PROJECT REPORT

Character-wise Natural Language Processing using sparse encodings

submitted to the

Marshallplan-Jubiläumsstiftung
Austrian Marshall Plan Foundation
Fostering Transatlantic Excellence

submitted by:

Martin Uray, BSc

FH Salzburg

UMASS
AMHERST

Supervisor SUAS:	FH-Prof. Univ.- Doz. Mag. Dr. Stefan Wegenkittl
Supervisor UMass:	Prof. James Allan

Salzburg, October 10, 2018

Abstract

The application of neural networks has gotten more attention over the past few years. This trend affected research on contradiction detection, where the goal is to predict whether the information contained in a premise and hypothesis entail or contradict each other. The most novel and best performing approaches rely on neural networks. In this work, a comparison of neural methods for Natural Language Inference with non-neural models is done. The effects of certain input features to a model will be shown, as well as the prediction on certain textual and linguistic phenomena for neural and non-neural models. Further, the influence of the textual properties of the input of premise and hypothesis, such as sentence length and similarity, are analyzed. The results show clearly that both the neural and non-neural approaches have their advantages and perform better than the other in certain cases.

Contents

1 Introduction 1

2 Background 3
- 2.1 Neural Network Models 3
  - 2.1.1 Feed-forward Neural Network 3
  - 2.1.2 Recurrent Neural Network (RNN) 8
  - 2.1.3 Convolutional Neural Network (CNN) 16
- 2.2 Textual Input Encodings 19
  - 2.2.1 One-hot encoding 19
  - 2.2.2 Dense Word representations 20
  - 2.2.3 Pre-trained Word Embeddings 21
  - 2.2.4 Character-wise Encodings 24

3 Neural Network Models for Contradiction Detection 26
- 3.1 Adapted NLP Four-Step Approach 26
- 3.2 Feed-forward Neural Networks for NLI 30
- 3.3 Recurrent Neural Network for NLI 30
- 3.4 Convolutional Neural Network for NLI 31
- 3.5 Effect of Network Types on Neural Models 32
- 3.6 Effect of Input Features on Neural Models 34

4 Exploring Contradiction with Neural Models for Natural Language Inference 35
- 4.1 The Neural Model 35
- 4.2 Experimental setup 36
- 4.3 Experiments 37

5 Conclusion 44

Abbreviations 45

List of Figures 47

List of Tables 48

Bibliography 49

1 Introduction

The author was allowed to stay at the Center for Intelligent Information Retrieval at the University of Massachusetts, Amherst during the summer semester, from March to August. This exchange is funded by the Austrian Marshall Plan Foundation.

The research department at the Salzburg University of Applied Sciences (SUAS) has a high interest in the methodologies of Natural Language Processing (NLP) for extracting informations from textual data. An example is the text annotation engine ARIE (Ferner et al., 2017). NLP in general is an area from computer science and artificial intelligence that is concerned with the human-machine interaction, in particular with a large amount of natural language. Very often the input data is represented in some form of word-level. Examples are *Bag-of-Words* or word-embeddings (Goldberg, 2017).

On the contrary, words can also be processed on word-level. This raw approach takes the sequence of characters as the input to the further processing. A reason for using the input on character-level is to be assumed, that here the algorithm can be adapted to another language much easier as well as no dictionary needs to be present and the relationships among all the words can be learned by a neural network during the training process. A disadvantage might be a much longer training time. As an example, Zhang, Zhao, and LeCun (2015) used a character-level input-encoding for text classification.

Here in this report, the result of the research during the exchange is described. The impact of using character-level input-encodings for NLP was analyzed. Here the problem of Natural Language Inference (NLI) (MacCartney, 2009) is used. NLI, also called Recognizing Textual Entailment (RTE), is the problem of identifying the semantic relation between two natural language elements, and whether they can be reasonably inferred from each other. The two elements of an NLI problem are called hypothesis h, and premise p, and can either be represented on sentence or document level. In the following example from Williams, Nangia, and Bowman (2017), the hypothesis is regarded to be contradicting to the premise.

Example 1.

p	*"we have provided an invoice to facilitate your gift."*
h	*"there's no invoice available for your gift."*

The hypothesis h of Example 1. can be considered as contradicting to the premise p. Any-

one who is presented the two sentences is very likely to confirm this assumption. Without any doubt, the statement expressed in h provides different information from those that can be found in p. A more formal description of contradiction is that there is no world in which the two statements A and B are both true at the same time. In simple words, both events are unlikely to be true simultaneously.

This report starts with a chapter (Chapter 2) with the fundamentals of the research. Here a short overview of neural network models with an application on NLI and commonly used embeddings for textual data is given. Chapter 3 shows the application of neural networks for NLI. Several experiments to compare word- and character-based approaches for neural methods are described in Chapter 4.

2 Background

This chapter will first introduce Neural Network (NN) models with usage on NLI (Section 2.1). The question of how natural language is encoded and embedded for NNs, especially with respect to NLI, will be answered in Section 2.2. This section includes methods of processing natural language character-wise or by means of *word embeddings*.

2.1 Neural Network Models

Over the last few years, the interest in the field of machine learning has rapidly grown. Also, the field of applications has gotten more diverse, which has lead to the creation of a rapidly growing new field of industry. Self-driving cars, speech recognizer, and enhanced translators would not be possible without machine learning techniques and methodologies. An approach to machine learning are *Artificial* NNs, which have a broad application in natural language.

In this following Section 2.1.1 the easiest form of NN, the feed-forward NN, will be explained. Following that, an insight into a state of the art types of NNs with application in NLI is given for Recurrent Neural Network (RNN) (Section 2.1.2) and Convolutional Neural Network (CNN) (Section 2.1.3).

2.1.1 Feed-forward Neural Network

The simplest form of a NN is a purely linear combination of the input data, adapted by coefficients. The parameters are learned through the training process. A simple three-layer NN can be seen in Figure 2.1. This one consists of input, hidden- and output-layer.

The output is some form of encoding of the desired classification task of the NN that is to be inferred from the training data.

A unit in a feed-forward network is called *neuron*. The output is calculated in two steps, based on a linear discrimination function. First, the *net activation* (or *net*) is calculated,

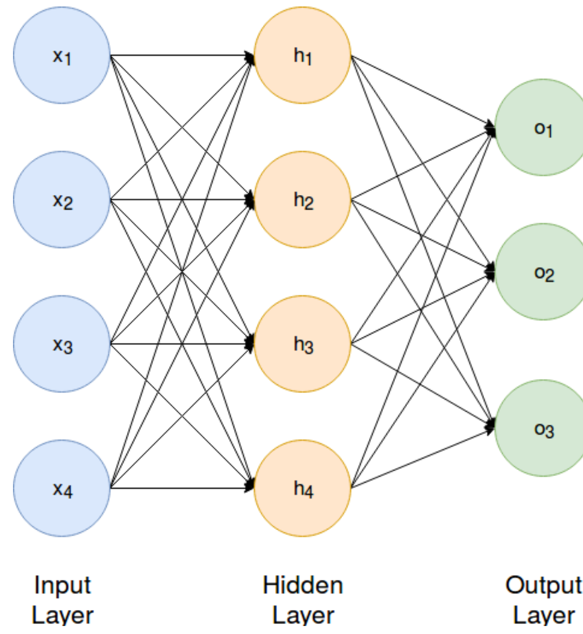


Figure 2.1: A simple three layer NN, consisting of four input nodes, four hidden nodes, and three output nodes.

which is the weighted sum of the neurons input. An additional bias is added. In Equation (2.1), this *net activation* is formally defined. x as the input and w as the weights are represented as vectors, where $W,X \in \mathbb{R}^n$ and w_n and x_n are the n-th element of W and X, respectively. b_0 is the added bias ($b_0 \in \mathbb{R}$). In other literature (Bishop, 1995), the bias is sometimes shown considered as the weight w_0. The corresponding x_0 value is set to 1.

$$net =< W^T, X > +b_0 = b_0 + \sum_{i=0}^{d} w_i \cdot x_i \tag{2.1}$$

For making other than linear relationships explainable by NNs, a non-linear *transfer function*:

$$y = a(net), \tag{2.2}$$

is applied to the net.

A simple two way linear activation function assigns the input to an output target that is defined by the decision rule

$$sign(z) = \begin{cases} 1, & \text{if } z \geq 1 \\ -1, & \text{if } z < 0 \end{cases} \tag{2.3}$$

The activation depends on the field of application. In certain fields, other *transfer functions* perform significantly better. The *sigmoid (σ) transfer function* (also called *logistic function*)

$$\sigma(z) = \frac{1}{1+e^{-z}} \tag{2.4}$$

transforms its input according a S-shaped curve into the range $[0,1]$. This function was to be considered as the choice for NNs. Currently it is stated to be deprecated, as other ones prove to perform significantly better (Goldberg, 2017).

NLP tasks mostly rely on the *tanh transfer function*

$$tanh(z) = \frac{e^{2z}-1}{e^{2z}+1}. \tag{2.5}$$

Each data point x is transformed into the range of $[-1,1]$. The Rectifier Linear Unit (ReLU), also known as the *ramp function*, is another special type of *transfer function*

$$RELU(z) = max(0,z). \tag{2.6}$$

CNNs (Section 2.1.3) often rely on ReLUs.

The illustration for the activation functions defined in Equation 2.3, 2.4, 2.5 and 2.6 can be seen in Figure 2.2.

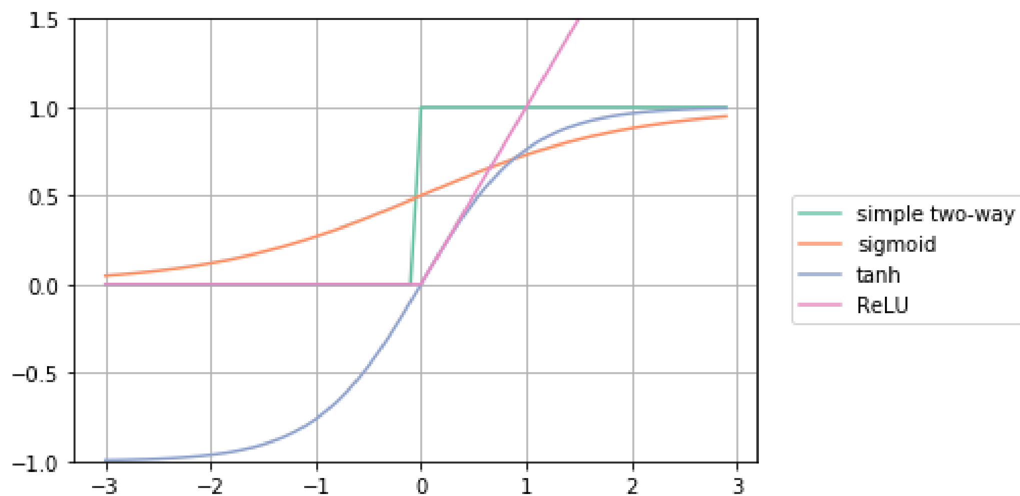


Figure 2.2: Different *transfer functions* as defined in Equation (2.3), (2.4), (2.5), and (2.6) respectively.

In Figure 2.3, a schematic illustration of this stated two step process can be seen.

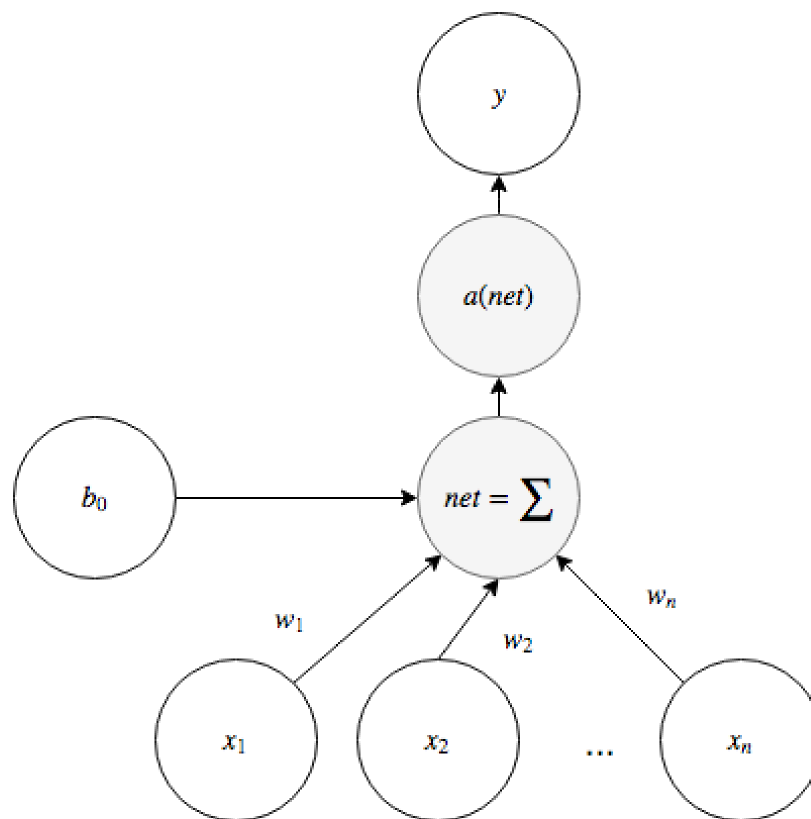


Figure 2.3: A simple Neuron of a NN, consisting of an input vector x, an vector with the weights w and a bias b_0. This illustration's formal definition can be found in Equation (2.1) and Equation (2.2).

For a fully connected layer, the definition becomes

$$net_k = W_k^T \cdot X + b_{k0} = \sum_{j=1}^{m_H} y_j \cdot w_{kj} + b_{k0}, \tag{2.7}$$

where k is the index for the unit in the output layer, and m_H the number of hidden units per layer. Analogous to a single unit, the *transfer function* for a layer looks like

$$y_k = a(net_k), \tag{2.8}$$

analogously.

The network structure presented in Figure 2.1 is a very simple one. For modern architectures, n hidden layers with varying m_i units per layer i are applied. The number n needs to be chosen carefully. The deeper the architecture of the model, the more training data is usually needed in order to use the network to its full capabilities. When to little data is available, the structure leads to over-fitting to the available training data. On the other hand, flat structures with only a small number of layers, may not be able to compute complex problems without using a very large and difficult to handle number of hidden nodes (Goodfellow, Bengio, & Courville, 2016).

The Kolomogorov's theorem states that any multi-variant, continuous function can be represented as a composition of a finite number of continuous functions with two parameters (Tikhomirov, 1991). This theorem has an intresting relationship to NNs. Applied to NNs, a continuous function $y(x)$ with d input variables x_i can be mapped to the output y by only using a three-layered NN. To meet this theorem, the first layer needs $d(2d+1)$ and the second $2d+1$ dimensions (Bishop, 1995).

The input data is applied to the input nodes of the network. These nodes are just cells that take the input data and represent it. At this stage, no processing is done. Starting with the input layer, this data is passed layer-wise through the whole network. Using exemplarily the three-layer network in Figure 2.1, the input in the cells $x_{1...4}$ is passed to the nodes $h_{1...4}$ in the hidden layer. Each unit computes its state according to its applied input and the known weights. The result of each node is transformed using an activation function. Following, the result is then passed from the hidden nodes $h_{1...4}$ to the output nodes $y_{1...3}$ in the same manner as before, and the same approach as on the hidden nodes is applied, differing only in the used weights. The output layer represents the network's prediction. As the data is always passed forward through the network, this type of network is called *feed-forward* NN.

The learning process of an NN is based on an error function. The goal is to minimize this function value with respect to the used weights and biases in the sum of the network's units. A common approach is the so-called *Error backpropagation (BP)*. Assuming, that the networks *transfer functions* are differentiable, the *net* of the output units become differentiable functions of the input variables, and of the weights and biases. Applying a differentiable error function to the network outputs leads to an error, which is itself a differentiable function of the network's weights. The derivatives of the error can, therefore, be evaluated with respect to these weights and are then used to find weight values that lead to a minimal outcome of the error function. A common optimization method that is used, is the gradient descent. This algorithm is called BP, as the propagation of error is evaluated backward through the network. For a deeper insight into optimization methods in general, the reader is referred to (Bishop, 1995, Chapter 7).

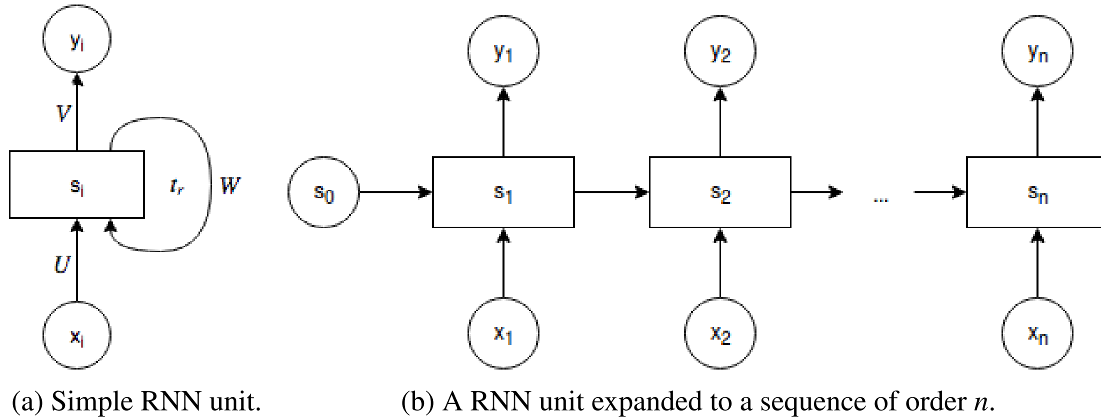
2.1.2 Recurrent Neural Network (RNN)

Applications, like language modeling, speech recognition, and machine translation, have to deal with sequential data. The application of feed-forward networks or CNNs (Section 2.1.3) is limited, as the length of the input data is constrained to a fixed size vector. For modeling natural language, RNNs are now widely used. These have to be shown state-of-the-art performance in many standard tasks (Mikolov, Kombrink, Deoras, Burget, & Cernocky, 2011).

Like the feed-forward network, basic RNNs are unit based networks. Each unit is unidirectionally connected to itself, which gives the unit knowledge about the previous step. This recurrence is weighted by the vector W. If this is set to zero, it poses to have the same characteristics as a *multilayer perceptron*. The activation for each unit is time-varying. The most simple form of a RNN with the sense of ordering elements of a sequence is the so-called *Elman Network* or Simple-RNN. It was applied for language modeling first by Mikolov (2012).

A bottleneck of classical feed-forward is that they are not capable of capturing the data's history. The value of the calculation at time step t is only based on the applied input. Hence it is not possible to use data or information from n previous time steps. RNNs ease this bottleneck by considering the intermediate state of n previous time steps by its recurrent feedback.

Another asset is that RNNs can represent more advanced data compared to feed-forward NNs. As an example, in *natural language* it is quite common that certain word-based pat-



(a) Simple RNN unit. (b) A RNN unit expanded to a sequence of order n.

Figure 2.4: The simple structure of RNN. Figure 2.4a shows a simple gate. The t_r indicates the delay of one step. This form can be also displayed in an unfolded form, as can be seen in Figure 2.4b

terns are shown on variable positions within the word. The usage of RNNs allows capturing those patterns, by remembering words in its hidden layers. In comparison, feed-forward NNs would need additional parameters for the position of the word within the sentence and much more training data.

An example of an RNN is illustrated in Figure 2.4a. RNNs have input, hidden and output layers. For each input and the hidden states, weights are applied. These are trained and represented by the vectors U, V and W respectively. As the data is fed sequentially, and the hidden state is passed to the next layer, this can be also represented in an expanded way (Figure 2.4b). But this still is just a representation, and the data is processed using a single node with recurrent flow. The values for the hidden state $s_j(t)$ and output $y_k(t)$ is computed as following, respectively:

$$s_j(t) = \sigma(\sum_i x_i(t) \cdot u_{ji} + \sum_l s_l(t-1) \cdot w_{jl}) \tag{2.9}$$

$$y_k(t) = g(\sum_j s_j(t) \cdot v_{kj}) \tag{2.10}$$

Using a vector representation, Equation (2.9) and Equation (2.10) can alternatively also be written as

$$s(t) = \sigma(U \cdot x(t) + W \cdot s(t-1)) \tag{2.11}$$

and

$$y(t) = g(V \cdot s(t)). \tag{2.12}$$

The used function $\sigma(.)$ is already described in Section 2.1.1, and $g(z)$ refers to the *softmax function*

$$softmax(z_i) = \frac{e^{z_i}}{\sum_{j=0}^{k} e^{z_j}}, \tag{2.13}$$

for $i = 0, 1, 2, \ldots, k$.

Applying an input to an RNN cell, the first element of this input x_i, as it appears in sequential order, is fed into the cell. First, it is multiplied by its specific input weight u_{ji}. The weighted input is then added to the output of the state before, $s(t-1)$. If there was no proceeding output, as this is the case for the first processed element, $s(t-1)$ is set to 0. The prior state is separately adjusted by the weight w_{jl}. On to the sum $s(t)$ of these two elements, the activation function σ is applied. The final result $y(t)$ is computed by multiplying the intermediate result after the activation function with the output weight v_{kj}, followed by the softmax function $g(z)$. This result is the returned. In the next step the next element x_{i+1} can be processed accordingly, with the previously processed intermediate state $s(t)$ as its previous state $s(t-1)$.

The network itself is trained by the method of stochastic gradient descent. Usually, a BP or backpropagation through time (BPTT) algorithm is applied (Rumelhart, Hinton, & Williams, 1986). On details, how the BPTT can be implemented, the reader is referred to Bodén (2002).

A major drawback of RNNs is that they do not capture long-term dependencies. The gradients for RNN, captioning these relations, either tend to vanish or explode. For the first case, learning long time lags takes a lot of time, or hardly work at all. For the latter, weights start to oscillate and tend to be unstable. This phenomena was discussed by Hochreiter (1991) and Bengio, Simard, and Frasconi (1994).

Long-Short Term Memory (LSTM)

To overcome the previous discussed restrictions to the RNN, Hochreiter and Schmidhuber (1997) first introduced an adapted RNN architecture, the so called Long-Short Term Memory (LSTM).

This adaption also included an appropriate gradient-based learning algorithm that keeps the error within the cell constant.

The LSTM unit is more complex and is called *memory cell*. An example for such one can be

seen in Figure 2.5. The cell embodies a constant error carrousel (CEC) in its architecture, to keep the error flow through the network constant. The cell is denoted by c_j, where j is the index for the memory cell in the layer. In the center of the cell, the CEC is a fixed linear, self-connected unit. Also an multiplicative *input gate unit* and *output gate unit* are introduced. These protect the memory contents stored in c_j from perturbation by irrelevant input or protects others from likewise.

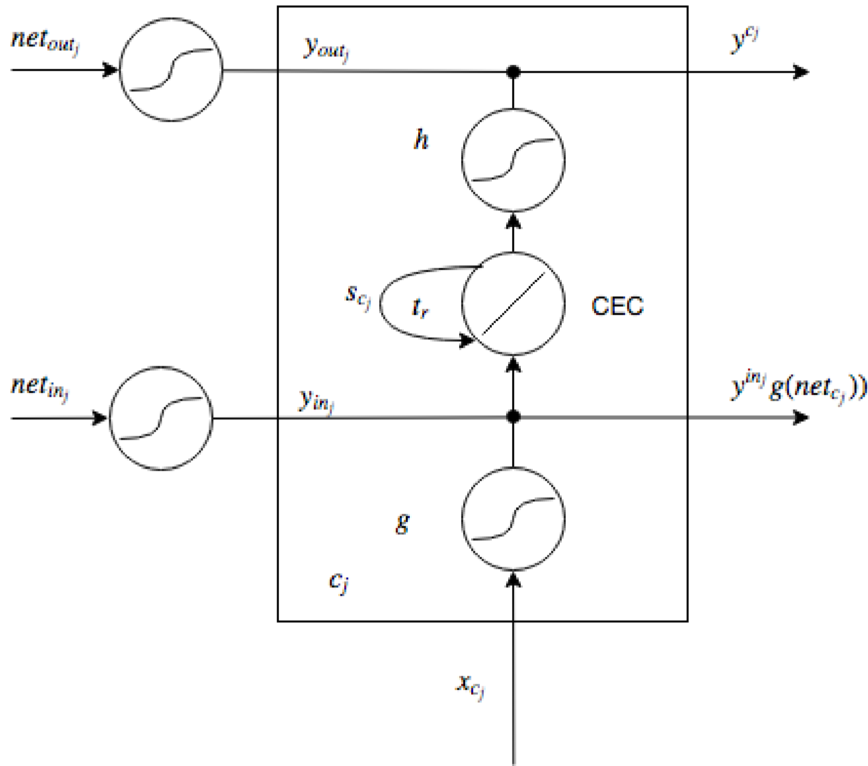


Figure 2.5: A simple LSTM cell, which embodies a CEC as the core and the activations g and h in its architecture (Hochreiter & Schmidhuber, 1997).

An LSTM memory cell has three inputs. The first one, x_{c_j}, is the input for the cell c_j. Furthermore the cell is provided with input from the multiplicative units in_j and out_j, where their activations $y_{in_j}(t)$ and $y_{out_j}(t)$ are defined by

$$y_{in_j}(t) = f_{in_j}(net_{in_j}(t)) \tag{2.14}$$

$$y_{out_j}(t) = f_{out_j}(net_{out_j}(t)). \tag{2.15}$$

The internal states $net_{out_j}(t)$, $net_{in_j}(t)$ and $x_{c_j}(t)$ are calculated by

$$net_{in_j}(t) = \sum_u w_{in_j u} \cdot y^u(t-1), \tag{2.16}$$

$$net_{out_j}(t) = \sum_u w_{out_j u} \cdot y^u(t-1), \tag{2.17}$$

and

$$x_{c_j}(t) = \sum_u w_{c_j u} \cdot y^u(t-1) \tag{2.18}$$

respectively. u are the connected units to the cell, in case there are such units. All the units in the cell can contain useful information about the current state. As an example, the input (or output) gate may use outputs from other gates to define if a state is stored in its cell or not.

The output of the memory cell at time t is calculated by

$$y^{c_j} = y^{out_j}(t) \cdot h(s_{c_j}(t)). \tag{2.19}$$

The state $s_{c_j}(t)$, that is represented in the CEC, is defined by

$$s_{c_j}(t) = \begin{cases} 0 & \text{if } t = 0 \\ s_{c_j}(t-1) + y_{in_j}(t) \cdot g(x_{c_j}(t)) & \text{if } t > 0 \end{cases} \tag{2.20}$$

The second part of the equation $y_{in_j}(t) \cdot g(x_{c_j}(t))$ denotes the internal state of the memory cell. This one is only dependent on the current input. In Figure 2.5 this is shown in the center of the memory cell, where the factor t_r denotes the delay of one full cycle. The function g and h are intended to squash x_{c_j} and scale the output of the internal state s_{c_j} respectively.

A limitation posed by simple LSTM cells is that the internal state s_c tends to grow during the processing of a time series. This results in a cell state that is growing without limit. A state of saturation of the output squashing function h may be the result. This saturation affects the cell by either blocking the incoming errors, by making h's derivate vanish and, following, making the output equal to the output gate activation. This makes the whole LSTM ineffective and reduces its functionality to a simple RNN gate (Gers, Schmidhuber, & Cummins, 1999).

To overcome this effect, Gers et al. (1999) introduced an adaption to the LSTM memory cell, by adding *forget gates*. The additional features can be seen in Figure 2.6. These gates enable the cell to learn to reset memory blocks when the content is not seen as useful anymore. The CECs constant delay of 1.0 is replaced by a multiplicative forget gate activation y^{φ_j}.

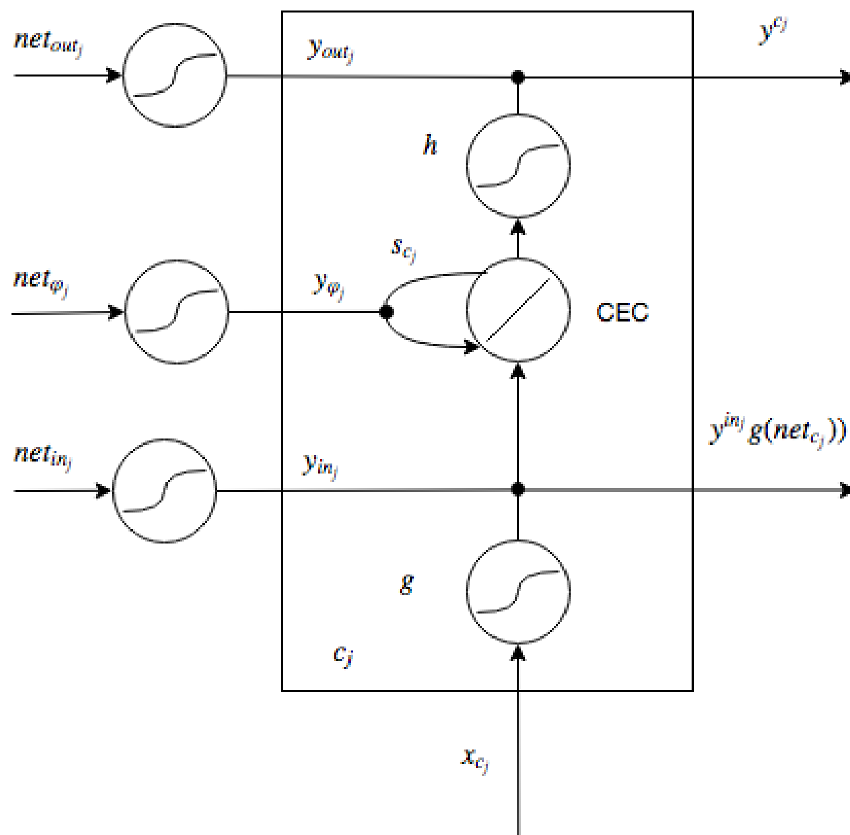


Figure 2.6: An adapted LSTM cell that embodies an additional forget gate. This gate is learned to trigger to reset the internal state to zero (Gers et al., 1999).

Similar to the activations of a simple LSTM (Equation (2.14) ...(2.17)), the forget gates activation y^{φ_j} is calculated by

$$y^{\varphi_j}(t) = f_{\varphi_j}(net_{\varphi_j}(t)) \tag{2.21}$$

where

$$net_{\varphi_j}(t) = \sum_u w_{\varphi_j u} \cdot y^u(t-1). \tag{2.22}$$

The forget gates activation is then used as a weight for the feedback connection of the internal state $s_{c_j}(t)$. The new equation for the internal state is

$$s_{c_j}(t) = \begin{cases} 0 & \text{if } t = 0 \\ y^{\varphi_j}(t) \cdot s_{c_j}(t-1) + y_{in_j}(t) \cdot g(x_{c_j}(t)) & \text{if } t > 0. \end{cases} \tag{2.23}$$

Another limitation, that is posed by basic LSTMs, restricts the gates to rely on the output of the cell. This means that the only information about its state a gate can observe is the output of the cell. Hence, no prediction about the actual current internal state s_{c_j} can be made. In the case that the gates are closed, no information can be seen (Gers, Schraudolph, & Schmidhuber, 2003).

Gers et al. introduced "peephole" connections to allow all gates to get information on the current cell state. As can be seen in Figure 2.7, weighted holes are connected from the CEC to the gates of the same memory block (Gers et al., 2003).

The resulting output gates activation changes accordingly to

$$net_{in_j}(t) = \sum_u w_{in_j u} \cdot y^u(t-1) + \sum_{v=1}^{s_j} w_{in_j c_j^v} \cdot s_{c_j^v}(t-1), \tag{2.24}$$

$$net_{out_j}(t) = \sum_u w_{out_j u} \cdot y^u(t-1) + \sum_{v=1}^{s_j} w_{out_j c_j^v} \cdot s_{c_j^v}(t-1), \tag{2.25}$$

and

$$net_{\varphi_j}(t) = \sum_u w_{\varphi_j u} \cdot y^u(t-1) + \sum_{v=1}^{s_j} w_{\varphi_j c_j^v} \cdot s_{c_j^v}(t-1), \tag{2.26}$$

whereas these equations replace Equation (2.16), (2.17), and (2.22), respectively.

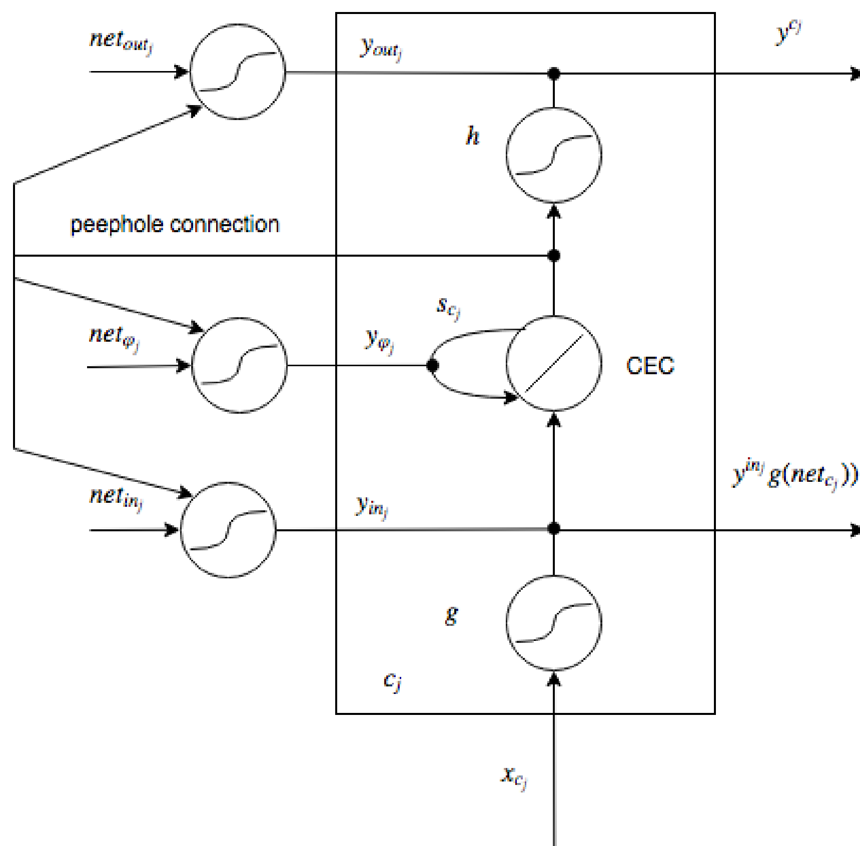


Figure 2.7: An adapted LSTM cell that embodies a *forget gate* and *peepholes*. The input gates are connected to the internal state to get information about the cells current state (Gers et al., 2003).

$$\overrightarrow{h_t} = \overrightarrow{LSTM}(w_1, \ldots, w_T) \tag{2.27}$$

$$\overleftarrow{h_t} = \overleftarrow{LSTM}(w_1, \ldots, w_T) \tag{2.28}$$

$$h_t = [\overrightarrow{h_t}, \overleftarrow{h_t}] \tag{2.29}$$

LSTMs can also be used bidirectional. This kind of LSTM is called Bidirectional LSTM (BiLSTM). Having a sequence of words, $w_{t_{t=1,\ldots,T}}$, applied to a BiLSTM, this computes a set of vectors h_{t_t} for $t \in [1, \ldots, T]$. Each h_t is a concatenation (Equation (2.29)) of an LSTM used forward (Equation (2.27)) and backwards (Equation (2.28)).

2.1.3 Convolutional Neural Network (CNN)

Input Images are mostly represented pixel-wise in two dimensions. Since feed-forward NNs are designed to handle vector shaped data, an application on these data would cause the network to have many more parameters, resulting in a proportionally grown training time. Additionally, in practice, more noise would be added during the training process. A CNN takes advantage of the data's multidimensionality.

CNNs combine three architectural ideas: local receptive fields, shared weights and spatial or temporal sub-sampling. This concept ensures that a reasonable degree of shift, scaling and distortion in the input does not need to be represented by the learned weights (LeCun, Bottou, Bengio, & Haffner, 1998).

The architecture of a CNN in general contains several independent, layer-wise organized steps. An example can be seen in Figure 2.8. The displayed network is designed to classify handwritten digits. The network is comprised of two convolutional layers, each followed by a sub-sampling layer, two fully connected layers, and a Gaussian connected layer.

The key component of the CNNs are the convolution layers. The convolution itself is a mathematical operation on two functions, resulting in a third. The convolution operator is an asterisk ($*$). On continuous data, the convolution is defined as (Smith, 1997):

$$(f * g)(t) = \int_{\tau=-\infty}^{\infty} f(\tau) \cdot g(t - \tau)\, d\tau.$$

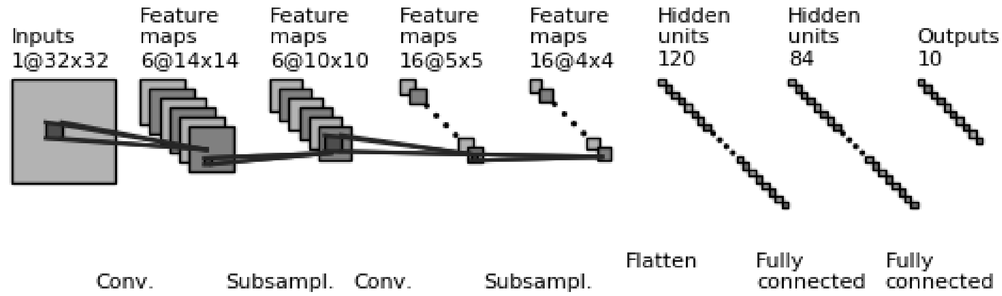


Figure 2.8: The network architecture for LeNet-5 with convolutional-, sub-sampling-, fully connected- and Gaussian connected layers. The networks purpose is to classify handwritten digits (LeCun et al., 1998).

In simple terms, the convolution at time t, can be seen as a weighted average of the function $f(\tau)$, where $g(-\tau)$ represents the weight. The discrete convolution of f and g is given by

$$(f * g)[i] = \sum_{j=0}^{M-1} f[j] \cdot g[i \cdot d - j],$$

where d is the stride. The discrete input function is $f(x) \in [1,l] \to \mathbb{R}$ and discrete kernel function $g(x) \in [1,k] \to \mathbb{R}$. The output of the convolution results in $(f * g) \in [1, \lfloor (l - k/d) \rfloor + 1] \to \mathbb{R}$.

The weights that are learned during the training process are the discrete kernel functions. These filter kernels aim to extract features from the data, like edges, endpoints or corners. In order to detect higher order features, these features are combined and further processed by the succeeding layers.

Each layer consists of several planes, which all share the same weights. The output over such units are called *feature maps*. To ensure that the three premises are fulfilled, the feature maps are constrained to perform the same operation on different parts of the image. A full convolution layer consists of several *feature maps*. For learning the weights during the training process, the BP algorithm is used (LeCun et al., 1998).

When setting up an architecture for a CNN, two parameters needs to be defined. These two define the filter kernel. The first parameter is the number of filters per layer. The second set of parameters that needs to be set are the shapes of the kernel filters. The shapes of these should be set according to the data fed into the convolution layer.

Passing the data through the network layers results in the necessity of a reduced dimensionality of the features. This is done by a *pooling-* or *sub-sampling layer*, what can be considered as a form of non-linear down-sampling. The most common pooling algorithm

is the *max pooling*. For this, the data is split into non-overlapping fields and the maximum value for each is selected. The same applies to the *average pooling*, as the average value of the input in the selected window is returned. In Figure 2.9 an example for the *max pooling* and the *average pooling* is shown. The result is used in the following layer. This reduction of dimensionality can be applied, under the assumption that the location of a feature is not relevant. When defining an architecture, one has to decide on the shapes of the pooling windows. Common shapes are $2x2$ or $4x4$. Research has shown that non-overlapping fields tend to show the best results (Scherer, Müller, & Behnke, 2010).

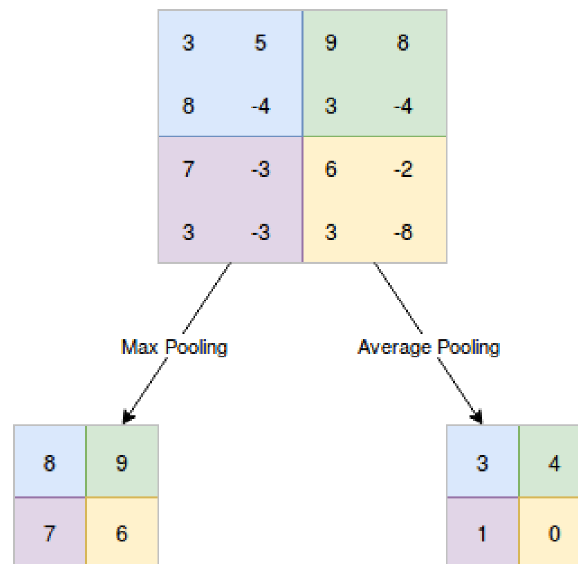


Figure 2.9: Graphical representation of the *max pooling* and the *average pooling*. Each non-overlapping window is shown with its own color. The results are shown in the corresponding fields, respectively.

After several convolution and pooling layers, fully connected layers are used to transform the data at a high-level abstraction into a final classification. The first neuron in a fully-connected layer is connected to all the outputs of the activation of the preceding layer.

As discussed in Section 2.1.1, the activation function that is mostly used with CNN is the ReLU. Applying this function, non-linear properties of the decision function and the whole network are increased. The receptive fields of the convolution layer are not affected. In comparison to other activation functions, like the sigmoid- or hyperbolic-tangent, the ReLU poses to be faster during training. Figure 2.10 shows the ReLU applied to sample data.

As already discussed, a benefit of CNNs is their capability of processing more dimensional data. Secondly, CNNs are easier to train, due to their reduced number of parameters. In comparison to feed-forward NN, fewer parameters are implemented, which results in a reduced training time. As a result of this lower number of parameters, less noise can influence the performance of the network.

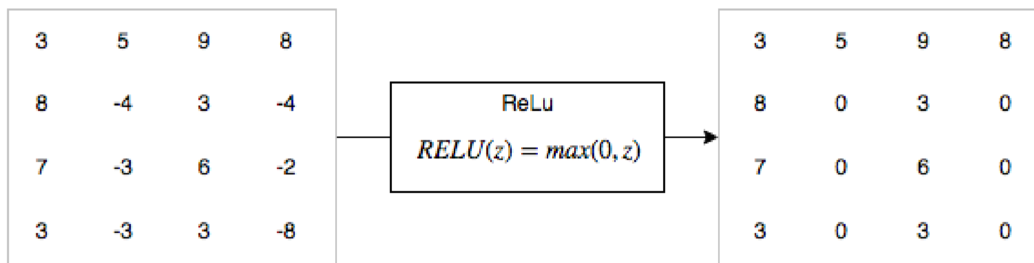


Figure 2.10: The ReLU function shown in example. The function $a(z) = max(0,z)$ is applied to every element.

2.2 Textual Input Encodings

As mentioned before, natural language can be presented in several forms. Mostly, one has to deal with input in form of words, letters, part-of-speech (POS) tags or acoustic speech (the latter one is not covered in this section). In this section, we will have a look into approaches on how to transform this kind of data into a form that can be processed by a statistical classifier or NNs.

2.2.1 One-hot encoding

An approach to convert textural data into a numeric representation is the application of a $1-of-m$ (or "one-hot") representation. Here, each dimension of the result represents a unique feature of the data. In this case, a word from the vocabulary is to be considered as one feature and dimension in this vector.

For illustration, let's consider an example of a collection of documents. The vocabulary that is created from this documents consists of $40,000$ unique words. Using a *Bag-of-Word* representation, the vector x will have a length of the vocabulary to consider. Let's assume a vocabulary v with $40,000$ different words and a document with 25 words. The resulting vector x will have $40,000$ elements, with utmost 25 non-zero values. In this example, say the word at index $34,768$ correspond to *cat* and index 6415 corresponds to *dog*. The vector x can clearly be considered as a sparse vector. An example, on how to construct an "one-hot" vector can be seen in Figure 2.11a.

Having such sparse vector encodings causes that similarity between words cannot be captured anyhow. As an example, the features "think" and "thinking" are not linked, and have the same relation to each other as to the word "car", for instance. This is easily illustrated

by means of the *dot product* that can show the similarity between vectors:

$$
\begin{aligned}
think \cdot thinks = & \\
& [0\,0\,0\,0\,0\,1\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0] \cdot \\
& [0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,1\,0\,0\,0\,0\,0\,0\,0]^T = 0
\end{aligned}
$$

Additionally, words, that are rarely seen during training will be represented poorly. Words that are not seen once during training cannot be classified later and will be treated as out-of-vocabulary. Languages with extremely large vocabularies, as agglutinative languages[1] (Finish and Turkish are being well studied), do get affected by this drawback (Bojanowski, Joulin, & Mikolov, 2015).

In real-world scenarios, textual input data often contain spelling mistakes and typos. The size of the vocabulary increases unnecessarily by adding several versions of the same word.

2.2.2 Dense Word representations

A milestone in the concept of representing data was done when moving to deeper non-linear models. For this purpose, not each feature had its own dimension anymore. Here the data is embedded into a fixed d dimensional vector space. A word is represented as a vector from that. The representation is on a much lower dimensionality compared to the sparse encoding. The dimensionality of this embedding space can vary and is usually between 100 to 200. This results in a much smaller dimension than the number of features. Such embedded representations are usually learned during the training, alongside other parameters of the network (Goldberg, 2017).

Word embeddings can be used for a more dense representation. The term word embedding is a collective term, containing a language model and feature learning techniques. Word embeddings are learned by a NN that includes the rich representation of the corpus into the new representation space. As a result, a word is mapped to a dense vector of real numbers.

Bengio, Ducharme, Vincent, and Janvin introduced the idea of learning a representation of the words over a corpus using NNs (Bengio et al., 2003). Besides NNs, other ways of creating dense representations include dimension reduction on word-level using the co-occurrence matrix (Levy & Goldberg, 2014b), probabilistic models (Globerson, Chechik, Pereira, & Tishby, 2007) or representation in terms of the word context (Levy & Goldberg, 2014a). An example of a word embedding is illustrated in Figure 2.11b. For this example,

[1] words that may contain different morphemes to determine their meanings

a fictional 6 dimensional word embedding is shown.

One advantage of using dense representations is that the dimensionality of the vectors to handle is much smaller. This causes the computation to be much faster, as NN toolkits don't work well with high dimensions. Another benefit that can be drawn from dense representation, is that similarities between words or terms can be learned. Let's assume a corpus, where the word *dog* occurs more often compared to the word *cat*. If each word is its own dimension, the word *dog* cannot say anything about the occurrence of *cat*. In a dense representation, the two words may share statistical properties, which leads to a similarity between those. This assumption, that during training the word *cat* has enough occurrences that such a relation can be learned. On pre-trained word embeddings (Section 2.2.3), these similarities can be observed (Goldberg, 2017).

2.2.3 Pre-trained Word Embeddings

Training an embedding only makes sense when there is a lot of training data available. In the contradicting case, it will be hard or impossible to train a model. For such cases, pre-trained embeddings that are trained on a huge amount of data can be used.

The first widely used word embedding algorithm was introduced by Collobert and Weston (also called C&W). This showcase proved to improve performance when a word embedding is trained on a large dataset that contains syntactic and semantic meaning (Collobert & Weston, 2008). As described in Goldberg (2017), natural language models are the basis of word embeddings, where the $softmax$ is essential. To overcome computing this costly function, a different objective function was used instead. The network is trained to output higher scores for a correct sequence of words. Incorrect sequences should get a lower score. To compare two sequences, a pairwise ranking-criterion was used. The resulting language model produces embeddings with already known properties, like relating words that are clustered together. Further details and an extensive explanation of the results can be found in Collobert et al. (2011).

The WORD2VEC is based on a series of publications (Mikolov, 2012; Mikolov, Chen, Corrado, & Dean, 2013). It is based on a language model that enables faster results. It uses the CBOW (Continues Bag-of-Words) and SKIP-GRAM as the context representation and *Negative-Sampling* and *Hierarchical Softmax* as optimization objectives.

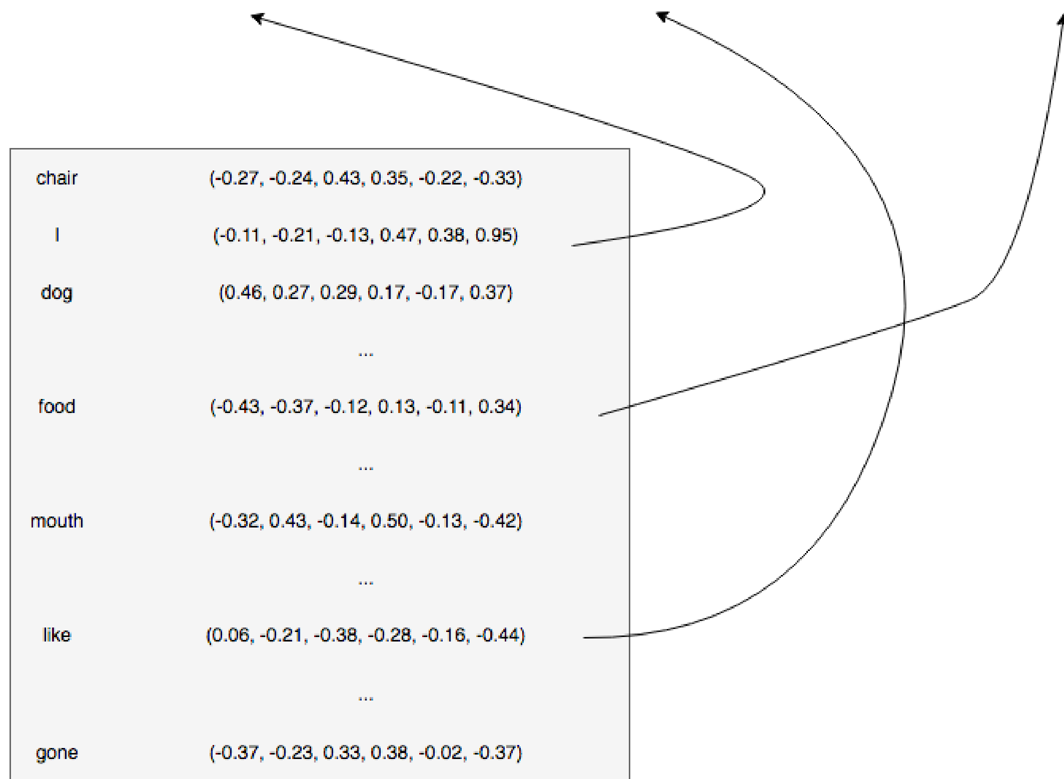
Figure 2.12 illustrates the application of the CBOW and SKIP-GRAM. CBOW's goal is, by means of a center words context, to predict the word in the center. According to a window around the center word, n words are used for the prediction. Using the Continues Bag-

$w = I$ $\quad$ $w = like$ $\quad$ $w = food$

$$x = (0,0,\ldots,0,1,0,\ldots,0)(0,0,\ldots,0,1,0,\ldots,0)(0,0,\ldots,0,1,0,\ldots,0)$$

(a) A sparse representation of features, in form of a *One-hot* encoding.

$$x = (-0.11,-0.21,-0.13,0.47,0.38,0.95)(0.06,-0.21,-0.38,-0.28,-0.16,-0.44)(-0.43,-0.37,-0.12,0.13,-0.11,0.34)$$



(b) A dense representation of the data, by means of a word embedding.

Figure 2.11: The comparison of the usage of sparse (a) to dense (b) vectors. (a) shows the concatenation of three "one-hot" vectors, describing the phrase *I like food*. (b) represents the concatenation of the embeddings in a figurative example with 6 dimensions of the same two words as in (a) (Goldberg, 2017).

of-Words Model, a probability for the center word is calculated. The SKIP-GRAM, on the other hand, does exactly the opposite. Having a word, the model provides probabilities for words that are likely to appear in the words context.

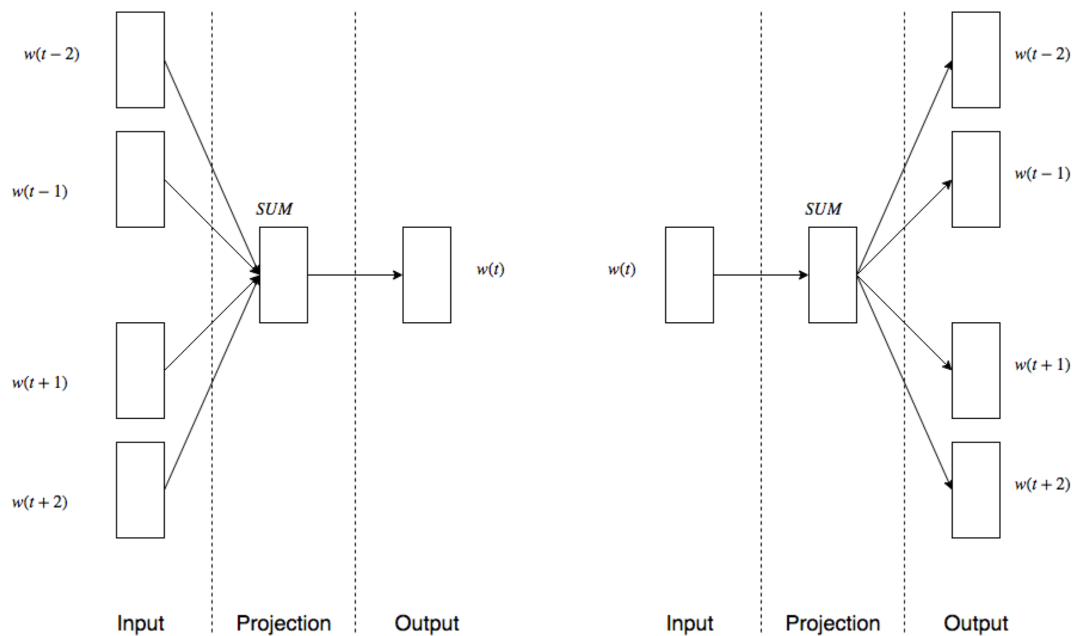


Figure 2.12: The usage of the CBOW (left) and SKIP-GRAM (right) applied on the WORD2VEC embedding model Mikolov et al. (2013).

To reconstruct the linguistic context of words, models are trained. The architecture of the models are shallow ones, with only two layers. With these models, two words sharing the same context are then positioned in the vector space so that they are close to each other. This relationship allows using these vectors to find similar relationships to other word pairs. An example, taken from Mikolov et al. (2013) shows how the analogy of "king is to queen as man is to woman" can be converted in the vector space to the equation $king - man + woman = queen$. Another similar example shows how the capital cities are related to their country: $Paris - France + Italy = Rome$.

Pennington, Socher, and Manning (2014) introduced the GLOBAL VECTORS (GLOVE). For this embedding, a specific weighted least squares model was introduced. This one trains global word-word co-occurrences counts. This makes this model efficient to use. The authors claim that this makes a meaningful structure within the learned embedding. The model was trained on a huge amount of data from several sources, like *Wikipedia*, crawled web data, and a Twitter dataset. Pre-trained GloVe embeddings are used in recently published NLI models (Gong, Luo, & Zhang, 2018; Tay, Tuan, & Hui, 2018).

More recently, Facebook introduced FASTTEXT (Joulin, Grave, Bojanowski, & Mikolov, 2017). This can be seen as an extension to the WORD2VEC model, as words are broken into

several *n-grams* (sub-words). After training, the embedding contains vectors for the sub-phrases seen during training. An advantage that this approach poses is that rare words can be represented, as their sub-phrases are more likely to appear in the trained vector space.

There are also embeddings published that do not focus explicitly on word-level. As an example, M. Chen (2017) represents each document as a simple average of word embeddings, thus ensuring a documents representation that captures semantic meanings during training. *Tweet2Vec* (Vosoughi, Vijayaraghavan, & Roy, 2016) represents Tweets in a vector format.

Two of the most common word embeddings are WORD2VEC (Mikolov et al., 2013) and GLOVE (Pennington et al., 2014). For further reading on pre-trained word embeddings in general, the reader is referred to (Goldberg, 2017, Chapter 10).

The usage of word embeddings has lead to state-of-the-art results in sequential tasks on textual data (LeCun, Bengio, & Hinton, 2015). Under certain conditions, traditional word similarity models can perform as well as word embeddings (Levy & Goldberg, 2014a).

2.2.4 Character-wise Encodings

A very low-level approach of textual input features is the usage of documents on character-level. Here the text is not preprocessed in any form. The sequence of text is fed as an input feature to the NN in a one-hot character-wise encoding.

An advantage of character-level features is that they are language independent. The dataset needs to provide enough data that semantic characteristics of the input language can be learned during training. Furthermore, no tools for textual preprocessing, like a tokenizer or lemmatizer, are needed.

Kanaris, Kanaris, Houvardas, and Stamatatos (2007) use *bag-of-character n-grams* for classifying spam in e-mail conversations. For this classification, just the content of the mail is used, without knowledge about the sender, other recipients, or an attachment is needed. Here they claimed, that the sparsity of data reduces when using character-level n-grams over n-grams on word-level. The reduced sparsity is an effect of the highly reduced number of character combinations that need to be set to zero. But the representation still has a significantly large feature set.

Characters are essential features for POS tagging. The goal of POS is to label a word according to its lexical categories, like noun, verb, pronoun or preposition. dos Santos, Nogueira and Zadrozny (2014) proposed a POS tagger that combines word- and character-level features of the input. Their architecture incorporates convolutional layers, which ex-

tracts character features from each word.

Zhang et al. (2015) explored how text classification can be applied on textual input in the format of characters by means of CNNs. They claim that their method could work without any need for features on word-level. Characters in the vocabulary are restricted to a certain set of 70 symbols, including 26 letters, 10 digits, 33 other symbols, and the newline character:

$$abcdefghijklmnopqrstuvwxyz0123456789$$
$$-,;.!?:"/\backslash|_@\#\$\%\hat{}\&*\sim' +- =<>()[]$$

The sequence of characters is then transformed into $1 - of - m$ (or "one-hot") encoding. Blank symbols (Padding) or characters in the sequence that do not occur in this vocabulary, are all set to zero in the transformed vector. This research discovered that on large datasets, not making a distinction between upper and lower case characters does perform better.

In contrast, Bojanowski et al. (2015) challenge RNNs with input on character level. First, they introduce an architecture where they combine character and word-level information. Second, an adapted RNN architecture (Char-RNN) is introduced to make the computation on character level more sparse.

A character-aware language model was introduced by Kim, Jernite, Sontag, and Rush (2016). This language model relies completely on character-based input. A Char-CNN leverages the subword information. The resulting output is fed into an RNN language model. Their implementation reduces the number of parameters by 60% and outperforms various baselines on morphologically rich languages.

A more recent work on NLI incorporates character-level information into the set of features. Gong et al. (2018) use a convolutional layer followed by a max pool layer (according to Zhang et al. (2015)) to extract character-level features. The set of features contains pre-trained word vectors, the character features, and syntactical features. The latter include POS tagging and binary exact match feature. In Wang, Hamza, and Florian (2017), a character-embedding is learned by a LSTM. The input is fed character-wise into the network. The embedding is then learned jointly with other network parameters.

3 Neural Network Models for Contradiction Detection

For the classification of NLI problems, several approaches are used. One of the earlier ones is a rule-based approach. The premise and hypothesis are first translated into a logical, formal form. The entailment classification is based on this representation. Examples for this approach can be found in Chatzikyriakidis and Bernardy (2017); Hickl and Bensley (2007); MacCartney and Manning (2007, 2009, 2014). The second one that is well discovered, is a graph-based approach. Here the prediction is formulated as a graph matching problem, where the two sentences are represented as graphs derived from syntactic dependency parses (Haghighi, Ng, & Manning, 2005). Both of these methods are not covered in this chapter.

Since the introduction of corpora with a lot of data, the interest in applying machine learning methodology to the problem of NLI has grown. From there on it was possible to use machine learning and neural models to find classifiers.

In this chapter, first, the general structure of a neural model with application to NLI will be discussed. As neural models are an assembly of several components, the application of NNs, introduced in Section 2.1, on NLI will be shown. Here the focus is set on the core components of the neural models in recent publications (Section 3.2, 3.3 and 3.4). This chapter closes with a discussion on the impact of network types in Section 3.5 and input features in Section 3.6 of neural models .

3.1 Adapted NLP Four-Step Approach

For their architecture, all neural NLP models tend to follow a certain scheme. Honnibal (2016) was the first to describe this scheme, the way it had already been used before. This scheme has four steps: *Embedding*, *Encoding*, *Attention* and *Prediction*.

Almost all recent publications on NLI propose models according to a certain scheme (Bowman, Angeli, Potts, & Manning, 2015; Conneau, Kiela, Schwenk, Barrault, & Bordes, 2017; Gong et al., 2018; Rocktäschel et al., 2015; Tay et al., 2018). In Figure 3.1 an adapted ver-

sion of the NLP scheme can be seen. This one is influenced by the the generic NLI model scheme (Conneau et al., 2017). The premise and hypothesis are on word-level, the classification according to the *three-way* labeling.

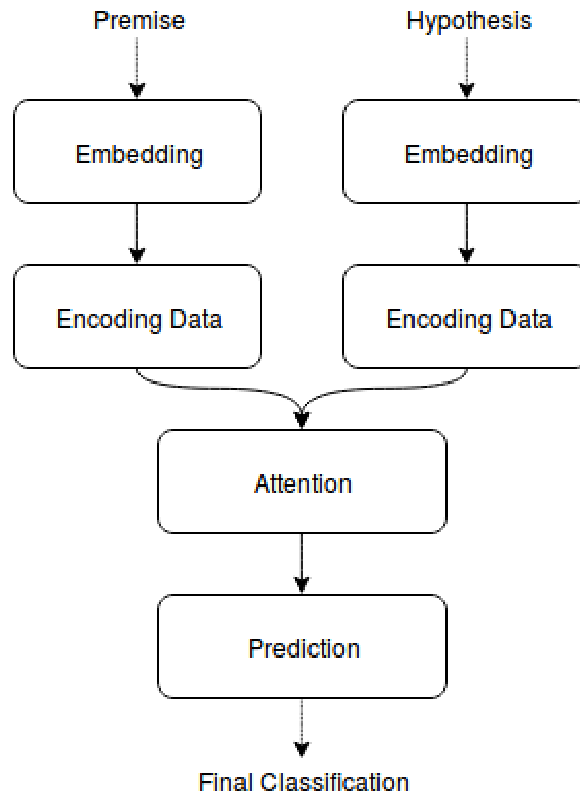


Figure 3.1: The NLP four-step approach according Honnibal (2016), adapted to the NLI problem. Premise and hypothesis are embedded and encoded separately. The attention forms a common representation on which the prediction is done. The adaption for NLI is based on the generic NLI model scheme (Conneau et al., 2017).

The *embedding* is done by converting the input word-wise into its word-vectors. This is done by means of word embeddings, as described earlier (Section 2.2.3). A very popular word-embedding for NLI are the GloVe. These are used in several recent publications on NLI (Q. Chen et al., 2017a, 2017b; Gong et al., 2018; Tay et al., 2018). Each word is then represented by an individual vector. Additional features can be appended to this. Gong et al. (2018) added additional features, like character features and syntactical features, including POS tags and binary exact match features, to increase their model's accuracy. This composition of input features is illustrated in Figure 3.2. This results in a matrix with the dimensions

$$d = n_W \times (l_e + l_f),$$

where n_W is the number of words in the sentence, l_e the length of the word-embedding and l_f the length of the additional features. In order that each sentence is represented in a matrix

with the same dimension, the sentence length is fixed to a certain length. Certain models (Gong et al., 2018) insist on a fixed length, where sentences longer than n_W are cropped and sentences shorter are filled up with padding. Other models (Tay et al., 2018) just pad the sentences to the length of the longest sentence in the batch.

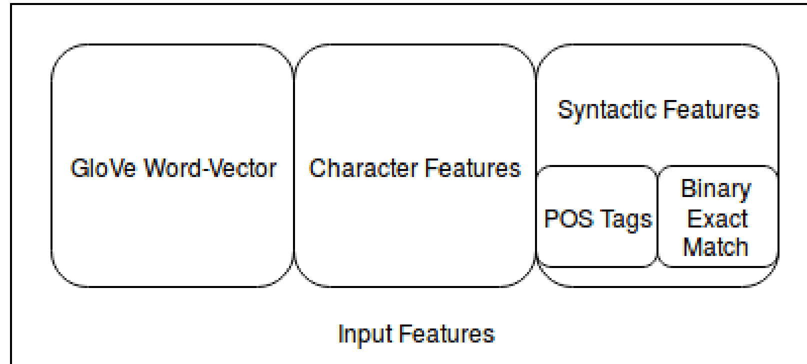


Figure 3.2: The input features, as used in the DENSELY INTERACTIVE INFERENCE NETWORK (DIIN) (Gong et al., 2018). First, for each word, the GloVe vectors are used. Character features, that are learned by a CNN, are appended. The final part of this input feature are syntactical features, including POS tags and binary exact match features.

Next, the embedded input will be *encoded* into a sentence representation. Till now, each word represents itself. This encoding step combines the embedded features into one representation. With the Stanford NLI (SNLI) corpus (Bowman et al., 2015) a simple baseline was released. For this, the authors compared a neural-based encoding (RNN and LSTM) with a simple averaging of the sentence's word embeddings. Here the LSTM outperformed in this competition.

LSTMs (Rocktäschel et al., 2015), BiLSTMs (Q. Chen et al., 2017b) or Highway Networks (Tay et al., 2018) are used neural network types for this step. This step is done for premise and hypothesis separately.

A crucial part is the *attention*. First introduced by Bahdanau, Cho, and Bengio (2015) for machine translation and adapted to NLI (Rocktäschel et al., 2015), the attention allows the model to attend over the past output. This enables to overcome the bottleneck of the LSTMs on its internal state. In case of an NLI system, the attention enables the cell not to need to capture the complete semantic information about the premise in its internal state. For that, it is only needed to pass on the output of the LSTM cell for the premise to the other for the hypothesis.

This *attention* step makes sure, that a final classification can be made. Therefore, the dimension of the previously encoded data must be reduced. For application on NLI, the two sentences are combined during this step as well. For this purpose, several methods are used

in literature. Q. Chen et al. (2017a) use a *Local Inference Modeling*. Tay et al. (2018) introduce a *Alignment Factorization*, where the alignment to the sentence itself and to the other is calculated. This is done by the alignment between the sentence itself and the other one. For this purpose, a factorization method $F_{fm}(x)$ (Rendle, 2010) of degree $d = 2$ is used

$$F_{fm}(x) = L(x) + P(x), \tag{3.1}$$

where

$$L(x) = w_0 + \sum_{i=1}^{n} w_i \cdot x_i$$

and

$$P(x) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} < \mathbf{v_i}, \mathbf{v_j} > \cdot x_i \cdot x_j.$$

$< .,. >$ indicates a dot product between two vectors, $w_0 \in \mathbb{R}$ is a global parameter, $\mathbf{w} \in \mathbb{R}^n$ and $\mathbf{V} \in \mathbb{R}^{n \times k}$. $F_{fm}(x)$ results in a scalar valued output. $\mathbf{v_i}$ is a row within $\mathbf{V}$ and describes the i-th variable with k factors, whereas $k \in \mathbb{N}_0^+$ defines the dimensionality of the factorization. $L(x)$ can be considered as a linear regression layer. $P(x)$ learns pairwise feature interactions. Therefore it factorizes the feature interaction matrix.

For combining the two sentences by means of heuristic, Mou et al. (2015) proposed an approach, where the result of the attention, the two properties $h1$ and $h2$ are assembled in to a vector m

$$m = [h1; h2; h1 - h2; h1 \cdot h2].$$

The "$-$" is the element-wise subtraction, $\cdot$ the element-wise multiplication and ; denotes a concatenation of two vectors. This technique is used to combine properties during the *attention* process.

An approach, that is slightly different from this scheme is introduced by Gong et al. (2018), where a combination of an *Interaction-* and *Feature Extraction Layer* is used. As the calculation of the *attention* cannot be done on the basis of a single sentence or on each sentence independently, the pair of sentences must be combined. This combination is part of the *attention* step.

The last step is the classification itself. For this, either fully-connected feed-forward NN (Q. Chen et al., 2017b; Gong et al., 2018; Tay et al., 2018), pooling-layer (Conneau et al., 2017) or soft-max layer (Q. Chen et al., 2017a) are used.

3.2 Feed-forward Neural Networks for NLI

In order to simply use a feed-forward architecture to classify a sentence pair, the sequence of the two sentences must be combined into a fixed length vector. Due to this possible cropping, information could get lost (Potts & Maccartney, 2016).

With the application in deep architectures, the feed-forward NN cannot be found in every model. A bunch of models, like the CAFE (Tay et al., 2018), use a simple n-layered feed-forward NN to classify the result of the attention layer. But, as discussed previously, other approaches can be used too.

The only work that was found, that completely relies on feed-forward NNs, is by Parikh, Täckström, Das, and Uszkoreit (2016). In this work, each word of the embedded premise and hypothesis is attended first using a feed-forward NN with ReLU separately. Attention weights are computed and afterwards compared with the initial representation. This comparison is also done by an NN of the same type. The results are aggregated and classified by a feed-forward NN with linear activation.

3.3 Recurrent Neural Network for NLI

Plain RNNs can be used in two different ways. A simple classifier is built by just using one RNN cell. The concatenated premise and hypothesis are fed into this one cell. On the other hand, two chained gates can be used. Here each sentence has a separate input and hidden weights. The RNN for the hypothesis is initialized by the state of the proceeding gates final state. It is also possible to combine the output of the two RNN cells, like a concatenation or other metrics like subtraction or multiplication (Mou et al., 2015; Potts & Maccartney, 2016).

As mentioned previously, in deep architectures RNNs, LSTMs more particularly, are often used during the encoding of the embedding into a sentence representation. For instance, CAFE (Tay et al., 2018) uses an LSTM right before the prediction. They encode the complete list of extracted features into one representation, that is then fed to a pooling layer. The result is then provided as input to the final classification.

Ghaeini et al. (2018) use for the same purpose a BiLSTM. An additional BiLSTM is used during the inference phase. Here two sequences of computed matching vectors from the attention phase are aggregated. The first to use LSTMs with an adaption on neural attention were Rocktäschel et al. (2015).

In recent publications (Gong et al., 2018; Yu et al., 2017) the usage of LSTMs was being avoided. Compared to other approaches, RNN methods are very time consuming, as the states of the memory cells and the attention weights must be computed at every time step (Yu et al., 2017).

3.4 Convolutional Neural Network for NLI

In general, CNN based models can be divided into two categories: sentence encoding (SE)-based or sentence interaction(SI)-based methods. The first type's aim is basically to learn a good representation of each sentence and apply a comparison function, which transforms both sentences into one single representation (Mou et al., 2016; Yin & Schütze, 2015). The second one directly models the interaction between the two sentences at the beginning and results in a final representation on top of this output (Hu, Lu, Li, & Chen, 2014; Pang et al., 2016).

Yu et al. (2017) analyzed both methodologies and combined them into one model. Their hybrid CNN model (hCNN) combines an SE-based model and an SI-based Pyramid model. The combination is performed by simply concatenating the results of the two models before further analysis is performed.

A further model that includes an SE-based approach, was introduced by Gong et al. (2018). The so-called DIIN model first creates a representation of the sentence. Additionally, another convolutional layer is used in this model. This layer aims to extract features from the result of the interaction layer. For this purpose a special network model is used, the so-called DENSENET (Huang, Liu, van der Maaten, & Weinberger, 2017).

DENSENET was introduced by Huang et al. (2017). With this type of network, the authors presented an approach, which connects each layer of the network with each layer to every other layer in a feed-forward style. This approach enables networks to be much deeper, but also improves accuracy and efficiency. The input of a layer are the feature-maps of the connected ones.

CNNs can also be used to pre-process data. As shown in Gong et al. (2018) and Tay et al. (2018), character features are extracted and appended to the list of input features. This preprocessing step is done similar to Zhang et al. (2015).

Model	Development Accuracy
DIIN (Gong et al., 2018)	88.0%
CAFE (Tay et al., 2018)	88.5%

Table 3.1: Comparison of DIIN (Gong et al., 2018) and CAFE (Tay et al., 2018) on SNLI (Bowman et al., 2015), as reported.

Model	MATCHED	MISMATCHED
DIIN (Gong et al., 2018)	78.8%	77.8%
CAFE (Tay et al., 2018)	78.7%	77.9%

Table 3.2: Comparison of DIIN and CAFE on the MULTI-GENRE NLI (MULTINLI) (Williams et al., 2017) development sets, as reported.

3.5 Effect of Network Types on Neural Models

As all neural models for NLI are a composition of several different layers. It is hard to compare the effect of a specific network type to the model. In order to explain the effects of certain network types, two recent models with different architectural components were selected. The DIIN (Gong et al., 2018) incorporates a convolutional structure for extracting the features. In comparison, the CAFE (Tay et al., 2018) model uses an LSTM for the same task. Booth models use the same types of input features and a three-way classification into *entailment* / *neutral* / *contradiction*.

The accuracy that these models achieve on the SNLI and MultiNLI corpus can be seen in Table 3.1 and 3.2, respectively. Here is shown, that the LSTM based CAFE model performs 0.5% better on the SNLI dataset, compared to the CNN based DIIN . Bot comparisons exhibit almost negligible (in comparison to variance in the single experiment) differences. Each of the models perform better on about 0.1% on one of the development sets.

The effect of certain components on a neural model can be shown, by replacing certain parts from the architecture. For the CAFE network, this ablation study is reported in Table 3.3.

As mentioned in Section 3.1, CAFE (Tay et al., 2018) introduces an *Alignment Factorization*, as described in Equation (3.1). In the first experiment, this factorization method F_{fm} is replaced by regular fully connected layers. The result of 77.7% for the matched and 77.9% for the mismatched development set shows the marginal impact of this method.

A more noticeable difference in the accuracy can be seen, when the intra-attention layer is being removed (Experiment 2 in Table 3.3). This layer aims to learn an alignment of sub-phrases between the premise and hypothesis. The difference from the original model of 3.8%/3.3% show the importance of this layer.

#	Experiment Name	MATCHED	MISMATCHED
	Original CAFE Model	79.0%	78.9%
1	replacing factorization method with fully-connected layer	77.7%	77.9%
2	removing inter attention	75.2%	75.6%
3	replace highway layer prediction with fully-connected layer	77.7%	77.9%
4	replace highway encoding with fully-connected layer	78.7%	78.7%
5	replace LSTM with a bidirectional LSTM	78.3%	78.4%
6	removing Character Embedding	78.1%	78.3%
7	removing syntactical Embedding	78.3%	78.4%

Table 3.3: Ablation study for CAFE on MultiNLI development sets. The accuracy of the adaption is reported in comparison to the original model (Tay et al., 2018).

#	Experiment Name	MATCHED	MISMATCHED
	Original DIIN Model	79.2%	79.1%
8	removing convolutional feature extractor (DENSENET)	73.2%	73.6%
9	removing encoding layer	73.5%	73.2%
10	removing self-attention and fuse gate	77.7%	77.3%
11	removing fuse gate	73.5%	73.8%

Table 3.4: Ablation study for DIIN on MultiNLI development sets. The accuracy of the adaption is reported in comparison to the original model (Gong et al., 2018).

Experiment 3 and 4 show how the highway layer has a positive influence on the accuracy of this network. Here the highway network of the prediction layer and for encoding of the input is replaced by fully-connected layers. The results show a drop of 1.2%/1.0% for the prediction layer (Experiment 3). The difference for experiment 4 (replacement of highway network for the encoding) is only marginal with 0.3%/0.2%.

For extraction of the final features before the prediction, CAFE uses an LSTM followed by a max- and avg-pooling layer. In a further experiment 5, this used LSTM is replaced by a bidirectional LSTM. By changing this one component exclusively in this layer, accuracy drops about 0.7%/0.5%.

The ablation study for the DIIN (Gong et al., 2018) can be seen in Table 3.4. For experiment 8, the convolutional feature extractor, based on DENSENET (Huang et al., 2017), is being replaced by a max-pooling of the sentence representation over time. The feature vectors for premise p and hypothesis h are then combined by $[p; h; p-h; p \cdot h]$ into the final feature vector for classification. The result of this experiment is comparable to other models, that are sentence encoding based (73.2%/73.6%).

The absence of the encoding layer is analyzed in experiment 9. When removing the whole layer, the accuracy drops to 73.5%/73.2%. Similar for experiment 10, the self-attention layer + fuse gate were removed. Experiment 11 just analyses the influence of the removed fuse gate. These experiments result in 77.7%/77.3% and 73.5%/73.8% respectively. As can be seen, interestingly the accuracy is better if both self-attention and fuse gate are removed, compared to just removing the fuse-gate.

3.6 Effect of Input Features on Neural Models

Similar to Section 3.5, a comparison can also be made for the input features of a network. For this analysis, the results of the CAFE network (Tay et al., 2018) are being considered.

This network uses, inspired by the DIIN (Gong et al., 2018), a concatenation of input features. As already discussed in Section 3.1 and illustrated in Figure 3.2, the list consists of GloVe word-vectors for each word, followed by character features obtained by an CNN and syntactical features. The results for the experiments on the CAFE network, can be seen in Table 3.3.

Experiment 6 and 7 show the effect of the character and syntactical features, respectively. The results for experiment 6 show that the absence of the character features drops the accuracy down to 78.1%/78.3%, which is a marginal reduction of 0.9%/0.6% to the original model. The impact of the syntactical features is lower, compared to the character features. With their absence, the model poses an accuracy of 78.3%/78.4% accuracy (difference of 0.7%/0.5%).

4 Exploring Contradiction with Neural Models for Natural Language Inference

As mentioned earlier, the era of NN with an application on NLI started with the introduction of larger corpora. Formerly, mostly non-neural approaches were used. Since then, the interest in applying neural methods rose (Bowman et al., 2015; Williams et al., 2017). Following this trend, the question arises, in what kind of form should the input be presented?

In the following chapter, an analysis of several kinds of inputs of neural models is conducted. Firstly, Section 4.1 describes the model used. In Section 4.2 the setup for the experiments is explained. The experiments are described in Section 4.3, where the effect of different types of input features are shown.

4.1 The Neural Model

A novel neural approach is introduced with the DENSELY INTERACTIVE INFERENCE NETWORK (DIIN) (Gong et al., 2018). The DIIN is a multi-layer network, predicting the entailment of two sentences, premise and hypothesis. The prediction is done according to a three-way classification into "Entailed", "Neutral" or "Contradicting".

The DIIN is an attention-based model. The architecture is split into five layers, and designed according to the generic NLI scheme (see Section 3.1). Each layer of the architecture is named after its responsibilities, namely embedding, encoding, interaction, feature extraction, and output layer.

At the embedding layer, each sentence, premise, and hypothesis separate, is converted into a concatenation of word embedding, character feature, and syntactical features. For the word-embedding, the pre-trained GloVe is used. The character-embedding features for each word are obtained by a convolutional network, followed by a max pooling over the inputs character vectors. The syntactic feature set contains one-hot POS tagging and binary exact match features. The latter indicates the index of the shared words.

The following encoding layer takes the concatenated list of embeddings and passes it

through a two-layered highway network. An intra-attention is applied on to the new representation. This one enables the network, to take the word order and context information into account. This is done for premise and hypothesis separately.

A common representation is formed by the Interaction Layer. The encoded representation of premise (P^{enc}) and hypothesis (H^{enc}) result in

$$I_{ij} = \beta(P_i^{enc}, P_j^{enc}) \in \mathbb{R}^d, \forall i \in [1, ..., p], \forall j \in [1, ..., h],$$

where P_i^{enc} denotes the i-th row vector of P^{enc}, and H_j^{enc} the j-th row vector of H^{enc} respectively. β is solely implemented as the element-wise product

$$\beta(a,b) = a \cdot b,$$

instead of the format used in other models and explained before (Section 3.1).

As already mentioned in Section 3.4, for the feature extraction, a DENSENET is used. The flattened result is classified by a linear layer using a three-way classification.

The reported results with this model on the SNLI (88.0%) and on MultiNLI (78.8%|77.8%) demonstrates its power in the competition.

4.2 Experimental setup

For the DIIN , the authors implementation is available online[1]. For the purpose of the experiments, this implementation was used without modifications. As the optimization algorithm, an Adadelta optimizer is used, with $\rho = 0.95$ and $\varepsilon = 1e-8$.

The learning rate is set initially to 0.5 and the batch-size is chosen to be, as in the reference implementation (Gong et al., 2018), to 70. Dropouts are used after the word-embedding and before each linear layer. In this implementation, a decay rate is applied for the dropout keep rate. First, 1.0 is used, after every 10,000 steps, this rate is multiplied by 0.977. For the word-embedding, pre-trained GloVe vectors are used. The 300D GloVe 840B is being used. Out-of-vocabulary words are initialized randomly, as well as the character embedding. All weights are set according to a $L2$ regularization. For further details on the implementation of this regularization, the reader is referred to Gong et al. (2018). For the training of the models, no data from the SNLI corpus was used, as done in Gong et al. implementation. The training completely relies on the MultiNLI training data.

[1] `https://github.com/YichenGong/Densely-Interactive-Inference-Network`

The used implementation of the DIIN (Gong et al., 2018) did not provide exactly the same results, as reported. The accuracy that was achieved is 77.90%|77.03%, compared to the reported 78.8%|77.8%. This difference is probably caused as the reference implementation uses 15% of the training data from the SNLI corpus. The achieved accuracy was used as the baseline in all the experiments.

4.3 Experiments

In this section the experiments are discussed.Different types of input features applied to the DIIN model, are applied and discussed.

As described in Section 2.2, there are several ways of supplying a model with input data. Different kinds of input features have different effects on the prediction. An example can be seen in Section 3.5, where the ablation study for the CAFE network shortly analyzes the impact of the different kinds of input features. In this section, the analysis will become deeper and consider different kinds of input formats.

As already described in Section 3.1, the DIIN uses three types of input features: for each word in the sentence the word embedding is used with character- and syntactical-features, including POS tags and binary exact match features. This, in the following, also called original feature-set, is used as the baseline (see Table 4.1) in the following experiments.

The first comparison (Experiment 1) is done by exclusively considering the used GloVe word-embeddings. For experiment 2, the character features, that are learned through a CNN are skipped. Experiment 3 uses the original feature-set without the POS tagging and binary exact matching features.

A different set of features, that has so far not been used in the studies, is applied for Experiment 4. Here the input is used on character-level. Similar to Zhang et al. (2015), each character in the input is converted into a one-hot encoded vector, with the length of the vocabulary. For this experiment, a reduced vocabulary is used, compared to Zhang et al. (2015). This consists of 53 characters, including 26 letters, 10 digits, 18 other symbols:

$$abcdefghijklmnopqrstuvwxyz0123456789$$
$$-,;.!?:"/'\$\%()[]$$

Other characters, including the space, are set to all zero. The selection was made by using all the characters in the train and development-set of the MultiNLI corpus, that have an

occurrence higher than 1000. The length of sentences is restricted to 278 characters. This number was chosen, as this length covers exactly the same amount of sentences in the whole MultiNLI corpus as used in the default setting on word-level. Sentences that are longer than the 278 characters are cropped and sentences that are shorter are padded with all zero vectors. Following these two constraints, the resulting input matrix for each sentence has the dimensions 278×53. This causes a sparsity level of $\frac{278}{278 \cdot 53} = 1.887\%$, whereas the original setup has one of roughly 100%.

In order to conduct this experiment, the number of layers in the self-attention encoding was set to zero. As this layer intends to calculate the similarity between each word in a sentence, this does not apply to character-level. In the case of this experiment, one feature on character-level has only a probability of $\frac{1}{53}$, as this is on word-level, depending on the used vocabulary, lower than $\frac{1}{640\ billion}$.

With these adoptions, the model still needs to do a convolution on the vocabulary in a two-dimensional space. As the vocabulary increases about five times, the need for memory grows quadratically. In order to still compute this memory intense experiments, the batch size was reduced to 18.

Experiment 5 is a modification of experiment 4, with fewer characters used. All non-alphabetic and non-numeric characters are omitted, so that just

$$abcdefghijklmnopqrstuvwxyz0123456789$$

are used.

As discussed before in Section 2.2.1, each feature in a one-hot encoded dataset is a own dimension. This causes it to consume a lot of memory. For experiment 6, the previously characterized input data is altered. In order to make the input sparse data representation easier to compute, an autoencoder is applied to the data to reduce dimensions. During this step, the dimensions of the data are reduced and another representation is learned. The autoencoder uses one feed-forward hidden layer. The hidden layer has a dimension of 50 and the output of 20. For each one-hot vector, the two subsequent vectors are taken, concatenated and fed into the autoencoder. The input of the autoencoder results in a vector with 159 dimensions. The resulting matrix has the dimensions of 276×20.

For the last experiment, the highway-layer is removed. Here it is assumed, that this layer learns a representation of the input data for this classification task. To still be able to learn a representation, the GloVe embedding used was set to be trainable.

#	Experiment Name	MATCHED	MISMATCHED
	original feature-set	77.90%	77.03%
1	only GLOVE word embedding	77.09%	76.36%
2	without character features	77.83%	77.29%
3	without syntactic features	77.42%	76.88%
4	sparse character-level features	65.12%	66.00%
5	sparse character-level features (less characters)	65.09%	65.87%
6	less-sparse character-level features	65.81%	66.38%
7	no highway-layer and updating word embedding	75.81%	75.81%

Table 4.1: Applying the the MULTINLI (Williams et al., 2017) development-sets in different formats of the Input Features on to the DIIN (Gong et al., 2018) network. All the experiments are done on the original model, with only modifying the input or as stated.

The results for all the experiments can be seen in Table 4.1. As mentioned in Section 4.2, it was not possible to get the reported accuracy, even with the original implementation of the DIIN . The maximal achieved accuracy is reported as the baseline with the original feature set.

Disabling all features and using solely the word-embedding, lets the accuracy drop to about $-0.81\%|-0.67\%$. Without the usage of the character embedding, the accuracy changes to $77.83\%|77.29\%$. The absolute difference is $-0.07\%|+0.26\%$ and means an increase in accuracy for the *mismatched* test set. Ignoring the syntactical features, the accuracy changes to $77.42\%|76.88\%$, which means a difference of $-2.48\%|-0.15\%$.

A significant drop in the accuracy can be observed when using one-hot character encoding. The resulting accuracy of $65.12\%|66.00\%$ is about $-12.78\%|-11.03\%$ lower, compared to the baseline with the original feature set. The training of the model with this set of features did take roughly about four times the training time of the DIIN baseline with the full feature list. However, it has to be considered, that the pre-training of the embedding models used should be taken into account for a fair comparison.

The results are very similar for experiment 5. With an accuracy of $65.09\%|65.87\%$, it is $-0.03\%|-0.13\%$ lower compared to experiment 4. This shows that the reduction of the vocabulary has only a little effect on the prediction. The difference of the baseline with the whole feature-set is $-12.81\%|-11.16\%$.

Trying to minimize the sparsity of the data from the character-level encoding, an autoencoder is applied for experiment 6. This causes a drop in the accuracy to $65.81\%|66.38\%$.

The result of experiment 7 shows the impact of setting the embeddings trainable and re-

moving the highway layer. Here the accuracy drops, compared to the baseline, about $-2.09\%|-1.22\%$. This result shows, that the highway layer creates another and better representation that boosts the performance of the classification task. Whereas it seems that the highway layer is capable of creating a representation, the word embedding is already biased in a way. The retraining does not perform as well, as the intermediate representation.

In order to judge the previously trained input features on the DIIN in terms of linguistic, a further analysis of the previously reported results was conducted.

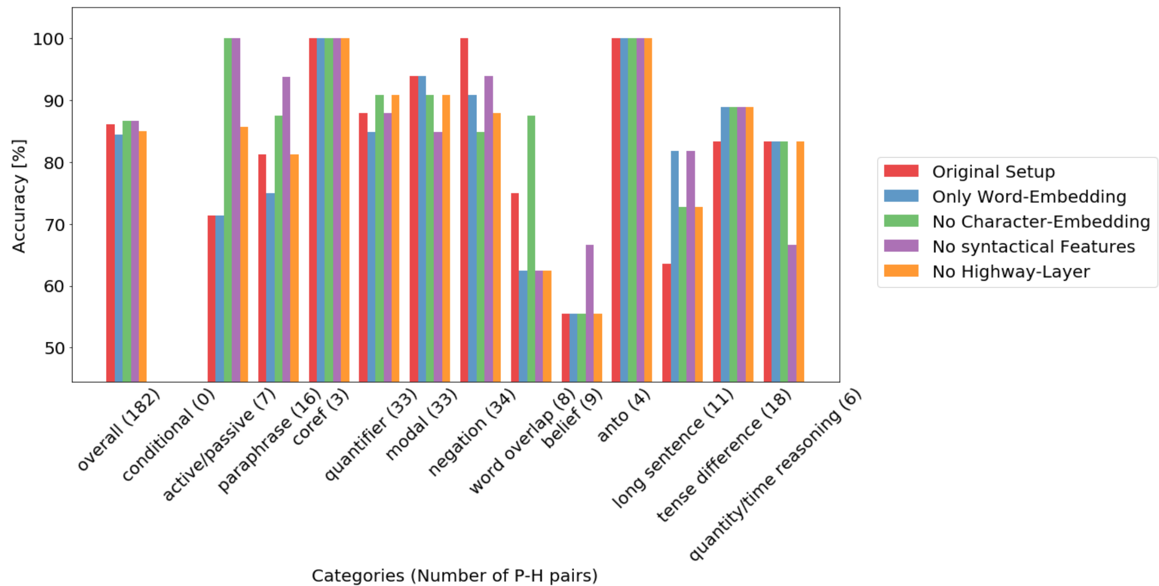
For the MultiNLI annotations exist, based on linguistic phenomena. These annotations indicate conditionals, active/passive, paraphrasing, coreferences, quantifiers, modals, belief, overlapping words, negations, antonyms, long sentences, tense differences and quantity or time reasoning. Whereas 1731 sentences of the MultiNLI development-sets are annotated with labels, just 340 of those only have one. These are split up into 182 for the matched, and 158 for the unmatched development-set. In the following experiment, only those 340 sentences are being considered, that have exclusively one tag. Otherwise, it would be not possible to lead back a classification decision to a certain phenomenon. In the other case, an entailment or contradiction decision could be based on a combination of the annotations.

The results can be seen in Figure 4.1 and 4.2 for word-level and character-level features, respectively.

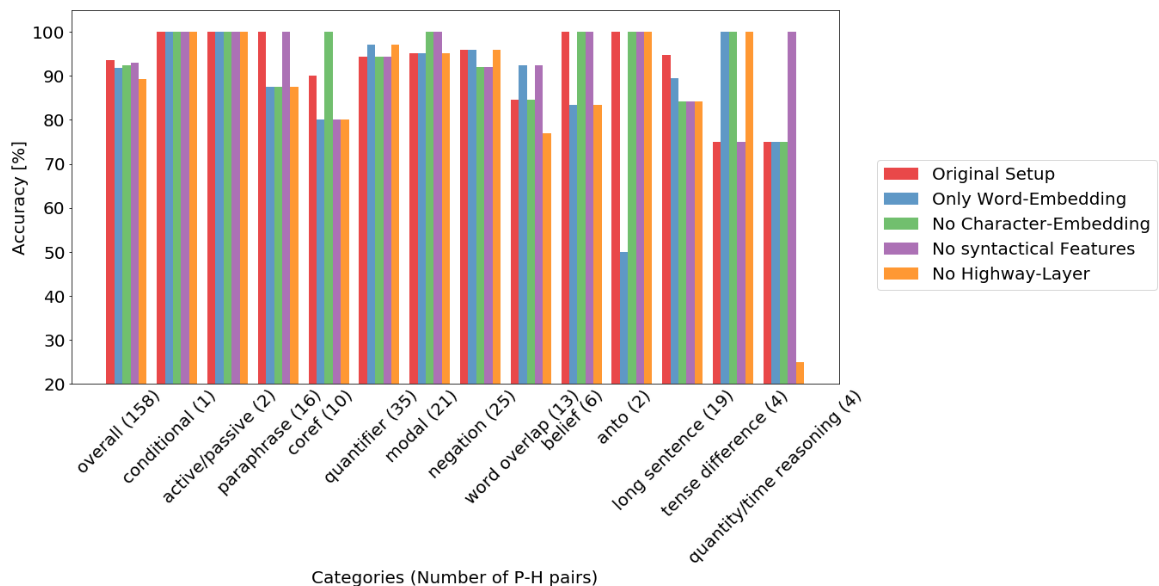
It can be seen with the overall results, that the model trained with no character features and no syntactical features perform roughly the same as the baseline. The model with only word embedding does just slightly worse and the model with one-hot character features about 8% worse. The same applies for the *mismatched* development-set.

In case of the *active/passive* annotations, the feature-sets with no character-embedding and no syntactical features perform the best. The *negation* profits the most from the original feature-set. By considering the *word overlap*, the set without character-feature tends to perform the best. Here, the classification really benefits from using either the syntactical features or the character-embedding, but both contradict each other. By removing the Highway Layer, similar results to the original setup are achieved. A significant difference in favor this setup can only be observed for *active/passive* and *long sentence* on the *matched* and for *tense difference* for the mismatched development set.

The one-hot character embedding performs similarly to the other models on word-level or worse. An example, where this feature captures a phenomenon significantly worse, is paraphrasing. It is assumed, as word-embeddings are close for similar words, this relation can be captured. As this information is missing, the performance on this task is worse. On

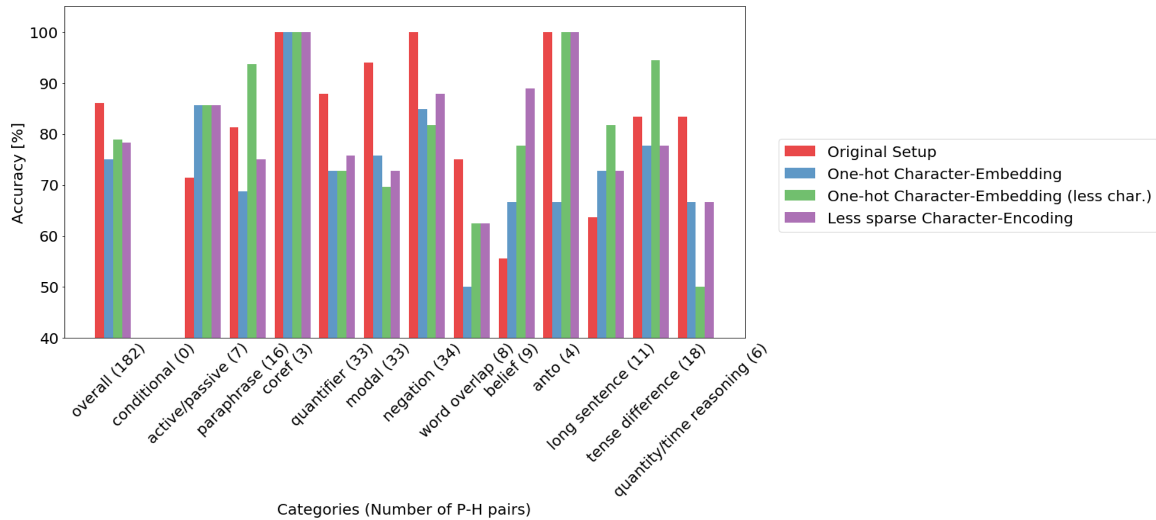


(a) Annotation Types n MultiNLI Development Matched for each of the categories of input features

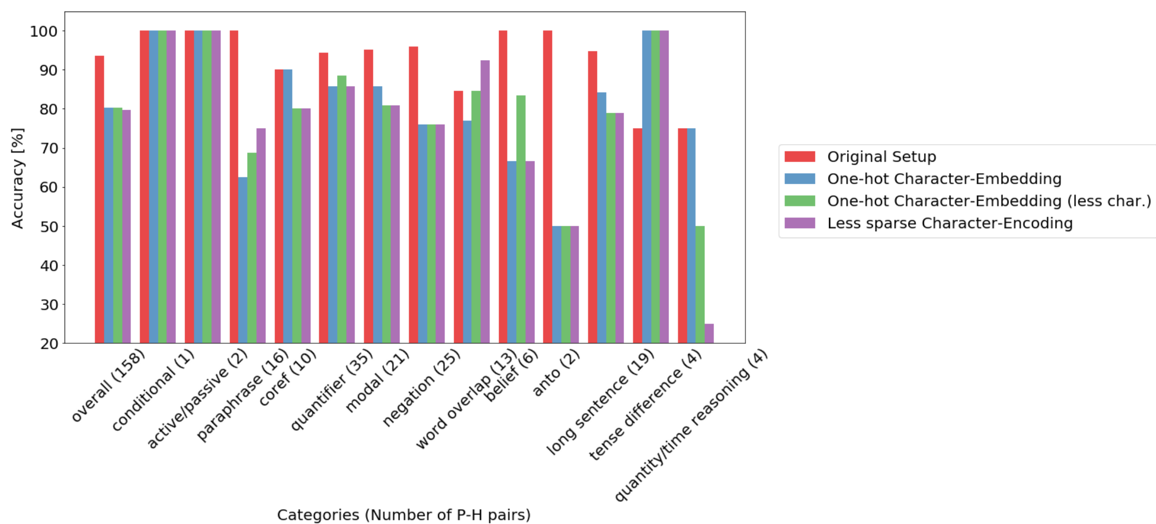


(b) Annotation Types n MultiNLI Development Mismatched for each of the categories of input features

Figure 4.1: Results for the analysis of linguistic phenomena on the MultiNLI dataset on each word-level feature set of the DIIN . 4.1a shows the results for the *matched*, and 4.1b for the mismatched development-set. The numbers in the brackets in the categories indicate the number of sentence pairs that are being considered for this analysis.



(a) Annotation Types n MultiNLI Development Matched for each of the categories of input features



(b) Annotation Types n MultiNLI Development Mismatched for each of the categories of input features

Figure 4.2: Results for the analysis of linguistic phenomena on the MultiNLI dataset on each character-level feature set of the DIIN . 4.2a shows the results for the *matched*, and 4.2b for the mismatched development-set. The numbers in the brackets in the categories indicate the number of sentence pairs that are being considered for this analysis.

the other hand, this feature-set and the original feature-set on the *coref* and *long sentence* annotations perform equally.

As can be seen in Figure 4.2a, all the experiments on character-level perform similarly. The model with less characters can outperform the others of its kind on *paraphrasing* and *tense difference* on the *matched* and on belief on the *mistmachted* development set. In only three categories, the less-sparse approach is able to achieve an better accuracy: *quantifier*, *negation* and *belief* (matched) and *paraphrase* and *word overlap* (mismatched).

5 Conclusion

This report concludes the author's research conducted from March to August 2018 at the Center for Intelligent Information Retrieval at the University of Massachusetts, Amherst. The initial goal was to take a look at different types of input encoding of the textual data used in novel NLP approaches. In the survey conducted, several types of input encodings and input features were selected and compared. Here, a more detailed look was taken on character-based encodings.

For the analysis of different types of input features, the DIIN model was used. The experiments showed that the selected features, as used with the model, clearly provide the best results on the training data. Adding a learned character-embedding and syntactical features pose a minor gain in the accuracy. Using only a one-hot character-embedding performs significantly worse on the overall results. The analysis of the linguistic phenomena shows that the character-embedding performs similarly to the one on word–level.

In a future work, it would be interesting to analyze the reason for the gap of the accuracy. Here the effect of an increased size of training data would be interesting. It is assumed that a better accuracy would be possible with much more training data. Furthermore, the resulting changes to the analysis of linguistic phenomena would give interesting insights.

Abbreviations

BiLSTM	Bidirectional LSTM
BP	backpropagation
CEC	constant error carrousel
CNN	Convolutional Neural Network
DIIN	Densely Interactive Inference Network
GloVe	Global Vectors
LSTM	Long-Short Term Memory
MultiNLI	Multi-Genre NLI
NLI	Natural Language Inference
NLP	Natural Language Processing
NN	Neural Network
POS	part-of-speech
ReLU	Rectifier Linear Unit
RNN	Recurrent Neural Network
RTE	Recognizing Textual Entailment
SNLI	Stanford NLI
SUAS	Salzburg University of Applied Sciences

List of Figures

2.1 A simple three layer NN, consisting of four input nodes, four hidden nodes, and three output nodes. 4
2.2 Different *transfer functions* as defined in Equation (2.3), (2.4), (2.5), and (2.6) respectively. 5
2.3 A simple Neuron of a NN, consisting of an input vector x, an vector with the weights w and a bias b_0. This illustration's formal definition can be found in Equation (2.1) and Equation (2.2). 6
2.4 The simple structure of RNN. Figure 2.4a shows a simple gate. The t_r indicates the delay of one step. This form can be also displayed in an unfolded form, as can be seen in Figure 2.4b 9
2.5 A simple LSTM cell, which embodies a CEC as the core and the activations g and h in its architecture (Hochreiter & Schmidhuber, 1997). 11
2.6 An adapted LSTM cell that embodies an additional forget gate. This gate is learned to trigger to reset the internal state to zero (Gers et al., 1999). 13
2.7 An adapted LSTM cell that embodies a *forget gate* and *peepholes*. The input gates are connected to the internal state to get information about the cells current state (Gers et al., 2003). 15
2.8 The network architecture for LeNet-5 with convolutional-, sub-sampling-, fully connected- and Gaussian connected layers. The networks purpose is to classify handwritten digits (LeCun et al., 1998). 17
2.9 Graphical representation of the *max pooling* and the *average pooling*. Each non-overlapping window is shown with its own color. The results are shown in the corresponding fields, respectively. 18
2.10 The ReLU function shown in example. The function $a(z) = max(0,z)$ is applied to every element. 19
2.11 The comparison of the usage of sparse (a) to dense (b) vectors. (a) shows the concatenation of three "one-hot" vectors, describing the phrase *I like food*. (b) represents the concatenation of the embeddings in a figurative example with 6 dimensions of the same two words as in (a) (Goldberg, 2017). 22
2.12 The usage of the CBOW (left) and SKIP-GRAM (right) applied on the WORD2VEC embedding model Mikolov et al. (2013). 23

3.1 The NLP four-step approach according Honnibal (2016), adapted to the NLI problem. Premise and hypothesis are embedded and encoded separately. The attention forms a common representation on which the prediction is done. The adaption for NLI is based on the generic NLI model scheme (Conneau et al., 2017). 27
3.2 The input features, as used in the DIIN (Gong et al., 2018). First, for each word, the GloVe vectors are used. Character features, that are learned by a CNN, are appended. The final part of this input feature are syntactical features, including POS tags and binary exact match features. 28

4.1 Results for the analysis of linguistic phenomena on the MultiNLI dataset on each word-level feature set of the DIIN . 4.1a shows the results for the *matched*, and 4.1b for the mismatched development-set. The numbers in the brackets in the categories indicate the number of sentence pairs that are being considered for this analysis. 41
4.2 Results for the analysis of linguistic phenomena on the MultiNLI dataset on each character-level feature set of the DIIN . 4.2a shows the results for the *matched*, and 4.2b for the mismatched development-set. The numbers in the brackets in the categories indicate the number of sentence pairs that are being considered for this analysis. 42

List of Tables

3.1 Comparison of DIIN (Gong et al., 2018) and CAFE (Tay et al., 2018) on SNLI (Bowman et al., 2015), as reported. 32
3.2 Comparison of DIIN and CAFE on the MULTINLI (Williams et al., 2017) development sets, as reported. 32
3.3 Ablation study for CAFE on MultiNLI development sets. The accuracy of the adaption is reported in comparison to the original model (Tay et al., 2018). 33
3.4 Ablation study for DIIN on MultiNLI development sets. The accuracy of the adaption is reported in comparison to the original model (Gong et al., 2018). 33

4.1 Applying the the MULTINLI (Williams et al., 2017) development-sets in different formats of the Input Features on to the DIIN (Gong et al., 2018) network. All the experiments are done on the original model, with only modifying the input or as stated. 39

Bibliography

Bahdanau, D., Cho, K., & Bengio, Y. (2015). Neural machine translation by jointly learning to align and translate. In Proceedings of ICLR. San Diego, California.

Bengio, Y., Ducharme, R., Vincent, P., & Janvin, C. (2003). A neural probabilistic language model. The Journal of Machine Learning Research, 3, 1137–1155.

Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. IEEE Transactions on Neural Networks, 5(2), 157–166. doi: 10.1109/72.279181

Bishop, C. M. (1995). Neural networks for pattern recognition. Oxford, New York: Clarendon Press Oxford University Press.

Bodén, M. (2002). A guide to recurrent neural networks and backpropagation (Tech. Rep.). doi: 10.1.1.16.6652

Bojanowski, P., Joulin, A., & Mikolov, T. (2015). Alternative structures for character-level rnns. CoRR, abs/1511.0.

Bowman, S. R., Angeli, G., Potts, C., & Manning, C. D. (2015, August). A large annotated corpus for learning natural language inference. In Proceedings of the 2015 conference on empirical methods in natural language processing. Association for Computational Linguistics.

Chatzikyriakidis, S., & Bernardy, J.-P. (2017). A type-theoretical system for the FraCaS test suite: Grammatical framework meets coq. In Proceedings of IWCS. Association for Computational Linguistics.

Chen, M. (2017). Efficient vector representation for documents through corruption. In 5th ICLR.

Chen, Q., Zhu, X., Ling, Z.-H. Z., Wei, S., Jiang, H., & Inkpen, D. (2017a, September). Enhanced lstm for natural language inference. In Proceedings of the 55th annual meeting of the association for computational linguistics (pp. 1657–1668). Association for Computational Linguistics. doi: 10.18653/v1/P17-1152

Chen, Q., Zhu, X., Ling, Z.-H. Z., Wei, S., Jiang, H., & Inkpen, D. (2017b). Recurrent neural network-based sentence encoder with gated attention for natural language inference. CoRR, abs/1708.0.

Collobert, R., & Weston, J. (2008). A unified architecture for natural language processing. In Proceedings of the 25th international conference on machine learning (pp. 160—-

167). New York, New York, USA: ACM Press. doi: 10.1145/1390156.1390177

Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., & Kuksa, P. (2011). Natural language processing (almost) from scratch. Journal of Machine Learning Research, 12, 2493–2537. doi: 10.1.1.231.4614

Conneau, A., Kiela, D., Schwenk, H., Barrault, L., & Bordes, A. (2017). Supervised learning of universal sentence representations from natural language inference data. CoRR, abs/1705.0.

dos Santos, Nogueira, C., & Zadrozny, B. (2014). Learning character-level representations for part-of-speech tagging. In Proceedings of the 31st international conference on international conference on machine learning (pp. II–1818). JMLR.org.

Ferner, C., Pomwenger, W., Wegenkittl, S., Schnöll, M., Haaf, V., & Keller, A. (2017). Information extraction engine for sentiment-topic matching in product intelligence applications bt - data science – analytics and applications. In P. Haber, T. Lampoltshammer, & M. Mayr (Eds.), (pp. 53–57). Wiesbaden: Springer Fachmedien Wiesbaden.

Gers, F. A., Schmidhuber, J., & Cummins, F. (1999). Learning to forget: Continual prediction with lstm. NEURAL COMPUTATION, 12, 2451—-2471.

Gers, F. A., Schraudolph, N. N., & Schmidhuber, J. (2003). Learning precise timing with lstm recurrent networks. The Journal of Machine Learning Research, 3, 115–143. doi: 10.1162/153244303768966139

Ghaeini, R., Hasan, S. A., Datla, V., Liu, J., Lee, K., Qadir, A., ... Farri, O. (2018, February). Dr-bilstm: Dependent reading bidirectional lstm for natural language inference. CoRR, abs/1802.0.

Globerson, A., Chechik, G., Pereira, F., & Tishby, N. (2007). Euclidean embedding of co-occurrence data. Journal of Machine Learning Research, 8(Oct), 2265–2295.

Goldberg, Y. (2017). Neural network methods for natural language processing (Vol. 10; U. o. T. Graeme Hirst, Ed.) (No. 1). San Rafael, California: Morgan & Claypool Publishers. doi: 10.2200/S00762ED1V01Y201703HLT037

Gong, Y., Luo, H., & Zhang, J. (2018). Natural language inference over interaction space. International Conference on Learning Representations.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT Press.

Haghighi, A., Ng, A., & Manning, C. (2005). Robust textual inference via graph matching. In Proceedings of human language technology conference and conference on empirical methods in natural language processing (hlt/emnlp) (pp. 387–394). Vancouver: Association for Computational Linguistics.

Hickl, A., & Bensley, J. (2007). A discourse commitment-based framework for recognizing textual entailment. In Proceedings of the ACL-PASCAL workshop on textual entailment and paraphrasing (pp. 171–176). Prague: Association for Computational

Linguistics.

Hochreiter, S. (1991). Untersuchungen zu dynamischen neuronalen netzen (Unpublished doctoral dissertation). Technische Universität München.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. Neural Computation, 9(8), 1735–1780.

Honnibal, M. (2016). Embed, encode, attend, predict: The new deep learning formula for state-of-the-art nlp models. Retrieved 2018-05-31, from `https://explosion.ai/blog/deep-learning-formula-nlp`

Hu, B., Lu, Z., Li, H., & Chen, Q. (2014). Convolutional neural network architectures for matching natural language sentences. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, & K. Q. Weinberger (Eds.), Advances in Neural Information Processing Systems 27 (pp. 2042–2050). Curran Associates, Inc.

Huang, G., Liu, Z., van der Maaten, L., & Weinberger, K. Q. (2017, July). Densely connected convolutional networks. In 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (pp. 2261–2269). Honolulu, HI, USA. doi: 10.1109/CVPR.2017.243

Joulin, A., Grave, E., Bojanowski, P., & Mikolov, T. (2017). Bag of tricks for efficient text classification. Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics, 2, 427—-431.

Kanaris, I., Kanaris, K., Houvardas, I., & Stamatatos, E. (2007). Words versus character n-grams for anti-spam filtering. International Journal on Artificial Intelligence Tools, 16, 1047–1067.

Kim, Y., Jernite, Y., Sontag, D., & Rush, A. M. (2016). Character-aware neural language models. In AAAI (pp. 2741–2749).

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. Nature, 521(7553), 436–444. doi: 10.1038/nature14539

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278–2324. doi: 10.1109/5.726791

Levy, O., & Goldberg, Y. (2014a). Linguistic regularities in sparse and explicit word representations. In Proceedings of the eighteenth conference on computational language learning (pp. 171–180). Baltimore, Maryland: Association for Computational Linguistics.

Levy, O., & Goldberg, Y. (2014b). Neural word embedding as implicit matrix factorization. In Proceedings of the 27th international conference on neural information processing systems (pp. 2177–2185). Montreal, Canada: MIT Press.

MacCartney, B. (2009). Natural langugage inference (PhD diss.). Stanford University.

MacCartney, B., & Manning, C. D. (2007). Natural logic for textual inference. In

Proceedings of the ACL-PASCAL workshop on textual entailment and paraphrasing (pp. 193–200). Prague, Czech Republic: Association for Computational Linguistics.

MacCartney, B., & Manning, C. D. (2009). An extended model of natural logic. In Proceedings of the eighth international conference on computational semantics (pp. 140–156). Stroudsburg, PA, USA: Association for Computational Linguistics.

MacCartney, B., & Manning, C. D. (2014). Natural logic and natural language inference. In H. Bunt, J. Bos, & S. Pulman (Eds.), Computing meaning: Volume 4 (pp. 129–147). Dordrecht: Springer Netherlands. doi: 10.1007/978-94-007-7284-7_8

Mikolov, T. (2012). Statistical language models based on neural networks (Unpublished doctoral dissertation). University of Brno.

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. CoRR, abs/1301.3.

Mikolov, T., Kombrink, S., Deoras, A., Burget, L., & Cernocky, J. H. (2011, December). Rnnlm - recurrent neural network language modeling toolkit..

Mou, L., Men, R., Li, G., Xu, Y., Zhang, L., Yan, R., & Jin, Z. (2015, December). Natural language inference by tree-based convolution and heuristic matching..

Mou, L., Meng, Z., Yan, R., Li, G., Xu, Y., Zhang, L., & Jin, Z. (2016, March). How transferable are neural networks in nlp applications? In Proceedings of the 2016 conference on empirical methods in natural language processing (pp. 478—-489).

Pang, L., Lan, Y., Guo, J., Xu, J., Wan, S., & Cheng, X. (2016). Text matching as image recognition. In Proceedings of the thirtieth aaai conference on artificial intelligence (pp. 2793–2799). AAAI Press.

Parikh, A. P., Täckström, O., Das, D., & Uszkoreit, J. (2016, June). A decomposable attention model for natural language inference. In Proceedings of the 2016 conference on empirical methods in natural language processing.

Pennington, J., Socher, R., & Manning, C. D. (2014). Glove: Global vectors for word representation. In Proceedings of the 2014 conference on empirical methods in natural language processing (pp. 1532–1543). doi: 10.3115/v1/D14-1162

Potts, C., & Maccartney, B. (2016). Models for natural language inference. Stanford, California: Stanford University. Retrieved from `http://web.stanford.edu/class/cs224u/https://web.stanford.edu/class/cs224u/materials/cs224u-2016-nli.pdf`

Rendle, S. (2010). Factorization machines. In Proceedings of the 2010 IEEE international conference on data mining (pp. 995–1000). Washington, DC, USA: IEEE Computer Society. doi: 10.1109/ICDM.2010.127

Rocktäschel, T., Grefenstette, E., Hermann, K. M., Kočiský, T., Blunsom, P., Kociský, T., & Blunsom, P. (2015, September). Reasoning about entailment with neural attention. CoRR, abs/1509.0.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986, October). Learning representations by back-propagating errors. Nature, 323, 533.

Scherer, D., Müller, A., & Behnke, S. (2010). Evaluation of pooling operations in convolutional architectures for object recognition. In 20th International Conference on Artificial Neural Networks (ICANN). Thessaloniki, Greece.

Smith, S. (1997). The scientist and engineer's guide to digital signal processing (2nd ed.). San Diego, Calif: California Technical Pub.

Tay, Y., Tuan, L. A., & Hui, S. C. (2018, December). A compare-propagate architecture with alignment factorization for natural language inference. In CoRR.

Tikhomirov, V. M. (1991). On the representation of continuous functions of several variables as superpositions of continuous functions of one variable and addition. In V. M. Tikhomirov (Ed.), Selected works of A. N. Kolmogorov: Volume I: Mathematics and Mechanics (pp. 383–387). Dordrecht: Springer Netherlands. doi: 10.1007/978-94-011-3030-1_56

Vosoughi, S., Vijayaraghavan, P., & Roy, D. (2016, July). Tweet2vec: Learning tweet embeddings using character-level cnn-lstm encoder-decoder. In Proceedings the 39th international ACM SIGIR conference. Pisa, Italy. doi: 10.1145/2911451.2914762

Wang, Z., Hamza, W., & Florian, R. (2017, February). Bilateral multi-perspective matching for natural language sentences. In Proceedings of the 26th international joint conference on artificial intelligence (pp. 4144–4150). AAAI Press. doi: 10.1145/2983323.2983769

Williams, A., Nangia, N., & Bowman, S. R. (2017, April). A broad-coverage challenge corpus for sentence understanding through inference. In Proceedings of the 2018 conference of the north american chapter of the association for computational linguistics: Human language technologies (pp. 1112–1122).

Yin, W., & Schütze, H. (2015). Convolutional neural network for paraphrase identification. In Human language technologies (pp. 901—-911). Denver, Colorado: Association for Computational Linguistics.

Yu, J., Qiu, M., Jiang, J., Huang, J., Song, S., Chu, W., & Chen, H. (2017, November). Modelling domain relationships for transfer learning on retrieval-based question answering systems in e-commerce. In CoRR (Vol. abs/1711.0).

Zhang, X., Zhao, J., & LeCun, Y. (2015, September). Character-level convolutional networks for text classification. In Proceedings of the 28th international conference on neural information processing systems (pp. 649—-657). Montreal, Canada: MIT Press.