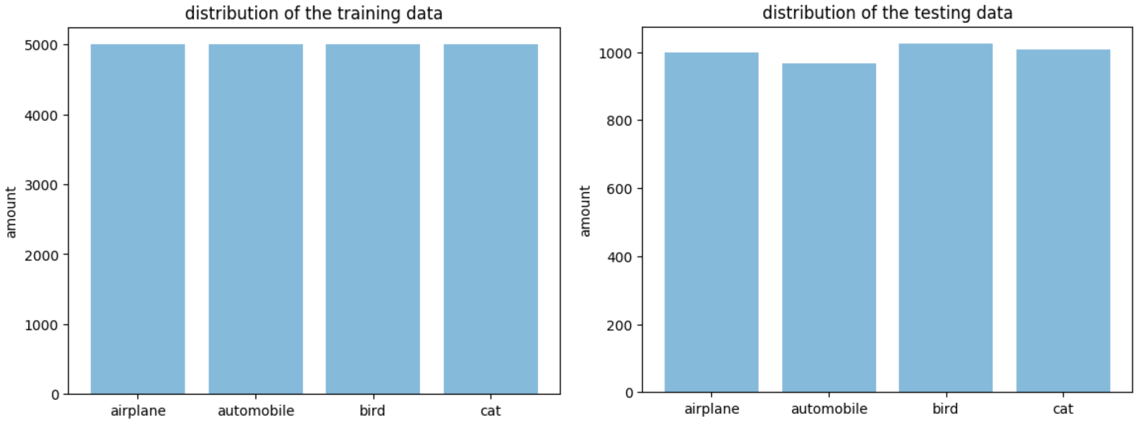


(c) Validation data distribution

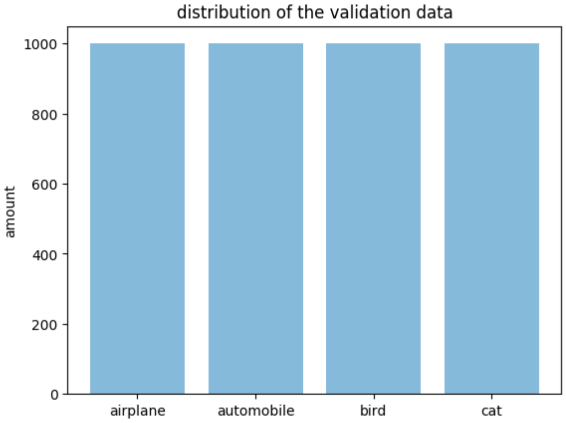
Figure 4.1: Data distribution for the MNIST dataset

The second dataset that will be used for implementing a CNN is the CIFAR10 dataset. The CIFAR10 includes 60000 32×32 pixel color images, which implies that these images are a bit bigger than the MNIST images. The 60000 images are divided into ten classes which are airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck. These classes are completely mutually exclusive, which means that an image can only be in one of the classes. Since the images are colored, compared to the ones in the MNIST dataset, each pixel will have three channels. Each image has a corresponding class label. The 60000 sets of image and label are divided into 50000 for training and 10000 for testing. To decrease computation time only the first four classes will be used in the implemented CNN. The training data therefore consists of 20000 datasets, the validation data consist of 4000 datasets and the test data consists of 3999 datasets.

Figure 4.2 shows the data distribution of the train-, test- and validation data. The distribution for the train- and test data appears to be quite similar and uniformly distributed. The test distribution, on the other hand, stands out with more variation in the amount of data for each class. The bird class seems to have the most data, and the automobile class seems to have the least amount of data. If it is harder to interpret a bird than an automobile, this could be a reason why the performance is worse on the test data or vice versa.



(a) Training data distribution (b) Test data distribution



(c) Validation data distribution

Figure 4.2: Data distribution for the CIFAR10 dataset

Implementation

CNN for the MNIST Dataset

The first implementation of CNN to perform image recognition is done in Python using the MNIST dataset described in Section 4.2. The goal is to predict the correct handwritten digits with good accuracy. The implementation is built on knowledge from [2, 5].

The only feature engineering operation done is normalization on the images. This is done by dividing every feature in the 28×28 matrix, representing an image, by 250.0. This way every feature, pixel, is in the range [0, 1].

Tensorflow's *Keras* is used to create a sequential model in which layers are added to. Keras is a high-level API for Tensorflow. The knowledge of the Keras implementation is obtained from

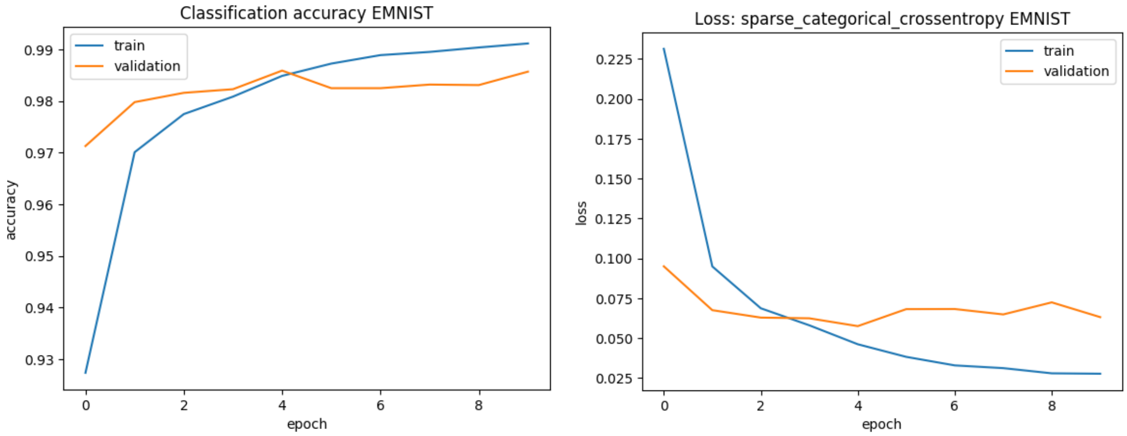
Tensorflow's homepage [5]. TensorFlow is developed by Google and has an emphasis on the manipulation of tensors, hence the name [10]. The first layer added is a 2 dimensional convolutional layer with 32 nodes or neurons. The layer's kernel is of size 3×3 and the activation function used is the ReLU function mentioned in Section 4.2. This is the only convolutional layer in this network, but it is enough for the network to be convolutional. It also has a *he uniform* kernel initializer.

The next layer added is just a flatten layer with no parameters. This layer flattens the dimension of its input, but does not affect the batch size. The flatten layer serves as a connection between the convolutional and dense layer. The flatten layer is thus followed by a dense layer, with 128 nodes and the ReLU activation function. A dense layer is a fully connected layer, which is explained in Section 4.2. The last layer is also a dense layer with 10 nodes but with the activation function *softmax*. The softmax activation function is used in the output layer to output the estimated class probabilities for the 10 mutual exclusive classes [2]. The epochs are set to ten, which means that this dense layer will go through the training set ten times. The number of epochs is set to the number of classes in the dataset.

This CNN model is compiled with the *Adam* optimizer and the *sparse categorical crossentropy* loss function. Finding the correct learning rate for the network might be tricky. If it is too high the training could diverge, and if it is too low the training will take quite a while to converge to the optimum [2]. Adam is an adaptive learning rate optimization algorithm, which makes the implemented CNN converge to the optimum, but it still takes some time. The first CNN created had a problem with overfitting, that is, the accuracy for the training data was quite higher than the one for the validation data. To solve this problem different regularisation techniques were tested. A dropout layer with a dropout of 0.5 was added to the model. The dropout layer is only in the model to optimize the its performance, but it slows down training. A dropout of 0.5 means that there is a 50 percent chance that the outputs from the previous layer, which will be the input to the next layer, is set to zero or dropped out. Adding this layer made the accuracy gap between the training data and the validation data smaller. Adding another dropout layer with the same dropout rate between the flatten and the dense layer resulted in an even smaller gap. The CNN implemented for the EMNIST dataset can be shown in Table 4.1.

Layer	Neurons	Kernel initializer	Kernel size	Activation function
Convolutional	32	He uniform	3×3	ReLU
Dropout	0.5 rate	-	-	-
Flatten	-	-	-	-
Dropout	0.5 rate	-	-	-
Dense	128	He uniform	-	ReLU
Dense	10	He uniform	-	Softmax

Table 4.1: The CNN architecture of the implementation for the MNIST dataset



(a) Accuracy

(b) Loss

Figure 4.3: CNN implementation for MNIST dataset evaluation

Figure 4.3 shows the resulting loss and accuracy for the training- and validation data. From the figure it looks like the training accuracy starts at approximately 0.93 in the very beginning of the first epoch. The CNN learns the relatively simple MNIST dataset quite fast, which means that even though the accuracy starts low as expected it goes up so fast that it is not showing in the figure. The validation accuracy is calculated after the model has trained on the training dataset and is therefore not expected to start low. Figure 4.3 shows that the validation accuracy starts at approximately 0.97 which is quite high. One reason for this could be that the validation data is quite similar to the training data and is therefore simple for the model to interpret. As shown in Section 4.2 the distribution of the training- and validation data is quite similar. For the first three epochs it looks like the model is underfitting the training data, but after epoch four the

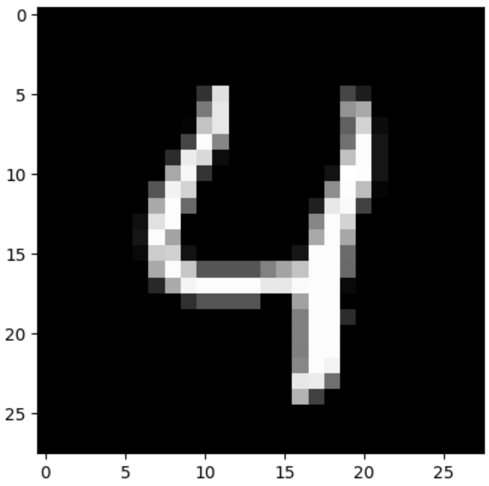


Figure 4.4: Digit number four in the test set for the MNIST data

model is overfitting the data. The overall loss for the training data is decreasing. The loss for the validation data is increasing until epoch four where it starts to increase, which is another proof of overfitting. The best accuracy achieved for the training data is 0.9912, and for the validation data it is 0.9857. The test accuracy is 0.986 and the test loss is 0.056 which is quite good!

The CNN implementation with its performance showed in Figure 4.3 predicted the digit in Figure 4.4 with a probability of 1 to be a digit number four. The next highest probability was of 1.2133750×10^{-6} for the digit number nine. This is a quite low probability, but the probabilities for the other digits starts at 10^{-11} and gets lower. It makes sense that the digit number nine has a quite high probability compared to the others since the digit in Figure 4.4 could be a number nine if it was closed on the top. As explained in the theory Section 4.2, the CNN looks at parts of the digit and puts the information together at the output of the network to give a probability. By not looking at the top of the digit in Figure 4.4 it could be both a four and a nine.

CNN for the CIFAR10 Dataset

A CNN is also implemented to perform image recognition on the CIFAR10 dataset described in 4.2. The goal is to predict the correct animals and vehicles with good accuracy. The implementation is built on the CNN network used for the MNIST dataset, which again was built on knowledge from [2, 5].

The feature engineering done for this dataset is the same as was done for the MNIST dataset. Normalization is performed by dividing each feature in the images for the datasets by 250.0, which gives features in the range [0,1].

Since this project thesis serves as a learning period three different CNN models will be implemented and applied to the CIFAR10 dataset to evaluate the performance. All of these three CNN models are compiled with the same optimizer and loss function as the CNN for the MNIST dataset, that is, the Adam optimizer and the sparse categorical crossentropy loss function. Each of these CNNs is built using a Sequential model from the Tensorflow's Keras library.

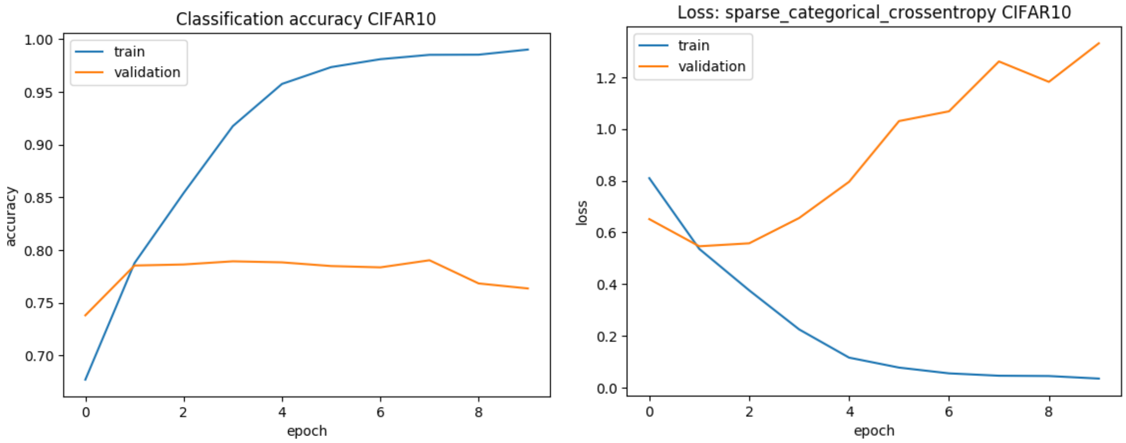
Model 1

Using the same CNN for the CIFAR10 dataset as for the MNIST dataset resulted in bad performance. To get better performance, two more convolutional layers were added to the CNN for the CIFAR10 dataset. All of these layers have 64 neurons, a kernel size of 3×3, the activation function ReLU and the he uniform kernel initializer. The first convolutional layer serves as input layer, while the other two serves as hidden layers. A flatten layer is implemented after the convolutional layers and before the dense layer. The flatten layer flattens the dimensions of the output from the convolutional layer to make it compatible for input data to the dense layer. The dense layer has 128 neurons and the ReLU activation function. After these hidden layers, a dense layer serves as the output layer. The output layer has 4 neurons, one for each of the classes to predict, and has the softmax activation function. This network is shown in Table 4.2.

The result from the CNN described Table 4.2 is shown in Figure 4.5.

Layer	Units	Kernel initializer	Kernel size	Activation function
Convolutional	64	He uniform	3×3	ReLU
Convolutional	64	He uniform	3×3	ReLU
Convolutional	64	He uniform	3×3	ReLU
Flatten	-	-	-	-
Dense	128	He uniform	-	ReLU
Dense	4	He uniform	-	Softmax

Table 4.2: The CNN architecture of model 1 for the CIFAR10 dataset



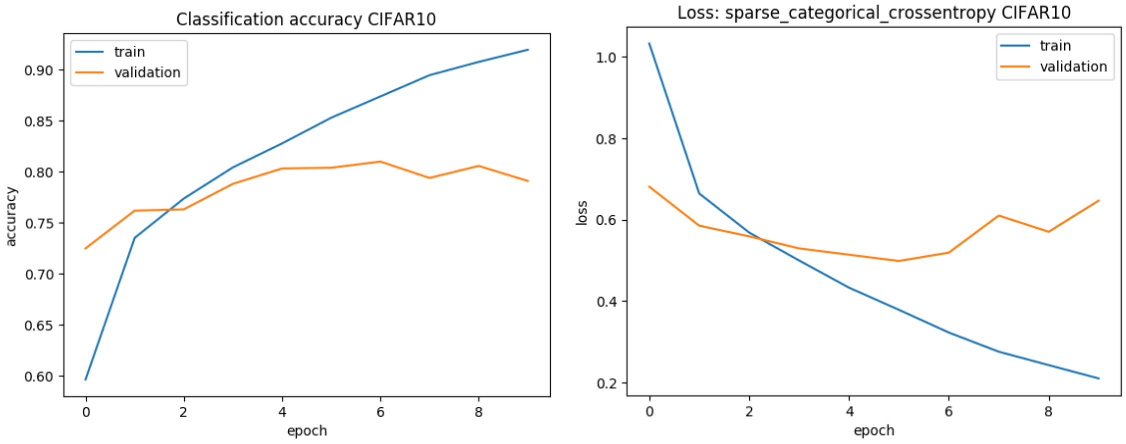
(a) Accuracy (b) Loss

Figure 4.5: Model 1: CNN implementation for MNIST dataset evaluation

These fiugres shows that the model is overfitting the training data, but the test accuracy is 0.9882. One reason for this could be that the quality of the images in the validation dataset overall is worse than it is for the test dataset, and that the validation dataset is a bit similar to the training dataset.

Model 2

To get rid of overfitting, regularization techniques can be used. The first technique tested is done by adding three dropout layers with a dropout of 0.4. With the dropout layers the accuracy gap between the training- and the test data got decreased, but the model is still overfitted, as can be seen in Figure 4.6. The test accuracy is 0.9517 for the second model. This could be another sign that the training data is more similar to the test data than the validation data, or the validation data images are of poor quality.



(a) Accuracy (b) Loss

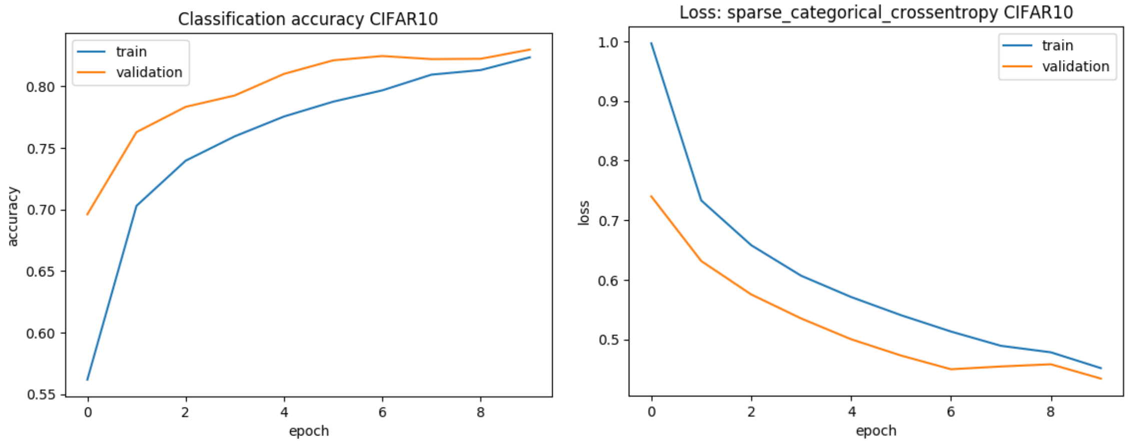
Figure 4.6: CNN implementation for MNIST dataset evaluation

Layer	Units	Kernel initializer	Kernel size	Activation function
Convolutional	64	He uniform	3×3	ReLU
Max pooling	2×2 dim.	-	-	-
Dropout	0.4 rate	-	-	-
Convolutional	64	He uniform	3×3	ReLU
Max pooling	2×2 dim.	-	-	-
Dropout	0.4 rate	-	-	-
Convolutional	64	He uniform	3×3	ReLU
Flatten	-	-	-	-
Dropout	0.4 rate	-	-	-
Dense	128	He uniform	-	ReLU
Dense	4	He uniform	-	Softmax

Table 4.3: The CNN architecture of model 3 for the CIFAR10 dataset

Model 3

The next regularization technique used to get rid of the overfitting is max pooling. Two max pooling layers with the 2×2 dimension was added to the CNN, and the resulting network is shown in the Table 4.3. The resulting performance from the network in Table 4.3, which is shown in Figure 4.7 is not overfitting the training data, but rather underfitting. The test accuracy is 0.8900 for this model.



(a) Accuracy (b) Loss

Figure 4.7: CNN implementation for MNIST dataset evaluation

The validation dataset is used after each epoch to validate the model. The validation loss and accuracy should indicate if the model is underfitting or overfitting the training data. For these three CNNs represented in this section it looks like the better the performance for the validation dataset is, the performance for the test dataset gets worse. From this it can be concluded that the validation dataset chosen is not the best for the training dataset, or the test dataset is too similar to the training dataset. The model can be tested on the different dataset to see if it is generalizing well. Cross-validation can also be used to see if there are better splits for the validation dataset.

Chapter 5

Generative Adversarial Networks

The theory and implementation in this section is mainly based on knowledge from [9, 4, 10].

In the proposed adversarial nets framework, the generative model is pitted against an adversary: a discriminative model that learns to determine whether a sample is from the model distribution or the data distribution [14].

5.1 Theory

The NNs and basic ML methods in the previous Sections 3 and 4, goes under the category of discriminative modeling. A discriminative model will be able to learn from labeled training data to predict the label of new data. For instance, in the CNN implementation for predicting the digits in the MNIST dataset the training data had images of digits between zero and nine and their corresponding label 4.2. When giving the trained CNN model new input it had never seen before it was able to predict the labels of the digits with quite a good accuracy. In section five 5, generative modeling will be introduced and utilized, together with discriminative modeling, to be able to generate new data from a set of training data.

Generative Models

The input for a generative model is usually unlabeled, but it can also be labeled. The output of the generative model will be a set of pixels, for instance, that have a high chance of being predicted to belong to one of the classes in the training dataset. With generative modeling it is possible to figure out how the data was generated, and not only make predictions on it as with discriminative models. The generative model is also probabilistic, which means that it does not produce the same output every time. This is accomplished by introducing a random element, which is done in the implementation below by generating a vector of random noise. A generative model aims is to learn the true data distribution of the training data, and then use this distribution to generate new data that looks like it comes from the same distribution.

The Naive Bayes model builds on the naive assumption that each feature is independent of every other feature [10]. This kind of model will estimate the probability of seeing each feature independently. The probability of the Naive Bayes model to generate an observation is calculated by multiplying the probability of the appearance of each feature itself. This model can work well as a generative model for features that are reasonable to expect to be independent. The Naive Bayes model does not work well on raw image data though. This is because the pixels, which will be the features, are not independent of each other. Creating a generative model for images involves DL for the model to find the features itself and learn the unknown probabilistic distribution function for the features in the training dataset.

Representational learning is used in a generative model to represent an observation, for instance an image, in a lower dimensional *latent* space. The generator finds a function to map a pixel point in the latent space to a point in the high-dimensional image [10]. The random noise vector fed into the generator in the implementation below is the representation of the image in the latent space. Using a latent space representation of the image instead of using raw pixels simplifies the problem and gives better performance. Humans also use a latent space representation when playing the Catch Phrase game for instance. This is a word guessing game where the player wants their partners to guess the word on the player's card without saying the word itself. For instance, if the word is "apple", the player can describe "batches" of pixels like its shape, color and so on and assume the partners have an idea of how these batches look like in pixels. The player's partners are then mapping the explanation in the latent space to pixels to generate an image of an apple. Latent space also comes handy for manipulations of an image, which we can call latent arithmetics [10]. For instance, it is possible to change the latent space vector of an image of a smiling person to make the decoded image be the same person but sad. This same technique can be used to morph between two faces. This way complex problems can be solved by dividing it into smaller and easier problems.

Deep Generative Models

Variational autoencoders (VAE) is a famous and fundamental generative DL model [10]. An autoencoder can be used to do the mapping from a high dimensional space to a low dimensional latent space and vice versa. An image can be encoded to a vector in the latent space, which again can be decoded to an image in the high dimensional space. When decoding back to the high dimensional space some information will be lost which means that it is hard to reconstruct the input image. An autoencoder network can be trained to find weights that will minimize this loss of information. Comparing the autoencoder to known models, one can see that the decoder is a bit similar to the generative model and the encoder is a bit similar to the discriminative model. The implementation is also quite the same. To create the autoencoder, a third model has to be implemented where the encoder model's output is taken as input in the decoder model. When this third model is created, it needs to be compiled with an optimizer and loss function, just as for the NN in section 4.2. The model will be fitted to a training dataset with corresponding labels. This model takes an image, passes it through the encoder and back through the decoder to reconstruct the image again. There are some drawbacks with the autoencoder which can be solved with the variational autoencoder. Each image is in the variational autoencoder mapped to a multi rate normal distribution around a point in the latent space, instead of just being mapped to a point in the latent space as was the case for the autoencoder [10]. This introduces a randomness to the model which results in a point in the latent space that has not been seen by the decoder before to still be decoded to a well-formed image. VAE works fine as a generative model when a low dimensional latent space, for instance of two dimensions, can be used. With higher dimensions of the latent space, the VAE will have trouble performing well.

VAE can be used to generate "fake" images of people based on a training dataset with a lot of images of "real" people. The images will have quite low quality and resolution, but it will still be possible to see that it is an image of a person. To generate images of people with a higher resolution, GANs can be utilized.

A GAN consists of two networks, one called the generative network and the other called the discriminative network. These two networks are adversarial, thus the name GANs. The networks are generative because they are probabilistic, which implies that the model does not produce the same output every time. The generative network generates samples from random noise, while the discriminative network takes both the actual data and the generated samples as input and labels it as "fake" or "real". The generative network wants to generate good "fake" data to "trick" the discriminative network to label it as "real", while the discriminative network does not want to be "tricked". This is how the networks are adversarial. If enough data are available the GAN can converge. The goal is to find an unknown probabilistic distribution that explains why some images are more likely to be found in the training dataset, and others a not [9].
An example of a GAN can be an iterative two-player minmax game with the value function $V(G,D)$, as shown in 5.1 [14]. The generator, G, minimizes its loss when its generated samples get a probability of one from the discriminator, D, that is $D(G(z_{latent})) = 1$. The discriminator, on the other hand, minimizes its loss when the probability is one for "real" data, $D(x) = 1$, and zero for the generated samples, $D(G(z_{latent})) = 0$. The main principle of GAN is the alternate training of the discriminator and generator and the aim is to reach convergence. The convergence requires enough capacity, computation time and data.

$$\min_{G}\max_{D} V(D,G) = \mathbb{E}_{x\sim p_{data}}[\log(D(x))] + \mathbb{E}_{z_{latent}\sim p_{z_{latent}}}[\log(1-G(z_{latent}))] \tag{5.1}$$

In the implementation GANs will be used for generating images GANs. The goal is to be able to generate new sets of pixels that look like they have been generated by the same set of rules as the pixel sets in the training data. The discriminator in the GAN is built with a convolutional architecture, in a similar way as the CNN is built in section 4.2. That is, starting with an image, getting the feature maps and down sampling the image [14]. For the CNN the output was either

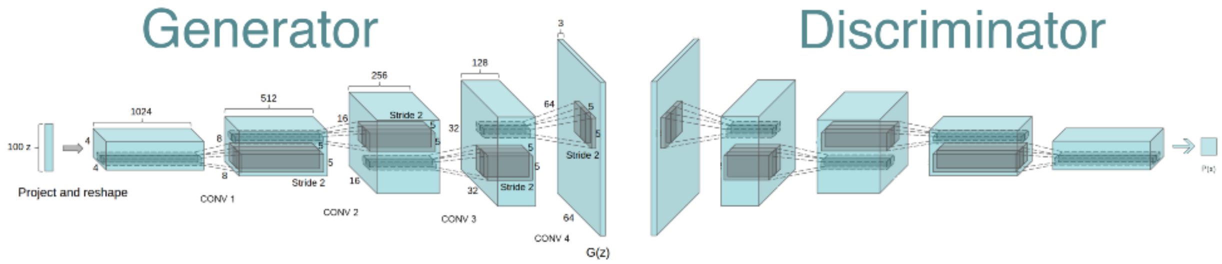


Figure 5.1: The layer architecture of the generator and descriminator of a GAN [10]

,

four or ten probabilities, but for the discriminator network there will always be just one probability in the output. This probability will be one if the image is a "real" sample, and zero if it is a generated sampled data.

The generative network is built on a transposed convolutional architecture, which is in a way opposite to the convolutional architecture. The generative network will not start with an image but with a latent vector and end up with an image. *Transposed* convolutional layers are used to increase the dimensionality from the dimensions of the latent vector to the dimensions of the image to be generated. The layers of the generator and discriminator is shown in Figure 5.1, which shows how the generator maps a vector to an image and the discriminator maps an image to a probability that the image is "real" or "fake" [10]. The generator generates the images only based on the vector, and not by using the training data. The dimensions in 5.1 is for an example of a GANs implementation done for the CIFAR10 dataset [10] and may differ a bit in the following implementation, but the concept is the same.

5.2 Implementation

GAN for the MNIST Dataset

This GAN implementation will be built on the CNN implementation described by 4.3, in section 4.2, where the MNIST dataset is described as well. The goal is to generate images of digits that will pass as digits from the MNIST dataset in the discriminator network. This will be implemented utilizing Tensorflow's Keras, as was done for the CNN.

Layer	Units	Input	Output	Kernel size	Activation	Strides
Convolutional	64	28×28	14×14	3×3	ReLU	2×2
Convolutional	128	14×14	7×7	3×3	ReLU	2×2
Dropout	0.5 rate	7×7	-	-	-	-
Flatten	-	7×7	-	-	-	-
Dense	1	-	-	-	Sigmoid	-

Table 5.1: The discriminator's architecture for the GAN implementation on the MNIST dataset

The feature engineering applied is scaling the pixels to a range of $[-1, 1]$ which is common practice when dealing with GANs for images. This is done by first dividing the pixels by 250.0 as was done for the CNN implementations in section 4.2. Secondly, the pixels are scaled by two and shifted by minus one.

The discriminator has the same principal as the CNN implemented in section 4.2. The code to create the discriminator will therefore be quite similar. One difference is that the last dense layer will not output ten probabilities anymore, but one. *Strides* will also be added to the convolutional layers, which will reduce the size of the width and height of the output tensor with respect to the input tensor. A stride of 2×2 will output a tensor with half the size of the input kernel. This is handy to decrease the dimensions. Padding will also be added to not lose any information. This means that if the stride is set to one and the padding to "same", the output size will be the same as the input size no matter the size of the kernel.

As can be seen in Table 5.1, which shows the discriminator's architecture, the stride reduces the width and height from input to output in the two convolutional layers. The first two layers also have 64 and 128 filters respectively. The output of a dropout layer is unknown, and is therefore not written in the table. The ReLU activation function and the dropout layers are taken from the CNN implementation 4.3. The output layer of the discriminator network is a dense layer with one node and the sigmoid activation function. The discriminator will output a single number between zero and one, thus the probability of ""real"" or "fake" image. The sigmoid function was introduced in the Logistic Regression section 3.4. If the discriminator outputs a number above 0.5, the input will be classified as "real", and if the output of the discriminator is below 0.5 it will be classified as "fake".

The generative network is also inspired by the CNN implemented 4.3, but the convolutional layers are replaced with transposed convolutional layers. The reason for this replacements is that

Layer	Units	Input	Output	Kernel size	Activation	Strides
Dense	12544	100×	12544	-	ReLU	-
Reshape	-	12544	$7 \times 7 \times 256$	-	-	-
Transposed Convolutional	128	$7 \times 7 \times 256$	$7 \times 7 \times 128$	3×3	ReLU	1×1
Transposed Convolutional	64	$7 \times 7 \times 128$	$14 \times 14 \times 64$	3×3	ReLU	2×2
Transposed Convolutional	1	$14 \times 14 \times 64$	$28 \times 28 \times 1$	3×3	tanh	-

Table 5.2: The generator's architecture for the GAN implementation on the MNIST dataset

the goal is to generate a $28 \times 28 \times 1$ dimensional image from a lower dimensional latent vector. The stride in a transposed convolutional layer will increase the dimensions, this is called *upsampling*. When the stride is 2×2 in a transposed convolutional layer the width and height is doubled, compared to the convolutional layer in the discriminator where the same stride made an output tensor with half the width and height as the input tensor. For the convolutional layer the gaps between pixels are filled with zeros, but in the transposed convolutional layer the existing pixel values are just repeated in the gaps [10].

As can be seen in Table 5.2, which shows the generator's architecture, the dimensions in the transposed convolutional layer is increasing with the stride dimension. The transposed convolutional layers also have padding set to "same" which means that no information is lost between layers. The output from the generator is a $28 \times 28 \times 1$ dimensional image. The generator takes a vector as input, which consists of 100 values per instance. This vector is generated by Numpy's *random.uniform* between minus one and one. This vector consists of the same amount of random noisy images as the batch size, which is 200. The activation function of the output layer is the *tanh* function which looks like the sigmoid function except it goes from minus one to one. This activation function is chosen because the pixels of the training data is scaled and shifted to be in the range $[-1, 1]$. The tanh function will map the range of the generator to the range of the image data, which is $[-1, 1]$ [10].

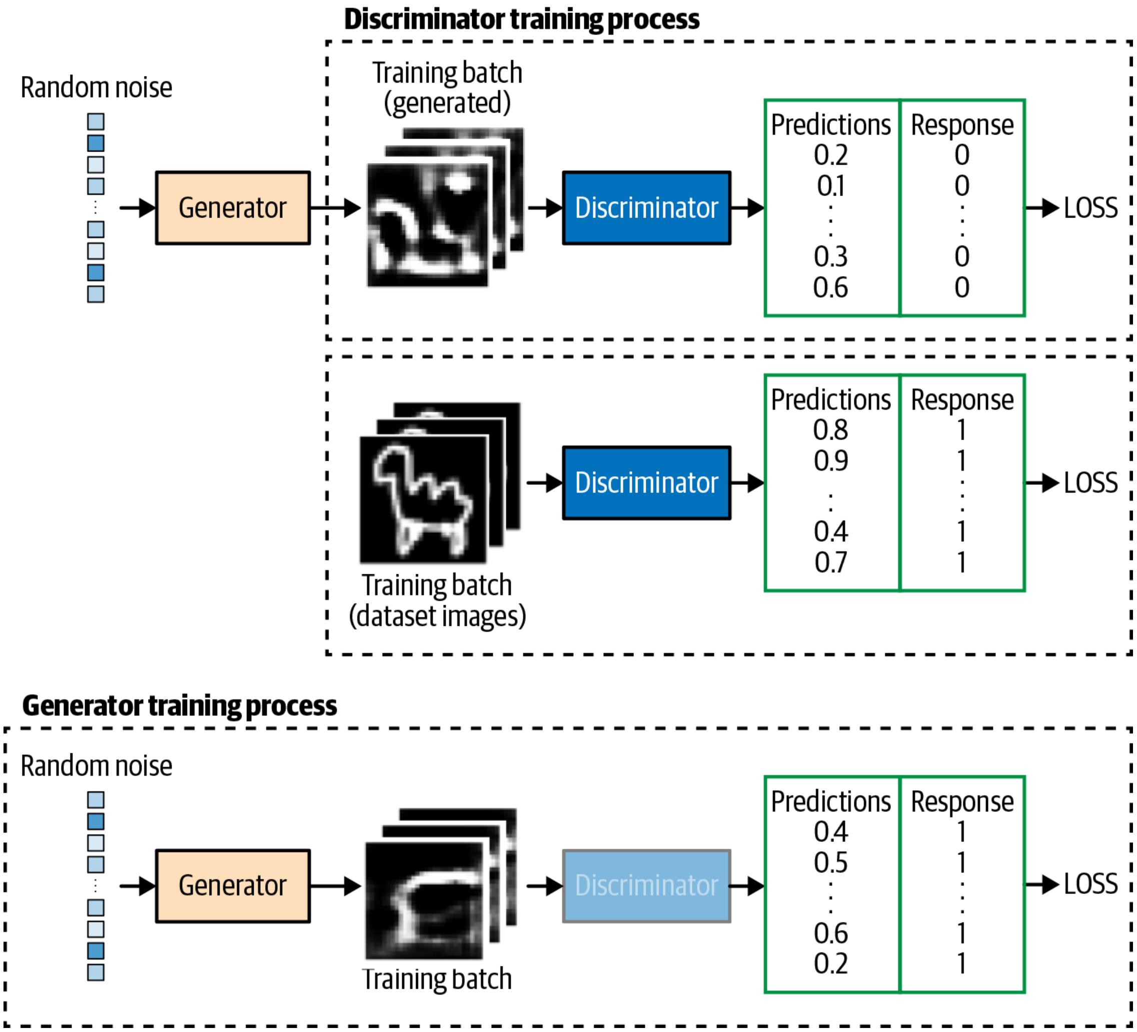


Figure 5.2: The training process of GAN illustrated [10]

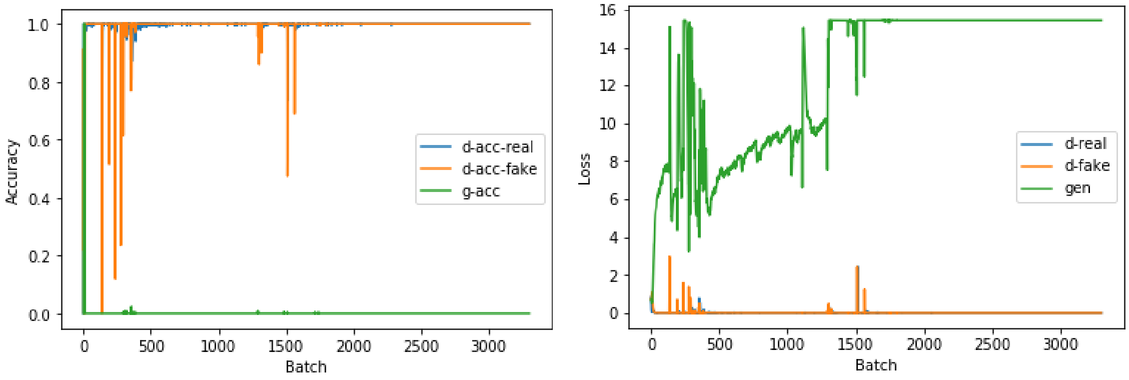
,

When the discriminator and generator are implemented, the training of the GAN can begin. Figure 5.2 illustrates the training process of a GAN on a dataset of animals. The training process for this implementation will be exactly the same except it will be done on the MNIST dataset.

Firstly the discriminator's training process will be explained. Random noise is generated and given as input for the generator network. The output from the generator based on the random noise vector given as input, is given as input to the discriminator. The output from the discriminator is a prediction of the label of this input. A label of zero indicates that the discriminator has predicted that the input comes from the generated training batch. The discriminator also gets inputs from the actual training batch, which will give a label of one if it is predicted as "real" data from this batch. Thus, in the training period for the discriminator, is given both generated images and "real" images from the batch. The discriminator uses a binary cross-entropy function to compare the predicted output from the generated samples with a tensor of zero. The predicted output from the discriminator when images from the training batch are the input is compared with a tensor of ones in the binary cross-entropy function, and a tensor of zeros if the images come from the generated batch. This means that the binary cross-entropy function is the loss function for the discriminator, and is the one to minimize. The Adam optimizer with default parameters is used when compiling the discriminator model [28].

For the generator's training process, a Keras model with both the generator and discriminator is needed. The weights of the discriminator is frozen so that they will not be updated during the generator's training process. The discriminator is added to the model, and then the generator is added. This model will take a random noise vector as input, and output a prediction. The generator aims to get this predicted output as close to one as possible. The loss function is a binary cross-entropy function comparing the predicted output from the discriminator, based on the generated training batch, with a tensor of ones. This loss function will be minimized since the generator wants its generated image to be predicted as a "real" image by the discriminator, that is, close to one. The Adam optimizer with default parameters is also used to compile this mixed model of the generator and discriminator. The weights of the discriminator in this mixed model will be updated during the discriminator's training process.

Figure 5.4a shows sixteen images of digits from the MNIST dataset. These are the ones fed into the discriminator, and should be predicted as "real". After running the GAN on the MNIST dataset for 8 epochs, Figure 5.4b shows some of the generated images. These generated images are not even looking like digits. Figure 5.3a shows that the accuracy for the generator stays at zero after approximately the first 300 batches. The discriminator's accuracy converges to one after approximately 1500 batches. Running the networks over more epochs could result in better generated images, but it will be time-consuming. BN layers could be added to the convolutional layers to speed up training and improve the performance. Using different minibatches to compute the normalization statistics on each training step results in fluctuation in the normalizing constants [14]. When these batches are small the fluctuations can become large, which could result in the fluctuations affecting the generated image more than the actual latent vector. A solution to this fluctuation problem is to utilize a *reference* BN. This solution consists of running the network once on the reference minibatch, and once on the actual minibatch to train on [14]. Mean and standard deviation is then computed for every feature utilizing this reference minibatch, and then the features in both batches are normalized using these normalization statistics. This can be a great solution to the fluctuation problem unless the model overfits the reference batch. Another BN called *virtual* BN is avoiding this overfitting problem by computing the mean and standard deviation by a union between the reference- and actual minibatch [14]. The minibatch's examples are processed independently for both of the new techniques introduced above. The accuracy for the generator stays at zero, and the accuracy for the discriminator stays at one after running the implementation over more epochs.



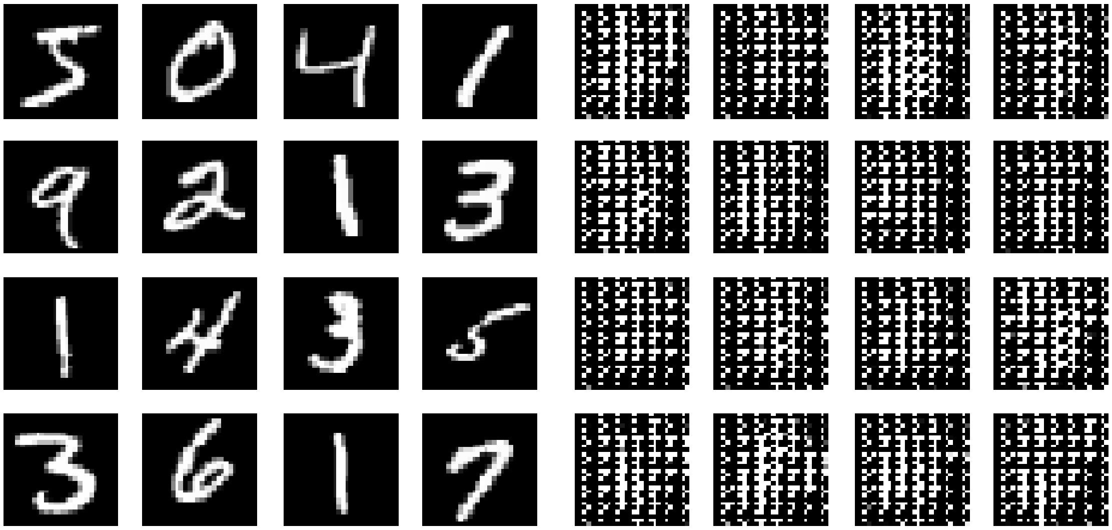
(a) Accuracy over 8 epochs

(b) Loss over 8 epochs

Figure 5.3: Evaluation of the GAN implementation without tuning over batches

Another problem when training the GAN can be inconsistent fluctuations in the loss of the discriminator and generator. It is desired to have a loss that stabilizes or gradually increases or decreases in the long term [9]. This is desired to make sure that the GAN will converge over time. As can be seen from Figure 5.3b the generator's and discriminator's loss has converged, but not to the desired value.

Modal collapse could also occur during the training of the GAN. This phenomenon could happen if the generator has been trained over several batches without updating the discriminator, and it finds a set of samples that always "fools" the discriminator [9]. All the points in the latent space will be mapped to this set of samples, thus the gradient of the loss function will collapse to near zero. Even if the discriminator is being trained and updated, the generator will have the upper hand and the GAN will get stuck. Figure 5.5 shows a result of the modal collapse [9].



(a) Images of digits from MNIST dataset (b) Generated images over 8 epochs

Figure 5.4: "real" and generated MNIST data digits without tuning

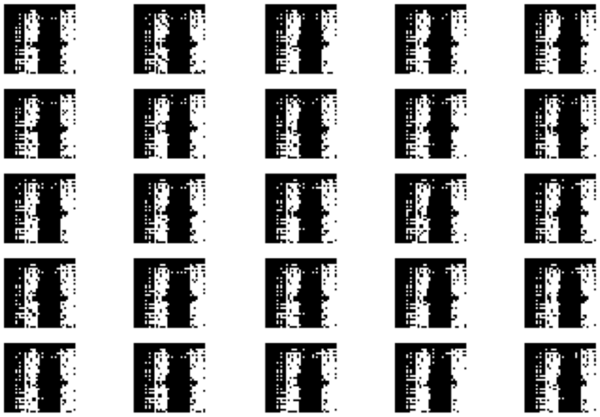


Figure 5.5: Example of modal collapse [9]

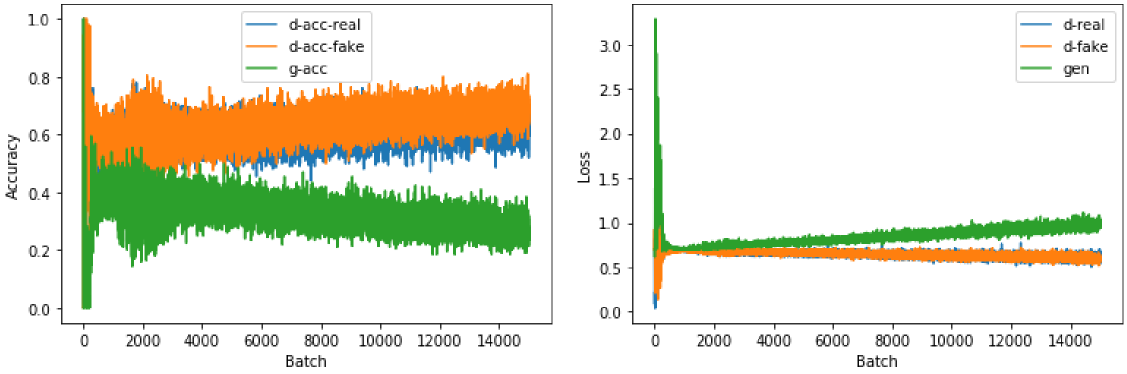
,

Modal collapse is not the issue for this implenetation since the generator's accuracy is zero, 5.3a, thus the discriminator has not been "fooled".

The overall performance of the GAN can be a bit difficult to evaluate during the training process because of the uninformative generator loss [9]. The loss of the generator can increase even though the quality of the generated images are improving and vice versa. During the training of the implementation in section 5 the performance over each epoch is evaluated by displaying the generated images as shown in Figure 5.4 and the accuracy for both the discriminator and generator as shown in Figure 5.3a.

The generated images could also be improved by changing the networks in the GAN. Hyperparameters like dropout, stride, learning rate, kernel size, BN parameters, latent space size, activation layers, convolutional filters and batch size could be tuned. The architecture for the generator and discriminator could also be changed to improve the performance of the implemented GAN. The tuning can be time-consuming and requires great knowledge in the field of GAN.

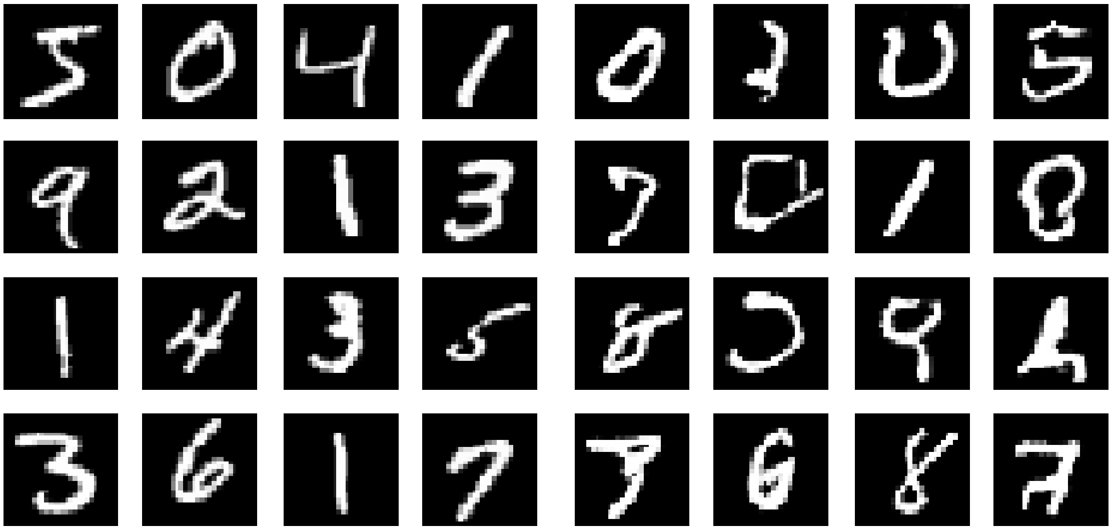
The optimizer used in this implementation is the Adam optimizer with the default hyperparameters, where the learning rate is 0.001, the momentum decay is 0.9 and the scaling decay is 0.999 [2]. In section 4.2 the default hyperparameter settings for the Adam optimizer worked fine, but for the GAN implementation they might need some tuning. Since the Adam optimizer is an adaptive learning rate algorithm, as mentioned in section 4.2, the learning rate requires less tuning and will therefore remain at the default value for now. The momentum decay parameter can be between zero and one. By keeping the implementation as it was for 5.3, and only decreasing the momentum decay parameter, the performance increases significantly! Figure 5.7 and 5.6 shows the resulting performance of the GAN implementation over 50 epochs by setting the momentum decay to 0.5. The accuracy of the discriminator is increasing while the accuracy of the generator is decreasing over time. This is an unwanted behavior which shows that more tuning is needed.



(a) Accuracy over 50 epochs

(b) Loss over 50 epochs

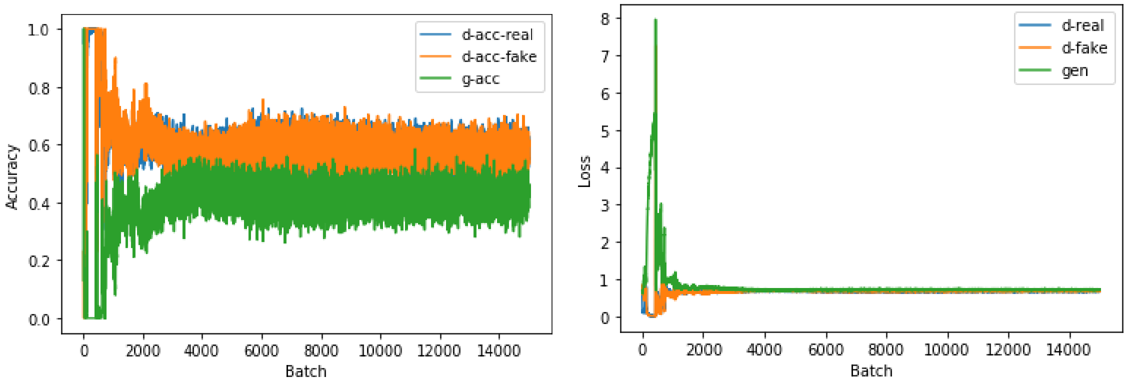
Figure 5.6: Evaluation of the GAN implementation with momentum of decay equal 0.5



(a) Images of digits from MNIST dataset (b) Generated images over 50 epochs

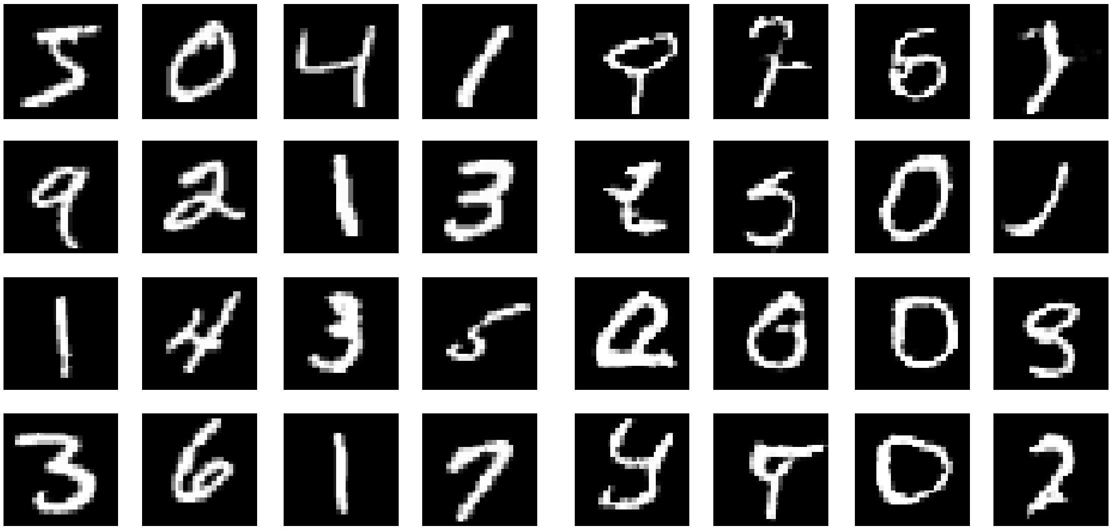
Figure 5.7: "real" and generated MNIST data digits with tuning

By setting the learning rate to 0.0002 and keeping the momentum decay at 0.5, inspired by [28], gave the results shown in Figure 5.8 and 5.9. Running this implementation over 50 epochs shows a good performance when looking at the accuracy for the generator and the discriminator 5.8a. Compared to the accuracy in Figure 5.6a, the accuracy for the generator and discriminator in Figure 5.8a appears more stable over 50 epochs.



(a) Accuracy over 50 epochs (b) Loss over 50 epochs

Figure 5.8: Evaluation of the GAN implementation with tuning over batches



(a) Images of digits from MNIST dataset (b) Generated images over 50 epochs

Figure 5.9: "real" and generated MNIST data digits with tuning

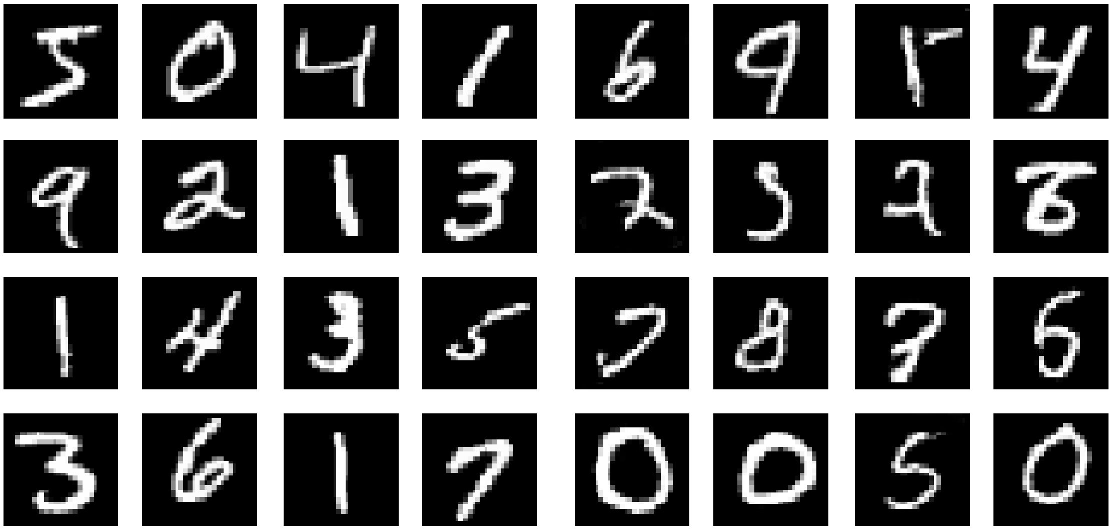
Implementing a GAN for the MNIST dataset is said to be relatively simple compared to implementations for other datasets. It is therefore assumed generated images with high quality when utilizing a "perfect" GAN implementation. Thus there are be room for improvement in the implementation discussed in section 5.

Executing the implementation over more than 50 epochs seemed to make the performance worse thus the quality of the generated images got slightly decreased. Trying to predict the generated digits with the CNN implementation in section 4.2 gave non-satisfactory results. The GAN implementation can therefore be changed according to the following papers to improve the results [29, 28, 30]. The ReLU function in the discriminator's layers will be changed to a *LeakyReLU* function with a slope of 0.2 [28]. This change made the predictions done by the CNN more accurate. After the success of adding the LeakyReLU to the discriminator, it is also added to the generator. The result from running this final GAN over 200 epochs can be shown in Figure 5.10 as well as its corresponding predictions which are shown in Table 5.3. When executing the implementation the predictions done by the CNN were displayed after each epoch, and it showed how it took a while to predict the different digits correctly. The digit in position $(4,1)$ in Figure 5.10b, which appears to represent zero, was not predicted to be a zero until more than 100 epochs were run. Before it was predicted to be a zero, it was predicted to be a num-

2	2	1	4
2	2	2	2
2	2	2	6
0	6	5	2

Table 5.3: Predictions done by the CNN implementation in section 4.2

ber two like the digit in position $(4,4)$ that appears to be zero in the Figure 5.10b. Running the implementation over 200 epochs have shown how the performance improves over time.



(a) Images of digits from MNIST dataset (b) Generated images over 200 epochs

Figure 5.10: "real" and generated MNIST data digits over 200 epochs

GAN for the CIFAR10 Dataset

The GAN implementation for the CIFAR10 dataset will be built on the GAN implementation for the MNIST digits dataset as well as the CNN implementation for the CIFAR10 dataset described in section 4.2. The CIFAR10 dataset is described in section 4.2, but for this GAN all of the ten classes will not be utilized. The goal is to generate synthetic images of airplanes that will pass as airplanes from the CIFAR10 dataset in the discriminator network. This will be implemented utilizing Tensorflow's Keras, as was done for the MNIST GAN and the CNN.

The feature engineering done is scaling the pixels to a range of $[-1,1]$ which is common practice when dealing with GANs for images. The procedure for this feature engineering technique

Layer	Units	Input	Output	Kernel size	Activation	Strides
Dense	4096	$100\times$	4096	-	LeakyReLU	-
Reshape	-	4096	$3 \times 3 \times 256$	-	-	-
Transposed Convolutional	128	$4 \times 4 \times 256$	$8 \times 8 \times 128$	3×3	LeakyReLU	1×1
Transposed Convolutional	128	$8 \times 8 \times 128$	$16 \times 16 \times 128$	3×3	LeakyReLU	2×2
Transposed Convolutional	128	$16 \times 16 \times 128$	$32 \times 32 \times 128$	3×3	LeakyReLU	2×2
Transposed Convolutional	3	$32 \times 32 \times 128$	$32 \times 32 \times 3$	3×3	tanh	-

Table 5.4: The generator in the GAN implementation for the airplanes from the CIFAR10 dataset

Layer	Units	Input	Output	Kernel size	Activation	Strides
Convolutional	64	32×32	32×32	3×3	LeakyReLU	-
Convolutional	128	32×32	16×16	3×3	LeakyReLU	2×2
Convolutional	128	16×16	8×8	3×3	LeakyReLU	2×2
Convolutional	128	8×8	4×4	3×3	LeakyReLU	2×2
Flatten	-	7×7	-	-	-	-
Dropout	0.5 rate	7×7	-	-	-	-
Dense	1	-	-	-	Sigmoid	-

Table 5.5: The discriminator in the GAN implementation for the airplanes from the CIFAR10 dataset

is the same as in the GANs implementation for the MNIST dataset. The architecture of both the discriminator and generator was changes to have three channels instead of one due to the colored images in the CIFAR10 dataset 4.2. The filters in the convolutional and transposed convolutional layers were also changed to fit the 32×32 sized images. The resulting architecture for the generator and discriminator can be shown in Table 5.4 and 5.5 respectively. The evaluation from running this implementation over 94 epochs is shown in Figure 5.13. It can be seen that the accuracy over 94 epochs is stable 5.13a, but could have been higher than oscillating around 0.2. The resulting generated images is shown in Figure 5.12, and the actual images of airplanes from the CIFAR10 dataset are shown in Figure 5.11. The figures shows that the actual images have a higher resolution, however, the generated images still have some airplane characteristics. Different techniques are tested to see if the resolution in the generated images can improve. The label smoothing technique [31] was tested to see if it could improve the performance. Its

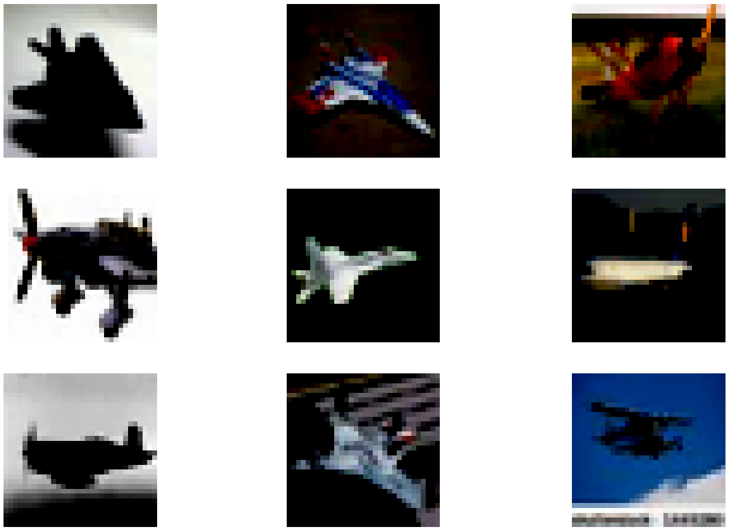


Figure 5.11: Actual images of airplanes from the CIFAR10 dataset

,

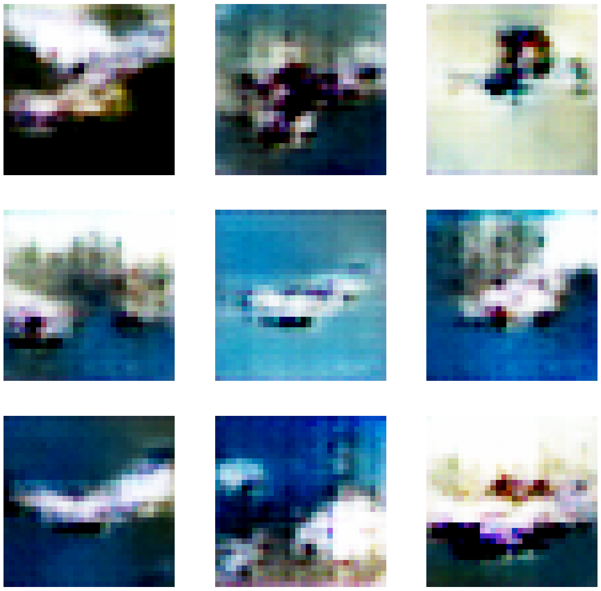
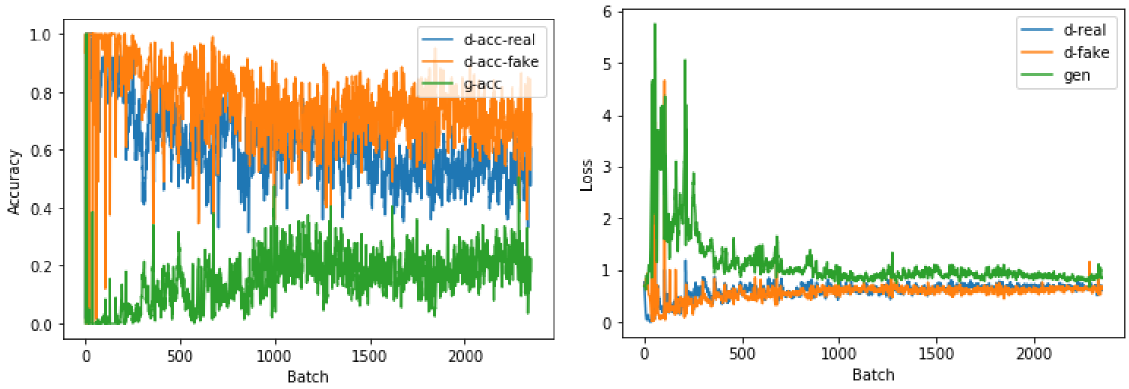


Figure 5.12: Generated images of airplanes over 94 epochs

,



(a) Accuracy over 94 epochs

(b) Loss over 94 epochs

Figure 5.13: Evaluation of the GAN implementation for airplanes in the CIFAR10 dataset

purpose is to prevent the model from making overly confident predictions on the training data [32]. The label smoothing is performed by using 0.1 instead of zero and 0.9 instead of one. Label smoothing resulted in the accuracy of both the discriminator and generator to start at zero and remain unchanged when running over 40 epochs, this modification is therefore removed. It was also tested to add batch normalization layers with a momentum of 0.9 after every convolutional layer in the discriminator and generator, which resulted in a modal collapse. Virtual batch normalization, described above, could have been tested but there were some problems implementing this. Weight normalization is said to be better than batch normalization for GANs, and could be used to get better performance [33]. Overall the resulting images in 5.12 and 5.10b is adequate for now, however, a better performance could be achieved by more tuning of hyperparameters, execution over a greater amount of epochs and more testing of different techniques [29, 28, 30].

Chapter 6

Summary and Conclusion

During this project thesis, a theoretical foundation and practical experience were obtained within the field of ML. In Section 3 basic ML methods like linear regression, logistic regression, SVM and decision trees were implemented. The results showed how SVM and decision trees worked better than logistic regression when the dataset includes a lot target variables and features. The linear regression method needed a linearly separable dataset, thus it could not be used for the whole dataset described in 3 unless manipulation was applied.

For predictions on more advanced datasets, like images, NN performs quite well if enough data is available. In Section 4.2 CNN was implemented and applied for both the MNIST digits- and CIFAR10 dataset described in Section 4.2. The results for the CNN implementation on the MNIST dataset were quite good, with a test accuracy of 0.986. The three different models implemented and applied on the CIFAR10 dataset shows how changes in the network's architecture and tuning of hyperparameters influences the overall performance of the model.

The knowledge and experience about CNN were utilized in Section 5 to implement a GAN applied on the MNIST digits dataset and a GAN for the airplane class in the CIFAR10 dataset to generate synthetic images. The implementations showed how tiny tweaks in the hyperparameters of the optimizer could boost the performance. Thus training a GAN is difficult and good knowledge in the field of GAN is needed to generate the desired data.

The results from this project thesis show the power of both the different techniques and the programmer utilizing them. This project thesis has been looking at some "basic" implementations due to the lack of knowledge beforehand. It can still be seen that GANs have great potential in many different fields. For future work in the master thesis, this technology will be utilized together with a simulator of a marine vessel for instance as well as actual data of the marine vessel. Images of the marine vessel will be extracted from the simulator and fed into the generator together with the "real" images of the marine vessel. The actual images are "expensive", and the aim is therefore to mainly use the images from the simulator together with some of the actual images to generate images that look like they came from the actual data distribution. This way it is possible to increase the amount of the data in the actual data distribution. This is one of the ways GANs can be utilized in the industry to save money and time, and as mentioned introduction wise, the field of GAN is yet to explore!

List of Figures

List of Tables

Appendix A

Table of Abbreviations

Abbreviation	Description
GAN	Generative Adversarial Network
NN	Neural Network
LSVRC	Large Scale Visual Recognition Challenge
MSE	Mean Square Error
ML	Machine Learning
DL	Deep Learning
CNN	Convolutional Neural Network
RL	Reinforcement Learning
KKT	Karush-Kuhn-Tucker
MLP	Multi Layer Perceptron
LTU	Linear Threshold Unit
ANN	Artificial Neural Network
BN	Batch Normalization
SGD	stochastic gradient descent
MNIST	Mixed National Institute of Standards and Technology
CIFAR10	Canadian Institute For Advanced Research 10
VAE	Variational Autoencoders

Appendix B

Table of Symbols

Symbol	Description
$\mathbf{x}$	Vector of input variables
$\mathbf{t}$	$\mathbf{x}$'s corresponding target values
$\phi(\mathbf{x})$	Basis function of input variables
$\mathbf{w}$	Vector of weights
$\mathbf{y}$	Predicted output based on $\mathbf{x}$
$E_D(\mathbf{w})$	Sum-of-squares error function
β	Noise precision parameter
$\sigma(.)$	Logistic sigmoid function
C_1	Classification $y = 1$
$p(\mathbf{t}\|\mathbf{w})$	The likelihood function
$\mathbf{H}$	The Hessian matrix
$\mathbf{R}$	$N \times N$ diagonal matrix with elements $y_n(1 - y_n)$
b	Bias parameter
η	Learning rate
τ	Current iteration of the algorithm
κ	Gain for $\mathbf{w}$
$\mathbf{a}$	The Lagrange multipliers

Symbol	Description
$\tilde{L}(\mathbf{a})$	Dual representation of the maximum margin problem
$\kappa(\mathbf{x},\mathbf{x}')$	Representation of the kernel function $\phi(\mathbf{x})^T\phi(\mathbf{x}')$
N_s	Total amount of support vectors
Q_τ	Gini index or cross-entropy error, with τ as leaf node
T	Total amount of leaf nodes
k	Class
R	Region
$p_{\tau k}$	Proportion of datapoints
$J(\boldsymbol{\theta})$	model dependent cost function
g	The activation function for a unit in NN mapping from $\mathbb{R}^m$ to $\mathbb{R}^n$
f	A function mapping from $\mathbb{R}^n$ to $\mathbb{R}$
z	Is $g(\mathbf{x})$ mapped by f
$s(i,j)$	Convolution between a kernel and an image, where i,j represents rows and columns respectively
I	An image
K	A kernel, also called a filter
G	Generator
D	Discriminator
$V(D,G)$	Value function of an iterative two-player minmax game
z_{latent}	Vector from in the latent space
p_{data}	Actual model variables
$p_{z_{latetnt}}$	Input noise variables

Bibliography

[1] Bishop CM. Pattern Recognition and Machine Learning (Information Science and Statistics). Springer-Verlag; 2006.

[2] Géron A. Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems. O'Reilly Media, Inc.; 2017.

[3] Marcopeix. marcopeix/ISL-linear-regression; 2018. Available from: `https://github.com/marcopeix/ISL-linear-regression`.

[4] Goodfellow I, Bengio Y, Courville A. Deep learning. MIT press Ltd; 2016.

[5] Abadi M, Agarwal A, Barham P, Brevdo E, Chen Z, Citro C, et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. CoRR. 2016;abs/1603.04467. Available from: `http://arxiv.org/abs/1603.04467`.

[6] Keras: The Python Deep Learning Library;. Available from: `https://keras.io/`.

[7] Python. Python Tutorials; 2016. Available from: `https://pythonspot.com/`.

[8] Brownlee J. Machine Learning Mastery;. Available from: `https://machinelearningmastery.com/`.

[9] Foster D. Generative Deep Learning: Teaching Machines to Paint, Write, Compose, and Play. O'Reilly Media, Inc.; 2019.

[10] Valle R. Hands-On Generative Adversarial Networks with Keras: Your guide to implementing next-generation generative adversarial networks. Packt Publishing Ltd.; 2019.

[11] Delude CM. Computing Intelligence. Brain Scan. 2011;Available from: https://mcgovern.mit.edu/wp-content/uploads/2019/01/brainscan{_}issue19.pdf.

[12] Krizhevsky A, Sutskever I, Hinton GE. ImageNet Classification with Deep Convolutional Neural Networks. In: Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States; 2012. p. 1106–1114. Available from: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.

[13] Rezvantalab A, Safigholi H, Karimijeshni S. Dermatologist Level Dermoscopy Skin Cancer Classification Using Different Deep Learning Convolutional Neural Networks Algorithms. CoRR. 2018;abs/1810.10348. Available from: http://arxiv.org/abs/1810.10348.

[14] Goodfellow IJ, Pouget-Abadie J, Mirza M, Xu B, Warde-Farley D, Ozair S, et al. Generative Adversarial Nets. In: Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada; 2014. p. 2672–2680. Available from: http://papers.nips.cc/paper/5423-generative-adversarial-nets.

[15] Shieber J. The makers of the virtual influencer, Lil Miquela, snag real money from Silicon Valley. TechCrunch; 2018. Available from: https://tcrn.ch/38Vj7UK.

[16] Vincent J. How three French students used borrowed code to put the first AI portrait in Christie's. The Verge; 2018. Available from: https://bit.ly/2RWYktE.

[17] Arjovsky M, Chintala S, Bottou L. Wasserstein GAN. CoRR. 2017;abs/1701.07875. Available from: http://arxiv.org/abs/1701.07875.

[18] Donahue C, McAuley JJ, Puckette MS. Adversarial Audio Synthesis. In: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019; 2019. Available from: https://openreview.net/forum?id=ByMVTsR5KQ.

[19] Durugkar IP, Gemp I, Mahadevan S. Generative Multi-Adversarial Networks. CoRR. 2016;abs/1611.01673. Available from: http://arxiv.org/abs/1611.01673.

[20] Ledig C, Theis L, Huszar F, Caballero J, Aitken AP, Tejani A, et al. Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network. CoRR. 2016;abs/1609.04802. Available from: http://arxiv.org/abs/1609.04802.

[21] Wu J, Zhang C, Xue T, Freeman B, Tenenbaum J. Learning a Probabilistic Latent Space of Object Shapes via 3D Generative-Adversarial Modeling. In: Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain; 2016. p. 82–90. Available from: https://bit.ly/38O7AGq.

[22] Zhang H, Xu T, Li H, Zhang S, Huang X, Wang X, et al. StackGAN: Text to Photo-realistic Image Synthesis with Stacked Generative Adversarial Networks. CoRR. 2016;abs/1612.03242. Available from: http://arxiv.org/abs/1612.03242.

[23] Mitchell TM. Machine Learning. McGraw-Hill, Inc.; 1997.

[24] Cauchy A. Méthode générale pour la résolution des systemes d'équations simultanées. Comp Rend Sci Paris. 1847;25. Available from: https://cs.uwaterloo.ca/~y328yu/classics/cauchy-en.pdf.

[25] Rosenblatt FF. The perceptron: a probabilistic model for information storage and organization in the brain. Psychological review. 1958;65 6. Available from: https://doi.org/10.1037/h0042519.

[26] Minsky M, Papert S. Perceptrons: An Introduction to Computational Geometry; 1969.

[27] Rumelhart DE, Hinton GE, Williams RJ. Learning representations by back-propagating errors. Nature. 1986;323. Available from: https://www.nature.com/articles/323533a0.

[28] Radford A, Metz L, Chintala S. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. In: 4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Con-

ference Track Proceedings; 2016. Available from: http://arxiv.org/abs/1511.06434.

[29] Arjovsky M, Bottou L. Towards Principled Methods for Training Generative Adversarial Networks. In: 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings; 2017. Available from: https://openreview.net/forum?id=Hk4_qw5xe.

[30] Salimans T, Goodfellow IJ, Zaremba W, Cheung V, Radford A, Chen X. Improved Techniques for Training GANs. In: Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain; 2016. p. 2226–2234. Available from: http://papers.nips.cc/paper/6125-improved-techniques-for-training-gans.

[31] Szegedy C, Vanhoucke V, Ioffe S, Shlens J, Wojna Z. Rethinking the Inception Architecture for Computer Vision. CoRR. 2015;abs/1512.00567. Available from: http://arxiv.org/abs/1512.00567.

[32] Hazan T, Papandreou G, Tarlow D. Perturbations, Optimization, and Statistics. MIT Press Ltd.; 2016.

[33] Xiang S, Li H. On The Effects of Batch and Weight Normalization in Generative Adversarial Networks. arXiv preprint arXiv:170403971. 2017;Available from: https://arxiv.org/abs/1704.03971.