Generating News Headlines Using a Generative Adversarial Network

Christa Höglinger

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im Juni 2019

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, June 25, 2019

Christa Höglinger

Contents

List of Figures

List of Tables

Acknowledgement

I would first like to thank my advisor Dr. Andreas Stöckl for always supporting me during the whole process of writing my Thesis and further improving the project. In addition, I want to thank for helping me to make the best decisions for my Master Thesis and finalize the research topic and also always taking time to read my Thesis several times. His door to office was always open whenever I had questions about my research or writing. I would also like to acknowledge David Hewlett as second reader of my Master Thesis, and I am gratefully for his very valuable comments on this Thesis. I am also very thankful to my college Christina Grafeneder, who always supported me whenever I needed any help throughout all the years of study. She always helped me to make the best decisions and putting all pieces together. Finally, I must express my very profound gratitude to my lovely family and friends, but especially to my parents, my sister Anna and to my boyfriend Jonas for providing me with unfailing support and continuous encouragement throughout all my years of study and through the entire process of working on my Thesis. This accomplishment would not have been possible without them. Thank you, I am so grateful to have all of them in my life.

Abstract

The technology *Generative Adversarial Network* is a part of the field of deep generative modelling and is often treated as such during research. This technology has shown impressive results in Computer Vision but has yet to be broadly applied to *Natural Language Generation*. This is the reason why this Master Thesis seeks to deal with the generation of text sequences and their evaluation and analysis. The objective of the Thesis is to generate high quality sentences in the form of news headlines and to make them indistinguishable from the real news headlines provided in the training dataset.

In the beginning, the basic concepts of related topics; *Artificial Intelligence*, *Machine Learning* and *Natural Language Processing* and the examples provided here, were introduced to provide a general overview. The main body of this Master Thesis includes the implementation of a Generative Adversarial Network model for Natural Language Generation, with special regard being given to news headlines (i.e., short text sequences). The Thesis concludes with a résumé of the results of the generation that are evaluated through metrics and the challenges encountered during the implementation of this Master Project are discussed here. The results of the headline generation are evaluated with using specific metrics, which are commonly used for the evaluation of Artificial Neural Network predictions. To seek for further improvement in this specific implementation, possible future works are mentioned to provide a direction for further optimizations.

Kurzfassung

Die Technologie *Generative Adversarial Networks* ist Teil des großen Themengebiets *Deep Generative Modelling* und wird auch oft in aktuellen Forschungen und Veröffentlichungen behandelt und erwähnt. Diese Technologie zeigte in der Vergangenheit sehr beeindruckende Resultate in der Bildverarbeitung, wurde jedoch bisher nur wenig in der Textverarbeitung und -generierung eingesetzt. Somit wurde dieses Problem der Textgenerierung zum Thema dieser Masterarbeit, welches zum Ziel hat, neue Texte beziehungsweise Sätze in Form von englischsprachigen Schlagzeilen in einer hohen Qualität zu generieren. Das bedeutet, dass diese generierten Schlagzeilen nicht mehr von echten Schlagzeilen aus dem Trainingsdatensatz unterschieden werden können.

Zu Beginn der Arbeit werden die Grundkonzepte verwandter Themen, wie Künstliche Intelligenz, Maschinelles Lernen und Textverarbeitung im Allgemeinen vorgestellt, um einen Überblick über diese Themen zu geben. Der Hauptteil der Masterarbeit beinhaltet die Implementierung eines Generative Adversarial Network Modells für Textgenerierung, insbesondere in Bezug auf Schlagzeilen, bei welchen es sich speziell um kurze Textsequenzen handelt. Schlussendlich werden Ergebnisse dieser Textgenerierung präsentiert und evaluiert, sowie die Herausforderungen, die während des gesamten Masterprojekts aufgetaucht sind, diskutiert. Für die Evaluierung der Ergebnisse wurden spezielle Metriken verwendet, die üblicherweise für die Evaluierung von Vorhersagen künstlicher neuronaler Netze verwendet werden. Zur weiteren Verbesserung der Implementierung des Anwendungsfalls werden mögliche zukünftige Methoden genannt, um eine Richtung für weitere Optimierungen zu geben.

Chapter 1

Introduction

In recent years, the use of machine-generated texts and the increase in the number of fake news articles especially in specific areas, such as reports concerning the stock market, elections or the economy, has multiplied considerably. For instance, an example of generated text could be scientific reports, (generated) code snippets or plays inspired by Shakespeare [37].

1.1 Motivation

There are multiple different models based on Deep Learning and neural networks, which can be used to generate complex text data. A recently developed and extremely effective model is the *Generative Adversarial Network (GAN)*, which was invented by Goodfellow et al. [13] in 2014. This model has primarily been used to create photorealistic images for the visualisation of various real objects; however, this model is rarely used for Natural Language Generation (i.e., text generation). Therefore, this Master Thesis chooses to deal with the generation of short text sequences in the form of high quality news headlines.

1.2 Problem Description

Since this technology has great potential, there are many possibilities for generating high-quality data. The basic idea behind GANs is to train two models simultaneously: the generative model produces new samples based on a training dataset and the discriminative model classifies the samples into two different classes (real or fake). The difficulties in using this technology for the purpose of text generation are that traditional GAN models can only work effectively with continuous data but are not intended for sequence generation of discrete tokens, such as text data based on those discrete tokens. The intention of this Master Thesis is to test and answer the following question:

1. Is it possible to generate (short) sequences, i.e., news headlines with a GAN model with comparable quality to sequences provided by real training data?

1.3 Goal of the Thesis

The main goal of this Thesis is to provide a GAN implementation with the aim of generating artificial news headlines that appear real and natural (i.e., written by a human). The generator has to be able to produce new, high-quality headlines based on a dataset, which are then classified by the discriminator. The objective is to produce legible headlines, which are readable, regardless of whether the content is true or false.

The research question presented at the end of the previous section should be answered by providing a measurement of the quality of the generated sequences with special metrics for the evaluation of Artificial Neural Network predictions. A comparison between the quality of the fake headlines and the real headlines from the training dataset is possible within the evaluation and analysis of the resulting outputs and performances. The selection of the right GAN model is essential before beginning the implementation to ensure it has the necessary potential of producing high-quality sequences. For this reason, a GAN model especially developed to solve Machine Learning task for Natural Language Generation, called SeqGAN introduced by Yu et al. [32], was used for the implementation.

To achieve the objective of this Master Thesis, several different areas were prepared and examined to ensure an understanding of the most important related topics. These parts are briefly described in the following section, which explains the structure of the Thesis.

1.4 Thesis Structure

The implementation and solution for the problem defined and treated in the Thesis requires wide-ranging knowledge about the basics of *Artificial Intelligence*, which are covered in Chapter 2. As well as providing an introduction to basic concepts of *Machine Learning* and mathematical and statistical algorithms, it also provides information concerning *Artificial Neural Networks* and the models required for the implementation.

Since the subject of this Thesis is text generation, it is pivotal to provide an overview of the field of *Natural Language Processing* and its subfield *Natural Language Generation* to help understand the theoretical and practical concepts that are applied. These topics are described in more detail in Chapter 3, where several inputs are illustrated to make comprehending the decision making process easier.

Up until recently, a lot of experiments and research had been made in the field of generating natural language (i.e., human language). In Chapter 4, a selection of related concepts and research is presented. This work consists of different deep generative models, which are related to the problem described within this Thesis. Basic concepts, algorithms, and specific GAN models designed for text generation are presented with the aim of providing a comprehensive view of the technology available for generating natural language.

A major part of implementing a Machine Learning model is to prepare the data right to be able to use it during training. The quality of GAN model predictions is depending on the quality of the dataset and its preparation. This is presented in Chapter 6, where the dataset on itself, its cleaning and pre-processing and also the handling of the data is mentioned.

As mentioned previously, the implementation was done with a GAN model called SeqGAN, which was specifically developed for Natural Language Generation tasks. First, the model was implemented based on the concept of SeqGAN with modifications made according to the particular use case required for generating short sequences in the form of news headlines. As a result of these modifications, a major part of the work in this Thesis Project was the adjustment of the hyperparameters used in the SeqGAN implementation. The process of hyperparameter tuning was essential and very important because the quality of the output strongly depends on the hyperparameter settings. In Chapter 5, the architecture of SeqGAN is introduced to provide a basic understanding of how this works so that the implementation of the GAN model in Chapter 7 can be traced.

Then the results were evaluated with specific metrics, which were calculated while training the model so as to be able to provide the answers required for the research question. The resulting output obtained during training were visualized and presented for analyzation in Chapter 8, along with the metrics, which are shown in the form of graph curves and are described and analyzed.

Finally, in Chapter 9, a conclusion consisting of a compact overview of the results is provided along with a presentation of the challenges encountered and possible further steps for the Thesis. This chapter provides the answer to the research question posed in Section 1.2 and summarizes the results obtained.

Chapter 2

Artificial Intelligence Basics

The aim of Artificial Intelligence (AI) is not only to understand, how people make decisions, but also to build intelligent systems, which attempt to reproduce certain decision-making patterns of humans. The first experiments and research in the field of AI began in 1956 and it is therefore a quite new field in science and engineering. AI can be applied in many areas and is divided into several subfields. These subfields cover a range of general tasks, such as learning and perception designed to suit tasks that are more specific; for example playing chess, writing a story or diagnosing disease [28, p. 1].

As early as 1950, the mathematician Alan Turing presented a concept, which remains the basis for a today's technique for testing the intelligent behaviour of a computer by providing a satisfactory operational definition of intelligence. This is called the *Turing Test* and it is based on an interpretation of Alan Turing's *Imitation Game* [30, pp. 433–434], which is passed if it cannot be distinguished whether the response to a question comes from a human or from a computer. The test setting consists of three people, a man A, a woman B and an interrogator C who can be either a man or a woman. $C's$ objective is to find out, who of the two people is the man or woman by asking A and B questions, which are designed to help deduce this; such as the length of their hair or the body size. In the end, C knows them by labels X and Y and should either say "X is A and Y is B" or "X is B and Y is A". If the interrogator C is then replaced by a computer and the test is performed using this scenario, it is called Turing Test. To be able to apply the Turing Test successfully, a computer requires certain capabilities. Firstly, *Natural Language Processing* is needed to enable successful communication in English. Additionally, *Knowledge Representation* is important as this enables it to store information regarding the knowledge already acquired and *Automated Reasoning* allows the system to draw new conclusions or answer questions by using previously stored information. Finally, *Machine Learning* is used to detect and extrapolate patterns and adapt them to new circumstances [28, p. 2].

This chapter covers the basic background of AI and related topics. In Section 2.1 the basics of Machine Learning are presented and then also the different types of Machine Learning are discussed. *Artificial Neural Networks* are examined in Section 2.2, firstly by giving an overview and a general description and then by explaining *Convolutional Neural Networks* and *Recurrent Neural Networks* in more detail because these are both used for the implementation of this Thesis work.

2.1 Machine Learning

Machine Learning (ML) is a discipline within the domain of AI, which seeks to find the answer to the question if a computer is capable of learning how to perform specific tasks without assistance. ML has been used since the 1990s and thanks to the availability of powerful hardware and large and complex datasets it has become the most popular and successful subfield of AI. In the traditional approach to programming, software is created with fixed input rules, with which the software processes data. The drawback with this form of programming is the enormous number of parameters and the massive data arrays it requires. ML software learns to perform these rules by processing the data along with expected outputs or answers. These rules can then be applied to new datasets to train the software to learn new rules. An ML application is able to act without being explicitly programmed or relying on predefined rules [6, pp. 23–24].

What the term *learning* is meant to describe is the knowledge gained from the data by using algorithms to acquire structural descriptions from the training data presented to the ML application. The system learns and gains new knowledge with which to identify the structures present within the information contained in the raw data [26, p. 2].

ML can be divided into different types of learning; the most common forms will be presented and explained in more detail in the following sections. Beginning with the field of *Supervised Learning* in Section 2.1.1, followed by *Unsupervised Learning* in Section 2.1.2 and finishing with *Reinforcement Learning* in Section 2.1.3. Each type is described in this Thesis because the implementation makes use of each of these types. However, there are no clear distinctions, which can be drawn between the three types, often the differences are blurred and they can also be used for a variety of tasks; this is especially true in the case of Supervised Learning and Unsupervised Learning.

2.1.1 Supervised Learning

The simplest and most popular type of ML is Supervised Learning, which is task-driven and provides data designed to assist in predicting the next set of values. It is very easy to understand and to implement in an ML application and can be considered as supervising or guiding by a teacher [3, p. 10]. The dataset provided acts as the teacher and the role of the dataset is to train the ML model by feeding entries into the model. This is the reason it is called training data and it consists of a pair of input and target data, in which the output should be predicted from the model. After predicting the target data, the ML model receives feedback if the output was correct by comparing it with the real target data [21, p. 6].

In order to be able to classify the input data correctly each example within the dataset is associated with a target or label. In Supervised Learning applications, the examples are observed with the assistance of a random input vector x and an associated target value y, which represents the label that should be predicted. Therefore, Supervised Learning methods generally try to construct a model by estimating a conditional probability $p(y \mid x)$ from the given training data. The term *Supervised Learning* originates from the point of view of the target label value y, which shows the model or ML application what to do and is provided by the teacher or instructor [12, pp. 105–106].

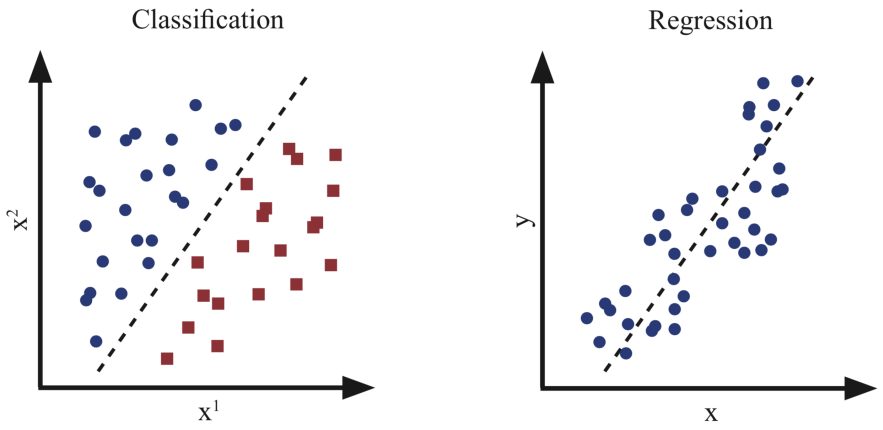


Figure 2.1: The left part of the figure shows a classification problem with 2 different classes, which are represented as blue dots and red squares. The data points are categorized in a particular observation. On the right side of the figure, a regression problem is shown. The blue dots represent the predicted continuous values based on a given dataset.

The training dataset with input-output pairs can be denoted as $(x_1, y_1), \ldots, (x_n, y_n)$ and consists of n entries. For the prediction a hypothesis h has to be generated to estimate the true function f, which is generated at the beginning by the function $y = f(x)$. The aim is to search for the best performing hypothesis through out the space of possible hypotheses $\mathcal{H}$ of both training data and new data not seen. The *generalization of a hypothesis* is very important to optimize the predictions of the Supervised Learning model. If the hypothesis generalization performs well, this means the label y was correctly predicted on a novel data example. Choosing the best hypothesis h^* considering the most probable data given can be calculated as [28, pp. 695–697]

$$h^* = \arg\max_{h \in \mathcal{H}} p\left(h \mid data\right). \tag{2.1}$$

The best performing hypothesis h^* can be chosen by applying the arg max function, where h has to be an element of all possible hypotheses $\mathcal{H}$. The probability of a hypothesis h is calculated based on the dataset given and then the highest probability is chosen for h^*.

Typically, the major segments within Supervised Learning are called *Classification* and *Regression* and can be described as following; classification is about predicting a label and regression is about predicting a specific value. The differences between classification and regression problems, which are shown in Figure 2.1, are described below.

Classification

A classification task is about predicting a class or category for the given input data and is shown on the left part of Figure 2.1. The predicted output label y is part of a finite set of discrete values, which means the variables are categorical and every data example belongs to precisely one class of the set. If the set of classes consists of only

two values, the classification problem can be called *binary classification* and is used for predicting the truth; i.e., if an email is spam or not spam. Another classification problem is called *multi-class classification* and consists of more than 2 class values. For instance if weather attributes are predicted, the classes could be *sunny*, *cloudy* or *rainy* [28, p. 696; 21, p. 8].

Regression

Regression tasks attempt to estimate the output label y for a specific input variable x. The difference compared with a classification problem is the type of output variable, because an output value of regression is numerical, while the output value of classification is categorical. A regression is a curve described by a mathematical function that passes as close as possible to the data points given, as can be seen in the graph on the right of Figure 2.1. Examples of use cases for a regression problem would be predicting house prices or predicting the temperature [28, p. 696; 21, p. 7].

2.1.2 Unsupervised Learning

The concept behind Supervised Learning problems is very different from that in Unsupervised Learning, because there are no labels available in the dataset. It is no longer possible to identify whether or not a prediction is correct or incorrect, because the training dataset contains no target data or form of scoring system. An Unsupervised Learning model learns through observations of the training data and is able to find specific structures within the data provided. These models try to understand the properties of the data and find patterns and relationships in the dataset even though no explicit feedback can be supplied [21, p. 281]. The aim of Unsupervised Learning is to discover which structures and patterns occur in the data more frequently than others do. This statistical approach is called *density estimation*, in which an unobservable underlying probability density function of observed data is estimated. After performing the density estimation, it is possible to solve the missing value imputation task by using the newly obtained data distribution $p(x)$ [12, p. 103].

There are multiple use cases for Unsupervised Learning problems, because most of the data available is unstructured and without labels. In this section, two examples are described in more detail to give a clearer idea of the possibilities presented by Unsupervised Learning. The concepts of *Clustering* and *Dimensionality Reduction* will now be described and visualized.

Clustering

One method of density estimation is clustering the data by grouping the input data. Data without any labels cannot be classified in a traditionally supervised way, because the correct classes cannot be identified. Instead, in order to cluster the data points that are similar to each other the identification of similarities between inputs must be found. Basically, this can be likened to a collection of objects based on the similarities and differences between them [21, p. 281]. As can be seen in Figure 2.2, a large amount of data points is placed in the diagram; these can be divided into a number of groups by using the concept of clustering. The data points in the same group are more similar

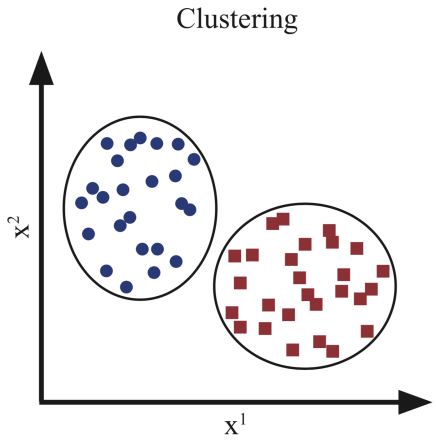


Figure 2.2: The data points in the graph can be clustered together and are then assigned to the same group. A data point could belong to any cluster, but the probability of the data point defines the membership of the cluster [3, p. 13].

to each other than to data points in other groups or clusters. When compared with a supervised classification problem, as shown in Figure 2.1, where the data points are classified based on the classes provided, clustering attempts to cluster the data without any labels available.

This technique of Unsupervised Learning can be used for several purposes, such as customer segmentation. This is commonly used in marketing analytics to identify different groups of customers with similar characteristics and then apply different marketing strategies aimed at them. Another example is in biology, where groups of genes with similar expression patterns are identified by clustering to obtain biological insights from DNA [8, p. 305].

Dimensionality Reduction

This technique reduces the number of random variables already considered by obtaining a set of main variables. In a dataset with a large amount of features, the probabilities of hidden relations can be very high, so visualizing and working with the data becomes increasingly difficult. The aim of dimensionality reduction is to find a low-dimensional representation of the data given while still retaining as much information as possible. The observed data includes individual measurable properties, called *features*, which are extracted from the input data observed. Often, these features are correlated or redundant and thus possess no additional values for the prediction [21, p. 129; 3, p. 323]. There are several reasons why dimensionality reduction is interesting and useful as a separate pre-processing step [1, pp. 109–110]:

- By reducing the number of input dimensions, the complexity of the ML algorithm can also be reduced thereby making it possible to reduce computation power and memory required.
- Reducing the dimensions without losing any information makes it easier to under-

stand and analyze the structure. Sometimes it is also possible to plot and visualize the structure and outliers.
- A better understanding of the process is possible when the data can be explained with fewer features, this also allows a better extraction of knowledge.
- A simplified model enables the use of smaller and simpler datasets during training, because the features depend less on details such as existing samples, noise, or outliers.

Dimensionality reduction can be divided into two main methods: *feature selection* and *feature extraction*. The aim of feature selection is the process whereby attributes that appear to be irrelevant are discarded. This will result in a smaller subset of the data with which to train the model which is nonetheless still based on the original set of variables (i.e., features). The dimensions that provide the most information are retained in the subset and the other values are discarded. In feature extraction, a new set of dimensions as combinations of the original dimensions is discovered and used to build an entirely new set of variables or features to represent the data, while still describing the original dataset [1, p. 110].

2.1.3 Reinforcement Learning

In reinforcement learning, quantitative information is provided by the environment to give feedback to the agent regarding the success or failure of a certain action in a state. This feedback is called *reward* and can be positive or negative and is summed up for the policy of the actions. Policy defines the strategy and best decision of actions the agent has to learn to be able to reach the highest expected immediate and cumulative reward. Therefore, the best policy maximizes the expected total reward by trying to capture the best knowledge from the past experience to make the best decision based on the feedback reward received. Once the reinforcement model has been trained over many iterations, it is able to accurately predict new data and produce the correct output [3, p. 14]. This method of ML has its roots in behavioural psychology and behaviour, which has been studied by animal psychologists for over 60 years. Animals receive a positive reward in form of pleasure or food and a negative reward in the form of pain and hunger [28, p. 830].

An essential part of reinforcement learning is the search over the state space for possible inputs and outputs. The algorithm searches over this space in order to maximize the reward by interacting between the agent and its environment. The components and their interaction within the reinforcement learning cycle are visualized and described in Figure 2.3. In this diagram can be seen how the interaction between the agent and its environment works: the decision-making agent completes the learning stage by making possible moves (actions) and the environment, in which the agent acts, produces the current input (state) and the rewards [21, pp. 231–232]. As already mentioned, the aim of the reinforcement learning approach is to maximize the total reward R from any time step t, which can be denoted as [15, p. 266]

$$R_t = \sum_{i=t}^{n} r_i = r_t + r_{t+1} + \cdots + r_n. \tag{2.2}$$

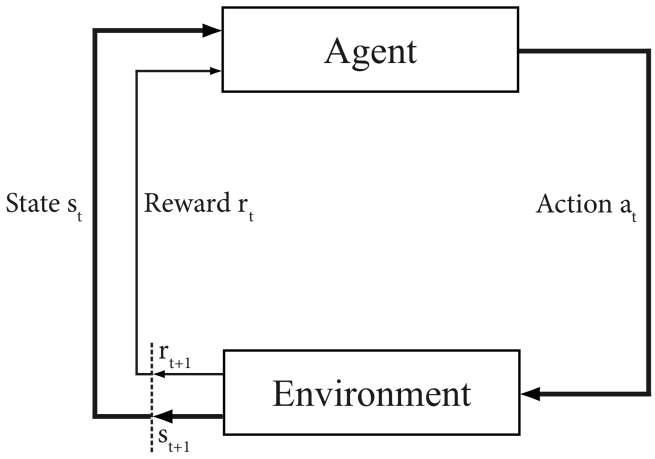


Figure 2.3: The principle of the reinforcement learning cycle shows that when the agent performs an action a_t in state s_t and receives the reward r_{t+1} from the environment it ends up in state s_{t+1} [21, p. 233].

The reward for every time step t out of a total amount of n rewards are summed up to the expected total reward R_t at a specific time step.

A large number of real-world applications exist for the use of reinforcement learning, which has become increasingly popular. Possible real-world applications are resource management, robotics, games, or self-driving cars. It can therefore be deduced that there are a large number of possible areas of applications, which have yet to be discovered [6, p. 131].

2.2 Artificial Neural Networks

The field of Artificial Neural Networks (ANNs) is based on discoveries made in the field of neurosciences, specifically the electrochemical activity in networks of brain cells. ANNs connect a set of input signals and output signals by using a model, which is inspired by a replica of a biological brain and how it responds to and processes the stimulus it receives from sensory inputs. A human brain is composed of about 90 billion neurons, which are connected with each other. Although a biological brain is far more complex than an ANN, this also has interconnected neurons and follows the general principle of how a brain works when solving ML tasks [8, p. 241; 26, p. 5]. This section first begins with a brief insight into the basic concepts and mathematical models of an ANN and concludes by presenting two special ANN models in greater detail.

The learning function of a human brain can be represented as a mathematical function, which was first introduced by McCulloch and Pitts [22] in 1943 and can be seen in Figure 2.4. This simple mathematical model of a neuron includes a set of weighted inputs, the summation of the input data and an activation function that decides, whether a neuron will fire or not. In an ANN the neuron input is received from the output of another neuron and is passed through the synapses and connections between the neurons. To decide which information should be passed through the interconnected ANN,

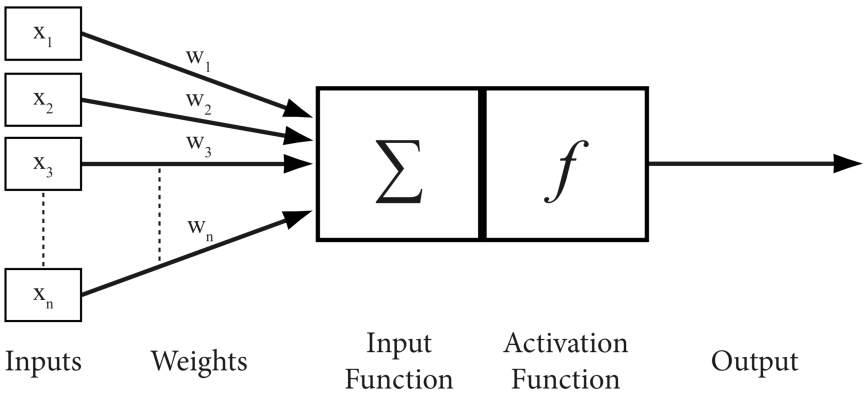


Figure 2.4: This diagram of a mathematical neuron is based on the paper of McCulloch and Pitts [22] shows how the inputs x and its weights w are fed into the input function to sum up the inputs and weights. This input sum is then passed on to the activation function $f(x)$ to produce the output [28, p. 728].

the synapses are weighted to influence the strength of the neural firing. The input is therefore multiplied by the weight value before it is fed into the neuron. Finally, the sum of the weighted input values determines whether the neuron will fire or not. The summation of these signals can be denoted as [21, p. 41]

$$h = \sum_{i=1}^{n} w_i x_i + b. \tag{2.3}$$

Each of the total amount of n inputs x is combined with their weights w by applying a matrix multiplication. An optional constant bias b can be added to fit the data better. The neuron will be fired, if the resulting value h is bigger than the defined threshold value θ (e.g., if $\theta = 0$ and $h = 0.5$, the neuron will fire, because $h > \theta$) [21, p. 41]. To get the final output value of the neuron, the resulting value h has to be fed into an *activation function* $f(x)$. Choosing the right activation function plays an essential role in aggregating the signals into the output signal, which will then be passed on to the following neurons of the network. This function takes the value h and performs a certain mathematical functional mapping on h. Depending on the structure of the ANN, there is a selection of many different types of activation functions to choose from. The most popular functions are mentioned and described in the following listing [8, pp. 242–243]:

- A *sigmoid function* $\sigma(x)$ maps a real-valued number to the range from 0 to 1.
- The *tanh function* $\tanh(x)$ takes real-valued numbers and outputs zero centred values in the range of $[-1, 1]$.
- With the *Rectified Linear Unit (ReLU) function* $f(x) = \max(0, x)$ it is possible to set negative values to zero and let positive values grow linearly. In comparison to the simpler functions *sigmoid* and *tanh*, this activation function is able to avoid and rectify vanishing gradient problems and is used in the majority of ML applications presently vailable.

For example, using a simple logistic or sigmoid activation function, an equation can be written as

$$y = f(h) = \begin{cases} 1 & \text{if } h > \theta, \\ 0 & \text{if } h \leq \theta. \end{cases} \tag{2.4}$$

The function $f(h)$ takes the dot product h and outputs the number 0 (i.e., not firing) or 1 (i.e., firing) depending on the threshold θ and the value of h. The output is the basis for further processing [21, p. 42].

A set of neurons is stacked together to form a layer and one or more layers create an ANN, in which these layers are connected together. There are many different types of ANNs available, which can be used for solving different ML problems. In this Thesis, the two important neural networks that are used in the implementation are presented. The first, *Convolutional Neural Network* is described in Section 2.2.1 this is followed by the description of the second ANN, *Recurrent Neural Networks* in Section 2.2.2.

2.2.1 Convolutional Neural Network

A Convolutional Neural Network (CNN) is a type of neural network that has not only shown impressive results in areas such as image recognition and image classification, but also in Natural Language Processing, where it has been successful in the various text classification tasks it was used for.

As the name indicates, this type of network employs a mathematical operation, called *convolution*, which is a special form of linear operation. This convolutional operation describes a rule for merging two sets of information by multiplying two functions f and g and is also known as a feature detector of a CNN. The convolution takes raw data or a feature map as input, applies a convolution kernel and returns a feature map as an output. A filter is often applied to the input data to reduce the information by filtering the data with the kernel. An operation of a one-dimensional convolution can be denoted as [26, p. 131]

$$h(i) = (f * g)(i) = \sum_{j=-\infty}^{+\infty} f(j) \cdot g(i-j). \tag{2.5}$$

The convolutional operation $h(i)$ defines the product of the input function f and the kernel function g in a one-dimensional domain. The convolution between f and g is calculated at the point i and acts on the same input data x. In the last part of the equation an integral over all values between negative and positive infinity is calculated. Then the value of $f(x)$ at the point j is multiplied with the value of $g(x)$ at the point $i - j$, which is the point of calculation of the convolution at a specific point j in the integral [12, pp. 331–332].

A CNN is basically a collection of layers of convolutions and consists of three different layers. A first layer performs several convolutions to produce an output and then in the second layer the output is run through a nonlinear activation function, such as *ReLU* or *tanh*, which have already been described in the introduction of Section 2.2. The third and final layer includes the *pooling* function to modify the output before it is processed further. A pooling layer replaces the output to reduce the output dimensionality so that it only focuses on the relevant information in the output data [12, pp. 339,342]. Filters, which are applied to the convolutions on the input matrix and generate feature maps,

onto which the pooling layers can be applied with the aim of further modifying the output are major components of a convolutional layer [26, p. 133].

The distinctive feature of this type of ANN is that not all the layers of the network are connected with each others; instead, each layer is connected to only a certain number of adjacent layers. They receive input, transform it in the layer and outputs the transformed input to the next layer. As already mentioned, the convolutional layer also consists of a set of filters with different sizes that transform the input data into a lower dimension. This transformation is then called a convolution operation [18, p. 49].

In a CNN model for Natural Language Processing tasks, the matrix consists of tokenized sentences, where the rows depict a word vector representation of each token. These vectors are word embeddings with a specified dimensionality, which is introduced in the next Chapter 3. The advantage of convolutional filters in text applications is their ability to learn good representations automatically, without representing the whole data vocabulary. By vectorizing discrete data (e.g., words or characters), it is possible to work with continuous data, which represent the discrete tokens. The filter sizes of a CNN for Natural Language Processing applications are equal to the dimensionality of the discrete input data, which means they are equal to the length of an input sentence. The layer then outputs feature maps, whose size influences the depth or dimensionality of the output. The hyperparameters for the convolutional filter are task-dependent and have to be defined based on the special use case of learning [33].

2.2.2 Recurrent Neural Network

A Recurrent Neural Network (RNN) is a special type of ANN, which takes sequential data as an input to solve ML problems, such as speech recognition, language modelling, sentiment analysis, or image captioning [15, p. 175]. RNNs make use of sharing parameters across numerous layers within the network. Therefore, making it possible to use the same weights over time and generalize across them. This functionality is important when special information occurs at multiple positions within the sequence [12, p. 373].

A typical RNN has nodes with recurrent connections to themselves, this enables to update and evolve the weights during the sequencing process. Each token of a sequence contributes new information, which then updates the current state of the RNN model. Based on a real-life example it can be compared with reading a sentence in a body of text: each new word that it is read in the sentence updates the current state of information, but it is also dependent on all the words that preceded it. This type of network is called *recurrent*, because as was previously mentioned, a task is performed for every token of an input sequence and its output depends on the previous computations. Figure 2.5 provides a visual representation of the architecture of an RNN network in form of a loop. On the left side of this visualization, it can be seen a general overview of the architecture in compact form and on the right, the chain-like nature with an unrolled structure of an RNN. If a sentence consists of five words, the network would also be unrolled into a 5-layer neural network, in which each layer belongs to a word of the sequence [18, p. 69].

To define the values of the hidden states of the network, the basic equation of an RNN is shown, using the variable h to represent the state,

$$h_t = f(h_{t-1}, x_t; \theta). \tag{2.6}$$

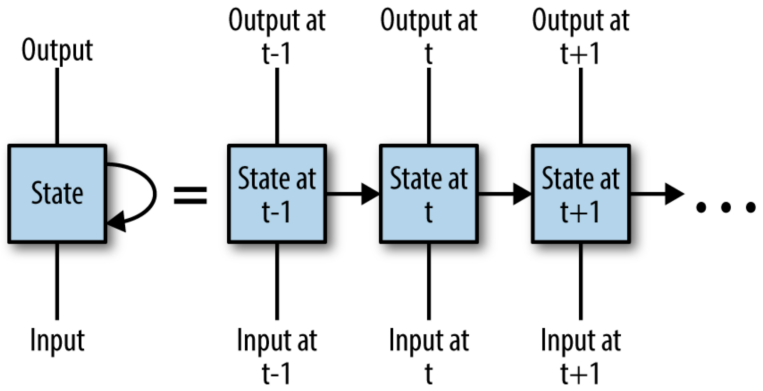


Figure 2.5: The unrolled structure of the RNN shows that the input x (i.e., token of a sequence) is observed at a time step t and the state vector h_t is updated from the previous state vector h_{t-1}. Processing new input always is dependent on the current state h_t and thus on the history of the sequence (see Figure 5-2 in [18, p. 69]).

This equation shows the calculation of the current state h_t, which represents the state of a hidden unit at the time t. This computation is based on the previous hidden state h_{t-1} and the current external input value x_t. To apply non-linearity to the output of the calculation, an activation function f with its parameters θ is used, which are shared during training of the model to generalize the model to the sequence length [12, pp. 376–377].

However, an RNN will reach its limits when handling longer sequences, because it is only capable of looking backwards for just a few steps. Therefore, the next technology presented seeks to resolve this limitation. *Long Short-Term Memory* solves this problem, which is also called *Vanishing Gradients Problem*; this occurs when the values of the gradient of a model are too small and the model requires too much time for training and as a result effectively stops learning.

Long Short-Term Memory

A Long Short-Term Memory (LSTM) network can be used to solve such tasks where the ability to be able to learn sequences with very long time spans in between is required. This network was introduced by Hochreiter and Schmidhuber in [17] in 1997. This network has a special memory mechanism with an improved ability to store information for longer periods of time. An LSTM network, which is capable of long-term dependencies is the most commonly used variation of RNN [15, p. 187].

The advantage of LSTM networks and the difference compared to traditional RNNs is the process by which they remember and forget information. Firstly, important information from the present should be remembered and unimportant information from the past can slowly be forgotten thereby providing space for newer recently acquired information. A special kind of mechanism, which is called *gate operation*, must therefore be used to apply this kind of knowledge filtering [4, p. 195]. This system of gating units, which is designed to control the flow of information within the network consists

of so-called *LSTM cells* and is located in an internal recurrence inside the RNN. The system consists of three gates, each of which are allocated different tasks, these will now be briefly described. The *forget gate* f defines how much information has to pass through the previous state h_{t-1} before it allows the network to forget less important information. The *input gate* i behaves in a very similar fashion to the forget gate f with the main difference being that the input gate defines how much information of the newly computed state for the input x will be kept in the network. Finally, the *output gate* o defines how much of the filtered information will be output by the LSTM cell [12, p. 410]. To be able to calculate the newly updated cell state c, it is also necessary to define an internal hidden state g, which is computed based on the current input x and the previous hidden state h_{t-1}. Then the cell state c at the time t is updated by multiplying the previous cell state c_{t-1} by the forget gate f and the internal hidden state g by the input gate i [15, p. 188].

Chapter 3

Natural Language Processing

The field of Natural Language Processing (NLP) deals with the process of understanding human (i.e., natural) language and how a computer can perform useful tasks with it. NLP combines computational linguistics, computer science, cognitive science, and AI while attempting to model a cognitive mechanism based on the underlying understanding of human language. Applications in NLP seek to facilitate the interaction between computers and human language and include tasks, such as speech recognition, natural language understanding, machine translation, sentiment analysis, or natural language generation [9, p. 1].

ML models work with text sequences transformed into vector space because it is necessary to work with continous data when trying to train a model. Therefore, the input sequences have to be mapped from words to lower-dimensional continuous vectors to enable a scalable representation, which then can be fed into an ANN layer. In addition, processing text data is very different compared with using image data during the training, as it possesses special properties, which make it difficult to handle at first. For this reason, natural language data requires some basic pre-processing steps before continuing with processing, these are shown in Chapter 6 [18, pp. 82, 93].

To build an efficient ML model for natural language, special techniques are required to process sequences of tokens. These tokens can consist of the following: words, individual characters or bytes. In this specific case, a *word-level model* was used to solve this ML problem. These word-level language models have to operate on extremely high-dimensional discrete space because the vocabulary size (i.e., the number of total words occurring in the dataset) is higher than in a set of characters occurring in the set of sequences [12, p. 461].

One possibility of representing the text data in a continuous space is to encode the words into a list of indices, which results in a sequence of integers corresponding to the words of a sentence. However, this technique, where each index is encoded in a binary categorical form will simply end up creating a problem, because this very large vocabulary set is not scalable for training purposes, which will result in data sparsity and computational issues [18, p. 87]. This problem is reffered to as the *curse of dimensionality*; this describes what occurs when an extremely high number of dimensions is present in the data. The resulting amount of possible configurations of a set of tokens increases exponentially as the number of tokens increases [12, p. 155]. For example, if a

joint probability of 10 words is applied to a dataset with a vocabulary size of 100,000, a potential number of $100000^{10} = 10^{50}$ parameters becomes possible. In a continuous space, it is possible to obtain a generalization to concatenate very short overlapping values and reduce the dimensions. However, in discrete spaces, this is not possible, because any alterations to the discrete variables could cause a very different result and may have a drastic impact on the value of the function [2].

To overcome this curse of dimensionality, a distributed representation of the words must be used to help recognize the similarity between two tokens or words without losing the ability to encode each word differently. In local representations, in contrast, tokens are modelled as discrete symbols and as has been previously mentioned their interactions with each other are encoded as a set of discrete relations. This distributed word representation is called *word embedding* and captures the relations and similarities between the tokens in form of activations in a vector. For instance, if the model has observed the word *dog* many times during training and the word *cat* only occurs a few times, correlations between these 2 words can be recognized, if the learned vectors of *dog* and *cat* are similar to each other; therefore, it is possible to share statistical strength between these 2 learned vectors [12, p. 464; 10, pp. 92, 117].

Each word of the vocabulary set is associated with a point (i.e., feature value) in a vector space with a d-dimensional vector. The number of features d is much smaller than the vocabulary size and is set in the process of hyperparameter tuning depending on the dataset and implemented ML model; this is described in greater detail for this specific implementation in Section 7.4. The mapping from the feature value to the d-dimensional vector space is implemented by means of an *embedding layer*, which performs as a look-up table [10, p. 49].

To summarize, a distributed representation or word embedding converts tokens, such as words into vectors in which the similarity between the vectors correlates with the semantic similarity between the words. Each word is transformed into its associated vector with a function φ and can be denoted as [15, p. 142]

$$\varphi(\text{“}Berlin\text{”}) - \varphi(\text{“}Germany\text{”}) \approx \varphi(\text{“}Paris\text{”}) - \varphi(\text{“}France\text{”}). \tag{3.1}$$

In this example, the word pairs (Paris, Berlin) and (France, Germany) are related in some way and cause the similarity between the vectors. Conversely, a possible value can also be deduced by reserving the equation to

$$\varphi(\text{“}Berlin\text{”}) - \varphi(\text{“}Germany\text{”}) + \varphi(\text{“}Paris\text{”}) \approx \varphi(\text{“}France\text{”}). \tag{3.2}$$

By knowing there is a semantic relationship between *Berlin* and *Germany*, the related value for *Paris* can be predicted using the associated semantic similarity between the vectors.

In this chapter, the focus is on tasks based on natural language, which was introduced within the bigger topic of *NLP*. The basic concept, which is about how we can turn text into structured data and let an ML system read or process it as natural language were presented. In the following section, the more specific topic of *Natural Language Generation* is described, where the aim is to show how an ML application is able to write natural language and turn structured data into text.

3.1 Natural Language Generation

The field of Natural Language Generation (NLG) is a subfield of NLP and has the task of producing new data based on a given training dataset. It does this by analyzing, interpreting and organizing data into comprehensive data. NLG is located within the field of AI and computational linguistics and is about constructing a computer system, which is able to produce understandable text in any human language from some underlying non-linguistic representation of the text [27].

With a *language model*, a probability distribution over the sequences of tokens in form of discrete values can be defined. The task of language modelling is about assigning a probability to a sequence and its tokens. In addition, the next token or word of a sequence will be predicted through a probability given to the token. To be able to model a sequence of tokens $w_{1:n}$, a probability has to be estimated for every part of the sequence. This can be done with a chain-rule of probability broken down to its component probabilities, which can be rewritten as [10, p. 105]

$$\begin{aligned} p(w_{1:n}) &= p(w_1)p(w_2 \mid w_1)p(w_3 \mid w_{1:2})p(w_4 \mid w_{1:3}) \cdots p(w_n \mid w_{1:n-1}) \\ &= \prod_{i=2}^{n} p(w_i \mid w_1, \cdots, w_{i-1}). \end{aligned} \tag{3.3}$$

The chain-rule makes it possible to predict a sequence of words, token by token. Assigning a probability score to the entire sequence is based on predicting tokens that are conditioned by the preceding tokens. As can be seen at the beginning of Equation 3.3, only the first word w_1 of a sentence is predicted and based on the first token, then a probability is assigned to the second token. The probability for the third token is then assigned based on the first and second token predictions $w_{1:2}$, and so on. This continues until the last token is reached, which is calculated based upon the previously modelled sequence $w_{1:n-1}$ [10, pp. 105–106].

Once the language model is trained on a given dataset, random sentences can be generated from the NLG model according to the following process: at the beginning, the first token or word of a sequence is predicted based on a predicted probability distribution over the first word conditioned on the start symbol. Then, the second word is predicted conditioned on the first word and the probability distribution predicted for the second word. This process is continued until the last token of the sequence is reached. The task of NLG is mostly accomplished by an RNN architecture with LSTM cells, because it performs very well and is very efficient at capturing statistical regularities in sequential inputs [10, pp. 112, 163].

Language models are used in many real-world applications to generate text or modify sequences for NLG or NLP. Possible uses are image captioning, text summarization, machine translation, spelling correction, response generation used for chatbots or personal assistants, or text generations, such as fake reviews or fake news [15, p. 178].

Chapter 4

Deep Generative Modelling

In this chapter, the focus is on the deep generative models used in Unsupervised Learning, which have been described in Section 2.1.2. The goal is to create new data based on a training dataset and its distribution p_{data} by representing an estimation of a probability distribution p_{model} [11, p. 2].

In contrast to generative models, discriminative models are used in Supervised Learning (described in Section 2.1.1) and map input data x to a particular output label y, which will be termed conditional probability and can be denoted as $p(y \mid x)$. However, a generative model learns the input data x as well as the output label y and estimates a joint probability distribution, denoted as $p(x, y)$. Which means these models possess the ability to analyse the underlying hidden structure of unlabelled training data p_{data} to produce a new probability distribution p_{model} [12, p. 105; 15, p. 248].

In the field of generative modelling, it is common practice to use the principle of *maximum likelihood estimation*. The goal of the maximum likelihood method is to estimate a probability distribution, parameterized by parameters θ to maximize the likelihood of the training data, given a set of observations [11, pp. 8–12]. The maximum likelihood estimator for parameters θ is then defined as

$$\theta_{ML} = \arg\max_{\theta} \prod_{t=1}^{n} p_{model}\left(x_t \mid \theta\right). \tag{4.1}$$

In this equation a set of n independent and identically distributed observations $\{x_1, \cdots, x_n\}$ from the unknown data distribution p_{data} is used. Therefore $p_{model}\left(x_t \mid \theta\right)$ represents a probability distribution over the same space indexed by θ and maps x to an estimated real number from the true probability $p_{data}(x)$. After calculating the product with $\prod_{t=1}^{n}$, this parameter, which maximizes the likelihood, can be searched by the $\arg\max_{\theta}$ function [12, p. 131].

Several generative models have been proposed in recent years and three common techniques for NLG in general are discussed in the following sections. First, the principles and related works for *Autoregressive Models* and *Variational Auto-Encoder* will be introduced and then *Generative Adversarial Networks* are described in more detail.

4.1 Autoregressive Models

Autoregressive models are also known as Fully Visible Belief Networks (FVBN), represent a very simple approach to deep generative modelling and are based on the idea of generating new samples using previous output data. They take the output from the previous step as an input for the regression of the next time step to predict new values. For this, the chain rule can be applied to decompose a joint probability distribution $p_{model}(x)$ over an n-dimensional vector x of observed variables to obtain a product of conditionals with [12, p. 705]

$$p_{model}(x) = \prod_{t=1}^{n} p_{model}\left(x_t \mid x_1, \cdots, x_{t-1}\right). \tag{4.2}$$

Autoregressive models are commonly used in deep generative modelling, such as image generation, audio generation or NLG. A very common autoregressive model introduced by van den Oord et al. in 2016 and used for the continuous data generation of images is PixelRNN [23]. This model assigns a probability to every pixel of an image based on the information derived from all the previously generated pixels. WaveNet [24] is another popular FVBN, which is used to produce raw audio data in the form of realistic human speech, this was also published by van den Oord et al. in 2016.

The best-known approach to the use of autoregressive models in NLG is the generation of text with RNNs, introduced in Section 2.2.2. Graves [14] described how RNNs could be used to generate complex sequences by predicting every data point one by one. This approach can be used for generating discrete data, such as text sequences and is visualized in Figure 4.1. The implementation of char-rnn [36], developed by Karpathy in 2015 uses this approach to generate text on character-level. The model receives a text file as input and learns to predict every character in a sequence by parameterizing the predictive distribution from the output vector for the next input.

4.2 Variational Auto-Encoder

Traditional auto-encoders are generally trained to generate new data similar to the original input data by attempting to copy the input to the output. Variational Auto-Encoders (VAE) [20] are a more probabilistic type of traditional auto-encoders. Instead of representing the data by just learning to compress data, such as auto-encoders, VAEs have the ability to generate new data by learning the parameters from a probability distribution model with which to represent the data. These auto-encoders consist of more complex architecture than autoregressive models and are made out of two RNNs, one of which represents the encoder and another the decoder network both of which are capable of producing parameters (see Figure 4.2).

To generate new samples, first a latent representation z is created from the distribution $p_{model}(z)$ and then the latent variables are decoded, to obtain new samples. The probabilistic encoder with the recognition model $q_\phi(z \mid x)$ produces a distribution over all the possible values of z based on the input x. Whereas model $p_\theta(x \mid z)$ represents the probabilistic decoder, which produces a distribution over all the possible corresponding values of x based on z [12, p. 696].

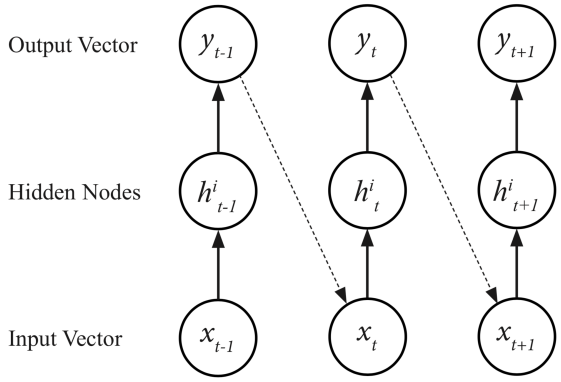


Figure 4.1: A vectorized input sequence $x = \{x_1, \cdots, x_t\}$ is passed to a hidden layer with a specific amount i of hidden nodes per time step $h^i = \{h^i_1, \cdots, h^i_t\}$. After computing the hidden vector sequences, the output vector sequence $y = \{y_1, \cdots, y_t\}$ can be calculated. The dashed arrows then represent the prediction process at one time step. For instance the prediction for x_t is based on y_{t-1} and can be denoted as $p(x_t \mid y_{t-1})$ [14, p. 3].

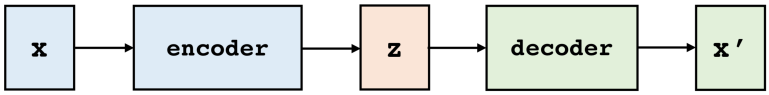


Figure 4.2: Basic structure of VAE models: the encoder $q(z \mid x)$ compresses data x into a lower-dimensional latent space z. To generate new samples, which look similar to the input data, the decoder $p(x \mid z)$ obtains input through the latent representation z and outputs its reconstruction x'.

An intent of VAEs is training by maximizing a variational lower bound $\mathcal{L}(\theta, \phi; x_i)$ of the data log-likelihood based on the data point x. Optimizing the variational lower bound affects both, the parameters of the encoder ϕ and the parameter of the decoder network θ. To measure and optimize the dissimilarity between the probability distributions of encoder and decoder, a *Kullback-Leibler (KL)* divergence $D_{KL}(q_\phi(z \mid x_i) \mid\mid p_\theta(z))$ is minimized to keep p close to q [12, p. 697; 20, p. 3].

There are many VAE implementations for continuous data, such as images, but there are only a few implementations for use with text, which is based on discrete data. Bowman et al. [5] published a paper in 2016 with regard to the case of NLG, which introduce an RNN-based VAE that integrated latent distributed representations of entire sentences. The advantage compared to traditional RNN language models is the explicit ability to model the holistic properties of sentences, such as the style or topic of a text. For that specific VAE model for text generation, two single-layer LSTM networks were used for the encoding and decoding network.

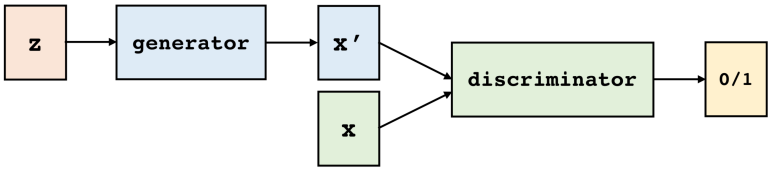


Figure 4.3: Basic structure of GAN models: generator turns random noise z into fake data x' and attempts to fool the discriminator. The discriminator tries to distinguish fake input x' from real input x.

4.3 Generative Adversarial Networks

The basic idea behind a Generative Adversarial Network (GAN), introduced by Goodfellow et al. [13] in 2014, is to train two models simultaneously: the generative model produces new samples based on a training dataset p_{data} and the discriminative model classifies the samples into two different classes (real or fake), as described in Figure 4.3. The real samples come from the training dataset p_{data} and the fake data samples were generated by the network and derive from p_{model}.

The generator G uses parameters θ_G and input data z to generate new samples with $x = G(z; \theta_G)$. The discriminator D, however, tries to distinguish between samples drawn from the training dataset x or samples drawn from the generator x' by taking the samples as input and using θ_D as parameters. The output of D should then indicate the probability, if the sample comes from p_{data} or p_{model} by emitting a probability value given by $D(x; \theta_D)$. The goal of both neural networks is to minimize their costs by controlling their own parameters θ_G and θ_D [12, p. 699].

This technology is a relatively new type of deep generative modelling, which provides training via an adversarial process and corresponds to a zero-sum or two-player minimax game. To obtain the best result for both models, the payoffs for both models G and D can be maximized by applying [13]

$$\min_G \max_D v(D, G) = \mathbb{E}_{x \sim p_{data}(x)} \log D(x) + \mathbb{E}_{x \sim p_{model}(x)} \log(1 - D(x)). \tag{4.3}$$

On the one hand the equation describes how the expected value of a sample x from the real data distribution p_{data} should be maximized. On the other hand, the expected value of a sample x from the generated distribution p_{model} should be minimized. At the beginning of the training, $D(x)$ is almost 1 and therefore $(1 - D(x))$ is almost 0 because the model can very easily classify them correctly based on their differences in quality. During the training process, both values will increase and decrease and should settle in by the end of the model training.

In recent years, many GAN models have been published for the use of NLG applications. The most popular and relevant works will be briefly described in the following Section 4.3.1.

4.3.1 GAN models

There are several GAN models for text generation, but the majority of them are only used in combination with images, for the purpose of text-to-image generation or image captioning. GAN models, which are only used for generating pure textual data, are not very common.

In this section, two GAN models, which are suitable for NLG tasks are presented. Their concepts will be introduced, differences highlighted and this will provide a bridge to the following chapter which is about the GAN model selected for the implementation. This model SeqGAN [32] was introduced by Yu et al. in 2016 and is described in greater detail in Chapter 5.

TextGAN [34], proposed by Zhang et al. in 2016 is based on a generic framework containing a LSTM network and a CNN for adversarial training to generate realistic text data. The objective of TextGAN is to match the feature distribution while training the generator and use techniques for pre-training the model. Instead of training the network policy, as is common with most other models, TextGAN matches high-dimensional latent feature distributions, which should help to avoid the common problem of mode collapse.

Another GAN model for NLG is LeakGAN [16] proposed by Guo et al. in 2017 and has been mentioned in many publications. The model was developed especially for the generation of very long text sequences. The special feature of this model is the integration of two separate modules *Worker* and *Manager* in addition to the common GAN components. The Manager module extracts features of the data currently generated and returns a latent vector to guide the Worker for the generation of the next set of words. In this manner, the generative model is able to incorporate additional informative signals to all the generation steps. Therefore, an RL architecture can be used as a promising mechanism with the objective of applying this leaked information into the generation process.

Chapter 5

SeqGAN Architecture

Multiple models have already been published for NLG with GANs, and can therefore be reused for implementation in further projects. However, the generation of short sequences (e.g., news headlines) has so far been neglected and leaves room for improvement. One of the models intended for short sequence generation is called SeqGAN, as already mentioned in the previous chapter. The following parts are based on the initial paper [32] from Yu et al.

To solve the problem of generating discrete tokens, this particular GAN model integrates an RL approach. SeqGAN was developed for text and music generation and has already been mentioned in many papers and publications. Additionally, the model has already been used as a basis for new GAN models with the aim of defining models for specific use cases. The paper shows the results of experiments based on generating Chinese poems and political speeches of Barack Obama with a sequence length of 2 for Chinese poem generation and 20 for political speech generation. For this reason the concept and approach of Yu et al. with SeqGAN was chosen to improve the implementation for generating news headlines (short sequences).

The architecture, shown in Figure 5.1, describes the procedure of sequence generation with SeqGAN. On the left side, the mixed data with real and fake samples is fed into the discriminative model, which is trained to differentiate between real and generated sequences. On the right, the generator is represented as policy in traditional RL. The states are previous tokens, which have been stored in hidden states and the action is the generation of the next token. To evaluate parts of a sequence, the rest of a sequence is filled up by *Monte Carlo* search.

In this special GAN model, the idea and functionality of RL, which was already described in Section 2.1.3 is applied to solve the problem of generating discrete tokens. The generative model G with the parameters θ produces sequences $Y_{1:T} = (y_1, \cdots, y_t, \cdots, y_T)$ based on the real training dataset and the resulting vocabulary set of candidate tokens, while y_1 is the first token generated and y_T is the last token generated in a sequence. State s in RL represents the tokens already produced $(y_1, \cdots, y_{t-1})$ within a specific timestep t and an action a is then the next token y_t generated.

The generator $G_\theta(y_t \mid Y_{1:t-1})$ is a stochastic policy model, because the probability distribution is generated over the actions. Every new token y_t of the sequence Y is generated based on the tokens already generated by the sequence $Y_{1:t-1}$. SeqGAN uses an RNN with LSTM cells, which was described in Section 2.2.2, as a generative model,

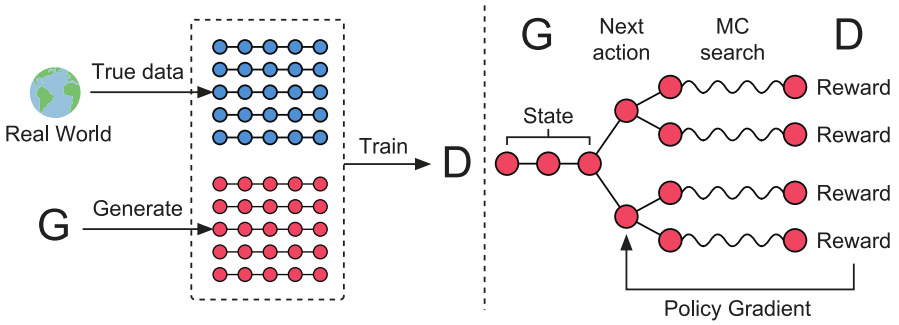


Figure 5.1: The architecture of SeqGAN on the left side of the diagram shows the process of the discriminative training and on the right side of the diagram how the training of the generative model including RL is approached (see Figure 1 in [32]).

which was able to apply backpropagation through time. First of all, input word embedding representations $x_1, \cdots, x_T$ are created out of the input sequences and are mapped into sequences of hidden states $h_1, \cdots, h_T$. This can be done with a update function g in a recursive way to get the current hidden state $h_t = g(h_{t-1}, x_t)$. Afterwards, the hidden states can be mapped into an output token distribution by adding a softmax output layer z with

$$p(y_t \mid x_1, \cdots, x_t) = z(h_t) = softmax(Wh_t + b). \tag{5.1}$$

The parameters of the softmax output layer include a bias vector b and a weights matrix W, which is updated during backpropagation. This final layer basically decides whether a neuron should be activated or not.

At the same time, the discriminative model D with its parameters ϕ is trained to improve the quality of the generator G_θ. The discriminator $D_\phi(Y_{1:T})$ predicts whether a sequence $Y_{1:T}$ is real or fake and its probability. The model is implemented as a CNN, introduced in Section 2.2.1 because it has already proven itself in classification tasks of text data. First of all, an embedding matrix is built out of every input sequence $x_1, \cdots, x_T$. Every input token x of sequence represents a token embedding and a matrix $\mathcal{E}_{1:T}$ is built, on which a new feature map can be produced by applying

$$c_i = \rho(w \otimes \mathcal{E}_{i:i+l-1} + b). \tag{5.2}$$

This feature map is created from a convolutional operation that is applied to a window size of l words by a kernel w. The number of kernels and window sizes can be easily changed to obtain various features and therefore achieve better results. At the end, a max-pooling layer is used over the feature maps $\tilde{c} = \max\{c_1, \cdots, c_{T-l+1}\}$ to extract the largest number of the vector c. More details with regard to convolutional operations are presented in Section 2.2.1. A special implementation feature of SeqGAN is the additional highway layer [29] to control the flow of information. Finally, a sigmoid activation function is used to predict the probability that a sentence is real.

During the complete training process, both models are trained simultaneously and their functions are called upon alternately. The objective of the generative model is to

Algorithm 5.1: This algorithm shows full details of SeqGAN.

Require: generator policy G_θ, rollout policy G_β, discriminator D_ϕ, training dataset $S = \{X_{1:T}\}$
Initialize G_θ and D_ϕ with their parameters θ and ϕ
Pre-train G_θ on dataset S using maximum likelihood estimation
G_θ generates fake samples for training D_ϕ
Pre-train D_ϕ with minimizing cross entropy loss
repeat
    for G training steps **do**
        Generate a sequence $Y_{1:T}$
        for timestep t in $1:T$ **do**
            Compute reward $Q(a, s)$
        end for
        Parameters of G are updated via policy gradient
    end for
    for D training steps **do**
        Fake sequences of G_θ and real sequences of S are combined
        D_ϕ is trained with combined dataset
    end for
until SeqGAN model converges

generate the sequence from the beginning with a start state s_0 to maximize the expected reward value. In this paper, therefore an action-value function $Q^{G\theta}_{D\phi}(a, s)$ is estimated with the use of the REINFORCE algorithm by Williams [31]. The probability estimated by the discriminative model $D_\theta(Y^n_{1:T})$ to be real is considered as the reward and can be denoted as

$$Q^{G\theta}_{D\phi}(a = y_T, s = Y_{1:T-1}) = D_\phi(Y_{1:T}). \tag{5.3}$$

The action-value function returns a value following policy G_θ, which means that the expected value is used for action a in a certain state s. The action represents the last token in a sequence y_T and the state is the collection of all previously generated tokens of a sequence $Y_{1:T-1}$. Since the discriminator only can analyse finished sequences, the state is always a collection of all tokens, except the last one. Therefore, the discriminative model can only provides a reward for finished sequences; however, for the training to be successful a long-term reward value is required at every timestep. To solve this problem and at the same time be able to evaluate an action-value for intermediate states, SeqGAN applies Monte Carlo search with a rollout policy G_β to generate the last unknown tokens of a sequence y_{T-t}. If the sequence is not yet generated until the end $t < T$ and no intermediate reward is available, the model iteratively creates the next-state value starting from the current state $s = Y_{1:t}$ rolling out to the end of a sequence based on the rollout policy G_β.

To get a better overview of the model's architecture, in Algorithm 5.1 every step of the process is specified in a pseudo code visualisation. After initializing the two neural networks G_θ and D_ϕ, both networks are pre-trained. This is done by means of maximum likelihood estimation (described in Chapter 4) for the generator based on the training

dataset. To pre-train the discriminative model by minimizing cross entropy loss, the generator first has to produce fake samples, which are fed into the discriminator. Then the training process starts in a loop until the model converges. This loop is further subdivided into two further separate loops, which are activated alternatively: one for the generator training steps and another for the discriminator training steps. In the G loop sequences $Y_{1:T}$ are generated and rewards $Q(a, s)$ for every timestep are computed. In the end, the parameters are updated via a policy gradient, which was described earlier in this chapter. In the training loop for D, the same amount of generated fake sequences of G and real sequences of the training dataset S are combined and fed into the discriminative model. In this way, the discriminator is able to keep pace with the generator because it is being periodically retrained with newly generated sequences from G_θ. The sequences of the combined training set permanently change during the training iterations to increase variability and improve predictions.

To evaluate and test the efficiency of the SeqGAN model, a real-world scenario is simulated with a random initialized LSTM network to capture the dependency of the tokens. This true model is an oracle designed to generate a real data distribution $p(x_t \mid x_1, \cdots, x_{t-1})$ for experiments. A big benefit of having such an oracle is that it evaluates the performance of the SeqGAN model, which would not be possible using real data alone. The best way to evaluate generated data is to use human observers to review the data. In this case, the evaluation is inspired by a human evaluation to create another distribution. The model only needs to minimize the *Negative Log-Likelihood (NLL)* of this natural data distribution, in which the oracle can be considered as a human observer for real-world problems. SeqGAN recommend a test stage where G_θ generates 100,000 sequences and calculates NLL_{oracle} from each sample of G_{oracle} and the average loss.

Chapter 6

Data Management

One of the most important tasks in a Machine Learning project is to handle the data correctly. The quality of the data generated not only depends on the implementation and the hyperparameters defined but also is strongly reliant on the data used and its quality. Therefore, the dataset used for the implementation is presented in Section 6.1 and the details of the dataset cleaning are shown in Section 6.2, as this is designed to improve the quality of the data. After data acquisition, it is also necessary to pre-process the data so as to be able to handle it correctly during the training process. This will be described in Section 6.3 and Section 6.4.

6.1 Dataset

One of the initial key requirements is to have enough data for purpose of training the model. The input sequences of the training data are fundamental for predictions and therefore also for the output sequences of the training. First of all, a suitable dataset for this special SeqGAN implementation for short sequences of news headlines has to be located. The dataset[1] used for the purpose of this implementation was downloaded from the platform Kaggle and provided 1,103,665 data entries. These where a collection of headlines from the newspaper New York Times[2] dated between 19 February 2003 and 17 October 2016 and all head different sequence lengths, which were UTF-8 encoded. The structure and entry examples of the dataset are visualized in Table 6.1.

6.2 Data Clearning

First, the data based on the original dataset had to be cleaned up to prepare it for further processing. To save memory and improve run time, unnecessary information was removed from the dataset before using it. The sequence length has been defined as five, as this is the average length of a headline (explained in more detail in Section 7.4.1). Therefore, all headlines with more than five words were deleted. Since only the text content of the headline was required, the date of publication was removed and a new field `id` added so that each headline could be uniquely identified. In addition, a new field

[1] https://www.kaggle.com/therohk/million-headlines
[2] https://www.nytimes.com/

Table 6.1: Structure and selection of entries of the original dataset.

publish_date	headline_text
20030219	aba decides against community broadcasting licence
20050812	figures highlight improved waiting times
20070315	uni innovation campus secures big it tenant
20090306	tourists staying closer to home
20110916	graduate nurse recruitment to proceed
20130703	tour de france simon gerrans in yellow as
20150326	my audio template
20161017	new royal adelaide hospital paper records foi

Table 6.2: Dataset of news headlines after data cleaning process.

id	headline_text	fake
118	omodei to stay in politics	0
1529	dead whale turning visitors away	0
62720	earthquake hits california	0
201799	indonesian fishermen spotted ashore	0
439793	robinho fined by city	0
653196	minister reveals earlier prison drama	0
919534	a league live streaming updates	0
1103611	2017 year in review	0

was added to the dataset, which contains the information, required to identify if the headline is real or fake. In the training dataset with the real sequences, all values of the new field `fake` are zero (i.e., false). This value means that the headlines marked with zero are real and these headlines marked with one are fake. With these data cleaning steps, the size of the dataset entries was reduced from 1,103,665 to 324,518. A selection of dataset entries after data cleaning is shown in Table 6.2.

6.3 Pre-Processing

Before the data can be used, it has to be pre-processed. Therefore, a CSV file has to be loaded, transformed and then mapped into a data frame:

```
1 df_real = pd.read_csv('headlines.csv', sep=',', usecols=['text', 'fake'])
2 df_real = df_real.sample(frac=1)
3
4 df_fake = pd.DataFrame(columns=['text', 'fake'])
5 df_evaluation = pd.DataFrame(columns=['text', 'fake'])
```

In line 2, the data frame is sampled and mixed up to improve variety. In addition to the data frame with the real samples, in line 4 a second empty data frame is created, which will be filled with the fake sequences during training. Another empty data frame is created in line 5 for the purpose of evaluating the adversarial training and the performance of the model and compares the results between a target generative LSTM model and the SeqGAN model. How this is implemented is described in Section 7.5 and how the evaluation works is explained in Chapter 8.

The way of preparation of the input sentences and how the information obtained was then processed within the neural network can be implemented in a simplified manner as follows:

```
1 tokenizer = Tokenizer(lower=False)
2 texts, labels = load_data(df_real)
3
4 tokenizer.fit_on_texts(texts)
5 sequences = tokenizer.texts_to_sequences(texts)
6 text_seq = pad_sequences(sequences, maxlen=SEQ_LENGTH)
7
8 labels = np.asarray(labels)
9 indices = np.arange(text_seq.shape[0])
10 np.random.shuffle[indices]
11 text_seq = text_seq[indices]
12 labels = labels[indices]
13
14 X_train = text_seq
15 y_train = to_categorical(labels, 2)
16
17 WORD_INDEX = [w for w, c in tokenizer.word_counts.items() if c > 5]
18 VOCAB_SIZE = len(WORD_INDEX)
19
20 def load_data(df):
21   texts = []
22   labels = []
23
24   for row in zip(df['text'], df['fake']):
25     texts.append(row[0].strip())
26     labels.append(row[1])
27
28   return texts, labels
```

At the beginning in line 1, a tokenizer had to be initialized and fitted into the training dataset. The tokenizer is designed to break up the sequences of strings into pieces of words (i.e., tokens). Before fitting the data, it is essential to split the data frame into texts and labels. This was accomplished with the function `load_data()` in lines 20 to

28. Here two iterable tuples based on the columns `text` and `fake` are extracted out of the data frame. Every entry of each tuple is added to its corresponding array. To be able to feed the data into the networks, first, the text has to fit the tokenizer and then in line 5, the word sequences have to be converted into the corresponding tokens with the function `texts_to_sequences()`. The sequences are then shaped to the same length (defined in `SEQ_LENGTH`) with the Keras function `pad_sequences()` in line 6 to fit the placeholder of the input variable in the networks. If a sequence is shorter than the defined sequence length, the missing tokens are filled up with zeros and in the beginning, longer sequences are removed from the dataset as was described in the previous Section 6.2. In lines 8 to 12 of the code snippet, the text and label arrays are arranged and shuffled to reduce variance and over-fitting (described in Section 7.4).

After pre-processing the input data, the word index with corresponding indices is created with the tokens, which appeared in the training dataset. Since headlines contain a large amount of different words, which includes people's names or company names, a very large word index will be created. To reduce the variance only tokens over five appearances are included in the word index. The resulting vocabulary size is calculated by getting the length of the word index, i.e., the number of tokens in the word index. What the variables `SEQ_LENGTH` and `VOCAB_SIZE` are used for is described in Section 7.4.

6.4 Data Handling

While the model is being trained, it is very important to load a suitable dataset, which all depends on the generative or discriminative training process. In addition to the real and fake data used in the adversarial training, also a data frame for evaluating the training exists. Therefore, four different functions need to be created to either load a specific dataset or return a mixed dataset:

```
1 def get_real_data()
2 def get_fake_data()
3 def get_evaluation_data()
4 def get_mixed_data()
```

The first three functions in lines 1 to 3 define which data to load from the specific datasets (`df_real`, `df_fake`, `df_evaluation`) in a defined batch size. In line 4 the function for loading a mixed dataset is defined. This composes a dataset with the same amount of real and fake sequences to be used for training the discriminator in the adversarial training process. The usage of each of the functions is shown in Section 7.5. For the training process, it is always important to feed different data into the models. This can be achieved by loading only parts of the dataset and by splitting the data into parts of batch sizes. The easiest way is to mix up the whole dataset and take the first n entries when n is the defined batch size (specified in Section 7.4). The newly generated samples are then directly included in the data frame for fake data to use them in the following training loops. At the end of training, this data frame (`df_fake`) is automatically saved in a CSV file with a unique name to document the data generated during the training process.

Chapter 7

SeqGAN Implementation

The intention of this chapter is to provide a deeper insight into the implementation by showing code snippets combined with corresponding explanations as to exactly why this solution was chosen. The technology used as a basis for the implementation is introduced in Section 7.1. The discriminative and the generative models are subsequently shown in detail in Sections 7.2 and 7.3. The structure and architecture of these models are applied according to the concept and approach of SeqGAN. In Section 7.4, the hyperparameter tuning is presented with an explanation of the values selected for the parameters and the reason why these were selected. The adversarial training process in Section 7.5, in which the combination and interaction between both network and training process is visualized and explained concludes the implementation chapter.

7.1 Technology Stack

It is crucial to ensure that enough computational power, for management of the amount of parameters required by the bigger and more complex neural networks is available, in this case to deal with Machine Learning (ML) and Deep Learning (DL) models. To handle this amount of data, a Graphics Processing Unit (GPU) instead of a Central Processing Unit (CPU) will be required. For this reason the cloud-based service *Google Colaboratory*[1] was selected for the implementation and training of the model. It is a research project initiated by Google created to support ML education and research tasks. The platform does not require any additional setup and runs entirely in the cloud. It provides upload and creation facilities for Jupyter Notebooks and the possibility of switching the runtime from CPU to GPU for the duration of a single 24-hours training session, when required. A GPU environment is essential for the training of a GAN because it is more efficient for matrix multiplications and it can provide more computational units and a higher bandwidth to retrieve data from memory.

Jupyter[2] notebook is a web application used in the development of open-source software for ML applications. It has the ability to create and share documents containing live code, equations, visualizations and descriptions in the form of text. In addition to ML tasks, Jupyter notebooks are also useful for data cleaning and transformation,

[1] https://colab.research.google.com/
[2] https://jupyter.org/

numerical simulation, statistical modelling or data visualization applications.

Python[3] is one of the most popular and powerful interpreted high-level server-side programming languages available for ML tasks; its use in this implementation is specifically to develop the DL tasks. Unlike other programming languages, Python is suitable for both research and development projects and provides a complete language and platform. The simple syntax and rapid implementation offered by Python make it possible to quickly and easily experiment with new ideas and create prototypes. The huge amount of inbuilt libraries and modules, which are available, are an added bonus when completing a variety of different tasks.

The open-source ML library *TensorFlow*[4], developed by Google, was chosen for use in this implementation on the grounds of its easy to use mathematical functionality. The library makes it possible to obtain high-performance numerical computations. The primary reason, why TensorFlow was preferred over other popular libraries, such as *Keras* or *PyTorch* was the integrated Reinforcement Learning functionality, which requires more modifications of calculations (e.g., a reward must be calculated within the network). As can be seen in many rankings of ML libraries, TensorFlow is extremely popular for most ML implementations. The popularity of TensorFlow ensures that the combined experience of a very large community and extensive documentation are available. In Figure 7.1 the *Deep Learning Framework Power Scores 2018* [35] from the data science platform *Towards Data Science* is displayed, which not only considered the popularity, but also the usage and interest. Following evaluation categories were chosen to provide a well-rounded view of DL frameworks based on popularity and interests:

- Online Job Listings,
- KDnuggets Usage Survey,
- Google Search Volume,
- Medium Articles,
- Amazon Books,
- ArXiv Articles,
- GitHub Activity.

As can be seen from the listing above, a variety of different perspectives is offered to guarantee an extensive evaluation of possible DL frameworks. These categories on the one hand includes the point of view of implementation, such as GitHub[5] activity rankings, the Google search volume or articles from the platform Medium[6] and on the other hand academical sources, such as research articles from ArXiv[7] or available books on Amazon[8]. Further details can be looked up in the original article [35].

To make the training process and the GAN computations easier to understand, it is necessary that the TensorFlow graph and quantitative metrics, illustrating loss and accuracy be plotted in the form of a diagram. For the purposes of this implementation, this was done with the suite of visualization tools provided by TensorFlow, called *Ten-*

[3] https://www.python.org/
[4] https://www.tensorflow.org/
[5] https://github.com/
[6] https://medium.com/
[7] https://arxiv.org/
[8] https://www.amazon.com/

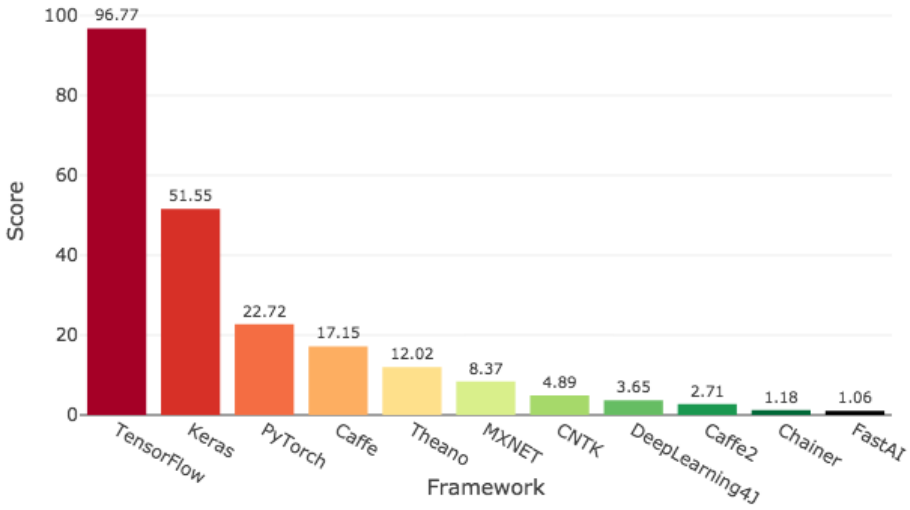


Figure 7.1: The *Deep Learning Framework Power Scores 2018* shows the popularity of a selection of DL frameworks. As can be seen in the diagram, TensorFlow occupies the first place with a score of 96.77 of the possible highest value of 100 [35].

sorBoard[9]. This tool is used to monitor, debug and analyze the training process and learning curves. The tool provides the most commonly used forms of visualizations, such as scalars, images, audio, histograms or graphs.

For the data management, described in Chapter 6, Python's data analysis library *pandas*[10] is used to handle the input data. This library makes it possible to load the data from a CSV file and transform it into a tabular form to which operations and manipulations on the data can be easily applied.

For pre-processing and preparing the data for usage in the training process, the DL library *Keras*[11] was selected. This is a high-level neural network API and is part of the TensorFlow core. In this implementation, Keras was able to vectorize an input text corpus into a sequence of integers and generate a word index required for use in pre-processing tasks.

The fundamental library for scientific computing in Python is called *NumPy*[12] and is an efficient tool for performing mathematical and logical operations. The library provides support for high-performance multidimensional array objects and various derived objects, such as masked arrays or matrices.

[9] https://www.tensorflow.org/guide/summaries_and_tensorboard
[10] https://pandas.pydata.org/
[11] https://keras.io/
[12] https://www.numpy.org/

7.2 Discriminator

The task of the discriminator is to classify the sequences. Based on the two different labels of the sequences, the input data can be differentiated between real and fake data in a supervised way. The goal is to predict high probabilities for real samples and low probabilities for fake samples. The discriminative model is implemented in the form of a CNN with an embedding layer for proceeding text input and a softmax layer for calculating the predictions. The calculation of the loss and accuracy values provides an estimate of how well the model performs.

For the moment, pre-training was performed before the actual adversarial training process began to optimize the model's performance. Yu et al. recommend this approach in the initial paper of SeqGAN implementation and it was used in their experiments. However, this procedure led to a deterioration of the process, a detailed explanation of which is provided in Section 7.5. For this reason, the decision was taken to train and optimize without pre-training.

To get a deeper insight into the core components of a CNN for NLP, such as word embeddings or convolutions, refer to the detailed illustration in Chapter 3. The structure and different components of the model in the implementation are defined as follows:

```
1 def build_model(self):
2   embedding_layer = self.build_embedding_layer()
3   conv_maxpool_layers = self.build_convolution_maxpool_layers()
4   scores, predictions = self.build_softmax_layer()
5
6   loss = self.calc_mean_cross_entropy_loss()
7   accuracy = self.calc_accuracy()
8   optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(loss)
9
10  d_summary = tf.summary.merge([
11    tf.summary.scalar('d_loss', loss),
12    tf.summary.scalar('d_accuracy', accuracy)
13  ])
```

The function of the *embedding layer* in line 2 is to map discrete tokens to vectors of continuous numbers. The aim of the mapping is to capture syntactic and semantic meanings and similarities between words and is a common feature of learning techniques in NLP. This layer consists of an embedding matrix initialized with a random distribution, which then learns connections during training [15, p. 140]. In line 3, *convolutional layers with max pooling operations* are created to represent the core component of the CNN. A layer transforms the input data with weights and biases from the neurons with a *ReLU* activation function. Every unit of a convolutional layer is connected to a specific number of units in a local region of the previous layer in an identical way and with the same weights. A pooling operation reduces the size of representations to only focus on the relevant changes. To achieve this reduction, the number of parameters and computations in the network is changed. In this implementation, the max pooling operation is used to capture the maximum of the input in each region [18, pp. 49–54]. The last layer in the network (line 4) the *softmax layer* produces the prediction output by applying matrix multiplications and picking the highest score. Before predicting the

probabilities, a highway architecture [29] is applied to improve the performance of the network. This architecture provides two additional gates, a transform gate and a carry gate, for applying non-linearity on some input data. A highway layer architecture can prevent problems with vanishing gradients, which can happen in DL applications. In this layer, a fully connected layer with sigmoid activation function used to generate probability outputs. To avoid over-fitting (refer to explanation in Section 7.4), dropout and L2 regularization are applied, to generalize the model beyond the training data. These regularization methods are essential to ensure the model can still make accurate predictions with data it has not seen before because it is too familiar with the training data. The dropout values are stated and justified in Chapter 7.4. Finally, a matrix multiplication calculates the scores and the resulting predictions are returned [32].

The quality and performance of the network is calculated and measured by the quantitative metrics *loss and accuracy* in lines 6 and 7 of the code snippet. The aim is to minimize the loss and maximize the accuracy during the training process. The loss is calculated with a mean cross-entropy between the ground truth label and the predicted probability. The accuracy is specified by the percentage of correctly predicted outputs. This value expresses the closeness of the predicted value to the labels known by the network and *Adam optimizer*, which is an adaptive learning rate method, defined in line 8, minimizes the loss. This optimizer is able to compute the learning rates separately for each of the different parameters. To enable a visual monitoring of the training progress and display the values of the metrics in a graph by means of TensorBoard, provided by the TensorFlow API the resulting values of accuracy and loss, which were updated after iteration of the training process were saved to a *summary*, these were later plotted in a TensorBoard diagram. Details and resulted graphs to the metrics loss and accuracy are shown in the results, refer to Chapter 8.

7.3 Generator

The generator has the task of generating high quality news sequences, which the discriminator should improve with its classification process during the adversarial training. The generator takes real input sequences and turns random noise according to the specific data distribution into fake data, which should resemble the real data. Implementation of the generator is in the form of an RNN with LSTM cells, the objective is to create sequences of words based on a vocabulary set in an unsupervised way. The goal is to generate data that is indistinguishable from real data.

At the end of the implementation process, a pre-training phase was included in the adversarial training; refer to the description in Section 7.5 for further details. To be able to apply the pre-training in the adversarial training, the implementation must be included in the generative model.

Before going further into the implementation of the model, first the network is described in general, which consists of four different parts:

1. Firstly, the core components of the RNN are constructed to make the model work.
2. The pre-training is defined so that it can be used in the adversarial training.
3. The components required for the training process are implemented; this represents the main part of the adversarial training.

4. Finally, the quantitative metrics are prepared to enable them to be visualized in TensorBoard diagrams.

How these parts are structured and implemented in the application can be seen in the following code snippet:

```
1 def build_model(self):
2   self.embedding_layer = self.build_embedding_layer(self.X_input)
3   self.outputs, final_state = self.build_lstm_layers()
4
5   pretrain_predictions, self.pretrain_loss = self.get_prediction_and_loss()
6   pretrain_optimizer = tf.train.AdamOptimizer(learning_rate=self.learning_rate)
7
8   self.predictions, self.loss = self.get_prediction_and_loss()
9   optimizer = tf.train.AdamOptimizer(learning_rate=self.learning_rate)
10
11  self.g_pretrain_summary = tf.summary.scalar('g_pretrain_loss', self.pretrain_loss)
12  self.g_train_summary = tf.summary.scalar('g_loss', self.loss)
```

The *embedding layer* in line 2 is implemented according to the same principle previously described in Section 7.2. The implementation and aim are identical to those used in the discriminative model: discrete tokens are mapped to vectors of numbers to capture meanings and similarities between words. The next layer in line 3 represents the main part of the network and builds up the *layers with LSTM cells* of the RNN. In this function, the layers created have corresponding hidden layer sizes; these are defined in the hyperparameters and are described in the next Section 7.4. In addition, to prevent over-fitting of the sequence generation each of the LSTM layers is wrapped in a dropout layer. This collection of layers is then stacked together to a `MultiRNNCell` which dynamically calculated the outputs and final state. In the beginning, an initial state is defined, where all values are zero. During the training, these weights are updated depending on the training progress.

The next two parts of the network consist of the same components and use the same operations, these are the pre-training, defined in lines 5, 6 and 11 and training defined in lines 8, 9 and 12. Both have the same objective: to calculate quantitative metrics and define the optimization process. Firstly, the *predictions and losses* must be calculated to provide the probabilities for newly generated sequences and the corresponding losses. The network output has to be multiplied with the weights to generate scores for each word of the word index. It generates a score for every word to predict, the likelihood of each word occurring as the next word in the sequence. To predict these probabilities of being the next token in the sequence, a softmax function is applied. This softmax function normalizes the prediction values to a range between 0 and 1. In addition, a temperature value is added to the logits before applying the softmax function to avoid word repetition and increase variety. For a detailed description of the temperature values, refer to the following Section 7.4. With the real sequences of the training dataset and the computed predictions, a softmax loss can be calculated to identify the difference between the predicted sequences and the data distribution of the real sequences of the training dataset. Just like the discriminator, the generative model consists of two *Adam optimizers* to cope with the pre-training and the training processes. This enables the losses to be reduced by applying the adaptive learning rate method. Details regarding the

learning rate used for this optimization process, along with the other hyperparameters, are presented in the following section. Finally, in line 11 (for pre-training) and 12 (for training), the quantitative metrics are fed into the *summary* again to track and visualize its changes in a TensorBoard graph. In this case, only the losses are used to create a diagram in TensorBoard.

A major reason, why SeqGAN is a very important GAN model for NLG is the ability to generate discrete tokens as previously mentioned in Chapter 5. The problem of passing the gradient update from the discriminative to the generative model does not exist as. SeqGAN provides a solution by directly performing the gradient policy update and modelling the generator as a stochastic policy in RL. Normally, it is a challenge to predict good outputs, because the discriminative model has no access to the complete sequence during generation. This is solved with intermediate state-action steps using a Monte Carlo search [32]. The implementation of this concept can be realized as follows in a simplified way:

```
1 def get_reward(self, sess, given_tokens, rollout_num, dis, dropout):
2   rewards = []
3
4   for i in range(rollout_num):
5     for given_num in range(1, self.seq_length):
6       feed_dict = { self.X_input: given_tokens, self.dropout_keep_prob: dropout }
7       samples = sess.run(self.predictions, feed_dict=feed_dict)
8       sentences, sequence = self.translate_samples(samples)
9       truth_probs = dis.get_truth_prob(sess, sequence)
10      if i == 0:
11        rewards.append(truth_probs)
12      else:
13        rewards[given_num-1] += truth_probs
14
15    truth_probs = dis.get_truth_prob(sess, given_tokens)
16    rewards[self.seq_length-1] += truth_probs
17
18  rewards /= rollout_num
19  return rewards
```

Firstly, the reward variable is initialized as an empty array. Next, two nested loops are defined, in which the functionality is implemented. The first loop defines the number of iterations the reward is calculated per sequence and the second loop is used to loop through the tokens in a sequence. So a reward is calculated for each token in a sequence. In line 7 new samples are generated by running the current session given as a parameter. As an input, the session uses the previously generated tokens and the predefined dropout value (explained in Section 7.4). The samples are generated in form of predictions, which have to be translated before further processing. This is done with the function called `translate_samples()`, which will be described in more detail in the following program code block. In line 9 the truth probability of each sequence token is calculated and initialized at the beginning of the function, which is provided by the discriminative model. In lines 10 to 13, the calculated truth probability of each sequence token is added to the reward variable, depending on whether it is the first value in the array or existing values are included. For the last token of a sequence, the procedure is a little

different, as can be seen in lines 15 and 16 of the code example. In this case, instead of using newly generated samples, to obtain the truth probability the given sequence generated from the adversarial training from the discriminator is used. Additionally, to complete the sentence reward, it is important to add the value at the end of the array, because it is the last token of the sequence. Before returning the rewards in line 19, the value is divided through the defined rollout number in line 18 to get an average reward value.

As already mentioned in the previous paragraph, the translation process of the sequences can be implemented with following function:

```
1 def translate_samples(self, sequence):
2   batch_softmax = np.reshape(sequence, [self.batch_size, self.seq_length, self.
      vocab_size])
3
4   sentences = []
5   vectors = []
6   for sequence in batch_softmax:
7     sentence = ''
8     vector = []
9     for pos in sequence:
10      vector_position = np.argmax(pos)
11      vector.append(vector_position)
12      word = self.word_index[vector_position]
13      sentence += word
14      sentence += ' '
15
16    sentences.append(sentence)
17    vectors.append(vector)
18
19  vectors = np.asarray(vectors)
20  return sentences, vectors
```

This function is used to translate the predictions into sequences of words and vectors. Therefore, in the adversarial training, it is possible either to print the newly generated sentences or to proceed further with the sequences of tokens. Firstly, in line 2 the sequence is reshaped to fit the translation process. For every position of a sequence in the current batch, the token is obtained by getting the highest value of the prediction (see line 10). The corresponding word is then obtained from the `WORD_INDEX` and added to the sentence, as can be seen in lines 12 to 14. Finally in line 20, two arrays are returned which both include sequences of words or tokens with the size of `BATCH_SIZE` and the length of `SEQ_LENGTH`. A description of the hyperparameters used in this function are listed in the next section.

7.4 Hyperparameter Tuning

The process of hyperparameter tuning is the way of finding the optimal combination of values to achieve the best outputs and metrics. With regard to training a GAN model, it is advisable to allow enough time with regard to planning this process, since such networks are very sensitive to changes in the hyperparameters. The performance can

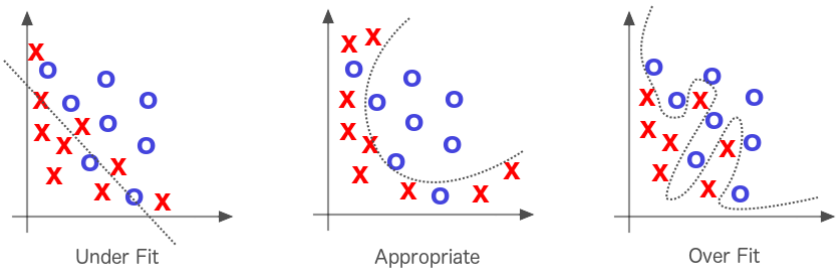


Figure 7.2: If the curve fits too well, as can be seen in the right diagram, the model is over-fitted, this means that it may have a low error rate for the training data, but poor output results. Under-fitting, in contrast, has the problem that important data patterns are not recognized and therefore it performs poorly, which is shown in the left diagram. The scatter plot in the middle illustrate an appropriate way to fit the data in a neural network (see Figure 1-7 in [26, p. 27]).

fluctuate greatly and this will lead to instability of the training process. The aim of this section is to give an insight into the decisions taken regarding the values used for the parameters. The meaning of the hyperparameters will be explained shortly and the decisions regarding the parameter values selected will be justified. The section is divided into 3 parts: Section 7.4.1 introduce the hyperparameters in general, Section 7.4.2 presents the specific parameters for the generative model and Section 7.4.3 presents the parameters for the discriminative model.

The wrong choice of hyperparameters often causes *under-fitting* or *over-fitting*, in which the data distribution is imitated in the wrong way. This is illustrated in form of different scatter plots as seen in Figure 7.2. When a model is excessively complex due to having too many parameters in relation to the number of data sequences, over-fitting occurs. If a model does not adapt enough from the underlying data distribution, under-fitting will occur [26, pp. 26–27].

Before presenting descriptions of the specific hyperparameters, a general description of one of the most important hyperparameters is explained in advance. The *learning rate* in a neural network has a strong impact on the stability and efficiency of the training process. It defines how the weights will be adjusted during training relative to the loss gradient. As can be seen in Figure 7.3, a very high learning rate can save training time but tend to overshoot the minimum, resulting in the algorithm bouncing backwards and forwards in the curve without ever finding the optimum point. A very low learning rate might find an error minimum, but will require a lot more training time and may reach the time limits of the scheduled training epochs. Another problem when learning rates are set too low is that the error minimum found, does not necessarily have to be the global minimum, but may be a local minimum [26, pp. 258–259]. The best choice for the learning rate always depends on the specific use case and implementation and needs a lot of adjustment effort and hyperparameter tuning to achieve the best result.

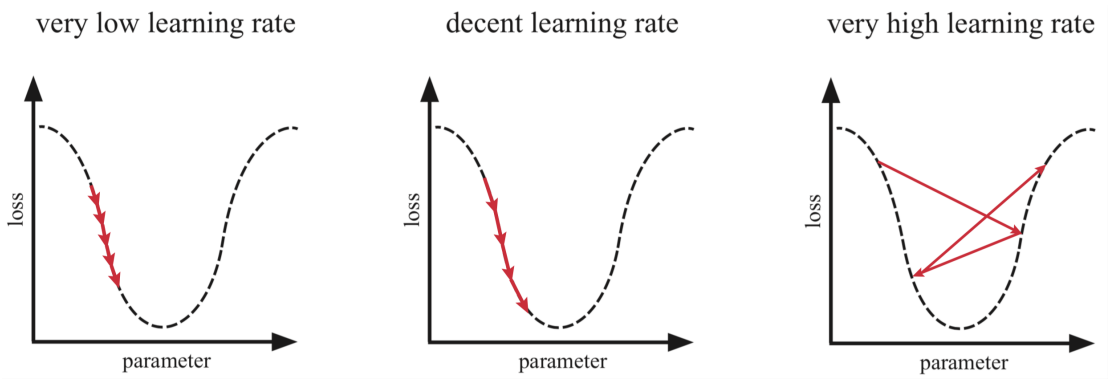


Figure 7.3: In the right diagram, a very high learning rate is shown, which results in a very unstable network training process. However, if the learning rate is very low, as seen in the diagram on the left, the training is extremely inefficient and will require a lengthy training period. The best example of a learning curve is presented in the middle diagram because the learning steps are suited to the curve [26, p. 258].

7.4.1 General Hyperparameters

In this section, all general hyperparameters used for the model implementation are presented. These general hyperparameters are either used for initializing the SeqGAN model, used for both models in the same way or used within the adversarial training process. Following the general hyperparameters defined in the implementation are presented:

```
1 BATCH_SIZE = 64
2 SEQ_LENGTH = 5
3 EMB_SIZE = 300
4 TOTAL_EPOCHS = 200
5 DROPOUT = 0.4
6 VOCAB_SIZE = len(WORD_INDEX)
```

The hyperparameters chosen for the implementation are explained and justified in the following section. This should help to give an insight into the decision making process.

Batch Size

The batch size defines the number of samples of a dataset that will be propagated through the network during one training epoch. The value is very small because the GPU used in this implementation cannot manage a higher batch size. As already mentioned in Section 7.1, the technology Google Colaboratory is used and its resources are limited. With a higher batch size, the GPU memory limit capacity is reached far sooner and the system stops the training of the network before it is completed. For this reason, the batch size was reduced to 64 to guarantee successful adversarial training. A smaller batch size resulted in the model learning very slowly combined with a tendency to under-fit.

Sequence Length

The value of the sequence length defines the final length the generated sentences should have. For this value, the average length of the headlines given in the original training dataset was calculated. In addition, this value should be appropriate for the specific use case (i.e., news headlines). Therefore, the value of the sequence is 5.

Embedding Size

As already described earlier in this chapter, a word embedding is used to represent words in a vector and map meanings and similarities between words. This hyperparameter embedding size, presented in line 3, specifies the dimension of a word embedding in the embedding layers of the generative and discriminative models. The length of the vectors, which represent the words, should be relatively high in this case to get a more expressive representation. For this reason, the value was set to 300 because the vocabulary size is very high and a larger size was required to be able to represent the relations between the word vectors. This meant that for each word a 300-dimensional vector is stored in the embedding layer. Smaller embedding sizes would be unable to represent the semantics between the word vectors adequately and bigger embedding sizes would slow down the complete training process. This can also cause hardware problems because Google Colaboratory only provides a single training session of 24 hours with a set GPU limit, which would be reached before the training could be completed.

Total Number of Epochs

This hyperparameter is defined in line 4 and it represents the number of iterations in the adversarial training process. However, the pre-training of the network is not included in this loop. All different training components are placed within this loop and are repeated with every training iteration. Exactly which components the adversarial training contains is shown in the next Section 7.5. The total number of epochs is relatively high because adjusting the parameter requires more time and iterations. In addition, it also is more expensive to train two neural networks at the same time alternately to adapt to each other. However, for the generative training process, a higher number of epochs would be a better solution to reduce further loss, but once again, the limits imposed by Google Colaboratory means that training stops after about 210 epochs because the limit of the GPU has been reached.

Dropout

This hyperparameter used for both networks, the discriminator and the generator, defines the percentage of values, which will be dropped out during the training process. Dropout is a common and powerful method of regularization technique to prevent overfitting, which was already described in general at the beginning of this section. Therefore, a random set of neurons is set to zero in each training iteration. In this form of implementation, a quarter of the full set of neurons is dropped out in both neural networks, which generalizes the network. As a result, there is an increased possibility that the networks will not be able to adapt to training data provided. In general, 0.5 is a very common dropout value, which works well on a big range of networks and goals [26,

p. 277]. The dropout value in this implementation is with 0.4 a bit smaller because in this case, a higher value led to under-fitting of the training data and the hyperparameter temperature was already introduced to prevent over-fitting.

Vocabulary Size

This hyperparameter, shown in line 6, defines the number of different words in the training dataset. As previously described in Section 6.3, the vocabulary size is obtained by the length of the word index, which is already reduced after the data cleaning process. The value of the vocabulary size is 17,751, which is relatively high because of the large amount of different words appearing in the training dataset.

7.4.2 Hyperparameters for Generator

This section presents the hyperparameters, which are specifically used for the generative model. These parameters help to improve the implementation of the RNN, which means that the generator trains more effectively and faster. The specific values chosen for the generative model are defined as follows:

```
1 G_PRETRAIN_EPOCHS = 100
2 G_HIDDEN_LAYER_SIZES = [128]
3 G_TEMPERATURE = 0.5
4 G_LEARNING_RATE = 0.01
```

The following paragraphs explain why these specific values are defined for the hyperparameters. Not only the meanings but also the decisions as to why the values of the hyperparameters were chosen are described and justified.

Epochs for Pre-Training

The number of epochs for the pre-training is important for the adversarial training process to begin this training with a higher performance of the generative model. It defines the length of pre-training or number of training iterations before starting the adversarial training to make the model learn patterns on its own. The number of epochs is defined as 100, because it reaches a well fitted graph, as can be seen in Section 8.3.2. A higher number would cause a data distribution too much adapted to the real training data and would end up in over-fitting. A lower number of epochs would not have such good effect of pre-training and would start adversarial training with worse performance.

Hidden Layer Sizes

A neural network can include one or more hidden layers, in which connections between the layers present weight values. The layer size then defines the number of neurons in a layer [26, p. 246]. In this specific case, only one layer with 128 hidden nodes was implemented. The reason for this is the batch size presented in Section 7.4.1. For an increase in hidden layers, a higher batch size is required, as a larger amount of data must be processed when the number of layers increases. A larger number of hidden layers containing less data can cause over-fitting as the network over adapts to the

training data and cannot generalize the findings. In this specific case, the increase in layers caused an increase in word repetition, as the network required more data to learn the underlying data structure correctly. If the layer has too few neurons in the hidden layer, the network cannot learn the structure within the total number of epochs defined. Too many nodes results in the same problem occurring as with the number of layers: the model tends to be over-fitted [26, p. 247]. The size of 128 nodes fits best because for the specific dataset size more nodes would cause a higher number of word repetition. Using fewer neurons in the hidden layers would result in the network under-fitting; this indicated that the model cannot approximate the data very well. How over-fitting and under-fitting are illustrated in a scatter plot is visualized in Figure 7.2 and is explained in detail at the beginning of the section.

Temperature

The temperature hyperparameter in line 3 is used to control the randomness of predictions by scaling the logits before applying a softmax function. In this implementation, the temperature is used to avoid word repetitions in the generated output sequences. The value has to be low because otherwise, the network will only predict the same words after a couple of epochs. A value lower than 0.5 will cause more diversity, but also an increase in mistakes occurring in the output sequences [25, p. 70]. The value 0.5 was chosen because it is the highest value where word repetitions no longer appeared.

Learning Rate

A general description of learning rates was included at the beginning of this section and it represents one of the most important hyperparameters in the training of a neural network. The learning rate of the generative model, presented in line 4, with its value of 0.01 is very high in contrast to the discriminative learning rate, presented in the next Section 7.4.3. The reason for this particular value is that the higher learning rate ran into an increase in word repetitions and the lower learning rate resulted in very small training steps, after half of the training epochs the losses stopped decreasing.

7.4.3 Hyperparameters for Discriminator

In this section, the optimal hyperparameters for the discriminative model are presented. These hyperparameters define the structure of the implemented CNN and helps to improve the performance of the model. The hyperparameters for the discriminative training were defined with following values:

```
1 D_NUM_CLASSES = 2
2 D_FILTER_SIZES = [1,2,3,4,5]
3 D_NUM_FILTERS = 128
4 D_LEARNING_RATE = 0.000005
```

The descriptions regarding why the specific values were chosen are listed below. Not only the meanings but also the decisions as to how the values of the hyperparameters were chosen are described and justified.

Number of Classes

The value of this hyperparameter, defined in line 1, represents the number of different classes in the classification process. In this case, two classes are available: a sequence can be real (first class) or fake (second class). The real sequences obtained from the training dataset are marked with zero and the fake sequence generated are marked with the number 1. Hence, the discriminator can check itself by the defined labels or classes if the classification of a sequence was correct.

Filter Sizes

This hyperparameter is a part of the filter or kernel of a CNN, which is explained in detail in Section 2.2.1. In brief, the filter size also called kernel size defines the number of tokens covered by the convolutional filters. Since the sequence length in this implementation is 5, a filter size can be defined for each word in the sequence. The filters slide over every word of the sequence to optimize the predictions. For example, the filter size of 2 means that the filter sees 2 words or embedding vectors at one time.

Number of Filters

As for the filter sizes, the hyperparameter for the number of filters is also described in Section 2.2.1. This hyperparameter, defined in line 3, represents the number of filters per filter size, where a filter stands for a neuron, which perform a specific convolution on the input data. In this case, for all 5 filter sizes, 128 filters are applied, which means 128 different convolutions are performed on the input data to the layer.

Learning Rate

As already mentioned in the learning rate description of the generator, this is one of the most important hyperparameters in neural network training and a general explanation is provided at the beginning of this section. The reason for the very low value of 0.000005, as defined in line 4, is the different training behaviour of the discriminative and the generative models. The classification process of the discriminator is optimized to run very fast and is able to classify every generated sequence as fake. So the generator is not able to optimize the generation process, because every sequence is classified as fake. Therefore, the discriminative learning processes have to learn at a slower pace so that they are adapted to each other. More details on the evaluation of the metrics are presented in Chapter 8.

7.5 Adversarial Training

After creating the discriminative and generative model, the SeqGAN model can be built up with these two neural networks. They are trained against each other to maximize and minimize the cost functions of the networks. To get the GAN working, the implemented networks have to be trained alternately. After every training epoch, the results of the currently trained network are passed on to the other network. So the predictions can be optimized after every iteration. However, a working model is not a guarantee that its

data output is good. This requires a lot of effort in hyperparameter tuning and training sessions. The aim of the implemented SeqGAN model is to train the neural networks in an adversarial way.

As mentioned in Section 7.3, a pre-training is included in the SeqGAN training process. This provides the advantage of optimizing the generative training before proceeding with the adversarial training. The reason, why a pre-training is used in this specific case is that the generator requires a lot more training to optimize its results. The discriminator has a much better performance during the training process and the resulting imbalance between these two networks must be compensated for with the pre-training of the generative model. This however means that the generator starts the adversarial training with the advantage of a better performance. Since a supervised classification process requires less training time and iterations, the pre-training for the discriminative model was deliberately omitted. The main reason for this is if the discriminator would also be pre-trained, the model would easily classify the generated sequences correctly and the adversarial training cannot correctly evolve. The pre-training process of the generative model is now shown and described:

```
1 for epoch in tnrange(G_PRETRAIN_EPOCHS, desc='gen_pretrain_loop'):
2   g_pretrain_losses = []
3
4   for _ in range(5):
5     X_train, y_train = get_real_data()
6     summary, g_pretrain_loss = generator.pretrain(sess, X_train, G_DROPOUT)
7     g_pretrain_losses.append(g_pretrain_loss)
8
9   g_pretrain_loss = np.mean(g_pretrain_losses)
10  writer.add_summary(summary, epoch)
```

For the pre-training loss, first, an empty array is initialized to obtain an average value from the results at the end of each epoch. Within the training loop in lines 4 to 7, another loop is created to pre-train the model five times to obtain the generated loss value. Before starting the pre-training process, real data has to be fetched to feed it into the generative model. The returned loss then is added to the loss array, from which the mean value in line 9 is later obtained. To visualize the training process, the values are once more added to a summary with which to plot a diagram in TensorBoard.

After pre-training of the generative model is finished, the actual training has to be managed and the main adversarial training process starts:

```
1 rollout = generator
2
3 for epoch in tnrange(TOTAL_EPOCHS, desc='gan_epoch_loop'):
4   X_train, y_train = get_real_data()
5   fake_sentences, fake_sequences = generator.generate(sess, X_train, G_DROPOUT)
6   df_fake = add_samples(fake_sentences, df_fake)
7
8   rewards = rollout.get_reward(sess, fake_sequences, 16, discriminator, G_DROPOUT)
9   summary = generator.train(sess, fake_sequences, rewards, G_DROPOUT)
10  writer.add_summary(summary, epoch)
11
12  if epoch % 5 == 0:
```

```
13    fake_sentences, fake_sequences = generator.generate(sess, X_train, G_DROPOUT)
14    df_evaluation = add_samples(fake_sentences, df_evaluation)
15    target_loss = target_loss(sess)
16
17  for _ in tnrange(5, desc='gen_train_loop'):
18    X_train, y_train = get_real_data()
19    fake_sentences, fake_sequences = generator.generate(sess, X_train, G_DROPOUT)
20    df_fake = add_samples(fake_sentences, df_fake)
21
22    for _ in tnrange(2, desc='dis_train_loop'):
23      for _ in range(BATCH_SIZE):
24        X_train, y_train = get_mixed_data()
25        summary = discriminator.train(sess, X_train, y_train, D_DROPOUT)
```

Before starting the main part of the adversarial training, a rollout has to be defined to obtain the reward. In line 1 of this code block, the generative model is duplicated and assigned to the new rollout variable. In line 3, the outer loop for the *adversarial GAN training* is created with a total epoch number of 200, which was described in the previous Section 7.4. Before starting the generation process, the model has to generate fake samples and calculate corresponding rewards out of the generated data. That should then become the basis for the following generation and optimization process. In lines 4 to 6, the real data is fetched by a data acquisition function, defined in Section 6.4 and fed into the generative model to create fake data. This function will, on the one hand, return the newly generated sequence in form of sentences with words and on the other hand create new sequences with corresponding vectors. These fake sentences are saved in the data frame for fake headlines and will then in turn be used in the adversarial training and extend the dataset for the generated headlines. In lines 8 to 10, the reward is calculated by the rollout model and is fed into the generative model. This function returns a summary with the calculated loss, which is then added to a writer to visualize the metric in a TensorBoard diagram. As already seen in the pre-training process of the generator the target loss is also calculated in the adversarial training. In lines 12 to 15 of the code snippet new fake sequences are generated and added to the evaluation dataset and then the target loss is calculated by the function `target_loss()`, explained in the following program code block. This calculation is performed once in five iterations of the adversarial training process.

In line 17, the *loop for the generative adversarial training* begins. First, a data batch with real sequences is fetched to feed it once more into the generative model and the returned fake sentences are added to the dataset with the fake samples. This loop includes five epochs within one epoch of adversarial training. In the generative loop in line 22, another loop is defined for the *discriminative training process*. In this inner training process, every batch of the defined batch size is classified by the discriminative model. Firstly, a data selection for the input data is fetched. In this case, the data should consist of an equal amount of fake and real data samples to train the discriminator and optimize the classification process. To train the model, not only input sequences but also the corresponding labels are needed for the process. Finally, again the summary is once again added to the writer to visualize the loss and accuracy in a TensorBoard diagram. This loop iterates through two epochs within one generative training epoch.

After every iteration of the adversarial training process, the last few samples gen-

erated within the training process are printed. This allows a rough estimation of the quality of the generated sequences.

7.5.1 Evaluation Approach

For Unsupervised Learning tasks, an evaluation is quite difficult to apply, because no labels exist for comparing if a sequence is true or false. In this generation task, a sequence should appear natural, however, that cannot be evaluated because it is not measurable. To solve the evaluation issue, another model was generated to obtain a basis for comparison.

Firstly, an oracle model `target_lstm` was initialized in the same way as the generative model to ensure it would have the same foundation. The oracle model is trained alongside the other two neural networks and is separated from the adversarial training. Since the oracle model and the generative model are initialized in the same way and both are pre-trained and trained in the same way, direct comparisons and evaluations of the adversarial training are possible. Therefore, the SeqGAN model can be evaluated by analyzing and comparing the losses of the oracle and generative model. This is described in more detail in the evaluation parts of Chapter 8.

To be able to perform a evaluation also the loss of the oracle model has to be calculated:

```
1 def target_loss(sess):
2   nll = []
3
4   for _ in range(5):
5     X_train, y_train = get_evaluation_data()
6     fake_sentences, fake_sequences = target_lstm.generate(sess, X_train, G_DROPOUT)
7     summary, g_pretrain_loss = target_lstm.pretrain(sess, X_train, G_DROPOUT)
8     nll.append(g_pretrain_loss)
9
10  return np.mean(nll)
```

For the target loss, an average *Negative Log-Likelihood (NLL)* is calculated by initializing an empty array in line 2, to which the calculated values are later added. Since the adversarial training loop only calls up this function once in five iterations, the loss is calculated 5 times in a loop within the `target_loss` function. This way, the deficit is compensated and an average loss value can be calculated. With every iteration, the *target LSTM* model is pre-trained with the evaluation dataset to minimize the NLL. The resulting loss value is then added to the defined variable for the NLL. At the end of the function in line 10, the average of the collected value is calculated and returned.

Chapter 8

Results and Evaluation

After going deeper into the implementation part of the Thesis, in this chapter, the results of the SeqGAN model implementation are visualized and analyzed in detail. Firstly, the sequences generated from the model are listed and evaluated in Section 8.1. The performance and metrics of the discriminative model are presented afterwards in Section 8.2. The performance of the generative model includes both the metrics and diagrams and also the comparison to the oracle model and the impact of the pre-training of the generator before starting the adversarial training. These results are presented and evaluated in Section 8.3.

8.1 Generated Headlines

Based on the SeqGAN implementation, presented in Chapter 7, a selection of headlines generated from the model are presented in this section. At the beginning of the training process, the sequences that are generated are of such poor quality that they cannot be identified as headlines. A selection of sequences generated after a few epochs during the training process is listed in Table 8.1 and can be easily identified as fake sequences. The resulting sentences are generated at random without structure or pattern, as can be seen from the first example “csl sienna academics 35b hoons” of Table 8.1. The sentence definition makes no sense and has no structure at all.

To offer a comparison to these poorly generated samples, the selection of outputs is shown below in Table 8.2, which were created in the middle to the end of the training process, already possess a clearly identifiable structure and are constructed in a logical way.

Additionally, a small set of real headlines of the training dataset is listed below in Table 8.3 to offer a comparison with the structure and characteristics of real sentences. As can be clearly seen from the sequences generated at the end of the training process in Table 8.2, the SeqGAN model was able to learn the special structure required to generate a plausible headline. These special characteristics of a headline can be identified by the given real samples: the headline starts with a noun followed by a verb and ends again with a noun. For instance the analysis of the first example “alcoholic independence changes general federalism” in Table 8.2 and the first example “australian flag celebrates 100th birthday” in Table 8.3 shows the structure learned from the model based on

Table 8.1: Headlines generated at the beginning of the adversarial training.

first generated headlines
csl sienna academics 35b hoons
underworld nimbin 22b 'god's 2908
piece cw genius parkinsonia norris
lebanon pulp guineas doll fc
lea cant monrovia roper toro
sniffers sepp einstein 93 springboks
2050 ambitious 200000 seaspray soaking

Table 8.2: Selection of well-generated headlines during the adversarial training.

last generated headlines
alcoholic independence changes general federalism
hollinger sacrifices oldham bareques cats'
scout criticise our stony improvements
stabber foresees dusty poison costing
qualify multinational contains cowal lessons
policemen guild reports overflowing critic
phil martin concerning grella artworks

concrete headlines: Both headlines begin with a noun ("alcoholic independence" and "australian flag"), are continued with a verb ("changes" and "celebrates") and end again with a noun ("general federalism" and "100th birthday").

Even if the model does not learn grammar, meaningfulness, or proper formulation, it is able to imitate the real sequences provided by the training dataset in a very believable way. A human might not be able to differentiate between the real and the fake sentences provided the content is disregarded.

8.2 Performance Discriminator

A very important factor is the performance of the discriminative model. It is responsible for the classification of the sequences and identifying if they are real or fake. In this section, the metrics of the discriminative model are analyzed and presented in the form of TensorBoard diagrams. In addition, possible learning curves with pre-training and

Table 8.3: Set of real headlines from the training dataset.

real headlines
australian flag celebrates 100th birthday
man charged over cooma murder
college to continue work experience
qantas plane makes emergency landing
hard work steals craftsmens identities
rising river isolates residents
nasa engineers foretold shuttle disaster

other hyperparameter values are shown, to help figure out problems with the interaction of discriminator and generator.

8.2.1 Accuracy

The most important performance measurement and evaluation method for classification tasks is the quantitative metric *accuracy*, which indicates the precision of the predictions during the classification process. The accuracy of the closeness of the predicted labels to the true values is calculated as [26, p. 38]

$$accuracy = (TP + TN) \div (TP + FP + TN + FN) \tag{8.1}$$

and can be achieved by dividing the correctly predicted sequences through the whole set of predictions. It is therefore possible to obtain the percentage of correct predictions based on all predictions. Correctly predicted sequences $TP + TN$ (True Positives and True Negatives) are the amount of all fake and real sequences, which were predicted correctly. This is the case if all generated sequences are classified as fake and all sequences of the training dataset are classified as real. The full amount predictions $TP + FP + TN + FN$ (True Positives, False Positives, True Negatives and False Negatives) includes not only the correctly predicted samples but also the incorrectly predicted sequences to get a basis for the division.

Figure 8.1 shows the learning curve of the accuracy created during the adversarial training process. A large number of irregularities are visible in the diagram. These irregularities occur due to the learning process of the generative model, which is presented in the next Section 8.3. The learning curve of the accuracy indicates that the generator learns to generate better sequences and the discriminator is no longer able to classify these fake sequences correctly. After more epochs of training, the discriminator then learns to classify them correctly and instructs the generator to optimize the sequences. With wrong predictions of the discriminator, the generator knows what was done well.

However, based on the entire learning curve of accuracy, it can be concluded that the discriminator is continuously improving and is therefore reliable during the training

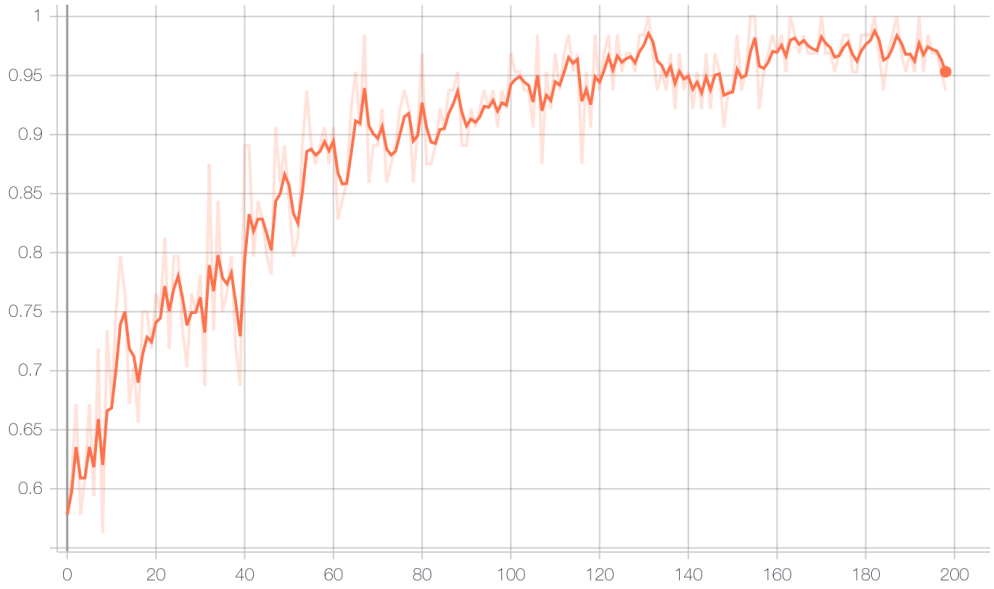


Figure 8.1: Accuracy of discriminative learning process in the form of a TensorBoard graph.

process. At the end of training after 200 epochs, the network is able to classify the sequences correctly with an accuracy of about 95% to 97%.

8.2.2 Loss

Another quantitative metric used in classification tasks is the *loss or error rate*, which indicated the proportion of samples for which the neural network produces an incorrect output [12, p. 104].

For this classification task, the *cross-entropy loss* is used for binary classification. It is a common loss function for classification models, whose output is a probability between the values 0 and 1. The more the prediction differs from the actual label, the larger the loss becomes. For instance, if a fake sequence is classified to the value of 0.05, the loss increases, because the prediction is closer to 0 than to 1 and thus deviates strongly from the real label value 1 [7, p. 80]. As can be seen in Figure 8.2, the discriminator is optimized constantly during the training process. The discriminator starts with a loss of 3.544, which can be reduced to 0.124 within 200 epochs of training. The shape of the learning curve is an indication of well-tuned hyperparameters with the loss decreasing until a point of stability around the epoch 120. A learning curve of an under-fitted model would be very flat or decrease rapidly until the end of training.

8.3 Performance Generator

The evaluation of the generator is very different compared to the discriminator, which is a Supervised Learning model; the generator is used for an Unsupervised Learning task.

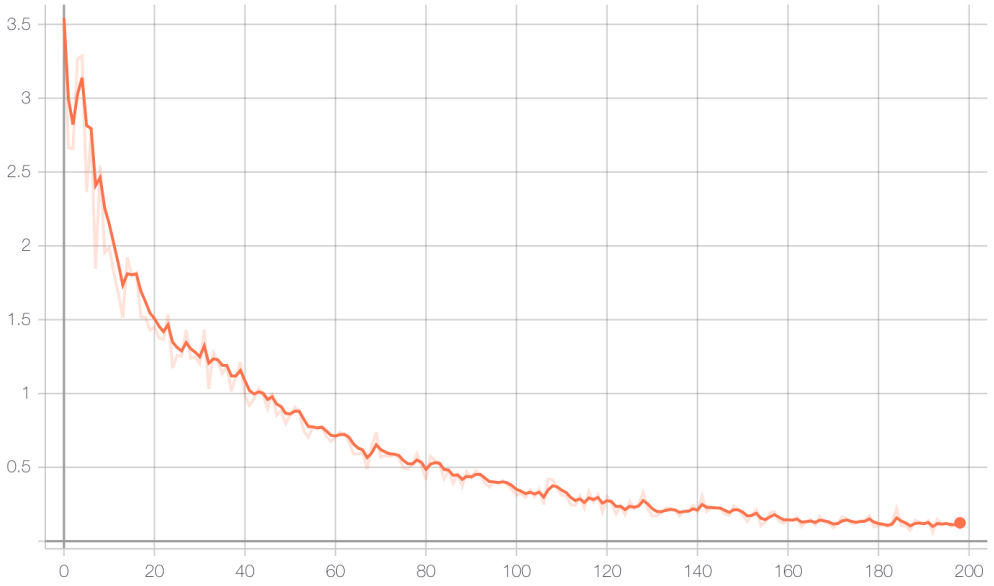


Figure 8.2: Loss of discriminative learning process in the form of a TensorBoard graph.

Therefore, it is not possible to calculate accuracy, because there are no labels available. As already seen in the previous section, the metrics and graphs are visualized in a TensorBaord diagram to provide a diagram to illustrate the results and performance. They are shown and described in the following sections.

8.3.1 Loss

The loss in an unsupervised model is calculated differently in case there are no labels with which to compare the prediction with the actual value. For the generator, a loss for reconstruction is used to obtain an error rate between the real and generated data distribution.

In Figure 8.3 the loss curve, illustrates the complete adversarial training process. In comparison to the loss curve of the discriminative model, the optimization process of the generator takes much longer. For this reason, it is important to use a pre-training session to optimize the model before starting adversarial training, which will be analyzed in detail in the following section. In the actual SeqGAN training, it is possible to improve the model further without starting from the beginning.

The loss starts with a pre-training value of 13.82 and ends with a value of 1.748 after 200 epochs of training. Without pre-training, the loss curve would start with a value of about 25 and would not be able to reduce the value by so much. In addition, Figure 8.3 shows that based on the learning curve the total epochs can be increased because the curve still is falling after 200 epochs of training. Unfortunately, as explained in Section 7.4 this is not possible when using the platform Google Colaboratory.

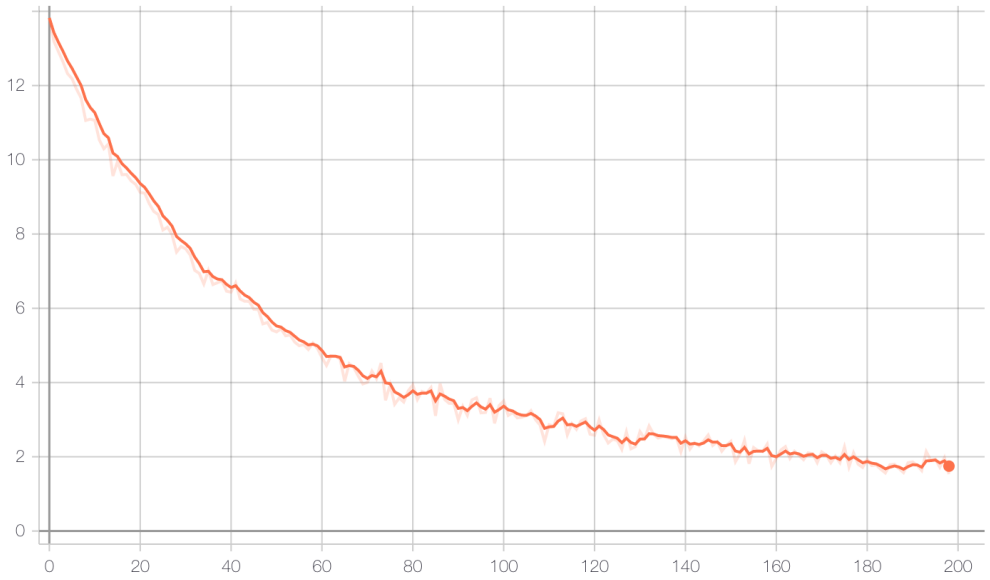


Figure 8.3: Loss of the generative learning process in form of a TensorBoard graph.

8.3.2 Pre-train Loss

As already mentioned before, the pre-training is important to optimize the generative model before starting the SeqGAN adversarial training process. This is necessary because the optimization of the generator is more advanced than the optimization of the discriminator.

For the purposes of this implementation, the same loss function as in the actual training process is used. This guarantees optimization in the same way and thus can be continued in the adversarial training. This has the advantage of adapting the parameters of the model in advance.

The loss curve of the generative pre-training is visualized in Figure 8.4. The training period is set to 100 epochs and shows continuous improvement. The number of epochs is not higher than 100, because the model should not be well adapted to the real data distribution before the adversarial training begins, as already described in Section 7.4.2.

At the beginning of pre-training, the generative loss has a value of 17.41 which was improved to 2.946 by the end of pre-training. This makes it possible to start the adversarial training with a much lower generative loss.

8.3.3 Oracle Model

As previously described in Chapter 5, an oracle model was implemented to have the possibility of evaluating the unsupervised generative model. The randomly initialized LSTM network is trained in the same way as the generator in the adversarial training. The difference compared to the SeqGAN training is the missing discriminative training within the generative training. As a result, it is possible to compare the performance of the SeqGAN model with a traditional training of an LSTM model.

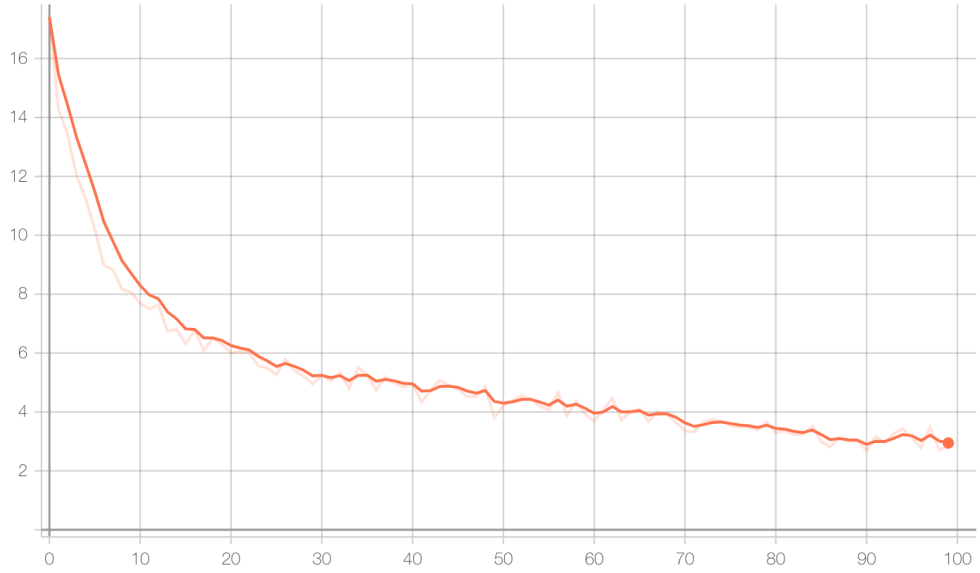


Figure 8.4: Pre-training loss of the generative learning process in the form of a TensorBoard graph.

Figure 8.5 represents the loss curve of the oracle model. After pre-training this evaluation model, as already shown in the previous Section 8.3.2 with the generative model, the loss for the oracle model is calculated so as to get a comparison value for the SeqGAN adversarial training. The loss can be reduced to a value of 3.568, which is more than twice the value of the generative loss. This indicates very good performance of the SeqGAN model implementation and training. In addition, the training process is more effective, because the oracle model starts with a value of 7.846 and can reduce the loss by a value of 4.278. The generative model in comparison can reduce the loss from 13.82 to 1.748, which results in a difference of 12.072.

These results prove that the adversarial training method provides better performance and is easier to optimize than a traditional LSTM model because the reduction of the loss from start to finish of the training process is significantly improved with the adversarial training.

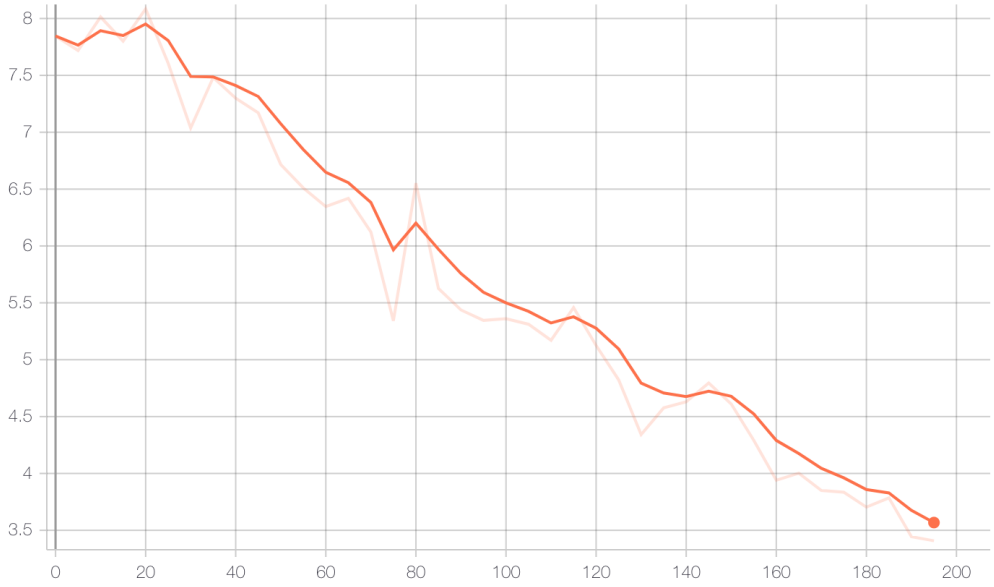


Figure 8.5: Loss curve of oracle model for evaluating the SeqGAN model in the form of a TensorBoard graph.

Chapter 9

Conclusion

The objective of this Thesis was to implement a GAN for the particular case of text generation of short sequences. To achieve this a lot of research was required to find out, what work had already been done in the field of deep generative modelling. After the decision had been taken to select the GAN model SeqGAN from Yu et al. [32], the main part of the Thesis Project began. As can be seen in Chapter 8, the GAN training outputs produced good results; thus, the goal was achieved, as it is definitely possible to implement tasks for NLG with GANs. Although the meanings of the headlines generated does not always make sense, the structure of the real headlines was learned very well and the fake headlines presented in Section 8.1 are indistinguishable from the real headlines provided one is prepared to disregard the final content. At the beginning of the Thesis Project, one of the most important things was to conduct enough research before proceeding with the actual implementation work. A sound knowledge of the subject helped to facilitate the implementation in many points. Another important issue was guaranteeing the quantity and suitability of the data available; this is reflected in the quality of the data generated during adversarial training. Without adequate data of suitable quality, it is impossible to produce high quality results. The challenges occurring during implementation are presented in the following Section 9.1, in which some recommendations are made and helpful information is provided for implementing an ML project with a GAN model. In Section 9.2 the prospects for further works is presented and how the Thesis Project could be modified or adapted for further implementation. Additional approaches for solving ML problems for NLG are also mentioned.

9.1 Challenges

Prior to and during the implementation of the Thesis Project, a few problems occurred, which had to be overcome. The most significant challenges and their solutions are mentioned in this section.

A challenge, which occurred right at the beginning of the implementation, was how to *handle the data* correctly. This challange kept reoccuring throughout the implementation. It is important for the training process to have the right balance between of real and fake data. The mixed dataset should not only consist of the same amount of real and fake data it should also be up to date. The samples that have just been generated should

also become part of the newly mixed datasets. One solution to this problem is to mix up the data before every data acquisition, so after every iteration new data is retrieved and as a result of the large number of epochs, the majority of the data is suitable for use. Another solution for the second part of the problem (i.e., including newly generated data) is to translate every generated sequence created within the current generation process. Therefore, the training function returns not only the predictions but also the final sequence and its corresponding translated sentence thereby making it possible to include these generated sentences into the data frame of the fake data.

Another challenge that emerged during the hyperparameter tuning process was the *extremely large word index* with many nouns (e.g., city names, people's and company names, including abbreviations, etc.). This was the main reason for the network failing in its attempts to learn how to recreate sentence structures and lead to inferior results. To solve this problem, the word index was reduced from 56,442 to 17,751 words. All words that appeared less than five times were removed making it possible to downsize the amount of words. The vocabulary size prior to the data cleaning, shown in Section 6.2 was 102,171 words, which posed considerable problems and resulted in very high loading times when recreating the sequences. The smaller vocabulary size and word index improved the GAN's ability to learn how to reconstruct sentences and build better headlines.

As already mentioned in Section 7.4, the amount of *word repetitions* increased after only a few epochs. The discriminator quickly learned how to differentiate between real and fake sequences; the metrics improved and it appeared that the GAN model was becoming good at learning, however the outputs of the generator deteriorated. The two solutions to the problem have already been described: firstly, the learning rates of the generator and discriminator should be well coordinated to ensure they do not end up in an imbalance during adversarial training, which will then cause over-fitting. The learning rate of the discriminator should not be higher than 0.000005, which is very low, otherwise, the learning steps become too high and the generator cannot find the optimal solution. If the learning rate of the discriminator would be higher than 0.000005, the generator will not receive adequate feedback if the predictions improve, because the discriminator will always identify the outputs as fake sequences. As the discriminator becomes more successful, the generator gradient vanishes and it ceases to learn, which results in diminished gradients. In addition, a temperature variable was added to the generative model to improve the learning rates and to enable more variety when predicting new sequences. This results in a greater diversity and helps avoid repetitions of words, but also increases the risk of predicting sequences that are randomly nested together.

Probably the biggest challenge was making the right decision regarding which *choice of hyperparameters* to use for both ANNs and their combination. As mentioned in other sections, it is very difficult to choose the best parameters, which is most suitable to achieve the best training process. Good results can only be generated if both networks have been well optimized. However, not only the individual performance is important, but also the interaction between these two networks has to be optimized, GAN models can be very sensitive and small deviations from the best parameter choice can cause inferior performance. A problem that was related to the choice of parameters was the limited resources of Google Colaboratory, because it often prevented the further optimi-

sation of the hyperparameters and the training of the GAN model had to be cancelled due to the limited GPU and memory space available.

An additional challenge presented by the Thesis Project was that there is quite a lot of research material at hand in the field of NLG with GANs, but there are *no experience reports or recommendations* available for the implementation of such models at the present. Nonetheless, it was possible to implement the concept in accordance with the initial paper of Yu et al. [32], but the detailed work required implementing the hyperparameter tuning was more about experimenting with various combinations and trying out different possibilities and analyzing the results of the best cases.

9.2 Future Work

The results of the implementation, which are referred to in Chapter 8, are impressive and confirm the claim that GANs are suitable for NLG. Nevertheless, further improvements could be made to the model in form of *hyperparameter tuning* which could further improve the overall performance of the GAN model. However, this would require a considerable increase in computational power and memory space, which is not presently available with Google Colaboratory. The use of other technologies or GPUs would be a possible future step for improving the implementation.

It is a very sophisticated task to coordinate the two networks with one another to obtain the best predictions; this required a lot of time and effort and the results had to be analyzed and evaluated in a comprehensible and professional way. To achieve this the training results had to be plotted and analyzed with the use of special metrics, this is also presented in Chapter 8. In addition to an evaluation with metrics, it would be a good idea to carry out a *human evaluation* combined with empirical studies to be able to measure the quality from a human way of thinking. This could be realized in many different ways, such as by using qualitative and quantitative research methods or a web platform providing real and fake sequences as a comparison. With human answers a representative study can be compiled to evaluate the results in an empirical way.

A few possible ways to further work on the implementation of the GAN model are presented and introduced as follows:

- Other GAN models, such as LeakGAN [16] or TextGAN [34], described in Section 4.3.1 could be tested for this specific use case to further improve the quality of the output sequences and analyze different ways to generate news headlines.
- A recent popular approach is to combine a Variational Auto-Encoder (VAE) and a GAN to use the advantages of both technologies to solve the ML problem. A possible concept for implementing this combination is provided for instance by Hu et al. [19]. This offers the possibility of using the generality and effectiveness of the techniques transferred from VAE and GAN.
- As mentioned in Section 7.1, Keras cannot be used for the implementation of this ML problem without applying modifications because it requires specific calculations for rewarding and passing feedback between the generative and discriminative models. The current model can be simplified if implementation is developed by combining with Keras technology and trying to modify the ML framework to be able to combine ANNs with the RL approach.

Appendix A

CD Contents

Format: CD, 700 MB

A.1 PDF-Dateien

Path: /

HoeglingerChrista2019.pdf Thesis

A.2 Source Code

Path: /project

datasets.zip	Folder with dataset used in the implementation
implementation.ipynb .	Source code of model implementation in a Jupyter notebook

References

Literature

[1] Ethem Alpaydin. *Introduction to Machine Learning*. MIT Press, 2010 (cit. on pp. 8, 9).

[2] Yoshua Bengio et al. "A Neural Probabilistic Language Model". *Journal of Machine Learning Research* 3 (2003), pp. 1137–1155 (cit. on p. 17).

[3] Giuseppe Bonaccorso. *Machine Learning Algorithms. A Reference Guide to Popular Algorithms for Data Science and Machine Learning*. Packt Publishing, 2017 (cit. on pp. 5, 8, 9).

[4] Rodolfo Bonnin. *Machine Learning for Developers*. Packt Publishing, 2017 (cit. on p. 14).

[5] Samuel R. Bowman et al. "Generating Sentences from a Continuous Space". In: *Proceedings of The 20th SIGNLL Conference on Computational Natural Language Learning*. 2016, pp. 10–21 (cit. on p. 21).

[6] Francois Chollet. *Deep Learning mit Python und Keras*. Mitp-Verlag, 2018 (cit. on pp. 5, 10).

[7] Francois Chollet. *Deep Learning with Python*. Manning Publications, 2017 (cit. on p. 52).

[8] Pratap Dangeti. *Statistics for Machine Learning*. Packt Publishing, 2017 (cit. on pp. 8, 10, 11).

[9] Li Deng and Yang Liu. *Deep Learning in Natural Language Processing*. Springer, 2018 (cit. on p. 16).

[10] Yoav Goldberg and Graeme Hirst. *Neural Network Methods in Natural Language Processing*. Morgan & Claypool Publishers, 2017 (cit. on pp. 17, 18).

[11] Ian J. Goodfellow. "NIPS 2016 Tutorial: Generative Adversarial Networks". 2017. URL: https://arxiv.org/abs/1701.00160. Pre-published (cit. on p. 19).

[12] Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016 (cit. on pp. 5, 7, 12–17, 19–22, 52).

[13] Ian J. Goodfellow et al. "Generative Adversarial Nets". In: *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*. NIPS'14. 2014, pp. 2672–2680 (cit. on pp. 1, 22).

[14] Alex Graves. “Generating Sequences With Recurrent Neural Networks”. 2014. URL: https://arxiv.org/abs/1308.0850. Pre-published (cit. on pp. 20, 21).

[15] Antonio Gulli and Sujit Pal. *Deep Learning with Keras*. Packt Publishing, 2017 (cit. on pp. 9, 13–15, 17–19, 35).

[16] Jiaxian Guo et al. “Long Text Generation via Adversarial Training with Leaked Information”. In: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*. AAAI’18. 2018, pp. 5141–5148 (cit. on pp. 23, 59).

[17] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. *Neural Computation* 9.8 (1997), pp. 1735–1780 (cit. on p. 14).

[18] Tom Hope, Yehezkel S. Resheff, and Itay Lieder. *Learning TensorFlow. A Guide to Building Deep Learning Systems*. O’Reilly, 2017 (cit. on pp. 13, 14, 16, 35).

[19] Zhiting Hu et al. “On Unifying Deep Generative Models”. In: *Proceedings of the 6th International Conference on Learning Representations*. ICLR’18. 2018 (cit. on p. 59).

[20] Diederik P. Kingma and Max Welling. “Auto-Encoding Variational Bayes”. In: *Proceedings of the 2nd International Conference on Learning Representations*. ICLR’14. 2014 (cit. on pp. 20, 21).

[21] Stephen Marsland. *Machine Learning. An Algorithmic Perspective*. 2nd. Chapman & Hall/CRC, 2014 (cit. on pp. 5, 7–12).

[22] Warren S. McCulloch and Walter Pitts. “A Logical Calculus of the Ideas Immanent in Nervous Activity”. *Bulletin of Mathematical Biology* 52 (1990), pp. 99–115 (cit. on pp. 10, 11).

[23] Aäron van den Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. “Pixel Recurrent Neural Networks”. In: *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*. ICML’16. 2016, pp. 1747–1756 (cit. on p. 20).

[24] Aäron van den Oord et al. “WaveNet: A Generative Model for Raw Audio”. In: *Proceedings of the 9th International Symposium on Computer Architecture Speech Synthesis Workshop*. ISCA’16. 2016 (cit. on p. 20).

[25] Douwe Osinga. *Deep Learning Kochbuch. Praxisrezepte für einen schnellen Einstieg*. O’Reilly, 2019 (cit. on p. 44).

[26] Josh Patterson and Adam Gibson. *Deep Learning. A Practitioner’s Approach*. O’Reilly, 2017 (cit. on pp. 5, 10, 12, 13, 40–44, 51).

[27] Ehud Reiter and Robert Dale. “Building Applied Natural Language Generation Systems”. *Natural Language Engineering* 3.1 (1997), pp. 57–87 (cit. on p. 18).

[28] Stuart J. Russell and Peter Norvig. *Artificial Intelligence. A Modern Approach*. 3rd. Prentice Hall, 2010 (cit. on pp. 4, 6, 7, 9, 11).

[29] Rupesh K. Srivastava, Klaus Greff, and Jürgen Schmidhuber. “Highway Networks”. 2015. URL: https://arxiv.org/abs/1505.00387. Pre-published (cit. on pp. 25, 36).

[30] Alan M. Turing. “Computing Machinery and Intelligence”. *Mind* 59.236 (1950), pp. 433–460 (cit. on p. 4).

[31] Ronald J. Williams. “Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning”. *Machine Learning* 8.3-4 (1992), pp. 229–256 (cit. on p. 26).

[32] Lantao Yu et al. “SeqGAN: Sequence Generative Adversarial Nets with Policy Gradient”. In: *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*. AAAI’17. 2017, pp. 2852–2858 (cit. on pp. 2, 23–25, 36, 38, 57, 59).

[33] Ye Zhang and Byron C. Wallace. “A Sensitivity Analysis of (and Practitioners’ Guide to) Convolutional Neural Networks for Sentence Classification”. In: *Proceedings of the 8th International Joint Conference on Natural Language Processing*. IJCNLP’17. 2017, pp. 253–263 (cit. on p. 13).

[34] Yizhe Zhang et al. “Adversarial Feature Matching for Text Generation”. In: *Proceedings of the 34th International Conference on Machine Learning*. PMLR, 2017, pp. 4006–4015 (cit. on pp. 23, 59).

Online sources

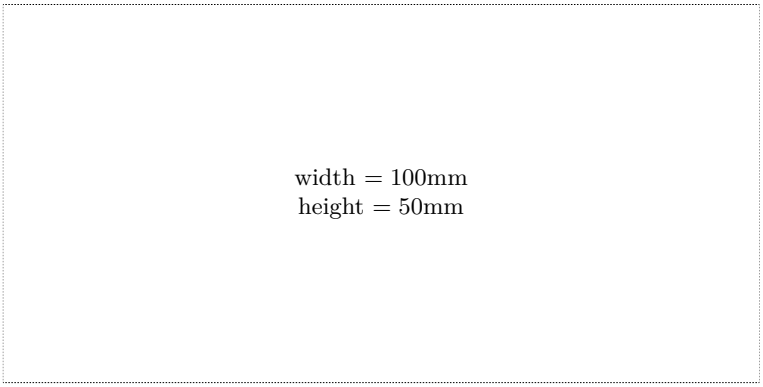
[35] Jeff Hale. *Deep Learning Framework Power Scores 2018*. 2018. URL: https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a (visited on 04/24/2019) (cit. on pp. 33, 34).

[36] Andrej Karpathy. *GitHub Repository: char-rnn*. 2015. URL: https://github.com/karpathy/char-rnn (visited on 03/28/2019) (cit. on p. 20).

[37] Andrej Karpathy. *The Unreasonable Effectiveness of Recurrent Neural Networks*. 2015. URL: http://karpathy.github.io/2015/05/21/rnn-effectiveness/ (visited on 03/16/2019) (cit. on p. 1).

Check Final Print Size

— Check final print size! —



— Remove this page after printing! —