TECHNISCHE UNIVERSITÄT WIEN

DIPLOMARBEIT

Comparison of Different Word Embeddings and Neural Network Types for Sentiment Analysis of German Political Speeches

zur erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Masterstudium Technische Mathematik
Schwerpunkt Diskrete Mathematik (DM)

eingereicht von

Daniel Kapla
Matrikelnummer 01128052

ausgeführt am Institut für Analysis und Scientific Computing
der Fakultät für Mathematik und Geoinformation der Technischen Universität Wien

Betreuung
Betreuer: Ao.Univ.Prof. DI Dr. Felix Breitenecker
Mitwirkung: Projektass. DI Dr. Nikolas Popper

Wien, 11.09.2019

Daniel Kapla

Dr. Felix Breitenecker

Abstract

The state of the art technology in natural language processing (NLP) is dominated by neural networks. These networks use word embeddings for text representation which are particularly well suited for representing meaning and relation between words. As this technology moves forward, word embeddings are freely available for other languages than English. In this thesis I apply this technology to Austrian political speeches to compare their performance in a sentiment analysis task. I use different word embeddings for the German language and combine them with different neural network architectures. In exhaustive experiments classifiers are trained and validated with sentences from stenographic protocols labeled with their level of negativity.

Zusammenfassung

Der Stand der Technik in der natürlichen Sprachverarbeitung (NLP) wird von neuronalen Netzwerken dominiert. Diese Netzwerke verwenden Worteinbettungen, die insbesondere dazu geeignet sind, um Bedeutung und Beziehungen zwischen Worten darzustellen. Heutzutage sind vortrainierte Worteinbettungen nicht nur für Englisch, sondern auch in anderen Sprechen frei erhältlich. In dieser Arbeit wende ich diese Technologie auf die Stimmungsanalyse von österreichischen politischen Reden an. Dafür werden unterschiedliche deutsche Worteinbettungen in Kombination mit verschiedenen neuronalen Netzwerken untersucht. In umfangreichen Experimenten werden Klassifikatoren mit Sätzen aus stenografischen Protokollen, welche mit ihrem Grad an Negativität gekennzeichnet sind, trainiert und miteinander verglichen.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The task of this work is to build and compare different sentiment classifiers for German text, especially for Austrian national parliamentary speeches. This thesis builds on the work of [Rudkowsky et al., 2017] where a continuous word embedding in combination with a neural network was used for text classification. This is the state of the art technology for natural text processing and as such subject for implementation in other languages than English. Continuing the work from [Rudkowsky et al., 2017] in this thesis different freely available word embeddings provided by the `Polyglot` library [Al-Rfou et al., 2013] which is a `word2vec` embedding[1] [Mikolov, Chen, Corrado and Dean, 2013; Mikolov, Sutskever, Chen, Corrado and Dean, 2013] and the `fastText` embedding [Bojanowski et al., 2016; Grave et al., 2018; Joulin et al., 2016; Mikolov et al., 2018] for the German language, are examined. With each of the different embeddings different neural network types are compared for their performance on the same text classification task. The variety of neural network types ranges from MLPs, which is the structure used in [Rudkowsky et al., 2017], over convolutional networks (CNNs) to different recurrent networks (RNNs) like GRUs and LSTMs. Even bidirectional recurrent networks and combinations of convolutional and recurrent networks are examined.

We start with an introduction to the field of machine learning with special attention to the given task in chapter 2. In chapter 3 neural networks are introduced and a variety of different architectures is presented, followed by chapter 4 which starts with some basics of natural text representations for processing with a special focus on the theoretical background of the `word2vec` and `fastText` embeddings. Finally, in chapter 5 the experimental setup

[1] There are three different embeddings provided by the `Polyglot` library, sadly only for the `cw` embedding its origin and type could be verified. The `cw` embedding is described in the `Polyglot` paper [Al-Rfou et al., 2013].

and technical details are presented, followed by stating the results of the experiments.

1.1 Datasets

The training data consists of more than 20.000 labeled sentences with labels from 0 to 4 where 0 means neutral while 4 stands for very negative. The sentences are gathered from press releases of Austrian political parties, parliamentary speech transcripts and media reports from 1996 to 2013. In sum, these are 56.000 speeches, consisting of more than 2 million sentences. This training data was initially created for [Rudkowsky et al., 2017].

The labels for the 20.000 sentences as training data were created via crowd coding. The human crowd coders where asked to classify training sentences without any context, meaning only the sentence itself was presented and should be labeled from neutral to very negative in 5 categories. To obtain a single score for a sentence the mean of all answers was computed for each sentence. Furthermore each single coder was checked for cheating or spamming by adding validation sentences with a predefined score into the data the coders where asked to score. Then all results of coders which did not reach an accuracy of over 75% at these test sentences were removed from the calculation of the final score. The final score was then computed by averaging the labels over all coders. This lead to a continuous score of negativity between 0 to 4 for each sentence in the dataset. With these scores the sentences were grouped into three classes, namely neutral C_0, negative C_1 and very negative C_2. The decision boundaries where chosen as 4/3 and 8/3, see figure 1.1.

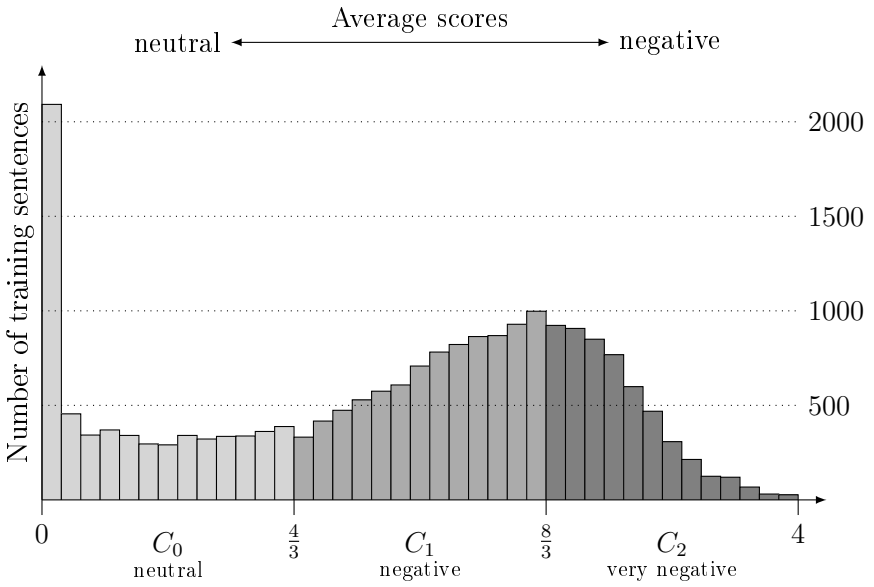

Figure 1.1: Average score distribution of all training sentences divided into three classes C_0 (neutral), C_1 (negative) and C_2 (very negative).

Chapter 2

Machine Learning

The term *machine learning* is coined by Arthur Lee Samuel who wrote in 1959 a paper called *"Some Studies in Machine Learning Using the Game of Checkers"* [Samuel, 1959]. There Samuel states:

> "The studies reported here have been concerned with the programming of a digital computer to behave in a way which, if done by human beings or animals, would be described as involving the process of learning."
>
> [Samuel, 1959]

Even though there is no exact definition given by Samuel he is attributed with a definition involving the phrase "without being explicitly programmed". This means that the knowledge incorporated in a learning machine is not explicitly hard coded or provided directly. Using this phrase machine learning is often defined similar to "field of study that gives computers the ability to learn without being explicitly programmed.".

But what does the term "learn" mean? An often cited definition given by Tom Mitchell in 1997 for learning of a computer program is:

> "A computer program is said to learn from experience **E** with respect to some class of tasks **T** and performance measure **P**, if its performance at tasks in **T**, as measured by **P**, improves with experience **E**."
>
> [Mitchell, 1997]

In general, there is a wide range of possible tasks **T**, experiences **E** and performance measures **P**. We will not give an overview here. Instead, the

following is intended to give an idea of these entities by taking the example of our assignment. For a more general introduction into machine learning, in particular deep learning, see [Goodfellow et al., 2016].

Task

Our assignment is to compare word embeddings and neural network structures for sentiment analysis of German political speeches by assigning one of three possible sentiment labels to sentences in political speeches. This already defines the task **T** for our assignment. This means we have a three class classification problem. Each sentence shall be assigned to one of the three classes by adding one of the class labels.

Performance Measures

To validate if a model learns a qualitative measure **P** is needed. For classification problems the *accuracy* is the most common performance measure. The accuracy is nothing else than the ratio of correctly labeled elements versus all with respect to a labeled dataset. But the accuracy itself is not sufficient for a thorough validation of a classification model. The most problematic case results from an imbalanced class distribution in the dataset used for validation. For example when we have a dataset with multiple classes and one class represents 90% of the entire dataset, then a model which classifies everything into this class has an accuracy of 90%. But this is a very useless model. This leads to other performance measures described below.

One of the main drawbacks of the accuracy is that the accuracy distinguishes not between classes. For a multi class problem it would be interesting to see how good the classifier behaves for different classes. For example for our classification problem with three classes C_0 (neutral), C_1 (negative) and C_2 (very negative) a classifier could be very good to distinguish between neutral C_0 and not neutral $C_1 \cup C_2$. On the other hand, when the question is which degree of negativity a sentence has, meaning the distinction between C_1 and C_2 the classifier could be very uncertain. To validate such behavior we need performance measures per class.

For a per class performance measure we call each element classified as an instance of our current class of interest a *positive* and all other elements a *negative*.

Next we denote a positive or negative prediction *true* if the class affiliation of the dataset coincide with the prediction, otherwise as *false*.

Therefore a positive which is actualy an instance of the class of interest is a *true positive* tp, otherwise its a *false positive* fp, meaning a instance of

any other class. On the other hand, a negative which belongs to any other class is a *true negative* tn, otherwise a *false negative* fn, see table 2.1.

		Labels/Predictions	
		Positive	Negative
Correctly classified	`True`	tp	tn
	`False`	fp	fn

Table 2.1: Performance measure categories.

With these categories the *precision* P is defined as

$$P = \frac{tp}{tp + fp}.$$

The precision is the ratio of correctly classified positive instances against all positive classified instances. For instance in web search engines the precision is very important. A human using a web search engine is interested in how relevant the results are and not if all results are found.

On the other hand, the *recall* R is defined as

$$R = \frac{tp}{tp + fn}.$$

This is the ratio of positive instances against all actual positive instances. For example consider a classifier for recognizing cancer. In this case it is more important to find as many persons which have cancer than to be very precise in the decision. A false positive can be corrected in further examination while leaving a patient untreated would most likely have severe consequences.

Finally we want to introduce the F_1 score which is a combination of recall and precision in a single value between 0 and 1 where 1 is a perfect result in both precision and recall given as

$$F_1 = 2\frac{P \cdot R}{P + R}.$$

What makes the F_1 interesting is that a high F_1 is only reached if both precision and recall are high.

Remark. For a two class problem only one class needs to be validated by these measures because precision, recall and F_1 for the first class determines precision, recall and F_1 for the other.

Experience

In a lot of machine learning applications the experience can be seen as the dataset used for training. In this case there are two major categories, *supervised* and *unsupervised* learning. Even though there is no rigorous definition they can be distinguished according the kind of data a model experiences in its training. Either the dataset consists of samples with a target value (label) or not.

In the supervised learning setup some kind of "teacher" determines what to expect from the model when looking at a specific data point. In our case we have a training set with labels where the labels are the expected outcome for a given sentence and the teacher could be seen as the human crowd coders who labeled the sentences in the dataset.

In contrast, unsupervised learning is when there are only data points but no target values. A common example for unsupervised learning is clustering. In NLP, word vectors are trained in an unsupervised setup.

Remark. A possible vantage point is to say there is no unsupervised learning. Meaning that by having a machine learning algorithm one must define some kind of performance measure on which the machine learning algorithm tries to improve on. By providing this performance measure, which tells if something is "good" or "bad", one introduces a supervision. Therefore there is no unsupervised learning. This is a philosophical question or a question about the definition of the terms, but I think it is worth mentioning. We will keep the term unsupervised in the absence of labels in the dataset.

Remark. Not all kind of machine learning trains on datasets. For example in *reinforcement learning* the learning algorithm interacts with an environment. The experience of the model results from a feed back after the model (agent) performs an action in the environment based on a current state in the environment.

To be able to learn from a dataset (experiencing the dataset) one has to find some kind of empirical connection between the dataset and the model. This means we want to find a connection between the data presented to the model and the parameters such that the performance measure validating the models performance improves.

The most common principle is the *maximum likelihood* principle. Consider a dataset $X = (x_i)_{i=1}^m$ drawn independent from a true but unknown probability distribution p_{data}. Let $p_{model}(x;\theta)$ be a parameterized family of probability distributions estimating the "true" probability p_{data}. But the probability p_{data} is unknown, therefore we take the empirical probability $\hat{p}_{data}$

defined by the given dataset X. Now the *likelihood* of a parameterization θ is defined as

$$\mathcal{L}(\theta) = p_{model}(X;\theta) \overset{iid}{=} \prod_{i=1}^{m} p_{model}(x_i;\theta).$$

The likelihood $\mathcal{L}$ is a measure of how likely it is that the given dataset X is drawn from the distribution $p_{model}(.;\theta)$. The *likelihood estimate* θ_{ML} is the most likely parameterization

$$\theta_{ML} = \arg\max_{\theta} \mathcal{L}(\theta).$$

The product of the probabilities in the definition of the likelihood is impractical to work with, especially for a numerical standpoint, leading to the *log-likelihood* as the logarithm of the likelihood

$$\log \mathcal{L}(\theta) = \log p_{model}(X;\theta) \overset{iid}{=} \sum_{i=1}^{m} \log p_{model}(x_i;\theta).$$

This does not change the likelihood estimate because the logarithm is continuous and strictly monotonically increasing. Additionally, one often normalizes by dividing though the dataset size $|X| = m$, this does not change the estimate as well but has the advantage to be interpreted as an expectation

$$\begin{aligned}\frac{1}{m}\log \mathcal{L}(\theta) &= \frac{1}{m}\sum_{i=1}^{m} \log p_{model}(x_i;\theta) \\ &= \sum_{i=1}^{m} \hat{p}_{data}(x_i) \log p_{model}(x_i;\theta) \\ &= \mathbb{E}_{x\sim\hat{p}_{data}} \log p_{model}(x;\theta).\end{aligned}$$

To apply the maximum likelihood to supervised training for data $X = (x_i)_{i=1}^m$ with labels $Y = (y_i)_{i=1}^m$ the probabilities are replaced by conditional probabilities $p_{model}(y|x;\theta)$ leading to the *conditional log-likelihood estimate* in order to model the prediction of y for given x

$$\begin{aligned}\frac{1}{m}\log \mathcal{L}(\theta) &= \frac{1}{m}\sum_{i=1}^{m} \log p_{model}(y_i|x_i;\theta) \\ &= \mathbb{E}_{x,y\sim\hat{p}_{data}} \log p_{model}(y|x;\theta).\end{aligned}$$

The next question is how to find a likelihood estimate for the parameters. Therefore we need to model the mentioned probability distribution $\hat{p}_{model}$ or at least define some objective to optimize.

2.1 Gradient Based Learning

Almost all neural network training algorithms are a variation of *gradient descent* (GD). This is a numerical optimization algorithm, where optimization refers to minimizing or maximizing a specific function $J(\theta)$ with respect to θ. In the context of machine learning, especially in deep learning, the function J is called *objective function*, *criterion*, *cost function*, *loss function* or *error function*[1]. The last three are almost always minimization problems. For the sake of simplicity we will only talk about minimization problems, but this restriction is without loss of generality because a maximization problem is equivalent to a minimization when replacing J with $-J$. A minimizer θ^* of $J : U \subseteq \mathbb{R}^n \to \mathbb{R}$ is given as

$$\theta^* = \arg\min_{\theta \in U} J(\theta).$$

A proper choice for the objective function J is the negative log-likelihood. Now we will give a more detailed description how the objective function for a likelihood estimate for the parameters can be constructed for our given task of a classification problem. Therefore consider a dataset $X = (x_i)_{i=1}^m$ with labels $Y = (y_i)_{i=1}^m$ where $y_i \in \{1, 2, ..., k\}$ for one of k classes. Then we choose as objective function

$$J(\theta) = -\mathbb{E}_{x,y \sim \hat{p}_{data}} \log p_{model}(y|x; \theta).$$

Usually the *cross-entropy* is used as a per-example loss for optimizing classification problems. Lets consider two probability distributions P, Q over a space Ω, then the cross-entropy of these distributions is given as

$$H(P, Q) = -\int_\Omega P \log Q \; d\lambda$$

and for a discrete space Ω this transforms into a sum

$$H(P, Q) = -\sum_{\omega \in \Omega} P(\omega) \log Q(\omega).$$

Broadly speaking the cross-entropy can be seen as a measure of how much two distributions differ, meaning when minimizing the cross-entropy of p_{model} to $\hat{p}_{data}$ we minimize the "difference" between the model distribution and empirical distribution. But how does this correspond to the likelihood?

[1] The function J is mostly be referred to as cost function and called J to be consistent with the rest of the text.

In our case the probability space Ω for the conditional distributions are the classes, meaning $\Omega = \{1, 2, ..., k\}$. Therefore one gets the cross-entropy of $\hat{p}_{data}(.|x)$ and $p_{model}(.|x;\theta)$ for a fixed x and θ as

$$H(\hat{p}_{data}(.|x), p_{model}(.|x;\theta)) = -\sum_{y=1}^{k} \hat{p}_{data}(y|x) \log p_{model}(y|x;\theta).$$

The index y of the sum iterates over the classes. But because $\hat{p}_{data}$ is an empirical distribution of the data, $\hat{p}_{data}$ is given as (corresponds to one-hot encoding)

$$\hat{p}_{data}(y_i = l|x_i) = \delta_{y_i,l} \qquad \forall i = 1, ..., m.$$

This is because the data is given, therefore the values are already determined.

So we can add zero in the cost function (negative log-likelihood) leading to

$$\begin{aligned} J(\theta) &= -\mathbb{E}_{x,y\sim\hat{p}_{data}} \log p_{model}(y|x;\theta) \\ &= -\frac{1}{m}\sum_{i=1}^{m} \log p_{model}(y_i|x_i;\theta) \\ &= -\frac{1}{m}\sum_{i=1}^{m}\sum_{y=1}^{k} \hat{p}_{data}(y|x_i) \log p_{model}(y|x_i;\theta) \\ &= \frac{1}{m}\sum_{i=1}^{m} H(\hat{p}_{data}(.|x_i), p_{model}(.|x_i;\theta)). \end{aligned}$$

This means we have as cost function J the mean of the per example categorical cross-entropy. Therefore defining a per example loss $\mathcal{L}$ through the categorical cross-entropy by

$$\mathcal{L}(y_i, f(x_i;\theta)) = H(\hat{p}_{data}(.|x_i), p(.|x_i;\theta))$$

we have a maximum Likelihood estimation for our model parameters. Meaning

$$\theta_{ML} = \theta^* = \arg\min_{\theta} J(\theta) = \arg\min_{\theta} \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(y_i, f(x_i;\theta))$$

is the minimization problem we want to solve to find a likelihood estimate for our classification model.

2.1.1 Gradient Descent (GD)

Gradient descent (GD) or *steepest descent* is an iterative algorithm to find (local) minima of J. First a starting point θ_0 is chosen, then for each step from a current point θ_t a next point θ_{t+1} is computed by moving into the direction of strongest decrease of J. The direction of strongest decrease of J is given by its negative gradient, so the next point is given as

$$\theta_{t+1} = \theta_t - \mu \nabla J(\theta_t).$$

The value $\mu > 0$ is called the *learning rate* or *step size*. There are multiple approaches to choose μ. The most simple and still very common approach is to set it to a small constant. Another would be to evaluate $J(\theta_t - \mu \nabla J(\theta_t))$ for different values of μ. This approach is called a *line search* because the function J is evaluated on different points along the line spanned from θ_t in direction of the gradient. Under certain conditions this even guaranties a convergence of the algorithm to a local minima, sadly in most machine learning applications this approach is not feasible due to high computational costs. Even worse, an actual gradient computation for training neural networks with this simple approach is way too expensive by computational means. Consider the negative log-likelihood cost for training a neural network on a dataset with m samples. This number of samples often ranges from thousands to billions of samples for training neural networks and with growing m the cost to compute the gradient grows. A common problem in machine learning that good generalization needs big datasets. This leads to the next algorithm.

2.1.2 Stochastic Gradient Descent (SGD)

To speed up the training process *stochastic gradient descent*[2] (SGD) approximates the gradient by taking only a subset of the entire dataset to compute the gradient of the cost function for one gradient step. Let us continue our example. To perform one gradient estimate one uniformly samples a small subset of samples $(x_i)_{i=1}^{m'}$ with associated labels $(y_i)_{i=1}^{m'}$ from the dataset. One such subset is called a *batch* with batch size m'. The batch size m' is typically a fixed constant, independent of the dataset size, in the range 1 to a few hundred. The approximated gradient g is computed as

$$g = \frac{1}{m'} \sum_{i=1}^{m'} \nabla_\theta \mathcal{L}(y_i, f(x_i; \theta)).$$

[2]Sometimes called *mini-batch stochastic gradient descent* in contrast to SGD where the batch size is set to 1. But in most of machine learning literature with SGD the mini-batch version is meant.

Usual implementation of stochastic gradient descent randomly group the dataset into batches of size m' and perform an approximated gradient update for each batch until all batches are exhausted. This is called an *epoch*, ensuring that each sample is seen exactly once in an epoch. This is done multiple times for training, meaning multiple epochs includeing new random creation of batches before each epoch.

2.1.3 Variations of Gradient Descent

This section addresses some problems the classic SGD can have as well as augmented algorithms which deal with the given problems.

First a proper learning rate can be hard to find, when the learning rate is too small the algorithm converges slowly but when the learning rate is too high the algorithm can become unstable. Additionally the learning is applied to the entire gradient which can result in unstable or slow convergence for some of the parameters, meaning that for a fast and stable convergence different learning rates for different parameters are needed. Another big problem, especially for neural networks, is the high dimensional parameter space for a highly non-convex cost function. The main problem is to be stuck in saddle points. Another problem can result from sparse data. For example when training word vectors infrequent words must be trained as well as very common words. Another example occurs in image recognition where special neurons detect special localized features followed by a max pooling. When a specific feature occurs rarely the gradient flows rarely over the weights of the neuron detecting this rare feature, roughly speaking this means that these weights have to learn faster than weights of more common features which are updated more often.

Momentum

Classic SGD struggles when moving through canyons. Imagine standing in a canyon, if you are not exactly in the middle of the canyon the direction of steepest descent is largely towards the river and not the flow direction of the river which carved the canyon. This results in a zigzag like movement through the canyon. Now consider (idealistic) a ball in the canyon. The ball starts rolling mostly towards the middle of the canyon while a tiny bit into the flow direction of the river. While he zigzags through the canyon the ball gains more and more momentum into the general direction through the canyon making the movement through the canyon faster as the ball goes on.

This momentum is introduced with an additional friction constant $0 < \gamma < 1$ for an adapted step direction leading to a new update schema for each

optimization step t

$$
\begin{aligned}
g_t &= \nabla J(\theta), \\
m_t &= \gamma g_{t-1} + g_t, \\
\theta &= \theta - \mu m_t.
\end{aligned}
$$

From the point of a single parameter, this means that when the parameter is consecutively updated into one direction its updates get bigger and bigger while on the other hand when the parameter updates alternate they get changed less and less.

Nesterov's Accelerated Gradient (NAG)

Again consider a ball rolling through an (idealistic) canyon, the classic momentum based approach for acceleration the motion through the canyon works good when the canyon is straightforward, but when the canyon makes a turn the ball will roll up hill in the curve until it "realizes" that the canyon made a turn. *Nesterov's accelerated Gradient* (NAG) tends to mitigate this effect by guessing the new position based on the balls velocity and corrects the step if a similar situation occurs. This is accomplished by the adapted update rule

$$
\begin{aligned}
g_t &= \nabla J(\theta - \mu m_{t-1}), \\
m_t &= \gamma g_{t-1} + g_t, \\
\theta &= \theta - \mu m_t.
\end{aligned}
$$

So the only difference between classic momentum and NAG is where the gradient is evaluated.

AdaGrad

In [Duchi et al., 2011] an *adaptive subgradient descent* (AdaGrad) algorithm is proposed which adapts the learning rate for each parameter separately depending on all previous gradients. This allows to slow down learning for parameters that already have changed a lot while accelerating learning for parameters that only changed slightly. The intention behind that is to stabilize learning between common and rare features.

The actual update rule is given by

$$
\begin{aligned}
g_t &= \nabla J(\theta), \\
n_t &= n_{t-1} + g_t \odot g_t, \\
\theta &= \theta - \frac{\mu}{\sqrt{n_t} + \varepsilon} \odot g_t.
\end{aligned}
$$

The term $g_t \odot g_t$ is the element wise squared gradient added in each step to the vector n_t accumulating all squared gradients. Now $\frac{\mu}{\sqrt{n_t}+\varepsilon}$ is the vector valued learning rate containing in each component the learning rate for one parameter applied element wise to the gradient for updating parameters.

A great result of the adaptive learning rate is that the learning rate μ can be set to a default value without the need to search for a good learning rate. But at the same time the accumulation of the squared gradients result in a monotonic decay of the learning rate. Therefore training basically stops at some point.

RMSProb

The *root mean square propagation* (RMSProb) algorithm is not officially published but it still is an often used algorithm similar to AdaDelta proposed in [Zeiler, 2012]. Both algorithms adapt AdaGrad with the intention to solve the problem of vanishing updates because of too small learning rates. The simple idea is not to sum all previous gradients, but instead keep a moving average for computing the adaptive learning rate. The RMSProb update rule is defined as

$$\begin{aligned} g_t &= \nabla J(\theta), \\ n_t &= \nu n_{t-1} + (1-\nu) g_t \odot g_t, \\ \theta &= \theta - \frac{\mu}{\sqrt{n_t}+\varepsilon} \odot g_t. \end{aligned}$$

with a typical value for $\nu = 0.9$.

Adam

The *adaptive moment estimation* (Adam) algorithm combines momentum with adaptive learning rates. This algorithm is basically RMSProb with momentum and a bias correction. The bias correction is for the early phase of the algorithm where initialization of momentum and squared gradient

accumulation to zero can cause instabilities.

$$
\begin{aligned}
g_t &= \nabla J(\theta), \\
m_t &= \gamma g_{t-1} + (1-\gamma) g_t, \\
\hat{m}_t &= \frac{1}{1-\gamma^t} m_t, \\
n_t &= \nu n_{t-1} + (1-\nu) g_t \odot g_t, \\
\hat{n}_t &= \frac{1}{1-\nu^t} n_t, \\
\theta &= \theta - \frac{\mu}{\sqrt{\hat{n}_t} + \varepsilon} \odot \hat{m}_t.
\end{aligned}
$$

Here m_t is the momentum and n_t the addaptive learning rate, the $\hat{m}_t, \hat{n}_t$ are the corrected momentum and learning rate which get less and less corrected as training proceeds.

2.2 Model Selection

2.2.1 Model Capacity

One of the biggest challenges in machine learning is *generalization*. This means that the models must not only perform well on the training set but also on unseen data (good performance **P** for new experiences **E** during execution of task **T**).

In most machine learning applications one has a dataset to train the model on. In this case one can compute an error measure called *training error* which is the error the trained model has on the given dataset. When only minimizing the training error the setup is just an optimization problem, but in machine learning one is also interested in the expected error the model would have for unseen data. This error is called the *test error*, *validation error* or *generalization error*. From a pure mathematical standpoint the test and training error for an untrained model are the same, then when training the model on a test set drawn from a specific but unknown distribution the training error gets optimized, additionally the test error will be higher than the training error but should be as close to the training error as possible to get a good generalization. Therefore, in practice, to find a good model we are interested in two points

1. to minimize the training error and
2. minimizing the gap between test error and training error.

These two points represent two challenges in machine learning namely *underfitting* and *overfitting*. Underfitting occurs when a model is not powerful enough to fit to the given training dataset to get a low training error. On the other hand overfitting is when the gap between the training and test error is high, meaning the model could fit well to the given training dataset but does not perform well for unseen data.

One way to influence overfitting and underfitting is to alter a models *capacity*. Basically the capacity of a model is its ability to fit different functions. In other words, the higher a models capacity the more complex relations can be modeled. A problem with the term capacity is that it is often casually used without a clear definition.

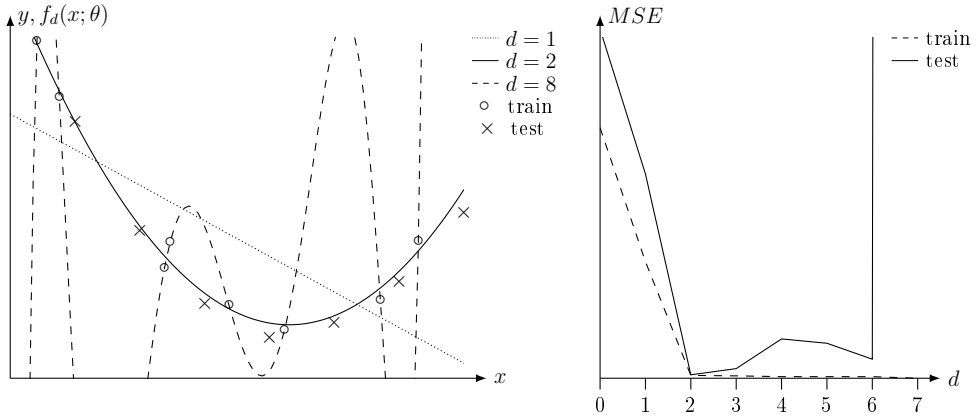

Figure 2.1: Example of polynomial regression with degree d. Left: Plot of 3 fitted polynomials, linear model $d = 1$, quadratic $d = 2$ and polynom of degree $d = 8$. Right: Training and test MSE over polynom degree d.

One of the well known measures of a models capacity is the *Vapnik-Chervonenkis dimension* (VC dimension) for binary classifiers. The VC dimension is basically the maximum dataset size which can be arbitrarily labeled by a binary classifier. More precisely let f be a binary classifier, its VC dimension is the size of the biggest possible configuration of data points X such that f is able to label all data points in X arbitrarily (note that X can be chosen adequately). That means that f can learn all $2^{|X|}$ possible label combinations of an appropriate dataset X. For example the VC dimension of the Rosenblatt Perceptron (see: section 3.1) for 2 dimensional data is 3. Because for a dataset $X = \{x_1, x_2, x_3\} \subset \mathbb{R}^2$ (not on a line) there exists a parameterization of the perceptron such that all 2^3 splits are correctly classified by the perceptron, but for any bigger dataset X, meaning $|X| > 3$, there does not exist a parameterization for each possible $2^{|X|}$ splits into two

groups as illustrated in figure 2.2. This is also known as the XOR problem resulting from the fact that the perceptron is a linear classifier but for any four different points there exists a labeling such that the two groups are not linear separable, see figure 2.2 for such a configuration.

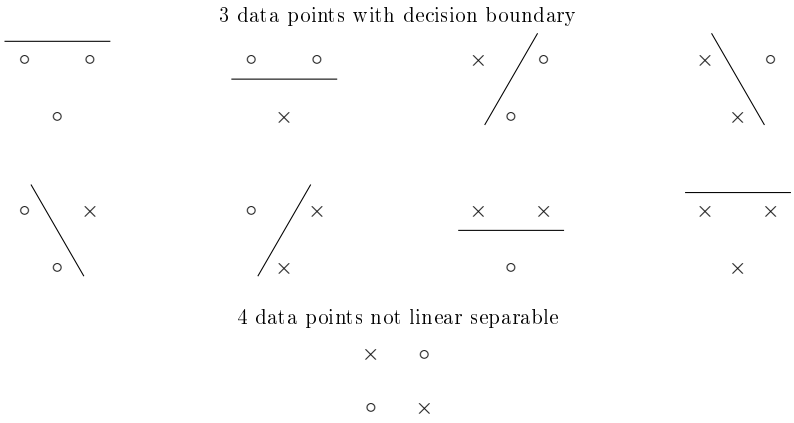

Figure 2.2: Top: 3 data points with all possible different labels and a linear decision boundary. Bottom: Illustration of the XOR problem, not linear separable.

However, in deep learning the VC dimension is not really used like any other well defined measure of capacity. In practice the only really used measure of the capacity of a deep learning model is the number of parameters. This is because there is a lack of well understood and sufficient capacity measures for the wide range of different deep learning models. For example when the activation function of the perceptron is changed to $x \mapsto H(\sin(x))$ one gets a VC dimension of ∞ (see A.5). On the other hand it is infeasible to actually determine the VC dimension of most deep neural networks not only because the models are complex and can have high varying capacities by "small" changes to the model but also that the actual capacity of a machine learning model is also dependent on the capability of the optimization algorithm used to train a neural network. This means that a capacity measure for deep learning models must not only consider the model itself but also the optimization algorithm used to train the model.

But how to determine if a model is underfitting or overfitting? As mentioned we do not have a thorough theory about the capacity of a deep learning model, but even if we would have an explicit capacity measure the capacity of the model is only a good hint for model selection but not what we are interested in. The problem is that depending on the task, low capacity models can still work very well. They can deliver low training and test errors as well as high capacity models. Additionally models with high capacity but

not well suited for a specific task can perform way worse than low capacity models specialized to the task on hand.

Remark. To address the problem of overfitting in deep learning regularization methods are used. These regularization methods range from L_2 *weight decay*, *max-norm regularization* to *dropout*. Further methods include adding noise to the input data or the weights during training. Basically there are two ideas, first the network has additional penalties or restrictions during training or second the experience of the network during training is manipulated such that the model will not experience the same over and over again. Well this is very broadly speaking. For more information see [Goodfellow et al., 2016; Srivastava et al., 2014].

2.2.2 Cross Validation (CV)

To divide the given dataset into a training set and a test set can have unwanted side effects, especially when the given dataset is "small". On one hand when from a small dataset a portion is split of and not be used for training the model may not be trained well because the given training data is too small while on the other hand the resulting "small" test set will probably be too small to get a statistically certain result. The goal of *cross validation* is to get a more accurate estimate of the test error. The idea is to randomly split the given data into training and test sets, train the model (from scratch) and evaluate the test error on the remaining test set (out-of-sample testing). When doing this multiple times the mean of all test errors is an estimate of the expected test error. Because when evaluating the test error multiple times the test set for each trial can be way smaller leaving more samples for the training set needed to have a well trained model. Furthermore with small test sets a statistical uncertainty is given when evaluating the test error, this can result in false test error estimates because of a possible unlucky choice of the test set. This problem is mitigated by taking multiple different test sets.

The most common variation of this approach is the *k-fold cross validation*. There the dataset is split randomly into k distinct subsets. Then each of the k subsets is chosen as the test set while the union of the remaining sets are the training set. Then the final test error is the mean of all k test errors.

2.2.3 Grid Search (GS)

In machine learning it is often not obvious which model is an appropriate model. Meaning when designing a model there are possible model types often with different hyperparameters to chose from. For example for a MLP (see:

section 3.2) the number of layers, the size of one layer (number of neurons in the layer), the kind of activation function and so on are all hyperparameters. To find an appropriate combination of hyperparameters suited for the given task a *grid search* (GS) can be used. A grid search is basically to try all possible hyperparameter combinations in combination with model validation metrics, then taking the model with the best performance with respect to the validation metrics. A common setup is to use a grid search with a cross validation for each model.

Chapter 3

Neural Networks

The idea behind Artificial Neural Networks (ANNs) is inspired by the brain. Basically, a brain is nothing else than a network of neurons where neurons are connected via synapses which are electrical (or chemical) conductors with a specific strength. By the different strengths of the synapses the electrical flow from neuron to neuron is controlled. When a neuron, connected to multiple others, receives electrical signals through its inbound synapses this neuron will fire when a certain threshold is reached as combination of all input signals. This results in a signal discharged into this neurons outbound synapses which transports this signal depending on the synapses strength. The stronger the synapses the stronger the signal received by the connected neurons making it more likely that the connected neurons will fire too. Broadly speaking, through the connectivity, the strengths of the synapses and the threshold of neurons the electrical flow through the brain is controlled. Building on this simple view of the brain, learning and memorizing is strengthening synapses which are used often while neglected synapses decay.

Inspired by the brain ANNs are models of this oversimplified version. Different types of ANNs will be introduced in the following sections, starting with Rosenblatt's Perceptron as the model of a single neuron, followed by Multi Layer Perceptrons (MLPs) and more complex models like Convolutional Neural Networks (CNNs) as well as various versions of Recurrent Neural Networks (RNNs).

Remark. Actually learning with SGD does not match the described learning in the oversimplified brain model. The SGD optimization tweaks the parameters in such a way that the model would make less errors for the current experience in contrast to strengthening the weights involved in the current experience while decaying the not (or weak) involved weights.

Remark. As the task on hand is a multi class classification problem we will talk about ANNs primarily with the intend to build a classifier. But the exact same ANNs can be used for regression problems as well. There are just a few common practices in which the two concepts differ, mostly in the common output activation functions and the used loss function for training. ANNs can also be used as generative models, these are models trained to produce "new data" by training an ANN to represent a probability distribution. For example a decoder which generates a summary of a text after an encoder "encoded" the text. In other words, the job of the decoder is to create a probable summary of the text based on a representation.

3.1 Perceptron, the Basic Building Block

The history of ANNs started with a simple model of a single neuron, known as *Rosenblatt's Perceptron*. The neuron is modeled as the weighted sum of its inputs plus a bias followed by a Heaviside step function. The weights correspond to a synapses strength and the bias models the neurons threshold. Finally, the Heaviside function lets the neuron either fire or stay quiet.

In more detail, let H be the Heaviside function and denote with $f(x; w, b)$ the model with weights $w \in \mathbb{R}^n$ and bias $b \in \mathbb{R}$. As model input the vector $x \in \mathbb{R}^n$ describes the signals delivered by the neurons inbound synapses. Each component x_i of the input corresponds to the signal of the i'th synapses with a weight (strength) of w_i.

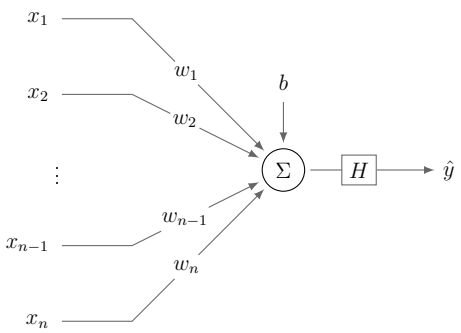

Figure 3.1: Architecture of Rosenblatt's Perceptron.

Mathematically, the perceptron is described as

$$f(x; w, b) = H(w^T x + b).$$

And it is nothing else than a linear binary classifier.

3.1.1 Activation Functions

The Rosenblatt Perceptron, as a model of a single neuron, is not suitable as a single unit in an actual ANN. The problem is that ANNs are trained with an SGD algorithm. These algorithms compute gradients for learning, but the derivative of the Heaviside function[1] is zero and therefore the gradient too, which means no learning. To resolve this issue, the Heaviside function is replaced by a piecewise differentiable function denoted as ϕ. These functions are called *activation functions*. This leads to the altered model of a single neuron as

$$f(x; w, b) = \phi(w^T x + b).$$

Additionally there is a set of well known activation functions, better suited for specific tasks, than the Heaviside function. Although, a theoretical background of the behavior of ANNs using specific activation functions is still not well understood and is an active field of research as part of high dimensional non convex numerical optimization. But based on experience, educated guesses can be made for an activation function to be used in an ANN. These educated guesses, where ReLU is a good fallback, are a good starting point to implement an ANN for a specific task, but there should be an experimental survey to find a suitable activation function for the specific task.

In the following a few of the most common activation functions are presented.

Rectified Linear Units: The *Rectified Linear Units* defined as

$$\mathrm{ReLU}(x) = \max(0, x)$$

is the most simple activation function and behaves well in a lot of applications. Due to its linearity the gradient optimization algorithms for learning like the back propagation algorithm is very efficient and encounters almost no problems. A special property, this almost linear units have, is that the derivative of the ReLU and similar functions does not introduce second order terms, which is helpful for gradient descent optimization algorithms.

There are a few variations, starting with the *leaky* ReLU, which has a small positive slope α that regulates a leak in the negative domain of the activation.

$$\mathrm{ReLU}_\alpha(x) = \max(0, x) + \alpha \min(x, 0).$$

[1] Problematic points, like the 0 for the Heaviside function, are typically ignored and set to an arbitrary value, for example to the one sided derivative, in the context of numerical optimization.

The leaky ReLU has one advantage in comparison to the ordinary ReLU, namely that the ReLU could be struck in the training process when almost all samples result in negative values for this unit. Then there is no activation and therefore no updates of weights leading to "dead" units. This behavior can be avoided with the leaky ReLU.

Furthermore the slope parameter could be learned too and not be fixed, this version is known as the *parametric* ReLU or PReLU.

Another variation is the *absolute value rectification* function that fixes the value $\alpha = -1$ resulting in $x \mapsto |x|$. As an example use case consider object recognition, where a specific object should be recognized regardless of a reverse of colors.

Sigmoid Functions: Sigmoid functions are S-shaped functions. The two most common used sigmoid functions as activation functions are the *logistic sigmoid* σ and the hyperbolic tangent.

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$
$$\tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$$

The sigmoid functions are harder to train, mostly because they saturate for big values, making it difficult for gradient based learning algorithms. This is the reason why the use of these activation functions is discouraged for simple feed forward ANNs. But there are special ANNs like gated RNNs that use the logistic sigmoid for neurons that control a gating mechanism or as output activation using its range of $(0, 1)$. In others than such special cases the hyperbolic tangent should be preferred, mostly because of its close to linear behavior around 0.

A common practice when using sigmoid functions is to approximate them with partial linear functions. The main advantage is that these approximations are faster to compute while experience shows that in appropriate places the performance of the models is not compromised. For example the *hard sigmoid* σ_h defined as

$$\sigma_h(x) = \begin{cases} 0, & \text{if } x < -\frac{5}{2} \\ 1, & \text{if } x > \frac{5}{2} \\ \frac{x}{5} + \frac{1}{2}, & \text{else} \end{cases}$$

is often used in gated RNNs as activation function for the gates.

Softmax: The softmax, in contrast to other activation functions presented here, involves more than one unit. This is an activation that involves all units in one layer.

$$(\text{softmax}(x))_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

The intention is to create a discrete probability distribution as output over multiple neurons. It is mostly used as an activation for the output layer of a classification model. As an example consider our task of a three class classification problem. All models that will be compared later have three output neurons, each neuron representing one class and as an activation the softmax is used. Now the values of the three output neurons for a given input are interpreted as the probability that the input belongs to a neurons class. Now the classification is done by taking the class with the highest probability. In addition the probability of this class can be seen as the certainty of the model that the input belongs to this class.

Further Activation Functions: There are countless other activation functions and a wide range of differentiable functions are well suited. For example the *softplus*

$$\text{softplus}(x) = \log(1 + \exp(x))$$

or *radial basis functions* or even trigonometric functions can be used. But the most common ones are listed above and for most tasks these functions perform as well as specialized activation functions.

3.2 Multi Layer Perceptron (MLP)

Like a single neuron the Rosenblatt Perceptron itself is only a linear classifier and not very powerful. The real strength of ANNs comes from connecting multiple perceptrons as nodes of a real network. This leads to *Multi Layer Perceptrons* (MLPs) which consist of multiple neurons (perceptrons) aside in multiple layers, hence the name. The concept is seen in figure 3.2 where each gray node in the network represents a single perceptron (neuron) and each arrow a directed connection (synapses).

The first layer consisting of neurons that receive the models input is called the *input layer*, the last layer is denoted as *output layer*. All other layers are *hidden layers* because the neurons in the hidden layers do not interact with the environment, only with other neurons in the network.

We started with a formal description by enumerating the layers starting by 1 from the first hidden layer. For example the MLP in figure 3.2 has the

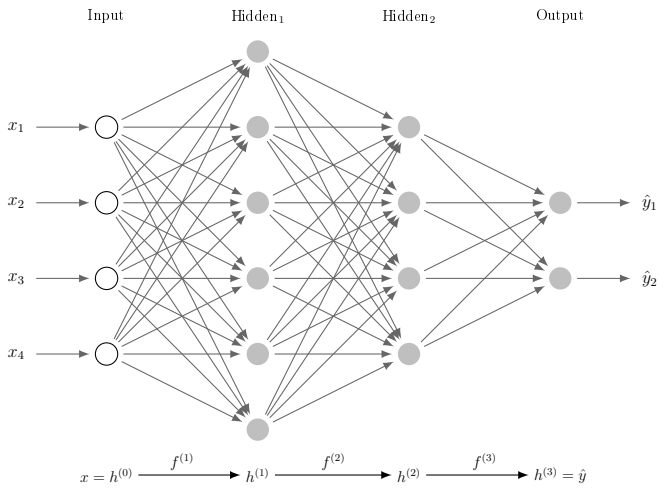

Figure 3.2: Example Architecture of a 3 Layer MLP.

layers $1, 2$ as hidden layers and 3 as its output layer. Each layer consists of n_l perceptrons where each perceptron has n_{l-1} inputs from the previous layer. These $n_l n_{l-1}$ weights are represented in matrix form and denoted as $W^{(l)} \in \mathbb{R}^{n_l \times n_{l-1}}$. Furthermore there are the bias each perceptron has which will be called $b^{(l)} \in \mathbb{R}^{n_l}$ which stores the bias of each neuron. Finally there was the activation function applied to a perceptrons output. It is common to have the same activation function for all neurons in one layer, therefore let $\phi^{(l)}$ be an activation function from section 3.1.1 for layer number l. Meaning that all neurons in the layer l use $\phi^{(l)}$ as its activation function. With this a single layer of an MLP can be described through the vector valued function $f^{(l)}(h^{(l-1)}; W^{(l)}, b^{(l)})$ where $h^{(l-1)}$ is the output of the previous layer.

$$f^{(l)}(h^{(l-1)}; W^{(l)}, b^{(l)}) = \phi^{(l)}(W^{(l)} h^{(l-1)} + b^{(l)})$$

Please note that the activation function is applied element wise as activation for each single perceptron where each element is the output of one perceptron.

For the sake of a more economic notation we will write $f^{(l)}$ as an abbreviation for the parameterized function $f^{(l)}(\,.\,; W^{(l)}, b^{(l)})$. To finally get a formulation of an MLP the functions $f^{(l)}$ describing a single layer need to be concatenated as

$$f = f^{(N)} \circ f^{(N-1)} \circ \ldots \circ f^{(1)}$$

to get a hole N layer MLP f with weights $W^{(l)}$ and biases $b^{(l)}$ as well as activation functions $\phi^{(l)}$ with an enumeration of the layers by $l = 1, \ldots, N$.

3.3 Convolutional Neural Network (CNN)

Convolutional Neural Networks (CNNs) are specialized ANNs originally developed for image processing. Their architecture is strongly correlated to the brain region V1 also known as the *primary visual cortex*. The V1 is the first region of the brain that performs real processing of image data. In the eye, where light is captured by the retina, the neurons in the retina perform a simple preprocessing and passe the data through the optic nerve into a region of the brain, which basically relays the signals further to V1 at the back of the brain. In our very simplified view, V1 is built up from multiple layers where each layer has a two dimensional spacial structure consisting of two kinds of neurons. The *simple neurons* and the *complex neurons*. The two dimensional structure enables V1 to localize features in an image to a position. For example when light is captured by the retina at the left upper part only the corresponding neurons in V1 activate. The next part are the simple cells which are localized and extract different patterns, meaning that different simple cells responsible for a specific region of an image activate for special local patterns. Consider to see a blank canvas with one dot, then the (localized) simple cells concerned with dots activate. Finally there are the complex cells, they behave similar to the simple cells except that they are invariant to small positional shifts and/or changes in the patterns.

Remark. The same concepts can be applied to sequential data with 1D convolution where the grid like data is represented by a feature axis and a time axis (sequence of features, for example a sequence of word vectors). The difference to 2D convolution described below is that the convolution is applied only to the time axis, therefore 1D CNNs.

Now CNNs mimic this oversimplified V1 architecture by using the following three concepts for a single CNN layer.

Sparse Interaction: The idea of *sparse interaction* (also referred to as *sparse connectivity* or *sparse weights*) is a localization of dependency. This means in a single layer one output neuron is only connected to a few input neurons.

Parameter Sharing: In contrast to the classic MLP the activation for a output neuron uses the same kernel for the calculation for its input for different output neurons. This means that in convolutional networks the activation of different neurons use the same kernel (or at least a very small number of kernels) to compute their input and not a matrix vector multiplication where each row (or column) is independently responsible for a neurons

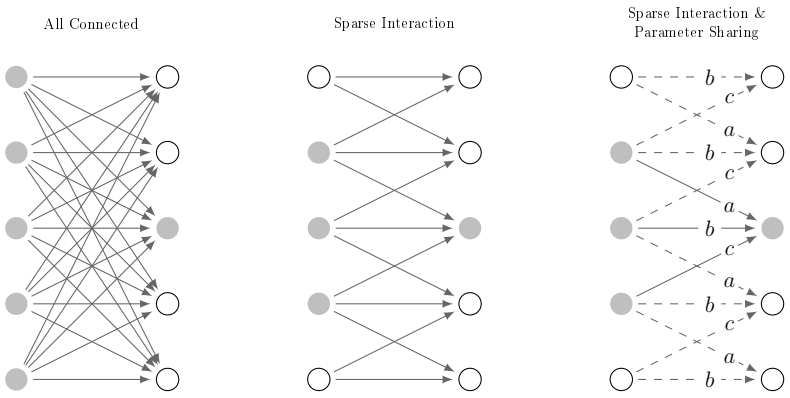

Figure 3.3: Left: A complete connected input to output layer (like in MLPs). Middle: A sparse connected layer. Right: A convolutional layer with a 3 element kernel with parameters a, b and c. Those are all parameters for this layer (except the bias).

input. This results in a significant reduction of free parameters. For an example of a network layer structure representation using parameter sharing see figure 3.3.

Equivariant Representation: A desired property called *equivariants* to translation means that shifting the input does not result in an entirely different output. It is only shifted equally.

Structure of a CNN Layer

Let us begin with modeling an actual CNN layer. A typical convolutional layer consists of 3 stages, first a convolution followed by the detector stage (applying the activation function) and finally a pooling stage.

A convolutional layer basically performs a convolution to the input data like the discrete convolution[2] described in section A.3. Second, the kernel in the convolution is usually much smaller than the input. Therefore a convolutional layer only uses a subset of the input data for the calculation of the output features of a node. For a visualization see figure 3.4. Furthermore the input to a convolutional layer is not just a grid of single values. Rather, it is a grid of vector valued inputs where the elements of these vectors are called *features* or also referred to as *channels*. The output is as well a grid of vectors, each feature representing a different localized property. To

[2]Actually, most frameworks do not implement a mathematical convolution but it is a very similar operation.

accomplish this affine transformation from the multi dimensional input to the multi dimensional output, tensors are used (consider the bias which is added after the kernel was introduced to the data, therefore the affine transformation). As an example consider a convolutional layer for image data. The image data forms a grid of x, y pixel positions, each pixel consists of 3 values, the red, green and blue intensity. A convolution on this image data could detect differen localized features like edges, dots or colors. This can be accomplish with a 4D[3] kernel. There are two positional dimensions and a channel dimension in the input and one in the output.

Now assume we have a 4D kernel tensor $\mathbf{K}$ with the channel indices i, j with i on the output, j on the input channel and the positional offsets in the rows and columns k, l relative to the current output position referring to the input elements (e.g. when calculating the output element with position (x, y) the indices k, l refer to the input element with position $(x + k, y + l)$). With this notation the tensor elements are $\mathbf{K}_{i,j,k,l}$. Assume our input is given as $\mathbf{X}$ with channel index i and positional indices x, y meaning the elements of the input data are $\mathbf{X}_{j,x,y}$ and the output $\mathbf{Y}_{i,x,y}$ has the same format. Then the discrete convolution would be

$$\mathbf{Y}_{i,x,y} = \sum_{j,k,l} X_{j,x+k,y+l}\mathbf{K}_{i,j,k,l}.$$

For the sake of simplicity the indices k, l are symmetric around 0.

The problem is that the upper convolution is actually not well defined because the relative indices k, l result in a reference of undefined input data, for example $X_{j,-1,-1}$. This leads to a special feature called *implicit zero padding*. That means that the input data is (implicitly) augmented with 0 values outside the boundaries of the data. There are different concepts for this problem and for more information see [Goodfellow et al., 2016].

3.3.1 Pooling

The pooling step is performed after the detection stage and it basically combines a local neighborhood into a summary. The most common is the *max-pooling* which gives the maximum activation of its neighborhood. Other pooling types are *average-pooling*, L_2 pooling or weighted averages with respect to the center of the neighborhood.

There are three hyperparameters for such a pooling step. Its size n_f, the stride n_s and a padding n_p. The size defines the size of the neighborhood, for

[3] Actual implementations usually work in batches, this introduces another dimension over the examples in the batch.

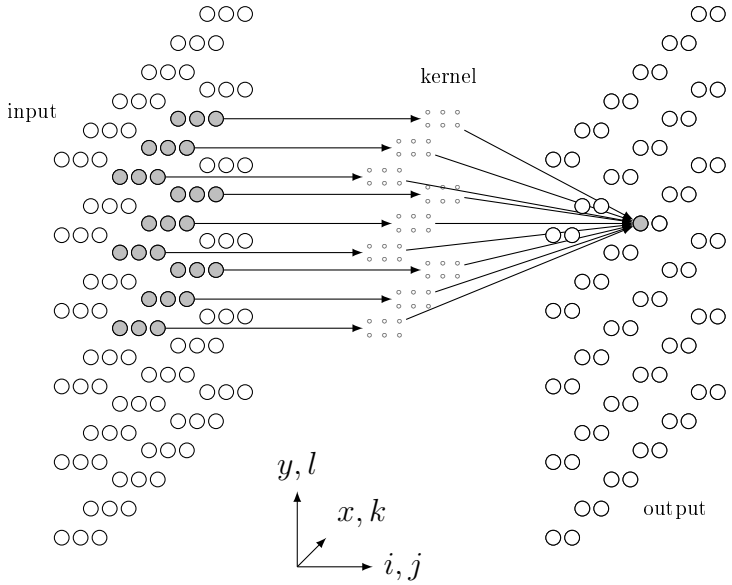

Figure 3.4: Calculation of a single output feature in a convolutional layer with a 3×3 kernel for a 3 feature input and 2 feature output, this means that the kernel is represented by a $2 \times 3 \times 3 \times 3$ tensor. (Inspired by image data, 2D convolution)

example a size of n_f gives a $n_f \times n_f$ neighborhood involved. Next the stride defines the shift from one neighborhood to the next, this means that a stride of n_s shifts the neighborhood for the next pooling output by n_s. Finally the padding sets an input border which (implicitly) augments the input by the padding size (typically with 0 values). Let us consider a few examples: First a common pooling configuration with distinct neighborhoods, let $n_f = n_s = 2$ and no padding $n_p = 0$. Then the pooling operation on a 8×8 grid gives a 4×4 output as illustrated left in figure 3.5. Another example visualized right in figure 3.5 is with a stride of $n_s = 1$, there the neighborhoods overlap.

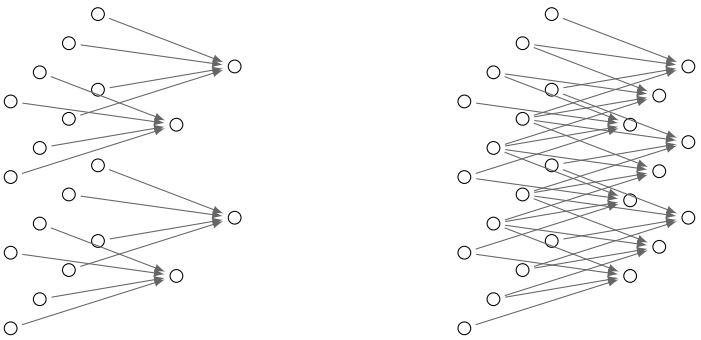

Figure 3.5: Left: Pooling operation of size and stride $n_f = n_s = 2$ and no padding. Right: Size $n_f = 2$ and stride $n_s = 1$ with no padding.

Convolution Layer Form

Now assume we have a convolutional layer in a convolutional network at an network depth l. The closed form on the convolutional layer output $h^{(l)}$ by an input denoted by $h^{(l-1)}$ with a kernel $K^{(l)}$, bias $b^{(l)}$, an activation function $\phi^{(l)}$ and a pooling operation $p^{(l)}$ would be

$$f^{(l)}(h^{(l-1)}; K^{(l)}, b^{(l)}) = p^{(l)}(\phi^{(l)}(K^{(l)} * h^{(l-1)} + b^{(l)})).$$

The activation functions $\phi^{(l)}$ used in convolutional networks are the same as in MLPs described in section 3.1.1, for instance the rectified linear unit or a sigmoid function.

3.4 Recurrent Neuronal Network (RNN)

Recurrent Neural Networks (RNNs) are a family of ANNs specialized for processing sequential data. The main working principle is inspired by cyclic

connections between neurons in the brain to keep track of previous information. RNNs incorporate this cyclic connections with internal back references in a recurrent manner with respect to sequential input.

Let us assume the sequential data processed by an RNN is temporal, meaning the RNN gets a sequence $(x_t)_{t=1}^{\tau}$ where t is a time index, as input. The input t is not necessarily a time index but it matches our task where an RNN gets a sequence of word vectors representing a sentence and t is interpreted as a time step index of the words. Furthermore a time series justifies terms like "past" and "future" which simplifies explanation. To motivate the design of RNNs let us consider such a sentence $(x_t)_{t=1}^{\tau}$ as a sequence of word vectors.

RNNs are designed to handle such sequences properly which the previous presented feed forward models, namely MLPs and CNNs, are not capable of. There are a few reasons why. First a sentence does not have a fixed length which is especially a problem for MLPs because they assume a fixed size input. To handle this limitation one could build the average over the word vectors resulting in a sentence summary of fixed size, but totally neglecting the order of the words in the sentence. This leads to the second design principle, information of the sequence order must be maintained. Third, specific patterns in the sequence need to be recognized, independent of their position. The second and third principle are already incorporated by CNNs by using parameter sharing, but only local, meaning they cannot keep track of long term dependencies leading to the fourth principle, tracking of long term dependencies.

Let us start with building a vanilla RNN for a basic illustration of the concept. Starting with the RNNs input sequences denoted as $(x_t)_{t=1}^{\tau}$ the RNN computes an output sequence $(\hat{y}_t)_{t=1}^{\tau}$ while maintaining an internal state $(h_t)_{t=0}^{\tau}$. For each time step t the RNN takes the input x_t at time t and the internal state of the previous step. By passing on the internal state from the previous time step the RNN is able to keep track of information from the past. Additionally there are three weight matrices, one for the input denoted U, another for the internal state called W and one for transforming the output denoted as V. Because of the weight matrix for the internal state the RNN is also capable of learning which information shall be passed on, which is a very important feature. Furthermore, it is important that for all time steps the same matrices U, V and W are used to incorporate the parameter sharing to be able to detect the same patterns at different positions in the sequence. Mathematically speaking, our vanilla RNN layer is, for time steps

$t = 1, ..., \tau$, recursively defined as

$$h_t = \tanh(Wh_{t-1} + Ux_t),$$
$$\hat{y}_t = Vh_t.$$

Note that a hyperbolic tangent was used as activation function for the internal state, this is just the most common activation for such simple RNNs. Also the initial internal state h_0 will just be set to 0 for our vanilla RNN. The structure is visualized in figure 3.6.

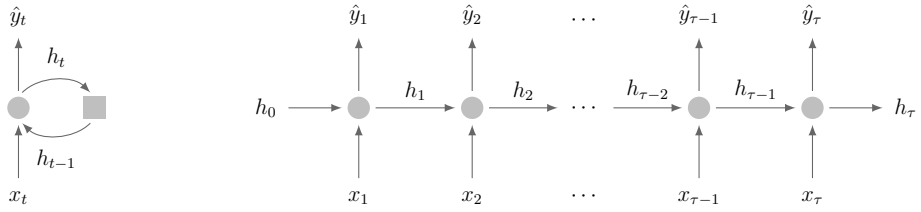

Figure 3.6: Information flow in our vanilla RNN. For each time step t the network processes the current input x_t as well as h_{t-1} and computes an output $\hat{y}_t$ while updating the internal state to h_t that is passed forward in time. Left: Circuit diagram. The rectangle represents a time delay. Right: The same network "unfolded" with an input sequence of length τ.

The next step is to adapt our vanilla RNN to specific tasks, as one can see our initially constructed vanilla RNN produces an output for each time step. This is not always a useful behavior. There are three different concepts for RNNs. First, the *Sequence-to-Sequence* structure like our vanilla RNN, second the *Sequence-to-One* structure which can be easily created by neglecting all the outputs except the last one or all outputs and then take the last internal state as the output. The third variant is the *One-to-Sequence* architecture where the RNN does not get an input sequence. The input to the network could be the initial internal state and the network creates a sequence of outputs. For a comparison see figure 3.7.

In our case of a classification problem the Sequence-to-Sequence structure is not useful as a last layer but will be used as previous layers for a stacked RNN where the output sequence will be the input of the next RNN and a final Sequence-to-One recurrent layer. The third version of a One-to-Sequence structure has no use for our classification problem, an example use case would be in an Encoder Decoder network for translation. There are a lot of different variations and for more details of basic construction principles of RNNs see [Goodfellow et al., 2016] or from an NLP viewpoint see [Goldberg, 2015].

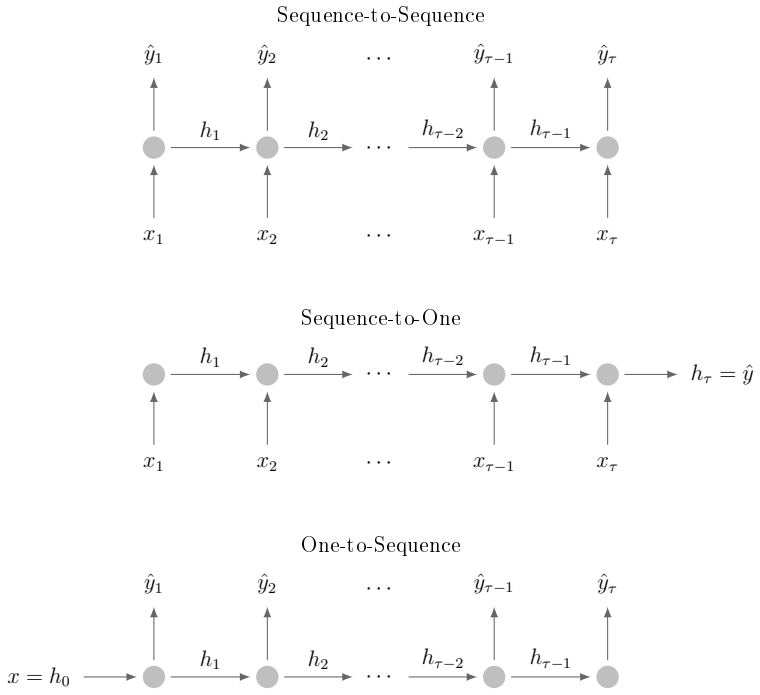

Figure 3.7: Comparison of different RNN structures. Top: A Sequence-to-Sequence structure where an input sequence $(x_t)_{t=1}^{\tau}$ is transformed into an output sequence $(\hat{y}_t)_{t=1}^{\tau}$, Middle: Sequence-to-One structure, processes an input sequence $(x_t)_{t=1}^{\tau}$ and summarizes the entire sequence into a fixed size output $\hat{y}$, Bottom: One-to-Sequence structure which gets a single input x and creates an output sequence $(\hat{y}_t)_{t=1}^{\tau}$.

Before diving into specific RNN structures the *vanishing-* and *exploding gradient problem* needs to be discussed. These problems result from the Back Propagation Through Time (BPTT) algorithm, described in 3.6, which is used to compute gradients for weight updates in the training process of an RNN. As the name suggests this is a back propagation algorithm closely related to the classic back propagation algorithm used in the training of feed forward neural networks but with an additional propagation through time inside an RNN layer. For our vanilla RNN the propagation through time of the gradients involves a matrix multiplication with the weight matrix W^T for each step back in time (see section 3.6). The longer the time series the more matrix multiplications of the same weight matrix are involved. This means that portions of the gradients from long term dependencies involve high matrix powers of the same matrix. Depending on the eigenvalues of the weight matrix W, the high powers result in either exploding gradients or the portion of the gradient resulting from these long term dependencies vanish. In the case of an exploding gradient the weight update damage the already learned information stored in the weights, while the vanishing gradient problem manifests itself in an incapability of learning long term dependencies (or at least do that very slowly). There are a variety of different approaches for these problems. For example gradient clipping against the exploding gradient, smart weight initialization schemes, other activation functions and alternative RNN structures. The most robust and common approaches are alternative RNN structures where the common and state of the art versions are *gated RNNs*. We will only discuss the gated RNN structures and do not go into detail of different methods of dealing with the vanishing- and exploding gradient problem. For more information see [Goodfellow et al., 2016].

3.4.1 Simple RNN

This *Simple RNN*[4] is a variation of our vanilla RNN with an additional bias and a possible different activation function ϕ, although in most cases the hyperbolic tangent is a good choice for ϕ. This leads to

$$h_t = \phi(Wh_{t-1} + Ux_t + b) \tag{3.1}$$

with h_0 initialized zero and ϕ an appropriate activation function. See the Sequence-to-One structure in figure 3.7 for a visualization of the information

[4]The name "Simple RNN" is used in the `python` library `Keras` to address this RNN structure. The main reason why this structure is mentioned explicitly is that it is one of the architectures used in the experiments.

flow in our Simple RNN. For a Sequence-to-Sequence structure the internal state at time step t can be used as this time steps output.

3.4.2 Long Short-Term Memory (LSTM)

The *Long Short-Term Memory* (LSTM) introduced in [Hochreiter and Schmidhuber, 1997] is a gated RNN which tries to solve the problem of tracking long term dependencies. This is approached by introducing gates that control the information flow from one time step to the next. There are three gates namely the *input*, *forget* and the *output* gate. The input gate controls which information passed on from the past is relevant at the current time step, the forget gate decides if information is needed later on or if it can be forgotten. Finally the output gate decides which information is the current time steps output. Furthermore the gating behavior is also learned, meaning the LSTM learns how to control the gates. With these gates the problematic matrix multiplication for propagating the internal state through time was replaced by a gated sum. When information is propagated through time, the gates learn to behave in a way to stay closed for long term dependencies until the information is needed (or forgotten), then the gates open. Therefore the long term dependencies can be passed on without interfering with processing of time steps where these dependencies are not needed.

This is the basic concept of the LSTM which is incorporated in the recursive definition for the input gate i_t, output gate o_t and the forget gate f_t values. Furthermore the internal states, the "carry" state c_t and the memory state h_t.

$$\begin{aligned} i_t &= \sigma\left(W_i h_{t-1} + U_i x_t + b_i\right), \\ f_t &= \sigma\left(W_f h_{t-1} + U_f x_t + b_f\right), \\ o_t &= \sigma\left(W_o h_{t-1} + U_o x_t + b_o\right), \\ c_t &= f_t \odot c_{t-1} + i_t \odot \phi\left(W h_{t-1} + U x_t + b\right), \\ h_t &= o_t \odot \phi(c_t). \end{aligned} \tag{3.2}$$

The first three equations define the three gates with their own weight matrices dependent on the input and the previous hidden state using a sigmoid σ as activation function to ensure an actual gating behavior. Meaning that each channel of the gates are in the range $[0, 1]$ where 0 is a closed and 1 an open gate. The internal state c_t is the internal state composed of the gated previous state and the new processed information with ϕ as activation function, the operation $\odot$ is the hadamard product which is an element wise multiplication operating with the forget gate on the information propagated

from the past as well as the input gate controlling the new processed information. Finally we have the sequence output with the same activation function ϕ gated by the forget gate. For a visualization of the information flow in a single LSTM cell see figure 3.8. Note that this is only one time step inside a LSTM layer.

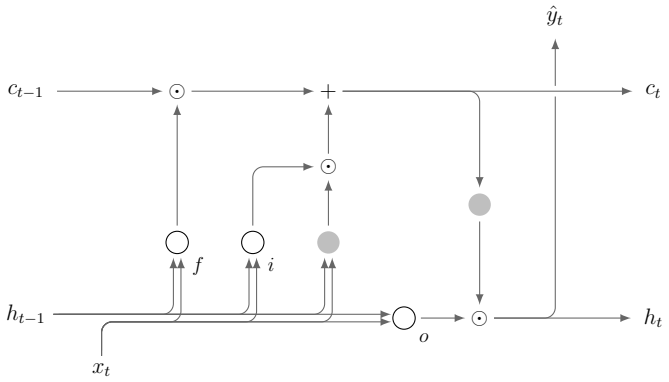

Figure 3.8: A single LSTM cell. At the current time step t the cell gets x_t as input as well as the internal state c_{t-1} and the output h_{t-1} from the previous step. The white circles ○ represent the gates and the gray circles ● compute new states and outputs. The $\odot$ represent an element wise multiplication (gating the information flow) and $+$ is a vector addition. After computation c_t and $h_t = \hat{y}_t$ is passed on to the same cell for the next time step and $\hat{y}_t$ is the output for the current step (if not discarded).

3.4.3 Gated Recurrent Unit (GRU)

The *Gated Recurrent Unit* (GRU) introduced in [Cho et al., 2014] is another gated RNN motivated by the LSTM. The main difference is given by using only two gates namely a *reset* gate and an *update* gate, furthermore the "carry" state is discarded while only a hidden state is propagated through time. The reset gate controls how much of the previous information is considered for computing the new hidden state, if the reset gate is closed the past is ignored and the hidden state is "reset" only with the current input. The update gate then decides how much of the previous hidden state shall be propagated fourth in combination with the new hidden state. The recursive definition for the update gate u_t and reset gate r_t as well as the hidden state h_t reads[5]

[5]The paper [Cho et al., 2014] which introduces the GRUs exists in multiple versions and there is a structural difference in the GRU between the versions. In earlier versions

$$
\begin{aligned}
r_t &= \sigma(W_r h_{t-1} + U_r x_t + b_r), \\
u_t &= \sigma(W_u h_{t-1} + U_u x_t + b_u), \\
h_t &= u_t \odot h_{t-1} + (1 - u_t) \odot \phi(W(r_t \odot h_{t-1}) + U x_t + b).
\end{aligned} \tag{3.3}
$$

The GRU structure for a single cell is visualized in figure 3.9.

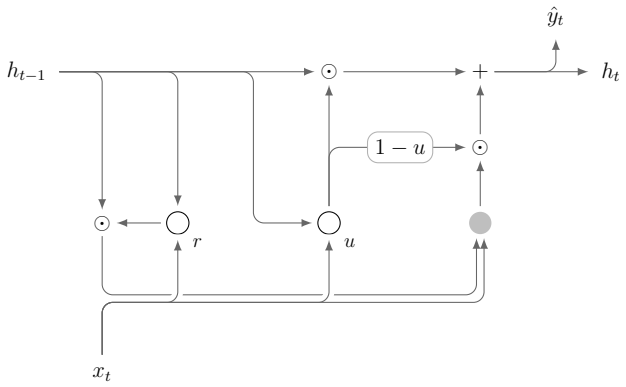

Figure 3.9: A single GRU cell. Inputs are x_t and the previous hidden state h_{t-1}, The white circles $\circ$ represent the gates where r is the reset gate and u the update gate. The $\odot$ operator is the element wise multiplication for applying the gating. Output is the new hidden state $h_t = \hat{y}_t$.

3.4.4 Bidirectional RNN

All the RNN structures considered until now have a causal structure, meaning at a time step t they are only aware of the past. But in natural language the interpretation of specific words is not only dependent of the past. This means that the meaning of a word can change dependent on future information. This is closely related to the problem of ambiguity in natural language. For example consider the two sentences "Auf der Bank liegt mein Hund." against "Auf der Bank liegt mein Geld."[6]. In both cases the interpretation of the word "Bank" is determined by the last word in the sentence, a dog lies on a bench and the money is on the bank. Another more specific example would

h_t was defined as

$$h_t = u_t \odot h_{t-1} + (1 - u_t) \odot \phi(r_t \odot W h_{t-1} + U x_t + b).$$

The difference is that the reset gate r is applied after the weight matrix. The experiments use the updated version presented in the text (not the one from the footnote).

[6] In English: "My dog is lying on the bank." and "My money is on the bank."

be “Die nächste Ente in den Nachrichten.”[7] where in this context the German word “Ente” means “hoax” or “fake news” and not the animal duck. In this example even the sentiment of the word changes through the reinterpretation of the word as being in the context of journalism.

Bidirectional RNNs address this issue by processing sequential data not only from the past to the future but in both directions. To construct such bidirectional RNNs one simply takes two RNNs, one processes the input sequence from the beginning and the other from the end and then, for each time step, the output is combined. The output combination for a time step t can be as easy as stacking the outputs or taking their sum or even point wise multiplication, the most usual case is to stack the outputs. Using this simple construction scheme all of the presented RNN structures can be implemented bidirectional. See figure 3.10 for a visualization of a bidirectional RNN layer.

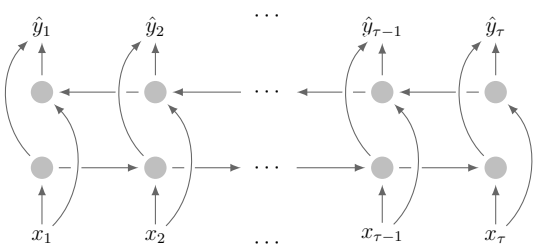

Figure 3.10: A Sequence-to-Sequence Bidirectional RNN structure.

3.5 Back Propagation (BP)

In this section the calculation of the gradient used in the optimization algorithms presented in section 2.1 is derived for the MLP. Extending the algorithm to more complex architectures like CNNs is straight forward but not for RNNs which is the content of section 3.6.

The *back propagation* (BP) algorithm is basically a computational efficient application of the chain rule for computing the gradient of neural networks for updating the weights while training as described in section 2.1. The idea is to apply the chain rule to the definition of a neural network in a functional form. The algorithm to actually compute the gradient is then a two step process consisting of the *forward pass* and the *backward pass*. In the forward pass the networks output is computed, this is just the evaluation of the neural networks output for given input. From an algorithmic point of view the BP algorithm also stores relevant information needed in the backward pass to

[7] In english: “The next hoax in the news.”

reduce computation overhead for values already computed. With the output of the network the gradient of the cost function $\nabla J(\theta)$ can be evaluated using the chain rule by propagating the evaluation of partial gradients from the output back to the input, therefore the name of the algorithm.

Let us explicitly derive the computation of the gradient for a simple MLP. Therefore we restate the definition of an MLP in a slightly different form which helps with the calculations. See 3.2 for the definition of the terms, now let

$$\begin{aligned} h^{(0)} &= x \\ a^{(l)} &= W^{(l)} h^{(l-1)} + b^{(l)}, && l = 1, ..., N \\ h^{(l)} &= \phi^{(l)}(a^{(l)}), && l = 1, ..., N \\ \hat{y} &= h^{(N)} \end{aligned}$$

where the function representing a single layer is

$$h^{(l)} = f^{(l)}(h^{(l-1)}; W^{(l)}, b^{(l)}) = \phi^{(l)}(W^{(l)} h^{(l-1)} + b^{(l)}) = \phi^{(l)}(a^{(l)})$$

which results in a function f representing the entire MLP as

$$f = f^{(N)} \circ f^{(N-1)} \circ ... \circ f^{(1)}.$$

To calculate the updates for the weights $W^{(l)}, b^{(l)}$ let θ represent all parameters in f. Furthermore, as described in section 2.1, let $\mathcal{L}$ be a per example loss for inputs $(x_i)_{i=1}^{m'}$ with target values $(y_i)_{i=1}^{m'}$. Then the cost function J is given as

$$J(\theta) = \frac{1}{m'} \sum_{i=1}^{m'} \mathcal{L}(y_i, \underbrace{f(x_i; \theta)}_{\hat{y}_i}).$$

This leads to

$$\nabla_{h^{(N)}} J = \nabla_{\hat{y}} J = \frac{1}{m'} \sum_{i=1}^{m'} \nabla_{\hat{y}} \mathcal{L}(y_i, \hat{y}_i)$$

which is the gradient of the cost function with respect to the last layers output. With the linearity it is sufficient to calculate the gradient for the per example loss $\mathcal{L}$. Now we compute the gradient parts in one MLP layer. Meaning we assume to have the gradient $\nabla_{h^{(l)}} \mathcal{L}$ of the layer l and derive the gradient for the weights in this layer as well as the loss with respect to this layers input, meaning $\nabla_{h^{(l-1)}} \mathcal{L}$. Because we got the gradient of the output layer the entire gradient is recursively defined and can be computed for all weights.

So lets assume we already have calculated $\nabla_{h^{(l)}}\mathcal{L}$ then the gradient of $\mathcal{L}$ for the linear layer output with the relation $h^{(l)} = \phi^{(l)}(a^{(l)})$ is

$$\begin{aligned}(\nabla_{a^{(l)}}\mathcal{L})_i &= \sum_{k=1}^{n_l} \frac{\partial \phi^{(l)}(a_k^{(l)})}{\partial a_i^{(l)}} (\nabla_{h^{(l)}}\mathcal{L})_k = \sum_{k=1}^{n_l} \delta_{ik} \phi^{(l)\prime}(a_k^{(l)}) (\nabla_{h^{(l)}}\mathcal{L})_k \\ &= \phi^{(l)\prime}(a_i^{(l)}) (\nabla_{h^{(l)}}\mathcal{L})_i\end{aligned}$$

where we get

$$\nabla_{a^{(l)}}\mathcal{L} = \phi^{(l)\prime}(a^{(l)}) \odot \nabla_{h^{(l)}}\mathcal{L}.$$

Now to the gradients for the weights $W^{(l)}, b^{(l)}$ which are the ones we are actually interested in. They derive from the identity $a^{(l)} = W^{(l)} h^{(l-1)} + b^{(l)}$ by

$$\begin{aligned}(\nabla_{W^{(l)}}\mathcal{L})_{ij} &= \sum_{k=1}^{n_l} \frac{\partial a_k^{(l)}}{\partial W_{ij}^{(l)}} (\nabla_{a^{(l)}}\mathcal{L})_k \\ &= \sum_{k=1}^{n_l} \frac{\partial \left(\sum_{t=1}^{n_{l-1}} W_{kt}^{(l)} h_t^{(l-1)} + b_k^{(l)} \right)}{\partial W_{ij}^{(l)}} (\nabla_{a^{(l)}}\mathcal{L})_k \\ &= \sum_{k=1}^{n_l} \sum_{t=1}^{n_{l-1}} \frac{\partial W_{kt}^{(l)} h_t^{(l-1)}}{\partial W_{ij}^{(l)}} (\nabla_{a^{(l)}}\mathcal{L})_k \\ &= \sum_{k=1}^{n_l} \sum_{t=1}^{n_{l-1}} \delta_{ki}\delta_{tj} h_t^{(l-1)} (\nabla_{a^{(l)}}\mathcal{L})_k \\ &= h_j^{(l-1)} (\nabla_{a^{(l)}}\mathcal{L})_i\end{aligned}$$

and therefore we obtain

$$\nabla_{W^{(l)}}\mathcal{L} = (\nabla_{a^{(l)}}\mathcal{L})\, h^{(l-1)^T}$$

and analog we get

$$\nabla_{b^{(l)}}\mathcal{L} = \nabla_{a^{(l)}}\mathcal{L}.$$

Finally we calculate the gradient of $\mathcal{L}$ with respect to the current layers

input using the relation $a^{(l)} = W^{(l)} h^{(l-1)} + b^{(l)}$ for

$$\begin{aligned}
(\nabla_{h^{(l-1)}} \mathcal{L})_i &= \sum_{k=1}^{n_l} \frac{\partial a_k^{(l)}}{\partial h_i^{(l-1)}} (\nabla_{a^{(l)}} \mathcal{L})_k \\
&= \sum_{k=1}^{n_l} \frac{\partial \left(\sum_{t=1}^{n_{l-1}} W_{kt}^{(l)} h_t^{(l-1)} + b_k^{(l)} \right)}{\partial h_i^{(l-1)}} (\nabla_{a^{(l)}} \mathcal{L})_k \\
&= \sum_{k=1}^{n_l} \sum_{t=1}^{n_{l-1}} \delta_{ti} W_{kt}^{(l)} (\nabla_{a^{(l)}} \mathcal{L})_k \\
&= \sum_{k=1}^{n_l} W_{ki}^{(l)} (\nabla_{a^{(l)}} \mathcal{L})_k.
\end{aligned}$$

Written in matrix form

$$\nabla_{h^{(l-1)}} \mathcal{L} = {W^{(l)}}^T \nabla_{a^{(l)}} \mathcal{L}.$$

Now we have calculated everything we need for the BP algorithm. We used the gradient of one layers output to compute the gradients of its weights as well as its input. Furthermore we have the gradient of the loss function after a forward pass through the network. Therefore we have a starting point and a recursive relation between the gradients.

To get the gradient of the cost function J we just need to average the per example gradients.

3.6 Back Propagation Through Time (BPTT)

The *Back Propagation Through Time* (BPTT) algorithm is an extension of the Back Propagation algorithm to compute the gradients in the backward pass for training RNNs. A RNN, in contrast to feed forward networks, has an additional time axis that consists of cyclic dependencies. The BPTT algorithm augments the classic BP algorithm to consider these cyclic dependencies inside a RNN layer.

For the sake of simplicity the BPTT will only be derived for our vanilla RNN, but the extension of the BPTT algorithm to more complex RNN structures follows the same principle. Let us restate our vanilla RNN layer which is given as

$$\begin{aligned}
h^{(t)} &= \tanh(W h^{(t-1)} + U x^{(t)}), \\
\hat{y}^{(t)} &= V h^{(t)}.
\end{aligned}$$

Furthermore let us assume we already have access to $\nabla_{\hat{y}^{(t)}}\mathcal{L}$. There are basically three possible cases for the gradient $\nabla_{\hat{y}^{(t)}}\mathcal{L}$ at time step t in the current RNN layer. First they are all zero except the last one $\nabla_{\hat{y}^{(\tau)}}\mathcal{L}$. This case would be the many-to-one RNN structure. This is the case for our classification models where only the last output (a complete sentence summary) is materialized. The second version is a many-to-many (or sequence-to-sequence) RNN structure with a target sequence $y_{t=1}^{\tau}$ and $\mathcal{L}$ the summation of all the losses $\mathcal{L}^{(t)}$. For example let for each time step t the loss be defined as $\mathcal{L}^{(t)}(y_t, \hat{y}_t) = \frac{1}{2}(y_t - \hat{y}_t)^2$ and the total loss function

$$\mathcal{L} = \sum_{t=1}^{\tau} \mathcal{L}^{(t)} = \frac{1}{2}\sum_{t=1}^{\tau}(y_t - \hat{y}_t)^2$$

then the gradient with respect to the time step t would read

$$\nabla_{\hat{y}^{(t)}}\mathcal{L} = y_t - \hat{y}_t.$$

For example consider the assignment to build a sentence tagger where the output sequence consists of word tags. The last possibility would be that $\nabla_{\hat{y}^{(t)}}\mathcal{L}$ is the already propagated gradient from following layers. Either way, assume we already know $\nabla_{\hat{y}^{(t)}}\mathcal{L}$.

We start to derive with the final time step and then go backwards in time,

$$\nabla_{h^{(\tau)}}\mathcal{L} = V^T \nabla_{\hat{y}^{(\tau)}}\mathcal{L}.$$

and for $t < \tau$ the internal state $h^{(t)}$ has $\hat{y}^{(t)}$ and $h^{(t+1)}$ as descendants, therefore we get

$$\nabla_{h^{(t)}}\mathcal{L} = \left(\frac{\partial h^{(t+1)}}{\partial h^{(t)}}\right)^T \nabla_{h^{(t+1)}}\mathcal{L} + \left(\frac{\partial \hat{y}^{(t)}}{\partial h^{(t)}}\right)^T \nabla_{\hat{y}^{(t)}}\mathcal{L}. \tag{3.4}$$

For a more explicit representation we compute the inner derivatives

$$\begin{aligned}\left(\frac{\partial h^{(t+1)}}{\partial h^{(t)}}\right)_{ij} &= \frac{\partial \tanh\left(\sum_k W_{ik} h_k^{(t)} + \sum_l U_{il} x_l^{(t+1)}\right)}{\partial h_j^{(t)}} \\ &= \left(1 - \tanh\left(W h^{(t)} + U x^{(t+1)}\right)^2\right)_i W_{ij} \\ &= \left(\operatorname{diag}\left(1 - h^{(t+1)^2}\right) W\right)_{ij}\end{aligned}$$

as well as the second

$$\frac{\partial \hat{y}^{(t)}}{\partial h^{(t)}} = \frac{\partial V h^{(t)}}{\partial h^{(t)}} = V.$$

By plugging into (3.4) we get

$$\nabla_{h^{(t)}}\mathcal{L} = W^T \operatorname{diag}\left(1 - h^{(t+1)^2}\right) \nabla_{h^{(t+1)}}\mathcal{L} + V^T \nabla_{\hat{y}^{(t)}}\mathcal{L}.$$

This is a recursive formula for the gradient where $\nabla_{h^{(t)}}\mathcal{L}$ depends on the gradient of the next time step $\nabla_{h^{(t+1)}}\mathcal{L}$. Eliminating the recursion leads to

$$\nabla_{h^{(t)}}\mathcal{L} = \sum_{k=t}^{\tau} \left(\prod_{l=k+1}^{\tau} W^T \operatorname{diag}\left(1 - h^{(l)^2}\right) \right) V^T \nabla_{\hat{y}^{(k)}}\mathcal{L}$$

To get the actually interesting gradient for the weights, we now compute the sum over all gradients for a specific time step and derive each loss with respect to the weights resulting in

$$\nabla_U \mathcal{L} = \sum_{t=1}^{\tau} \left(\frac{\partial h^{(t)}}{\partial U} \right)^T \nabla_{h^{(t)}}\mathcal{L} = \sum_{t=1}^{\tau} \operatorname{diag}\left(1 - h^{(t)^2}\right) \left(\nabla_{h^{(t)}}\mathcal{L}\right) h^{(t-1)^T},$$

$$\nabla_V \mathcal{L} = \sum_{t=1}^{\tau} \left(\frac{\partial \hat{y}^{(t)}}{\partial V} \right)^T \nabla_{\hat{y}^{(t)}}\mathcal{L} = \sum_{t=1}^{\tau} \left(\nabla_{\hat{y}^{(t)}}\mathcal{L}\right) h^{(t)^T},$$

$$\nabla_W \mathcal{L} = \sum_{t=1}^{\tau} \left(\frac{\partial h^{(t)}}{\partial W} \right)^T \nabla_{h^{(t)}}\mathcal{L} = \sum_{t=1}^{\tau} \operatorname{diag}\left(1 - h^{(t)^2}\right) \left(\nabla_{h^{(t)}}\mathcal{L}\right) h^{(t-1)^T}.$$

Chapter 4

Word Embeddings

When working with natural text we need to be able to represent the text in a form that the downstream processes can work with. This means we need representations of single entities in the text. This single entities can be characters, words, sentences or even entire documents. We work on a word level, meaning our smallest entities are words.

Remark. In the following we will talk mostly about words. But when being precise we actually mean *tokens*, for example in most NLP systems tokens like `'?'` or `'2019'` are in the vocabulary of known "words". In addition some tokens can represent multiple different "words" (character sequences). For example the `Polyglot` library has number expansions built in which lets the preprocessing represent all numbers of the same length (number of digits) by the same token. For example the character sequences `'1105'` and `'1729'` are both represented by the same token. Further approaches like stemming, replacing of similar meanings (for example abbreviations) or other data preprocessing and cleaning methods are involved. Anyway, in the following we assume a sequence of words (tokens).

4.1 Basics

In this section we briefly describe some basic text representation methods and a few useful statistics on text data on a word level.

Let $\mathcal{D}$ be a text corpus, meaning $\mathcal{D}$ is a set of documents. A *document* $D \in \mathcal{D}$ is just a sequence of words. Given a corpus $\mathcal{D}$ we build a *vocabulary* V of words in the corpus

$$V = \{w \in D : D \in \mathcal{D}\}.$$

Now given a vocabulary V we start by indexing the vocabulary. Let $i(w)$ be the index of a word. Mathematically the indexing is a bijection $i : V \to \{1, ..., |V|\}$ providing a one-to-one correspondence between a word and its index. Equivalent is the *one-hot-encoding* which is a vector of dimension $|V|$ where each component is associated with the word of this components index. Let $v(w)$ be this one-hot vector of a word w then $v(w) = (\delta_{i(w),j})_{j=1}^{|V|}$ is everywhere zero except when $j = i(w)$, therefore one-hot. At this point we are already able to represent entire documents as numbers, just representing a document as a sequence of word indices.

4.1.1 Word Counting Statistics

The basis of text analysis starts by counting words. The first is the *term frequency* (TF) which in its simplest form just counts occurrences of a term w in a document D. For our use case we assume our term w to be a word, although terms can be almost anything starting by a single character to entire text parts, in most cases a term is a single or a few words.

Raw counts are impractical (in most cases) because the size of a document is not considered. A more usual version is

$$\mathrm{TF}(w, D) = \frac{1}{|D|} \sum_{\tilde{w} \in D} \delta_{w,\tilde{w}}.$$

This is the most common TF version and also the version we use. There are multiple other variations like boolean count 1 if $w \in D$ else 0 or a logarithmic version $\log(1 + \mathrm{TF}(w, D))$.

Another word counting statistic is the *inverse document frequency* (IDF) which is intended as a measure of information a word (term) possesses for a document in relation to the entire corpus. The idea is that rare words in the corpus occurring in a single document must be relevant for this document. On the other hand common words in the corpus are words not relevant for a specific document. Like the term frequency the inverse document frequency comes in different variations. In other words, the IDF tells if finding a term in a document tells something about the document or not. For example finding the word `'und'`[1] in a document tells almost nothing about the document because this word is so common that the occurrence of `'und'` does not distinguish the document from others. One common version is

$$\mathrm{IDF}(w, \mathcal{D}) = -\log\left(\frac{|\{D \in \mathcal{D} : w \in D\}|}{|\mathcal{D}|}\right)$$

[1] In English: and

So the more documents containing a word the lower the IDF.

Combining both statistics leads to the *term frequency-inverse document frequency* (TF-IDF) statistic which is just the product of a TF and a IDF statistic

$$\text{TF-IDF}(w, D, \mathcal{D}) = \text{TF}(w, D)\,\text{IDF}(w, \mathcal{D}).$$

4.1.2 Bag of Words and n-Grams

A *Bag of Words* is a document representation by counting the words occurring in a document. With respect to a vocabulary V a document D can be represented by a $|V|$ dimensional vector. There the j-th component contains the raw count of the word w in the document with index $i(w) = j$ for each $w \in V$. There are multiple different alternatives, for example we can use TF-IDF statistics as alternative to raw word counts. Using a Bag of Words representation an entire corpus can be represented as a *document-term matrix* where each row of the matrix represents a document in the corpus and each column a word in our vocabulary. Therefore a document-term matrix is a $|\mathcal{D}| \times |V|$ dimensional matrix.

Let us consider an example, assume we have a document corpus $\mathcal{D}$ consisting of the following documents

$$\begin{aligned}
D_1 &= \texttt{'mach die Tür zu'} \\
D_2 &= \texttt{'mach die Tür auf'} \\
D_3 &= \texttt{'ist die Tür zu'} \\
D_4 &= \texttt{'ist die Tür auf'} \\
D_5 &= \texttt{'die Tür ist zu'} \\
D_6 &= \texttt{'die Tür ist auf'} \\
D_7 &= \texttt{'die Tür ist zu mach die Tür auf'} \\
D_8 &= \texttt{'die Tür ist auf mach die Tür zu'}.
\end{aligned}$$

First we create a vocabulary, there are six different words

$$V = (\texttt{'die'}, \texttt{'Tür'}, \texttt{'mach'}, \texttt{'zu'}, \texttt{'auf'}, \texttt{'ist'}).$$

With the vocabulary we create the document-term matrix

$$X = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ 2 & 2 & 1 & 1 & 1 & 1 \\ 2 & 2 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

Each row represents a document and each column a word.

A major drawback of the simple bag of words approach is that words are only considered by their occurrence counts. Meaning that co-occurrences are only captured on a document level. For example the documents D_7 and D_8 have the same bag of words representation. This leads to the n-gram model which considers n long sub sequences of words as terms and performs the same statistics on these n-grams. For example for $n = 2$ the following are all n-grams of our example corpus:

$$\begin{gathered} (\texttt{'mach'}, \texttt{'die'}), (\texttt{'die'}, \texttt{'Tür'}), (\texttt{'Tür'}, \texttt{'zu'}), \\ (\texttt{'Tür'}, \texttt{'auf'}), (\texttt{'ist'}, \texttt{'die'}), (\texttt{'Tür'}, \texttt{'ist'}), \\ (\texttt{'ist'}, \texttt{'zu'}), (\texttt{'ist'}, \texttt{'auf'}), (\texttt{'zu'}, \texttt{'mach'}), \\ (\texttt{'auf'}, \texttt{'mach'}) \end{gathered}$$

Again with the n-gram model a document-term matrix can be constructed with different statistics depending on the task. But we will not go into details, for more information see [Bird et al., 2009].

4.1.3 Latent Semantic Analysis (LSA)

The *latent semantic analysis* (LSA) is a method to study the relationship between documents and terms. The main idea is that terms with similar meaning occur in related documents.

Assuming a document-term matrix X, then LSA is a Singular Value Decomposition of the document-term matrix X. According to linear algebra there exist orthogonal matrices U and W as well as a diagonal matrix Σ with non negative diagonal elements (singular values) such that the document-term matrix can be written as

$$X = U\Sigma W^T.$$

This is known as the *Singular Value Decomposition* (SVD) of a matrix. Injecting the decomposition of X into X^TX and XX^T leads to

$$X^TX = W\Sigma U^T U\Sigma W^T = W\Sigma^2 W^T$$
$$XX^T = U\Sigma W^T W\Sigma U^T = U\Sigma^2 U^T$$

which leads to the fact that the columns of W and U are the eigenvectors of X^TX and XX^T respectively.

The SVD gives vector representations stored as columns (eigenvectors) of U and W for terms and documents respectively. With these representations documents and terms can be analyzed and new documents and terms can be embedded by using these matrices.

One improvement is to compute a low dimensional approximation and not a complete SVD which makes it possible to get representations in low dimensional spaces (not the vocabulary size). But as the corpus and vocabulary sizes grow LSA gets very expensive.

4.2 word2vec

The `word2vec` are a few neural networks to train word embeddings. They are initially proposed in [Mikolov, Chen, Corrado and Dean, 2013] and optimized in [Mikolov, Sutskever, Chen, Corrado and Dean, 2013]. Building on previous work which showed that for neural networks for NLP tasks a two step training is a good approach, the `word2vec` models where created for the first training step. This means that a neural network for a specific NLP task is created in two steps, first a neural network is trained for a representation of words, then a part of the first network (which is intended to have learned good representation of words) is used in the final network which is then trained for the actual NLP task. This approach is known as *transfer learning* where one part of a trained network is transferred to another network. The `word2vec` models only the first neural network in such a two step neural network modeling approach. There the projection matrix will be the *embedding matrix* used in further neural networks specialized for specific NLP tasks.

The `word2vec` models are intended to be very simple and efficient in training while having high quality representations. To get the efficiency in perspective the `word2vec` models are created to be able to be trained on multiple billions of words for vocabularies of millions of words. The quality is measured on different word similarity tasks. One such task is simple word similarity for instance that `'Hund'` and `'Katze'`[2] are close in the resulting vector

[2]In English: dog and cat

space (according to cosine distance). More complex similarity tasks like if `'groß'` is similar to `'größer'` in the same sense as `'klein'` to `'kleiner'`[3]. To validate these similarities the authors used algebraic operations on the word vectors and computed $v(\texttt{'größer'}) - v(\texttt{'groß'}) + v(\texttt{'klein'})$ and searched for the closest vector which is $v(\texttt{'kleiner'})$. Even more remarkable is that when trained on huge datasets, which the `word2vec` models are capable of, even more complex relations are encoded in the embedding structure like[4]

$$v(\texttt{'König'}) - v(\texttt{'Mann'}) + v(\texttt{'Frau'}) \simeq v(\texttt{'Königin'}).$$

In the following we introduce the two `word2vec` architectures. Therefore consider a training corpus X as a sequence of words $w_1, w_2, ..., w_{|X|}$. Then a vocabulary V of known words is created, typically taking the most common words in X. The vocabulary is also to be seen as a sequence of distinct words, this enables us to index the known words in our vocabulary. Now let $i(w)$ be the index of a word $w \in V$ and $v(w)$ be its continuous vector representation as a d dimensional vector.

Remark. The vocabulary has fixed size and does not contain all possible words. In most cases not even all words in the training corpus. Words not contained in the vocabulary are called *Out Of Vocabulary* (OOV) words. For handling these words we add a special OOV word to the vocabulary and tread all unknown words as this OOV word.

4.2.1 Continuous Bag-of-Words (CBOW)

The first model from [Mikolov, Chen, Corrado and Dean, 2013] is the *Continuous Bag-of-Words* (CBOW). This model is basically a log-linear classifier. The main objective is to train the classifier to predict a word given its context. The context of a word are all c previous and following words.

More precisely, let $w_t \in X$ be the t-th word in X, then its context $C_t = (w_{t+j})_{j=\pm 1,...,\pm c}$.

The CBOW model consists of an initial embedding layer which embeds the context words into $\mathbb{R}^d$, the weights of the embedding layer, namely the *embedding matrix*, contains after training the word vectors and the rest of the model will be discarded. After embedding the context words their representations are averaged to a continuous context representation, therefore also

[3] In English: big to bigger as small to smaller

[4] Equation in English:

$$v(\texttt{'king'}) - v(\texttt{'man'}) + v(\texttt{'woman'}) \simeq v(\texttt{'queen'})$$

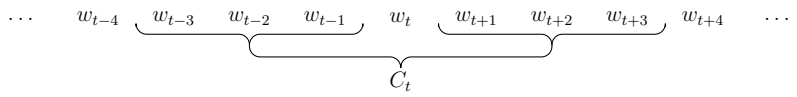

Figure 4.1: Context words with window size $c = 3$ for word w_t.

the name because the context representation only bags the context words by averaging. The context representation is then passed to a log-linear classifier which has its own weights with a softmax output. The output is because of the softmax a probability distribution with $|V|$ nodes. The j-th output node represents the modeled probability $p(i(w_t) = j|C_t)$ given its context C_t. The training objective is to minimize the negative log-likelihood for the models parameters θ (consisting of the embedding matrix and the classifier weights)

$$J(\theta) = -\frac{1}{|X|} \log \mathcal{L}(\theta) = -\frac{1}{|X|} \sum_{t=1}^{|X|} \log p(w_t|C_t).$$

The probability $p(w_t|C_t)$ is associated to the output neuron representing the word w_t.

In more detail, let $E \in \mathbb{R}^{|V| \times d}$ be the *embedding matrix* which represents the word vectors in its rows, meaning $v(w) = E_{i(w),.}$. The context vector is computed as the average of the context word vectors

$$v(C_t) = \frac{1}{|C_t|} \sum_{w \in C_t} v(w).$$

This context vector is the input to the log-linear classifier with weights $U \in \mathbb{R}^{|V| \times d}$. The rows of U can be interpreted as a context "word" representation $u(w) = U_{i(w),.}$ similar to the embedding matrix, meaning that the model has two vector representations for a word, namely a word and a context representation. Now the classifier compares each context representation $u(w)$ with the given context vector $v(C_t)$ to determine their similarity and models the probability of a word $w \in V$ being the center word of the context C_t as

$$p(w|C_t) = \text{softmax}(v(C_t)^T u(w)).$$

This is the basic architecture of the CBOW model, see figure 4.2 for a visualization.

This approach is still very expensive because to evaluate the probability $p(w_t|C_t)$ the softmax must be evaluated

$$p(w_t|C_t) = \text{softmax}(v(C_t)^T u(w_t)) = \frac{\exp(v(C_t)^T u(w_t))}{\sum_{w \in V} \exp(v(C_t)^T u(w))}.$$

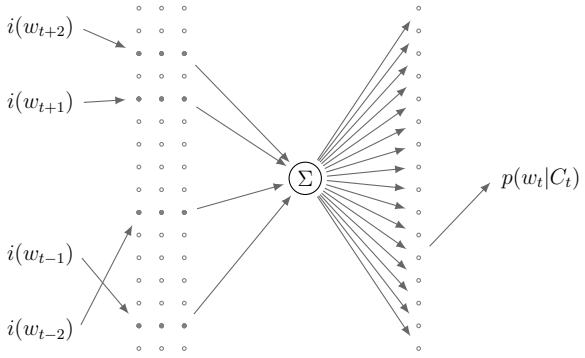

Figure 4.2: The CBOW model with softmax activation.

The problem with the softmax activation is the denominator which is a sum over the entire vocabulary and therefore the computational cost is still proportional to the vocabulary size. But the softmax function can be approximated with the *hierarchical softmax*. The computational cost to evaluate the hierarchical softmax is proportional to $d \times \log |V|$. Because for training we are only interested in the probability $p(w_t|C_t)$ for the single word w_t (and not for all words to compute the maximum for classification) which leads to a significant speedup.

The hierarchical softmax uses a binary tree[5] representation of the output layer with $|V|$ leaves. Each leave corresponds to one word, each node represents the relative probability of all its children. Therefore by walking from the root to a leave and accumulating the relative probabilities leads to the probability of the leave node associated with a word.

More precisely, each word w can be reached via a unique path from the root to the leave associated to the word w. Let $n(l, w)$ be the l-th node in the path from the root to the word w and $L(w)$ the length of the path. Additionally the context representation is no longer present, in replacement each node gets a node representation $u(n)$. Furthermore let $[[x]]$ be a truth evaluation as $[[x]] = 1$ if x is true and -1 otherwise. Finally let $\text{left}(n)$ be the left child of a node n in the tree representation[6]. Then the hierarchical softmax defines the probability of the center word w_t given its context C_t as

$$p(w_t|C_t) = \prod_{l=1}^{L(w_t)-1} \sigma\left([[n(w_t, l+1) = \text{left}(n(w_t, l))]] v(C_t)^T u(n(w_t, l))\right).$$

[5]The tree structure used in [Mikolov, Chen, Corrado and Dean, 2013; Mikolov, Sutskever, Chen, Corrado and Dean, 2013] was a binary Huffman tree.

[6]Choosing the left over the right child is arbitrary, can be either way.

with $\sigma(x) = \frac{1}{1+\exp(-x)}$ the logistic sigmoid. In figure 4.3 the CBOW model with a hierarchical softmax using a binary tree is visualized.

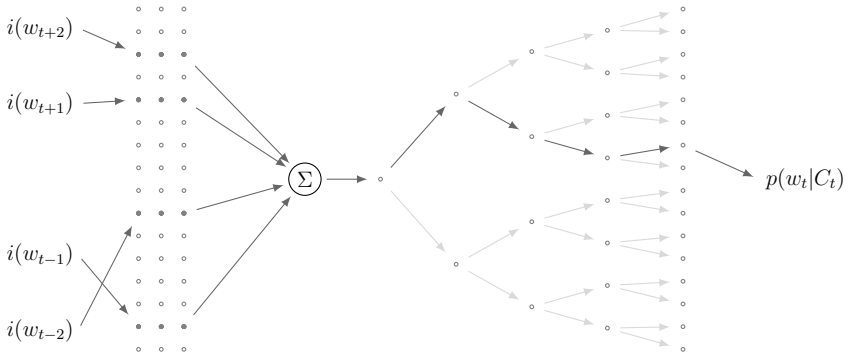

Figure 4.3: The CBOW model with hierarchical softmax output layer.

4.2.2 Skip-Gram

The *Skip-Gram* model is the second `word2vec` architecture from [Mikolov, Chen, Corrado and Dean, 2013]. This model is similar to the CBOW model except that for a given word the Skip-Gram model predicts context words. In contrast to the CBOW model which learns to predict a word given its context the Skip-Gram model learns to predict the context given a word.

With the same notation the learning objective is to minimize the negative log-likelihood given as

$$J(\theta) = -\frac{1}{|X|} \log \mathcal{L}(\theta) = -\frac{1}{|X|} \sum_{t=1}^{|X|} \log p(C_t|w_t).$$

So the difference is in the conditional probability. The log of $p(C_t|w_t)$ is given as

$$\log p(C_t|w_t) = \log \prod_{w \in C_t} p(w|w_t) = \sum_{w \in C_t} \log p(w|w_t).$$

The model structure is similar to the CBOW starting with an embedding matrix of size $|V| \times d$ for word vector representations of dimension d. But the Skip-Gram model gets as input only a single word (the center words) whose word vector $v(w_t)$ is then the input to a log-linear classifier. Then the classifier predicts context words given the center word. Like the CBOW model the Skip-Gram model should be trained with hierarchical softmax for computational reasons. See figure 4.4.

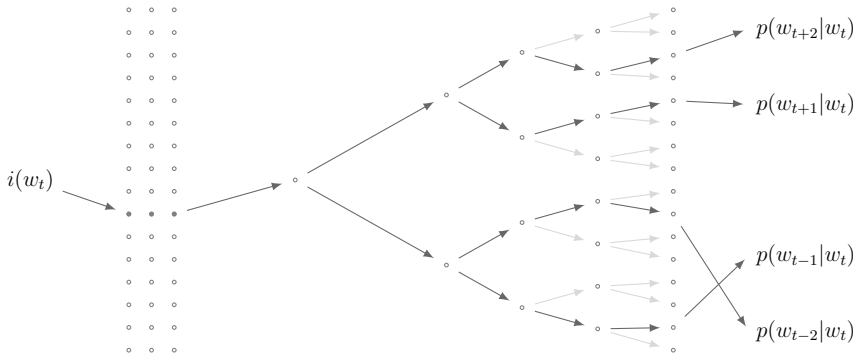

Figure 4.4: Skip-Gram model architecture with hierarchical softmax output layer.

4.2.3 Negative Sampling and Subsampling of Frequent Words

An alternative to the hierarchical softmax is *negative sampling* introduced in [Mikolov, Sutskever, Chen, Corrado and Dean, 2013]. The idea is that language models should be able to distinguish actual data from noise. Therefore the multi class prediction is replaced by a binary classification to distinguish between real and negative samples. The data seen by the model is now the center word w_t, its context C_t and negative samples $w_t \notin N_t \subset V$. Using a noise distribution p_n the negative samples $w \in N_t$ are drawn from the vocabulary with $w \sim p_n$, the number of negative samples $|N_t|$ range between $2-20$ depending on the given training corpus size (small corpora need more negative samples). With the negative samples the new training objective for the CBOW model is

$$J(\theta) = -\frac{1}{|X|}\sum_{t=1}^{|X|}\left(\log\sigma(v(C_t)^T u(w_t)) - \sum_{w\in N_t}\log\sigma(v(C_t)^T u(w))\right).$$

This can be applied in an analog manner to the Skip-Gram model as well.

Another improvement from the paper [Mikolov, Sutskever, Chen, Corrado and Dean, 2013] for learning `word2vec` vectors is *subsampling of frequent words*. The idea is that high frequent words can occur extremely often in comparison to others. These high frequent words are usually less informative than low frequent words, especially when learning word representations based on context word relations, because most of these high frequent words can be present in almost every context, for example articles or conjunctions. During training these high frequent words dominate the training and can even harm the learning of lower frequent words. To mitigate this harmful effect of highly

occurring words, words w_t in the training corpus X are discarded with a probability p_d given as

$$p_d(w_t) = 1 - \sqrt{\frac{T}{\mathrm{TF}(w_t, X)}}.$$

There T is a constant threshold (around 10^{-5}, see: [Mikolov, Sutskever, Chen, Corrado and Dean, 2013]) and TF is the term frequency. The discarding probability was chosen heuristically but seems to work well in practice and was designed to drastically discard high frequent words with a frequency above T while keeping low frequent words and still preserves the frequency order (see: figure 4.5).

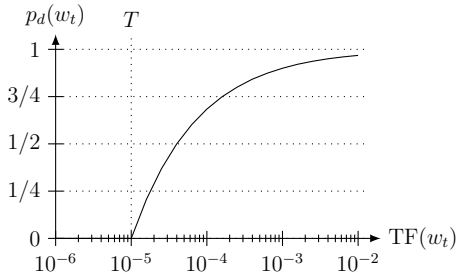

Figure 4.5: Discarding probability for a term frequency threshold of 10^{-5}.

4.3 fastText

Inspired by [Levy et al., 2015; Mikolov, Chen, Corrado and Dean, 2013] `fastText` initially is a linear text classifier with rank constraint and hierarchical softmax output layer introduced in [Joulin et al., 2016]. As the name suggests this classifier is very fast to train. Furthermore it does not use any pretrained embeddings. Building on their linear classifier in [Mikolov et al., 2018] the authors adapted their classifier model to train word vectors. Similar to [Mikolov, Chen, Corrado and Dean, 2013; Mikolov, Sutskever, Chen, Corrado and Dean, 2013] their embedding model comes in two flavors, namely the CBOW and the Skip-Gram architecture although in [Mikolov et al., 2018] the `fastText` embedding was introduced with the CBOW architecture. Therefore we assume the CBOW architecture for the rest of this section.

The `fastText` CBOW model is very similar to the original `word2vec` version (see: section 4.2.1) except two changes:

Position Dependent Weighting: The `word2vec` model ignores the order of the context words by calculating the context vectors as average of the context word vectors. To consider their order the calculation of the context vectors $v(C_t)$ are replaced by a positional weighted sum where the weights are learned too. Now for each relative position $j = \pm 1, ..., \pm c$ in the context a weighting vector d_p is associated. Then the context vector is computed as

$$v(C_t) = \sum_{j=\pm 1,...,\pm c} d_j \odot w_{t+j}.$$

Subword Information: Typical word vectors ignore morphological information of words. To encode this information in the embedding a bag of character n-gram's in the word is added to the standard word vector. Meaning that character n-grams also have vectors associated which are derived from a singular value decomposition or are learned similar to the word vectors from a text corpus. Now to get a vector representation of a word w, the word is first decomposed into a set of known n-grams N in the word w where x_n denotes the vector representation of a n-gram $n \in N$. Let $v(w)$ be the classic word vector for w, then its vector with subword information is

$$v(w) + \frac{1}{|N|} \sum_{n \in N} x_n.$$

The size of the n-grams is typically restricted from 3 to 6 characters. The n-gram vectors can also be used to calculate word vectors for unknown words, meaning for OOV words, by simply initializing $v(w)$ as zero vector or using a trained OOV representation. Therefore the `fastText` embedding produces different word vectors for different OOV words.

Remark. Another well known word embedding is the `GloVe` word embedding introduced in [Pennington et al., 2014].

Chapter 5

Experiments

5.1 Experiment Setup

The code for the experiments was written in `python`. There I used pretrained word embeddings provided by the `Polyglot` [Al-Rfou et al., 2013] library as well as the `fastText` [Grave et al., 2018] embedding with `gensim` [Řehůřek and Sojka, 2010] as the interface. The neural networks are implemented with `Keras` [Chollet et al., 2015] using `TensorFlow` as backend.

5.1.1 Preprocessing

This is the first step in the processing chain. The preprocessing cleans the raw data (sentences or entire texts), then splits the cleaned data into tokens (words, punctuations, dates, ...) and finally, dependent on the parameterization, either removes or leaves stop words and/or punctuations.

Escaping, Replacing and Tokenization

Before removing and splitting some special characters must be escaped. This way they can be reconstructed after the cleanup and do not interfere with the splitting. For example consider the raw data sentence `'Am 3. Februar wurde § 3 Abs. 4 bespro-\nchen.'`. There are two dots that are not sentence endings, the character `'§'` which has an explicit meaning but may not have an embedding and the date `'3. Februar'` which should be a single token[1] despite the fact that there is a space in it. Furthermore the `-\n` must be deleted as the word splitting over a line break. All of these special cases are handled by replacing and escaping. At first abbreviations, escape

[1] It is more common to split the term `3. Februar` in three tokens ⟨3⟩ ⟨.⟩ ⟨Februar⟩, either way the dot is not a sentence delimiter.

sequences and special characters are replaced by actual words, for a few examples see table 5.1. This transforms our example sentence to `'Am 3. Februar wurde Paragraph 3 Absatz 4 bespro-\nchen.'`. The next step is to escape characters that should be kept but not be treated as sentence or token delimiters like the dot and space in the date of our example. The characters to be escaped are searched by specialized RegEx (for example dates and numbers). Then all remaining sentence and token delimiters are actual sentence and token delimiters as well as all remaining "illegal" characters can be removed safely. After the removal of the illegal characters the text is split with respect to the remaining delimiters. Afterwards the escaped tokens are restored. For our example we get the following sequence of tokens ⟨Am⟩ ⟨3. Februar⟩ ⟨wurde⟩ ⟨Paragraph⟩ ⟨3⟩ ⟨Abschnitt⟩ ⟨4⟩ ⟨besprochen⟩ ⟨.⟩.

abbreviation	replacement
Dr.	Doktor
Abg.	Abgeordneter
Nr.	Nummer
z.B.	zum Beispiel
§	Paragraph
€	Euro
€	Euro

Table 5.1: Examples of abbreviation and special character replacements

5.1.2 Word Embeddings

In the experiments 4 different word embeddings where compared, 3 provided by `Polyglot` and the `fastText` embedding using `gensim` as an interface. The sentence embedding is done by a data generator which gets the preprocessed sentences and passes batches of embedded sentences to the `Keras` models.[2]

Polyglot Embeddings

The `Polyglot`[3] library provides three different embeddings directly with a download utility for a wide range of different languages, in our case German.

[2]The `Polyglot` embeddings have fixed size and could be implemented with a `Keras` embedding layer, but to provide a uniform processing chain (more or less independent of the embedding) all embeddings are performed in a data generator.

[3]For documentation see: `https://polyglot.readthedocs.io/en/latest/` and for the `GitHub` project see: `https://github.com/aboSamoor/polyglot`.

In the tests all three different embeddings, namely `cw`, `ue` and `sgns`, where compared. The `cw` embedding was trained on Wikipedia and contains the 100.000 most common words in Wikipedia. A detailed description of the `cw` embedding can be found in [Al-Rfou et al., 2013]. Sadly, for the other two embeddings I was not able to find out how they where created or where they came from. For a comparison of the vocabulary size and the word vector dimensions see table 5.2.

Name	Known Words	Vector Dim.
`cw`	100.004	64
`ue`	100.000	128
`sgns`	483.629	256
`fastText`	2.000.000	300

Table 5.2: Sizes of the used Embeddings

Finally, depending on the options, stop words and punctuations are kept or removed from the sequence of tokens.

fastText Embedding

The `fastText` embedding was also trained on Wikipedia data for multiple different languages. I used the pretrained word embeddings for German in the binary format[4] with `gensim`. The `gensim` library provides a `python` interface to embed arbitrary words with the `fastText` embedding. The binary format in combination with `gensim` was chosen because it provides the capability to embed OOV words using subword information as described in section 4.3. But there is a drawback from a technical point of view, because with embeddings of OOV words the `Keras` embedding layer cannot be used any more. The reason is that the embedding layer is basically a word vector lookup table for all known words, but if the vocabulary size is not fixed because of OOV words, there cannot be a lookup table for all possible words. This is one of two reasons, why the embedding layer was replaced by a data generator which provides already embedded sentences using `gensim`.

Data Generator

A Data Generator in `Keras` provides batched data as model input for either training and validation or prediction. In the case of training and validation

[4]File Source (Accessed Sep. 1, 2018): `https://github.com/facebookresearch/fastText/blob/master/pretrained-vectors.md`

the model expects the embedded sentences and the labels for each sentence. For prediction only the embedded sentences are passed to the model. The data generator gets a list of all sentences (preprocessed) and provides batches of maximum size, meaning number of sentences in a batch, grouped by sentence length. When the model requests a batch for training, validation or prediction, the data generator passes one of the batches to the model until all batches are exhausted. In the case of a model training multiple runs through the entire dataset (epochs) are needed and the order of the training examples should be shuffled for each iteration. To accomplish this shuffling with the constraint of grouping the sentences by length the data generator shuffled the sentence groups after each epoch and recreates the batches. This way the batches contain different combinations of sentences for each epoch.

5.1.3 Model Builder

A comparison of different ANN models with a wide range of different possible parameters makes it difficult for an automated comparison. Furthermore not all model types are capable of processing varying sizes of input data, which is required in our case, because sentences vary in length. Therefore a model builder was implemented which is capable of building different `Keras` models only parameterized by a model name (type) and model type specific parameters described below. In combination with the data generator all of these models can process and be trained on the same data without any specialization (the main problem is the varying length of sentences and the wish to use the capability of the `fastText` embedding to embed OOV words). The model builder makes it possible to perform automatic model parameter comparisons with performance measures like accuracy, precision, recall or F_1 scores. Also libraries like `scikit-learn`[5] can be used for parameter optimization over multiple different types of ANNs.

Models

There are three base types MLPs, CNNs and RNNs. All of them are different in their internal structure and behavior. The MLPs and CNNs need to summarize the embedded sentences of varying size because both cannot summarize the different length input into a fixed output by themselves. In the case of an MLP this summary needs to be done before even getting into the "actual" MLP by adding a global pooling layer as first layer, which is either an averaging or a max pooling over all word vectors, resulting in a summary

[5] See: `https://scikit-learn.org/stable/index.html`.

of the input sentence of fixed size. These are the two different MLP model types called *MLP (Max)* and *MLP (Avg)*. For the case of CNNs the convolutional layers perform a convolution over the different sized input sequences, but after being processed by the convolutional layers the data is still of varying size. Then, like in the MLP models, they are summarized and fed into a final MLP for classification resulting in the two CNN types *CNN (Max)* and *CNN (Avg)*. In contrary to the MLPs and CNNs the different RNNs, namely *SimpleRNN*, *GRU* and *LSTM* are specialized for sequence processing and create a fixed input to a final MLP for classification. In addition the RNNs can easily be adapted to bidirectional RNNs by taking two RNN layers and combining the outputs of the two RNNs where one processes the sequential input in the opposite direction as the other one. The bidirectional RNNs are the *BiSimpleRNN*, *BiGRU* and the *BiLSTM*. Finally a combination of CNNs with RNNs can be created by replacing the global pooling of a CNN with a many-to-one RNN layer. This leads to models like *CNN (SimpleRNN)*, *CNN (BiSimpleRNN)*, *CNN (GRU)* and so on.

For more details and the corresponding theory see section 3.

Parameters

All of the different models have hyperparameters to explicitly define their structure. Because of their conceptual divergence they are not all parameterized by the same hyperparameters, although some are shared by all models, see table 5.3 for a list of shared parameters and fixed values if these parameters are not part of the grid search. Hyperparameters specific to special model types are listed in table 5.4 where the model parameters are marked with a $\times$ when they are a hyperparameter of the model type.

Following a short description of the hyperparameters.

Units: These parameters define the hidden units in the network. In more detail the `units` are a list of output dimensions of a convolutional or recurrent layer while the `MLP_units` are the dimensions of the hidden layers in the final MLP without the output layer which is set fixed to 3 by the `output_units` parameter. This is because of the given 3 class classification problem.

Activation: Defines which activation function is used. There are four parameters, first the `activation` setting the activation in the convolutional or recurrent layers, next the `MLP_activation` defines the activation function in the final MLP. The `recurrent_activation` is the activation of the gates in GRUs and LSTMs (typically the logistic sigmoid σ) and finally the fixed `output_activation` is set to a softmax as output activation of all networks.

Hyperparameters	Fixed Value(s)
`output_units`	3
`output_activation`	softmax
`MLP_units`	
`MLP_activation`	
`MLP_kernel_constraint`	
`dropout`	
`loss`	cross entropy
`metrics`	accuracy, F_1, P, R
`epochs`	50 (60 in fine-tuning)
`batch_size`	
`optimizer`	Adam

Table 5.3: Hyperparameters common to all models. If no fixed value is given, then this parameter contributes in the grid search.

Pooling: The pooling in the CNNs is defined through the `pool_type` and `pool_size`, the first defines the pooling type which is either a max or average pooling and the `pool_size` defines the size of the pooling area.

Kernel Size: The `kernel_size` parameter defines the size of CNN kernels, meaning the kernel size used in a constructional layer.

Dropout: The `dropout` parameter gives the dropout probability applied to each hidden neuron in the entire network, meaning in convolutional or recurrent layers as well as in the MLP layers. Only dropout of the states propagated through time in recurrent layers is not set by the `dropout` parameter, for that the `recurrent_dropout` parameter is responsible.

Constraints: The `MLP_kernel_constraint` sets a max-constraint to the weights of the MLP. The `kernel_constraint` parameter sets constraints to convolutional or recurrent layers. For RNNs defined by the equations (3.1), (3.2) and (3.3) the `kernel_constraint` is applied to the U matrices and the `recurrent_constraint` to the W weights of RNNs.

Merge Mode: The `merge_mode` is specific for bidirectional RNNs determining the combination of outputs from the two time opposed RNNs in a bidirectional layer.

	units	activation	kernel_constraint	kernel_size	pool_type	pool_size	recurrent_activation	recurrent_constraint	recurrent_dropout	merge_mode
MLP (Max)										
MLP (Avg)										
CNN (Max)	×	×	×	×	×	×				
CNN (Avg)	×	×	×	×	×	×				
SimpleRNN	×	×	×							
GRU	×	×	×				×	×	×	
LSTM	×	×	×				×	×	×	
BiSimpleRNN	×	×	×					×	×	×
BiGRU	×	×	×				×	×	×	×
BiLSTM	×	×	×				×	×	×	×
CNN (SimpleRNN)	×	×	×	×	×	×		×	×	
CNN (BiSimpleRNN)	×	×	×	×	×	×		×	×	×
CNN (GRU)	×	×	×	×	×	×	×	×	×	
CNN (BiGRU)	×	×	×	×	×	×	×	×	×	×
CNN (LSTM)	×	×	×	×	×	×	×	×	×	
CNN (BiLSTM)	×	×	×	×	×	×	×	×	×	×

Table 5.4: Hyperparameter grid of parameters not shared by all models. The × means that the model has this hyperparameter.

5.2 Model Selection

Model selection and comparison were performed by multiple grid searches. Because the number of all possible hyperparameter combinations is way to large for an exhaustive search through the entire grid, multiple smaller grid searches where performed. These smaller grids where (like the initial choice of useful parameters) chosen according to "educated guesses" about useful and interesting parameter combinations. For example the `recurrent_activation` parameter was only for a few examples set to a different activation function then the logistic sigmoid σ and fixed for almost all other searches.

Each model validation was performed via accuracy, precision P, recall R and F_1 with a 3-fold cross validation where the final metrics where computed as the average over each fold. Furthermore the search was performed on 90% of the entire dataset and the withhold 10% test set was only used later on for final validation of the selected models.

For a theoretical description of a grid search and cross validation as well as details about the performance metrics see chapter 2.

5.2.1 Grid Search

The possible values for the hyperparameters are listed in table 5.5.

Hyperparameter	Domain (Specification)
`MLP_units` `units`	0 to 3 hidden layers, each with one of 32, 64, 128 or 256 neurons per layer

`MLP_activation`
`activation`
`recurrent_activation`
	One of ReLU, tanh, logistic sigmoid σ or linear (no activation)

`MLP_kernel_constraint`
`kernel_constraint`
`recurrent_constraint`
	No constraint or max constraint of 1 or 4

`dropout` `recurrent_dropout`	No dropout or 20%, 30% or 40%
`batch_size`	Either 32, 64, 128, 256 or 512
`kernel_size`	3
`pool_type`	Max or Average
`pool_size`	2 or 3
`merge_mode`	Sum, product or stacked

Table 5.5: Grid search parameter domain.

The search consisted of 1314 different parameter combinations using the `cw` and the `fastText` embeddings. To save time, the `ue` and `sgns` embeddings where not evaluated for all models (650 per embedding) because the results were not too interesting as explained in setion 5.3.1. All together this leads to 3928 models for comparison, each model needed to be trained 3 times because of the 3-fold cross validation in the grid search. For the search the parameterization of the preprocessing was fixed to keep all stopwords and punctuation. The different preprocessing variations where compared only for the best performing models of the grid search, see section 5.2.2.

The total run time of all grid searches was about 6 weeks of computation on my personal computer[6].

5.2.2 Fine-Tuning

After the grid searches the results where analyzed and three of the best performing models where chosen for further tuning. Again the models where validated with a 3-fold cross validation. But in contrast to the model selection grid search all random number generators[7] where seeded for repeatable results. Another difference is that in the fine-tuning the training epochs where increased to 60. Also all preprocessing parameter combinations where compared by performing a cross validation for each different preprocessing parameterization of the training data for each of the chosen models.

5.3 Results

In this section the results of the experiments described above are presented, starting with an overview of the grid search results. In the following we will use the term 'score' for the mean accuracy of a 3-fold cross validation of a model. In figure 5.1 the minimum to maximum score grouped by model type and used embedding are presented in order of the maximum score. Comparing the group minimum values has to be taken cautiously. They are hardly comparable because this overview contains all models including models which are configured to study specific behavior like overfitting or

[6] About 80% CPU usage on all 4 cores with an Intel® Core™ i7-7500U CPU @ 2.70GHz × 4 and 15.6GiB RAM.

[7] Random number generators involved are from `python`, `numpy`, `Keras` as well as the `TensorFlow` backend. Furthermore only a single `TensorFlow` session can run and the session itself must run single threaded, otherwise race conditions could lead to different results despite seeding all random number generators. Additionally note that the data generator randomly rearranges the training batches after each epoch and therefore changing the data generator would also break the repeatability.

parameter combinations which were known not to be a good choice from the beginning. On the other hand the maximum scores give a good overview of a models capability in combination with a specific embedding.

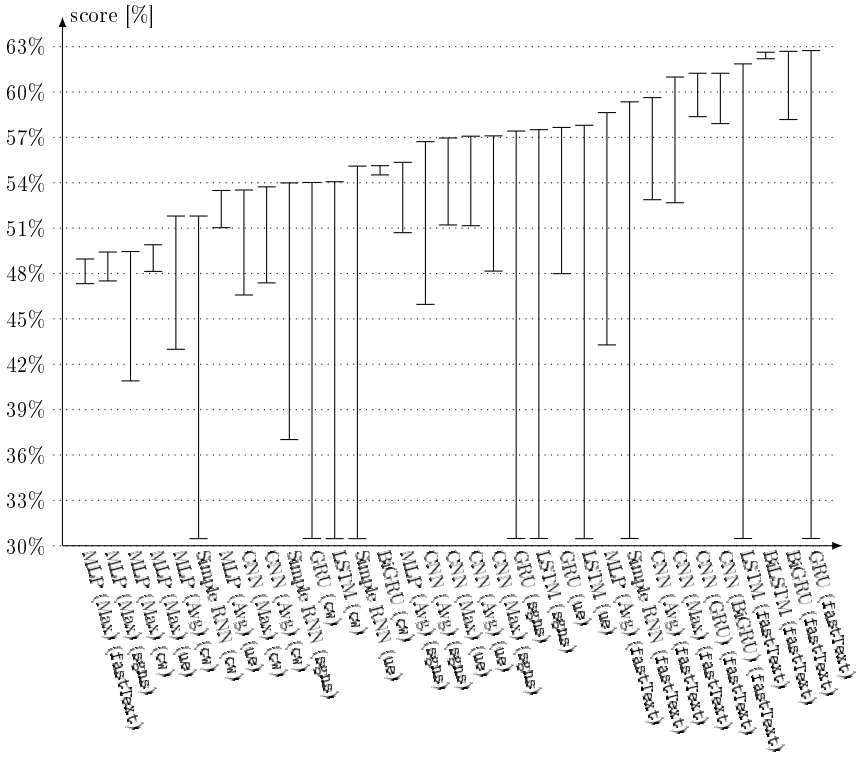

Figure 5.1: Comparison of all grid search models with embedding.

5.3.1 Embeddings

As already seen in figure 5.1 the embeddings can be clearly distinguished by their capability. One can see that the `fastText` embedding is by far the most powerful. This is not surprising because it is the biggest and the most advanced embedding, see: section 4.3. The second place is shared by `sgns` and `ue` which is a bit surprising if one checks the sizes of those two embeddings as listed in table 5.2. Even though the `sgns` embedding is bigger in the size of known words and the word vector dimension they still perform alike. By far the worst is the `cw` embedding. In the following only the `cw` and `fastText` embedding are compared. This is because almost all measures of the `sgns` and `ue` are in between the other two. Despite their experimental measures, which are rarely interesting as bounded by the

other two embeddings, their construction or origin could not be investigated (except that they are provided by the `Polyglot` library).

For an overall comparison of the embeddings their precision against recall was compared as well, meaning the precision and recall of all models building upon a specific embedding. As visualized in figure 5.2 it is not only the embedding that leads to a good classifier even though models using the `fastText` embedding tend to be more balanced between precision and recall as well as differences between classes.

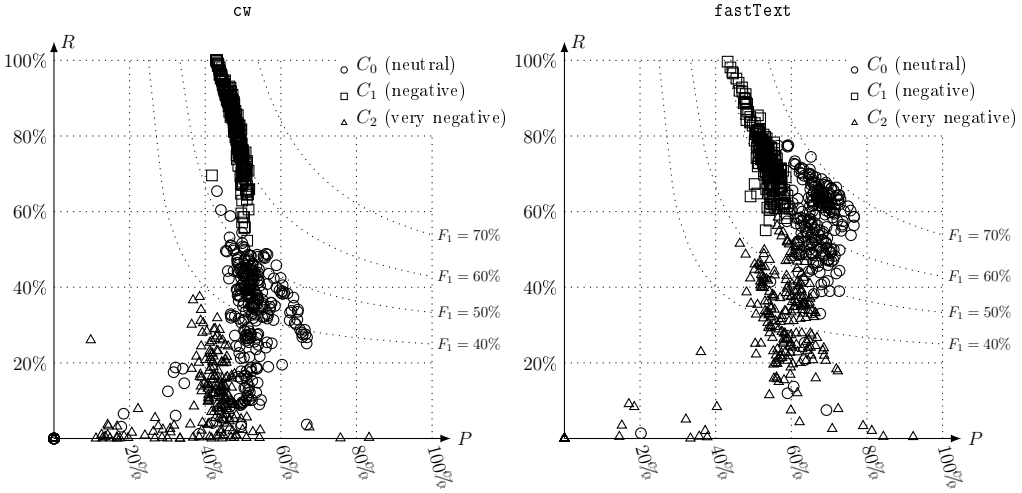

Figure 5.2: Precision P against recall R for all models with the same embedding. Left: `cw`, Right: `fastText`.

When further only the "good" models are compared for a specific embedding, meaning for the `cw` embedding only models with a score higher then 50% and for the `fastText` embedding all models with a score higher than 60%, see figure 5.3. There are a few points where the embeddings differ, first the maximum score of the models which is also seen in the class specific analysis. Second, the models built on the `cw` embedding are clearly biased in favor of the biggest class C_1 (negative), with and without class weights, while the `fastText` embedding models do not have this bias. This is seen in the very high recall of class C_1 (negative) for `cw` in contrast to the bit lower precision but higher recall for `fastText` models. Another indicator for a low bias in the `fastText` models is given through a balanced distribution of precision against recall for all three classes. Additionally the `fastText` models are way better in distinguishing if a sentence is neutral C_0 or not $C_1 \cup C_2$ while it is more challenging to determine how negative a sentence is. This is supported by the really good metrics for the class C_0 (neutral). The only real evidence of an imbalanced class distribution in the recall and precision

metrics of the `fastText` models is seen in the higher recall for the class C_1 (negative) in comparison to C_2 (very negative). This shows the high quality and the better capability of the `fastText` embedding over the `cw` embedding.

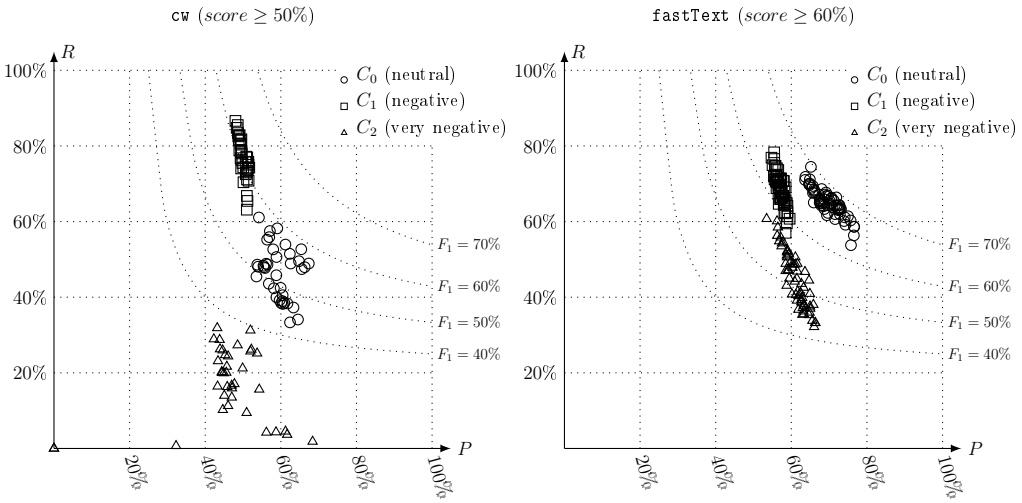

Figure 5.3: Precision P against recall R for all "good" models with the same embedding. Left: `cw` and a score $\geq 50\%$, Right: `fastText` and a score $\geq 60\%$.

5.3.2 Models

Before discussing each model type in more detail some remarks that apply to all of them.

Shuffling Data and Batch Sizes: The optimization algorithm used is `Adam` (see: section 2.1.3). The batch size was limited with `batch_size` samples per batch. The term "limited" is used because the sentences were grouped by their length and then each group was split into batches with a maximum size of `batch_size` samples. Furthermore after each epoch the groups where shuffled and new batches were created. This way for each epoch the batches consisted of different samples which improved learning.[8] The batch size of 64

[8]Most deep learning frameworks or libraries, including the `python` library `Keras` that was used here, support such a feature, it is even the default. But to use data shuffling all samples need to be of equal size which is not the case. The most common approach to this problem is to pad the sequences for training and use masking. Sadly (at least in the version of `Keras` used) not all model structures support masking. Furthermore, for the actual classification of the political speeches with roughly $2 \cdot 10^6$ sentences the same data generator can be easily adapted to speed up processing significantly.

turned out to be a good choice and was used as the value for the `batch_size` for most model training, except when the choice of 64 as batch size was validated. In comparison to [Rudkowsky et al., 2017] a batch size of 100 was used which is of the same magnitude and behaves similar. When increasing or decreasing the batch size about a magnitude, then some models will not be trained as well.

Overfitting: A detailed analysis about the impact of dropout and constraints for the model performance is hard because of the size of the given dataset. For example see figure 5.4 where a big GRU without dropout or constraints is trained. One can see that the validation score (solid) starts dropping around epoch 15 but does not drop below 57% until the training set was almost entirely memorized, meaning to reach a score above 95% on the current training set (dashed line). When looking at the precision, recall or F_1 metrics of models trained without any prevention against overfitting the impact to those metrics is lower than some other parameter variations. But that is not a big problem for any of the presented models, the reason is that when applying a dropout in a range of 20% to 40% in all layers the validation scores as well as other metrics stay stable when they reach their limit. Even though the presented example is already an extreme example of overfitting, meaning that smaller or simpler models have almost no problem with overfitting, adding a dropout does not hurt. Additional weight constraints are also valid for avoiding overfitting. For example a maximum weight constraint of 4 does a good job for most models, although it is a bit harder to adjust the constraint value for a peak performance. If the constraint is too hard the performance drops and if it is too weak constraints have no effect.

Precision and Recall: The precision and recall of all the models validated in the grid search does first depend on the used embedding as long as the model itself has a reasonable score as discussed in section 5.3.1 even though the variation in these metrics for different model types is not negligible. But there is a strong correlation between a model's score, its precision and recall and therefore F_1 as well. The basic rule of thumb is that a better score means better precision and recall as well as a higher balanced classifier. Furthermore the metrics relation between classes, for models of high scores, depends also mostly on the used embedding. Models with a very low score have a very strong variation in these metrics, but they should be avoided anyway.

With the above results in mind a more detailed discussion of the different model types is presented below.

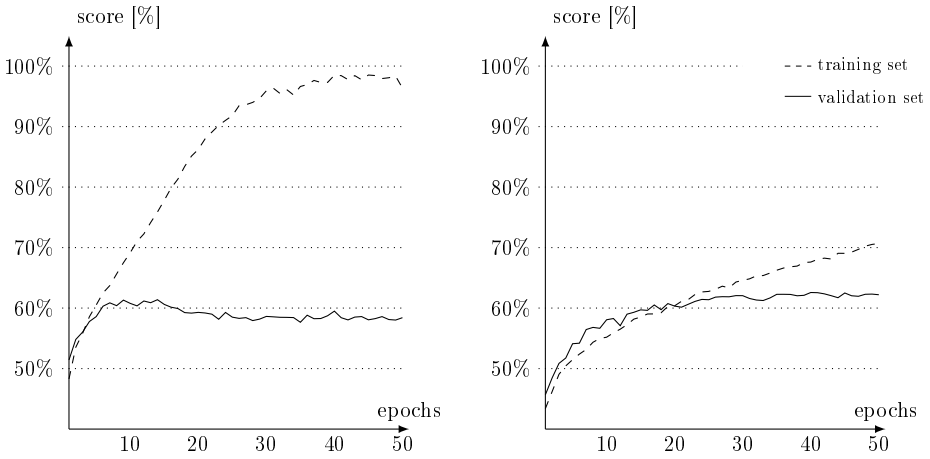

Figure 5.4: Scores over training progress. Left: A GRU model without any prevention against overfitting, Right: The same model with a dropout of 40% and a weight constraint of 4.

MLPs

A simple MLP with a powerful embedding like the `fastText` embedding reaches already up to 58% accuracy in a cross validation. In contrast to a weaker embedding like the `cw` embedding where all MLP scores are bounded by 52%. When comparing these results with [Rudkowsky et al., 2017], a score of 58% was given for the same models using the `cw` embedding. The main difference is the validation of the models. In [Rudkowsky et al., 2017] the models where validated with random sampling while the results presented here are from a cross validation. When validating the exact same model used in [Rudkowsky et al., 2017], denoted as `MTBOW`, with a cross validation on the same dataset, the model has a score of 50%, which is consistent with the results presented here.

There is also a huge difference between the two MLP types. The `MLP (Max)` which performs a max pooling over the word vectors to get a sentence embedding is a very bad choice for a model. The way better version is to build the average of the word vectors as a sentence embedding. This is theoretically reasonable in regard to the word embeddings used.

For the impact of single parameters, the choice of the activation function as ReLU or a hyperbolic tangents is a good one. If is important to have at least one hidden layer, but the difference between a single and multiple hidden layers is surprisingly low. For a comparison of MLPs with `fastText` embedding see figure 5.5.

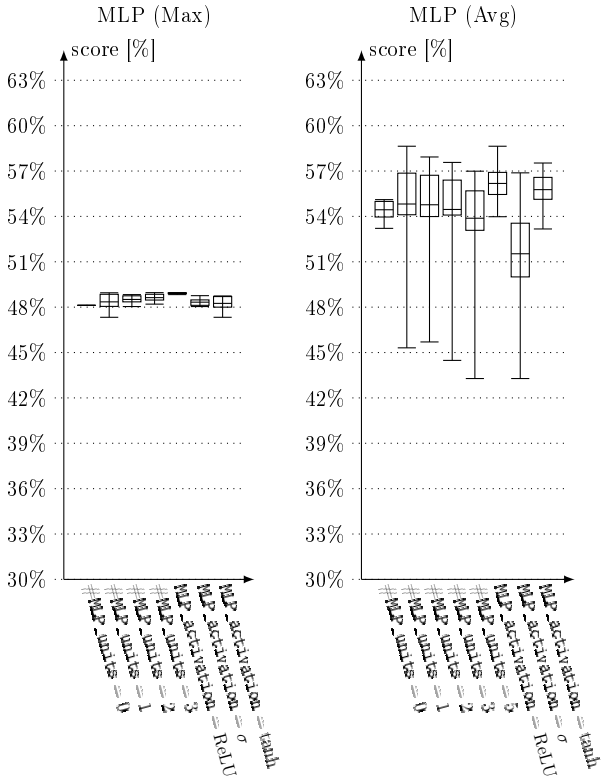

Figure 5.5: Impact comparison of a single parameter of an MLP with `fastText` embedding. Left: MLP (Max), right: MLP (Avg)

CNNs

The CNNs are very good models with up to 61% accuracy. They are definitely better than an MLP and have even outperformed simple RNNs. Furthermore they are very robust against changes in the parameters and the entire group of CNNs is very reliable in their performance when compared to RNNs, meaning that a poorly parameterized CNN still reaches a reasonable score while RNNs do not.

The most important parameter is the activation function in the convolutional layer where a sigmoid function or ReLU is a good choice. Furthermore the type of global pooling is important where the max pooling seems to be superior over the averaging. At this point it is still surprising that the CNNs reach their peak performance with a single convolutional layer. A good choice of the number of features (or hidden layer size) is in the range of 64 neurons. See figure 5.6.

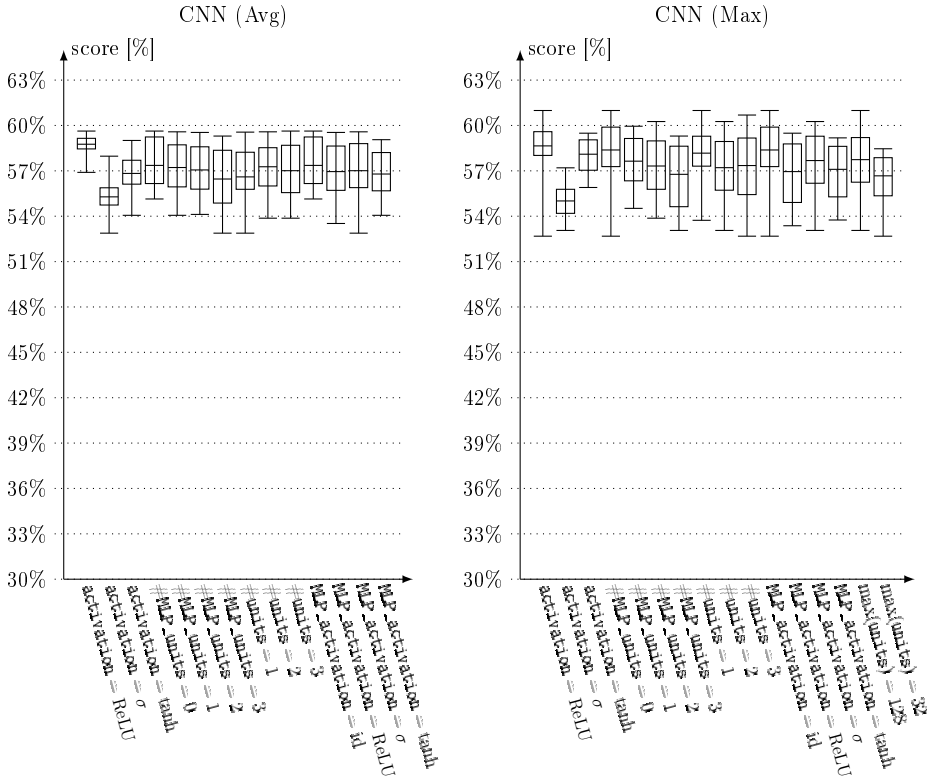

Figure 5.6: Impact comparison of a single parameter of CNNs with `fastText` embedding. Left: CNN (Avg), Right: CNN (Max)

Simple RNNs

The performance of Simple RNNs (also the gated RNNs) is very dependent on a good parameterization as shown in figure 5.7 and still they reach only up to a score of 59%. This is only slightly better than MLPs and they are outperformed by all other model types.

Figure 5.7 shows that the activation function in the recurrent layers needs to be ReLU but they are still unreliable and need to be carefully parameterized for the task on hand.

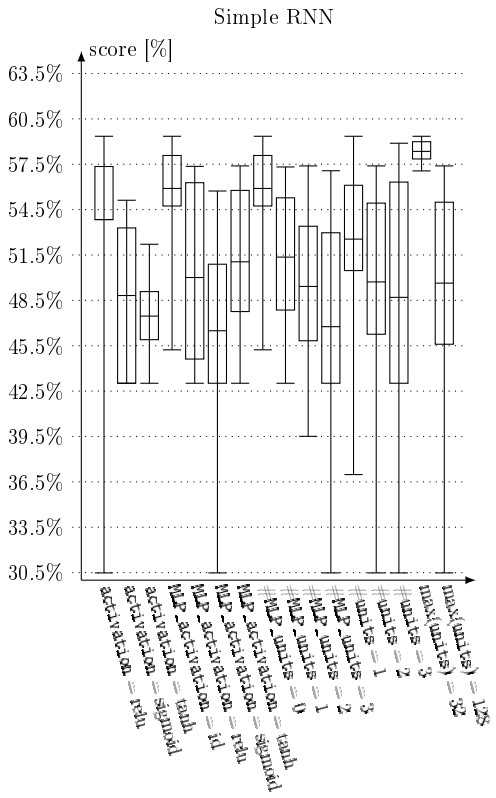

Figure 5.7: Parameter impact comparison for Simple RNNs with `fastText` embedding.

Gated RNNs (LSTMs / GRUs)

The gated RNNs are very well suited for the given task which is not surprising because they are developed exactly for this kind of sequence processing. But like the simple RNNs they are also very dependent on a good parameterization. However, with a well parameterized model they outperform the

CNNs and reach up to a score of 63%. Both the GRUs and LSTMs behave alike except that LSTMs are a bit weaker with a highest score of 62%.

Furthermore they have the most extreme case of a single parameter which must be chosen correctly. The recurrent activation function (not to be mixed-up with the output activation in the recurrent layers) should be a `sigmoid` function as seen in the left plot of figure 5.8 in contrast to the right plot where the recurrent activation is fixed to `sigmoid`. Although this is not surprising because the recurrent activation is intended as value for a gating mechanism, therefore a function with a range $[0, 1]$ is the only really useful choice. For the other activation functions ReLU or even a tanh is a good choice. The depth of the models does not change much in their performance. Although, this is probably only because the size of the training dataset is not sufficient to teach a deeper net much more than a shallow one can learn already. In addition to the single layer a number of 32 neurons is already sufficient.

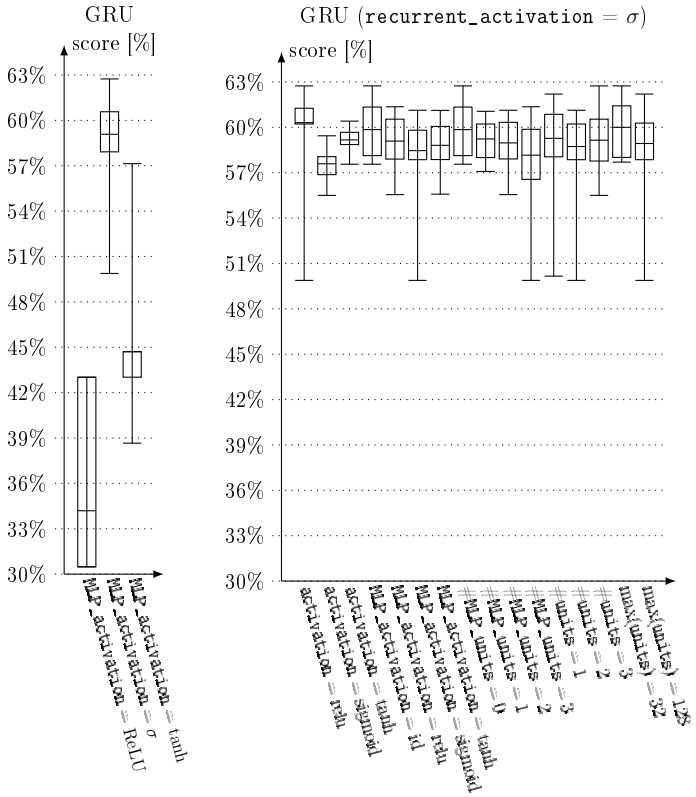

Figure 5.8: Impact comparison of a single parameter of GRUs with `fastText` embedding. Left: All parameters involved in the grid search for a GRU. Right: Additional constraint of $\phi_{rec} =$ `sigmoid` in contrast to left.

Bidirectional RNNs (BiLSTMs / BiGRUs)

The bidirectional RNNs performe exactly like their normal counterparts. They had the same peak performance of 63%, which is the best of the results achieved, as well as in their behavior on other performance matrices. The influence of hyperparameters was also the same. The only difference was that they are a bit slower because of the increased complexity.

CNN - RNN Hybrids

Finally, CNNs with a single RNN layer between the convolutional layers and the final MLP. These models are a CNN stacked with a single layer many-to-one RNN (which can also be bidirectional) and finally an MLP. The models of this type are CNN (SimpleRNN), CNN (BiSimpleRNN), CNN (GRU), CNN (BiGRU), CNN (LSTM) and CNN (BiLSTM). All of them can be directly assessed by taking the results from their not convolutional counterparts.

5.3.3 Qualitative Analysis

For a qualitative analysis a few models where trained with the training set (90% of the entire dataset) and used to classify a wide range of entire political speeches. These were about 58.000 speeches containing about $2 \cdot 10^6$ sentences, see section 1.1.

First the results were visualized for manual validation. The visualization used shows simply the speeches as text where each sentence had its class encoded as background color. When randomly browsing through the speeches one could immediately see a significant bias between the models built on the `cw` embedding and the ones using the `fastText` embedding. Although one could only see which embedding was used, the difference in the results were huge. The models build on the `cw` embedding classified more than half of all sentences as negative or very negative, even though they definitely were neutral (or positive). The models build on the `fastText` embedding labeled most of the sentences as neutral, which they were, and only a few sentences as negative or even very negative. Even though these where just randomly browsing through the visualizations and manually looking at them, one has to say that the performance of the models using the `fastText` embedding in contrast to the `cw` embedding are significantly more reliable which is consistent with the results presented in section 5.3.1.

In addition to the visual comparison the visualization of scored speeches revealed that all models made mistakes in labeling common phrases. For example "`Sehr geehrte Damen und Herren.`" was labeled negative by all

models. This behavior is simply explained by the given training set. A search for such common phrases in the training set showed that, on one hand almost no stand alone phrases are included in the training set, but on the other hand these phrases occurred as parts of negative training samples. But such common phrases occur very often in German political speeches.

As a further example see figure 5.9 where the "average negativity" of all speeches per year are computed from the classification by two models. The first is the MTBOW (left y-axis) model build on the `cw` embedding and the second is a GRU (right y-axis) with `fastText` embedding. There are two interesting things to see. First the course over the years is actually quite alike. Second, there is a huge difference in the bias. To get the "average negativity" in perspective it has to be mentioned that this is the average of a sentence "negativity" from all speeches in a specific year. The sentence "negativity" is set to 0% if the sentence is labeled to be C_0 (neutral), 50% if C_1 (negative) and 100% for C_2 (very negative). With this explained it is not reasonable to assume that the "average negativity" is around 60%. The range of 32% to 36% seems (at least for me) a bit high but not unreasonable.

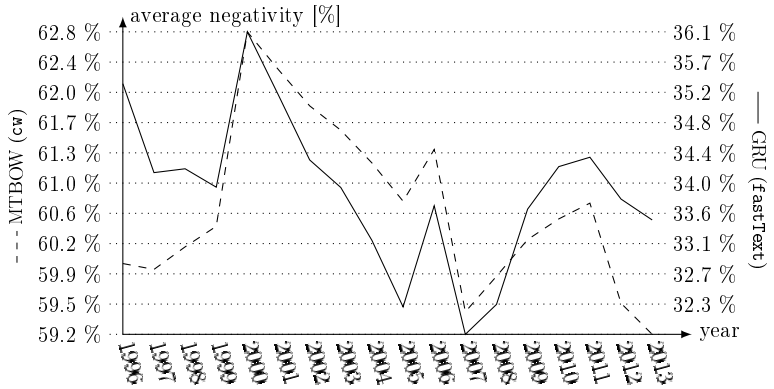

Figure 5.9: Comparison of "Average Negativity" of Speeches over Years labeled by MTBOW (`cw`) and GRU (`fastText`).

5.3.4 Model Fine-Tuning

First the best models of the grid search where chosen and manually fine tuned. This mostly involved training the models with seeded random number generators and altering some parameters by hand as well as letting the trained models classify entire political speeches and visualizing the results. Then finding obvious mistakes and adding these wrongly classified sentences with

a correct label to the training data[9]. For example most salutations where classified negative. This was because most salutations do not occur in the training set but were part of longer negative or ironical sentences as described in section 5.3.3.

Additionally the impact of differences in the data preprocessing was examined. Therefore the data was preprocessed with different configurations, namely to keep or remove either stopwords and/or punctuation from the training and validation data. Then the models where trained and validated, again with a cross validation, with these four differently preprocessed training sets.

The results are listed in table 5.6. Compared to the results from the grid search the performance metrics did not change, the best score of 63% is the same for the fine tuned models as well as the grid search "winners" as well as the accuracy, precision, recall and F_1. But a qualitative analysis of the fine tuned models showed that the fine tuned models made less obvious errors when classifying entire speeches, especially the beginning of speeches which is almost always a greeting of the audience that is no longer classified as negative after augmenting the training dataset with a hand full of common phrases. As mentioned this common phrases are not considered by the given dataset but for possible usage of the classifiers they are crucial. Also the preprocessing configuration for the grid search was to keep stopwords and punctuations which is the same configuration as for the best score presented in table 5.6. These results were also validated with a 10% test set which was split from the entire dataset before starting the model selection process.

To the comparison of the preprocessing, there is not a big variation in altering the preprocessing, not even on a class specific level. Although it seems to be a bit better to keeping tokens and let the nets learn for themselves which tokens are important and which are not rather then presenting the model only the most essential information. But it has to be mentioned that the list of stopwords which was removed was chosen very considerate. As an example the German word `'nicht'`[10] is in the list of stopwords provided from `NLTK` but is a word that must not be removed in our context.

For the precision and recall per class the GRUs and CNNs are quite balanced and both types behave alike for the "middle" class C_1 (negative) but they do behave a bit different on C_0 (neutral) and C_2 (very negative). The GRUs have a higher precision but lower recall on C_0 and for C_2 it is the other way around.

[9]The models where validated with cross validation and therefore even with the augmented training set it is an out of sample validation. Furthermore the number of added samples is about 20 which is 0,1% of the training set.

[10]In English: not

Type	Preprocessing keep punctuation	keep stopwords	score	P	R	F_1
GRU	Yes	Yes	**63.0**	**63.8**	59.7	60.9
GRU	Yes	No	62.7	62.6	**61.1**	**61.7**
GRU	No	Yes	62.7	63.0	60.4	61.3
GRU	No	No	62.2	61.0	60.5	60.3
CNN (Max)	No	Yes	61.8	62.0	59.3	59.9
CNN (Max)	Yes	Yes	61.7	61.9	59.9	60.6
CNN (Max)	No	No	61.6	61.0	58.7	58.5
CNN (Max)	Yes	No	61.4	61.5	59.4	60.0

Table 5.6: Comparison of winning models with different preprocessing parameters (score (accuracy), P (precision), R (recall) and F_1 in % as class average)

Chapter 6

Conclusion

Building on the work of [Rudkowsky et al., 2017] my experiments compared different word embeddings and neural network types for sentiment analysis of German natural text in a political context. I compared different classifiers for their performance of classifying single sentences into three categories, namely neutral, negative and very negative. For this comparison different performance metrics were used, starting with the classification accuracy as well as per class precision, recall and F_1 metrics. These performance metrics were gathered via a 3-fold cross validation[1]. In addition the best models were retrained and submitted to a qualitative analysis. Finally the performance impact on altering the preprocessing in keeping or removing stopwords and punctuations was analized for the best models.

The four German word embeddings compared were `cw`, `sgns` and `ue` provided by the `Polyglot` library [Al-Rfou et al., 2013] and the `fastText` embedding [Bojanowski et al., 2016; Grave et al., 2018; Joulin et al., 2016; Mikolov et al., 2018]. The experiments showed a strong influence on the peak performance of the resulting classifiers depending on the embeddings used for all metrics. The scores reached by the best models of an embedding were 55% for the `cw` embedding, next the `sgns` and `ue` embeddings have models with 58% accuracy but by far the best is the `fastText` embedding with 63%. A comparison on a class level showed that the classifiers using the `cw` embedding tended to have a bias towards the biggest and "central" class, namely the negative class. This effect was also discovered in classifiers using the other embeddings but not so strong. The well performing classifiers using

[1] In [Rudkowsky et al., 2017] the models where compared using random sampling resulting in higher scores than the ones presented in this thesis. When evaluating the MLP model presented in [Rudkowsky et al., 2017] with cross validation (50% accuracy with `cw` embedding which was the embedding used) the results were consistent with the results presented here.

the `fastText` embedding showed almost no bias towards a single class.

The differences between the model types were smaller[2], for example a MLP using the `fastText` embedding reaches a score of 59% which is better than all other models using any other embedding. But still, using better suited model types increases the classifiers performance for all embeddings. Results showed that CNNs are definitely better than MLPs but they are outperformed by gated RNNs. Surprisingly gated RNNs with a single layer are already amongst the best performing models sharing their place with bidirectional gated RNNs and CNN/RNN combinations. Basically for the given task with respect to the given dataset a GRU or LSTM with a single layer is the model of choice.

Altering the preprocessing to remove or keep stopwords and punctuations showed no real impact. Although it seems a bit better to keep everything and let the models learn for themselves what is important.

Finally, a qualitative comparison of some models verified the results and even suggested a stronger impact than the numerical results. Especially when comparing classification of entire speeches where the winning models using the `fastText` embedding, with some fine-tuning, showed way better results in comparison to other embeddings. The main problem was the strong bias towards the negative class which resulted in speeches classified almost entirely as negative. Using the `fastText` embedding this effect did not occur and the main part (neutral sentences) of a speech was classified neutral, as it should be.

[2]Excluding the theoretically and experimentally useless MLP (Avg) model type.

Bibliography

Al-Rfou, R., Perozzi, B. and Skiena, S. [2013], Polyglot: Distributed word representations for multilingual nlp, *in* 'Proceedings of the Seventeenth Conference on Computational Natural Language Learning', Association for Computational Linguistics, Sofia, Bulgaria, pp. 183–192.
URL: *http://www.aclweb.org/anthology/W13-3520*

Bird, S., Klein, E. and Loper, E. [2009], *Natural Language Processing with Python*, O'Reilly Media.
URL: *http://www.nltk.org/book/*

Bojanowski, P., Grave, E., Joulin, A. and Mikolov, T. [2016], 'Enriching word vectors with subword information', *CoRR* **abs/1607.04606**.
URL: *http://arxiv.org/abs/1607.04606*

Cho, K., van Merrienboer, B., Gülçehre, Ç., Bougares, F., Schwenk, H. and Bengio, Y. [2014], 'Learning phrase representations using RNN encoder-decoder for statistical machine translation', *CoRR* **abs/1406.1078**.
URL: *http://arxiv.org/abs/1406.1078v3*

Chollet, F. et al. [2015], 'Keras', `https://keras.io`.

Duchi, J., Hazan, E. and Singer, Y. [2011], 'Adaptive subgradient methods for online learning and stochastic optimization', *J. Mach. Learn. Res.* **12**, 2121–2159.
URL: *http://dl.acm.org/citation.cfm?id=1953048.2021068*

Goldberg, Y. [2015], 'A primer on neural network models for natural language processing', **57**.

Goodfellow, I., Bengio, Y. and Courville, A. [2016], *Deep Learning*, MIT Press.
URL: *http://www.deeplearningbook.org*

Grave, E., Bojanowski, P., Gupta, P., Joulin, A. and Mikolov, T. [2018], Learning word vectors for 157 languages, *in* 'Proceedings of the International Conference on Language Resources and Evaluation (LREC 2018)'.

Hochreiter, S. and Schmidhuber, J. [1997], 'Long short-term memory', *Neural computation* **9**, 1735–80.

Joulin, A., Grave, E., Bojanowski, P. and Mikolov, T. [2016], 'Bag of tricks for efficient text classification', *CoRR* **abs/1607.01759**.
URL: *http://arxiv.org/abs/1607.01759*

Levy, O., Goldberg, Y. and Dagan, I. [2015], 'Improving distributional similarity with lessons learned from word embeddings', *Transactions of the Association for Computational Linguistics, 3:211–225. CC Liebrecht, FA Kunneman, and APJ* .

Mikolov, T., Chen, K., Corrado, G. and Dean, J. [2013], 'Efficient estimation of word representations in vector space', *CoRR* **abs/1301.3781**.
URL: *http://arxiv.org/abs/1301.3781*

Mikolov, T., Grave, E., Bojanowski, P., Puhrsch, C. and Joulin, A. [2018], Advances in pre-training distributed word representations, *in* 'Proceedings of the International Conference on Language Resources and Evaluation (LREC 2018)'.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. and Dean, J. [2013], 'Distributed representations of words and phrases and their compositionality', *CoRR* **abs/1310.4546**.
URL: *http://arxiv.org/abs/1310.4546*

Mitchell, T. M. [1997], *Machine Learning*, McGraw Hill.

Pennington, J., Socher, R. and Manning, C. D. [2014], Glove: Global vectors for word representation, *in* 'Empirical Methods in Natural Language Processing (EMNLP)', pp. 1532–1543.
URL: *http://www.aclweb.org/anthology/D14-1162*

Řehůřek, R. and Sojka, P. [2010], Software Framework for Topic Modelling with Large Corpora, *in* 'Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks', ELRA, Valletta, Malta, pp. 45–50. `http://is.muni.cz/publication/884893/en`.

Rudkowsky, E., Haselmayer, M., Wastian, M., Jenny, M., Štefan Emrich and Sedlmair, M. [2017], More than bags of words: Sentiment analysis

with word embeddings, *in* 'Communication Methods and Measures, Special Issue on Computational Methods'.

Samuel, A. L. [1959], 'Some studies in machine learning using the game of checkers', *IBM JOURNAL OF RESEARCH AND DEVELOPMENT* pp. 71–105.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R. [2014], 'Dropout: A simple way to prevent neural networks from overfitting', *Journal of Machine Learning Research* **15**, 1929–1958.

Zeiler, M. D. [2012], 'ADADELTA: an adaptive learning rate method', *CoRR* **abs/1212.5701**.
URL: *http://arxiv.org/abs/1212.5701*

Appendix A

Appendix

A.1 Abbreviations

ANN ..... Artificial Neural Network
BiGRU ..... Bidirectional Gated Recurrent Unit
BiLSTM ..... Bidirectional Long-Short Term Memory
BiRNN ..... Bidirectional Recurrent Neuronal Network
BP ..... Back Propagation
BPTT ..... Back Propagation Through Time
CBOW ..... Continuous Bag Of Words
CNN ..... Convolutional Neural Network
CV ..... Cross Validation
GD ..... Gradient Descent
GRU ..... Gated Recurrent Unit
GS ..... Grid Search
IDF ..... Inverse Document Frequency
i.i.d ..... identical independent distributed
LSTM ..... Long-Short Term Memory
MLP ..... Multi Layer Perceptron
MSE ..... Mean Square Error
OOV ..... Out Of Vocabulary
RNN ..... Recurrent Neuronal Network

SGD ..... Stochastic Gradient Descent

SVD ..... Singular Value Decomposition

TF-IDF ..... Text Frequency Inverse Document Frequency

TF ..... Text Frequency

A.2 Mathematical Symbols and Functions

A^{-1} ..... Inverse of Matrix A

A^T ..... Transpose of Matrix A

$A \odot B$ ..... Hadamard Product: Element-wise product of A and B

$f \circ g$ ..... Concatenation operation: $(f \circ g)(x) = f(g(x))$.

$f * g$ ..... Convolution of the functions f and g

$\|.\|$ ..... Euclidean or L^2 Norm depending on the context

$\delta_{i,j}$ ..... Kronecker delta: 1 if $i = j$ and 0 otherwise

$\mathbb{E}$, $\mathbb{E}_x$ ..... Expectation or Expectation with respect to x

Var ..... Variance

Cov ..... Covariance

∇f, $\nabla_x f$ ..... Gradient of f and Gradient of f with respect to x

$\mathcal{L}$ ..... Likelihood function / per sample loss

J ..... Objective function for training

$U(a, b)$ ..... Uniform Distribution in the interval $[a, b]$

$\mathcal{N}(\mu, \Sigma)$ ..... Gaussian Distribution with mean μ and covariance Σ

σ ..... Logistic sigmoid function

ReLU ..... Rectified Linear Unit

softmax ..... Softmax function

A.3 Convolution and Cross Correlation

The *convolution* is an operation on two real valued functions $g, k : \mathbb{R}^n \to \mathbb{R}$ defined as

$$s(t) = (g * k)(t) = \int_{\mathbb{R}^n} g(t - \tau)k(\tau)\, d\tau$$

and the "output" of the convolution s is again a function $\mathbb{R}^n \to \mathbb{R}$. In the context of convolutional networks the first argument is often donated as the *input* (in this case g) and the second as the *kernel* (in our case k). The function s, meaning the output of the convolution, is sometimes called the *feature map*.

The discrete version is given as

$$s(t) = (g * k)(t) = \sum_{\tau \in \mathbb{Z}^n} g(t - \tau)k(\tau).$$

Now the feature map s is a discrete functions in $t \in \mathbb{Z}^n$. For actual computational purposes we have to use a bounded domain for our functions s, g and k.

The *Cross Correlation* between two functions is very similar to the convolution. Let $\star$ donate the cross correlation operation which is defined as

$$(f \star g)(t) = \int_{\mathbb{R}^n} g(t + \tau)k(\tau)\, d\tau.$$

So the only difference between cross correlation and convolution lies in the direction of the shift. This leads to the *flipping operation* for the kernel defined as

$$\tilde{k}(t) = k(-t)$$

where $\tilde{k}$ donates the flipped function k. With the flipping operation a relationship between the convolution and the cross correlation is given by

$$(g \star k)(t) = \int_{\mathbb{R}^n} g(t + \tau)k(\tau)\, d\tau = \int_{\mathbb{R}^n} g(\tau)\tilde{k}(t - \tau)\, d\tau = (g * \tilde{k})(t),$$

which is an important relation in the context of CNNs because it is used in the back propagation algorithm. The discrete version of the cross correlation is analog to the convolutional discrete version.

A.4 Example: Polynomial Regression

As an example we present the *polynomial regression* algorithm (an extension of linear regression). There we want to model a relation of $x \in \mathbb{R}$ to $y \in \mathbb{R}$ via a polynomial model $f_d(x; \theta) = y$ with degree $d \in \mathbb{N}_0$. Therefore we have a dataset $X = (x_i)_{i=1}^m \subset \mathbb{R}$ with target values $Y = (y_i)_{i=1}^m \subset \mathbb{R}$ defining an empirical estimate of the true data distribution we want to model. We assume that the given data is i.i.d. drawn from a data-generation process with noise described via $Z \sim \mathcal{N}(0, \sigma^2)$.

$$f_d(X; \theta) = Y + Z$$

Our polynomial model f_d is given as

$$f_d(x;\theta) = \sum_{j=0}^{p} \theta_j x^j$$

with model parameters $\theta \in \mathbb{R}^{d+1}$. With the zero mean of the noise the likelihood for a parameterization θ of a model f_d is given by

$$\mathcal{L}(\theta) = p(Y|X;\theta) \overset{iid}{=} \prod_{i=1}^{m} p(y_i|x_i;\theta) = \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(f_d(x_i;\theta)-y_i)^2}{2\sigma^2}\right)$$

This leads to the negative log-likelihood of

$$-\log \mathcal{L}(\theta) = \frac{m}{2}\log(2\pi\sigma^2) + \frac{1}{2\sigma^2}\sum_{i=1}^{m}(f_d(x_i;\theta)-y_i)^2.$$

Minimizing the negative log-likelihood (maximizing the likelihood) is equivalent to minimizing the mean square error (MSE)

$$MSE(\theta) = \frac{1}{m}\sum_{i=1}^{m}(f_d(x_i;\theta)-y_i)^2$$

which is the classical approach for polynomial regression. This is a quadratic minimization problem for the parameters θ which can be explicitly solved. But before solving the minimization problem the MSE will be rewritten in matrix notation which is more compact and better suited for numerical computation. To do so define

$$X_d = \begin{pmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^d \\ 1 & x_2 & x_2^2 & \cdots & x_2^d \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_m & x_m^2 & \cdots & x_m^d \end{pmatrix}$$

which is a matrix $X_d \in \mathbb{R}^{m\times(d+1)}$ as well as $Y \in \mathbb{R}^m$. In matrix form the MSE can now be written as

$$MSE(\theta) = \frac{1}{m}(X_d\theta - Y)^2.$$

Differentiation by θ and setting zero for solving the minimization

$$0 \overset{!}{=} \frac{d}{d\theta}MSE(\theta) = \frac{2}{m}(X_d\theta - Y)^T X_d.$$

Solving for θ leads to the explicit solution

$$\theta = (X_d^T X_d)^{-1} X_d^T Y.$$

The higher the degree d the more complex relations can be modeled with f_d. But if the models capacity is to high it tends to be very well fit to the given training data ignoring the underlying relation for generalization as shown in 5.4 where the data consists of 8 data points with quadratic relation with gaussean noise. One sees that the linear regression ($d = 1$) is to week to model the relation, the quadratic ($d = 2$) does a good job but is not perfect while it will do a good job for new data as well. But the latter fitted with a polynom of degree $d = 8$ perfectly fits the 8 data points but is not suited for prediction of a quadratic relation.

A.5 Example: VC Dimension for Perceptron with Sinus Activation

In this section we take the perceptron defined in section 3.1 with an alternative activation function, namely $\phi(x) = H(\sin(x))$. This is still a binary classifier with the same number of parameters. We get a model $f : \mathbb{R}^n \to \{0, 1\}$ with weights $w \in \mathbb{R}^n$ and bias $b \in \mathbb{R}$ as

$$f(x; w, b) = H(\sin(w^T x + b)).$$

For this model the VC dimension is ∞. We will show this explicitly, therefore take any $m \in \mathbb{N}$ fixed and let $X = (x_i)_{i=1}^m$ be a configuration of data points in $\mathbb{R}^n$ such that $x_i = (4^{-i}, 0, ..., 0)$. Now we show that there exist weights w and bias b to let our model classify any binary labeling $Y = (y_i)_{i=1}^m \subset \{0, 1\}^m$ of X be correctly classified. To do so we simple define such weights and biases and show the correctness of the classification for an arbitrary but fixed labeling Y. Set the bias $b = 0$ and the weight vector $w = (w_1, 0, 0, ..., 0)$ with

$$w_1 = \pi \left(1 + \sum_{j=1}^{m} (1 - y_j) 4^j \right).$$

So we ignore everything except the first component.

Now a case study:

1. case: Let $x_i \in X$ be such that its label is zero, meaning $y_i = 0$. Then

$$\begin{aligned} w^T x_i &= \pi\Big(1 + \sum_{j=1}^{m}(1-y_j)4^j\Big)4^{-i} \\ &= \pi\Big(1 + \sum_{1\le j\le m : y_j = 0} 4^j\Big)4^{-i} \\ &= \pi\Big(\underbrace{4^{-i} + \sum_{0<j<i : y_j=0} 4^{j-i}}_{S} + 1 + \underbrace{\sum_{i<j\le m : y_j=0} 4^{j-i}}_{\in 4\mathbb{N}_0}\Big) \end{aligned}$$

With $i > 0$ one gets bounds for S by

$$0 < 4^{-i} \le S = 4^{-i} + \sum_{0<j<i : y_j=0} 4^{j-i} < 4^{-i} + \sum_{i=1}^{\infty} 4^{-j} = 4^{-i} + \frac{4}{3} - 1 < 1.$$

It follows with appropriate $k \in \mathbb{N}_0$ that

$$\pi(2k+1) < \pi(S + 2k + 1) = w^T x_i < \pi(2k+2)$$

and therefore $\sin(w^T x_i) < 0$ which gives $f(x_i; w, b) = 0 = y_i \ \forall i : y_i = 0$.

2. case: For $x_i \in X$ with label 1, meaning $y_i = 1$ one gets

$$w^T x_i = \pi\Big(\underbrace{4^{-i} + \sum_{0<j<i : y_j=0} 4^{j-i}}_{S} + \underbrace{\sum_{i<j\le m : y_j=0} 4^{j-i}}_{\in 4\mathbb{N}_0}\Big)$$

which gives

$$2k\pi < \pi(S + 2k) = w^T x_i < \pi(1 + 2k)$$

leading to $\sin(w^T x_i) > 0$ which gives $f(x_i; w, b) = 1 = y_i \ \forall i : y_i = 1$.

Appendix B

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources and resources and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Date, Signature