



Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Masterarbeit

Sentiment Analysis of Informal Online Texts with Neural Networks

vorgelegt von

Tim Alexander Dobert

Matrikelnummer 6427948

Studiengang Informatik

April 2019

Betreuer: Steffen Remus

Erstgutachter: Prof. Dr. Chris Biemann

Zweitgutachter: Steffen Remus

Abstract

With the large amounts of user generated text on social media and the internet in general, automated text processing and classification is becoming a more and more attractive tool for maintaining an overview of popular topics and trends. Sentiment analysis specifically can be valuable to monitor public perception and the social climate of a topic. This requires reliable and accurate methods, otherwise the gathered data is of little use. In this work, we examine the properties of informal online text communication and compare the performance of different neural networks in on this data. Among these are the recently introduced capsule networks, which have already achieved promising results on image classification tasks. We examine how their new ideas like routing by agreement and reconstruction modules fare on text and how these networks as a whole perform in this domain. Comparing this new type of network to more traditional CNNs (convolutional neural networks) and LSTM (long short-term memory) networks on two different datasets, shows that they perform better in some scenarios and worse in others. Using relatively simple, shallow models with pretrained word embeddings, we get results that are not too far from the the state-of-the-art. Even when classification scores are low, diving into the results in detail can still provide information on trends and patterns in the datasets. The reconstructions, which can produce impressive results in simple image classification, are much more difficult to use on text. Even with multiple adjustments, they provide no improvement in text classification.

Contents

1	Overview	4
1.1	Introduction	4
1.2	Motivation	4
1.3	Hypothesis	6
2	Related Work	7
2.1	Text Processing Basics	7
2.2	Text Classification in General	8
2.3	Convolutional Neural Networks	10
2.4	LSTM Networks	11
2.5	Capsule Networks	12
3	Methodology	16
3.1	Preprocessing	16
3.2	CNN and LSTM Network	17
3.3	Capsule Network	19
3.4	Hyperparameter Optimization	21
4	Evaluation	24
4.1	Datasets	24
4.2	Grid Search Results	29
4.3	Test Results	30
5	Further Insights	36
5.1	Reconstructions	36
5.2	Discussion	40
6	Conclusion	42
6.1	Conclusion	42
6.2	Future Work	43
	Bibliography	44
	Appendices	46

1 Overview

1.1 Introduction

Despite the prevalence of smart phones with front facing cameras and increasing access to high bandwidths, a large part of online communication is still text based. Whether social media, message boards or comments on various websites, the written word is one of the most common forms of user generated content on the web. Because of its abundance, text can be a great resource for gaining insight into a community's social climate, stances on certain topics or other sentiments. Whether for market research or because of rising concerns about hate speech, there are more and more reasons to look at user generated text at a large scale. The sheer volume of text makes this a monumental task however, with Twitter alone boasting over 500 million posts a day¹. Since comprehensive manual analysis is out of the question, accurate, automatic procedures would be extremely helpful in this endeavor. Luckily, the advances in machine learning in the past years are being used to develop sophisticated methods for picking up on the nuances of human language, which is necessary for building accurate text classifiers in these areas.

This work examines the properties of informal, user generated online texts and compares different types of neural networks for the classification of informal online communication, especially in regards to sentiment analysis. The recently introduced capsule networks receive a special focus in this regard, in order to evaluate their potential for text processing in general and the classification of short informal texts in particular. The goal is not to develop the most accurate classifier possible, but rather to provide an overview of common methods. The results should serve as an indication of how useful these methods are for this kind of task and reveal the areas in which they could improve the most.

1.2 Motivation: Sentiment Analysis and Online Harassment

The field of sentiment analysis is a growing subsection of text classification. It deals with extracting attitudes, value judgments and opinions from speech, most often in the form of plain text. Doing this on spoken language can be an easier task, since emotion is more easily conveyed in that domain, but there are many applications where only text is available. Commonly, sentiment analysis is performed to gather a large-scale view on people's opinions on events, politics or products. This can be useful from two perspectives. As a consumer doing online research on a product, it can be overwhelming to manually read through a vast number of reviews, blog posts and comments to get a picture of the general opinion. An automatically maintained "opinion database" could provide a much quicker way to arrive at a conclusion. Similarly, from the manufacturer's side, understanding the reception of a product is important for future business decisions. (Pang, Lee, et al., 2008)

¹<https://business.twitter.com/> (accessed March 31, 2019)

Adjacent to the field of sentiment analysis is the detection of personal attacks, harassment and toxicity in user comments and messages. Surveys show that a majority of people have witnessed harassment online and that many of the victims reduce their participation in discussions and content generation as a result of these attacks (Wulczyn et al., 2017). Even if they are not the target themselves, many of these people report being discouraged from making further contributions (Duggan, 2017)². This is concerning on both a personal level and from a purely business oriented point of view. Since users generate most, if not all, of the content on many of these online platforms, it would be beneficial to reduce these attacks not only for moral reasons, but also to improve the public image. This requires a monumental amount of work on large platforms though. Hence, Wulczyn et al. (2017) suggest that automatic classifiers would help moderators reduce personal attacks and toxicity in Wikipedia comments. The fact that attacks are dispersed among a large number of users, but tend to cluster when they occur, indicates, that setting a standard of what is acceptable by curbing harassment quickly before it poisons the discussion, would improve the overall climate. A well enforced standard and well maintained climate would likely discourage this kind of behavior and help prevent it. Considering that only a fraction of victims report the attacks², automatic detection could not only help human moderators find offending posts, but can also provide information on the history of a user's contributions, reducing the manual work necessary to decide, whether they are problematic or not. This improvement to both efficacy and efficiency could be a great help in the endeavor to reduce harassment and toxicity.

But for automated moderation tools to actually be of help, they must be accurate. Otherwise the manual overhead required to differentiate between good and bad predictions outweighs the benefit of having them. As such, existing text classification and sentiment analysis methods need to improve before this becomes practical. Machine learning and especially neural networks have been rising in popularity in recent years. They are being used in many automation and artificial intelligence related areas, including natural language processing. One potentially exciting new development in this field are capsule networks, which were introduced in 2017 by Hinton et al. Originally intended to improve upon convolutional neural networks (CNNs) in computer vision tasks by more closely emulating the process of human vision, their principles are general enough to be used for other machine learning applications. As the name suggests, these networks contain capsules, each of which represents one entity, an aspect or pattern which frequently occurs in the data. These capsules are small networks themselves. Unlike a usual neuron, which outputs a single scalar, they output a vector containing the detected instantiation parameters of that entity, with the overall length representing the confidence that it is, in fact, present.

By exploring the potential of these capsule networks for sentiment analysis we can hopefully get closer to a practical application of sentiment analysis and the automated detection of personal attacks and toxicity in text.

²Wikimedia Commons contributors, "File:Harassment Survey 2015 - Results Report.pdf," Wikimedia Commons, the free media repository, https://commons.wikimedia.org/w/index.php?title=File:Harassment_Survey_2015_-_Results_Report.pdf&oldid=224534499 (accessed March 27, 2019)

1.3 Hypothesis and Structure

The goal of this work is to compare different kinds of neural networks on this specialized task. Informal online communication has several properties that make it more difficult to work with than more formal literature. Exploring this task with different methods might bring new insights into the strengths and weaknesses of different approaches and reveal, which established methods and heuristics work well and which need to be rethought. This is especially true for the capsule networks which are still relatively unexplored in this regard. Doing this hopefully brings insight into the practicability of using these systems for large scale applications and shows, what benefit can be gained from their employment.

The procedure and structure of this thesis is as follows: Chapter 2 provides an overview of the current relevant technologies used for this task, including basic text processing and different neural network types that are commonly used in similar tasks. Chapter 3 goes over the implementations and experimental setup in detail. The evaluation, which includes a description and analysis of the datasets, as well as all relevant results, is presented Chapter 4. Following that is Chapter 5, discussing the use of the capsule network’s reconstructions on text and the results in general. Finally, the conclusion and possible future work are found in Chapter 6.

2 Related Work

This chapter goes over the established groundwork related to text classification with neural networks. It summarizes the current state of text preprocessing, formally introduces the task of text classification and describes the network types commonly used for it. This includes a detailed explanation of capsule networks. Relevant related work with practical applications of these networks on similar tasks is also discussed.

2.1 Text Processing Basics

Modern language and text processing consists of many steps, most of which would be considered preprocessing for the task at hand. In order for most neural networks to be able to process data, samples must be formatted as tensors of floating point values. Unlike images or audio, whose digital representations lends itself easily to this conversion, it is a bit more complicated with text. The naive approach would be to encode every occurring character as a scalar or a one-hot encoded vector. This ignores a lot of information though, as many characters are more closely related to some than to others. Having a representation that can include differences between punctuation, digits, vowels, consonants or other kinds of groupings gives a neural network more information to work with. The most common solution for this problem in text processing are vector embeddings. In this type of embedding, every occurring token, a character or a word, is represented by an n -dimensional vector. Ideally, closely related words are clustered together in this vector space, with the spacial difference corresponding to the difference in meaning. These embeddings are usually generated with a machine learning approach like the skip-gram model introduced by Mikolov et al. (2013). This representation also does not have to remain static while it is being used in a downstream task. In most applications, they are implemented as an embedding layer, whose weights are adjusted during training, just like all the other weights in the network.

Characters alone do not hold much meaning however. When people read a text, they mostly recognize words as a unit instead of going through it character by character. Character embeddings do work, but the additional information that word embeddings can contain makes them very attractive for classification tasks. However, this leads to the issue of dividing the text into separate words. This is the task of tokenization. Just splitting everything by spaces will produce suboptimal results due to a number of common edge cases. Punctuation for example is usually appended to the the preceding word. Every word that occurs at the end of a sentence would therefore be embedded twice. Luckily, problems like this are largely solved, with modern tokenizers reaching accuracies of 99.8% (Beißwenger et al., 2016). After obtaining the tokens, it is often useful to filter out the ones that are not needed. For many tasks, punctuation and stop words (e.g. "the", "is") are irrelevant and are removed from the data before the text is processed. This should remove noise and increase the density of relevant information. The process of getting a floating-point tensor representation from raw text is illustrated in figure 2.1

Raw text	"The quick, brown fox..."
After tokenization	["The", "quick", ",", "brown", "fox", "..."]
After filtering	["quick", "brown", "fox", "..."]
After Indexing	[3, 1, 2, ...]

After one-hot-encoding

$$\begin{bmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ \vdots \end{pmatrix} & \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ \vdots \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ \vdots \end{pmatrix} & \dots \end{bmatrix}$$

After embedding

$$\begin{bmatrix} \begin{pmatrix} 0.23 \\ 0.55 \\ 0.01 \\ 0.07 \\ \vdots \end{pmatrix} & \begin{pmatrix} 0.36 \\ 0.64 \\ 0.86 \\ 0.90 \\ \vdots \end{pmatrix} & \begin{pmatrix} 0.03 \\ 0.33 \\ 0.43 \\ 0.65 \\ \vdots \end{pmatrix} & \dots \end{bmatrix}$$

Figure 2.1: This is an overview of process of transforming plain text to a tensor. The indexing and one-hot encoding steps might be skipped depending on the processing pipeline and the interface of the neural network. The values were chosen arbitrarily as an example.

Most corpora contain words not found in pretrained word embeddings. In those cases, the most common solution is to assign every one of those words a randomly generated vector as an embedding. Using sophisticated methods that take the distribution of the other word vectors into account, can yield better results than a standard, unbiased random number generation method (Kim, 2014). Another approach is to build the vectors for these words from pretrained character embeddings (dos Santos and Gatti, 2014). To avoid generating new vectors for otherwise known words, it is important to have a good tokenizer, since incorrectly split tokens are rarely present in pretrained embeddings. Even if they are, due to their rare occurrence in text, their vector representation will not be as meaningful as the one belonging to the correctly tokenized word.

Besides the word and character representations, there are other features that can be gained from text. Taggers can annotate words with grammatical or syntactic properties, like part of speech, or tags that indicate meaning and how the word is generally used. These tags can then be used to build parse trees, which organize the relationships between the words in a sentence. This can help with long range dependencies. Building a parse tree is usually done on tokenized text.

In this thesis, we only use tokenization, filtering and word embeddings. This minimizes dependencies, as we only have end-to-end systems that produce output based on raw text and no extra input.

2.2 Text Classification in General

Formally, text classification, sometimes called text categorization, can be described as assigning a binary value of either *True* or *False* to every $(d_j, c_i) \in \mathcal{D} \times \mathcal{C}$. This tuple represents a pair of a text document d_j , which is part of a larger corpus \mathcal{D} , and a class or

category c_i that is part of a predefined set of categories \mathcal{C} (Sebastiani, 2002). Labeling a pair as *True*, means that the document belongs to the class. Going one step further, we can define the set of correct classifications L as follows: $(d_j, c_i) \in L \subseteq \mathcal{D} \times \mathcal{C}$ if and only if the correct label for (d_j, c_i) is *True*. Like with all classification tasks, the goal is to approximate a target function, which dictates how every pair should be labeled.

In general, classification tasks are divided into three types. The simplest of these is a binary classification, where every document belongs to exactly one of two possible classes. Most of these tasks define one property and then define one class as having the property and the other as lacking it. If the samples are represented as n -dimensional feature vectors, then the target function divides \mathbb{R}^n into two sub-spaces, each representing one of the two classes. In cases where this division can be modeled as a hyperplane, a neural network with two layers is enough to find it. For non-linear functions, a minimum of one hidden layer is needed. A more complex task is multi-class classification. In these tasks, every document belongs to exactly one of n classes with $n > 2$. In practice, this means, that a classification procedure should output the probabilities $p_1 \dots p_n$ where p_i represents the likelihood or confidence, that the document belongs to class c_i . To ensure that all of these probabilities sum to 1, models use either a hardmax or softmax function. A hardmax simply sets the highest probability to one and the all others to zero. This loses valuable information for the back propagation though, as the magnitude of the error is lost. Therefore, a function that retains this information is preferred. The softmax function fulfills theses requirements. It is defined as $\text{softmax}(x_i) = \frac{\exp(x_i)}{\sum_j^n \exp(x_j)}$ where x is an n -dimensional vector. After applying one of these functions to the output, the predicted class is the one with the highest probability. Many multi-class problems operate under the closed world assumption, which states that $|\mathcal{C}|$ is finite and known and that $\forall d \in \mathcal{D} : \exists c \in \mathcal{C} : (d, c) \in L$. In other words: No document belongs to a class that has not been established beforehand. If this can not be guaranteed, then an *orphan* or *none of the above* class can be used. Though at that point, the task arguably becomes a multi-label classification problem, the most complex of the three types. Multi-label classification is similar to multi-class classification, with the difference that a document can belong to multiple classes or none of them. Similar to the binary classification, in practice, the categories often represent properties that a document might have, except that here, a single document can have any number of these properties. As such, this type of task could be broken down into a series of binary classifications. It can also be modeled as a multi-class task with $2^{|\mathcal{C}|}$ classes, where every class represents a combination of labels. When doing this conversion, one should be aware that, depending on the exact implementation, some correlations between predictions might be lost, making the classification process more difficult than it actually is.

Sentiment analysis is a special case of text classification. Usually with these tasks, there are a predetermined number of sentiments which every document might have. They can take the form of any of the three classification types. A multi-class task might have the mutually exclusive classes of *positive*, *neutral*, and *negative*, like the Stanford Sentiment Treebank¹ for example. In a multi-label task, a document might display several sentiments or emotions at once, or be completely devoid of them. These tasks can be approached with either traditional machine learning methods, such as support vector machines (SVM) and Naive Bayes (Joachims, 1998), or with neural networks, which will be explored in the next sections.

¹<https://nlp.stanford.edu/sentiment/index.html>

2.3 Convolutional Neural Networks

Convolutional neural networks (CNNs) specialize in processing organized, grid based data. Since these networks are often used in image processing, the input often takes the shape of a two-dimensional grid, if the images are black and white, or a three-dimensional grid, if they have RGB information. The principle works on data of any dimensionality, though. The following explanation is based on the detailed descriptions by Dumoulin and Visin (2016) and Goodfellow et al. (2016).

The modern form of CNNs (LeCun et al., 1989) is based on neurobiological findings on how vision works in animals. At the heart of CNNs is the convolution operation. It is performed in the spacial domain with a kernel (sometimes also called a filter) that moves over the input. A convolution layer is defined by its kernel size k , stride s and number of different kernels n . k and s are both tuples with one entry for every input dimension. These entries must not be larger than the input size. n is a positive integer that defines how many different kernels get applied. Based on these variables, a number of kernels are initialized. The kernels are tensors of the same dimensionality as the input. A feature map is obtained by overlaying the corresponding kernel on the input and performing an element-wise multiplication, the results of which get converted into a single floating point value, typically by just adding them all together. This value is then passed through a nonlinear activation function. This is done on a number of predefined positions in the input defined by the kernel size and stride. The stride defines how much the kernel moves in between each calculation. With a stride of one, the kernel position shifts by only one element from one calculation to the next. This creates an overlap, where most elements are part of multiple kernel calculations. Thus, if the stride is equal to the kernel size, there is no overlap and every entry of the feature map is calculated based on different, disjunct parts of the input. The resulting output is one feature map for each of these kernels. The relationship between these parameters, the size of the input i and size of the resulting feature map m in dimension d is as follows: $m_d = \frac{i_d - k_d}{s_d} + 1$. The values in the kernels are the trainable weights of the layer. Ideally, by the end of the training, every kernel has specialized in detecting different features of the data.

These convolutions are usually followed by pooling layers. These are not trainable, they only exist to reduce the size of the intermediate representation at certain points in the network, by sacrificing less relevant information. This in turn reduces the number of trainable weights in some of the following layers, greatly simplifying the network. These layers are defined by the pooling method, a window size and a stride. The formula for the output size is the same as for convolution layers. CNNs mostly employ max-pooling to find the most prominent feature(s) on each of the feature maps. Pooling layers do not use an activation function, since they do not model neurons. A deep CNN usually consists of a number of alternating convolutional and pooling layers. After those, the data is flattened into a one-dimensional tensor so that it can be passed into a standard linear layer. The output shape of this final layer depends on the task. In a multi-class task it is usually a one-dimensional tensor with one entry for each class, which represents the probability or confidence that this class is the correct one. The softmax function is used as the activation function for this final layer, so that all the probabilities sum to one and to make the back propagation more effective. An illustration of this basic architecture can be seen in Figure 2.2.

Input \rightarrow (Convolution \rightarrow ReLU \rightarrow Pooling) $^+$ \rightarrow feed-forward network \rightarrow Output

Figure 2.2: Most CNN classifiers follow this basic architecture. The data gets passed through a number of alternating convolution and pooling layers, before a feed-forward network of linear layers produces the predictions.

Even though CNNs were introduced to solve image recognition tasks, they are surprisingly effective in many other areas, including text classification. Kim (2014) propose a simple CNN architecture for sentence classification consisting of one convolution and one max-pooling layer (Figure 2.3). By concatenating all the word vectors in a sentence, a two-dimensional input is created. The kernels of the convolution layer have varying window sizes in the dimension of the sentence length, but all of them encompass the whole vector embedding in the dimension of the word vectors. Using pretrained word embeddings with this architecture, they achieve competitive results on multiple datasets.

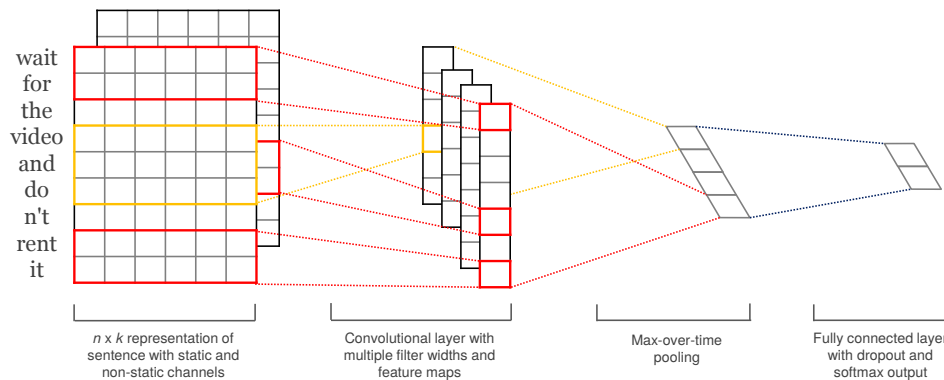


Figure 2.3: The diagram Kim (2014) use to illustrate their CNN architecture

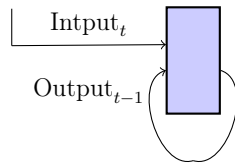
A more sophisticated architecture is proposed by dos Santos and Gatti (2014). Their model starts at the character level and uses a combination of one convolution and one max-pooling layer to compute a vector representation for each word, which is then combined with a pretrained embedding for that word. A full sentence consisting of these combined vectors is passed into a convolution layer, followed by another pooling layer, to get a sentence level representation, which is fed into a two-layer linear network to arrive at the predictions. They show that the addition of character based word vectors improves scores on their Twitter dataset.

2.4 LSTM Networks

Long Short-Term Memory networks (Hochreiter and Schmidhuber, 1997) are a special type of recurrent neural networks (RNNs). These types of networks are useful for processing sequential information of variable length. When multi-dimensional data is passed into an RNN, one of those dimensions is interpreted as time. The input is divided into discrete time steps along this dimension. These steps are then fed into the network one after another. At every point in this sequence, in addition to the input data of that step, the network also receives its output of the last time step as input. That way, the output at

every time step is influenced by everything that came before. Since there is no previous output on the first time step, the network usually starts with a randomized output. RNNs can be bidirectional, in which case it produces a second output that is the result of starting with the last time step and going backwards. See Figure 2.4 for an illustration of this process. Since the output at the final time step should contain all relevant information, most networks only use the final output for further processing. When used this way, RNNs have a constant output length despite having a variable input size, which makes them a good fit for language processing where samples often differ in length. Some applications, e.g. sequence taggers, also make use of the output at specific time steps other than the last.

Basic recurrent neural network:



Unrolled:

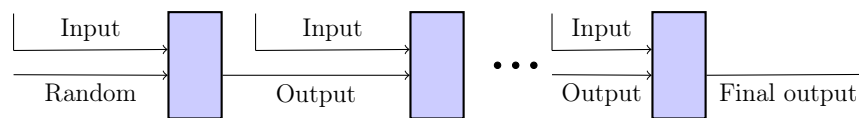


Figure 2.4: RNNs can be unrolled to illustrate how they process data.

One disadvantage of this process is that it will often prioritize later time steps, because the gradient either vanishes or explodes as it gets further away from the end. LSTMs were developed to improve upon RNNs by mitigating this weakness (Hochreiter and Schmidhuber, 1997). They do this by introducing a cell state that can retain important information. This state is controlled by a set of gates. Every time step, these gates can decide to let in new and push out old information. They are a trainable part of the network so they should learn to only keep relevant data in the cell state.

2.5 Capsule Networks

Capsule networks are a proposed enhancement to CNNs for image recognition tasks, aiming to improve results by emulating biological vision more closely (Hinton et al., 2011; Sabour et al., 2017). There are two major parts to understand about these networks, the capsules themselves and the way that they can be used classification networks. Formally, a capsule is a part of a network that receives an input tensor and outputs a vector. Since single neurons usually only output a single scalar, it can be thought of as a more complex version of those. The purpose of these vectors is that their entries are instantiation parameters for entities, which are learned automatically instead of being feature engineered by hand. An entity in this context is a property or pattern which occurs throughout the dataset and appears in many if not all of the samples. In image processing, an entity could be something as simple as a line or as complicated as a face. Like the instantiation parameters,

these are not defined beforehand and are instead learned during training. Occurrences of these entities in the dataset will have various differences between them, which can be abstracted as transformations. The instantiation parameters put out by a capsule would be equivariant with transformations of the corresponding entity in the input. This means that, for all important features, there is not just a single value representing the confidence that they are present, but multiple values indicating their properties in this particular instance. This should make the network more robust against affine transformations of the input, since they could easily be modeled by the capsule output. One admitted weakness of this approach is crowding. Since one capsule only deals with a single entity, if many instances of the same entity are grouped closely together, the network will have trouble detecting them all and differentiating between them. (Sabour et al., 2017)

To ensure, that the instantiation parameters learned by the capsule are relevant and useful, they are passed into a reconstruction module. Using the capsule as an auto-encoder and minimizing the difference between the reconstruction and the target should encourage the capsules to have the output vectors be the best possible representations of the input and contain the most relevant transformations. If the primary purpose of the network is to classify, then this should only be a small part of the total loss. In that case it works as a regularization method, since it makes the output more generalizable (Sabour et al., 2017). Wang et al. (2018) report good accuracy scores when LSTM based capsules are used for sentence classification in this way.

The next step is to build a deep network with multiple capsule layers that takes full advantage of their output format. Assuming that the network is supposed to detect complex entities that are made of multiple simple entities, capsules can be used to represent this parent-child relationship. If a capsule represents a simple entity, then its output vector contains information about the transformations applied to a particular instance of the entity. If multiple child entities predict the same transformations for a parent entity that they are all part of, then the probability is high, that the parent entity is present with these predicted transformations. This is the idea behind "routing by agreement". The exact implementation of this idea can vary, but Sabour et al. (2017) propose the following process to calculate the output of capsule layer c_2 based on the output of capsule layer c_1 :

1. The probability that the entity detected by a capsule is present is represented by length of the output vector. To facilitate this, the output vector v is "squashed" so that $\|v\| \in [0, 1]$. This is accomplished with the following function: $v_{squashed} = v \cdot \frac{\|v\|}{1+\|v\|^2}$
2. A four-dimensional weight matrix W links every entry of every capsule in c_1 to every entry of every capsule in c_2 . This not only defines, which output parameters of c_1 affect which output parameters of c_2 , it also represents base probabilities of which child entities are likely part of which parent entities. These weights are learned as the network trains.
3. Multiplying the output of c_1 with W and squashing the result produces the first set of predictions. The capsules of c_1 then divide their output among the capsules of c_2 based on these predictions. Capsules with a longer prediction vector receive a bigger portion of the output. Repeating this process with the adjusted outputs of c_1 yields the next set of predictions. After a number of repetitions, the predictions are used as the final output. The paper suggests three iterations.

The use of this algorithm separates the capsules layers in a network into two types. Primary capsule layers contain capsules that detect simple entities in the input and produce output vectors. How exactly they do that is up to the implementation and can differ from task to task. Capsule layers whose input is the output of another capsule layer are secondary capsule layers. They produce their output via routing algorithm. Primary capsules are usually place-coded in that every capsule only sees a part of the input (Sabour et al., 2017). That means the position of a detected entity is encoded by which capsule detected it. Since the routing connects every capsule of the previous layer to every capsule of the next, this specific property is lost in the process. However, this information should be translated to be part the output parameters of the secondary capsule layer by the routing. So it is still there, except that now, it is rate-coded. It is also possible to circumvent this transition by having groups of capsules route their output to only a subset of the capsules in the next layer, which would preserve the place-coding to some degree. Conceptually, capsules in later layers should have more output parameters, since they represent more complex entities and need to store previously place-coded data.

This division provides a basic framework for building a capsule network classifier. Since the length of the output vector of a capsule represents a probability, an additional feed-forward network to produce the predictions is not needed. The last layer of a network can just be a secondary capsule layer with one capsule for each class. In this case, the entity each capsule represent is decided beforehand. This layer can be preceded by more secondary capsule layers or be directly connected to a primary capsule layer, which in turn, can operate directly on the input or receive a representation that was already processed by more traditional neural network layers. A simplified illustration of this architecture can be seen in Figure 2.5.

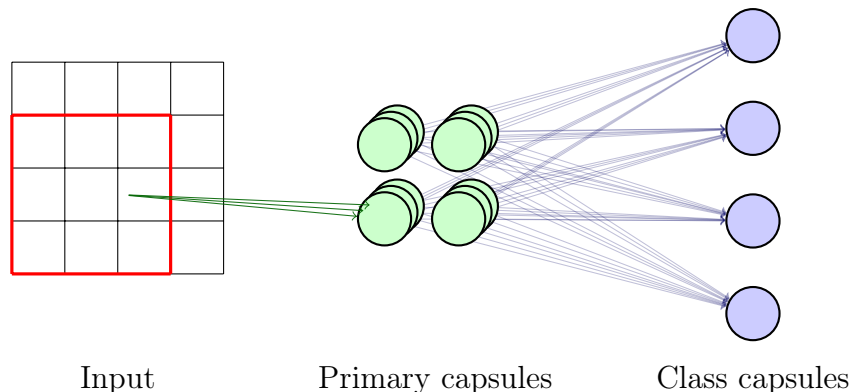


Figure 2.5: Basic structure of a simple capsule network where the primary capsule layer is implemented with a convolution layer.

Sabour et al. (2017) achieve promising results on the MNIST dataset² with a network consisting of one convolution layer, one primary capsule layer and one secondary capsule layer. The last layer functions of their model functions as an output layer and is connected to a reconstruction module. Their primary capsule layer is implemented as a convolution layer with each kernel position defining the receptive field of a group of primary capsules. This is done by arranging the resulting feature maps in such a way that a number of

²<http://yann.lecun.com/exdb/mnist/>

entries, all taken from the same position in different maps, are used as the entries of the capsule output vectors. Therefore, the number of primary capsules is defined by the size of the input and kernel, the stride, and the number of filters, since those determine the output size of the convolution layer. The network seems especially good at recognizing multiple digits in a single image after having been trained on one digit per image. A similar architecture has also been used for text classification. Yang et al. (2018) present a deeper network with one additional routing layer. This layer has restricted connections so that the place coding is preserved until right before the final capsule layer. They achieve good results, showing that capsules have great potential for transferring from single to multi-label problems on this task as well.

3 Methodology

This chapter describes our methodology detail. It includes a detailed description of the preprocessing steps, as well as the concrete implementation of the models that are evaluated. Lastly we describe the hyperparameter optimization process. Everything is implemented in Python3, using the Pytorch framework¹ (version 1.0) for all neural network related aspects.

3.1 Preprocessing

As mentioned in Section 2.1, raw text needs to be preprocessed before it can be fed into a neural network. Not only does the text need to be converted into a floating point representation using word vector embeddings, additional clean up can remove noise and make training more effective. Upon loading all data samples as a list of strings and all labels as a list of integers, the following preprocessing steps are applied:

1. Replace all sequences of whitespace characters with single spaces.
2. Remove whitespace at the start and end of samples.
3. Remove samples of length zero. Of the two datasets used in the evaluation this step removes six samples from one dataset and none from the other.
4. Replace alternative characters. Some unicode characters look almost identical to ASCII characters. If they are also used the same way, there is no need to differentiate between them. For example, the fullwidth exclamation mark (U+FF01) is replaced with the standard ASCII exclamation mark.
5. Tokenize with spaCy².
6. Convert all text to lowercase. We use the GloVe 42B word embeddings³ which are trained on lowercased text (Pennington et al., 2014).
7. Careful spell check: Replacing misspelled words with their correct counterpart improves the effectiveness of word embeddings. It also introduces possible mistakes, so this should be done carefully. In our approach, only tokens that occur no more than once in the corpus are considered for correction. We assume a token is a misspelling of another word, if the other word occurs more than once in the corpus and is within a string editing distance of one. If there are multiple candidates, the one that occurs more often is chosen. Despite being very conservative, this method will still lead to mistakes, but should overall improve the quality of the data.

¹<https://pytorch.org/>

²<https://spacy.io/>

³<https://nlp.stanford.edu/projects/glove/>

8. Filter out stop words and non-alphanumeric tokens. This reduces noise and increases information density.
9. The embedding process is started by collecting a vocabulary of all occurring tokens and a set of all tokens that occur only once. The former is used to create a dictionary that maps every word to a unique integer. The latter is used in step 14.
10. Perform an analysis of the dataset to determine the size of the vocabulary, sample length, class distribution, and other statistics. This only serves to provide insight into the dataset. See section 4.1 for specific details.
11. In order to train with a batch size other than one, all samples must all have the same length. To achieve this, a maximum length is chosen. Every sample that is longer is truncated, every sample that is shorter is padded. We choose the 75th percentile of sample lengths for this maximum length.
12. Convert the samples from words to integers using the dictionary constructed in step 9. At this point in the pipeline, the samples and dictionary are saved, so that previous steps can be skipped, the next time the dataset is loaded.
13. Load pretrained word embeddings. Construct a dictionary, that maps every word integer to the corresponding word vector in the embedding. Use a random vector for words not present in the embedding.
14. Every token that occurs only once in the entire corpus and is not present in the pretrained embeddings contains little usable information. To prevent overfitting on these low frequency unknown tokens (referred to as LFUTs from now on) and to keep the size of the embedding layer reasonable, all LFUTs are replaced with a generic *<lfut>* token. This token is then mapped to a single, randomly generated vector.
15. Add the padding token to the word-to-int dictionary and its embedding vector (the zero vector) to the integer-to-vector dictionary.
16. Convert the samples from a list of integers to a one-dimensional tensor of integers.
17. Normalize all embedding vectors. This process loses some information, but it is necessary to keep all input values between -1 and 1. This is especially relevant if a reconstruction module is used, since it limits the value range of the input to $[0, 1]$.

The final conversion of the samples to a two-dimensional tensor of floating point values happens within the networks themselves. The first layer of every network is an embedding layer, initialized with the integer-to-vector dictionary. After that, every network proceeds differently.

3.2 CNN and LSTM Network

The convolutional neural network used in the experiments is based on the architecture described by Kim (2014). As such, the convolution window covers the whole word embedding, meaning the resulting feature maps are one-dimensional. The only difference to the paper, is that this CNN only uses one window size. This simplifies the implementation and reduces the search space for hyperparameter optimization. The resulting network

therefore consists of the following layers in order: Input \rightarrow embedding layer \rightarrow convolution layer \rightarrow max-pooling layer \rightarrow linear layer \rightarrow output.

The LSTM network is kept just as simple, for the purpose of comparison. Instead of the convolution and pooling layers, it has one LSTM layer. Unlike the other networks in this comparison, it receives the original length of the sample as an additional input in every batch. This is necessitated by the padding contained in most samples. The output of the LSTM layer contains the results of every step. This includes the time steps that only contain padding information, so the original length is used to pick the last relevant one. Doing it this way precludes the effective use of a bidirectional model without significant workarounds, since starting at the end and going in reverse make the network go through the padding first. The LSTM layer is therefore unidirectional. No additional regularization layers like dropout (Srivastava et al., 2014) are used in order to make the comparison to capsule networks more fair. Since dropout layers are usually placed in between two linear layers that produce the final prediction and capsule networks do not use linear layers there, any implementation of them would be inherently uneven. Since Pytorch provides implementations of both convolution and LSTM layers, the code (see Listing 3.1) for these is quite short. Figure 3.1 illustrates the CNN and LSTM models.

```

1  def forward(self, x):      #CNN
2      x.unsqueeze_(1)        # 1 input channel for the convolution layer
3      x = F.relu(self.conv(x))
4      x = self.pool(x)
5      x = x.view([...])      # Changing the format so it can be passed to self.lin
6      x = F.softmax(self.lin(x), dim=1)
7      return x
8
9  def forward(self, x, length): #LSTM
10     h0 = torch.rand([...])   # randomly initializing hidden layer
11     c0 = torch.rand([...])   # randomly initializing cell state
12     x, hidden = self.lstm(x, (h0, c0))
13     # Taking the last relevant time step:
14     x = torch.stack([x[i, length[i]-1] for i in range(x.size(0))])
15     x = torch.relu(x)
16     x = F.softmax(self.lin(x), dim=1)
17     return x

```

Listing 3.1: Code of the forward passes of the CNN and LSTM network. These snippets are simplified from the actual source code to be more readable.

In addition to the presented CNN, we examine another network with an alternative pooling method, inspired by the capsule networks. The output of the convolution layer, is a set of one-dimensional feature maps. The subsequent pooling layer has a window size equal to the length of these feature maps, which means that every map gets reduced to a single value. The most common pooling mechanism for this is max-pooling, which is what we use for the standard CNN. However, other pooling mechanisms, like average-pooling or min-pooling would work as well, but would likely differ in effectiveness. A one dimensional feature map can be interpreted as a vector, which opens up additional possibilities for reducing it to a single value. The alternative CNN uses the exact same architecture as the previously presented one, but instead of using max-pooling, it takes the squashed length of the feature map as if it were a vector. Since the squashing acts as a activation function, it is used instead of ReLu after the convolution layer. This variation can be thought of as something of a compromise between max-pooling and average-pooling. Since taking the length of a vector squares all entries before the sum, large entries will have a disproportionately larger impact on the resulting value than smaller entries. But small

w : Window size
 s : Stride
 f : Filter number

l : Input size
 e : Embedding size
 c : Number of classes

h : Hidden dimension

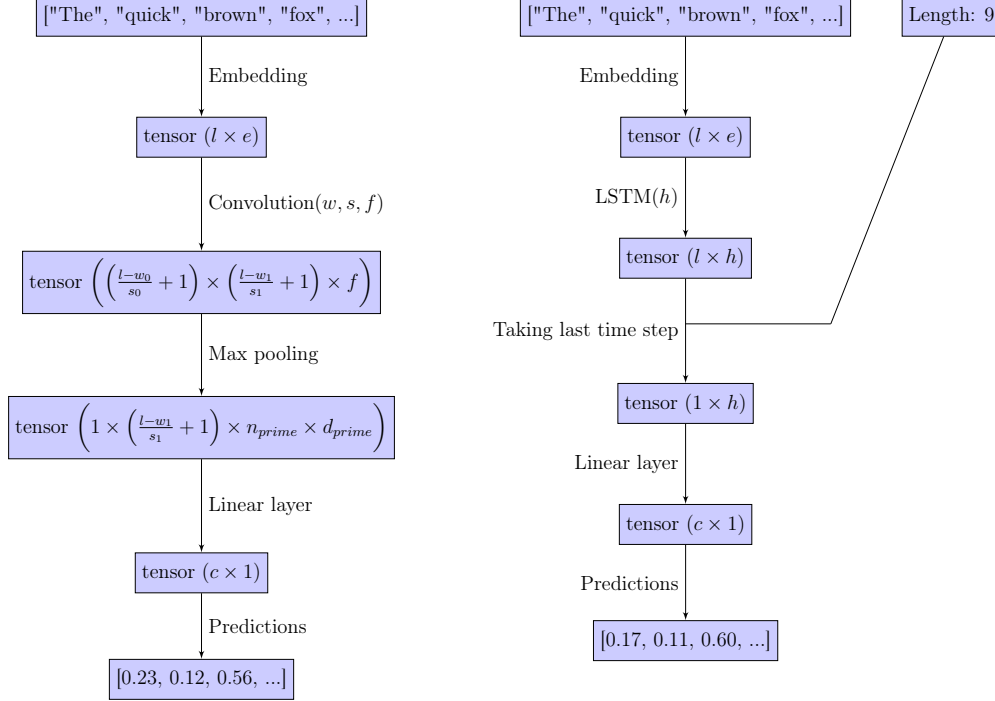


Figure 3.1: Architecture of the convolutional and LSTM networks.

values do still affect the sum, which differentiates it from max-pooling. None of the pooling methods take the order of entries into account, so this method would still work with differently sized pooling windows. The entries of that window only have to be reshaped into a vector. This variation of the CNN receives no extra hyperparameter optimization. It uses the same hyperparameter settings as the standard CNN. We will refer to this variation as squash CNN during evaluation.

3.3 Capsule Network

The implementation of the capsule network follows Sabour et al. (2017), with the exception that the first convolution layer is omitted. That way, the number of trainable layers is equal between all three of the networks, making the architectures and thus the performance more comparable. The reconstruction module too is omitted for that reason and others. Using a regularization method in only one of the networks would make for an unfair comparison. In addition, using reconstructions on text presents a number of difficulties, which are discussed later. So to avoid potential caveats to the end results, this aspect of capsule networks is not used in our evaluation. An overview of the implemented architecture can be seen in Figure 3.2.

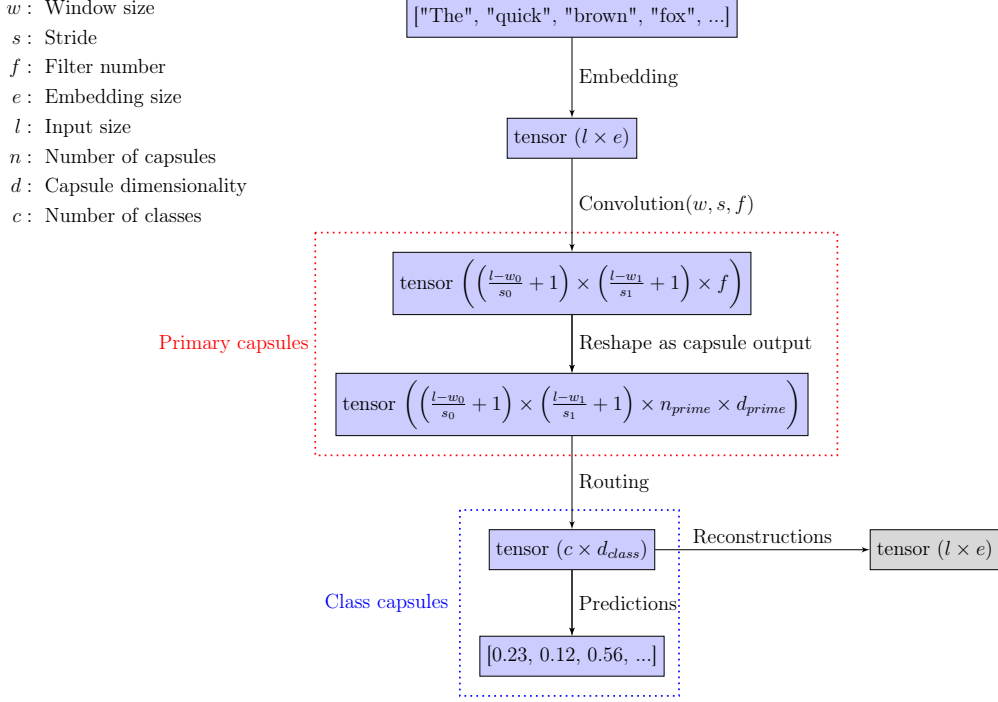


Figure 3.2: Architecture of the capsule network implementation. For our purposes w_1 is always equal to e . The diagram also shows where a reconstruction module would fit in if one was used.

The first capsule layer is implemented as a standard Pytorch convolution layer. Its output is reshaped to represent groups of capsules with limited receptive fields, as described in the paper. For a more direct comparison to the CNN, the kernel size of the convolution encompasses the entire embedding dimension in this network as well. This will also help with hyperparameter optimization, since it eliminates one variable. Considering that capsule networks already have the most hyperparameters of the three network, this choice should help with finding a good setting in a reasonable amount of time. The class capsule layer does not have an equivalent in Pytorch, so a custom implementation of the dynamic routing algorithm is used. To verify that this implementation works as intended, it was tested on the MNIST dataset, where it produced results comparable to the ones described in the paper. Listing 3.2 shows the source code of the implementation. In order to make the implementation more efficient, most steps are formulated as matrix multiplications. Matrix multiplications are very well optimized both algorithmically and in their implementation, so using them as much as possible is important to avoid ending up with impractically long training times.

```

1 class CapsuleLayerDense(nn.Module):
2     def __init__(self, in_caps_num, in_caps_dim, num_capsules, output_dim,
3               iterations=3):
4         super(CapsuleLayerDense, self).__init__()
5         self.in_caps_num = in_caps_num; self.in_caps_dim = in_caps_dim; [...]
6         weights = torch.randn(1, num_capsules, in_caps_num, output_dim, in_caps_dim)
7         self.weights = nn.Parameter(weights)      # Making the weights trainable
8
9     def forward(self, x):
10        b = torch.zeros((x.size(0), self.num_capsules, self.in_caps_num),
11                        requires_grad=True)
12        x = x.unsqueeze(1).unsqueeze(-1) #Adjusting format for the coming operations
13        predictions = torch.matmul(self.weights, x).squeeze(-1)
14        for n in range(self.iterations):
15            coupling = F.softmax(b, dim=-1).unsqueeze(-2)
16            output = torch.matmul(coupling, predictions).transpose(-1,-2)
17            if n != self.iterations - 1:      # Update coupling for next iteration
18                output = squash(output, dim=-2)
19                output = torch.matmul(predictions, output).squeeze(-1)
20                b = b + output
21        return output.squeeze(-1)

```

Listing 3.2: Implementation of the routing process. Matrix multiplications are used wherever possible to optimize performance. This snippet is slightly simplified for readability

Since the vector lengths of the class capsules can be used as the final prediction, no additional linear layers are necessary to produce the output. But some additional processing before calculating the loss can make training more efficient. The squashing function tends to reduce the length of vectors to a very small value, especially in a newly initialized network. As a result, the numeric difference between the probabilities for each classes can also be very small. To mitigate this, the prediction vector, which contains the lengths of the class capsule vectors, is also normalized. This emphasizes the differences between classes and avoids situations where multiple classes end up having the highest probability at the same time. Initial tests confirm that this causes the network to make faster progress during training. We also use a simplified squashing calculation to avoid dividing by the vector length. In general, when dealing with the capsule output vectors, it is important to be aware of any calculations that involve normalizing or squashing them, since that would lead to errors with the zero vector. It can easily lead to a NaN (Not a Number) value, which, through back-propagation will break the whole network. In our case, the padding consists of zero vectors, at least initially, so capsules operating exclusively on padding are likely to put out a zero vector as well. As a precaution, when the norm of the vector is used as a denominator, a small ϵ should be added to avoid a division by zero. This will prevent errors, but it can still cause problems during training. That is why we use a simplified version of the squashing formula which does not divide by the norm. If the unaltered squashing function is used, the padding vector should be randomly generated to avoid this problem.

3.4 Hyperparameter Optimization

The efficacy of a network is not just dependent on its architecture. Proper hyperparameter settings also play a big role. These include properties like the the kernel size in convolution layers or the dimension of the hidden layer in an LSTM layer. Unfortunately, just choosing the largest possible setting is rarely practical or optimal. Since memory and computing

time are limited in realistic applications, some hyperparameters will have to be prioritized over others. But even maxing out the most important ones is often not the best solution. Arbitrarily chosen hyperparameters make a comparison between networks meaningless, since it is easily possible, that some networks ended up with much less optimal settings by chance. Therefore, an automatic optimization is necessary. The most effective approach to this problem is something close to an exhaustive search usually implemented as a grid search.

To optimize the models detailed in the previous section, every hyperparameter for every network type gets assigned a range of possible values. In addition to network specific ones, the batch size and learning rate also have a big influence over the result and should therefore be included. For every possible combination of these hyperparameters, a network is trained and then evaluated on a separate part of the dataset. This is done in a random order, so that trends can be seen more easily before the search has been fully completed. Since this is a very time intensive process, it is important to make the search as efficient as possible. Some hyperparameter combinations can be disregarded without the need to train a model with them. If the stride of a convolution layer is bigger than its kernel size, some data in the input would be skipped. A network that carelessly ignores parts of the data is probably not going to be optimal, therefore these settings can be skipped. Similarly, the difference between the input size and the kernel size should not be smaller than the stride, since this will result in the exact same kernel positions as other settings with a smaller stride. The training should be as deterministic as possible, so evaluating what is effectively the same network multiple times is not necessary.

In order to train a number of networks sequentially without manual interference, conditions must be established, under which the training of a network ends. We use the following conditions, training ends if one of them is met:

- The loss has decreased by less than a given delta for the last 3 epochs. This indicates, that the network has practically converged and will not improve much more, even with further training.
- A certain maximum number of epochs is reached. This number is set before the search is started. It is the same for every network, but can differ between datasets. An epoch is defined as number of samples, so that it is the same for every dataset.
- The output of the network or the loss contains NaN. This might happen for various reasons, the most common of which is a division by zero somewhere in the network. In this case, a checkpoint from the previous epoch is used for the evaluation.

When training only one network, the test score is a useful metric to determine when to stop. If it does not increase over several epochs, the training should end. Using this metric in the grid search is not really feasible, though, since evaluating the network after every epoch significantly increases the search time. When one of the exit conditions is met and training ends, the network is evaluated on a validation set. This set is completely disjunct from both the training and test set. It is usually the smallest of the three, making up about 10% of the whole dataset. The training data is 70% of the dataset and the remaining 20% are the testing data used for the final evaluation. After all hyperparameter combinations are processed this way, the best one is determined by the highest F1 score achieved in the evaluation step.

Finally, the best networks, as determined by this method, are evaluated on the test set. It is possible that the network with the best F1 score is especially good at learning to predict the particular samples contained in the validation set, so to get a more general idea about the performance of the network, a different part of the dataset, the test set, is used for the final evaluation.

4 Evaluation

This chapter contains the evaluation and comparison of the networks on different datasets. The two datasets used for this evaluation are introduced and their properties and statistics are detailed and discussed. Before comparing the scores and training statistics of the networks, the results of the grid search are presented.

4.1 Datasets

The models are evaluated on two datasets. For both of them, the networks will operate purely on the text and word vector embeddings. No additional features or sentiment annotations per word or sentence are used as targets or as data for predictions.

Properties of User Generated Online Texts

The processing of short informal online posts presents many difficulties not found in more traditional literature. Correct tokenization and subsequent embedding can be challenging because of this. Two of these aspects are incorrect and unconventional spelling and formatting. Without additional preprocessing, a misspelled word would not map to the vector of the word it actually represents. Unless the exact misspelling is common enough to appear in the pretrained embedding, it either receives a randomly generated vector or be classified as a LFUT. This goes for spelling mistakes ("apparently" instead of "apparentlly"), typos ("teh" instead of "the") and repeating characters for emphasis ("waaaaayyy too much"). A person can easily recognize the intended word, but standard word embeddings are not a nuanced process. Either that exact word exists in it or it does not. A spell check can be employed to help with this issue, but the question of which words to correct is not trivial. In such an informal environment, some colloquial or slang expressions might be unknown to the spell checker and lead to false corrections. Among other things, the keyboard layout should be taken into account when deciding what the correct version of a misspelled word is. For example, leaving other aspects aside, the incorrectly spelled "solition" should be corrected to "solution" rather than "volition", since the letters *u* and *i* are next to each other on a qwerty keyboard, but *s* and *v* are not. Most standard spell checks and tokenizers are also unprepared to deal with unusual formatting like putting a space in between every letter of a word, omitting the spaces between words, or the repetition of characters mentioned above. Then there are intentional misspellings, which can occur when a person is mocking someone else or emulating an accent. In general, this can be thought of as a difficult denoising problem.

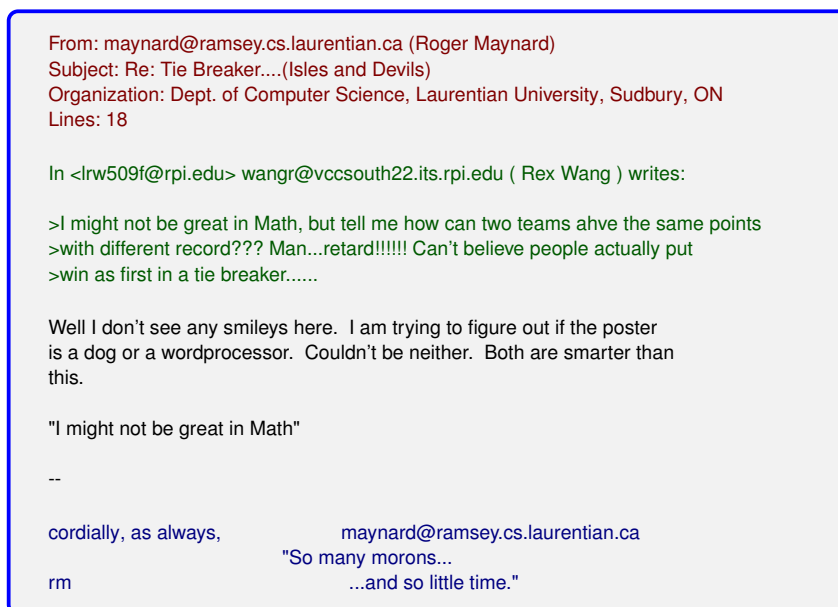
Sophisticated methods to correct these kinds of misspellings (Wint et al., 2017) and replace slang (Singh and Kumari, 2016) exist, but transformative approaches always lose some information in the process. All of the discussed aspects leave an impression on a human reader who, in most cases, still recognizes the original words underneath. These

often stylistic alterations provide a layer of meaning in addition to what a proper, correct version of the text would contain. Trying to model these aspects and taking all of these possibilities into account would require a much different preprocessing methodology and different, more complex network architectures to take advantage of them.

20 Newsgroups

The 20 newsgroups dataset¹ is a collection of almost 19,000 documents, each belonging to one of twenty different newsgroups. Predicting the newsgroup a document belongs to based on its content, is a multi-class, single-label problem. Their style is mostly informal and they often take the form of inquiries and discussions. This dataset specifically is not related to sentiment analysis, it represents a straight-forward text classification task. As such, it is a good basis, to test the pipeline and networks and to get a general idea about the potential of each of the evaluated models.

We use the scikit-learn implementation² of this dataset, which provides the option of filtering out headers, footers and quotes from other users. See Figure 4.1 for an example.



From: maynard@ramsey.cs.laurentian.ca (Roger Maynard)
Subject: Re: Tie Breaker....(Isles and Devils)
Organization: Dept. of Computer Science, Laurentian University, Sudbury, ON
Lines: 18

In <lrw509f@rpi.edu> wangr@vccsouth22.its.rpi.edu (Rex Wang) writes:

>I might not be great in Math, but tell me how can two teams ahve the same points
>with different record??? Man...retard!!!!!! Can't believe people actually put
>win as first in a tie breaker.....

Well I don't see any smileys here. I am trying to figure out if the poster
is a dog or a wordprocessor. Couldn't be neither. Both are smarter than
this.

"I might not be great in Math"

--

cordially, as always, maynard@ramsey.cs.laurentian.ca
rm "So many morons...
 ...and so little time."

Figure 4.1: This is one sample form the 20 newsgroups dataset. It belongs to the class *rec.sport.hockey*. The header is marked in red, quotes in green and the footer in blue.

The user guide recommends removing all three to prevent the model from classifying based on user names and article IDs. Many of the samples heavily feature quotes however, often with very little original text, so removing them removes most of the context. Additionally, in some of the samples, the actual text of the message is mistakenly marked as being part of one of those three, so removing them all results in a quite a few empty samples, which have to be discarded. With this in mind, we filter out headers and footers, but keep the quotes. Thus, the total amount of samples is slightly reduced, but the resulting network would

¹<http://qwone.com/~jason/20Newsgroups/>

²https://scikit-learn.org/0.19/datasets/twenty_newsgroups.html

likely generalize better to documents from a different time frame. Leaving in quotations will leave in article IDs and user names, but since they contain non-alphanumeric characters, they are filtered out later in the preprocessing. Leaving in quotations still has a noticeable effect, though, even after filtering. Many quotes begin with some form of "In article <id>, <name> writes:". Since both "article" and "writes" are alphanumeric and not stop words, they end up being the two most common tokens in the dataset after filtering.

	Unfiltered	Filtered
Average sample length	310.4	106.9
75 th percentile length	315	106
Number of different tokens	149K	63K
Type/Token ratio	0.0255	0.0314
10 most common tokens ¹	<i>the to of a and i in is that it</i>	<i>writes article like people know x think use time good</i>
% ² occupied by top 10	23.9%	4.1%
% ² occupied by LFUT	1.0%	0.2%

¹ Not counting common punctuation

² Percentage of the whole corpus

Table 4.1: Statistics of the 20 newsgroups dataset. These are created during the preprocessing described in section 3.1. The lengths are in number of tokens. The token "x" occurs often in the technology related newsgroups.

Comparing the statistics for the filtered and unfiltered dataset (Table 4.1), the benefits of filtering stop words and non-alphanumeric tokens become clear. In the filtered dataset, the most common tokens occupy a much smaller percentage of all data. The same goes for LFUTs. Their high number in the unfiltered data is likely due to the prevalence of user names and article IDs. Since both the most common tokens and the LFUTs provide little basis for predictions due to their presence in most samples, reducing these percentages should increase the density of relevant information and thus, reduce noise. Another statistic, which is significantly reduced by filtering, is the length of the samples. The 75th percentile length, which is what all samples are trimmed to as part of the preprocessing, is reduced by about two thirds. Even though more information is always better in theory, short samples have some practical advantages, since network size often scales with sample length. This is the case for both the CNN and capsule network. To get results in a reasonable amount of time, networks are trained on GPUs. Since GPU memory is much more limited than RAM on most computers, this puts a limit on the number of parameters in a network, as well as the batch size. Even if memory is not an issue, smaller networks usually run significantly faster. As a result, more hyperparameter combinations can be tested via grid search, resulting in better settings.

The distribution of samples between the classes of the dataset is almost balanced, but they differ greatly in average length (see Table 4.3, left). Samples of *talk.politics.mideast* are almost 4 times as long on average as samples from *misc.forsale*. How these two statistics affect the accuracy for each class is an aspect worth looking into. It is very possible, that some classes have strong signifiers and are easy to predict, despite a low sample count or short length. Similarly, long sample length and high sample count are no guarantee that a class will be predicted with a high accuracy.

The state-of-the-art for this dataset achieves an F1 score of 92.6 (Shu et al., 2017)

2018 Semeval Task 2: Twitter Emojis

The English dataset of the 2018 Semeval Task 2 (Barbieri et al., 2018)³ consists of the text of roughly 450,000 tweets. Each of these tweets uses exactly one of the 20 most commonly used emojis, which is used as the label. It can appear any number of times in the tweet. The task is to predict the used emoji based on the text of the tweet after it has been removed. As such, it is a multi-class, single-label classification problem.

The organizers provide a script for gathering the dataset from Twitter. It also applies some text preprocessing steps as part of the collection process. These include removing all emojis, as the nature of the task necessitates, removing links, and replacing all user names with a generic "@user". For our evaluation this preprocessing is changed slightly. Since both the links and emojis can appear at different positions in the tweet and in varying numbers, removing them removes more information than necessary. Instead, links and emojis are replaced with the generic "<link>" and "<emoji>" tokens respectively. Figure 4.2 shows an Example. This alters the task slightly, but the goal is to compare different models to each other rather than to the state-of-the-art. The tokenization is also adjusted for this dataset. If the character "#" is followed by more characters without a space, it is not split into a separate token. That way, whole hashtags are preserved as a single token.

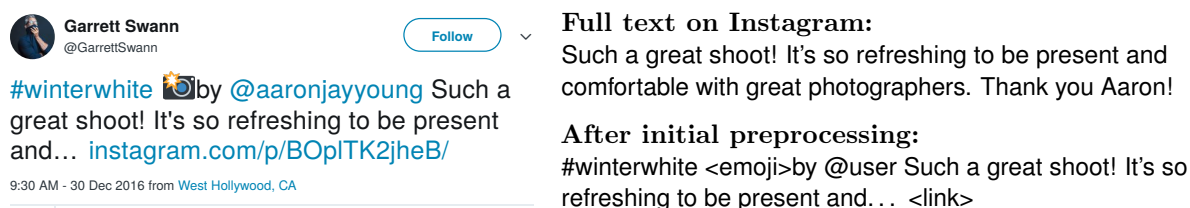


Figure 4.2: This is one sample from the English 2018 Semeval Task 2 dataset. It shows the original tweet on the left, the full text of the corresponding instagram post, and the sample after initial preprocessing.

Looking at the statistics in Table 4.2, filtering out stop words and non-alphanumeric tokens in this dataset is not as beneficial as it is in the 20 newsgroups dataset. It decreases the average sample length to only 6.4 tokens and actually increases the proportion of LFUTs in the dataset. Besides the statistics, there are other factors to consider for filtering out non-alphanumeric characters. People occasionally build emoticons and other images out of various, often obscure characters. In addition, it is likely, that punctuation, especially exclamation marks, contain relevant information in this domain. For these reasons, the dataset is not filtered. A more detailed filtering approach that keeps tokens like hashtags and meaningful punctuation would probably be the best option, but this type of optimization usually brings very little benefit for the amount of time it takes to perfect it.

The class distribution of the dataset (Table 4.3, right) is noticeably unbalanced, with the most common emoji, *red heart* (❤️), occurring almost nine times as often as the least

³<https://competitions.codalab.org/competitions/17344>

	Unfiltered	Filtered
Average sample length	15.9	6.4
75 th percentile length	19	8
Number of different words	284K	234K
Type/Token ratio	0.0394	0.0814
10 most common tokens ¹	<emoji> <link> @ ... the @user i my to a	love new day happy night today beach time york good
% ² occupied by top 10	29.8%	6.2%
% ² occupied by LFUT	2.4%	4.7%

¹ Not counting common punctuation

² Percentage of the whole corpus

Table 4.2: Statistics of the emoji prediction dataset. The @ being so common is not a tokenization mistake, many users use it in place of "at" when they describe where they are.





















Class	Share	Length	Class	Share	Length
rec.sport.hockey	5.3%	106.6		21.7%	16.3
soc.religion.christian	5.3%	140.7		10.5%	15.5
rec.motorcycles	5.3%	65.7		10.3%	17.1
rec.sport.baseball	5.3%	83.3		5.4%	15.2
sci.crypt	5.3%	128.7		5.1%	16.7
rec.autos	5.3%	76.6		4.7%	16.0
sci.med	5.3%	120.6		4.4%	15.1
comp.windows.x	5.2%	117.5		3.7%	15.7
sci.space	5.2%	118.4		3.4%	15.0
sci.electronics	5.2%	74.8		3.3%	16.3
comp.os.ms-windows.misc	5.2%	71.4		3.3%	15.6
comp.sys.ibm.pc.hardware	5.2%	69.9		3.1%	14.6
comp.graphics	5.2%	110.2		2.8%	15.4
misc.forsale	5.1%	57.7		2.7%	17.0
comp.sys.mac.hardware	5.1%	64.6		2.7%	15.3
talk.politics.mideast	5.0%	204.3		2.7%	15.9
talk.politics.guns	4.8%	127.7		2.7%	16.4
alt.atheism	4.2%	126.0		2.6%	14.8
talk.politics.misc	4.1%	170.0		2.6%	15.3
talk.religion.misc	3.3%	132.6		2.5%	16.3
Standard deviation	0.5%	37.4	Std. dev.	4.4%	0.7

Table 4.3: Class distribution and average length of samples in each class for the two datasets. The lengths are given in numbers of tokens for the filtered 20 news-groups dataset (left) and the unfiltered Twitter dataset (right).

common one, *winking face with tongue* (🙄). It is also worth noting, that some of the emojis are very similar to each other, e.g. the camera with flash (📷) and the camera without flash (📱). Other emojis have multiple, often overlapping meanings, which is why even humans struggle with this kind of task (Barbieri et al., 2017). In addition, tweets often link to pictures, which are ignored by a purely text based classifier. Many of them are reposts from Instagram, which are linked in the tweet. In those cases, the text of the tweet is a shortened version of the text in the Instagram post, ending with the horizontal ellipses character (U+2026). This explains, why it is one of the most common tokens in the unfiltered dataset. As a result, these tweets often lack important context information. All of these aspects make this particular dataset very difficult. The best system of the task achieves an F1 score of 35.99 with an SVM classifier (Barbieri et al., 2018).

4.2 Grid Search Results

The hyperparameter settings of the networks are optimized via grid search on both datasets. Because of the size and difficulty of the Twitter dataset, some concessions had to be made in the hyperparameter optimization. In order to get meaningful results in a reasonable amount of time, some settings had to be skipped. These choices were made based on the results of the same network on the 20 newsgroups dataset. If a parameter combination produced NaN during training or had an F1 score that is not significantly better than random guessing, no network is trained with the same settings on the Twitter dataset. With these specific exceptions, all network and dataset combinations have completed the search. The alternative CNN, which uses the squashed norm of each feature map instead of the maximum, is not optimized separately. It uses the same hyperparameters as the standard CNN in the final evaluation.

Parameter	Candidates	20 newsgroups			Twitter		
		CNN	LSTM	CapsNet	CNN	LSTM	CapsNet
Learning rate	[1e−4, 5e−4, 1e−3]	0.0005	0.001	0.001	0.0001	0.0005	0.001
Batch size	[8, 16, 32]	32	8	32	16	16	32
Kernel x	[3, 5, 10]	5	-	5	3	-	5
Stride x	[1, 2, 4, 8]	1	-	1	1	-	1
Filters	[100, 200, 400, 800]	800	-	512	200	-	512
Hidden dim.	[200, 400, 800]	-	200	-	-	200	-
Prime caps num.	[8, 16, 32]	-	-	32	-	-	32
Prime caps dim.	[8, 16, 32]	-	-	16	-	-	16
Class caps dim.	[8, 16, 32]	-	-	16	-	-	16

Table 4.4: Results of the grid search for all 6 network and dataset combinations. The number of filters is grayed out for capsule networks, because it is not set directly as a hyperparameter. It is determined by the number and dimension of prime capsules.

The Table 4.4 shows the results of the grid search. The full range of both the learning rate and batch size is represented in the results, confirming, that they are an important part of the optimization. While the optimal settings vary for both the CNN and LSTM networks depending on the dataset, the capsule network’s settings stay the same. The traditional networks decrease their learning rate going from the 20 newsgroups dataset to the Twitter dataset, probably because the latter is larger and more difficult. A lower learning rate is

less likely to skip over more nuanced differences between samples. The CNN also uses a smaller kernel size, which is expected considering how much shorter the Twitter samples are. The reason for the sharp decrease in the preferred filter number is not as easy to explain. It is possible, that the increased size of a CNN with a large filter number makes it more prone to overfitting. Interestingly enough, this does not seem to apply to the capsule network. Since the CNN goes to both extremes, a more extensive grid search with a wider range would be necessary to determine, how far in either direction the optimal setting lies. The same goes for the dimension of the hidden layer in the LSTM, which is the lowest value in the range for both datasets. Looking at the hyperparameters unique to the capsule networks, we can not confirm that the class capsule dimension should be larger than the prime capsule dimension. However, the grid search results do seem to indicate that it should not be lower.

The grid search presented here was by no means exhaustive, so these hyperparameter settings are not definitive. They do however show tendencies on what settings are better for what kinds of datasets. Having done this optimization also makes for a fairer comparison between networks.

4.3 Test Results

The final comparison between the networks is done by training models with the optimal parameter settings found by the grid search. A negative log likelihood loss and Adam optimizer (Kingma and Ba, 2014) are used for this training, the same as for the grid search. There are several reasons why a new model is trained instead of directly using the model created by the grid search. The first is logging training statistics. Since the grid search should be as fast as possible to examine as many settings as possible, additional logging could actually impact the end result. Considering that these logs are only relevant for one out of all examined models, doing it in an extra step is more sensible. During training, the validation F1 score and average loss are logged after every epoch. Every time a new best score is achieved, a checkpoint of the model is saved. When a number of epochs pass without the best score being beaten, the training stops automatically. After stopping, the last checkpoint contains a model that has not started overfitting yet. This is the second reason for training a new model. During the grid search, the difference in loss is used to determine when to stop training, so many of the models produced there end up overfitting, especially since no additional regularization methods are used. Training the final model in a separate step and using the validation F1 score as an indicator, avoids this issue. It also allows setting the samples per epoch a bit lower to more accurately hit the point where the model performs best. After training, the models are evaluated on the test set to get a final F1 score, which will serve as the main point of comparison. Macro averaging is used for both the validation and test scores. An overview of these scores in addition to some other statistics can be found in Table 4.5. The graphs showing the progression of the statistics during training can be seen in Figure 4.3.

The final F1 test scores for the 20 newsgroup dataset are good across the board. The squash CNN performs the best on this dataset with an F1 score of 88.14. This is considerably better than the standard CNN's score of 86.11. Despite changing only a small aspect of the network, the alternative to max-pooling has great effects on the performance in this task. The capsule network, too, performs better than the standard CNN. Scores are much

20 newsgroups dataset

	Test F1	Validation F1	Parameters ¹	Time per sample ²
CNN	86.11	85.37	1.2M	9.91 ms
Squash CNN	88.14	87.87	1.2M	10.21 ms
LSTM	83.26	82.63	406K	10.83 ms
CapsNet	87.51	87.86	17.5M	27.53 ms

SemEval 2018 Twitter dataset

	Test F1	Validation F1	Parameters ¹	Time per sample ²
CNN	35.40	35.49	184K	11.17 ms
Squash CNN	34.31	34.47	184K	11.76 ms
LSTM	35.31	35.36	406K	12.60 ms
CapsNet	31.48	31.49	3.2M	16.01 ms

¹Model parameters excluding the embedding layer

²When trained with a batch size of 1

Table 4.5: These two tables show the test results of the different networks on the 20 newsgroups dataset (top) and the Twitter dataset (bottom). Scores are out of 100. The time per sample is supposed to serve as a point of comparison for the computational complexity of the networks, rather than a precise measurement of the individual performance.

lower on the more difficult Twitter dataset. The best score here, 35.40, achieved by the standard CNN, gets close to the best score (35.99) of the shared task (Barbieri et al., 2018). Although, this is most likely due to the different preprocessing rather than the strength of the model itself. The capsule network performs much worse on this dataset, even when accounting for the higher difficulty, achieving a score of only 31.48. Both the LSTM network and the CNN perform well here, but not quite as good as the standard CNN. Comparing the test F1 scores with the validation F1 scores, the capsule network has the lowest difference between the two on both datasets. This suggests that they are a bit more stable in their performance than the other networks, but to confirm this, a cross validation is necessary.

Technical Aspects

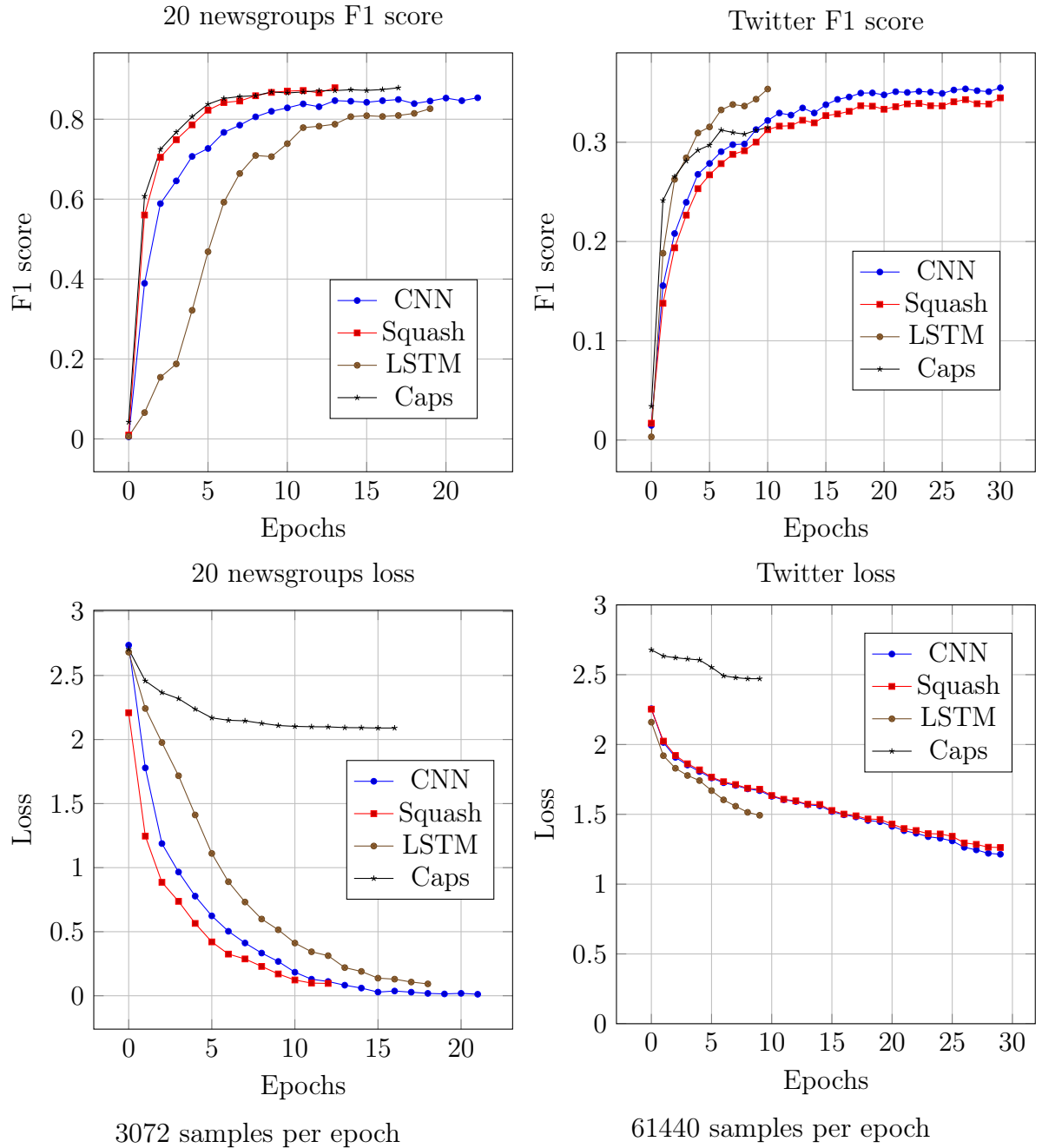
In addition to the scores and training statistics, there are some technical aspects that are worth looking into. Out of the tested networks, the capsule network is by far the most resource intensive. It has not only the highest parameter count, but also takes the most time per sample on both datasets. This is caused by the class capsule layer, which, as a secondary capsule layer, implements the routing by agreement algorithm. This routing algorithm uses a four-dimensional tensor, which contains the trainable weights of that layer. Every entry of every ingoing capsule vector is connected to every entry of every outgoing capsule vector with a unique weight. Thus, it scales heavily with the sample length, the number and dimension of the prime capsules, and the number and dimension of the class capsules. In addition, the routing iterations can not be parallelized, further increasing the computation time. Even when using efficient matrix multiplications for

all calculations that allow it, the sheer amount of parameters makes capsule networks significantly slower than most other network types. Since the samples in the Twitter dataset are much shorter than those in the 20 newsgroups dataset, the capsule network, as well as the CNN are much smaller on there. In most cases this would mean that they are also faster, but this is not true for the CNN here. The reason for this is most likely the much larger vocabulary of the Twitter dataset, which heavily increases the size of the embedding layer. The capsule network is faster on this dataset, though. It seems that its decrease in size due to shorter samples more than offsets the increase in size of the embedding layer. While a shorter sample size does not affect the number of parameters in an LSTM network, it does increase the number of iterations in the recurrent layers, so it is not surprising to see the time of the LSTM change in the same way as the CNN. Comparing the two CNN variations, it also seems that calculating the squashed norm is slightly more computationally expensive than max-pooling. The differences in times are not significant enough to make a definitive statement, however. They could just as easily be the product of load fluctuations on the server. More thorough experiments would be needed to confirm this, but that lies outside of the scope of this thesis. Overall, CNNs, at least the way they are implemented in Pytorch, are the most efficient in terms of time, while LSTM networks are more efficient in terms of space. Capsule networks are very resource intensive in both areas, especially if samples are long.

Training Statistics

Looking at the training statistics of the networks can provide clues to the reasons behind their performance. Despite the different learning rates, the capsule network and both CNNs plateau at around 10 epochs on the 20 newsgroups dataset. With 3072 samples per epoch that corresponds to roughly 2.3 iterations through the training data. The LSTM network has a much slower start and takes a bit longer before the score becomes stable. This might be due to the relatively high length of the samples. Since only the output of the last time step (before padding begins) is used for classification, it probably takes some time to learn how to preserve relevant information all the way to the end. It is also the network that ends with the lowest training accuracy, meaning, that it would probably benefit the most from dropout or other regularization methods.

The loss progressions on this dataset for the LSTM and CNNs look similar, even if the former has a consistently higher average loss. This is due to the fact that the loss per batch is less stable for the LSTM and varies more widely even late in the training. The shape of the loss progression for the capsule network has the same shape as the other three, but converges to an unusually high value. Apparently, this does not seem to affect the end result, though, as the capsule network achieves the second highest score on this dataset. On the Twitter dataset it starts at a higher value than the other networks and never even reaches their starting point. There is probably a connection between this and the comparatively poor performance of the capsule network on this dataset, as the starting loss is usually fairly consistent if the loss function and dataset are the same. In addition, it stops improving much faster than both CNNs. Investigating this discrepancy would be a good starting point for improving the performance of the capsule network on this dataset. Interestingly, the LSTM network, which performs much better than the capsule network on this dataset also stops after 10 epochs, which is only about 2 iterations through the dataset. Similar to the 20 newsgroups dataset it has the lowest training accuracy of all



Training Accuracy

CNN:	99.74	CNN:	63.51
Squash CNN	98.21	Squash CNN	62.03
LSTM network:	97.79	LSTM network:	54.00
CapsNet:	99.41	CapsNet:	60.10

Figure 4.3: These graphs show the development of the F1 scores (top) and the loss (bottom) of the 20 newsgroups dataset (left) and the Twitter dataset (right). Differing numbers of samples per epoch were used to make the graphs more readable. Note that the networks trained with different learning rates, so their training speed is not directly comparable.

the networks when it stops. The CNNs both train for 30 epochs here which is equal to about 5.9 iterations.

Confusion Matrices

The confusion matrix of a model can yield insight into its strengths and weaknesses. In our results, all networks produce similar matrices and do not display any different patterns. Figure 4.4 shows the matrix of the best network for each dataset, which will serve as a representative for all the others. Additional confusion matrices for the capsule network can be found in the Appendix.

At a glance, there does not seem to be a strong correlation between the accuracy and average text length of each class on the 20 newsgroups dataset. The most frequent mistakes in this dataset are not surprising, considering how closely related the topics are. Samples of *talk.politics.misc* are often classified as *talk.politics.guns* and samples of *talk.religion.misc* are often classified as *soc.religion.christian*. These two misclassifications specifically are more common than the reverse due to the asymmetry in distribution. If there are more samples of *soc.religion.christian* than *talk.religion.misc* and a sample looks like it could belong to either, it is smarter to predict *soc.religion.christian*, since that has a higher chance of being correct. The dataset as a whole is mostly balanced, but the difference between these two (second most common, 5.3% of the dataset, versus least common, 3.3% of the dataset) is enough to have a noticeable effect. Content wise, these are the kind of mistakes one would expect to be most common in a dataset like this. The themes and language of these classes likely overlap heavily. This is true for the other most common mistake, confusing *alt.atheism* with *talk.religion.misc* and vice-versa, as well as less frequent mistakes, like confusion between *comp.sys.ibm.pc.hardware* and *sci.electronics*. Both directions of misclassification are equally likely for those two.

Similar patterns can be observed in the confusion matrix for the Twitter dataset. Since *face with heart eyes* (😍) and the *face with tears of joy* (😂) are the second and third most common emojis in the dataset, many less frequent emojis are often misclassified as one of these two. This is especially true if they have a similar meaning, like many of the other faces. Among all emojis, *beaming face with smiling eyes* (😄) is the hardest to predict correctly and *red heart* (❤️) is the easiest. *Christmas tree* (🎄) has the second highest accuracy, which is notable, as it is the third least common emoji and the one with the shortest average sample length. Unlike most of the other less common emojis, it has a very specific meaning, so it is probably used in a much more consistent context. As a result, it is comparably easy to predict. Also interesting are the values for *camera with flash* and *camera* (📷, no flash). There are frequent mistakes in both directions, but only one type of mistake is more common than correct predictions, at least among these four possibilities. This suggests that there actually is a small difference in how these emojis are used. In fact, many emojis are apparently easier to differentiate than expected. *Two hearts* (💕), for example, is rarely misclassified as *red heart* (❤️), despite the latter being about four times as common.

Even if the scores for the Twitter dataset are not very impressive, looking at the confusion matrix provides interesting insights into how the emojis are used in the dataset.

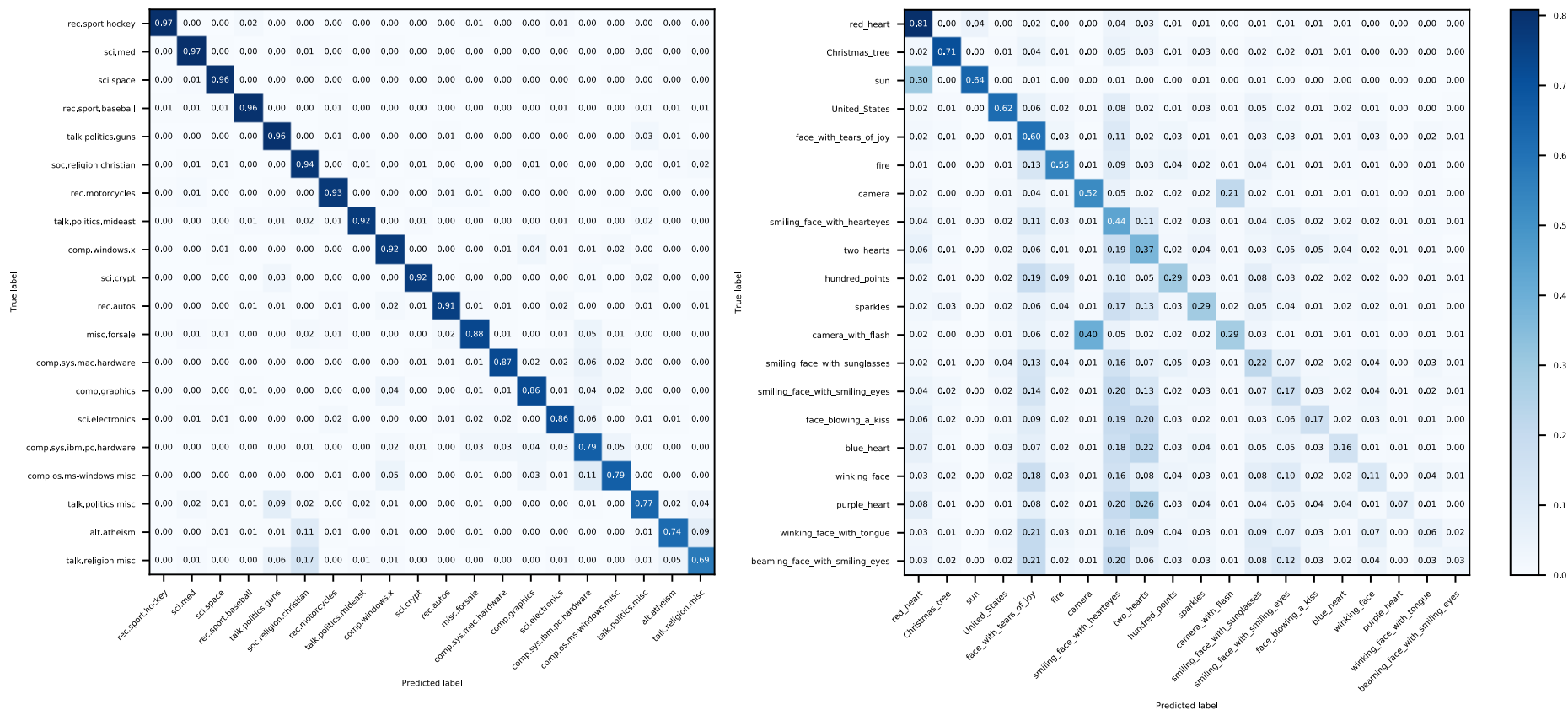


Figure 4.4: These are the confusion matrices for both dataset. They depict the normalized accuracy for the best network on each, the alternative CNN for 20 newsgroups (top) and the standard CNN for Twitter (bottom). As an example, the 0.17 in the entry for *talk.religion.misc* - *soc.religion.christian* (row 20, column 6) means that 17% of all *talk.religion.misc* samples were classified as *soc.religion.christian*. Some rows do not add up to 1.0 due to rounding.

5 Further Insights











Using capsule networks for text classification and comparing them to other networks on this task reveals some of their strengths and weaknesses. This chapter discusses their potential for text processing, based on the results of the previous chapters.

5.1 Reconstructions

The network architecture introduced in Sabour et al. (2017) includes a reconstruction module, placed after the class capsule layer. It receives the output vector of a capsule as input and it produces a tensor with the same dimensions and value range as the input sample. As the name implies, the point of this module is to recreate the original input based on the output of a capsule. An additional loss is calculated based on the difference between the reconstruction and the original. This essentially turns the network into an auto-encoder. Since the primary purpose of the network remains classification, the additional loss is multiplied with a small factor λ before it is added to the prediction loss, so that it influences the training, but does not dominate it. This procedure should encourage the capsules to learn generalized, meaningful output parameters. As such, it works as regularization method. The most straightforward implementation of such a module is as a series of linear layers. The last of those layers must have an output size equal to the number of entries in the input tensor and use an activation function with the same value range as the samples. This one-dimensional output tensor is then reshaped to resemble the original input, before it is compared to it.

There are two ways to build a reconstruction module directly into a capsule network. The first is to have one module per capsule in the final layer (Hinton et al., 2011). In that case, all capsules produce a reconstruction, but only the one made by the capsule corresponding to the true label is used to calculate the loss. This is important, as it allows each module to learn the commonalities between different instances of the entity represented by its capsule. That way, the entries in the capsule output vector represent parameters, which describe how a particular instance differs from the base representation. A more efficient way of using reconstruction modules is to use a single module, which is connected to the output of all capsules of the last layer (Sabour et al., 2017). When the output of that layer is passed to this module, each capsules output is set to zero, except for the one representing the true label of the sample. That way, the information of what exactly is being reconstructed is preserved, even though there is only one reconstruction module for all capsules.

One of the benefits of this regularization method is that it can provide insight into what the network has learned. Using it on digit recognition results in very concrete instantiation parameters for the class capsules, whose effect on the reconstructions is easy to visualize and understand (see Figure 5.1). Previous applications of capsule networks on text either seem to not use reconstructions (Yang et al., 2018) or do not mention how they handle

(l, p, r)	(2, 2, 2)	(5, 5, 5)	(8, 8, 8)	(9, 9, 9)	(5, 3, 5)
Input					
Output					


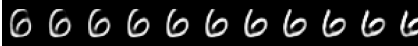
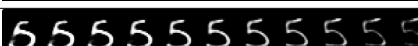
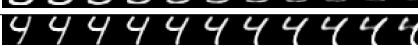

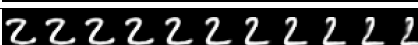
Scale and thickness	
Localized part	
Stroke thickness	
Localized skew	
Width and translation	
Localized part	

Figure 5.1: Examples of reconstructions from the MNIST dataset by Sabour et al. (2017). The bottom table shows the results of manipulating a single entry in the capsule output vector.

these difficulties (Wang et al., 2018). To our knowledge, no work has been done to examine, whether reconstructions can produce sensible results on text, like they can on digit recognition.

All the capsule networks evaluated in Chapter 4 have no reconstruction module. The reasons for this are twofold. Using a regularization method in one network but not the others would weaken the comparison between them. Other regularization methods could be used on the CNN and LSTM network, but this too would have made a direct comparison less meaningful, as it would not be clear, whether the differences in performance are due to the networks themselves or the regularization method. The other reason is that using auto-encoders on text is more difficult than it is on images. Unlike images, text needs to be preprocessed before it can be fed into a neural network. Raw text is not represented by a tensor of floating point values, which is the format most networks expect as input. Thus, there is no ground truth to compare the reconstruction against and no direct way to calculate a loss. The closest alternative is the embedded input, since it is a sequence of vectors. This also presents a problem however, since the embedding process is usually implemented as a trainable embedding layer in the network. Because of this, the representation, the target of the reconstruction, changes over the course of the training. The embedding layer can be made static to avoid this, but that sacrifices the improvement to classification results that it brings. The different lengths of samples present another challenge. To allow for batched processing, samples are all padded to the same length with a padding vector. Without additional measures, this causes the reconstruction module to learn that producing a reconstruction made entirely of these padding vectors is a good way to minimize the average difference to the original.

Despite these challenges, reconstructions are a unique quality of capsule networks, so their use in text should still be explored. To judge, whether reconstructions have any potential in this domain, the following steps are taken to mitigate the problems described above:

- To solve the problem of the moving target, a capsule network without reconstruction module is trained first. Then a new capsule network with a reconstruction module is

created and its embedding layer is initialized with the word vectors extracted from the previous network. This embedding layer is made static, so that the word vectors do not change over the course of the training. This method should provide the benefits of a trainable embedding layer without drawbacks for the reconstructions.

- Before loading the trained word vectors into the second network, they are normalized again and the padding vector is set to zero. Also, before calculating the difference loss between the reconstruction and the original, every word vector in the reconstruction is normalized. This prevents the network from filling the reconstruction with padding vectors.
- The squashing function, which is used on all capsule output vectors as an activation function, often leads to very small output values. This is especially true for a newly initialized network. Because of this, the initial linear layer of the reconstruction module should not have a bias, since it will overshadow the actual input. As a result, the reconstruction would always be the same, regardless of input.
- Since all word vectors in the target are normalized, all entries are between -1 and 1 . Therefore, \tanh is used as the activation function of the final layer in the reconstruction module.

With these adjustments, reconstructions make sense on a mathematical level. That is enough to test whether they improve the networks performance or not, but in order to judge whether the results are sensible at all, the reconstructed tensor first has to be turned back into a representation that is readable by humans. To do this, the two-dimensional reconstruction tensor is split up into a sequence of word vectors. Then the embedding is searched for the nearest vector and word represented by that vector is taken. Concatenating all those words with spaces results in a readable output (see Figure 5.2). Another way to visualize the result is to perform a principle component analysis (PCA) on both the word vectors of the original sample and those of the reconstruction, so that they can be graphed in a 2D scatter plot. In order to better compare the resulting positions, the PCA should be performed on all words in the embedding at once, so that all graphed vectors have the same frame of reference and different samples can be compared. The spacial distribution of word vectors of a good reconstruction should be similar to that of the original. Figure 5.3 shows these scatter plots.

rec.sport.baseball

"article writes williams writes writes thought thought hit baseball
hit year baseball baseball baseball players hit players hit baseball
baseball hit sure hit hit players sure sure hit [...]"

soc.religion.christian

"article writes writes writes think think reason reason god god reason
god god reason reason reason reason god god reason reason reason
reason reason reason think think think reason [...]"

Figure 5.2: Two examples for reconstructions from the 20 newsgroups dataset.

Despite all of these steps, the results for reconstructions on the 20 newsgroups dataset still barely resemble the original text. This is in stark contrast to the reconstructions on the MNIST dataset, which not only show clearly recognizable digits, but also mimic some

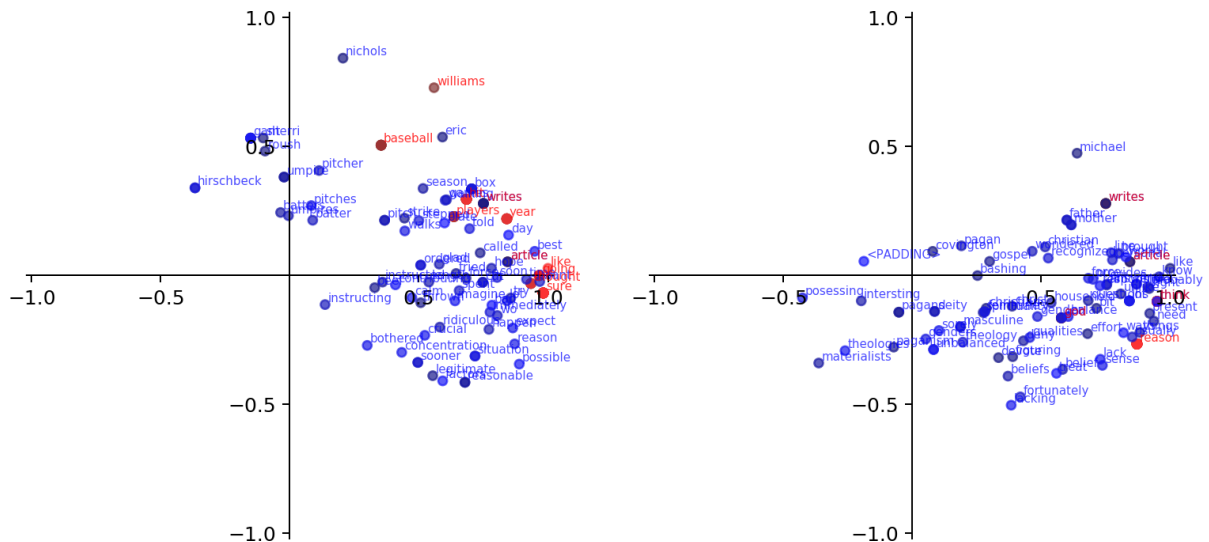


Figure 5.3: The scatter plots for the reconstructions from Figure 5.2. The one the left belongs to the sample from *rec.sport.baseball*, the one on the right belongs to the sample from *soc.religion.christian*. The words of the original sample are in blue, the words of the reconstructions in red.

of the specific properties of the input. Still, the strings produced by the reconstruction module here seem like they belong in the correct class and they display some structural elements. Many samples begin with quotes, so the reconstructions start with "article" and "writes", which are the only words left in the introduction of a quote after filtering out stop words and non-alphanumeric tokens. Overall, they appear more like a collection of prominent words of the class, rather than the reconstruction of a specific sample. Using reconstructions did not increase the classification score either. To get a reasonable result, a class capsule dimension of 32 had to be used. A network with optimized hyperparameters and a class capsule dimension of 16 produced worse results than the ones shown here. Several things could explain this sub-par performance. In the MNIST dataset, samples of the same class are much more similar to each other than in the 20 newsgroups dataset. They also contain essentially no noise. Since the loss is computed with a simple mean square error (MSE), reconstructing noise becomes part of the goal. This blurs the purpose of the instantiation parameters as they must account for this. A possible solution for this is to have an additional orphan capsule representing noise. Sabour et al. (2017) do this to improve classification results on the CIFAR-10 image classification dataset¹, in which the background of the images is considered noise. Sadly, they do not mention how this affects the reconstruction. In our experiments, using an orphan capsule does not improve results for the 20 newsgroups dataset, regardless of whether its input is used by the reconstruction module or not. Trying reconstructions on the unfiltered 20 newsgroup dataset shows how much of a problem noise is. In that case, the reconstructions contain nothing but the most common stop words. Also, a series of linear layers and MSE loss might just work much better on images than it does on text.

These results do not mean that reconstructions as a whole are useless on text. One possible avenue of improving reconstructions is to reduce their size. After all of the preprocessing is done, only a fraction of the words in the pretrained embeddings actually

¹<https://www.cs.toronto.edu/~kriz/cifar.html>

appear in the dataset. Theoretically, this means less dimensions are needed to properly represent their differences. Using PCA to reduce the size of the word embeddings reduces the size of the input, which also reduces the size of the reconstruction. Shrinking the output, while leaving the parameter count of the network the same should increase the fidelity. Unfortunately, reducing the word vectors with PCA also reduces the classification performance, so this method is not worth the trade-off.

In summation, reconstructions in their current form do not help with the text classification tasks presented in this thesis, but they deserve further attention in future work.

5.2 Discussion

The results in Sections 4.3 and 5.1 show some success for using capsule networks and their concepts in text classification. Using the squashed norm instead of max-pooling in a CNN network significantly improved the performance of a simple CNN on the 20 newsgroups dataset, getting it relatively close to the state of the art. Since this is a simple change with little effect on the networks complexity, it should require little effort to test whether this improvement translates to other tasks as well. The fact that it serves the exact same function as a pooling layer, taking a number of values with no regard to their placement within the window and calculating a new, single value based on them, makes this a real alternative. It would be interesting to see if this method is still beneficial, if the window of the pooling layer does not encompass the entire feature map. Considering that it brought no benefit over the standard method on the Twitter dataset, its applications might be limited. Since the capsule network itself also performs better than the CNN on the 20 newsgroups dataset and worse on the Twitter dataset, it seems likely that this reflects the strengths and weaknesses of capsules. Still, they do clearly have potential for text classification tasks, as the results on 20 newsgroups show. It would be interesting to see how deeper capsule networks fare against deeper CNNs. A big part the argument why capsule networks are theoretically superior is that they use routing instead of pooling, which does not lose any information. Therefore, the benefits or drawbacks should become more apparent if the network has more of these layers. Based on our results, deep capsule networks would require very powerful hardware, so these kinds of comparisons are currently not very practical. The comparatively low score of the the capsule network on the Twitter dataset might be explained by its large number of parameters, which makes it more prone to overfitting. The fact that the optimal CNN uses as little filters as possible seems to support this hypothesis. Looking at the results through this lens, it seems like the chosen hyperparameter setting are not actually optimal for the Twitter dataset. A more thorough grid search might have lead to better results. Even if that is the case, it likely would still not achieve the highest score on that dataset, since that requires an increase of more than four points. The relatively equal performance of all the other networks also suggests that more sophisticated models and regularization methods are needed to achieve better scores here. Since the ranking of the networks differs so much between the two datasets, it is safe to say none of the examined network are strictly better than the others. Though one should acknowledge the consistently good performance of simple CNNs. Since they seem to produce decent results on almost any task and train quickly thanks to good optimization, they should make for good baselines in many applications.

In addition to the networks themselves, the preprocessing on the Twitter dataset has much room for improvement. The high number of LFUTs could be reduced with better tokenization. Incorrect or unusual formatting often results in strings that are not split correctly. For example, if punctuation is followed by a letter or digit, spaCy will not split it into its own token. That means, if two sentences are not separated properly, the last word of the first sentence and the first word of the second sentence end up as a single token, together with the punctuation. Since it is rather unlikely that this mistake happens again with the exact same words, tokens like these are mostly marked as LFUTs as they occur only once in the corpus and can not be found in the pretrained embedding. The Twitter dataset contains many mistakes of that or a similar nature, which is one of the reasons why the number of LFUTs ends up being so high. A simple rule that always splits punctuation into its own token would lead to other mistakes though, so more sophisticated tokenization would be necessary to solve this problem. Similarly, many hashtags are marked as LFUTs, despite containing information. Even hashtags that occur only once in the corpus are often made up of normal words. Deriving a word vector based on the components would provide more information than using a random vector or mapping the hashtag to the generic LFUT like we currently do. Even just using a generic *<unique hashtag>* token would probably be better than the current approach. Deriving an additional word vector from pretrained character embeddings is another option and has been shown to work well on Twitter data (dos Santos and Gatti, 2014).

Aside from preprocessing, it is also possible that the performance of the capsule networks themselves is not as good as it could be. Additional tests show that using three routing iterations instead of one only improves performance if the capsule dimension of the routing layer is higher than that of the ingoing capsule layer, like the original paper suggests. With only one routing iteration, the layer is equivalent to a normal, fully connected, linear layer. It is possible that the optimal hyperparameter settings for the capsule network found with the grid search would be even better, if the class capsule dimension was changed from 16 to 32. Even if that was the case, the grid search would not have found this setting though, because the resulting network is too large to fit on the GPU used for training. Still, it is possible that a usable, more optimal setting with a higher class capsule dimension and lower prime capsule number exists, but was not found by the grid search. Since no settings between 16 and 32 were examined for these parameters, the optimum might actually lie somewhere in between. The epoch length and loss delta can also have an effect on which hyperparameters end up being declared as optimal. Since the main criterion for stopping the training during the search is the change in loss, networks that overfit more easily are graded lower, even if they would achieve better validation results earlier during training.

Whether reconstructions can improve text classification results of a capsule network is something that needs to be investigated more. The results presented here and in the literature certainly show that good performance can be achieved without them, but that does not mean that they have no purpose in text processing. Though, to make good use of them, other reconstruction methods need to be explored, as the naive approach does not yield good results, even with some adjustments. It is also worth noting, that a reconstruction module further increases the size of the already large networks, so using them on long texts might not even be practical with current hardware. Since RNNs can produce sequential output of any length independent of their own size, they might be the best alternative here.

6 Conclusion

6.1 Conclusion

Sentiment analysis as a specialized form of text classification is becoming increasingly useful with the wealth of user generated text on the web. It can be used to gain insight on opinions at a large scale and could help manage toxicity in online communities. Since many platforms rely on active, participating users, this is important for both moral and business oriented reasons. This domain presents many challenges for text processing however, ranging from spelling mistakes and unusual formatting to the incomplete nature of the raw text in posts which contain images and links. The established preprocessing method of using common tokenization methods and pretrained word embeddings alone has many shortcomings in that regard. Perhaps that is the reason why even simple neural networks can still come close to state-of-the-art results on some of these tasks. Especially in the case of Twitter data, this suggests that looking into the problems related to preprocessing would be just as, if not more beneficial than improving the models themselves. Even with all the challenges and low classification scores, training a model on this data can still be useful. The confusion matrices can show the presence or absence of correlations, which provide all kinds of insights into the dataset.

Capsule networks were introduced as a proposed improvement to CNNs, so it is encouraging to see that they achieve better results on one of the two datasets used for evaluation. In addition, using one of its concepts, the squashed norm, in CNNs instead of max-pooling improved results considerably. Since both of these improvements over the standard CNN are limited to only one of the datasets, we can conclude, that capsule networks have a place in text classification, but are not a strictly superior version of CNNs. They also have their drawbacks, as they are much larger in size and considerably slower. Since the size scales heavily with both hyperparameters and sample length, capsule networks might not be suitable for some tasks for technical reasons alone. This also has to be taken into account during hyperparameter optimization, to avoid situations, where potentially good settings are ignored, due to the resulting network not fitting into memory. This is true even when no reconstruction module is being used. Using reconstructions on text is only really possible with many adjustments and even then, they do not improve classification performance like they do on image classification. This conclusion is only based on a single, simple way of using this concept however, so it should not necessarily be discounted entirely.

In conclusion, capsule networks are a useful alternative to traditional network types in text processing. With a high computational complexity and a reconstruction module that currently provides no benefit on the tasks presented here, it seems that they still have untapped potential.

6.2 Future Work

It has been shown that capsule networks can achieve good results on text related tasks, but that does not mean that there are no further improvements to be made. As discussed, reconstructions are one area, where more work could be done. There are many different approaches that could lead to new insights there. For example, some kind of generative language model would probably result in more sensible output. Using RNNs for this would also solve the problem of the reconstruction module scaling with input size, which is a considerable hurdle in some applications. If that is not a concern, a capsule based reconstruction module might actually be more appropriate, since capsules output vectors and the goal is to produce a series of word vectors. Using a loss function more suited for vectors would also be worth exploring. Producing reconstructions that seem reasonable to a human is technically only a nice side effect though, the primary goal is still to improve classification performance by using them as a regularization method. These two goals are probably correlated, so it is definitely worth looking into. Considering the generative nature of capsules, they might also be useful in generative adversarial networks (Goodfellow et al., 2014). Having a discriminative model rate the quality of a reconstruction instead of using a numeric loss seems like an interesting alternative. All of these are potential improvements for the architecture used in this thesis, but that is not the only way to implement the concept of capsules. It is entirely possible to construct a primary capsule layer from something other than a convolution layer. This would potentially affect some assumptions like the place coding and receptive fields, which might have to be taken into account in the routing layer. The routing algorithm itself, including the concept of having the vector length represent a probability, is not definitive either. Other ways of modeling the agreement between child entities on their predictions of parent entities could be explored. In summary, the concept of capsule networks is promising and flexible enough to provide a wide array of possibilities for its use.

Considering the properties of short informal online texts like tweets and the evaluation results, there is still work to be done when it comes to handling their unique challenges. Modeling spelling mistakes, unusual formatting and other stylistic choices separate from the underlying word would be much closer to the way humans read these kinds of texts. Done correctly, it would not only provide additional information to a classifier, but also improve the information that is already there by better separating it from noise. This kind of process needs a very different pipeline though, so a novel approach to preprocessing would need to be developed.

Finally, the effects of using the squashed norm in an otherwise unaltered CNN need to be explored more. If it turns out to be a real alternative to other pooling methods, it could potentially be useful in many network architectures and applications.

Bibliography

- Barbieri, Francesco, Miguel Ballesteros, and Horacio Saggion (2017). *Are Emojis Predictable?* In: *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics*. Valencia, Spain, pp. 105–111.
- Barbieri, Francesco, Jose Camacho-Collados, Francesco Ronzano, Luis Espinosa Anke, Miguel Ballesteros, Valerio Basile, Viviana Patti, and Horacio Saggion (2018). *SemEval 2018 Task 2: Multilingual Emoji Prediction*. In: *Proceedings of The 12th International Workshop on Semantic Evaluation*. New Orleans, USA, pp. 24–33.
- Beißwenger, Michael, Sabine Bartsch, Stefan Evert, and Kay-Michael Würzner (2016). *EmpiriST 2015: A shared task on the automatic linguistic annotation of computer-mediated communication and web corpora*. In: *Proceedings of the 10th Web as Corpus Workshop*. Berlin, Germany, pp. 44–56.
- dos Santos, Cícero and Maíra Gatti (2014). *Deep convolutional neural networks for sentiment analysis of short texts*. In: *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers*. Dublin, Ireland, pp. 69–78.
- Duggan, Maeve (2017). *Online harassment 2017*. Pew Research Center. Technical Report.
- Dumoulin, Vincent and Francesco Visin (2016). *A guide to convolution arithmetic for deep learning*. In: *ArXiv e-prints*. eprint: 1603.07285.
- Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. 1 Rogers Street, Cambridge, MA: MIT Press.
- Goodfellow, Ian, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio (2014). *Generative Adversarial Nets*. In: *Advances in Neural Information Processing Systems 27*. Curran Associates, Inc., pp. 2672–2680.
- Hinton, Geoffrey E., Alex Krizhevsky, and Sida D. Wang (2011). *Transforming auto-encoders*. In: *International Conference on Artificial Neural Networks*. Espoo, Finland, pp. 44–51.
- Hochreiter, Sepp and Jürgen Schmidhuber (1997). *Long short-term memory*. In: *Neural computation* 9.8, pp. 1735–1780.
- Joachims, Thorsten (1998). *Text categorization with Support Vector Machines: Learning with many relevant features*. In: *Machine Learning: ECML-98*. Ed. by Claire Nédellec and Céline Rouveirol. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 137–142.
- Kim, Yoon (2014). *Convolutional Neural Networks for Sentence Classification*. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, 2014, Doha, Qatar*, pp. 1746–1751.
- Kingma, Diederik and Jimmy Ba (Dec. 2014). *Adam: A Method for Stochastic Optimization*. In: *International Conference on Learning Representations*. Banff, Canada.
- LeCun, Yann, Bernhard Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne Hubbard, and Lawrence D. Jackel (1989). *Backpropagation Applied to Hand-written Zip Code Recognition*. In: *Neural Computation* 1.4, pp. 541–551.
- Mikolov, Tomas, Ilya Sutskever, Kai Chen, Greg S. Corrado, and Jeff Dean (2013). *Distributed representations of words and phrases and their compositionality*. In: *Advances in Neural Information Processing Systems*. Lake Tahoe, USA, pp. 3111–3119.

- Pang, Bo, Lillian Lee, et al. (2008). *Opinion mining and sentiment analysis*. In: *Foundations and Trends in Information Retrieval* 2.1–2, pp. 1–135.
- Pennington, Jeffrey, Richard Socher, and Christopher D. Manning (2014). *GloVe: Global Vectors for Word Representation*. In: *Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar, pp. 1532–1543.
- Sabour, Sara, Nicholas Frosst, and Geoffrey E. Hinton (2017). *Dynamic routing between capsules*. In: *Advances in Neural Information Processing Systems*. Long Beach, USA, pp. 3856–3866.
- Sebastiani, Fabrizio (2002). *Machine Learning in Automated Text Categorization*. In: *ACM Comput. Surv.* 34.1, pp. 1–47.
- Shu, Lei, Hu Xu, and Bing Liu (2017). *DOC: Deep Open Classification of Text Documents*. In: *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. Copenhagen, Denmark, pp. 2911–2916.
- Singh, Tajinder and Madhu Kumari (2016). *Role of text pre-processing in twitter sentiment analysis*. In: *Procedia Computer Science* 89.1, pp. 549–554.
- Srivastava, Nitish, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov (2014). *Dropout: a simple way to prevent neural networks from overfitting*. In: *The Journal of Machine Learning Research* 15.1, pp. 1929–1958.
- Wang, Yequan, Aixin Sun, Jialong Han, Ying Liu, and Xiaoyan Zhu (2018). *Sentiment analysis by capsules*. In: *Proceedings of the 2018 World Wide Web Conference on World Wide Web*. Lyon, France, pp. 1165–1174.
- Wint, Zar Z., Théo Ducros, and Masayoshi Aritsugi (2017). *Spell corrector to social media datasets in message filtering systems*. In: *2017 Twelfth International Conference on Digital Information Management (ICDIM)*. Fukuoka, Japan, pp. 209–215.
- Wulczyn, Ellery, Nithum Thain, and Lucas Dixon (2017). *Ex machina: Personal attacks seen at scale*. In: *Proceedings of the 26th International Conference on World Wide Web*. Perth, Australia, pp. 1391–1399.
- Yang, Min, Wei Zhao, Jianbo Ye, Zeyang Lei, Zhou Zhao, and Soufei Zhang (2018). *Investigating Capsule Networks with Dynamic Routing for Text Classification*. In: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Brussels, Belgium, pp. 3110–3119.

Appendix

List of Figures

2.1	Text representation	8
2.2	General CNN architecture	11
2.3	CNN architecture by Kim (2014)	11
2.4	RNN illustration	12
2.5	Capsule network architecture	14
3.1	CNN & LSTM Architectures	19
3.2	CapsNet Architecture	20
4.1	20 newsgroups dataset sample	25
4.2	Tweet example	27
4.3	Training statistics	33
4.4	Confusion matrices	35
5.1	MNIST Reconstructions	37
5.2	Reconstruction Examples	38
5.3	Reconstruction Scatter Plot	39

List of Tables

4.1	20 newsgroups statistics	26
4.2	English 2018 Semeval Task 2 dataset statistics	28
4.3	Dataset distributions	28
4.4	Grid search results	29
4.5	Evaluation results	31

List of Acronyms

CNN	Convolutional neural network
RNN	Recurrent neural network
LSTM	Long short-term memory
LFUT	Low frequency unknown token
NaN	Not a Number
SVM	Support vector machine
PCA	Principle component analysis
MSE	Mean square error

Source Code

For the sake of clarity, the code snippets presented here are altered slightly from the actual source code. These alterations include changing variable names to be more in line with the terminology used in this document and removing code which has no effect on the output of the network.

Reconstruction Module

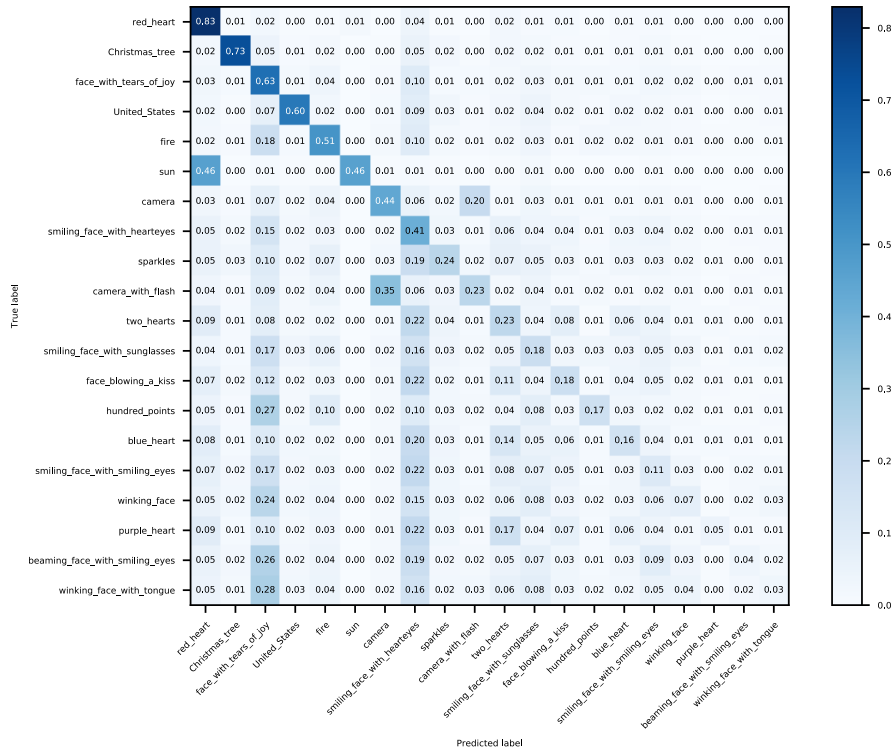
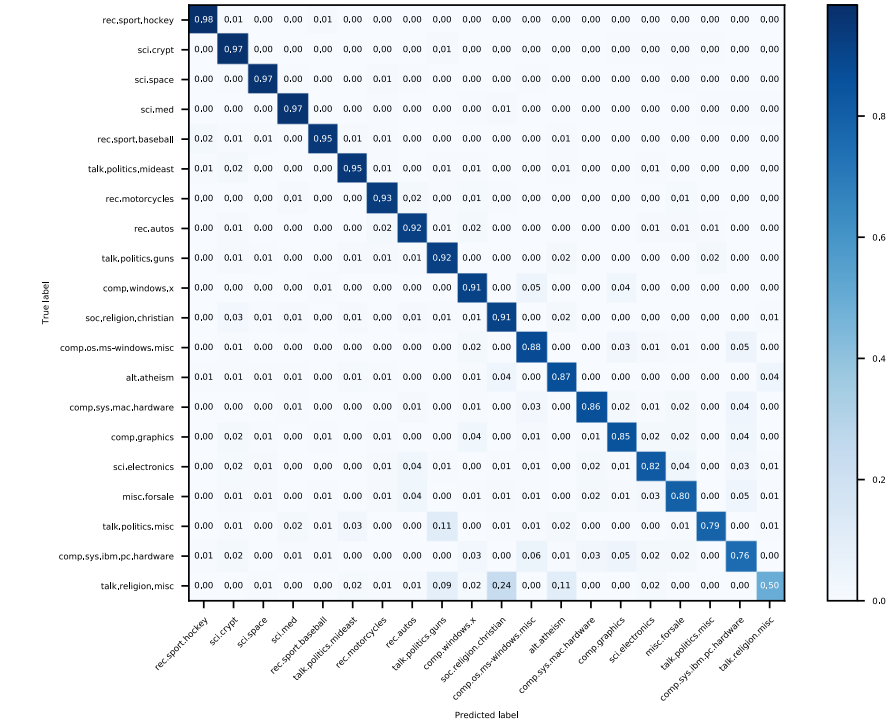
```
1 class Reconstructor(torch.nn.Module):
2     def __init__(self, class_caps_num, class_caps_dim, output_shape, orphan_caps=False):
3         super(Reconstructor, self).__init__()
4         numel = output_shape[0] * output_shape[1]
5         self.output_shape = [1] + list(output_shape)
6         size_limit = 4096
7         inter_size = min(2 * numel, size_limit)
8
9         self.lin1 = nn.Linear(class_caps_num * class_caps_dim, inter_size, bias=False)
10        self.lin2 = nn.Linear(inter_size, numel)
11
12        self.orphan_switch = torch.zeros(class_caps_num, class_caps_num)
13        if orphan_caps: # If there is a orphan capsule and it's output should be used
14            self.orphan_switch[class_caps_num-1, class_caps_num-1] = 1
15
16        def forward(self, x, y):
17            self.output_shape[0] = x.size(0) #replacing the placeholder 1
18            if x.is_cuda:
19                self.orphan_switch = self.orphan_switch.cuda()
20            y = y + self.orphan_switch
21            x = torch.matmul(y, x) # y is the label with a 2d one-hot encoding...
22                # ...meaning it's a matrix where y[i, j] = 1 only if i=j=label
23            x = x.view(x.size(0), -1)
24            x = torch.tanh(self.lin1(x))
25            x = torch.tanh(self.lin2(x))
26            x = x.view(self.output_shape)
27            n = x.norm(dim=-1, keepdim=True)
28            x = x/n
29            return x
```

Capsule Network

```
1 class CapsNet(text.TextNet):
2     def __init__(self, num_classes, max_length, embedding_dim, window, stride,
3         prime_num, prime_dim, class_dim, orphan_caps=False, recon_type="None"):
4         super(CapsNet, self).__init__(num_classes, (max_length, embedding_dim))
5         self.type = "caps"
6         self.num_classes = num_classes
7         self.window = window
8         self.stride = stride
9         self.prime_caps_num = prime_num
10        self.prime_caps_dim = prime_dim
11        self.filter_num = self.prime_caps_num * self.prime_caps_dim
12        self.class_caps_dim = class_dim
13        self.orphan_caps = orphan_caps
14
15        self.caps_conv = torch.nn.Conv2d(
16            in_channels = 1,
17            out_channels = self.filter_num,
18            kernel_size = self.window,
19            stride = self.stride
20        )
21        conv_caps_output_size = conv_output_size(self.input_size, self.window,
22            self.stride)
23
24        self.class_caps = caps.CapsuleLayerDense(
25            in_caps_num = conv_caps_output_size[0] * conv_caps_output_size[1] *
26                self.prime_caps_num,
27            in_caps_dim = self.prime_caps_dim,
28            num_capsules = num_classes,
29            output_dim = self.class_caps_dim
30        )
31
32        if recon_type != "None":
33            self.reconstructor = caps.Reconstructor(num_classes, class_caps_dim,
34                self.input_size, orphan_caps)
35
36    def forward(self, x, length=None, y=None):
37        x = x.unsqueeze(1) #Unsqueeze for in_channel = 1
38        x = self.caps_conv(x)
39        x = x.view(x.size(0), -1, self.prime_caps_dim) #formatting as capsules
40        x = caps.squash(x)
41
42        x = self.class_caps(x)
43        x = caps.squash(x)
44        classes = x.norm(dim=-1) #vectors -> length
45        classes = F.normalize(classes, dim=-1, eps=sys.float_info.epsilon)
46        classes = torch.softmax(classes, dim=-1)
47        x = self.reconstructor(x, y)
48
49    return classes, x
```


Additional Statistics

Confusion matrices of the capsule network



Erklärung

Ich versichere, dass ich die Arbeit selbstständig verfasst und keine anderen, als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internetquellen – benutzt habe, die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ich bin mit der Einstellung der Master-Arbeit in den Bestand der Bibliothek des Fachbereichs Informatik einverstanden.

Hamburg, den