Karlsruhe Institute of Technology

Term Extraction as Sequence Labeling Task using Recurrent Neural Networks

Master Thesis
of

Maren Kucza

at the Department of Informatics
Institute for Anthropomatics and Robotics

First Reviewer:	Prof. Dr. A. Waibel
Second Reviewer:	Prof. Dr. T. Asfour
Advisors:	Dr. Sebastian Stüker

Time Period: 1st December 2017 – 31st May 2018

Abstract

Terminology extraction is mostly performed in two steps. In the first step, the text is filtered in order to identify word groups, which fit predefined syntactic patterns. In the second step, these identified words are ranked using special metrics or machine learning techniques.

However, it is possible to view the input sentences as entities and extract terminology based on observed features in them. This view has the advantage that it does not rely on additional systems, e.g. a part-of-speech tagger. Additionally, no linguistic knowledge of the input language is required.

A recurrent neural network was implemented for the classification of two text collections. Such a collection contains only texts of the same field. Precision, recall, loss and run time were collected for each test run in order to compare the results. The influences of hyperparameters were tested and different regularization techniques were applied.

The results were very good and stable on the larger one of the two text collections. On the smaller one, the scores sank. The application of a trained network to texts of a different field also resulted in worse scores.

The results have shown that this implementation is not a generally applicable solution and its performance depends on the size and label balance of the input text. However, it has been shown that the application to texts of the same field has potential and can provide a new point of view for existing systems.

Zusammenfassung

Terminologie Extraktion wird meist in zwei Stufen durchgeführt. Der erste Teil umfasst Filterung des Textes nach Wortgruppen, die in manuell definierte syntaktische Gruppen fallen. Diese ausgewählten Wort Untergruppen werden im zweiten Schritt mithilfe von speziellen Metriken oder maschinellen Lernalgorithmen nach ihrer Signifikanz sortiert.

Es ist jedoch auch möglich die eingegebenen Sätze als Ganzes zu betrachten und anhand von darin erkennbaren Merkmalen Terminologie zu bestimmen. Dies birgt den Vorteil unabhängig von zusätzlichen Systemen wie beispielsweise eines Part-of-Speech Taggers zu sein. Zudem ist linguistisches Wissen über die Eingangssprache nicht erforderlich.

Für die Klassifizierung wurde ein rekurrentes neuronales Netz implementiert, welches auf zwei Textsammlungen angewandt wurde. Solch eine Sammlung enhält nur Texte eines Fachgebiets. Zum Vergleich der Ergebnisse wurde für jeden Testlauf precision, recall, loss und Laufzeit gesammelt. Die Einflüsse verschiedener Parameter des neuronalen Netzes wurden ausgetestet und verschiedene Regularisierungstechniken angewandt.

Auf dem größeren der Textsammlungen waren die Ergebnisse sehr gut und stabil. Auf dem kleineren sanken die Werte. Bei der Anwendung eines trainierten neuronalen Netzes auf einen Text der anderen Fachrichtung ergaben sich ebenfalls schlechtere Werte.

Die Ergebnisse zeigen, dass diese Implementation keine allgemein anwendbare Lösung ist und dessen Performanz von der Größe und Ausgewogenheit des Eingabetextes abhängt. Jedoch zeigt sich auch, dass die Anwendung auf Texte einer Fachrichtung Potenzial hat und bisherigen Systemen einen neuen Blickwinkel bieten kann.

Contents

List of Figures

List of Tables

1. Introduction

Terminology extraction (sometimes also called term(inology) recognition) is the task of automatically identifying all domain-specific terminology from a given corpus. In this, termimonlogy extraction is a sub-task of information retrieval, which can aid various other fields, such as machine translation, ontology creation, knowledge management or document indexing. Contrary to *keyword extraction*, which aims at determining only the most representative words for a given input document, the number of identified terminology is not limited to a few words. Instead it is intended to extract terminology in its entirety. Thereby, the terms extracted can be single words or can be sequences of several words.

Most recent terminology extraction systems in literature use a two step approach. First candidates are extracted by either noun-phrase part-of-speech (PoS) patterns or by simply taking all possible ngrams, which do not contain stop words, into account. Stop words are frequent words, which hold no deep semantic content, such as conjunctions, articles or prepositions. These candidates are then classified by scoring them using various techniques, among them machine learning.

An approach using machine learning is proposed by [YuGZ17]. It starts by tokenizing and splitting the corpus into 1-5 grams. To reduce the number of ngrams, they are filtered by stop words. For each of these ngrams ten common term extraction features such as total term frequency, c-value, weirdness, etc. are selected. The selected features are classified using either a Random Forest, a Linear Support Vector Machine, a Multinomial Naive Bayes classifier, Logistic Regression or and SGD classifier. While there is no algorithm, which performs best on all test sets, Random Forest achieves the highest scores most often.

Another machine learning approach is given in [dSCPR13]. Before selecting candidate terms, the system standardizes word variations from the text and annotates PoS tags to it. Numbers, punctuation, single characters, stop words and conjugations of 'to be' are removed. The remaining words are considered possible terms. Features from statistical and linguistic knowledge and contrastive corpora are calculated for each unigram. The most representative features for term classification are selected by an algorithm based on either correlation or consistency and used with JRip, Naive Bayes, J48 or SMO from WEKA, a java based text classifier. Though none of the algorithms performs best on all of the test sets, the approach achieved state-of-the-art scores on unigram extraction in Brazilian Portuguese.

[WaLM16] proposes a method enabling co-training using neural networks. Before candidate terms are selected, plurals of nouns are removed and all words are converted to lowercase. An identifier based on a noun phrase PoS pattern and an identifier based on ngram chunking and stop word delimiters select possible terms. Co-training aims to build a stable classifier with only limited annotated data and a majority of unannotated data. It requires two different views on the data, which is achieved by putting the candidate terms in an LSTM and a CNN. After each iteration the most confident term predictions are added to the annotated data.

However, terminology extraction can also be seen as a sequence labeling problem. Whether a word qualifies as a term depends on its syntactic and semantic properties. Especially syntactic properties depend on the features of the surrounding words. Because of this, sequence classification techniques can be applied to entire sentences and not to filtered candidates. The classification result is a sequence of labels, which is interpreted parallel to the input sentence.

In the following, an approach to extract terminology as sequence labeling task is presented. The major difference to other systems is its independence from prior filtering and PoS tags. Furthermore, sequence-to-sequence classification is performed. For this, the basics of neural networks and especially recurrent neural networks, the structure and necessary preparation of data, as well as the complete model will be introduced in sections 3 and 4. The metrics for comparison and results of test runs with different parameters and regularization techniques are presented in section 5. The remaining misclassifications are analyzed and a possible approach to reduce such is given in section 6. In the last section, the results are summarized and possible further developments are mentioned.

2. General Approach

2.1 Common Terminology Extraction Systems

As already mentioned in the introduction, many terminology extraction systems use a hybrid approach where linguistic filtering is applied before ranking the selected candidate terms.

Linguistic filtering focuses on the fact that terminology usually have certain syntactic properties in common, i.e. they are *noun phrases* most of the time. To apply filtering on the text, part-of-speech (PoS) tags, which identify a words grammatical role in a sentence, are necessary. The filtering itself has been subject to many studies and experiments and is done in different ways.

For this, in general, four major steps are included in many terminology extraction systems. First, parsing and assigning PoS tags to the text. Second, syntactic filtering, which allows candidate terms which fit certain predefined PoS patterns, e.g. *adj noun* or *noun noun*. Third, eliminate semantic variations of a term in the text by transforming them into a single form. For example, 'word list' is a semantic variation of 'list of words' and can be used as a synonym. Fourth, other linguistic filtering to refine the candidate terminology, such as *stop-word filtering*.

Ranking candidate terms is possible in many ways. Statistical analysis, where the ranking is done based on statistical metrics, is one of them. The metrics used can be divided into two categories: metrics evaluating *unithood* (i.e. shows the strength of association between words) and those evaluating *termhood* (i.e. shows the words significance to the field). Extensive research has resulted in a large variety of statistical metrics [PaPZ05]. Other ranking techniques incorporate machine learning. These classifiers can be trained based on the extracted features of the selected words [WaLM16, YuGZ17, dSCPR13].

This approach on terminology extraction requires a reliable PoS tagger and linguistic knowledge of the respective language, as it is necessary to identify common syntactic patterns of terminology.

2.2 Sequence Labeling

Sequence Labeling is a part of machine learning, in which a sequence of observed features is classified. The sequence in this can be regarded as an entity, as its content is strongly related and its features depend on each other.

A good example for sequence labeling is part-of-speech tagging. A sentence is split into its words and for each of them its grammatical role in the sentence is determined. Classification can be done on all words independently, however accuracy usually improves if surrounding context information is added to the classification process [Bril00]. A word on its own can be placed as different grammatical roles. For example, considering the word 'ships', it can either be placed as a noun ('The ships are sinking.'), or as a verb ('He ships goods.') The disambiguation can be done by considering the word in front of 'ships'. In the first case, the article 'the' makes it a noun. In the second case, 'he' clarifies its role as a verb. Without the knowledge of the previous word, the classification is much more error prone.

2.3 Term Extraction as Sequence Labeling Task

Terminology extraction can be seen as a sequence labeling problem as well. It has already been mentioned that terminology extraction often considers words, which fall in a specific syntactic pattern. The PoS tags, these patterns use as a basis for distinction, are assigned using sequence labeling. Considering this, it should be possible to directly label terminology.

Taking this assumption as point of beginning, a general approach is easily implemented. The input text is taken as a whole and split into its words. For each word, features are selected, which are than classified using machine learning, e.g. recurrent neural networks, as they have shown to be especially good at sequence labeling [SuVL14, YZHS$^+$13]. The kind of feature is not restricted to a specific kind. Various different ones are possible. In the resulting output sequence, the terminology is labeled and can be used further, depending on the actual task.

This kind of approach has a two advantages compared to common techniques. First, a reliable PoS tagger is no longer necessary for the classification. And second, no linguistic knowledge and manually defined patterns for filtering are required.

3. Corpus

In context of linguistics, a *corpus* is a large structured set of texts. Depending on its purpose, parts of the text can be annotated with different labels. For this, a possible way is XML formatting.

As neural networks are part of supervised machine learning strategies [RuNo12], corpora containing annotated terminology are necessary for the training here. To use the data in a neural network architecture, the corpus data is split into sentences and then partitioned randomly into 80% training data, 10% validation data and 10% test data. A specified number of sentences, i.e. a *batch*, is input to the neural network. Since all sequences in a batch must be of equal length, the sentences are sorted by length in each of the sets before they are assigned to batches. Shorter sequences are padded at the end with special tokens. By sorting the sentences before they are assigned batches, less padding is required. Sentences, which consist of only one word, do not hold any substantial content and are removed from the data.

3.1 GENIA Corpus

The first corpus considered here, which contains annotated terminology, is the GENIA corpus [KOTT03]. It contains 1,999 Medline abstracts from PubMed[1]. In this collection of biomedical literature, terms are assigned to several categories, e.g., *DNA domain or region*. For the sake of simplicity, there is no differentiation between these categories of terms, since they are all specific to the biomedical field. This results in a total of 18,427 sentences with 437,307 words and 76,463 annotated terms. Counting the actual number of words in each annotated term, 37.91% of words in the corpus are terms.

3.2 ACL RD-TEC 2.0 Corpus

The second corpus used is the ACL RD-TEC 2.0 corpus [QaSc16] (in the following only ACL corpus). This corpus consists of 300 abstracts from the Association for

[1]https://www.ncbi.nlm.nih.gov/pubmed/

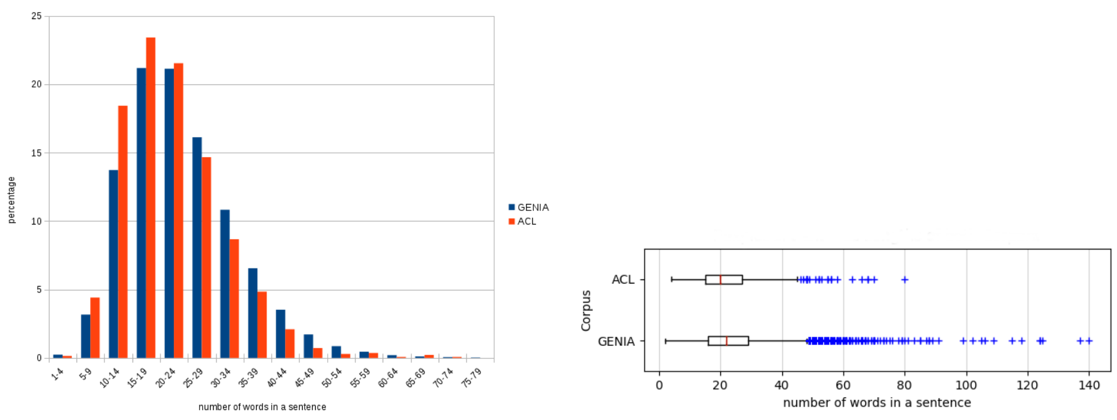
Computational Linguistics (ACL) Anthology Corpus. It contains 1,384 valid sentences with 29,921 words and 2,104 annotated terms. Here, 16.02% of words in the corpus are terminology.

As the GENIA corpus, the terms in the ACL corpus are assigned different classes such as *lr* or *tech*. One category, *other*, holds terms such as 'languages', 'text' or 'input'. This kind of term is not specific to the computational linguistic field and is not treated as term in the following work.

3.3 Comparison of these Corpora

As these copora are from entirely different fields, it is interesting to see, whether differences in sentence and word structure from a statistical point of view can be noted. For better comparability, all following distributions show percentages.

First of all, regarding the sentence lengths in both corpora, slight differences can be seen. A visual representation is given in figure 3.1. Figure 3.1(a) groups the sentence length in intervals of five and shows their percentage in the corpus. It can be seen, that the ACL corpus contains slightly shorter sentences than the GENIA corpus. The graphic shows, that the ACL corpus has more sentences with up to 25 words. The difference is largest for sentences with 10 to 15 words. This is also shown by the median, which is 20 for the ACL corpus and 23 for the GENIA corpus. Additionally, the sentence length varies less as it is shown in figure 3.1(b). In the GENIA corpus, the outliers are spread over a larger range, which is indicated by the number of blue marks.

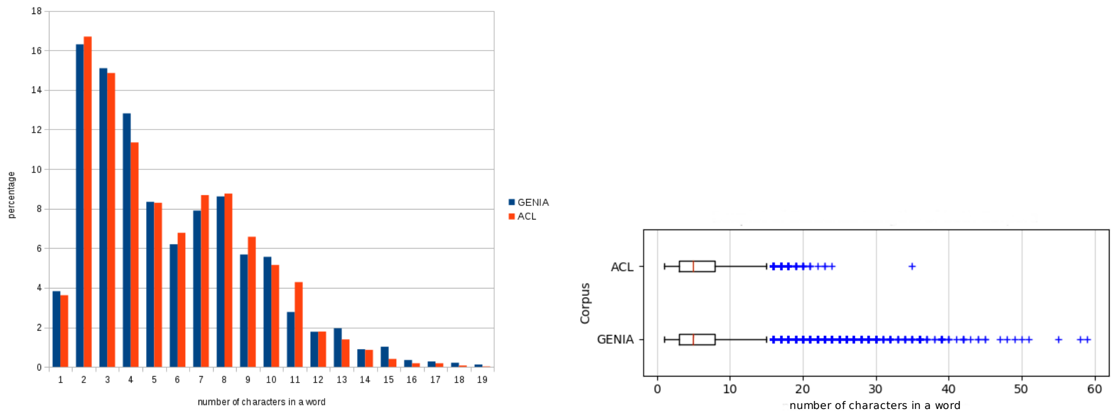


(a) by percentage (b) boxplot

Figure 3.1: Distribution of sentence lengths.

Considering the length of words in the corpora, the data is visualized in figure 3.2. As the similar distribution of word lengths shown in figure 3.2(a) indicates, both corpora are similar in terms of word length. Only slight differences can be noted. Short words prevail in both corpora. This can be shown by counting the words consisting of one to four characters. In the GENIA corpus, 48.04% of words and 46.52% of words in the ACL corpus belong to this category. Also, in both distributions a rise for words of seven to eight characters can be noted. For words with more tan 8 characters, the percentages in the GENIA corpus decreases faster with increasing length. However, the graph is stretched out in this dimension, as there are a few very long words in the GENIA corpus. The median of both corpora is five, which also shows their similarity for short words.

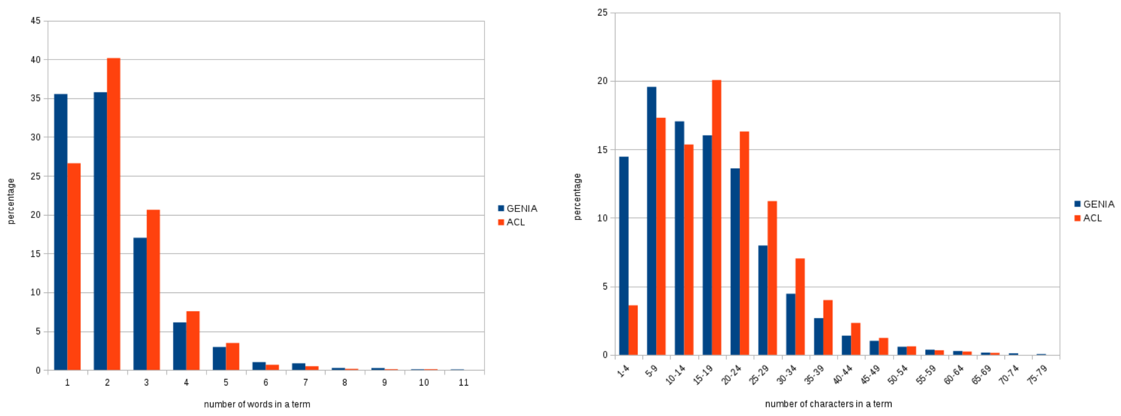
There are only two clear differences: first, again, the outliers are spread over a larger range in the GENIA corpus, which is depicted by the blue marks in figure 3.2(b). Second, the maximum word length encountered, which is 59 characters in the GENIA corpus and only 35 in the ACL corpus.



(a) by percentage (b) boxplot

Figure 3.2: Distribution of word lengths.

As for terms in the corpora, two different perspectives are shown in figure 3.3. First, the length of terms counted by their number of words, which is visualized in figure 3.3(a). Second, the number of characters in a term. In this case, multi-word terms are treated as one word and their number of characters are summed over all words, which are part of it. This can lead to extremely long words, so the percentages shown in figure 3.3(b) are formed over intervals of size five.



(a) wordwise (b) characterwise

Figure 3.3: Distribution of term lengths.

The distributions show significantly shorter terms in the GENIA corpus. Here, 50% of terms consist of 1 to 14 characters. Most terms are 5 to 9 characters long. Additionally, there are considerably more single-word terms in it than in the ACL corpus. The distribution shows about equally many single and two-word terms. In the ACL corpus, most terms consist of 15 to 19 characters. The majority of terms (63%) is 15 characters or longer.

Even though short words prevail in both corpora (see figure 3.2(a)), in the ACL corpus only 3.61% of terms are one to four characters long, whereas in the GENIA corpus 14.46% of terms are this short.

From a statistic point of view, the sentences in both corpora are quite similar. The major difference is noted when considering the structure of terms. Here, it was shown that terms in the GENIA corpus are shorter both word- and characterwise.

4. Model Architecture

4.1 Feed-Forward Neural Networks

Feed-Forward neural networks consist of several units, arranged in one or more layers. If there is more than one layer, the first layer is usually referred to as input layer, the last layer as output layer. The layers in between, if existent, are called hidden layers. Units in one layer are not connected to each other. The connections between layers are directed and form an acyclic graph leading from the input layer to the output layer [GoBC16, RuNo12].

Each unit in a layer calculates an output $o_j \in \mathbb{R}$ in the form of a linear combination of its inputs $x \in \mathbb{R}^n$ with $x_0 = 1$ as bias input and a weight vector $w_j \in \mathbb{R}^n$, which is passed through an activation function $g(\cdot)$ (see equation 4.1).

$$o_j = g(in_j) = g\left(\sum_{i=0}^{n} w_{i,j} x_j\right) \tag{4.1}$$

Combining the outputs of all units in the respective layer, the output vector $o \in \mathbb{R}^m$ is formed [RuNo12]. Also, the weight vectors w_j can be assembled into a weight matrix $W \in \mathbb{R}^{m \times n}$ to simplify representation.

For the activation function, many different functions are possible. Their influence on classification capabilities has been studied and their advantages and disadvantages compared against each other. Commonly used functions, among others, are sigmoid, tanh and ReLU [GoBC16]. A plot of these functions can be seen in figure 4.1.

Compared to ReLU, sigmoid and tanh posses the advantage that they are continuously differentiable, which is a useful property for the training algorithm introduced later on.

4.2 Recurrent Neural Networks

Normal feed-forward neural networks resemble a function computing an output based on its current input. Except for the weight matrices, this kind of network

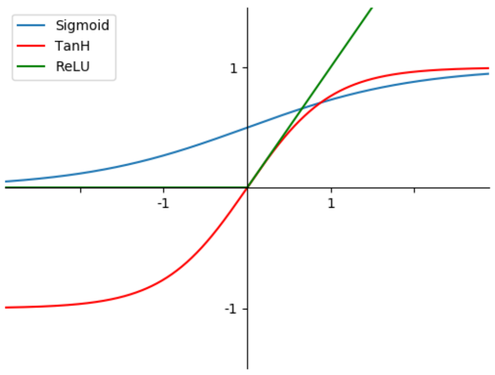


Figure 4.1: Plots of three commonly used activation functions.

does not posses any sort of internal state [RuNo12]. Contrary to this, recurrent neural networks (RNN) have an additional (recurrent) connection, which allows its hidden units to access their previously calculated output (sometimes also called state). Through these connections, RNNs gain a sort of memory [Jord97, Elma90]. In figure 4.2 a simple RNN is shown.

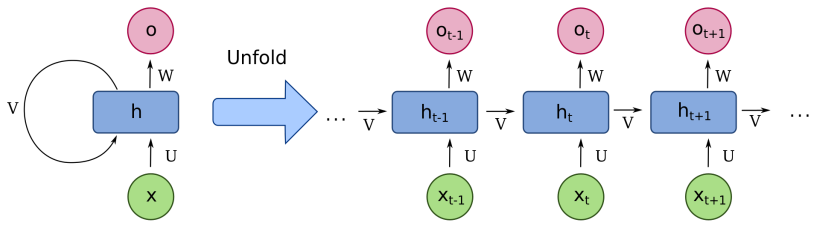


Figure 4.2: Simple recurrent neural network unit and its unfolded representation[1].

The input of a RNN is a sequence of T inputs $x_1, \dots, x_T$ with $x_t \in \mathbb{R}^n \, (t = 1, \dots, T)$. Each element in the sequence represents the current input of time step t. (As before, the biases are included in the weight matrices. In literature, equations stating the bias explicitly can be found as well.)

$$h_t = f(Ux_t + Vh_{t-1}) \tag{4.2}$$
$$o_t = g(Wh_t) \tag{4.3}$$

Using the input x_t, the hidden state $h_t \in \mathbb{R}^m$ is calculated using equation 4.2 where $U \in \mathbb{R}^{m \times n}$ and $V \in \mathbb{R}^{m \times m}$ denote the weight matrices and $f(\cdot)$ a non-linear activation function, which is applied to each element of the vector. The previous state is included by vector $h_{t-1} \in \mathbb{R}^m$. For the first iteration, $t = 1$, h_0 is usually initialized as zero vector. The output $o_t \in \mathbb{R}^o$ is calculated as shown in equation 4.3 with weight matrix $W \in \mathbb{R}^{o \times m}$ and a non-linear activation function $g(\cdot)$ [Suts13].

[1]Source: https://commons.wikimedia.org/wiki/File:Recurrent_neural_network_unfold.svg

4.2.1 Long Short-Term Memory

Long Short-Term Memory (LSTM) networks [HoSc97] are a variation of RNNs. Minor design changes led to a large variety of LSTMs. One basic form is seen in figure 4.3

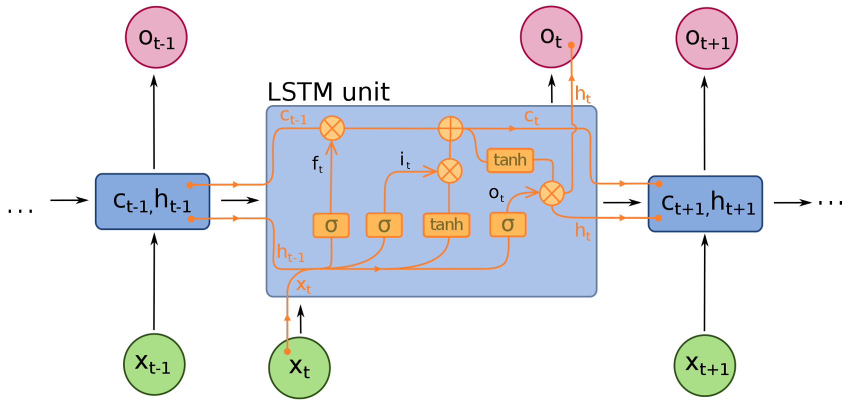


Figure 4.3: Basic form of a Long Short-Term Memory unit.[2]

The major refinement of the LSTM is the concept of a memory cell C. The information stored in the memory cell undergoes only minor linear interactions and functions as information conveyor. The removal or addition of information to the memory cell is controlled by *gates*. Gates make use of the sigmoid function, which outputs values between 0 and 1 and can be seen as a filter that restricts how much information the gate should let through. Outputting a 1 means to keep the information, 0 means to get rid of it. A LSTM has three gates: the forget gate f_t (this gate was proposed by [GeSC99] in a later refinement of the original LSTM), the input gate i_t and the output gate o_t.

The forget gate controls what information should be removed from the memory cell state and is calculated using equation 4.4 with $U^f \in \mathbb{R}^{m \times n}$ and $V^f \in \mathbb{R}^{m \times m}$ as weight matrices. It determines which parts of the information stored in the memory cell are important and should be remembered and which are not. This function is important, when a change in subject occurs in the input sequence. In this case, the information must be removed from the memory cell, which is achieved by a forget gate output near 0.

$$f_t = \sigma(U^f x_t + V^f h_{t-1}) \tag{4.4}$$

The input of information into the memory cell requires two calculations: first, a candidate memory cell state $\tilde{C}_t$ is calculated using equation 4.5 where $U^C \in \mathbb{R}^{m \times n}$ and $V^C \in \mathbb{R}^{m \times m}$ denote the weight matrices. This candidate state is derived from the previous hidden state $h_{t-1} \in \mathbb{R}^m$ and the current input x_t. Second, a filter is generated that selects parts from the candidate memory cell state that will be put into the memory cell (see equation 4.6 with weight matrices $U^i \in \mathbb{R}^{m \times n}$ and $V^i \in \mathbb{R}^{m \times m}$).

$$\tilde{C}_t = tanh(U^C x_t + V^C h_{t-1}) \tag{4.5}$$
$$i_t = \sigma(U^i x_t + V^i h_{t-1}) \tag{4.6}$$

[2]Source: https://commons.wikimedia.org/wiki/File:Long_Short-Term_Memory.svg

The information in the memory cell is updated by removing the information determined by the forget gate and adding the parts of the candidate cell state, the input gate selected, using calculation 4.7. In this calculation $\circ$ denotes the Hadamard product.

$$C_t = f_t \circ C_{t-1} + i_t \circ \tilde{C}_t \tag{4.7}$$

The last step is to calculate the output h_t of the LSTM, which is basically a filtered version of the memory cell state. The filter is generated by the output gate as seen in calculation 4.8 (with weight matrices $U^o \in \mathbb{R}^{m \times n}$ and $V^o \in \mathbb{R}^{m \times m}$), and multiplied with the output of the tanh function applied to the cell state which puts its values into the interval $[-1, 1]$.

$$o_t = \sigma(U^o x_t + V^o h_{t-1}) \tag{4.8}$$
$$h_t = o_t \circ tanh(C_t) \tag{4.9}$$

4.2.2 Gated Recurrent Unit

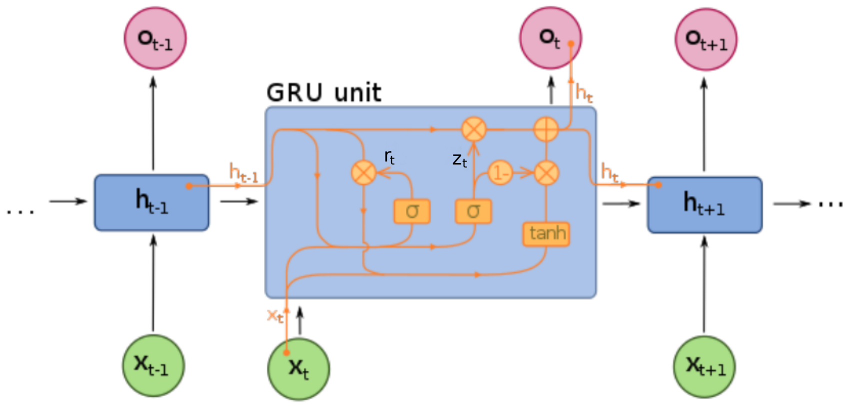


Figure 4.4: Basic form of a Gated Recurrent Unit[3].

The Gated Recurrent Unit (GRU) [CMGB+14] is a variation of a LSTM. It has only two instead of three gates, the reset gate r_t and the update gate z_t, and thus requires fewer parameters to be optimized. A picture of a GRU is given in figure 4.4.

The reset gate is calculated using equation 4.10 with weight matrices $U^r \in \mathbb{R}^{m \times n}$ and $V^r \in \mathbb{R}^{m \times m}$. Similar to this, the update gate is calculated using equation 4.11 with $U^z \in \mathbb{R}^{m \times n}$ and $V^z \in \mathbb{R}^{m \times m}$.

$$r_t = \sigma(U^r x_t + V^r h_{t-1}) \tag{4.10}$$
$$z_t = \sigma(U^z x_t + V^z h_{t-1}) \tag{4.11}$$

The new candidate hidden state $\tilde{h}_t$ is calculated using the reset gate. The final memory update combines the new candidate hidden state and the last hidden state using the update gate. As before, the hidden state is also the output of the network.

$$\tilde{h}_t = tanh(U^h x_t + V^h (r_t \circ h_{t-1})) \tag{4.12}$$
$$h_t = z_t \circ h_{t-1} + (1 - z_t) \circ \tilde{h}_t \tag{4.13}$$

[3]Source: https://commons.wikimedia.org/wiki/File:Gated_Recurrent_Unit.svg

If the output of the reset gate is close to zero, equation 4.12 shows that the previous hidden state is dropped and only the current input x_t is used. The update gate controls how much from the previous hidden state remains in the new one. As all hidden units in a layer posses their own reset and update gate, it is possible for each of the units to learn to store information over a different time span. Units, where the reset gate is frequently activated, can keep information only for short time periods, while others, where the update gate is activated more, learn long-term dependencies [CMGB+14].

4.2.3 Bidirectional Recurrent Layer

The recurrent connections of RNNs make it possible to consider past inputs for the current classification. In natural language however, preceding *and* subsequent words can affect the role of a word. For this purpose bidirectional recurrent layers can be used. This alteration virtually splits the hidden units in two halves. One half handles the input sequence in normal order, the other half in reverse order. The outputs from forward units have no connection to any input of the reverse units. The same goes for the other way round. This structure of a recurrent neural network simplifies the usage of past and future information to the respective time step [ScPa97].

4.3 Embedding Layer

To use a neural network for classification, it is necessary to pick a fitting representation of the input. In this case, the input is a sentence, which is split into its words. For each word, features, which represent the characteristics of and identify the word, are selected. Additional features can be statistical features such as word frequency. As for identification of the word, one-hot encoding is a possible way to encode the input words. In this encoding, the input vector has the size of the vocabulary. The vector is entirely zero except for the index, which represents the word in the vocabulary. However, with increasing vocabulary, this leads huge input vectors. The term *curse of dimensionality* is frequently used in context of machine learning when referring to problems with very high input dimensionality [BeCo57, BeBe61, GoBC16]. In the setting of this work, the input dimensionality is the size of the vocabulary used in the input text, which can easily exceed several thousands words. An efficient representation of input sequences is essential to good performance. Additional statistical features are not intended to be used, for the time being.

In natural language processing, so called *embeddings* are commonly used, which translate a one-hot-vector representing a specific word (in some implementations a unique index is used instead) to a lower dimensional continuous vector space [BDVJ03]. The training of such embedding layers is usually done by building a classifier for an auxiliary classification task, for example the building of a language model, and letting the network train a fitting representation of the input.

Since test data will always feature words that have not been seen during training, the proper handling of unknown words is important when performing embedding. There are two general methods for dealing with out-of-vocabulary words:

1. A pretrained word embedding with an *UNK* token, which is used for all out-of-vocabulary words.
2. A character-level embedding, which is independent from the vocabulary size.

4.3.1 Word-level Embedding

For word-level embeddings, a pretrained embedding, namely the GloVe 6B token embedding with 300 dimensional vectors [PeSM14], is used. This word representation is trained in an unsupervised manner on the non-zero entries of a word-word co-occurence matrix. Each entry represents the frequency of which two words occur along with each other. 6B refers to the size of the data used to build the co-occurence matrix. For the GloVe 6B, Wikipedia 2014 and Gigaword 5 have been used for the collection of data. There are several different versions of GloVe, which can be downloaded from the Stanford website[4]. Those other packages are trained on larger text collections and have a larger vocabulary. They have not been used for this work, as they exceed the available memory.

These word vectors were set to untrainable so that they could not be altered during the training of the network. To handle the problem of out-of-vocabulary words an UNK token needs to be added manually. Additionally, a *start-of-sentence token* (SOS) and an *end-of-sentence token* (EOS) are manually included in the vocabulary. Adding the tokens to the vocabulary size given by the GloVe embedding, the vocabulary size is now 400,003.

4.3.2 Character-level Embedding

The character embedding input vocabulary consists of Python's *string.printable* characters [Lutz13] as well as an UNK token. The UNK token is included, as crashing of the program in case there are non-ASCII characters in the input sentences when no preprocessing is applied, should be avoided. Shorter sequences are padded with a '.' instead of a special EOS token. Thus, the vocabulary size is 101.

The character embedding can either be trained end-to-end with the input sequences or as part of a language model (intending to predict the next most likely character). The results of both strategies are compared against each other in section 5. In the latter case, the vectors are set to untrainable so that they cannot be altered during the training of the sequence classifier. Various tests indicated good results for an embedding dimension of 50.

4.4 Linear Mapping

The number of hidden units in the recurrent layer is variable and different sizes are selected in order to analyze the effect on the classification capabilities. Therefore, a final mapping to the number of target classes is necessary. This is achieved by a linear layer, which applies a linear transformation to sample $x_t \in \mathbb{R}^n$ using a weight matrix $A \in \mathbb{R}^{m \times n}$ and bias $b \in \mathbb{R}^m$ [PGCC18]. For the application here, m must be set to the number of classes.

$$y_t = Ax_t + b \tag{4.14}$$

$$softmax(\boldsymbol{y_t})_j = \frac{e^{y_{tj}}}{\sum_{c=1}^{C} e^{y_{tc}}} \tag{4.15}$$

The resulting probability distribution $y_t \in \mathbb{R}^C$ is normalized using the softmax function seen in equation 4.15 (with C the number of classes and j as the jth entry in y_t), so that the sum over all entries becomes one. The value in y_{tj} represents the probability of the sample belonging to class j.

[4]https://nlp.stanford.edu/projects/glove/

4.5 Network Training

The basic idea of training a neural network is to fit its predictions to the training data provided. This means, the error made by the network should be minimal. To do so, the weights of all weight matrices in the network need to be adjusted with respect to the error made.

4.5.1 Loss Function

In order to measure the error, loss functions (sometimes also error or cost function) are used. As the output of this neural network model is a probability distribution over the number of classes, a measure to describe the similarity (or dissimilarity) between the target distribution and the predicted distribution is needed. The target probability distribution is represented by a one-hot vector, where the position of the 1 in the vector marks the index of the target class.

For that matter, cross entropy, which is closely related to the Kullback-Leibler Divergence $\mathbb{D}_{KL}(p||q)$, is used. In [GoBC16], cross entropy is defined as:

$$H(p,q) = H(p) + \mathbb{D}_{KL}(p||q) \tag{4.16}$$

In this equation, p and q are both probability distributions. For the discrete case, this can be simplified using the functions 4.17 for the entropy $H(p)$ and 4.18 for the Kullback-Leibler Divergence $\mathbb{D}_{KL}(p||q)$.

$$H(p) = \sum_x p(x) log \frac{1}{p(x)} \tag{4.17}$$

$$\mathbb{D}_{KL}(p||q) = \sum_x p(x) log \frac{p(x)}{q(x)} \tag{4.18}$$

$$\begin{aligned} H(p,q) &= \sum_x p(x) log \frac{1}{p(x)} + \sum_x p(x) log \frac{p(x)}{q(x)} \\ &= \sum_x p(x) \left(log \frac{1}{p(x)} + log \frac{p(x)}{q(x)} \right) \\ &= \sum_x p(x)(log\, 1 - log\, p(x) + log\, p(x) - log\, q(x)) \\ &= -\sum_x p(x) log\, q(x) \end{aligned} \tag{4.19}$$

To form a loss function from cross entropy, the mean loss over all samples must be computed. Assuming that $y_n \in \mathbb{R}^C$ is the target and $\hat{y}_n \in \mathbb{R}^C$ that estimated probability distribution for the nth of N samples, the cross entropy loss function is calculated as stated in equation 4.20.

$$\begin{aligned} \mathcal{J}(y,\hat{y}) &= -\frac{1}{N} \sum_{n=1}^{N} H(y_n, \hat{y}_n) \\ &= -\frac{1}{N} \sum_{n=1}^{N} \left(\sum_{c=1}^{C} y_{nc} \cdot log(\hat{y}_{nc}) \right) \end{aligned} \tag{4.20}$$

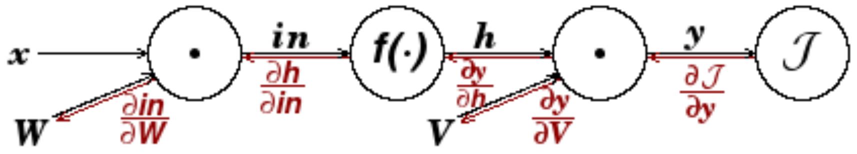


Figure 4.5: Computational graph of a two-layered feed-forward neural network including the loss function $\mathcal{J}$. The red edges mark the derivatives calculated during backpropagation.

4.5.2 Backpropagation

The *backpropagation* algorithm [RuHW86] is used to calculate the gradient of the error with respect to the weights of a neural network. To do so, using a computational graph (see figure 4.5), the chain rule is applied iteratively to all edges. These gradients will be used later on for the weight adaptation.

To explain the basic mechanism, a simple feed-forward neural network with two layers is assumed. Here, $x \in \mathbb{R}^n$ is the input vector, $W \in \mathbb{R}^{m \times n}$ and $V \in \mathbb{R}^{o \times n}$ and $f(\cdot)$ an activation function. The computational graph of this network is shown in figure 4.5.

$$in = Wx \tag{4.21}$$
$$h = f(in) \tag{4.22}$$
$$y = Vh \tag{4.23}$$

As the total loss is an accumulation of losses generated by each unit, it is necessary to determine each weight's influence on it. This is achieved by calculating the partial derivative with respect to every weight. The computational graph shows that y depends on h, and h in turn depends on in. For such cases, the chain rule needs to be applied. Assuming that $z = f(y)$ and $y = g(x)$ with mapping functions f and g, the derivative of z with respect to x is calculated as follows:

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x} \tag{4.24}$$

Using this equation, it is possible to calculate the gradient with respect to the weight matrices W and V as follows.

$$\frac{\partial J}{\partial V} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial V} \tag{4.25}$$
$$\frac{\partial J}{\partial W} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial h}\frac{\partial h}{\partial in}\frac{\partial in}{\partial W} \tag{4.26}$$

As these calculations show, each partial derivative can be assigned to an edge in the computational graph. Thus, it is possible to reuse previously calculated derivatives while passing through the graph from the end to the beginning [GoBC16].

4.5.3 Backpropagation Through Time

To apply the backpropagation algorithm to a recurrent neural network, the network must be unrolled (as seen in figure 4.2). Backpropagation is applied to the unrolled model. So the size of the model increases with the number of timesteps, which need to be unrolled. The process, to calcuate the gradient for this, is called *Backpropagation Through Time* (BPTT) [Suts13, PaMB13].

Not considering the trivial case for the weight matrix W, the derivative can be calculated as follows [PaMB13]:

$$\begin{aligned} \frac{\partial E}{\partial \theta} &= \sum_{1 \le t \le T} \frac{\partial E_t}{\partial \theta} && (4.27) \\ &= \sum_{1 \le t \le T} \sum_{1 \le k \le t} \left(\frac{\partial E_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial \theta} \right) \\ \text{with} \quad \frac{\partial h_t}{\partial h_k} &= \prod_{t \ge i > k} \frac{\partial h_i}{\partial h_{i-1}} = \prod_{t \ge i > k} V^T diag\left(f'(h_{i-1})\right) && (4.28) \end{aligned}$$

In equation 4.28, the function $diag(\cdot)$ converts a vector into a diagonal matrix and $f'(\cdot)$ is the derivative of activation function $f(\cdot)$, which is applied element-wise to the passed vector.

The problem with BPTT lies with the term $\frac{\partial h_t}{\partial h_k}$, which is the product of $t-k$ matrices. Each of these matrices is the product of two matrices. Thus, its 2-norm is bounded by the 2-norms of both factor matrices.

$$\forall k, \left\| \frac{\partial h_{k+1}}{\partial h_k} \right\| \le \left\| V^T \right\| \, \left\| diag(f'(h_k)) \right\| \tag{4.29}$$

With $|f'(x)|$ bounded, the diagonal matrix $\|diag\left(f'(h_{i-1})\right)\| \le \gamma \in \mathbb{R}$ is bounded as well. Assuming λ_1 as the largest singular value of V^T, two developments for $\frac{\partial h_t}{\partial h_k}$ are possible. If $\lambda_1 < \frac{1}{\gamma}$, the multiplication of V^T and $diag(f'(x_k))$ leads to a number $\eta < 1$, $\eta \in \mathbb{R}$, so $\left\| \frac{\partial h_i}{\partial h_{i-1}} \right\| \le \eta < 1$. Inserting this into the product leads to $\left\| \left(\prod_{t \ge i > t} \frac{\partial h_i}{\partial h_{i-1}} \right) \right\| \le \eta^{t-k}$. As $\eta < 1$, $\lim_{t-k \to \infty} \eta^{t-k} = 0$. If $\lambda_1 > \frac{1}{\gamma}$, the multiplication of matrices leads to $1 < \left\| \frac{\partial h_i}{\partial h_{i-1}} \right\| \le \eta$. Now, the product of matrices leads to an exponentially growing number with increasing $t-k$. The first case is called *vanishing gradient problem*, the second *exploding gradient problem*. Because of these to issues, RNNs have difficulties learning long-term dependencies [PaMB13, Suts13, BeSF94, Hoch91]. Both LSTMs and GRUs do not suffer from this problem, since their memory is protected by the gates [HoSc97].

4.5.4 Optimizer

The adaptation of weights during training is done by an optimizer, e.g. *stochastic gradient descent* (SGD) [Bott10]. SGD is a simplification of gradient descent (equation 4.30), a method to optimize a neural network or another kind of classifier, by minimizing the loss function. The gradient is calculated using the partial derivatives $\frac{\partial \mathcal{J}}{\partial w}$ and can be seen as a vector pointing in the direction of a local minimum. The learning rate η (sometimes step size) can be interpreted as a weighting factor that determines the magnitude of the gradient's influence on the newly calculated weights w_{t+1}.

With gradient descent, the weights are adjusted based on the averaged sum of all derivatives of the loss function $\mathcal{J}$ over *all* samples N in the training set. As this is computationally expensive, a *stochastic* element is introduced. For stochastic gradient descent, as seen in equation 4.31, a random sample is picked to calculate the derivative of the loss function and adjust the weights based on it [Bott10]. A compromise between both extremes is to calculate the derivative of the loss based on a batch of random samples and adjust the weights accordingly. This is called mini-batch gradient descent [GoBC16].

$$w_{t+1} := w_t - \eta \frac{1}{N} \sum_n \frac{\partial \mathcal{J}}{\partial w} \tag{4.30}$$

$$w_{t+1} := w_t - \eta \frac{\partial \mathcal{J}}{\partial w} \tag{4.31}$$

4.5.5 Scheduler

When one epoch (i.e. complete iteration over all training samples) is finished the average loss over all validation batches is calculated. As with the training batches, the validation batches are shuffled before they are passed into the network. If the loss on the validation batches does not decrease by a specified threshold or increases again, the learning rate must be reduced. This is done by a *scheduler*. Here, the newbob scheduler [dpwe00] is used, which multiplies the learning rate with 0.5 after each iteration once the scheduler has been activated.

The training of the network is finished, if the learning rate becomes too small or the validation loss decreases too little even though the scheduler has already been activated.

4.5.6 Regularization

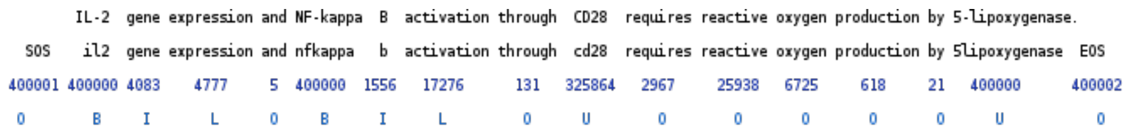
As the network learns classification based only on the training data, it is a central problem how to ensure its good performance on new input as well. Models, which are heavily adapted to the training data, usually generate high loss on the test data. This is called *over-fitting*. Over the years, a variety of strategies have been proposed that aim to reduce the test error (sometimes also called generalization error in this context). The collective term for such techniques is *regularization*.

The various regularization strategies modify different parts of the learning algorithm. Restrictions to the parameter values of the model or adding objective functions, which impose soft constraints on the parameter values are possible. Others create penalties from previously obtained knowledge. The variety of strategies is seemingly endless [GoBC16].

4.6 Data Preparation

4.6.1 Labeling

For the embedding layer, the input sequence must be translated into a sequence of unique indices, each representing a specific word or token. Parallel to the input sequence, a target label sequence is needed. For this purpose, BILOU [RaRo09]



```
       IL-2  gene expression and NF-kappa  B  activation through  CD28  requires reactive oxygen production by 5-lipoxygenase.
  SOS    il2  gene expression and nfkappa   b  activation through  cd28  requires reactive oxygen production by 5lipoxygenase  EOS
400001 400000 4083   4777    5 400000 1556  17276    131  325864  2967   25938  6725   618   21  400000    400002
  O      B     I      L      O    B     I     L       O     U      O      O      O     O     O     U         O
```

Figure 4.6: Labeling and preprocessing for word embedding applied to a sentence.

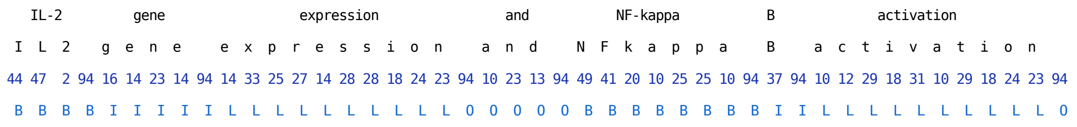
encoding of the labels is used. BILOU stands for *begin, inside, last, outside, unit*. Each of these labels represents a word's role in the sequence. Single-word terms are tagged with `u` *(unit)*. For multi-word terms, the first word is tagged with `b` *(begin)*, the last with `l` *(last)*. The words in between are labeled with `i`*(inside)*. This results in a label sequence, which can be expressed by the regular expression `b i* l`. All other words in the input, i.e. non-terms, are labeled `o`*(outside)*.

Given the label scheme, the distribution of labels in the corpora is a factor influencing the classification capabilities. Calculating the percentages for each class in each corpus, the resulting label balance is shown in table 4.1. The percentages show that the ACL corpus is very imbalanced.

	Begin	Inside	Last	Outside	Unit
GENIA	10.39	8.43	10.39	65.06	5.73
ACL	4.71	3.48	4.71	85.40	1.71

Table 4.1: Calculated percentages of each label in both corpora.

Figure 4.6 shows text preprocessing and labeling applied to an example sentence. In the first line the original sentence is seen. In the second, punctuation and capital letters have been removed. The third line represents the set of unique indices defined by the GloVe word vectors. As the index 400000 represents an unknown word, it shows that three words in this sentences are unknown, i.e. *il2, nfkappa* and *5lipoxygenase*. The last line shows the generated label sequence.



```
 IL-2      gene          expression          and      NF-kappa      B        activation
 I  L  2   g  e  n  e   e  x  p  r  e  s  s  i  o  n   a  n  d   N  F  k  a  p  p  a   B   a  c  t  i  v  a  t  i  o  n
44 47  2 94 16 14 23 14 94 14 33 25 27 14 28 28 18 24 23 94 10 23 13 94 49 41 20 10 25 25 10 94 37 94 10 12 29 18 31 10 29 18 24 23 94
 B  B  B  B  I  I  I  I  I  L  L  L  L  L  L  L  L  L  L  O  O  O  O  O  B  B  B  B  B  B  B  B  I  I  L  L  L  L  L  L  L  L  L  L  O
```

Figure 4.7: Labeling and preprocessing for character embedding applied to part of a sentence.

When character embedding is used, the words are split into their characters as seen in the second line in figure 4.7. All of the single characters, even spaces, are given a unique index and a label. Spaces inside multi-word terms are given the label of their predecessor, as seen between the words *IL-2* and *gene*.

4.6.2 Text Preprocessing

To use the GloVe word vectors, all words need to be lower case. Additionally, all punctuation is removed from the input sequences.

With character embedding, different grades of preprocessing of the text are possible, i.e. no preprocessing at all, transformation to lowercase characters or removal of all punctuation combined with only lowercase characters. To ensure that these different grades of preprocessing do not lead to different number of labels in the same sentence, which may distort comparability of the scores, an additional word list and word target label list is generated.

During the label calculation, the character sequence is aligned with the word list. This results in a probability matrix of size $word\,length \times classes$. From this matrix, the class can be calculated by either majority voting or score summation. In the first case, the class with the highest probability is determined for each row. The most frequent class over all rows is then chosen as word label. In the latter case, the probabilities are summed column wise and normalized afterwards. From the resulting probability distribution, the highest probability determines the word class.

4.7 Model Topology

The general structure of the complete neural network model consists of three segments: first an embedding layer, second, a sequence classifying layer and last, a linear layer mapping to the five output classes. For the second segment, different configurations are compared against each other:

a) one softmax layer with input window sizes 1, 3 or 5.

b) unidirectional LSTM or GRU

c) bidirectional LSTM or GRU

d) multi-layer LSTM or GRU

5. Experiments

5.1 Metrics

The chosen metric is supposed to reflect the quality of the classifier. For example, accuracy is given by the number of correct predictions divided by the total number of predictions. However, there is a flaw in accuracy when rare classes are contained in the data. Assuming that class B is only present in 5% of the data, the achieved accuracy is 95% by simply selecting class A for all samples. The accuracy is high, even though the trained classifier performs terrible.

So instead of accuracy, three different metrics are calculated to measure the performance of the network. *Precision* is used to give an estimate of how many of the predicted labels are correct and is calculated by equation 5.1. In context of terminology extraction, it describes how many of the extracted words really *are* terminology.

$$Precision = \frac{true\,positive}{true\,positive + false\,positive} \tag{5.1}$$

The second measure is *recall*, which is calculated with equation 5.2. It describes how many elements of a class were correctly selected. Thus, it estimates how many of the terminology were found in the text.

$$Recall = \frac{true\,positive}{true\,positive + false\,negative} \tag{5.2}$$

The last score is the F-Score, which is used to give an estimate of the quality of the classifier in most literature in this field. It is the harmonic mean of precision and recall (see equation 5.3) [Powe11].

$$F - Score = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{5.3}$$

For the calculation the *scikit-learn* module from python was used, which calculates the scores for each class label separately. The resulting scores are averaged over the number of classes to achieve a batch score. These batch scores are again averaged arithmetically over all batches in a set.

To compare the set scores, a baseline is calculated. It is set to the score achieved, when the most common label *Outside* (`o`) is chose for every word.

Additionally, average cross entropy loss, run time and average number of epochs are used as additional characteristics for comparison in some of the experiments. For all these variables, the average absolute deviation can be calculated to indicate the dispersion of the results.

$$\begin{aligned} \mathcal{AAD}(X) &= \frac{1}{N}\sum_{n=1}^{N} |x_n - m(X)| \qquad (5.4)\\ \text{with } X &= \{x_1, \ldots, x_N\} \text{ and}\\ m(X) &= \frac{1}{N}\sum_{n=1}^{N} x_n \end{aligned}$$

All training runs were executed on the same graphics card, a Nvidia GeForce GTX 1070, using the cuda API[1]. The neural networks are implemented in Python using the Pytorch framework [PGCC18].

5.2 Results

As the test, validation and training set are selected randomly from a set of sentences, the test results may vary slightly for individual test runs. To give a better estimate, several test runs were conducted with the same topology and hyperparameters and the results averaged over the number of test runs. The numbers in the following are all average numbers, if not stated otherwise.

5.2.1 Relation between F-Score and Loss

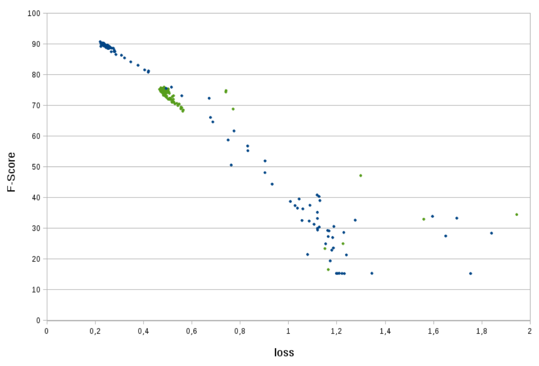
As seen in section 4.5.1, the network is trained using a loss function, in this case, cross entropy loss. However, the metric used to measure performance of the network is the F-Score. So, there must be a correlation between F-Score and loss for the training to be successful (at least for test runs with relatively small loss). Otherwise, a well-trained network with only little test loss cannot guarantee a high F-Score.

Since most test runs were conducted with a topology and hyperparameters aimed to achieve minimal loss, the data for large losses is sparse. Furthermore, the loss can become very large, which spreads the calculated F-Scores for these runs over an even larger area. This influences the visibility of a possible correlation in the respective interval.

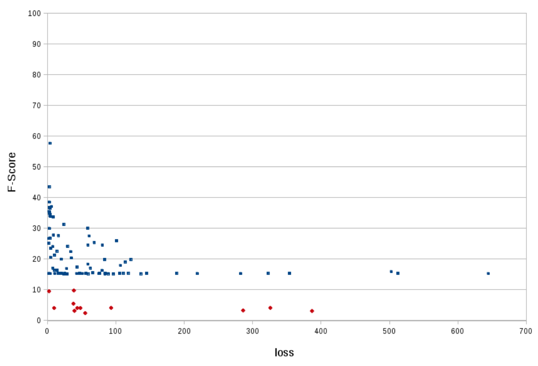
In figure 5.1, the achieved F-Scores of individual test runs with character embedding are plotted over the respective loss. Those test runs include several different hidden sizes and number of layers. The green dots in this figure, represent test runs using a unidirectional LSTM.

For better visibility, test runs with small losses between 0 and 2 are shown separately in figure 5.1(a). Larger losses between 2 and 700 are plotted in figure 5.1(b). There

[1] https://developer.nvidia.com/cuda-zone



(a) Loss interval $[0, 2]$



(b) Loss interval $(2, 700]$

Figure 5.1: Achieved F-Scores plotted over the loss.

are very few runs with even greater loss. However, these were omitted in this graph, as it would decrease the discriminability of data points in the interval (2,100].

For losses between 0 and 0.6 a correlation is evident. The data points are very close to each other and from a visible line. Between 0.6 and 1.2 the correlation is not as strong as before, but still visible. The dispersion of points increases, but it is still possible to approximate a representative curve using the data points. For losses above 1.2, the F-Scores lies relatively random in the interval $[15, 50]$.

Figure 5.1(b) shows that most of the runs do not drop below the baseline 15. For losses between 2 and 100, there are various cases where the F-Score is significantly better than the baseline. In this interval no clear correlation between loss and F-Score is visible. For larger losses between 100 and 700, the achieved score is usually around the baseline.

There are, however, very few peculiar cases (marked in red in figure 5.1(b)), where the F-Score becomes so low, that it appears the network learned the exact opposite of what it was supposed to learn. These particular cases occur only very rarely (and only if a batch size above 25 is used) and cannot be caused deliberately. Finding a cause for these results is currently not possible given their rare and unpredictable occurrences.

From these observations, it is possible to postulate that there is indeed a correlation between F-Score and loss for small losses below 1.

5.2.2 Parameter Tuning

There are a number of parameters, which may influence performance and training time of the network, such as learning rate, batch size, the number of layers and the size of the hidden layers. As an exhaustive search of all possible parameter combinations takes impossibly long, the influence of one parameter at a time is analyzed. This is done by conducting experiments where the respective parameter is varied.

5.2.2.1 Experiments with different Learning Rates

Learning rate was introduced in section 4.5.1 as a weighting factor for the gradient and thus affects the weight update. A larger learning rate can reduce the time it takes to converge to a local minimum during training, but it can also obstruct convergence by constantly ’overstepping’ the local minimum. A too small learning rate increases the training time and may cause the training to end prematurely, since the validation improvement between two epochs is too small and training stops.

Choosing a ’good’ learning rate depends on various other factors. Networks with more and larger layers require smaller learning rates. For smaller networks, training converges faster with slightly larger learning rates. With increasing batch size, the learning rate must be chosen smaller.

In general, experiments have shown, that the learning rate should be smaller than 0.001 but larger than $1e^{-6}$ for both LSTMs and GRUs.

5.2.2.2 Experiments with different Batch Sizes

In this network setup, mini-batch gradient descent is used, where the gradient is calculated over a random batch. Thus, with growing batch size the possible error made grows as well. This means that the batch size influences the adjusting of the weights by influencing the calculated loss. As with the learning rate, a larger batch size may reduce training time, but may as well obstruct convergence. Smaller batch sizes increase the chance to converge to a local minimum, but also increase training time.

For the experiment, batch sizes between 5 and 50 in steps of 5 have been tested. The learning rate has been set to 0.001 for all tests. Four topologies using character embedding were used. One unidirectional LSTM with one layer and hidden size 400, two bidirectional LSTMs with one layer and hidden sizes 200 and 400 and one bidirectional LSTM with two layers and a hidden size of 400.

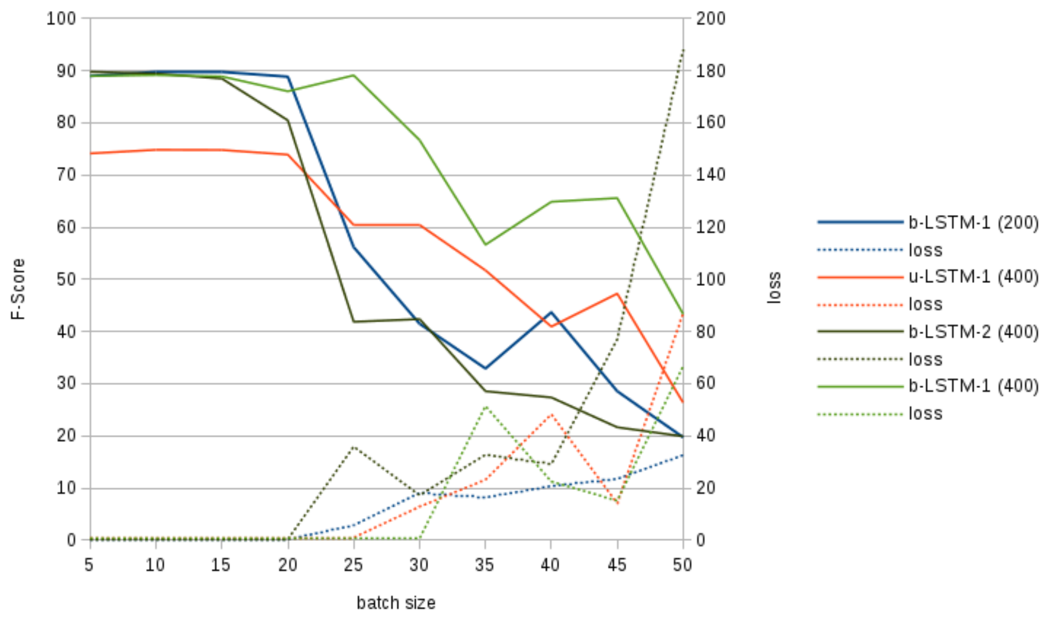


Figure 5.2: Impact of different batch sizes on F-Score. In x-LSTM-n x denotes direction and n number of layers. The numbers in brackets refer to the size of the hidden layer(s).

In figure 5.2 the average F-Score of test runs is plotted over the batch size. The dotted lines represent the average loss of the respective configuration.

The graphic shows that for all tested configurations batch sizes up to 15 have no positive or negative impact on the F-Score. For a batch size of 20, the average performance slightly decreases. Especially for b-LSTM-2 (400), the F-Score decreases by 10 between batch size 15 and 20. For batch sizes above 20, the performance becomes increasingly unstable. The decrease in F-Score is not consistent for any of the shown topologies. In general, it is shown that the F-Score decreases with the batch size, but rises and falls randomly between two compared batch sizes. This shows that the random distribution into test sets, the random choice of batches during training and the weight initialization done by the framework can all lead to a single run with a high F-Score even though all other test runs with the same batch size achieve very low scores. In other words, luck attributes to good test runs.

Due to the random decrease, it is difficult to determine a correlation between topology and deterioration speed. The two-layered network (b-LSTM-2 (400)) performs worst most of the time. However, the single layer LSTM with the smaller hidden size (b-LSTM-1 (200)) deteriorates faster than the one with the larger hidden size (b-LSTM-1 (400)). So, number of weights does not appear to be a factor for the deterioration rate. The score of the unidirectional network (u-LSTM-1 (400)) appears to sink slowest most of the time. A possible factor may be the number of directions, but this cannot be proven using this data alone.

As the loss of a good test run, with a F-Score above 70, normally lies between 0.2 and 0.8, an average loss significantly larger than this indicates an increase in failed test runs. Failed meaning the achieved F-Score deviates from the best performance with the respective topology by more than 10. Analyzing the loss of the listed networks confirms the previous observation. For batch sizes smaller or equal to 20, the loss remains small and the scores are (relatively) constant. For larger batch sizes, the loss and in turn the number of failed test runs rises. The random decrease in F-Score is reflected by the fluctuations in loss for batch sizes larger than 25. It is shown again, that the deterioration is not linearly dependent on the batch size.

Considering the possible advantages for training time, the average number of epochs and the average epoch duration is calculated. The results are slightly distorted, as the the measured time does not only include the training time but also the time needed for testing. So the error made for runs with fewer training epochs is slightly larger than for others with more training epochs. However, testing takes only few seconds, so the error made is negligible for the observation of a general trend, which is seen in figure 5.3.

For the average number of epochs no general trend is given. With batch sizes up to 15, where most training runs succeed, the number of epochs is relatively stable. The larger the batch size becomes, the more the graphs fluctuate. One possible reason lies in the number of epochs for failed test runs. While successful test runs require 8 to 12 training iterations in most cases, failing test runs either abort after up to four runs or are aborted by the program after exceeding the defined maximum number of epochs, which is set to 15. This unpredictable behaviour affects the average number for larger batch sizes.

The average epoch duration shows that two-layered networks generally take longer for one epoch by a relatively constant time period. Making the network bidirectional also slightly increases the average epoch time. This increase is not as constant as

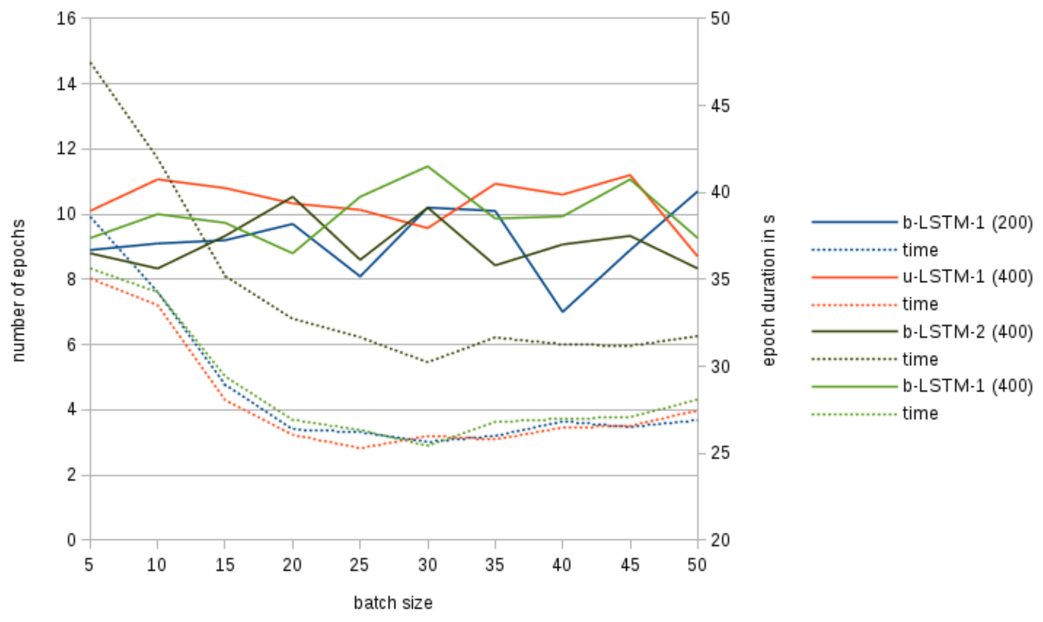


Figure 5.3: Number of epochs and epoch duration for listed topologies plotted over the selected batch size. Notation of networks as before.

with the second layer and becomes more vague with increasing batch size. The hidden size does not seem to influence the epoch time as the graphs of b-LSTM-1 (200) and b-LSTM-1 (400) show.

The figure also shows that the time needed for one epoch decreases up to batch size 20 and stays around the same level after that. Further analysis of this observation has shown that the epoch time depends on three factors: the time needed to pass one batch through the network (including the calculation of loss and derivatives and the update of weights), the number of batches in the training set and the time to pass the validation set.

The validation time is almost constant for each test run. It decreases from 3s with batch size 5 to 1.9s with batch size 15, but does not sink any further; it remains around 2s. The time needed to pass one batch through the network does not increase linearly with the batch size. For example, with a batch size of 5, the time needed is 0.011s on average, the time needed for a batch size of 10 is 0.018s. While the number of batches halves, the time needed for one batch does not double. This leads to a decrease in average epoch duration in the interval between 5 and 20.

The larger the batch size, the larger the increase in epoch duration becomes. Comparing batch size 15 to 30, the average epoch time increases by the factor 1.96. This means, as the number of batches halved, the average time needed for one batch nearly doubled. Due to this development, the average epoch time remains around the same level for larger batch sizes.

A batch size of 15 appears to be the best trade-off between more training iterations on average and reduced training time per epoch. Additionally, up to this point in time, for none of the test runs with a batch size of 15, the F-Score differed by more than 1 from the calculated mean F-Score.

5.2.2.3 Experiments with different Hidden Sizes and Number of Layers

Topological parameters such as number and size of layers generally determine the neural networks ability to learn a task. The necessary minimum size depends on the complexity of the task the network is supposed to learn. The more complex a problem is to learn, i.e. the harder the classes are to separate from each other, the more units and layers are necessary. However, a too large topology contains the risk to overfit the network to the training data and achieve bad generalization as a result.

In this experiment stability of the results is considered as an additional criterion. As it has been seen in the experiments with varying batch size, the results of two runs may differ notably. In this context, stable results mean that the F-Score differs by less than one from the calculated mean F-Score (when the same network topology is used). The stability is measured by the average absolute deviation of the results.

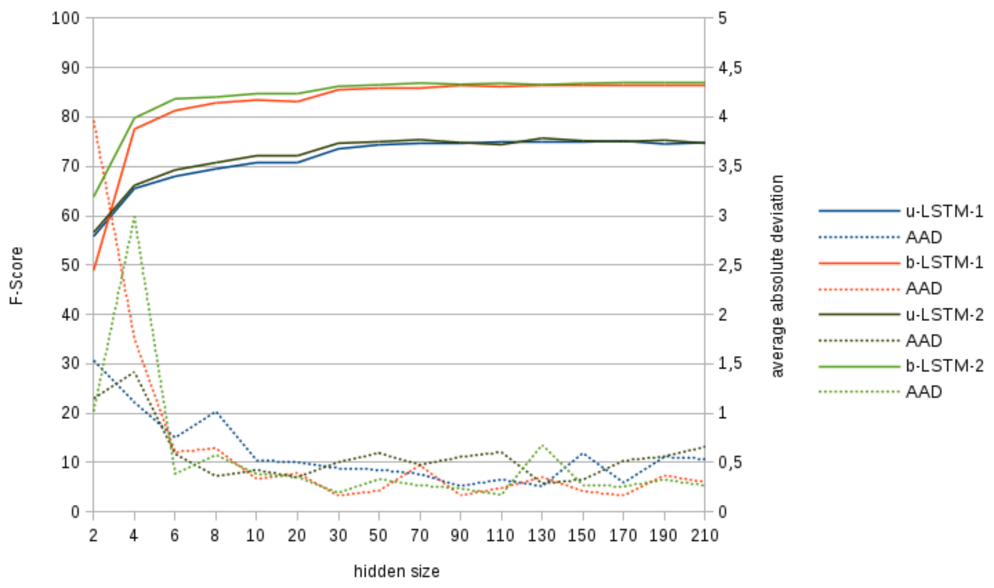


Figure 5.4: F-Score of networks using word embedding plotted over the selected hidden size. Notation of networks as before. (*Remark*: for better visibility, the scaling of the x-axis is *not* linear. The interval [2,20] is spread out.)

For better visibility, the results seen in figures 5.4 and 5.5 are separated by the type of embedding used. The dotted lines in the graph show the average absolute deviation of test runs. The smaller this value, the less individual test runs differ from the mean F-Score and in turn, the stabler the results.

Considering figure 5.4, a few characteristics can be noted. First, The performance rises quickly for hidden sizes between 2 and 8. This increase is stronger for the bidirectional networks. For hidden sizes larger than 8, the increase in performance is slower, but steady. Second, the minimum hidden size, i.e. the hidden size necessary to achieve approximately the best results, appears to be 30. For sizes above 30 the increase, if there is any at all, is barely visible. Furthermore, the difference between unidirectional and bidirectional networks becomes negligible. Third, adding a second layer to the network shows an improvement for hidden sizes smaller than 30. In the interval between 2 and 30, the two-layered networks are constantly better.

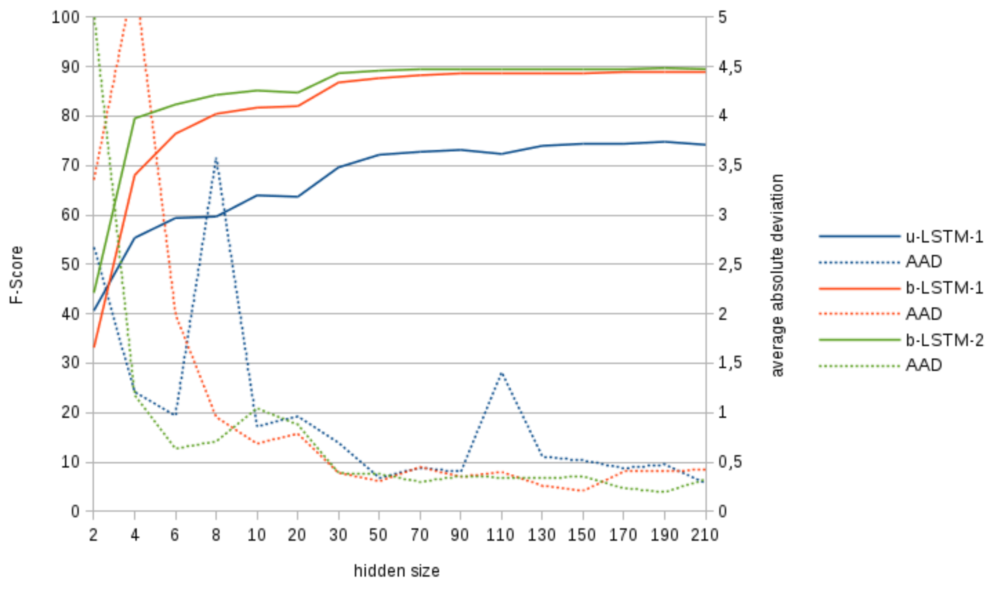


Figure 5.5: F-Score of networks using character embedding plotted over the selected hidden size. (*Remark*: for better visibility, the scaling of the x-axis is *not* linear. The interval [2,20] is spread out.)

Fourth, the bidirectional networks are slightly stabler than the unidirectional ones for hidden sizes larger than 6. The average absolute deviation is slightly smaller for bidirectional networks for most of the hidden sizes. And last, in general, the stability of the results increases with growing topology, as the decrease in average absolute deviation shows. For hidden sizes above 10, the average absolute deviation slightly varies, but stays around the same level.

On the other hand, considering figure 5.5, when character embedding is used the F-Score starts at a significantly lower level. Though the increase of the F-Score for hidden sizes between 2 and 6 is stronger than with word embedding, the score of networks using character embedding rises slower. Here the minimum hidden size is around 130. As with word embedding, the addition of a second layer improves the score for hidden sizes below the minimum hidden size. For character embedding, this increase is even stronger than with word embedding. Especially for hidden sizes below 30, the difference is notable. Compared to word embedding, here, the average absolute deviation is significantly larger for small hidden sizes, decreases slower and shows much higher spikes in the interval [2,30]. This means, the classification with very small networks using character embedding is highly unstable. For larger networks, the stability increases relatively steadily. Especially the bidirectional networks show fewer spikes for hidden sizes between 30 and 210 than when word embedding was used. Comparing the bidirectional networks, the second layer appears to increase stability in the mentioned interval.

Overall, the average absolute deviation remains around the same level for character and word embedding. This means, word and character embedding are about equal in terms of stability, however, the character embedding obtains slightly better results once the minimum hidden size is passed.

In terms of training time, word embedding is by far faster. Epochs take about twice as long when character embedding is used. Adding a second layer generally adds a relatively constant time period to the epoch duration. For word embedding, this time period is about 7s, for character embedding about 10s. The same development goes for making the LSTM bidirectional. Here the increase is about 3s for word embedding and 8s for character embedding. This makes word embedding significantly faster, even though the average number of epochs is slightly higher with this embedding.

5.2.3 Classification Capabilities

The classification performance is measured by precision, recall and the F-Score. Sequences, i.e. sentences, are passed through the network and terms are marked using the BILOU encoding. The results are shown for various topologies using all types of embedding and grades of prepocessing. For each experiment only the few most notable examples are shown.

5.2.3.1 Using Word Embedding

To ensure comparability, the topology and network parameters were kept the same as far as possible. For all experiments, the learning rate was set to 0.001 and the batch size to 5. Preprocessing of the text was applied as written in section 4.6.2. The hidden size for the LSTMs and GRUs was set to 200 and both unidirectional and bidirectional networks were tested. The hidden size was set larger than necessary as run time was not a critical factor in this experiment and it was intended to compute an average over the best possible results.

For the softmax layers the input size was set to *window size* $\cdot$ 300, where 300 is the size of the word embedding defined by the used GloVe vectors. To allow windows as input to these softmax layers, x (with x as the window size) embedding vectors were concatenated. In order to also consider the first and last elements, the sequence must be padded with additional SOS and EOS tokens, as the window stretches beyond the boundaries sequence when it is put on the edge elements of the sequence. For the window size 3, one token at each edge suffices, for the window size 5, two are necessary at each edge. This leads to an input matrix of size *sequence length* $\times$ *window size* $\cdot$ 300.

	F-Score	precision	recall
baseline GENIA	15.99	20.27	13.28
softmax-1	51.78	49.67	54.08
softmax-3	78.56	77.72	79.41
softmax-5	79.87	79.67	80.08
u-LSTM-1 (200)	75.47	75.81	75.13
b-LSTM-1 (200)	86.38	87.06	85.72
b-LSTM-2 (200)	86.73	87.03	86.18
u-GRU-1 (200)	74.70	74.65	74.75
b-GRU-1 (200)	86.17	86.71	85.64
b-GRU-2 (200)	**86.89**	**87.27**	**86.51**

Table 5.1: Results of classification with word embedding. linear-n indicates one softmax layer with window size n. In x-LSTM/GRU-n, x defines direction, n accounts for the number of layers. The number in brackets states the hidden size.

The results of the stated topologies with word embedding as embedding layer are shown in Table 5.1. In this table, two conspicuous observations can be made: First, the inclusion of surrounding words distinctly improves performance. This is shown by a large increase of the F-Score between softmax-1 and softmax-3. Increasing the window size further to 5 improves the F-Score only a little.

Second, the use of bidirectional recurrent neural networks provides a great advantage compared to unidirectional ones. This becomes clear when comparing the results of u-LSTM-1 (200) with b-LSTM-1 (200) and u-GRU-1 (200) with b-GRU-1 (200). In both cases, there is a significant increase of the F-Score.

These observations prove that words prior and subsequent to the considered word are essential for its correct classification. Furthermore, it shows that the memory of recurrent neural networks is superior to only considering a few surrounding elements to the current input. Comparing b-LSTM-1 (200) and b-GRU-1 (200) to softmax-3 and softmax-5, the improvement is obvious. This is because, LSTMs and GRUs are capable of learning long-term dependencies, which cannot be included in the classification using a sliding window as it was used for the softmax networks.

GRUs require fewer weights to be trained, which in some cases improves the results. From the given scores this expectation cannot be confirmed completely. For a single-layer recurrent network, the results for unidirectional and bidirectional LSTMs are slightly above the results of the respective GRUs. However, the two-layered GRU performs slightly better than the respective LSTM. There is no consistent improvement recognizable.

This observation corresponds with the observations on the influence of randomness on the F-Score in sections 5.2.2.2 and 5.2.2.3. Here again, the result of a test run also depends on 'luck' during the random selection of sets and batches. It is very well possible that this small difference in scores between the LSTMs and GRUs may be reversed after a few more test runs.

5.2.3.2 Using Character Embedding

For the networks using character embedding, the learning rate and batch size were set to the same values as with word embedding. As GRUs did not provide notable difference of the scores, they are not included in the list here. The hidden size was set to 400 for all topologies.

	end-to-end			pretrained		
	F-Score	Precision	Recall	F-Score	Precision	Recall
u-LSTM-1 (400)	73.48	73.40	73.56	74.15	73.14	75.18
u-LSTM-2 (400)	**75.07**	**74.24**	**75.91**	74.93	74.07	75.82
b-LSTM-1 (400)	88.97	89.19	88.75	88.98	89.04	88.92
b-LSTM-2 (400)	89.56	89.84	**89.29**	**89.78**	**90.30**	89.27

Table 5.2: F-Scores of different topologies with end-to-end and pretrained character embeddings. No text preprocessing is applied to the sentences. Notation of networks as before.

The results shown in table 5.2 contrast the use of an independently trained and an end-to-end trained character embedding. The pretrained character embedding was trained as part of a 'character language' model, which intends to predict the next most likely character of an input sequence, on a small corpus of English news articles.

From the table, a few things can be noted. First, both the pretrained and the end-to-end trained embedding are almost even in their performance. The scores achieved differ only very little between the two types of training. Second, the pretrained embedding is slightly better when bidirectional networks are used. However, this difference is negligible and may be reversed with more test runs. Third, the use of bidirectional networks instead of unidirectional ones provides a significant increase in score. This increase is even greater than with word embedding. Fourth, for unidirectional networks, recall is generally higher, for bidirectional, precision is higher. And last, adding a second layer slightly, but consistently improves the scores.

From the first observation, it can be concluded that the end-to-end trained and the pretrained embedding are equal in their ability to represent the data in a beneficial way. As there is no definite superiority of either of these visible, the network appears to train a (for this purpose) fitting representation of the characters by itself. For further evaluation in this matter, a character embedding trained on a larger corpus is required.

5.2.3.3 Comparing Word and Character Embedding

Both word and character embedding have shown, that context information is essential for the correct classification of terms. With both kinds of embedding, the score increased significantly when bidirectional networks were used.

For the comparison between character and word embedding, only the LSTM networks are used. The results have shown that word embedding is slightly better when unidirectional networks are considered. With bidirectional networks, the character embedding achieves higher scores. The difference between word and character embedding is larger than the difference when unidirectional networks were used.

5.2.3.4 Impact of Preprocessing

As mentioned in section 4.6.2, different grades of preprocessing are possible. All grades are tested with both kinds of character embeddings, which where introduced above. For both kinds of training, two layers with a hidden size of 400 were selected. The batch size was set to 5 and the learning rate to 0.001.

The results of this experiment, which is stated in table 5.3, sorts the results by preprocessing grade. To objectively compare the performance, all scores were calculated over the word list (refer to section 4.6.2). Thus, the enlarged number of characters (due to not removing punctuation) does not influence the label balance of the text.

The previous observations on the quality of both kinds of character embedding is confirmed by these results. The end-to-end trained character embedding achieves slightly better scores for 'lowercase and no punctuation', but slightly worse scores on 'lowercase' and 'no preprocessing'. A greater difference between the embeddings only shows itself when considering the dispersion of the results. The average absolute deviation listed in this table shows that the results for the pretrained character embedding vary more. Due to this, it is possible that more iterations are necessary to achieve the optimal results when this kind of embedding is used.

	end-to-end			pretrained		
	F-Score	Precision	Recall	F-Score	Precision	Recall
no preprocessing	89.22	89.69	88.75	**89.77**	**90.30**	**89.25**
	0.231	0.454	0.420	0.380	0.355	0.737
lowercase	88.89	89.20	88.59	88.92	89.14	88.70
	0.195	0.4425	0.284	0.309	0.577	0.308
lowercase and no punctuation	87.11	87.45	86.78	85.55	86.37	84.40
	0.236	0.375	0.312	0.576	1.160	0.428

Table 5.3: Results for different grades of preprocessing applied to the input sequence. For all tests, the same topology is used. The small numbers state the average absolute deviation for the characteristic above it.

Comparing the first and third row of the table, it shows that the removal of capital characters worsens the score only slightly. However, the dispersion of the results sinks, which makes the over all results stabler. Adding the fifth row to this comparison, the decrease of the F-Score is clearer. Additionally, the dispersion increases again. These results indicate that removing punctuation does not improve performance. Labeling sentences directly without any kind of preprocessing applied to them yields the best results.

From these observations, it is possible to conclude that capital letters are an important feature for the classification of terminology. Yet, it is not possible to prove whether the punctuation itself provides useful content-related information, which leads to the better scores of 'no preprocessing', or whether this is a result of the slightly enlarged amount of training data.

5.2.4 Out-of-domain corpus

All previous results were calculated on the GENIA corpus, which is a corpus of bio-medical abstracts. In computational linguistics, fields are also called *domains*. This makes all terminology related to the bio-medical field (/domain) *in-domain* and all other existent terminology *out-of-domain*.

Considering this property, the ACL corpus, which contains only abstracts of an entirely different field, can be used as an *out-of-domain* corpus. Observing the performance of the neural network on a different domain, on which it has not been trained, can show its generalization ability and its applicability as a general approach.

Batch size and learning rate were set to the same values as in the previous experiments. As for topology of the neural network, a few different configurations have been tested, with both character and word embedding, to see their influence on the F-Score achieved on the out-of-domain corpus. For character embedding, unidirectional and bidirectional with one and two layers, each with a hidden size of 400, have been selected. Since pretrained and end-to-end trained character embedding differ very little, only results with pretrained character embedding are listed for this experiment. No preprocessing was applied to the text, as this returned the best results in prior experiments (see section 5.2.3.4). For word embedding, a unidirectional and a bidirectional with one layer and a two-layered bidirectional, all with a hidden size of 200, were chosen. The results are seen in table 5.4 below.

	F-Score	precision	recall
baseline ACL	20.71	22.27	19.35
char u-LSTM-1 (400)	36.55	35.04	38.19
char u-LSTM-2 (400)	36.69	34.86	38.72
char b-LSTM-1 (400)	47.00	43.09	51.68
char b-LSTM-2 (400)	**48.00**	**44.35**	**52.31**
word u-LSTM-1 (200)	39.59	32.87	49.75
word b-LSTM-1 (200)	44.11	40.46	48.48
word b-LSTM-2 (200)	45.66	41.79	50.33

Table 5.4: Comparison of scores on the out-of-domain corpus for different topologies. char/word denotes the embedding used. Notation of networks as before.

The baseline of the ACL corpus shows larger values for both precision and recall. This means that compared to the GENIA corpus, the label balance is shifted towards class `o`. The number of samples of this class has increased while the sum of samples over all other classes has decreased.

Recall is higher than precision for all listed topologies. Considering the formulas for precision (refer to equation 5.1) and recall (refer to equation 5.2), this leads to the conclusion that the network generates more false positives than false negatives. Considering the shift of the label balance mentioned above, the network has trained to label more terminology because in the GENIA corpus the amount of terminology labels (`b`, `i`, `l` and `u`) was larger.

Over all, the scores significantly dropped. Especially precision decreased by more than 40 on the bidirectional networks. This decrease is present in both kinds of embedding. Comparing uni- with bidirectional networks, as before, the bidirectional networks achieve significantly better results.

Also, it is shown that the scores achieved by the unidirectional networks using character embedding are worse than those using word embedding. Especially recall is significantly lower for the networks using character embedding. For bidirectional networks, the situation is reversed. Both char b-LSTM-1 and char b-LSTM-2 achieve higher scores than word b-LSTM-1 and word b-LSTM-2. Adding a second layer to the network shows a larger increase of all scores (except for recall of char u-LSTM-2) than it did on the GENIA corpus. The best results are achieved with char b-LSTM-2.

These results suggest, that the network, in a way, adapts to the frequency of occurrences of terms and to certain properties in the sentence or word structure, which differ for bio-medical and computational linguistic diction.

5.2.5 Dropout

Dropout layers are a technique of regularization. The layers set random units to zero based on the set probability. It is a used to prevent overfitting by reducing the chance to create co-dependencies between certain units [HSKS$^+$12]. Here, the dropout is applied to the output of the LSTM [PGCC18]. Several dropout probabilities were applied to different networks using all three kinds of embedding. The test runs are compared in F-Score, loss and run time.

	GENIA F-Score	GENIA loss	ACL F-Score	ACL loss	epoch	epoch duration
	dropout 0.0					
char-p LSTM-2 (200)	89.07	0.25	47.39	0.67	8.38	55.10
	1.218	0.022	2.326	0.025		1.046
word LSTM-2 (100)	86.93	0.23	44.57	0.52	9.74	22.97
	0.307	0.005	1.624	0.022		0.267
char-e LSTM-2 (200)	89.53	0.24	46.18	0.70	8.50	54.07
	0.411	0.007	1.702	0.027		0.232
	dropout 0.2					
char-p LSTM-2 (200)	89.51	0.24	**48.74**	0.65	8.38	68.94
	0.369	0.007	0.875	0.033		2.041
word LSTM-2 (100)	**87.07**	0.22	45.47	0.48	11.25	27.61
	0.308	0.006	1.013	0.011		0.369
char-e LSTM-2 (200)	**89.78**	0.24	**48.23**	0.68	9.26	67.20
	0.416	0.006	0.777	0.036		0.441
	dropout 0.4					
char-p LSTM-2 (200)	**89.72**	0.23	48.30	0.67	9.63	73.90
	0.252	0.004	0.652	0.029		1.919
word LSTM-2 (100)	87.02	0.22	**46.35**	0.47	12.38	26.72
	0.203	0.005	0.680	0.011		0.272
char-e LSTM-2 (200)	89.69	0.23	47.77	0.67	9.88	67.04
	0.306	0.007	1.523	0.037		0.325

Table 5.5: Application of different dropout probabilities. The letter after char marks the type of training used for the embedding. Notation of networks as before.

The first segment in table 5.5 is the average of test runs with the same topology but without dropout. These numbers are treated as the baseline in order to analyze in what way the achieved scores are altered through the use of a dropout layer.

The results in the table show that the effects of dropout on the data are very little but consistent. Considering the scores on the GENIA corpus, a slight increase for both char-e LSTM-2 and word LSTM-2 can be noted with a dropout rate of 0.2. If the rate is increased further, the scores are still better than the baseline, however worse than before. For char-p LSTM-2, the scores are further increased with dropout 0.4. The average absolute deviation of word and end-to-end trained character embedding increases very little for dropout 0.2 and sinks with the larger dropout. With the pretrained character embedding, a strong decrease of the average absolute deviation can be noted for both dropout values. This means, the dispersion sinks in general. The effects on the loss are negligible.

The improvement achieved by adding dropout is more notable when considering the ACL F-Score. For word embedding, the best results are achieved with dropout 0.4. Additionally, the average absolute deviation drops to less than half of the initial value. This reduced dispersion partially causes the increased score. The closer the scores are, the less the influence of outliers on the average. For both character embeddings, the best scores are achieved with dropout 0.2. These are even better than the ones from the previous section, which were calculated with nets double in size. Again, the dispersion of results has decreased to less than half. For the pretrained embedding it drops to nearly one third of the initial value. Again, the effects on the loss are only very little.

For all embeddings, both average epoch duration and average number of epochs rises with increasing dropout. At each iteration, the network, or to be more precise, the weights, which are trained with SGD, differ from the ones before, as some units have been zeroed. This makes the updating of weights more noisy [HSKS+12].

The results have shown that dropout has a slight positive influence on the maximum F-Score. Furthermore, it distinctly increases the stability of the results. Especially on the out-of-domain corpus, the stability has shown great improvement.

As only one topology for each embedding has been tested, it is within the scope of possibilities to further increase the results by exhaustively testing more topologies. This is not done here, as only the general effects of dropout were analyzed.

5.2.6 Training on ACL corpus

The ACL corpus is a relatively small corpus and has only sparse annotation. Since it differs greatly from the GENIA corpus, topological parameters may have other effects on the resulting scores. For that matter, a variety of different topologies was trained. This is to see, what parameters influence the F-Score, positively or negatively. The results of several different bidirectional networks, trained and tested on the ACL corpus and additionally tested on the GENIA corpus as out-of-domain corpus, are given below in table 5.6. The presented results are an extract and show only a few of the best results.

It is obvious that the results are significantly worse than for networks trained on the GENIA corpus. Additionally, the average absolute deviation of the F-Scores on both the ACL and the GENIA corpus are very large compared to deviations achieved with networks trained on the GENIA corpus. Here it is between 2.0 and 6.5 for the ACL F-Score and between 1.5 and 4.5 for the GENIA F-Score although it is only a *mean* value. In contrast to this, in previous experiments the average absolute deviation was around 0.3 for most topologies. This means that individual test results vary significantly from the mean value calculated for the respective topology. The results for the GENIA corpus vary slightly less, but still, are considerably more unstable than previous results on the OOD corpus. Due to this considerable increase, it is impossible to determine a single best architecture. However, there are a few interesting developments, which can be observed in this table.

In terms of run time, prior observations are reflected again here. Word embedding generally takes a few more iterations to train, however a single epoch is usually shorter. Adding a second layer appears to add a relatively constant amount of time. Passing one single epoch takes a considerably shorter time period than before, which is due to the difference in size between the two corpora. The training set of the GENIA corpus is more than 8 times larger.

The hidden size has great influence on the calculated F-Score for both the ACL and the GENIA corpus. For both the end-to-end trained character embedding and word embedding, a minimum hidden size can be determined. For word embedding, the minimum hidden size is 40, for end-to-end trained character embedding 60. However, once this hidden size is passed, the scores vary below the top score in a wider range

	ACL F-Score	ACL loss	GENIA F-Score	GENIA loss	epochs	epoch duration
char-p LSTM-1 (200)	56.49	0.43	**42.54**	1.09	10.88	5.14
	2.992	0.036	2.431	0.059		0.544
char-p LSTM-2 (80)	58.38	0.39	39.38	1.20	10.77	7.26
	4.525	0.027	2.568	0.066		0.828
word LSTM-1 (200)	61.15	0.27	34.69	0.97	13.89	3.20
	2.358	0.019	2.551	0.046		0.329
word LSTM-2 (40)	60.22	0.29	36.62	1.04	15.15	4.02
	3.353	0.024	3.389	0.071		0.203
char-e LSTM-1 (80)	60.61	0.39	38.25	1.23	11.00	6.73
	5.743	0.042	**3.486**	0.080		0.715
char-e LSTM-2 (70)	**63.33**	0.37	41.34	1.24	10.77	8.07
	2.811	0.019	2.683	0.091		0.539
	dropout 0.2					
char-p LSTM-2 (40)	57.67	0.38	41.28	1.13	11.74	7.05
	4.315	0.023	2.247	0.052		0.740
word LSTM-2 (50)	59.78	0.29	34.56	1.06	15.25	3.34
	2.479	0.013	1.983	0.058		0.188
char-e LSTM-2 (80)	61.71	0.36	40.66	1.23	11.50	7.24
	2.167	0.027	2.158	0.072		0.796

Table 5.6: A few of the best test runs trained on the ACL corpus. The GENIA corpus is used as out-of-domain corpus. Notation of networks as before.

than seen with networks trained on the GENIA corpus. Selecting hidden size larger than the minimum has no positive effect on the stability of the results either. The average absolute deviation remains more or less random in the same interval.

For the pretrained character embedding, the results are different. The hidden size appears to have no enhancing or stabilizing effect at all. The results vary in a range from 48 to 58 almost entirely independent from the hidden size.

The number of layers has a positive influence on the performance on the ACL corpus. Especially when the end-to-end trained character embedding, the increase is notable. On the GENIA corpus, the improvement of the F-Score by adding a second layer is even larger. However, it has a negative influence on the stability of the results, which is shown by the stronger fluctuations of the average absolute deviation. The only exception for this are the scores on the GENIA corpus with networks using end-to-end trained character embedding. In this case the results become slightly stabler.

Compared to prior results, where the kind of training used for character embedding did not matter, here, the difference between the pretrained and the end-to-end trained character embedding has increased distinctly. In almost every tested topology, end-to-end trained character embedding performs significantly better on the ACL corpus. This shows, that the network can learn a more fitting representation of characters for this task. When tested on the out-of-domain corpus, this difference is not confirmed. Networks of both kinds vary around the same level. Also, the best result on the OOD corpus is achieved by the pretrained character embedding.

These observations lead to the assumption that the requirements for the character embedding differ greatly for both corpora. The characteristics of the pretrained embedding fit the requirements for the GENIA corpus better than those of the ACL corpus.

Reviewing the stability over all tested topologies, the two-layered network with pretrained character embedding closely followed by the networks using end-to-end trained character embedding are the worst. The most 'stable' networks are the ones using word embedding. On the GENIA corpus, the single layered word embedding network is the most stable. The scores of the other networks more or less vary around the same level.

The effects of dropout are not as notable as in section 5.2.5. Considering the ACL F-Score, none of the networks shows improvement. The scores are in the upper half of the range of achieved scores, however none of them is near the best possible one. In terms of stability, slight improvements can be noted, if compared to the other two-layered networks. On the GENIA corpus, the alterations of scores is very similar to the stated observations on the ACL corpus. Again, none of the networks shows top performance, however the stability increases. The average epoch duration does not increase compared to the networks without dropout. Over all, the training time remains the same.

As GRUs require fewer weights, the results may be better for smaller training sets [CGCB14]. For that matter, test runs with the same learning rate, batch size and hidden sizes were conducted using GRUs instead of LSTMs. The best results are shown in table 5.7

	ACL F-Score	ACL loss	GENIA F-Score	GENIA loss	epochs	epoch duration
char-p GRU-1 (150)	51.35	0.44	35.98	1.19	9.13	5.87
	7.632	0.057	**4.114**	0.075		0.992
word GRU-1 (50)	**69.22**	0.27	39.38	1.36	6.63	5.94
	3.008	0.031	3.627	0.165		0.590
char-e GRU-1 (100)	65.71	0.37	**40.83**	1.28	9.63	5.41
	2.771	0.040	1.579	1.219		0.422

Table 5.7: The best scores using GRUs on the ACL corpus. Notation as before.

The hidden size has similar effects on the results as with LSTMs. However, the minimum hidden size is slightly smaller than before. Using hidden sizes larger than the minimum hidden size, the scores remain in the same range and do not increase any further. This range has slightly decreased compared to the LSTMs. For pretrained character embedding, this development could not be observed. Again, the scores vary between 41 and 53 with no perceptible correlation to the hidden size.

In terms of epoch duration, a slight decrease could be noted for the character embeddings. With word embeddings, the time slightly increased. Additionally, the average number of epochs decreased for all embeddings, which decreases the over all necessary training time.

Comparing these results on the ACL corpus to the results of the LSTM, significant changes could be noted. For the pretrained character embedding, both the F-Score and stability sank. Especially the average absolute deviation more than doubled, which indicates large dispersion of the results. However, for end-to-end trained character embedding and word embedding, the results improved. Word embedding in particular showed distinct enhancement. Here, the F-Score increased by 8 and the stability slightly decreased. For end-to-end character embedding, the improvement was smaller, but the stability increased. The loss did not change for any of these topologies.

On the GENIA corpus, the scores sank for both character embeddings. This decrease is especially notable for pretrained character embedding. Additionally, the dispersion of scores increased. With end-to-end trained character embedding, the F-Score decreased only slightly, but the stability increased. When word embedding was used, the F-Score increased, however the stability became worse.

The results show, that GRUs provide no improvement at all for pretrained character embedding. The results are more unstable and lead to worse scores than before. For the other two embeddings, improvements could be noted. Especially networks using word embedding achieve higher scores on both corpora when GRUs are used. Additionally, the training time decreased.

It is difficult to conclude the cause(s) of these results from the stated observations. One possible factor is the size of the ACL corpuse. With only 1384 sentences, it is significantly smaller than the GENIA corpus. Additionally, percentagewise, there are fewer terms annotated in this corpus and, as it has been shown in section 4.6, the labels of this corpus are very imbalanced. Another possible influence is the difference in general sentence and word structure. While the GENIA corpus contains a lot of uncommon words, terms in the ACL corpus are often combinations of commonly used words, e.g. 'sentence analysis'.

5.2.7 Optimizer Tuning

Compared to the results of networks trained on the GENIA corpus, those trained on the ACL corpus have shown poor performance in terms of both score and stability. For that reason, the optimizer is slightly modified and the resulting effects are analyzed.

5.2.7.1 Weight Decay

Weight decay is a method of regularization imposed on the weights of the network. During the calculation of the loss, an additional term is added, which grows proportionally with the size of the weights of the network. Thus, very large weights are punished.

$$\mathcal{J}(w) = \mathcal{J}_0(w) + \frac{1}{2}\lambda \sum_i w_i^2 \tag{5.5}$$

Equation 5.5 shows the calculation of loss with included weight decay, where $\mathcal{J}_0(w)$ is an arbitrary loss function and λ the weighting factor of the penalty [KrHe92, Mood92].

To test the influence of weight decay on networks trained on the ACL corpus, the best performing topology for each configuration is trained using SGD with an initial

	ACL	ACL	GENIA	GENIA		epoch
	F-Score	loss	F-Score	loss	epochs	duration
	weight decay 0.1					
char-p LSTM-2 (80)	55.42	0.41	38.80	1.20	10.75	7.49
	4.408	0.058	3.527	0.094		0.323
word LSTM-1 (200)	68.41	0.24	38.03	1.02	7.76	5.88
	3.697	0.015	3.209	0.085		1.500
char-e LSTM-2 (70)	59.10	0.39	38.92	1.16	10.01	12.74
	4.886	0.022	4.482	0.057		1.152
	weight decay 0.01					
char-p LSTM-2 (80)	53.88	0.42	39.00	1.15	9.51	7.89
	2.349	0.045	2.636	0.034		0.935
word LSTM-1 (200)	68.68	0.23	**40.79**	1.21	7.38	7.64
	1.022	0.012	4.035	0.138		0.798
char-e LSTM-2 (70)	58.48	0.39	37.49	1.29	9.63	13.21
	4.122	0.022	2.452	0.088		1.468
	weight decay 0.001					
char-p LSTM-2 (80)	51.98	0.41	37.64	1.18	10.76	6.94
	4.950	0.033	2.820	0.059		0.478
word LSTM-1 (200)	**70.73**	0.23	39.84	1.21	7.36	7.62
	4.155	0.015	1.962	0.084		0.779
char-e LSTM-2 (70)	61.86	0.36	38.49	1.34	11.75	12.00
	5.615	0.033	3.697	0.044		1.400

Table 5.8: Best LSTM architectures from previous experiment with different values for weight decay. GENIA corpus is used as OOD corpus. Notation of networks as before.

learning rate of 0.001 for character embedding and 0.01 for word embedding and with different weight decay factors. The results are given below.

Using weight decay alters the scores distinctly as shown in table 5.8. Observing the developments when character embedding is used, it can be noted that the over all performance has decreased. The F-Score on both the ACL and the GENIA corpus sank. For most configurations, the loss slightly increased. Also, the training time slightly increased, as both average epoch duration and number of epochs increased.

The two character embeddings show different changes when weight decay is used. While the score of the end-to-end trained embedding generally decreases with growing weight decay, the effects on the pretrained embedding are opposite. On the GENIA corpus, the changes in F-Score are converse for these embeddings. Comparing weight decay 0.1 with 0.01, the results for pretrained embedding increase, yet the scores of end-to-end trained embedding decrease. Comparing 0.01 with 0.001, the scores of pretrained embedding decrease and those of end-to-end trained embedding increase.

Contrary to the general results of character embedding, the effects on word embedding are very positive. Some of these results even surpassed the best noted GRU scores from the previous section. The F-Score on the ACL corpus increased by about 9. On the GENIA corpus, the F-Score increased by 5-6. However, the stability of the results mostly decreased. Considering the training time, the average number of epochs sank while the training time for one epoch increased. So, the training time changed very little.

All presented results are sensitive to the chosen learning rate. With character embedding, the expected behaviour eventuates: the smaller the learning rate, the slower the training, but the better the results. However, with word embedding, opposite behaviour could be observed. Test runs with a learning rate of 0.001 took longer to train, as it was expected, but the results were by far worse than with a learning rate of 0.01. Up to this moment, no explanation has been found.

The scores have shown very contrastive changes for the different kinds of embedding when weight decay was used. Comparing the topology of the tested networks, only two major differences stand out: the two-layered architecture for character embedding and the higher input dimensionality of word embedding. In order to determine, which of these characteristics has more influence, additional tests were conducted: word embedding was tested with a two-layered architecture and character embedding with a single-layered one. The results have shown no distinct changes in the calculated scores. HConcluding from this, it is more likely that the input dimensionality, or more specifically, the increased sparsity of the larger input when word embedding is used, has greater influence than the topology of the network.

When weight decay is applied to networks with GRUs instead of LSTMs, slightly different developments can be noted. For these test runs, the same setup as with LSTMs was used: the previous best network topologies were trained with weight decays selected from the range 0.0001 to 0.1. Pretrained character embedding is not listed below, as the previous results showed poor performance with GRUs. The best results are listed in table 5.9.

	ACL F-Score	ACL loss	GENIA F-Score	GENIA loss	epochs	epoch duration
	weight decay 0.0001					
char-e LSTM-1 (100)	64.57	0.37	**39.94**	1.28	10.38	5.46
	3.579	0.0326	1.639	0.086		0.543
word LSTM-1 (50)	63.74	0.27	37.34	1.16	12.99	3.82
	4.072	0.033	2.947	0.087		0.550
	weight decay 0.01					
char-e LSTM-1 (100)	60.93	0.38	39.81	1.26	9.11	5.93
	2.276	0.053	3.067	0.082		0.492
word LSTM-1 (50)	**66.42**	0.25	36.02	1.17	13.25	4.14
	5.269	0.017	2.543	0.069		0.572

Table 5.9: The few best results using GRUs with weight decay.

Compared to the results without weight decay, the scores of networks using end-to-end trained character embedding have slightly decreased on both corpora. Also, the stability of the results has slightly worsened for test runs on the ACL corpus, but slightly improved for test runs on the GENIA corpus. Considering run time, no major changes were noted.

The over all decrease in score is smaller with GRUs than it is with LSTMs. Furthermore, GRUs appear to be more sensitive to the chosen weight decay than LSTMs.

This is indicated by two observations: first, the decrease of scores on the ACL corpus is stronger with increasing weight decay and second, the best score was achieved with a smaller weight decay.

For word embedding, the scores on both corpora also notably decreased compared to results without weight decay. This development is opposed to the development with LSTMs. However, similarly to the LSTM results, the stability sank. The training takes slightly longer than without weight decay, as the average number of epochs increased while the epoch duration decreased only very little. This increased training time is still less than the LSTM takes to train.

Contrary to character embedding, the results are less sensitive to the chosen weight decay. The best scores are achieved with weight decay set to 0.01. Also, the scores with other weight decays differ less from the best score.

5.2.7.2 Adagrad

Adagrad is another gradient-based optimizer like SGD, which adapts the learning rate based on the frequency of features. The updates for rare features are larger, which makes it a good choice for sparse data [DuHS11].

Learning rate, learning rate decay and weight decay are the parameters of this optimizer, which can be adjusted in the test runs. The learning rate decay changes the learning rate at each iteration. So this parameter replaces the scheduler. Weight decay is the same as in the previous section.

With this setup, five parameters can influence the performance of the network: number of layers, hidden size, learning rate, weight decay and learning rate decay. The learning rate is handled by the Adagrad algorithm, so it can be set to 0.01 and not changed in the test runs. As this section deals with the Adagrad optimizer, the topological parameters (number of layers and hidden size) are left static as well. The results from the same topology using SGD are used as baseline.

Word embedding and character embedding are used with a one-layered LSTM with 300 hidden units. Both weight decay and learning rate decay were selected from $[0.0, 0.0001, 0.001, 0.01]$ and every possible combination was tested. A few of the best results are listed below. Since pretrained and end-to-end trained character embedding show the same developments, one is omitted in the table showing the results. Furthermore, the use of GRUs has shown only negligibly small differences in the scores and is also omitted from the table here.

Comparing the influence of learning rate decay and weight decay, the impact of the former is more visible. For learning rate decay values between 0.0 and 0.001, the F-Scores remain around the same level. With 0.01, the scores sink to a lower level. This is visible for all kinds of embedding. The influence of weight decay is more ambiguous. There is no clear development visible, neither in loss nor in F-Score nor in any of the other measured characteristics. Even in the average absolute deviation, there is no upward or downward trend notable. The scores remain in the same range and slightly sink or rise with no correlation to weight decay visible.

Considering the results from character embedding, a few changes compared to the previous tests with weight decay are visible. First of all, the scores decreased distinctly compared to the last tests, for some runs by more than 7. The average

	ACL F-Score	ACL loss	GENIA F-Score	GENIA loss	epochs	epoch duration
char-e baseline	60.09	0.40	**40.34**	1.19	11.01	5.27
word baseline	58.86	0.29	34.72	0.97	13.63	3.82
	learning rate decay 0.0 weight decay 0.0001					
char-e LSTM-1 (300)	57.30	0.39	38.33	1.18	6.63	7.47
	5.586	0.040	3.883	0.038		1.824
word LSTM-1 (300)	70.16	0.23	37.30	1.16	4.75	11.09
	1.960	0.008	2.055	0.057		0.721
	learning rate decay 0.0001 weight decay 0.01					
char-e LSTM-1 (300)	56.60	0.41	39.04	6.75	7.20	
	5.611	0.030	3.139	0.093		1.21
word LSTM-1 (300)	**70.83**	0.23	38.88	1.14	4.25	12.04
	1.274	0.019	2.626	0.093		0.802
	learning rate decay 0.001 weight decay 0.001					
char-e LSTM-1 (300)	54.63	0.43	39.24	1.11	7.13	6.87
	3.10	0.030	1.010	0.064		0.998
word LSTM-1 (300)	70.13	0.23	38.83	1.07	5.02	11.93
	1.892	0.009	2.747	0.077		1.132

Table 5.10: Results of end-to-end trained character embedding and word embedding with different Adagrad parameters. Notation of networks as before.

absolute deviation remained in the same range. Second, the results are even more sensitive to the learning rate decay than word embedding. Setting this parameter to 0.0001 already results in a slight decreases of the scores. For higher values the drop becomes more obvious. And last, the training became significantly faster, as both the average number of epochs and the average epoch duration decreased.

For word embedding, the results are quite similar to the last test run. As before, a significant increase of the scores on the ACL corpus is shown. Over all, the scores are around the best score of the previous experiment, however the average absolute deviation has significantly decreased. The average number of epochs for training has decreased as well. But since the average duration has increased, the total time for one test run has changed only negligibly little.

To further examine the different results of word and character embedding, the embedding size of end-to-end character embedding was increased to 200. Comparing these results against the ones with smaller embedding size, a significant increase of the F-Score becomes apparent. For some parameter configurations, the score rises by nearly 10, which leads to better results than those with the usual embedding size and without any modifications (shown in table 5.6). The scores on the out-of-domain corpus increased as well, some of them even surpassing all previously achieved results with end-to-end character embedding. This shows that the sparsity of the input affects the performance of the Adagrad learning algorithm.

Still, these scores are worse than the best with word embedding. The reason for this cannot be determined from the given test runs alone. A possible influencing factor is the amount of data used to calculate the embedding, which is significantly larger for word embedding.

For the sake of completeness, test were also run with GENIA as the in-domain and ACL as the out-of-domain corpus. The topology and hyperparameters were left the same as stated above. For learning rate decay and weight decay the same combination of values was tested.

For character embedding, the scores on the GENIA corpus slightly decreased from 89 to 87. The same development is observed on the out-of-domain corpus, where the scores decreased from 48 to 46. For both corpora, the dispersion of results has slightly increased. This means, the results are more unstable than with SGD. In terms of training time, a clear improvement was noted. Both the average number of epochs and the average epoch duration decreased.

With word embedding, the scores on both the GENIA and the ACL corpus remained around the same level. Also, the dispersion is similar to the dispersion when SGD is used. However the average number of epochs has halved compared to networks trained with SGD. This greatly improves training time.

For both character and word embedding, the Adagrad training algorithm did not improve the scores if applied to networks using the GENIA corpus as in-domain corpus. With character embedding, the scores even slightly decreased. However, training time could be reduced. For word embedding, Adagrad is a sensible alternative to normal SGD.

6. Error Analysis

The experiments have shown that there is a threshold for the score, which cannot be surpassed by changing the topology or the hyperparameters. For that matter, the remaining errors are viewed in closer detail.

6.1 Confusion Matrix

For the test results of an experiment, the confusion matrix can be computed. Due to the random selection of sentences into the sets, the total number of labels can differ slightly between different test runs even though the same corpus was used. To compare the results, percentages are calculated based on the matrices.

The total number of labels in the test set of the GENIA corpus andthe number of labels in the OOD test set are in a similar range, since the test set contains only 10% of the sample sentences. This results in 1844 sentences for the GENIA corpus and 1384 sentences for the ACL corpus, which is the entire data available in this corpus.

The following matrices were generated with bidirectional two-layered LSTMs with hidden size 200. For stochastic gradient descent the learning rate was set to 0.001. The networks were trained on the GENIA corpus, the ACL corpus was used as out-of-domain corpus.

	o	u	b	i	l
o	29494	214	315	143	245
u	178	2132	63	13	104
b	561	241	4214	167	8
i	249	32	276	3699	142
l	536	126	14	141	4383

	o	u	b	i	l
o	26422	406	981	689	965
u	226	137	44	12	48
b	466	10	471	68	5
i	189	0	40	293	8
l	432	6	3	76	513

Table 6.1: Confusion Matrix calculated for results of test runs with word embeddings for both the GENIA (left) and ACL (right) corpus. The column is the target label, the row the predicted label. The grey numbers are the numbers of correct labels For each class.

Considering the confusion matrix of the GENIA corpus (6.1 left), the best recognized class is outside. With respect to the classes indicating terminology, the best class is unit. inside and last are detected about equally well, begin is detected worst.

The presented results can be transformed into a binary classification scheme, term non-term. In this sense, the first field in the first column is the true negative field. The rest of the first column becomes the false positive. In this case, 1524 words were wrongly classified as terms, which is about 5% of all terms. Adding the numbers of the first row, except for the first field, the number of overseen terminology words, the false negative, is calculated. Here, 917 term words were overseen. Comparing this to the overall 16806 term words in the test set, about 5% were not recognized.

On the other hand, the worse F-Score on the OOD corpus is reflected by the confusion matrix for the ACL corpus (6.1 right). Considering only the terminology classes, the best class is inside with about 55% correctly predicted labels. begin and last are recognized about equally well. unit is detected worst. Only 29% of the predicted unit labels are correct, 48% of the words labeled as unit are actually outside.

Converting the matrix to a binary classification scheme, the false negative rate is about 63% and the false positive rate about 5%. This means, 63% of the terminology in this corpus was overseen. The low false positive rate is due to the prevalence of the outside label.

When only binary classification is considered, *informedness* (with $informedness = TPR + TNR - 1$) gives an additional estimate of the quality of the results. For the GENIA corpus, 90% informedness is achieved, for the ACL corpus 31%. These values reflect the worse performance of the networks on the OOD corpus.

	o	u	b	i	l
o	25770	140	313	132	227
u	140	2296	161	6	55
b	478	44	4237	118	0
i	149	14	196	3555	80
l	458	71	2	78	4430

	o	u	b	i	l
o	23596	394	957	611	927
u	258	147	46	14	61
b	445	5	492	82	1
i	132	0	40	353	6
l	432	13	5	80	545

Table 6.2: Confusion Matrix calculated for results of test runs with character embeddings for both the GENIA (left) and ACL (right) corpus. Notation as before

The improvement of the F-Score, when character embedding is used, becomes clear when calculating the percentages of correct labels. While outside and unit are about the same as before, the percentages for begin, inside and last increased from 79% to 86%, from 84% to 89% and from 84% to 87%. With 86%, the worst recognized class is now unit.

Considering this confusion matrix as binary classification, both kinds of embedding achieve nearly identical false positive and false negative rates.

The confusion matrix calculated on the ACL corpus shows that the correct prediction of unit further decreased from 29% to only 27%. The scores for begin, inside and last increased. The most notable increase was of class inside, which rose from 55% to 66%. So, the worst recognized class still is unit.

The transformation to the binary case shows that the true positive rate has increased from 36% to 39%. Additionally, the false negative rate has decreased to 60%. The informedness for the matrices returns 90% for the GENIA corpus and 34% for the ACL corpus, which reflects the better performance of character embedding shown in previous experiments.

6.2 Correctable Errors

Without knowledge of the target labels, correcting predicted labels is restricted to only very few cases. Considering the BILOU label scheme, only certain patterns may occur. There are certain transitions between labels, which are prohibited by the design of the scheme.

A correct BILOU sequence can be described using a context-free grammar:

$$\begin{aligned} \texttt{S} &\longrightarrow \texttt{W} \\ \texttt{W} &\longrightarrow \texttt{W W | b I l | u | o} \\ \texttt{I} &\longrightarrow \texttt{i I | } \epsilon \end{aligned}$$

The terminals, defined by the labels of BILOU, are written in lower case, variables are written in capital letters. The starting symbol is `S`.

From this it is easy to identify twelve impossible transitions between labels: `o i, o l, u i, u l, i o, i u, i b, l i, l l, b o, b u, b b`. These wrong transitions can be identified without knowledge of the true target labels and possibly corrected. The difficulty lies in the correction itself. Supposing the wrong pattern `o l o` has been identified, several corrections are imaginable: `o u o, b l o, o o o`. These are only a few suggestions. It is not certain, whether the correct label combination is part of these.

The number of wrong transitions greatly differs for each run and depends on the type of embedding, topology of the network, hyperparameters and the randomly selected sets and batches. The mean value and average absolute deviation of the number of wrong transitions over all test runs gives a rough approximation of its general scale. For the GENIA corpus, the mean is 1003.53 with an average absolute deviation of 101.49. For the ACL corpus the mean is 838.93 with average absolute deviation 126.73. The higher dispersion reflects the smaller score and the higher instability on the out-of-domain corpus shown in the test runs in section 5.2.6.

6.3 Postprocessing

Post processing is applied in order to replace such wrong patterns. The ideal replacement is a pattern, which meets two criteria: first, the pattern is correct, this means it can be generated with the stated grammar, and second, it fits the target. The second criterion is unattainable. The closest possible approximation of this characteristic is the use of heuristics to select the most likely replacement.

In general, two approaches have potential:

1. definition of hand-written rules based on statistical analysis of the wrong patterns and their correct replacements

2. using machine learning techniques to learn the best replacement

Both approaches have their advantages. With the former approach, the correctness of patterns can be guaranteed, as only the hand-written replacements are possible. However, it is questionable if these rules actually proof to be the best possible choice as a replacement. Using machine learning cannot guarantee the correctness of the replacement. However, it is most likely faster to implement as no statistical analysis and consideration of rules is necessary.

6.3.1 Considerations

To implement a postprocessing unit based on machine learning techniques, various aspects must be considered. The first aspect deals with the selection of data format and encoding. One possibility is to simply take the identified wrong label. On the other hand, it is possible to use the predicted probabilities of each label generated by the network as additional information for the classifier. In this case, instead of one label index, a vector of five elements will be used for the classifier. Additionally to this, the window size of the input must be defined. The number of surrounding labels (or in case two, probability vectors) may positively influence the classification capabilities.

Also, if the postprocessing is applied to a network using character embedding, it must be consider whether to operate on the word or character list. As both lists must follow the rules of the label scheme, postprocessing is possible for both of them. However, the character list allows that labels `b` and `l` are repeated inside the word. Duplications beyond word borders are not allowed. Furthermore, the labels inside a word should not change. These additional constraints must be implemented in this context.

The next thing to consider is the actual process of replacing the label. Since label pairs are used to identify wrong transitions, replacing both labels is a plausible option. For all transitions, either of the two labels can be the one that needs to be replaced. Without further knowledge of the actual target, it is not possible to tell which of the labels causes the problem. From this point of view, it is more sensible to replace both labels at once. In this context, it is also necessary to decide whether the replacement should be applied to the sequence, which is put into the classifier, as well. On the one hand, it is possible that two wrong patterns, which fall in the same window, contain useful information for each other. On the other hand, the correction of the label may add more helpful information than the original sequence contained in that place.

In order to evaluate the quality of the postprocessing, a baseline must be determined. For the calculation, it is necessary to count the most common replacement for each label. If label pairs are used, the number of possible replacements is 13, since every possible combination of labels except for the prohibited transitions, must be considered. Counting the frequency, which pair replaces which, probabilities for the replacement can be calculated. These are listed in the following matrix:

For most of the wrong transitions, a clear replacement is given. Considering the columns `ui`, `bo`, `io` and `ll`, there are two replacement pairs, which occur about equally often. For the calculation of the baseline, each of the patterns in the columns

	oi	ol	ui	ul	bo	bu	bb	io	iu	ib	li	ll
oo	0.02	0.18	*0.37*	0.02	0.04	0.04	0.03	*0.37*	0.03	0.03	0.24	0.02
ou	0.01	0.01	0.09	0.00	0.01	0.01	0.02	0.01	0.18	0.02	0.00	0.05
ob	0.18	0.30	0.00	0.01	0.00	0.02	0.00	0.01	0.01	0.40	0.01	0.00
uo	0.00	0.00	0.01	0.00	0.00	0.15	0.20	0.11	0.02	0.00	0.03	0.01
uu	0.00	0.00	0.01	0.00	0.01	0.06	0.21	0.00	0.17	0.01	0.00	0.04
ub	0.01	0.00	0.00	0.01	0.00	0.27	0.01	0.00	0.01	0.06	0.00	0.00
bi	0.04	*0.37*	0.00	0.01	0.00	*0.31*	0.01	0.12	0.04	*0.42*	0.01	0.00
bl	0.00	0.01	*0.37*	0.00	0.04	0.03	*0.42*	0.36	*0.50*	0.02	0.03	0.05
ii	*0.43*	0.10	0.00	*0.40*	0.01	0.07	0.00	0.00	0.00	0.02	0.11	0.06
il	0.02	0.00	0.13	0.02	*0.37*	0.00	0.06	0.00	0.02	0.00	*0.36*	*0.38*
lo	0.01	0.00	0.01	0.09	0.14	0.01	0.01	0.00	0.00	0.00	0.18	0.01
lu	0.02	0.00	0.00	0.08	0.35	0.01	0.01	0.00	0.02	0.00	0.01	0.35
lb	0.25	0.01	0.00	0.34	0.02	0.02	0.00	0.00	0.00	0.00	0.02	0.02

Table 6.3: Averaged replacement probabilities rounded to two decimal places for each wrong transition. The first row marks the detected wrong pattern. The first column states the replacing patterns. Field a_{ij} states the probability of pattern a_{0j} being replaced with a_{i0}. The largest probabilities are marked in blue.

will be replaced by the pattern with the highest probability. The baseline is determined by the sum of correct replacement labels over all replaced pairs.

Another consideration concerns the choice of classifier. As no further knowledge on the data (e.g. separability) can be obtained, the choice is more 'trial and error' than actual consideration. With the selection of a specific classifier, parameter optimization is the logical next concern.

For the necessary training data, it must be considered, whether only wrong patterns, which are obtained during training of the neural network, should be considered or whether all kinds of patterns should be used. Using patterns from all training data has the advantage that there will be more training data available. However, at the same time, these additional patterns may bias the postprocessor by adding unrepresentative and irrelevant data.

The last aspect to address concerns the replacing process. For better illustration, consider the extract `... o o b i o o ...`. While examining the sequence, transition `i o` at index 3 was identified. Subsequently, `i o` was replaced with `o o`, which led to a prohibited transition `b o` at index 2. Since index 2 has already passed, this wrong transition will not be found and cannot be corrected. Given this situation, there are two possible courses of action: ignore these cases and continue with the examination, or implement a backward pass.

The backward pass steps from the current position in the direction of the start and aims to replace wrong patterns, which were caused during the forward pass. Without certain restrictions this is very problematic as will be shown with the following example. The rules, which are applied, are based on the probability table 6.3. The red character marks the position of the current index of the backward pass

```
o o o o o o b i o o ...     # replace i o with o o
o o o o o o b o o o ...     # replace b o with i l
o o o o o o i l o o ...     # replace o i with i i
o o o o o i i l o o ...     # replace o i with i i
o o o o i i i l o o ...     # replace o i with i i
...
```

In this example, the application of rule `o i` → `i i` caused the replacement of all preceeding `o` labels. Without restrictions, the backward pass most likely replaces more correct labels than it corrects wrong transitions.

6.3.2 Implementation

Taking the previously stated considerations into account, a basic postprocessor for word embedding has been implemented. For the input, the probability vectors of each label were chosen, with a variable window size for the surrounding elements. Each label of the predicted label pair is translated into one-hot encoding and the pair is replaced directly in the input sequence. A decision tree, implemented by the python *scikit-learn* module, was chosen as classifier. The training samples were collected from the wrong transitions in the training set of the network during a forward pass after the network finished training.

To compare the results with and without postprocessing, the test set is labeled once with and once without the postprocessor and the scores are compared against each other. Additionally, the number of correct replacement labels is compared against the baseline.

6.3.3 Results

Using this implementation of a postprocessor, a few tests were run. For the decision tree, different maximum depths were set. The results are shown below.

	avg total GENIA	correctly replaced	%	avg total ACL	correctly replaced	%
baseline	2100.50	527.62	25.12	1800.29	495.73	27.12
max depth 3	2071.20	940.27	45.40	2088.00	825.33	39.53
max depth 5	2156.93	1063.33	**49.30**	1855.87	735.93	39.65
max depth 8	2023.14	981.14	48.50	1608.43	668.86	**41.58**
no limit	2150.71	940.71	43.74	1648.86	632.79	38.38

Table 6.4: Number of correctly replaced labels compared to the average total number of labels (column total [CORPUS]). The total number of labels differs for each test run and is averaged over the test runs with the same parameter configurations.

As the results show, the performance is rather bad. Not even half of the labels are predicted correctly. Compared to the baseline, all decision trees show distinct improvement. As expected, setting the maximum depth of the decision tree to no limit, caused the tree to overfit the data, which in turn led to the worst percentages of all decision trees. The best results are achieved with a maximum depth of 5 or 8.

The changes on the F-Score are shown in table 6.5. There is no direct correlation between decision tree performance and score changes visible, as the scores are averaged over all target classes. So it is possible that the correction improves one class, but at the same time worsens another, which is less represented in the data. The decrease in score is stronger for the class with fewer samples, which causes a decrease for the average F-Score.

However, the data shows that if there is a positive difference, precision increases more than recall. It also shows that the results are better on the GENIA corpus.

	GENIA			ACL		
	precision	recall	F-Score	precision	recall	F-Score
baseline						
max depth 3	+0.0018	-0.0002	+0.0812	+0.0053	-0.0026	**+0.2083**
max depth 5	+0.0049	+0.0030	**+0.3953**	-0.0020	-0.0040	-0.2802
max depth 8	+0.0040	+0.0007	+0.2285	-0.0017	-0.0108	-0.5320
no limit	-0.0009	-0.0012	-0.1041	-0.0081	-0.0160	-1.1345

Table 6.5: Differences between average scores before and after postprocessing.

One cause is probably the difference in label balances. All previous test runs have shown worse performance for the out-of-domain corpus, which is partially reflected in theses results.

Assuming there exists a better postprocessor with a hit rate above 90%, the increase of both precision and recall is about 4-6%. This shows that further improvement of the postprocessor can slightly improve the scores, however, the development is beyond the scope of this thesis.

7. Conclusion

In this thesis, a general approach how recurrent neural networks and especially Long Short-Term Memory networks can be applied to perform terminology extraction was presented. Starting with an input sentence, processing of the text, feature extraction and labeling were explained. For the neural network architecture, problems with recurrent neural networks were reviewed and the advantages of Long Short-Term Memory networks and Gated Recurrent Units were stated. As network training algorithm, mini-batch gradient descent was introduced.

Based on the presented neural network architecture, a complete model capable of sequence-to-sequence classification was proposed. Several experiments were conducted in which the effects of different hyperparameters and topological characteristics were observed. As metrics, precision recall and F-Score were introduced. Furthermore, characteristics such as loss and training time were observed as well.

The experiments using the GENIA corpus have shown that increasing the hidden size and the number of layers has a positive and stabilizing effect on the F-Score. Furthermore, it has been shown that a growing batch size has a negative effect. Also, the effects of the presented embeddings were analyzed and the results compared against each other. These experiments have shown that character embedding performs better for most setups. In this context, experiments with different grades of preprocessing were conducted, which demonstrated the importance of capitalization and punctuation on the results.

To test the limitations of the model, tests on an out-of-domain corpus have been conducted. The results revealed that the GENIA and ACL corpus are very different in structure to which the network adapts and thus performs significantly worse on the out-of-domain corpus. Moreover, further experiments using the ACL corpus have shown the negative influences of label imbalance and little training data on the performance of the network. In this context, GRUs have shown an enhancing effect on the scores, which could not be noted in experiments with the previously mentioned setups.

In order to further enhance the results, different forms of regularization were applied to the neural networks. Dropout has shown slight positive effects when the GENIA corpus was used as training data. On the ACL corpus, no improvement was noted. Weight decay has shown to boost the results of word embedding, but has shown negative effect on character embedding and GRUs. Similar observations were made when Adagrad was used as learning algorithm. The use of GRUs has shown no positive or negative effects in these experiments.

Lastly, the classification errors were analyzed and possible ways of postprocessing were introduced. The implementation of a postprocessor has proven to be tedious and error prone. Using this implementation, the changes of the results were negligibly small, however it has been shown that better postprocessing can help to improve the scores.

Comparing the performance of this approach to existing systems is difficult for two reasons. First of all, here, sequence-to-sequence classification is performed. This means, there is no list of terms, which can be compared to against a predetermined list for the F-Score calculation. Secondly, there is currently no possibility to identify *nested terms* (terms, which consist of several words, of which substrings are defined as terms as well), as a word can only be assigned one label at a time. Nevertheless, the scores achieved with this model are clearly impressive considering the simplicity of the approach.

Compared to common systems, this approach poses a few advantages. The performance does not rely on the quality of the PoS tagger and does not require any linguistic knowledge. This makes it easy to apply to other languages even without knowledge of their syntactic properties. The achieved recall and precision do not differ much from each other, i.e. the improvement of one usually does not go at the expense of the other. This also shows the stability of the approach. Especially on in-domain data, the results are striking. Even on sparse data, minor changes can be applied to improve the classification results.

In the future, this approach can aid common system by providing a second view on the input data. Further application to other corpora and the use of different types of neural networks may give better insight on the mechanics and improve the achieved scores.

Bibliography

[BDVJ03] Y. Bengio, R. Ducharme, P. Vincent and C. Jauvin. A neural probabilistic language model. *Journal of machine learning research* 3(Feb), 2003, p. 1137–1155.

[BeBe61] R. Bellman and R. Bellman. *Adaptive Control Processes: A Guided Tour*. Princeton Legacy Library. Princeton University Press. 1961.

[BeCo57] R. Bellman and R. Corporation. *Dynamic Programming*. Rand Corporation research study. Princeton University Press. 1957.

[Beng12] Y. Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*, p. 437–478. Springer, 2012.

[BeSF94] Y. Bengio, P. Simard and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks* 5(2), 1994, p. 157–166.

[Bott10] L. Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, p. 177–186. Springer, 2010.

[Bril00] E. Brill. Part-of-speech tagging. *Handbook of natural language processing*, 2000, p. 403–414.

[CGCB14] J. Chung, Ç. Gülçehre, K. Cho and Y. Bengio. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *CoRR* Volume abs/1412.3555, 2014.

[CMGB$^+$14] K. Cho, B. van Merrienboer, Ç. Gülçehre, F. Bougares, H. Schwenk and Y. Bengio. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *CoRR* Volume abs/1406.1078, 2014.

[dpwe00] dpwe. ICSI Speech FAQ: 6.3 How are neural nets trained?, 2000.

[dSCPR13] M. da Silva Conrado, T. A. S. Pardo and S. O. Rezende. A Machine Learning Approach to Automatic Term Extraction using a Rich Feature Set. In *Proceedings of the NAACL HLT 2013 Student Research Workshop*, Atlanta, Georgia, June 2013. p. 16–23.

[DuHS11] J. Duchi, E. Hazan and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research* 12(Jul), 2011, p. 2121–2159.

[Elma90] J. L. Elman. Finding Structure in Time. *Cognitive Science* 14(2), 1990, p. 179–211.

[GeSC99] F. A. Gers, J. Schmidhuber and F. Cummins. Learning to Forget: Continual Prediction with LSTM. *Neural Computation* Volume 12, 1999, p. 2451–2471.

[GFGS06] A. Graves, S. Fernández, F. Gomez and J. Schmidhuber. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In *The ICML '06 Proceedings of the 23rd international conference on Machine learning*, Pittsburgh, PA, USA, June 2006.

[GoBC16] I. Goodfellow, Y. Bengio and A. Courville. *Deep Learning.* MIT Press. http://www.deeplearningbook.org, 2016.

[GoLe14] Y. Goldberg and O. Levy. word2vec Explained: deriving Mikolov et al.'s negative-sampling word-embedding method. *CoRR* Volume abs/1402.3722, 2014.

[GSKS$^+$17] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink and J. Schmidhuber. LSTM: A search space odyssey. *IEEE transactions on neural networks and learning systems* 28(10), 2017, p. 2222–2232.

[Guo13] J. Guo. Backpropagation through time. *Unpubl. ms., Harbin Institute of Technology*, 2013.

[Hayk09] S. S. Haykin. *Neural networks and learning machines.* Pearson Education, Upper Saddle River, NJ. Third. Edition, 2009.

[Hoch91] S. Hochreiter. Untersuchungen zu dynamischen neuronalen Netzen. *Diploma, Technische Universität München* Volume 91, 1991, p. 1.

[HoSc97] S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. *Neural Computation* 9(8), 1997, p. 1735–1780.

[HSKS$^+$12] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever and R. R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[Jord97] M. I. Jordan. Chapter 25 - Serial Order: A Parallel Distributed Processing Approach. In J. W. Donahoe and V. P. Dorsel (ed.), *Neural-Network Models of Cognition*, Volume 121 of *Advances in Psychology*, p. 471 – 495. North-Holland, 1997.

[JoZS15] R. Jozefowicz, W. Zaremba and I. Sutskever. An empirical exploration of recurrent network architectures. In *International Conference on Machine Learning*, 2015, p. 2342–2350.

[KOTT03] J.-D. Kim, T. Ohta, Y. Tateisi and J. Tsujii. GENIA corpus — A semantically annotated corpus for bio-textmining. *Bioinformatics (Oxford, England)* Volume 19 Suppl 1, 02 2003, p. i180–2.

[KrHe92] A. Krogh and J. A. Hertz. A simple weight decay can improve generalization. In *Advances in neural information processing systems*, 1992, p. 950–957.

[Lutz13] M. Lutz. *Learning Python*. O'Reilly. 5. Edition, 2013.

[LVJRT14] J. A. Lossio-Ventura, C. Jonquet, M. Roche and M. Teisseire. Yet Another Ranking Function for Automatic Multiword Term Extraction. In A. Przepiórkowski and M. Ogrodniczuk (ed.), *Advances in Natural Language Processing*, Cham, 2014. Springer International Publishing, p. 52–64.

[MCCD13] T. Mikolov, K. Chen, G. Corrado and J. Dean. Efficient Estimation of Word Representations in Vector Space. *CoRR* Volume abs/1301.3781, 2013.

[MiBa16] M. Miwa and M. Bansal. End-to-end Relation Extraction using LSTMs on Sequences and Tree Structures. *CoRR* Volume abs/1601.00770, 2016.

[Mood92] J. E. Moody. The effective number of parameters: An analysis of generalization and regularization in nonlinear learning systems. In *Advances in neural information processing systems*, 1992, p. 847–854.

[Olah15] C. Olah. Understanding LSTM Networks, 2015.

[PaMB13] R. Pascanu, T. Mikolov and Y. Bengio. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, 2013, p. 1310–1318.

[PaPZ05] M. T. Pazienza, M. Pennacchiotti and F. M. Zanzotto. Terminology extraction: an analysis of linguistic and statistical approaches. In *Knowledge mining*, p. 255–279. Springer, 2005.

[PeSM14] J. Pennington, R. Socher and C. D. Manning. GloVe: Global Vectors for Word Representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2014, p. 1532–1543.

[PGCC18] A. Paszke, S. Gross, S. Chintala and G. Chanan. *Pytorch Documentation*, 2018.

[Powe11] D. M. W. Powers. Evaluation: from Precision, Recall and F-measure to ROC, Informedness, Markedness and Correlation. *Journal of Machine Learning Technologies* 2(1), 2011, p. 37—63.

[QaSc16] B. QasemiZadeh and A.-K. Schumann. The ACL RD-TEC 2.0: A Language Resource for Evaluating Term Extraction and Entity Recognition Methods. In N. C. C. Chair), K. Choukri, T. Declerck, S. Goggi, M. Grobelnik, B. Maegaard, J. Mariani, H. Mazo, A. Moreno, J. Odijk and S. Piperidis (ed.), *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016)*, Paris, France, may 2016. European Language Resources Association (ELRA).

[RaRo09] L. Ratinov and D. Roth. Design challenges and misconceptions in named entity recognition. In *Proceedings of the Thirteenth Conference on Computational Natural Language Learning CoNLL '09)*, Boulder, CO, USA, June 2009.

[RuHW86] D. E. Rumelhart, G. E. Hinton and R. J. Williams. Learning representations by back-propagating errors. *nature* 323(6088), 1986, p. 533.

[RuNo12] S. Russell and P. Norbvig. *Künstliche Intelligenz*. Pearson. 3. Edition, 2012.

[ScPa97] M. Schuster and K. K. Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing* 45(11), 1997, p. 2673–2681.

[SoMN10] R. Socher, C. D. Manning and A. Y. Ng. Learning continuous phrase representations and syntactic parsing with recursive neural networks. In *Proceedings of the NIPS-2010 Deep Learning and Unsupervised Feature Learning Workshop*, Volume 2010, 2010, p. 1–9.

[Ston15] J. V. Stone. *Information Theory*. Sebtel Press. 2015.

[Suts13] I. Sutskever. Training recurrent neural networks. *University of Toronto, Toronto, Ont., Canada*, 2013.

[SuVL14] I. Sutskever, O. Vinyals and Q. V. Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, 2014, p. 3104–3112.

[WaLM16] R. Wang, W. Liu and C. McDonald. Featureless Domain-Specific Term Extraction with Minimal Labelled Data. In *Proceedings of the Australasian Language Technology Association Workshop 2016*, Melbourne, Australia, December 2016. p. 103–112.

[YuGZ17] Y. Yuan, J. Gao and Y. Zhang. Supervised Learning for Robust Term Extraction. In *International Conference on Asian Language Processing (IALP 2017)*. IEEE, October 2017. © 2017 IEEE.

[YZHS$^+$13] K. Yao, G. Zweig, M.-Y. Hwang, Y. Shi and D. Yu. Recurrent neural networks for language understanding. In *Interspeech*, 2013, p. 2524–2528.