HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

Transfer Learning for Short Text Classification Applications

Bachelor Thesis

by

Leon Dummer

student number: 2285386
born in
Schwelm

submitted to

Linguistic and Information Science
Chair of Prof. Dr. Laura Kallmeyer
Heinrich-Heine-Universität Düsseldorf

21 December 2018

First Supervisor: Dr. Behrang Qasemi Zadeh
Second Supervisor: Univ.-Prof. Dr. Laura Kallmeyer

Abstract

Inductive transfer learning is widely known to machine learning practitioners as a great means of improving model performance for computer vision related problems. This year Howard and Ruder (2018) proposed a method to effectively use inductive transfer learning for any task in the field of natural language processing. In this thesis, I investigate their methods, specifically on short-texts, as these have always been strongly relying on good feature engineering and external feature extraction. During my investigation, I found out that inductive transfer learning is indeed a viable method for text based tasks, using Howard and Ruder's fine-tuning methods, and does not necessarily need to be performed with two different datasets. Anyhow, it does not inevitably present a new state-of-the-art performance on short-text-classification tasks. Furthermore, I propose a method to further improve results on imbalanced datasets.

The relevant Github repository can be accessed here.

Acknowledgments

I'd like to express my gratitude to my supervisor, Dr. Behrang Qasemi Zadeh, without whom this thesis would not have been written. He has inspired me to learn more about machine learning and challenged me in my writing, to help me improve, where improvement was highly needed. Also, I thank him and the fast.ai community for all the help provided in understanding the general concepts of machine learning.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Transfer Learning

Usually, when training a machine learning model, a task-specific dataset is necessary, which consists of training examples and their labels. Subsequently, features specific to this dataset/task are defined, optimized (through a so-called feature engineering task - more details about this are given in Chapter 2.1.5), and then used to train a machine learning model. But these trained models and engineered features can not be reused for different tasks. A model trained on the classification of emotions in text documents (e.g., articles, tweets, etc.), can not classify text documents by their topics (e.g., as news, sport, literature, etc.).

However, engineering new features and training a model from scratch for each new task is time consuming. For the classification of a text, the model first must learn to understand the basic components (which could be semantics, syntax, context, etc.) of that text/the language it is written in. But, if this information can be reused, not only time can be saved on training, but maybe even the overall accuracy of such a classification system can be raised (Pan et al. 2010, Chp. 3.1). Also, if engineered features and models for one task could be reused (and the information structure that they carry) in another task, not only could time and effort be saved for developing a new machine learning model, but also, perhaps, the overall performance (e.g., accuracy) of the new model could be boosted (Pan et al. 2010). Fortunately, these methods (i.e., reusing learned models and features) are available and can be used across tasks: transfer learning, an idea which was originally proposed and used in the area of image recognition, as exemplified below.

For example, if there is a dataset with thousands of images of flowers, labelled with their names as training data and this is used to train a machine learning model to learn these labels, the learned model can classify and label flower images with their names, perhaps with a high accuracy, but it will not be able to label any other objects in images with their names (e.g., cats as Persian, Ragdoll, etc.). However, it has been proposed (e.g., see Oquab et al. (2014)) that the model trained on the set of flower images has captured and encoded certain knowledge regarding general shapes of objects in the model that it has learned. In turn, this model can be reused in tasks other than flower image classification (e.g., the cat images), by transferring the knowledge of the objects to a new task in the form of features. Yosinski et al. (2014) even suggest that, even though transferring features lowers the overall accuracy with increasing distance between source and target task, it is still better than using random features (Yosinski et al. 2014). The scheme that I exemplified above is the core idea behind the transfer learning investigated in this thesis: To use the learned representations and encoded knowledge in machine learning models, obtained out of efforts for developing a model fitted for one task and to apply it to another task.

In order to understand the concept of transfer learning in more detail, I have to explain its components first. As described before, features and labels are crucial for a model, these two combined constitute the training data. Features describe the object to be classified as numbers in a vectorized form. For instance, if I want to create features for a text object, I could take the length of the text in words, the number of nouns, number of verbs, and other properties as respective features. The vector would then consist of these counted values and the model would try to understand texts based on these counts. The labels are a vectorized variation of the results for each feature vector. For a classification task, the labels could be numbers representing each possible class in the dataset. Datasets with features and labels are defined as a domain from now on. To give the model the ability to learn from these feature and label vectors, predictive functions are necessary. Predictive functions can be a combination of varying algorithms that change the weights of neurons (neurons represent the learned information in form of numbers/weights), based on the difference between the expected results and the model generated results. This whole system is described in more detail in Chapter 2.1.5.

Three different approaches of transfer learning currently exist. The main difference between these approaches consists in their usage of training data and in the existence of labels in the training data. In transfer learning, a source domain and a target domain are always necessary. The basics of those three approaches of transfer learning can be described as follows:

Inductive Transfer Learning helps to improve the learning of the predictive function for the target domain with a dataset of a source domain. The source task is unequal the target task, but they are related. It is possible to do this form of transfer learning with or without labels in the source domain.

Transductive Transfer Learning helps to improve the learning of the predictive function for the target domain with a dataset of a source domain and some unlabeled target domain data. The source task is equal to the target task. The source domain and the target domain are unequal, but related.

Unsupervised Transfer Learning helps to improve the learning of the predictive function for the target domain with a dataset of the source domain. The source task and domain are unequal to those in the target, but related. Source domain and target domain do not have any labels.

(Pan et al. 2010)

1.2 Transfer Learning for Short-Text Classification

In this thesis, I applied transfer learning for three different classification tasks, one of which is the classification of text into emotion, but how can a text be classified into emotions? One might look for certain emotional words like "happy", "sad" or "angry", and use this as a simple method for classification. But, what happens if these words are used ironically or not at all? For tasks that are more complex than the application of simple rules, machine learning can be used. In Chapter 2.1.2 I explain in more detail, how machine learning works and why it is a good approach at solving complex tasks. But machine learning is no miracle system that always delivers perfect solutions, in fact, even for tasks where the system just needs to make binary decisions, researchers could not achieve completely error free results yet[1](Loukides 2017). For testing the effectiveness of machine learning models, a multitude of measures can be used, one of which is accuracy. More details about a variety of commonly used measurements can be found in Chapter 2.1.5.

Short-text classification is an especially difficult task to perform with high accuracy, because the model can not learn proper semantic and syntactic properties of this kind of texts, as they are send in chats, tweets, and similar short-text forms. Often a lot of words are omitted and/or the meaning can only be understood, based on previous context. Thus, the model is not able to learn proper syntax and semantics, because of missing information.

1.2.1 Why are Semantics and Syntax Important for Classification?

Natural language contains ambiguities, which can confuse an algorithm in its classification process. For example, if someone tried to classify the emotional state of a person stating "My boss is such a cold person", one might say: the person is sad. But if the training data of a model consists of more examples where the word "cold" has a positive connotation as in "There is nothing better than a cold drink after doing sports.", a model might classify the word "cold" as an indicator for a positive

[1]Which can be seen by the usage of measurements like error rate or accuracy among others in most machine learning related papers.

emotional state. To prevent this kind of error, the model needs to understand in which contexts the word cold has a positive connotation and in which it is negatively connotated.

1.3 Transfer Learning for Natural Language Processing

In a recent paper by Howard and Ruder (2018), transfer learning for natural language processing (NLP) was successfully used for the first time. They found a way to make transfer learning not only usable in NLP, but even found a system, which outperformed six state-of-the-art representative text classification tasks. Their system is named ULMFiT and it is supposed to be applicable to any NLP task (Howard and Ruder 2018).

For this thesis I made use of fast.ai's library (fast.ai 2018) for deep learning, to apply the papers' techniques and train three different short-text classification systems on three different pre-trained language models. One language model is based on the WikiText-103 dataset, created by Merity et al. (2016), the second one, I created based on Kaggle's Sentiment140 twitter dataset (Go et al. 2009) and the third transfers the knowledge from the WikiText-103 dataset as a basis for learning a language model from the Sentiment140 dataset. In Chapter 2.1.2 I explain, why machine learning is required for solving short-text classification tasks, whereas in Chapter 3.1 I describe the method to create a pre-trained language model. Furthermore, I explain the fine-tuning settings for the language model in Chapter 2.1.5 and in Chapter 4, I show the results of using a Wikipedia based language model (WikiText-103) on short-text classification, compared to a short-text based (Sentiment140) and a transfer learning based language model on three different classification tasks. The source code with a few changes can be found at the repository for this thesis: Transfer Learning for Short Text Classification Applications.

Chapter 2

Background, and Related Work

2.1 Background

2.1.1 What is a Classification Problem, and how to Approach it?

What does classification actually stand for? Bird et al. (2009) define classification as "the task of choosing the correct class label for a given input" (Bird et al. 2009, p.221). These class labels can be topics of articles, the content of images, the gender of speakers in audio files, etc. But one input does not necessarily have to be classified with just a single label. For example, an article in a magazine could be political, and philosophical at the same time or an image could contain a dog, and a cat.

Labeling inputs with classes seems to be a trivial task. Humans can read texts or see images and can usually tell, which class might be appropriate for the given input. But letting a human classify a dataset with a billion entries takes more time, than the task might be worth spending on. So how do humans classify inputs? Can these methods be applied to computers? For texts it be might assumed, that certain keywords (e.g., "political party" for the class politics, "soccer, goal" for sports, etc.) give relevant information, regarding the class a text belongs to, like I did in Chapter 1.2 with the emotion keyword example. But in this context we already discovered that this method might work sometimes, but still yields problems because of the use of irony or other stylistics, that change the literal meaning of a word. Certainly the human brain does not just look at keywords to understand the general topic of a text. It might have a deeper understanding of the whole context, background knowledge of certain names (e.g., Kant was a philosopher, Euler a mathematician, etc.) occurring in the text, etc. But it is not possible to manually program all the background information on each famous name, rules for the understanding of context, etc., due to the sheer amount of possible names, and the missing knowledge to describe our understanding of context and other relevant features in programmable rules. An easier way would be, to make a computer learn in a way similar to the way humans learn. To this end, researches created machine learning.

2.1.2 What is Machine Learning?

According to Ng [1], "Machine learning is the science of getting computers to act without being explicitly programmed" (Ng 2018). This quote gives a general idea of why machine learning is the solution to the problem described in Chapter 2.1.1, and what its basic concept is. Machine learning enables a computer to build models based on given data, and to use these models in order to solve a wide variety of tasks, depending on which type of machine learning is used, and also on the given data. But how can a machine learn? Mitchell et al. (1997) describes the learning process like this "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.". We make use of a multitude of concepts to enable the machine to learn, and make use of the learned information. Those concepts are described in the following Chapter 2.1.5.

2.1.3 Overview of the Forms of Machine Learning

Even though different types of machine learning exist (the major three being supervised learning, unsupervised learning & reinforcement learning), I only describe supervised learning in more detail, because it is the type of machine learning I used in this thesis. To shortly sum up unsupervised learning and reinforcement learning: For unsupervised learning a dataset without labels is used as a learning basis. The idea is to let the system infer the properties of the dataset by itself and it is usually applied to more complex tasks Hastie et al. (2009). Just like unsupervised learning, reinforcement learning has no labeled data to learn from. However, reinforcement learning is not about understanding data, but about making actions to reach a specified goal. In the context of supervised learning, a goal is maximization of the value of a measure for correctness in its understanding of data (accuracy, based on the labeled data for example), while the goal in reinforcement learning could be reaching the maximum score in playing a game repeatedly. A reinforcement learning model has no information on what to do in a certain situation and needs to Figure out its possibilities for reaching its goal by trial and error Sutton et al. (1998).

The difference between supervised learning and reinforcement learning can also be described figuratively: Supervised learning is like learning a language while having words from your own language as features and words from the other language as your labels, based on which you create your understanding of the new language. Reinforcement learning is like playing a game without any instructions and you only know that you need to maximize your score or reach the final goal and you play until you figure out what you have to do, to reach that goal.

[1]Former head of Baidu AI Group/Google Brain, and professor at Stanford.

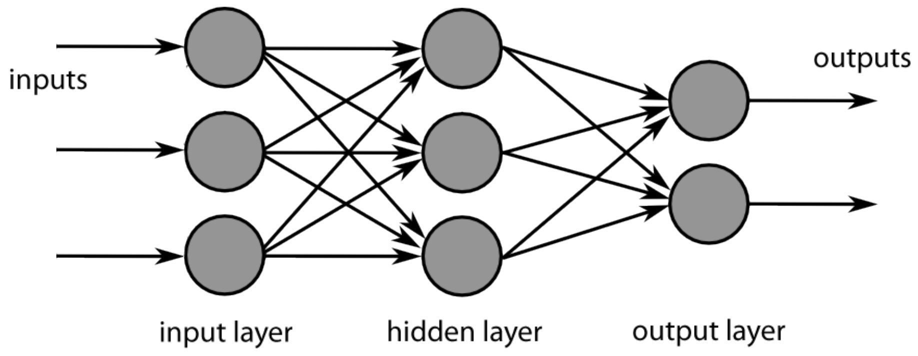


Figure 2.1: Example of a Feedforward Artificial Neural Network.
Each circle represents a Neuron and each line is a connection from Neuron output to Neuron input. Wikipedia, the free encyclopedia (2010)

2.1.4 The Feed Forward Neural Network

Neurons

As a lot of technologies[2], Artificial Neural Networks (ANNs) are inspired by nature, in this case by the brain. Our brains are made up of a lot of neurons, interconnected tiny cells, sending and receiving signals. This exchange of signals enables our thinking. With ANN we try to simulate[3] this structure to create a new approach of solving currently difficult tasks. Each individual Neuron contains a number, called *weight* and has a number of other neurons connected as an input and a number of neurons connected from the output. Even though the human brain consists of 100-120[4] billion neurons (Herculano-Houzel 2009), current ANNs used to be smaller than a human brain, with up to 11.2-15 billion Neurons, but that recently changed with a 160billion neuron sized ANN from Digital Reasoning (2015). However, architectures of that size are not just very slow, due to the large amounts of computations necessary, as can be understood by looking at the functions described in Chapter 2.1.5, but also very expensive to store memory-wise.

Layer Structure

The neurons are set up in multiple layers of a size (Number of Neurons in a layer) dependent on the task. More complex tasks, relying on a deeper level of information need bigger sized layers and

[2]E.g., Velcro, Shinkansen Bullet train's aerodynamic form & electric grid structures.
[3]Hence "Artificial".
[4]Neuroscientists used to think the brain consists of around 100 billion neurons for a long time, but this number was challenged in 2009 and was supposed to need further research. But today it still seems widely accepted to say, that the human brain consists of a 100billion neurons.

simpler tasks with more shallow level of information need smaller sized layers. An exception of this rule are the input and output layers. The input layer is as big as the number of features present in the training data and the output layer's size is equal to the number of individual labels/classes, the inputs should be classified into. In Figure 2.1 an input layer of size three is shown, connected to a hidden layer of size three, which is connected to an output layer of size two (left to right).

2.1.5 General Concepts of Supervised Learning

Before I can explain how supervised learning works in detail, I need to describe some basic concepts that are used in machine learning and are therefore necessary to understand the principles of supervised learning in detail:

Feature Engineering

Feature engineering is probably the most important task for a successful machine learning application, considering the features build the whole basis for the learning process. If they do not provide enough information for the model, the model can not extract any knowledge about its dataset, and therefore is not able to produce accurate classifications. In Chapter 1.1 I described an example for feature engineering in the form of counting nouns, verbs, etc. in a sentence. Typically features are chosen by intuition about which information might be relevant for the current problem. A concrete and simple example would be the problem of gender identification only on the basis of a person's name: Our intuition might tell us that names ending in "a", "e", or "i" are likely to be female and names ending in "k", "o", "r", "s", or "t" are likely to be male (Bird et al. 2009, p. 222). Therefore, to extract the features for this problem, we could reduce our datasets, consisting of male and female names with their corresponding label (male or female), to their ending letter and we would have a one element long feature vector (Bird et al. 2009).

For longer texts researchers made use of a technique called *bag of words*. In this technique each feature represents a single word. The problem with this technique is that words like "jump" and "jumped" or a word written with, and without a capital letter, would be represented with different values. Therefore, researchers usually normalize the analyzed texts by removing elements, such as punctuation or case information. Also, words appearing too rarely and too frequent (e.g., stop words like "the", "and", etc.) are removed, because the information they contain might be too irrelevant (Scott and Matwin 1999).

But the more we normalize the data, the more information we loose. Also, representing each word with a single number simplifies the information contained in a word. Therefore researchers started creating word embeddings, which are vectors representing the linguistic context of words. These vector representations are created by feeding huge text datasets with billions of words and vocabulary sizes around the millions into a Neural Network. Research has shown that these representations can be transformed by simple algebraic operations. For example vector(”King”) - vector(”Man”) + vector(”Woman”) creates a vector representation that is closest to vector("Queen") (Mikolov et al. 2013).

Classification and Regression

Even though I only used classification problems in my examples before, they are not the only kind of problems we can solve with supervised machine learning. Generally supervised machine learning is differentiated into classification and regression problems. While classification tries to predict a discrete[5] value for an input, regression tries to predict a continuous[6] value. Furthermore it is discerned between binary classification (classify the input into a single class out of a set of 2 possible classes), multi-class classification (classify the input into a single class out of a set of more than 2 classes), and multi-label classification (classify one input into multiple classes at once). Similarly, we distinguish between simple regression (one input variable), multivariate regression (multiple input variables), and time series forecasting (input variables ordered by time) (Brownlee 2017).

Performance Measures

As the name suggests, performance measures are used to evaluate the model’s performance. Because some measures are more informative, by means of giving a result more correct to the actual model’s performance, than others, I present a few commonly used measures and their problems, as well as their advantages here.

Terminology

True positive	= Value predicted as class x, when supposed to be class x.
False positive	= Value predicted as class x, when supposed to be class y.
True negative	= Value predicted as class y, when supposed to be class y.
False negative	= Value predicted as class y, when supposed to be class x.

[5]Can only take a certain value. Number of possible values is infinite, but no grey area between two values. Each value is unique. Examples: 2, red, cat, dog.

[6]Can take any value within a range. Possibly infinite amount of other values between two values. Examples: weight (can be in kg, g, normal, obese, etc.), size (can be tall, small, cm, m, etc.

A **Confusion matrix**, or also known as contingency table, can represent a classifiers results with the values shown in the above terminology. An example confusion matrix can be seen in Table 4.5. This matrix is the basis of the following performance measures, as these can be treated as functions, build from the x and y axis of the matrix (Davis and Goadrich 2006).

Accuracy is the number of correct predictions divided by the total number of predictions.
Problem: The accuracy alone does not give any information on how the model performs on a single class set. With a dataset consisting of 95 examples of cats, and 5 examples of dogs, and the model identifying everything as cats, we have 95% accuracy, but the model is not able to identify dogs at all.

Precision is the number of true positives divided by the number of true positives plus false positives. For example, Precision shows how many of the images, classified as containing a cat, do in fact contain a cat.

Recall is the number of true positives, divided by the number of true positives plus false negatives. For instance, Recall shows how many of our images that actually contain cats, were classified as containing cats. Problem: Using the same example as used for Accuracy, with the system classifying all inputs as cats, we get a recall value for cats of 100%, which seems great, but does not reflect how bad the model's predictions are.

F1 Score is the harmonic mean of recall and precision (Equation 2.1).

$$F1Score = 2 * precision * recall/(precision + recall) \tag{2.1}$$

The F1 Score balances the measures of precision and recall in a way, that if one score of them both is extremely low and the other is extremely high, the performance is not 50%, as it would happen with using the average of those two values. For example, precision = 0.9, recall = 0.1:

$$Average = 0.9 + 0.1/2 = 0.5$$

$$F1Score = 2 * 0.9 * 0.1/(0.9 + 0.1) = 0.18$$

Generalization and Overfitting

Generalization is the concept of the model being able to not just classify the data it was trained on, but also to be able to classify yet unseen data correctly. The ability of generalization is important, for the effective use of the trained model. When the training is finished, the model classifies a test set of unseen data and uses the performance measures, comparing the model classifications to the original labels, to test if the generalization works satisfactorily.

Overfitting describes a phenomenon where a model learns only an understanding of the training data, but cannot perform well on any other data. The model does not learn a general concept for this kind of task, but a specific concept just for the current dataset. An overfitted model is not able to generalize satisfactorily. Overfitting can be detected through the usage of *cross-validation*.

Cross-validation splits the dataset into a training and a validation set and makes it possible for the model to calculate these sets' respective losses (loss is described in the following chapter). K-fold cross-validation is a variation of cross-validation, where the original data is split randomly into k% sized parts. The default value for k is usually 10 (Kohavi et al. (1995) suggested using a value between 10-20), so we end up with 10% of the size of the original for validation and 90% of the original[7] for training. Is the validation loss visibly higher than the training loss, then it can be assumed that the model is overfitted, because it performs way better on the training data than on the comparison dataset.

Activation Function, Loss Function, and Backpropagation

In the following I describe the algorithms that enable a model to learn, based on the general concepts explained above.

The Activation Function takes the inputs of a neuron, calculates their sum, weights it, and adds a bias. This value is passed on to the next neuron, also called "firing" of the neuron, just like the signals send by neurons in the brain. Popular activation functions are the *Sigmoid*, *Tanh*, and *ReLU* Function:

$$Sigmoid(x) = \frac{1}{1+e^{-x}} \quad (2.2) \qquad Tanh(x) = \frac{2}{1+e^{-2x}} - 1 \quad (2.3) \qquad ReLU(x) = max(0,x) \quad (2.4)$$

The Loss Function evaluates the performance of a model on the training set, by comparing its prediction outputs with the actual labels. For this step it is important to keep the order of the inputs and outputs, otherwise the model evaluates with wrong labels. The smaller the value returned by the loss function, the better the performance of the model. Commonly used loss functions for classification are *SVMLoss/Hinge Loss* and *Cross Entropy Loss*:

[7]Usually the whole dataset is split into 3 sets: training, validation, and test. The test set contains examples yet unseen by the model, to test the performance on new data. Therefore "original" refers to the training set after splitting the dataset into training and test. From the number of training data examples, k% were used for validation and the rest remained as the training set.

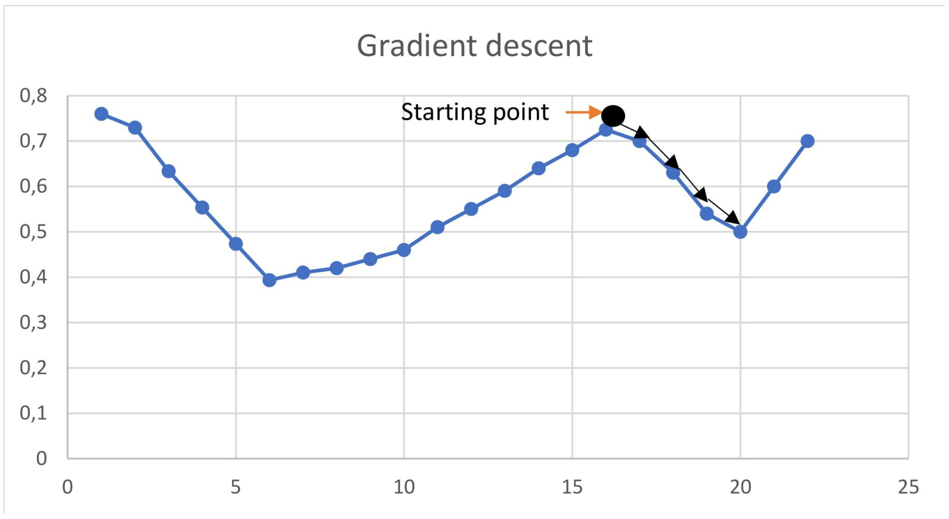


Figure 2.2: Gradient Descent Example
The blue graph shows a function with 2 minima. The black dots represent an example progression of a gradient descent towards a local minimum.

$$SVMLoss_i = \sum_{j \neq y_i} max(0, S_j - S_y i + 1) \tag{2.5}$$

where:

S_j = score of the classifier on example i
y_i = value of the true label for i

$$CrossEntropyLoss = -\sum_{c=1}^{M} y_{o,c} log(P_{o,c}) \tag{2.6}$$

where:

M = number of classes
y = 1 if class c is correct classification for observation o, else 0
p = predicted probability of o for class c

Gradient Descent

The gradient descent is an optimization algorithm, which tries to minimize a function by moving towards the function's minimum through the steepest descent. In Figure 2.2 an example for gradient descent is illustrated. The black arrows represent the gradients, vectors pointing to the direction of the next minimum for each step. In the graph the gradient descent finds a local minimum, but not the global minimum, which would lead to a better performance of the model. This algorithm is used to update the weights in neural networks. The three most popular variations of gradient descent are: *batch gradient descent* (Equation 2.7), *stochastic gradient descent* (SGD) (Equation 2.8), and *mini-batch gradient descent* (Equation 2.9).

The learning rate defines the size of each step of the gradient descent. If the learning rate is set too high, the algorithm might jump over the minimum and never find it. If the learning rate is set too low, the algorithm might take too long/too many steps to find the minimum.

$$\text{Batch gradient descent}(\theta) = \theta - \eta * \nabla_\theta J(\theta) \tag{2.7}$$

$$\text{Stochastic gradient descent}(\theta) = \theta - \eta * \nabla_\theta J(\theta; x^i; y^i) \tag{2.8}$$

$$\text{Mini-batch gradient descent}(\theta) = \theta - \eta * \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)}) \tag{2.9}$$

Where:

$\theta \in \mathbb{R}^d$	= model's parameters
$J(\theta)$	= loss function
$\nabla_\theta J(\theta)$	= gradient of objective function
η	= learning rate
x^i	= training example
y^i	= label

Batch gradient descent (Equation 2.7): Updates the parameters based on a cost function over the whole training set. With bigger training sets, the number of weight changes increase to a point, where large learning rates are necessary for a computation in reasonable time, which in turn leads to overshooting of minima. So for proper minimum convergence on large training sets, small learning rates are obligatory, which increase the amount of computing necessary (Wilson and Martinez 2003). Batch gradient descent converges definitely to a global minimum in convex surface, and to local minimum in non-convex surfaces.

Stochastic gradient descent (Equation 2.8): Updates the parameters for each training example, and label, and is therefore less calculation-intensive than batch gradient descent. Due to the frequent updates with high variance, stochastic gradient descent fluctuates strongly, and therefore does not get stuck in a local minimum, but tends not to converge to a minimum properly at all (Darken and Moody 1991). This can be fixed by using decreasing learning rates, which in turn reduce the fluctuations, and therefore lead to a convergence into a (not necessarily global) minimum.

Mini-batch gradient descent (Equation 2.9): Updates the parameters similarly to stochastic gradient descent, but on a mini-batch base, of sizes usually varying between 50 and 256, instead of on a base of examples and labels.

Problems with the three basic variations are the choice of learning rate, as described before, because too high learning rates tend to miss the minimum, and too low learning rates tend to be too time consuming to converge properly. This is especially true when we want the learning rate to decrease gradually to enable the stochastic gradient descent to converge to a minimum. These changing learning rates are called *learning rate schedules*. Because adaptive schedules[8] are usually too expensive to calculate, learning rate schedules need to be fixed beforehand (Darken and Moody 1991).

Methods to compute adaptive learning rates
Since Darken and Moody (1991)'s paper, computing power increased strongly, while prices for it decreased and algorithms improved, which enabled computing adaptive learning rates effectively as Duchi et al. (2011) prove in their paper, presenting Adagrad. Adagrad is an algorithm that adjusts the learning rate based on the frequency in which certain features occur. Adagrad has one problem though, it can happen that the learning rate shrinks to a point, where the gradient descent is not able to move forward anymore.

$$\text{first moment} = \frac{1}{n}\sum_{i=1}^{n} a_i^1 \tag{2.10}$$

$$\text{second moment} = \frac{1}{n}\sum_{i=1}^{n} a_i^2 \tag{2.11}$$

An alternative for Adagrad is the, nowadays widely used, **Adam** (Adaptive Moment Estimation) (Kingma and Ba 2014). Adam is computationally efficient and thus can handle large datasets. It makes use of the first moment(Equation 2.10)/average and the second moment (Equation 2.11)/variance. It uses four configurable parameters, alpha/learning rate, beta1, beta2, and ε. Alpha is the initial learning rate, while beta1 and beta2 are the exponential decay rates respectively for the first and second moment estimates. Beta1 works best with a default value of 0.9 and beta2 with a default value of 0.999. ε prevents division by zero and should be set to 1e-8 (Kingma and Ba 2014). The whole algorithm is shown at Algorithm 1.

Batches and epochs

Batches are used to separate the input data into smaller chunks of data. This is done on one hand to reduce the memory necessary for the learning process, on the other hand it is a crucial part of

[8]Changing with the changes of the parameters.

Algorithm 1 *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $gt \odot gt$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\varepsilon = 10e^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

```
Require: $\alpha$: Stepsize
Require: $\beta_1, \beta_2 \in [0,1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters $\theta$
Require: $\theta_0$: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow (Initialize 2^{\text{nd}} moment vector)$
  $t \leftarrow 0 (Initialize timestamp)$
  while $\theta_t$ not converged do
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestamp $t$)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \varepsilon)$ (Update parameters)
  end while
  return $\theta_t$ (Resulting parameters)
  (Kingma and Ba 2014, p. 2)
```

the hyper-parameter tuning (more about this is explained in Chapter 2.1.5). This is on the grounds that after each batch the parameters can be updated, depending on the gradient descent function, and therefore, the batch size can determine the amount of updates per training cycle.

An epoch is one full training cycle on the whole set of batches/on the training set.

Regularization

As we now know, generalization is highly important for effective machine learning (see Chapter 2.1.5). To achieve generalization and prevent overfitting, regularization is used. Regularization discourages the model from forming too complex or extreme representations of the current problem. In formulas regularization is usually represented by the parameter λ. Forms of regularization are the learning rate, batch size, dropout, and weight decay (Smith 2018).

Weight decay multiplies the weights of the model in every update (e.g., after each batch) with a value between 0 and 1. This prevents the weights from growing too big, lowers the overall signal strength of each neuron, and therefore reduces the chance of overfitting (Krogh and Hertz 1992).

$$\text{Weight decay:} \quad E(w) = E_0(w) + \frac{1}{2}\lambda \sum_i w_i^2 \tag{2.12}$$

Where:

E_0 = loss function
λ = regularization parameter, describes how strongly large weights are penalized
w = weight vector

Dropout is a regularization technique in which units/neurons are temporarily deactivated and do not receive or send information anymore. This can be done with a simple probability p that can be set to 0.5, giving every neuron a chance of 50% of being active each cycle (Srivastava et al. 2014).

Gradient Clipping is another way to prevent gradients from exploding or vanishing. During training the algorithm checks if the gradients surpass a preset threshold and either sets them to that threshold instead, or calculates a new normalization value over the whole set of gradients.

Hyper-Parameter Tuning

A hyper-parameter is every parameter that we set before the training process. These can be the learning rate, weight decay, batch size, the β values of the Adam optimizer, size of the architecture (e.g., hidden layer size, embedding size, and amount of layers), etc. Looking back at the definitions of all these functions and parameters, it can quickly be seen, that it is crucial to choose the right values for a good performance of the model. These values are usually chosen based on intuition, prior knowledge based on papers by other researchers, or by trial and error (van Rijn and Hutter 2017). By performance does not just include the accuracy in solving a problem, but also the speed in which we can train the model. Methods to automatically tune/choose hyper-parameters do exist, but they are very computationally expensive and time consuming.

Even though "model zoo"'s, which are collections of pre-built models with their corresponding hyper-parameters, exist, it is not recommended to use these as they are (Smith 2018). Every dataset is different and therefore does not perform optimally with the settings of another dataset. For example, if we use a model that was fine-tuned for big multi-class classification tasks with billions of examples and thousands of features, and use this for a small binary classification with ten features, the architecture might be too big and not able to acquire proper knowledge for the task at hand.

Long Short-Term Memory Recurrent Neural Network

So far I have described the algorithms, functions used for processing the inputs, the Neurons, and the concepts of layers. Before I can continue to describe how all those work together to create supervised learning, I have to describe a special kind of architecture, which I used for my experiments, the Recurrent Neural Network (RNN). The RNN is similar to a feedforward neural network as presented in Figure 2.1 by passing the input from layer to layer and applying the algorithms explained in Chapter 2.1.5 to it until an output is produced. However, the Recurrent Neural Network does not just take a single input, but several inputs through time and reuses the information of the previous time step (and therefore all the information added in every time step before the current on) as an addition to its current input. This reuse of previous information enables the RNN to process text word by word, without loosing information of the current word's previous context. To prepare the data for this kind of network architecture, the inputs are ordered in a set of time steps and use backpropagation through time (BPTT) (Elman 1990; Mikolov et al. 2010).

Backpropagation through time (BPTT) is a special form of the backpropagation used in feedforward neural networks. Normal backpropagation takes the final error and passes it back layer by layer, to calculate new weights for each neuron based on the result of the gradient descent to decrease the overall error. BPTT basically does the same thing, but on a time basis. Instead of performing backpropagation after one whole cycle, the backpropagation is performed after each time step, carrying on the weights of the previous τ time steps. After all the error deltas for each time step τ have been calculated, they are folded back up, leading to one big change in weights. Therefore every time step in a RNN can be seen as its own whole layer.

$$\text{BPTT:} \quad \delta_{pj}(t-1) = \sum_{h}^{m} \delta_{ph}(t) u_{hj} f'(y_{pj}(t-1)) \tag{2.13}$$

Where:

h = index for activation receiving node
j = index for activation sending node (one time step back)
t = time step
m = number of state nodes
f = output function

(Boden 2002)

Long Short Term Memory (LSTM) networks are a variation of the regular RNN. The up-folding in BPTT can lead to vanishing or exploding gradients and therefore the LSTM introduces the so called gating. Gating enables the network to temporarily forget inputs and remember them at certain time steps. The whole LSTM module consists of an input, an output, and 3 gates (each gate contains a sigmoid activation function) in between. The inputs of the LSTM (previous hidden state and current input) pass through these 3 gates before they end up at the output:

forget gate	Values close to 0 are forgotten and values close to 1 are kept
input gate	Consists of sigmoid + tanh function. Sigmoid decides which values are updated and tanh decides which values might contain relevant information. Those two functions' vectors are multiplied with each other.
output gate	Consists of sigmoid + tanh function. Sigmoid takes previous hidden state and current input together as new input, tanh function takes calculated output of the previous gates as input.

The input is first multiplied by the output vector of the forget gate and then added to the output vector of the input gate, the resulting value is the new output. That new output vector goes into the tanh function of the output gate and is multiplied with the output of the output gate's sigmoid function and creates the new hidden state. This output vector and hidden state are then passed on to the next time step (Hochreiter and Schmidhuber 1997).

2.1.6 Supervised Learning

In Chapter 1.1 datasets have been discussed and that it is essential to define features for these first. This is not only true for Supervised Learning, but also for Unsupervised Learning. However, unique to Supervised Learning is, that we also need the labels/classes, into which we want the model to classify its inputs, connected to each input. The existence of these labels is why this form of learning is called supervised. The model receives information, about which of the inputs is an example for which class and from those examples, the model can learn how the classes can be defined. This leads us to the question, how can a supervised machine learning model learn from the defined features? In the following I showcase, how the concepts from Chapter 2.1.5 are used to transform the dataset into useful knowledge for the model.

Feature Selection

In Figure 2.3 we can see the general order of a supervised machine learning process. We start with a dataset for the task. The first step for the machine learning practitioner should always be to understand

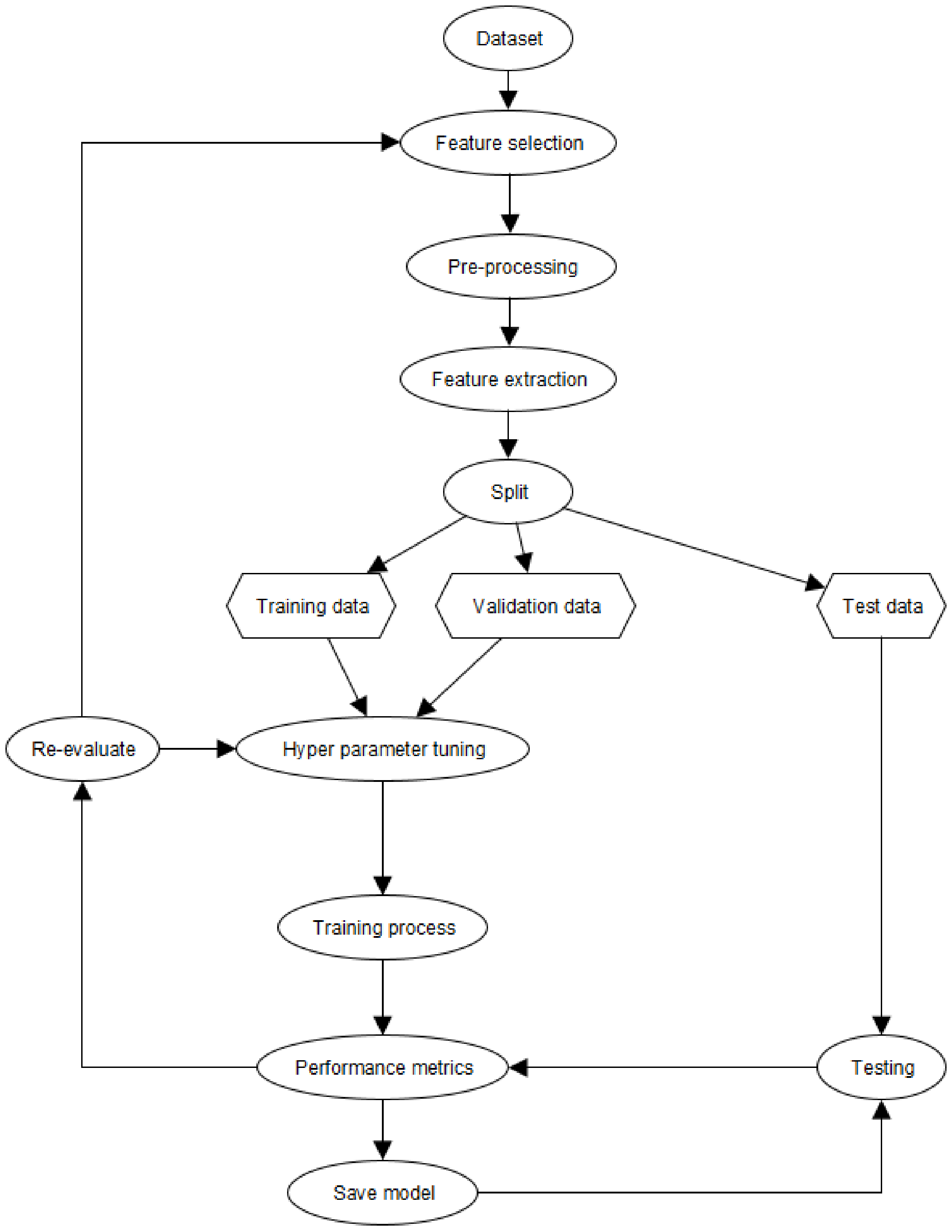


Figure 2.3: Machine Learning Flowchart

the data, the underlying problem, and find the relevant features in the dataset for the system to be able to create a most favorable model. This can either be done by simple intuition or continuous testing of different feature sets, which can turn into a very long process, because the training process' results have to be reviewed repeatedly. Guyon and Elisseeff (2003) suggest a check list of ten steps for feature selection, a few of which are: Check for interdependence of features, is pruning necessary for cost, speed or understanding reasons, create a feature ranking to understand the influence, and sort out useless features, etc. (Guyon and Elisseeff 2003).

Pre-Processing the Dataset

When the first features are selected, the dataset is pre-processed to normalize (e.g., change different currencies, weights etc. to one of a kind, remove redundant information, etc.) it, before we extract the feature vectors. Normalization does not just improve the accuracy of a model, but can also increase the speed of the training process (Sola and Sevilla 1997). For example, if we take a look at a text classification task, where a bag of words method (See Chapter 2.1.5) is used, it becomes obvious, that a single word without prior normalization occupies multiple indices in each vector, due to its several forms (Is, is, are, was, be, etc.). An increasing vector size leads to an increasing amount of calculations for most functions and therefore to increased computing time. But, normalization has to be performed carefully, since crucial information could be lost with too much editing of the raw data. In conclusion we can say that data should be kept as clean as possible, while preserving all crucial information.

Feature Extraction

In the feature extraction we simply transform a dataset into vectors, containing the features we engineered before: One vector for the training examples/input and one vector for the labels/output. It is very important that these two vectors are always in the same order, otherwise the loss function calculates the losses based on wrong targets. If the training data is sorted by its classes, it is highly recommended to shuffle the data first, to prevent mini-batches full of examples of the same class. This simple method can boost the validation accuracy by about 1% (Ioffe and Szegedy 2015).

Furthermore the overall accuracy for classification tasks can be improved by fixing imbalanced datasets[9] through the use of weights (He and Garcia 2008). These weights are applied when calculating the loss and lead to an increase of the error values based on the difference of example counts between classes. This way errors in lower represented classes lead to a bigger overall loss.

[9]Every class is not represented by the same amount of examples.

Splitting the Dataset and Training Process

Next the dataset is split into training, validation, and test sets, as described in Chapter 2.1.5. The training set and validation set are used for the training process directly, while the test set is used to check the learned model on unknown data.

After the first hyper-parameters are set, the training process can start. The following is just one example of an order in which the training process can happen, it might vary from task to task. If a dropout is used, we can apply it immediately to a vectorized input, then the information continues through each hidden layer, in this case LSTM layers. The calculations can be defined like this for each sequence in a single layer:

$$\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{(t-1)} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{(t-1)} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{(t-1)} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{(t-1)} + b_{ho}) \\
c_t &= f_t c_{(t-1)} + i_t g_t \\
h_t &= o_t \tanh(c_t)
\end{aligned} \tag{2.14}$$

Where:

h_t = hidden state at time t
c_t = cell state at time t
x_t = input at time t
i_t = input gate
f_t = forget gate
g_t = cell
o_t = output gate
σ = sigmoid function

After each layer, dropout can be applied again, except if it is the last layer, to prevent removing information from the final output. Afterwards a loss functions is used to calculate the error between the output and the expected labels and backpropagate the new weights based on the loss. If regularization functions like weight decay are used, they need to be applied on the new calculated weights before they are backpropagated. This process is repeated for the amount of epochs that were set. If the model starts overfitting, the training process can be interrupted early. In that case it might be enough to change the hyper-parameters, reset the model, and restart the training. When the training loss, validation loss, and accuracy look satisfactorily, the new model can be run on an unknown dataset, the test data. If the model performs a little worse on the test data, it is to be expected, but if the results perform way worse than in validation, the hyper-parameters may have to be re-evaluated or, more likely, even the correctness of the chosen features and/or the whole dataset itself has to be re-evaluated. The training process in general can be described in a simple form like this:

$$f_0 = argmin_{f \in F}(\frac{1}{n}\sum_{i=1}^{n} Loss(f(v_i), l_i)) \tag{2.15}$$

Where:

f_0	$=$ function that minimizes the error
F	$=$ hypothesis space containing the input vectors and corresponding labels
$f(v_i)$	$=$ model output of input v_i
l_i	$=$ corresponding label
$Loss$	$=$ Loss function
$argmin$	$=$ returns the minimum of a set of functions

(QasemiZadeh 2015, Chp. 2.4)

2.2 Related work

2.2.1 Transfer Learning

As mentioned before, the idea for this thesis is based on the recent paper of Howard and Ruder (2018). They pioneered transfer learning for text classification tasks by training a "universal language model" that is used as a base for further machine learning tasks in the problem field of that language. However, there were approaches of using transfer learning for text related machine learning tasks before. Do and Ng (2006) "meta-learned" a new learning algorithm g from various other text classification problems, which was used on non neural network-based machine learning approaches, which are strongly outperformed in most fields of machine learning research, as Benediktsson et al. (1990) shows. Other famous transfer learning approaches for text related tasks were word embeddings like word2vec by Mikolov et al. (2013) or glove by Pennington et al. (2014). But word embeddings only contain knowledge about the similarity of single words as described in Chapter 2.1.5 opposed to the knowledge of a language as a whole, as Howard and Ruder (2018) ULMFiT model offers.

2.2.2 Short Text Classification

Considering the nature of short texts, classical methods like bag of words only offer limited viability. Therefore Banerjee et al. (2007) propose the addition of extra features from other sources than the short texts themselves, e.g. Wikipedia. Similarly Sriram et al. (2010) makes use of additional features from twitter users for tweet classification tasks, e.g. authorship, participants (tagged users), place, time, etc. Another example of these external knowledge collection approaches is made by Phan et al. (2008): they gather a topically related, large scale data collection for each classification task and add the information from the collection to the training data's own features. All of these approaches assume that the classification on short texts alone works poorly due to data sparsity. However, a newer approach by Zeng et al. (2018) tackles the problem of data sparsity without the use of external knowledge. They invented the "topic memory networks" (TMN), which captures co-occurrence patterns on a document level. Further they use a topic memory mechanism that detects indicative latent topics, which is useful for the classification, and enriches the short texts with these (Zeng et al. 2018).

Chapter 3

Methods

All functions and algorithms used in my code are based on a library provided by fast.ai (2018). Therefore most of the ideas and concepts for pre-processing the data, the training process, etc. are also based on their code.

3.1 Creating a Language Model

Based on the frequent use of abbreviations and the disregard of proper grammar, short texts might be seen as their own language and should thus be treated with their own language model. Therefore as a training set for my English short text language model, I used the Sentiment140 dataset (Go et al. 2009). The Sentiment140 dataset consists of 1,600,000 examples of positive and negative tweets. Due to its size and the length restriction of tweets, this dataset seems ideal as a training basis for my language model. To build a language model from this dataset, I created a machine learning model, that predicts the next word for its current input. In the following sections I explain the steps taken to prepare the dataset and the setup of the model for the training process.

Table 3.1: General Information on the Used Datasets

Dataset information			
Dataset name	Examples	vocab size	token
Sentiment140 (Go et al. 2009)	1,600,000	320,000	22,500,000
WikiText-103 (Merity et al. 2016)	28,500	267,000	103,690,000
EmoContext (Provided by Microsoft (2018))	33,000	14,000	512,000
Hate Speech (Davidson et al. 2017)	25,000	19,000	320,000

Basic information for all datasets used in this thesis. The Sentiment140 dataset is the biggest in numbers of examples, while the WikiText-103 contains a bigger number of tokens. This is due to the nature of the examples. Sentiment140 is made up of tweets, whereas in WikiText-103 every example is a whole Wikipedia article.

3.1.1 Feature Selection

Addition or removal of features might add noise to the model in its task of predicting the next word in a natural short text and is therefore not performed. Each individual word is projected as an index, based on its frequency. Words that appear less than 6 times do not appear in this frequency list and were replaced with the token "_unk_". Also I added a token "_pad_" for words to fill empty indices in fixed length vectors, respectively. Therefore the final vector size for the vocabulary is the number of unique words, appearing more often than six times in the whole dataset plus two.

3.1.2 Text Pre-Processing and Feature Extraction

There is no need for twitter specific content (e.g.,name tags, links, "RT" tag, etc.) in a general short text language model, therefore name tags and links were removed, so all that is left, is bare text. For the vector transformation, a word frequency list was created, as described in Chapter 3.1.2. For this step the texts need to be regularized to a certain degree first, so that words in upper cases and negated words do not get completely different IDs than their normally written counterparts. Therefore each tweet was tokenized with the following rules:

- Repetitions with more than three symbols were replaced with the token "tk_rep" the repetition count and then the single symbol that was repeated.
  Example: *"!!!!" -> "tk_rep 4 !"*

- Word repetitions were similarly marked with "tk_wrep", the repetition count and the repeated word if they were followed by any kind of non-alphanumeric Character.
  Example: *" you!you!you!you!" -> "tk_wrep 4 you!"*

- If a word contained more than one upper-case letter, they were changed to lower-case letters, combined with a token "t_up" preceding the lower-case word, if it contained just one upper-case letter, this letter was simply transformed to a lower-case letter.
  Example: *"YOU" -> "t_up you"* or *"You" -> "you"*

- Hashtags were separated from their connected words to create separate tokens.
  Example: *"#AI" -> "#", "AI"*

- Finally, I used the Spacy Tokenizer (explosion.ai 2016) for the English language to separate words, punctuation, and endings like n't, 's, 'd, etc.
  Example: *"I'd go to Spencer's supermarket if you don't want to" -> ["I", "'d", "go", "to", "spencer", "'s", "supermarket", "if", "you", "do", "n't", "want", "to"]*

With these operations the overall number of individual words is reduced, but the semantic properties remain in form of tokens. To further remove words that have low relevance, all word occurrences with a frequency of less than 6 were removed. Effective vocabulary size after regularization was 41813.

3.1.3 Splitting

I used a regular 10-fold cross-validation, so I ended up with a training set of size 1,440,000 and a validation set of size 160,000. The overall vocabulary size turned out to be 41813 unique words, plus the before mentioned tokens. The sentences are sorted by size[1] to reduce the amount of the "_pad_" tokens necessary in each mini-batch.

3.1.4 Hyper-Parameter Tuning

Architecture: For the model architecture, I chose an embedding size of 400, 3 hidden LSTM layers with a size of 1150 each and a linear output layer. The full architecture is shown in figure 3.3

Other functions: BPTT with a randomized sequence length at a minimum of 45 (minimum of 10 with 5% chance) and maximum of 95 was used, to limit the amount of data on the GPU at one point of time[2]. The randomization of the BPTT length helps regularizing the LSTM and fixes a problem where any element divisible by N with a fixed BPTT length of N has nothing to backprop to (Merity et al. 2017). Let X be a random variable that represents the range [0.0, 1.0], the sequence length can be calculated as shown in Equation 3.1.

$$\begin{aligned} \mu &= \begin{cases} \mu, & \text{if } X < 0.95 \\ \mu/2, & \text{otherwise} \end{cases} \\ seqlen(\mu,\sigma) &= max(\sigma, N(\mu,\sigma^2)) \end{aligned} \tag{3.1}$$

where:

X = random value in the range of {0.0, 1.0}
μ = expectation
σ = standard deviation
N = Normal distribution

[1]Size of a sentence means the number of words contained in it.
[2]The Graphics Processing Unit (GPU) can just handle a limited amount of data at the same time, depending on the size of ram, therefore a limit is necessary.

To reach the maximum of 95 and minimum of 45 I use an expectation value of 70 and a standard deviation of 5. The fixed batch size is 60, so I end up with tensors of the size 60*seq_len. Those can be increased/decreased depending on the available GPU Ram. (Keskar et al. 2016) observed that small batch sizes decrease learning speed, because they decrease the overall amount of computations per iteration and thus reduce effective parallelism. Shallue et al. (2018) proved that increasing batch sizes leads to a reduction of necessary computational steps. Anyhow, batch size can not be increased infinitely to reduce the amount of computation steps respectively. At a certain size (differing from dataset to dataset), the number of computational steps stops decreasing and thus stops improving calculation time. In my case the batch size was chosen, because it is the maximum that fits on a 8GB Ram graphics card, without running out of memory for a dataset of this size and seems big enough to perform well in an acceptable time frame[3]. For gradient descent, I use SGD with a base momentum of 0.9.

Regularization settings: Dropouts are as followed:

- Dropout to apply to the activation's going from one LSTM layer to another: 0.015
- Dropout to apply to the input layer: 0.025
- Dropout to apply to the embedding layer: 0.002
- Dropout to apply to the LSTM's internal or hidden recurrent weights: 0.02

Due to the big number of possible classes/words, very small dropout rates can be used. Duyck et al. (2014) even observed that no dropout at all might perform better for datasets with high amounts of classes and improves training speed. Smith (2018) suggests using small weight decay values to reach super convergence while Smith and Topin (2017) even assume that larger datasets do not need much regularization, because more complex data provides regularization by itself, therefore I did not use weight decay at all. I also make use of gradient clipping with a threshold of 0.12 to further prevent exploding or vanishing gradients.

To choose the optimum learning rate, a function that runs the training process on the dataset and increases the learning rate until the loss starts increasing too strongly or until a fixed number of iterations is passed was used. Fast.ai took the approach for this function from Smith (2017). By considering the function graph, depicted in Figure 3.1, it was decided to choose a learning rate of 10, because it seems to be the highest learning rate before the loss starts increasing/flatting out, even though the optimum learning rate would be around 2, because at this point the loss strongly decreases. Anyways, considering the use of cyclical learning rates, as described in (Smith 2017), the optimal

[3]Overall learning duration was 3.5 hours on a Nvidia Geforce GTX 1080.

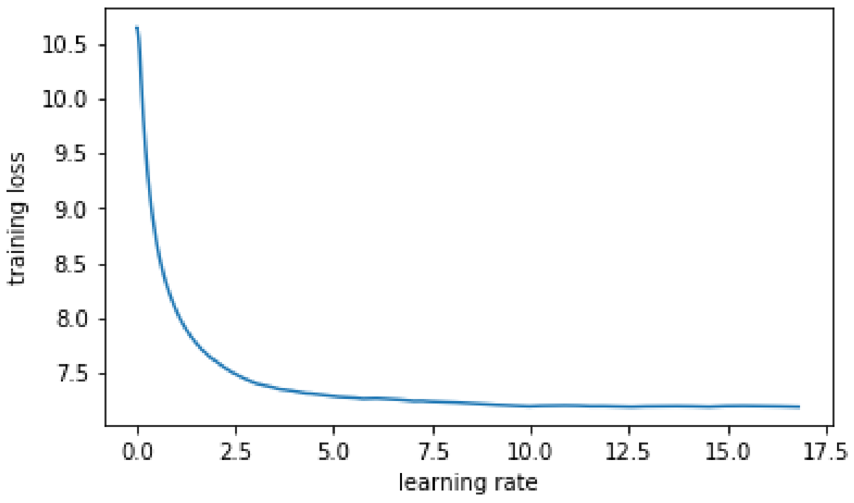


Figure 3.1: Learning Rate Finder
The graph shows the loss with increasing learning rates. From this graph the optimal learning rate can be chosen by looking for the point of the highest learning rate, where the loss still strongly decreases. Which would be a learning rate at about 2 here. However, because cyclical learning rates are used in this paper, 10 was chosen as a maximum learning rate, because it is the highest learning rate, before the loss stops decreasing.

learning rate of 2 is part of the overall learning rate cycle. The following values for the cyclical learning rates were used:

$$\begin{aligned} \text{ratio between start and end learning rate} &= 10 \\ \text{percentage of the cycle spend on the last iterations} &= 10 \\ \text{maximum momentum} &= 0.95 \\ \text{minimum momentum} &= 0.85 \end{aligned}$$

The resulting learning rate and momentum can be seen in Figure 3.2. I trained the language model for 10 epochs. The final perplexity[4] is 64.07. To the best of my knowledge there are no other language models based on this dataset. Current state-of-the-art perplexity on the WikiText-103 dataset is 29.2 by Rae et al. (2018). The big difference might be due to the lack of proper hyper-parameter tuning and/or due to the nature of the dataset (Short-texts: No proper grammar, lots of abbreviations, etc.) I used, compared to the WikiText-103 dataset (Fully written texts with compliance to proper grammar, etc).

I repeated the whole process, based on the provided WikiText-103 English language model from Merity et al. (2016) by copying the weights into my neural network before training, to create the transfer learning based Sentiment140 language model.

[4]Measurement of how well a model predicts a sample. The lower the value the better the prediction.

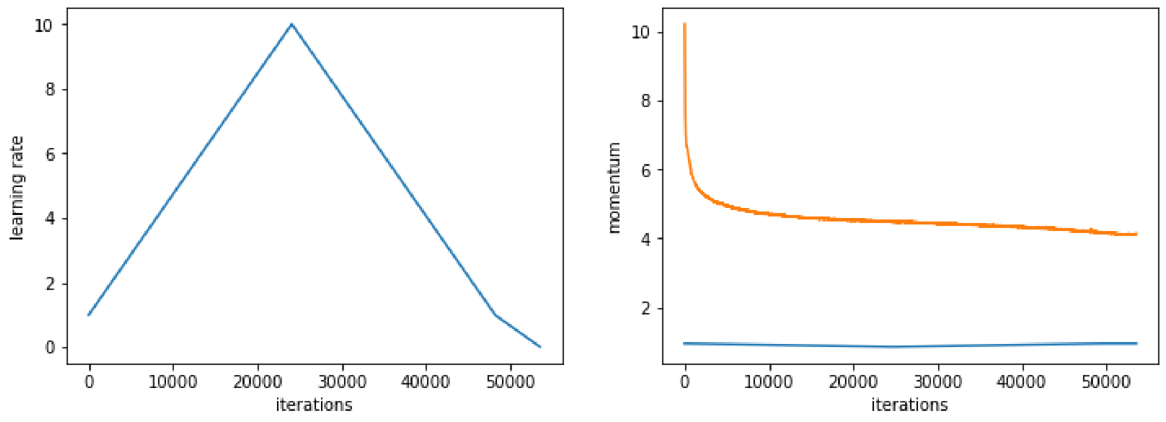


Figure 3.2: Learning Rate with Cyclical Learning Rates
On the left the learning rate by iteration count is shown. On the right in blue the momentum and in orange the loss by iteration count. The maximum loss of 10 is visible in the left graph at the peak, while the optimum learning of 2 is also used very early and at the very end of the training process.

Table 3.2: Language Model Comparison

Hyper-parameters					
Language model	embedding-size	hidden-size	layer	batch size	cyclical learning rate
Sentiment140 from scratch (1)	400	1150	3	60	1-10
Sentiment140 WikiText-103 (2)	400	1150	3	60	1.25-12.5

Hyper-parameters				Results			
Language model	momentum	epochs	clipping	training loss	validation loss	accuracy	perplexity
(1)	0.85-0.95	10	0.12	4.16	4.45	0.239	64.072
(2)	0.85-0.95	10	0.12	4.05	4.39	0.244	57.397

The (1) and (2) represent the language models marked in the upper table. The general architecture (embedding-size, hidden-size, layer count and used batch size) used for both language models is the same. Also the usage of cyclical learning rates, clipping and number of epochs are the same. Just the learning rate differs, due to the difference in weight initialization before training. The general results suggests that model (1) should perform better as a language model, due to its lower perplexity after training.

Table 3.3: EmoContext - Class distributions

Datasets	Class 1	Class 2	Class 3	Class 4
EmoContext	18.1%	18.2%	14.1%	49.6%
Sentiment140	50.1%	49.9%		
Hate speech	8.8%	73.5%	17.5%	

Shown here are all class distributions for the three classification datasets. The Sentiment140 dataset is the only set with a roughly equal distribution, whereas class 2 in the hate speech dataset is the most represented class and class 4 is the most represented class in the EmoContext dataset. However, it is noteworthy that class 4 is not included for the f1-score calculation and therefore the hate speech dataset is the only classification task with a very strong imbalance.

3.2 Classification Tasks

In this section I explain my setup for one example classification task, like I did for the language model in Chapter 3.1. The dataset used is from the SemEval2019 Task3[5](Microsoft 2018). For the comparison tasks, I used the Sentiment140 dataset (Go et al. 2009) and a hate speech dataset (Davidson et al. 2017). Both datasets contain short text messages with labels. Inductive Transfer Learning was used for all tasks, with three different language models as a source domain. The first language model is the WikiText-103 (Merity et al. 2016) English language model, the second language model is the one created as described in Chapter 3.1. The third language model is a transfer learning approach of both of the aforementioned datasets: Wikitext-103 weights are used to initialize the Sentiment140 dataset. Basically an English language model was fine-tuned to perform on short texts as they appear in Sentiment140.

For each classification task the language models were fine-tuned to the corresponding dataset and then performed the actual classification with an additional classification layer. Usually in inductive transfer learning the source domain and target domain are different. In the case of the Sentiment140 task, transfer learning with a language model generated from the same domain was performed. In Chapter 4 I show that this seems to be a very viable procedure.

3.2.1 Task Description

Task 3 of the SemEval2019 "EmoContext: Contextual Emotion Detection in Text" provides a dataset with chat conversations that contain three messages, going back and forth between two users. These conversations are labeled with a related emotion: happy, sad, angry or others. For measuring the performance, a micro-averaged f1 score is calculated based on the predictions of the labels happy, sad, and angry.

[5]Data provided by Microsoft.

3.2.2 Feature Selection and Splitting

Repeating what I did for the language model in Chapter 3.1.1, I did not add any additional features besides the text itself and the necessary tokens. I added a token "eot" after each message as a separation mark in the three message conversation and a "xbos" token at the beginning of every whole conversation to separate them clearly. These tokens are added to the overall vocabulary just like the _unk_ and _pad_ tokens.

Regarding pre-processing and feature extraction: The text was tokenized and regularized, as described in Chapter 3.1.2. Due to the smaller number of examples in this dataset the vocabulary frequency minimum was set to 5, with which it ended up with an effective vocabulary size of 2925 words out of an original 14076 words.

Similar to the language model, a regular 10-fold cross-validation was used, which resulted in a training set consisting of 27144 examples and a validation set consisting of 3016 examples. The test set is provided by the task organizers and therefore has a preset size of 2755 examples.

3.2.3 Hyper-Parameters

Architecture: The model also consists of 3 LSTM Layers with a hidden size of 1150 neurons each, an embedding size of 400, and an output layer equal to the size of the vocabulary, to fit the language model.

Other parameters: Table 3.4 consists of two columns, the first showing the hyper-parameters for the fine-tuning process of the transferred language model, the second showing the hyper-parameters for the training process of the classifier. The BPTT size is calculated as shown in Equation 3.1 with a standard deviation of 5 and an expectation value of 70. I used differing dropouts for the embedding layer, hidden weights, input layer, and activations between the LSTM layers, similarly to the language model laid out in Chapter 3.1. Also I made use of cyclical learning rates. For the language model I used them without momentum and for the classification task in combination with momentum. This has been decided by experimentation, on the grounds that the usage of momentum seemed to work better for classification than for fine-tuning. Due to the smaller amount of data to handle, I was able to work with bigger batch sizes, which lead to a decrease in training time. Despite the diminishing returns in computational time with increased batch size, Smith (2018) advises to use large batch sizes with their recommended pre-scheduled cyclical learning rates to further increase performance while reducing necessary iterations.

Functions: As shown in Table 3.3 there is a strong class imbalance in this dataset, therefore I made use of a weighted loss function for the classification process, to increase the error for less represented classes as described in Chapter 2.1.6. The loss function used for all classification tasks is cross entropy (Equation 2.6), for optimization I use Adam with decay rates beta1, beta2 = 0.8, 0.99.

3.2.4 Fine-Tuning and Classification

For fine-tuning I loaded the pre-trained language model's weights into the model for classification and calculated the average weights for tokens unknown to the pre-trained model. Following the idea of Yosinski et al. (2014), the model should perform better with averaged weights for words from the classification dataset, which did not appear in the language model's dataset, than randomly initialized weights. After loading the created weight set into the model, I let the model train only on the last embedding layer. This tunes the weights of the previously unknown words. For fine-tuning I calculated the accuracy based on how well the system predicts the next word, but the value that is actually important, is the perplexity. The perplexity is equal to the exponent of the loss calculated by the cross-entropy loss function, and is typically used as a performance measure for language models (Jelinek et al. 1977). Next I train on all layers until training and validation loss are nearly equal.

The fine-tuned language model is then used as the backbone of the classification model. This time the model predicts the corresponding labels to the inputs, instead of predicting the next word. Also for classification the aim is to increase the accuracy instead of minimizing the losses until a point of near equality. The new classification model consists entirely of the previous language model, only the output layer is different with a hidden layer equal to the number of unique labels instead of the size of the vocabulary. To further improve the classification results, the reversed class imbalance weights were multiplied onto the model's outputs, additionally to using the weights in the loss function. The new weight vector for the output can be calculated as shown in Equation 3.2.

$$w_{new} = \begin{bmatrix} 1/x_1 \\ 1/x_2 \\ \vdots \\ 1/x_n \end{bmatrix} \tag{3.2}$$

where:

x_n = nth value in loss weighting vector.

Table 3.4: Hyper-Parameters for Fine-Tuning and Classification of Sentiment140 Language Model

hyper-parameters	Fine-tuning	Classification
BPTT expectation value	70	70
dropouts[a]	[0.3, 0.24, 0.024, 0.18]	[0.48, 0.06, 0.36, 0.48]
cyclical learning rates [b]	[20, 10]	[10, 10, 0.95, 0.85]
batch size	150	150
clipping	None	0.12
learning rate [c]	[0.175, 0.011]	[0.01, 0.0006]
weight decay	1e-6	1e-6

[a][on input layer, on internal hidden weights, on embedding layer, activations between LSTM layers]
[b][ratio between start and end, percentage spend on last iterations,(maximum momentum, minimum momentum)]
[c][last layer, all layers]

3.2.5 Notes About the Other Datasets

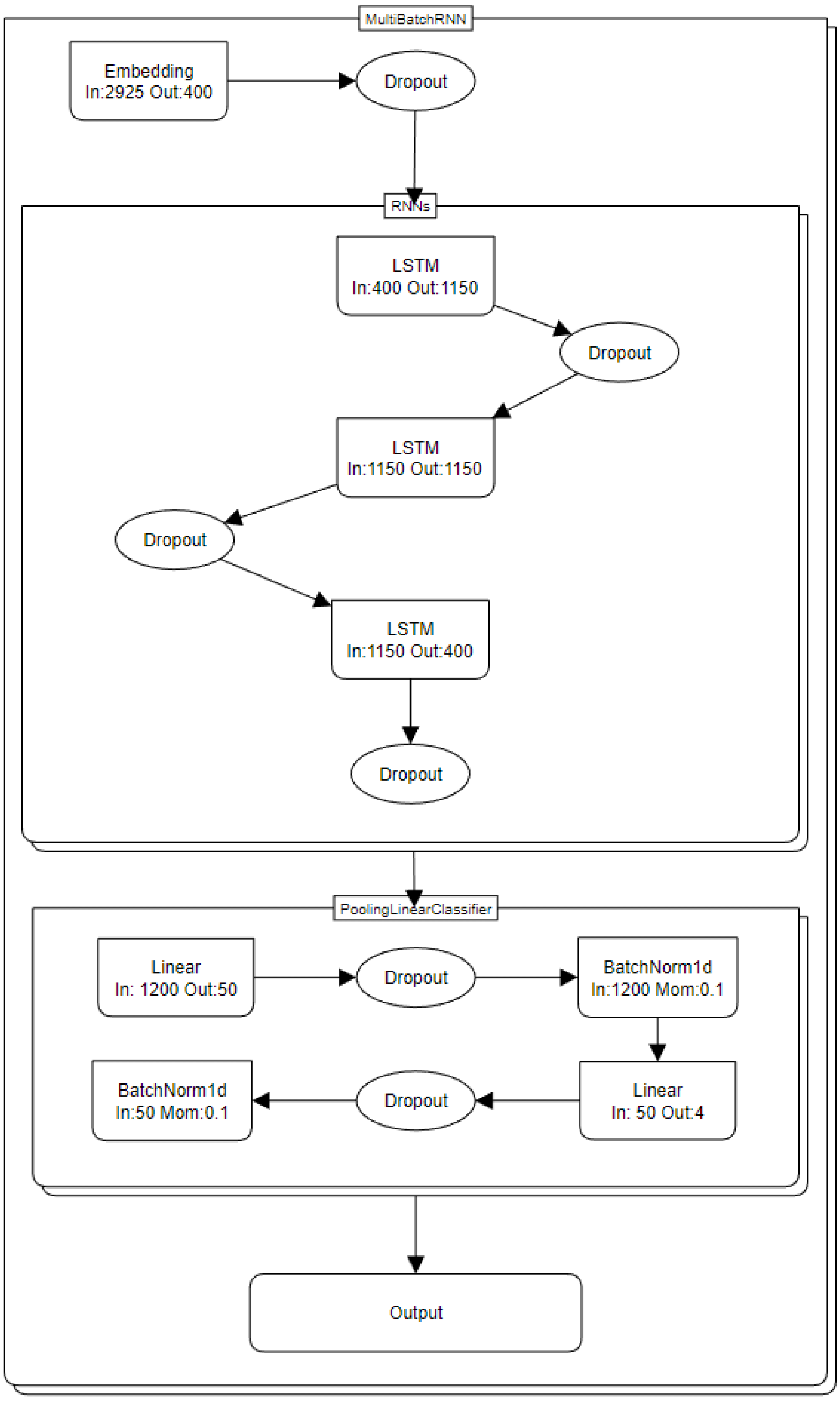
The hate speech and Sentiment140 dataset were trained very similarly to the SemEval2019 dataset. Little differences can be found in the way the pre-processing was done. For the Sentiment140 dataset links were removed and twitter specific name tags were exchanged with a general "xnametag" tag. For the hate speech dataset the twitter specific name tags were exchanged similarly, emoji's HTML entities[6] were also exchanged with a general "xemojitag" tag. Based on the idea that emojis can express information too, I tried to keep them with their HTML entities, but it turned out that the model did not just perform slower due to increased vocabulary size, but also performed worse accuracy wise on the final classifications. Due to the usually single occurrence of name tags, they would be simply replaced with the "unk" tag in the normalization process. To prevent losing information about someone being addressed, the tag "xnametag" was placed instead of the original name tags. Also I reduced the Sentiment140 dataset to half of its size (800.000 examples) to reduce overall training time, after randomly mixing it up[7]. The dataset is still big enough to allow a comparison in performance between big and small datasets and carries the same class distribution as the full dataset.

3.2.6 Baselines

I compare each dataset to the respective state-of-the-art performance, except for SemEval2019 Task, because the evaluation is still ongoing, I compare to the current ranking of the competition. Unfortunately there is no information yet about which techniques and models were used to reach the corresponding results for this task. For the hate speech dataset I additionally compared to the oracle baseline from Malmasi and Zampieri (2017).

[6]The hate speech dataset is provided in an encoding which represents emojis with their HTML entities like this: &#12345.
[7]Using Numpy's random seed 42 for reproducibility and to prevent strongly differing results.

Figure 3.3: Multi Batch RNN Architecture



In this graph the different layers and applied dropouts of the models, used for the training processes are shown.

Chapter 4

Results

The results of the three classification tasks, based on the three language models can be seen in Table 4.1. While this thesis focuses on English short-text classification, the technique used can be applied for any form of text in any language with corresponding language model. Results in comparison to the state-of-the-art for each corresponding task are shown in Tables 4.2, 4.3, 4.4.

4.1 Fine-tuning and Classification Results

All models have been tuned with slightly different hyper-parameters, but for comparability I tried to keep them as equal as possible. Nonetheless, every dataset is different and therefore needs different learning rates, weight decay, etc. for optimal performance. Therefore differences could not be completely prevented. Looking at Table 4.1, in the fine-tuning column it can be seen that most of the language models are slightly overfitted[1]. While testing different settings for fine-tuning the language models, it turned out that models with a slight overfit (± 0.3) had a higher accuracy in predicting the next word, than their non over-fitted counterparts.

The current state-of-the-art perplexity on the WikiText-103 dataset is 29.2 (Rae et al. 2018), the Sentiment140 based language model, created in this thesis, has a perplexity of 64.072 and the WikiText-103 based Sentiment140 model, a perplexity of 57.397, which is nearly twice as high, but still in an acceptable range, being close to the 40.8 achieved by Grave et al. (2016) and between the 31.9-80.9 reached with a variety of models by Tang and Lin (2018). Against my expectations, I could achieve better perplexities after fine-tuning the language models on the emotion and Sentiment140 dataset with my own language models, than with the WikiText-103 language model. Comparing the micro averaged f1 score with the weighted micro average f1 score, it also becomes visible that multiplying the reversed initial weights, used for loss weighting onto the model's outputs, can increase the f1-score by 1%-7%,

[1]The training loss is smaller than the validation loss.

Table 4.1: Language Model Fine-Tuning and Classification Results

		Fine-tuning			f1-scores	
Dataset	Model	training loss	validation loss	perplexity	micro average f1	weighted micro average f1
Emotion	wt103	3.45	3.58	35.87	**0.691**	0.73
	sem140	**3.12**	3.47	**32.14**	**0.691**	0.735
	wt103+sem140	3.21	3.47	**32.14**	0.686	**0.753**
Hate-speech	wt103	**3.66**	**4.22**	**68.03**	**0.814**	0.824
	sem140	4.38	4.42	83.09	0.798	0.802
	wt103+sem140	4.36	4.399	81.37	0.797	**0.832**
Sentiment140	wt103	3.97	4.03	56.26	0.854	0.854
	sem140	3.93	3.98	53.52	0.854	0.854
	wt103+sem140	**3.92**	**3.91**	**49.90**	**0.856**	**0.856**

Best values for each column and dataset are presented in **bold** numbers. All values presented are not absolute, but averages with varying small deviations. wt103 stands for the WikiText-103 dataset, sem140 for the Sentiment140 dataset and wt103+sem140 is the Sentiment140 language model based on the WikiText-103 language model, using transfer learning.

which is to the best of my knowledge a novel approach to improving performance on unbalanced datasets.

Not all papers present their findings, using the same performance measures therefore the classification task comparison tables are split into two result columns. One showing the performances as f1-score and one showing the performances as accuracy.

In Table 4.2, I present the results of my models compared to the current state-of-the-art models on the hate speech dataset. For the f1 scores, my models perform 25%-29% better than the baseline, but about 13% worse than the state of the art by Zhang et al. (2018).

Table 4.3 shows the results of this paper in comparison to the current state-of-the-art classification systems on the Sentiment140 dataset. Considering the f1-score, this approach created a new state-of-the-art classification performance on the Sentiment140 dataset with all used language model variants, with the transfer learning based language model performing best out of these. The WikiText-103 based Sentiment140 model has a 29.7% higher f1 score than the baseline and 1.9% higher than the next best result by Saif et al. (2012). Comparing on accuracy, my models perform about 71.4% better than the baseline and about 3.15% worse than the best performing model by Bakliwal et al. (2012).

For the last dataset shown in Table 4.4, there are no published papers as to how those results were achieved yet. The results (and therefore the ranking) can be changed daily, therefore the shown table is just the current state of the 29th November 2018. Until that date, I could achieve a result ranking this model in the top 10 with the WikiText-103 based Sentiment140 language model. The micro averaged

Table 4.2: Hate Speech Classification Results

Model	Accuracy (%)	F1
Majority Class Baseline	73.5	0.64[a]
Oracle (Malmasi and Zampieri 2017)	91.6	
CNN (Zhang et al. 2018)		**0.94**
CNN + GRU^B (Zhang et al. 2018)		**0.94**
CNN + GRU (Zhang et al. 2018)		**0.94**
Logistic regression with L2 regularization (Davidson et al. 2017)		0.91
Transfer learning - wt103+sem140 (this paper)	**79.6**	0.83
Transfer learning - wt103 (this paper)	79.0	0.82
Character 4-grams (Malmasi and Zampieri 2017)	78.0	
Transfer learning - sem140 (this paper)	77.9	0.80

[a]If all predictions are of the majority class with weighted f1-score.

Best values for each column are presented in **bold** numbers and the models described in this paper in **bold** names.

f1 score of this paper's model is just 0.79% lower than the best ranking model (by hchenyang) and 128% higher than the baseline.

Table 4.3: Sentiment140 Classification Results

Model	Accuracy (%)	F1
Baseline	0.5	0.66
Transfer learning - wt103+sem140 (this paper)	85.7	**0.856**
Transfer learning - wt103 (this paper)	85.5	0.854
Transfer learning - sem140 (this paper)	85.3	0.854
Semantic features (Saif et al. 2012)		0.84
ENS(LR + RF + MNB)-BoW + lex (Da Silva et al. 2014)		0.81
MNB-BoW + lex (Da Silva et al. 2014)		0.79
wNI + Popularity Score (Bakliwal et al. 2012)	**88.4**	
Dynamic Convolutional Neural Network (Kalchbrenner et al. 2014)	87.4	
SVM (Kalchbrenner et al. 2014)	81.6	

Best values for each column are presented in **bold** numbers and the models described in this paper in **bold** names.

Table 4.4: SemEval2019 Task 3 Classification Results (as of 29.11.2018)

Model	Micro averaged F1
Baseline	0.33
hchenyang	**0.759**
soujanyaporia	0.757
jogonba2	0.756
rafalposwiata	0.755
Transfer learning - wt103+sem140 (this paper)	0.753
Transfer learning - sem140 (this paper)	0.735
Transfer learning - wt103 (this paper)	0.730

Best values for each column are presented in **bold** numbers and the models described in this paper in **bold** names.

Table 4.5: Hate Speech WikiText-103 Based Confusion Matrix

		Truth			
		Hate Speech	Offensive	Neither	Total
Predictions	Hate Speech	124	108	33	265
	Offensive	129	1520	120	1769
	Neither	11	34	399	444
	Total	264	1662	552	2478

Every row shows the predictions made for each class and every columns shows the true labels for each class. The model resulting in this confusion matrix, seems to perform pretty badly on the hate speech dataset, labeling half of the examples as offensive. This might be due to the large amount of offensive examples, as can be seen it performs fairly well on offensive examples with 91% correct predictions.

Chapter 5

Discussion and Future Work

Looking at the Tables 4.2 - 4.4, it becomes evident that transfer learning by itself is not the definite state-of-the-art technique for solving short text classification problems, but it is definitely a very viable and quick solution to getting well performing[1] models. Furthermore it seems that the WikiText-103 based Sentiment140 language model performs better on all tasks, which creates the assumption, that a deeper level of transfer[2] seems to further improve results. This opens the question, if there is a limit as to how deep knowledge transfer leads to an increase of performance. Transfer learning enables the machine learning engineer to quickly use previously learned models and apply them to new problems without further feature engineering or any other human labor and time intensive tasks. I assume that, if a transfer learning based system is used in combination with well thought out feature engineering in the form of ensemble networks[3], new state-of-the-art models can be created. Even though transfer learning does not bring a revolution to short text classification, as it did to the field of visual computing, it definitely does improve the time needed for developing models. Due to the nature of short texts, as described in Chapter 2.2.2, I presuppose that transfer learning for NLP is much more efficient with long texts that consist of proper grammar and do not strongly rely on abbreviations or previous context[4]. The regularity in grammar and vocabulary enables the model to start its classification on the base of a clearer understanding of a language than on the basis of text that is very noisy due to irregular use of grammar and vocabulary.

5.1 Language Model Choice and Fine-Tuning

As can be seen in Table 4.1, most of the language models are slightly overfitted while fine-tuning. As explained in Chapter 4.1 the accuracy kept increasing despite fine-tuning, but I could not test, if the

[1]Close to or better than current state-of-the-art

[2]Multiple usages of transfer learning for a single task. In this case WikiText-103 $\rightarrow$ Sentiment140 $\rightarrow$ Classification task.

[3]A combination of multiple models that create e.g. one classification system together.

[4]Previous context in the form of messages that are being replied to or tweet references etc.

increased accuracy in fine-tuning also leads to an increased performance for the later classification task. It needs further testing if the model performs better in classification with a fine-tuning, focused on improving the language model's accuracy or focused on getting the lowest loss without over- or underfitting. Also, despite the differences in perplexity of the language models themselves, there seems to be no visible performance difference after fine-tuning, as presented in Table 4.1. Therefore I assume that with proper fine-tuning for the classification task, the initial perplexity of a language model, used as a basis, does not influence the overall classification task too strongly, as can be seen in the f1 scores in Table 4.1. This can also result due to the differences in hyper-parameter tuning, which are not completely avoidable and thus needs further research and testing.

5.2 Class Imbalance Correction

In this thesis I introduced a novel approach for improving model performance on datasets with unbalanced class distributions by simply multiplying the reversed weights (Equation 3.2 for each class with the model's outputs. This method is just tested with a weighted cross-entropy loss function and might differ for other weighted loss functions. In small tests it seemed, that this output correction does improve the overall f1 score, when the loss function did not use weights, but the magnitude of improvement was very small. However, these tests were not sufficient enough to present in this thesis and also need further research and testing. Also it might be possible to even further increase the performance with more complex optimization algorithms on the model's outputs. As shown in Table 4.1 this method does not affect datasets with nearly equal class distribution (i.e. Sentiment140). It would also be interesting to test, if the magnitude of the imbalance influences the effectiveness of this method.

Chapter 6

Conclusion

In this thesis I compared different language models on their impact for transfer learning based short text classification models. I discovered that language models that are created through transfer learning themselves seem to perform better than from scratch created language models. For this I tested 3 language models, one based on the Sentiment140 dataset, one based on the WikiText-103 dataset and one is a transfer learning based dataset with the WikiText-103 as a transfer base and the Sentiment140 as a target dataset. These three language models were tested each on three different short text classification tasks. A new state of the art performance was achieved on the Sentiment140 dataset (f1-score of 0,856) with the transfer learned language model approach, while decent results were achieved on the hate speech dataset (f1 score of 0,83) and a top 10 result was achieved on the EmoContext dataset (micro averaged f1 score of 0,753). Transfer learning seems to be an effective means for quick short text classification model creation. It might not always be the best approach by itself, but very likely the approach giving the best results with the fewest time investment. Combined with proper feature engineering, transfer learning easily has the potential to be a cornerstone for future state-of-the-art short text classification models. Additionally I proposed a new method to improve classification performance on imbalanced datasets, which improved the f1-score by about 1%-7%.

Bibliography

Bakliwal, A., Arora, P., Madhappan, S., Kapre, N., Singh, M., and Varma, V. (2012). Mining sentiments from tweets. In *Proceedings of the 3rd Workshop in Computational Approaches to Subjectivity and Sentiment Analysis*, pages 11–18.

Banerjee, S., Ramanathan, K., and Gupta, A. (2007). Clustering short texts using wikipedia. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 787–788. ACM.

Benediktsson, J. A., Swain, P. H., and Ersoy, O. K. (1990). Neural network approaches versus statistical methods in classification of multisource remote sensing data.

Bird, S., Klein, E., and Loper, E. (2009). *Natural language processing with Python: analyzing text with the natural language toolkit.* " O'Reilly Media, Inc.".

Boden, M. (2002). A guide to recurrent neural networks and backpropagation. *the Dallas project*.

Brownlee, J. (2017). Difference between classification and regression in machine learning. Available at `https://machinelearningmastery.com/classification-versus-regression-in-machine-learning/`.

Da Silva, N. F., Hruschka, E. R., and Hruschka Jr, E. R. (2014). Tweet sentiment analysis with classifier ensembles. *Decision Support Systems*, 66:170–179.

Darken, C. and Moody, J. E. (1991). Note on learning rate schedules for stochastic optimization. In *Advances in neural information processing systems*, pages 832–838.

Davidson, T., Warmsley, D., Macy, M., and Weber, I. (2017). Automated hate speech detection and the problem of offensive language. In *Proceedings of the 11th International AAAI Conference on Web and Social Media*, ICWSM '17.

Davis, J. and Goadrich, M. (2006). The relationship between precision-recall and roc curves. In *Proceedings of the 23rd international conference on Machine learning*, pages 233–240. ACM.

Do, C. B. and Ng, A. Y. (2006). Transfer learning for text classification. In *Advances in Neural Information Processing Systems*, pages 299–306.

Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159.

Duyck, J., Lee, M. H., and Lei, E. (2014). Modified dropout for training neural network. *School Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, USA, Advanced Introduction to Machine Learning Course, Tech. Rep*, pages 10–715.

Elman, J. L. (1990). Finding structure in time. *Cognitive science*, 14(2):179–211.

explosion.ai (2016). Spacy tokenizer. Available at `https://spacy.io/api/tokenizer`.

fast.ai (2018). fastai - the fast.ai deep learning library, lessons and tutorials. Available at `https://github.com/fastai`.

Go, A., Bhayani, R., and Huang, L. (2009). Twitter sentiment classification using distant supervision. *CS224N Project Report, Stanford*, 1(12).

Grave, E., Joulin, A., and Usunier, N. (2016). Improving neural language models with a continuous cache. *arXiv preprint arXiv:1612.04426*.

Guyon, I. and Elisseeff, A. (2003). An introduction to variable and feature selection. *Journal of machine learning research*, 3(Mar):1157–1182.

Hastie, T., Tibshirani, R., and Friedman, J. (2009). Unsupervised learning. In *The elements of statistical learning*, pages 485–585. Springer.

He, H. and Garcia, E. A. (2008). Learning from imbalanced data. *IEEE Transactions on Knowledge & Data Engineering*, (9):1263–1284.

Herculano-Houzel, S. (2009). The human brain in numbers: a linearly scaled-up primate brain. *Frontiers in human neuroscience*, 3:31.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

Howard, J. and Ruder, S. (2018). Universal language model fine-tuning for text classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 328–339.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*.

Jelinek, F., Mercer, R. L., Bahl, L. R., and Baker, J. K. (1977). Perplexity—a measure of the difficulty of speech recognition tasks. *The Journal of the Acoustical Society of America*, 62(S1):S63–S63.

Kalchbrenner, N., Grefenstette, E., and Blunsom, P. (2014). A convolutional neural network for modelling sentences. *arXiv preprint arXiv:1404.2188*.

Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., and Tang, P. T. P. (2016). On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Kohavi, R. et al. (1995). A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, pages 1137–1145. Montreal, Canada.

Krogh, A. and Hertz, J. A. (1992). A simple weight decay can improve generalization. In *Advances in neural information processing systems*, pages 950–957.

Loukides, M. (2017). The machine learning paradox. Available at `https://www.oreilly.com/ideas/the-machine-learning-paradox`.

Malmasi, S. and Zampieri, M. (2017). Detecting hate speech in social media. *arXiv preprint arXiv:1712.06427*.

Merity, S., Keskar, N. S., and Socher, R. (2017). Regularizing and optimizing lstm language models. *arXiv preprint arXiv:1708.02182*.

Merity, S., Xiong, C., Bradbury, J., and Socher, R. (2016). Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*.

Microsoft (2018). Emocontext - humanizing ai. Available at `https://www.humanizing-ai.com/emocontext.html`.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

Mikolov, T., Karafiát, M., Burget, L., Černockỳ, J., and Khudanpur, S. (2010). Recurrent neural network based language model. In *Eleventh Annual Conference of the International Speech Communication Association*.

Mitchell, T. M. et al. (1997). Machine learning. wcb.

Ng, A. (2018). Coursera course: Machine learning. Available at `https://www.coursera.org/learn/machine-learning`.

Oquab, M., Bottou, L., Laptev, I., and Sivic, J. (2014). Learning and transferring mid-level image representations using convolutional neural networks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Pan, S. J., Yang, Q., et al. (2010). A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359.

Pennington, J., Socher, R., and Manning, C. (2014). Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543.

Phan, X.-H., Nguyen, L.-M., and Horiguchi, S. (2008). Learning to classify short and sparse text & web with hidden topics from large-scale data collections. In *Proceedings of the 17th international conference on World Wide Web*, pages 91–100. ACM.

QasemiZadeh, B. (2015). *Investigating the use of distributional semantic models for co-hyponym identification in special corpora*. PhD thesis.

Rae, J. W., Dyer, C., Dayan, P., and Lillicrap, T. P. (2018). Fast parametric learning with activation memorization. *arXiv preprint arXiv:1803.10049*.

Reasoning, D. (2015). Digital reasoning trains world's largest neural network, shatters record previously set by google.

Saif, H., He, Y., and Alani, H. (2012). Semantic sentiment analysis of twitter. In *International semantic web conference*, pages 508–524. Springer.

Scott, S. and Matwin, S. (1999). Feature engineering for text classification. In *ICML*, volume 99, pages 379–388. Citeseer.

Shallue, C. J., Lee, J., Antognini, J., Sohl-Dickstein, J., Frostig, R., and Dahl, G. E. (2018). Measuring the effects of data parallelism on neural network training. *arXiv preprint arXiv:1811.03600*.

Smith, L. N. (2017). Cyclical learning rates for training neural networks. In *Applications of Computer Vision (WACV), 2017 IEEE Winter Conference on*, pages 464–472. IEEE.

Smith, L. N. (2018). A disciplined approach to neural network hyper-parameters: Part 1–learning rate, batch size, momentum, and weight decay. *arXiv preprint arXiv:1803.09820*.

Smith, L. N. and Topin, N. (2017). Super-convergence: Very fast training of residual networks using large learning rates. *arXiv preprint arXiv:1708.07120*.

Sola, J. and Sevilla, J. (1997). Importance of input data normalization for the application of neural networks to complex industrial problems. *IEEE Transactions on Nuclear Science*, 44(3):1464–1468.

Sriram, B., Fuhry, D., Demir, E., Ferhatosmanoglu, H., and Demirbas, M. (2010). Short text classification in twitter to improve information filtering. In *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*, pages 841–842. ACM.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958.

Sutton, R. S., Barto, A. G., Bach, F., et al. (1998). *Reinforcement learning: An introduction*. MIT press.

Tang, R. and Lin, J. (2018). Adaptive pruning of neural language models for mobile devices. *arXiv preprint arXiv:1809.10282*.

van Rijn, J. N. and Hutter, F. (2017). An empirical study of hyperparameter importance across datasets. *Proc. of AutoML*, pages 97–104.

Wikipedia, the free encyclopedia (2010). Multilayerneuralnetworkbigger english. [Online; accessed October 10, 2018].

Wilson, D. R. and Martinez, T. R. (2003). The general inefficiency of batch training for gradient descent learning. *Neural Networks*, 16(10):1429–1451.

Yosinski, J., Clune, J., Bengio, Y., and Lipson, H. (2014). How transferable are features in deep neural networks? In *Advances in neural information processing systems*, pages 3320–3328.

Zeng, J., Li, J., Song, Y., Gao, C., Lyu, M. R., and King, I. (2018). Topic memory networks for short text classification. *arXiv preprint arXiv:1809.03664*.

Zhang, Z., Robinson, D., and Tepper, J. (2018). Detecting hate speech on twitter using a convolution-gru based deep neural network. In *European Semantic Web Conference*, pages 745–760. Springer.

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

[illegible]

Düsseldorf, 13.December 2018 Leon Dummer