Modelling Entailment with Neural Networks

Todor Davchev

s1045064

Master of Science

Artificial Intelligence

School of Informatics

University of Edinburgh

2019

Abstract

Sentence classification is currently among the unresolved challenges of Natural Language Understanding and Machine Learning. In this thesis we focus on modelling entailment relations which can be considered as a sub-problem of sentence classification.

We show that the results from the currently adopted Recurrent Neural Networks and Long Short-Term Memory models can be matched and even outperformed for recognising textual entailment. More specifically, we show that other techniques, such as Convolutional Neural Networks (CNNs), tackle the problem in a similar in terms of accuracy, however simpler in terms of feature engineering approach.

We propose a novel Siamese-like 3-CNN-wide architecture. We extend that model by applying a variety of mathematical operations to the intermediate input of the third CNN. More precisely, we exploit the low dimensionality representation of the already processed initial inputs via a series of linear and multiplicative operands. We then show that our approach achieves better results than the existing techniques, however significantly increasing the size of the parameters trained.

Nevertheless, our implementation has a modular and loosely coupled architecture.

Acknowledgements

I wish to express my sincere gratitude to my project supervisor, Prof. Mirella Lapata for her support, guidance and the constant feeling of inspiration.

Also, I would like to thank my family and friends for the unceasing encouragement, support and understanding throughout this venture. I am also thankful to all those who engaged in numerous discussions with regards to this topic.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Todor Davchev*
s1045064)

Dedicated to Bozhin and Yuliya.

Contents

Chapter 1

Introduction

Solving intelligence is among the unanswered challenges of the modern world. Understanding and building a system that is fully autonomous is an AI-hard problem. Those issues are hypothesised to involve computer vision, natural language processing (NLP) and dealing with unseen, unexpected situations (Shapiro, 1992).

Natural language understanding is a subtopic of NLP that deals with machine reading comprehension. Understanding and reasoning about natural language can lead to alternative solutions to issues related with visual impairments, colourblindness or help people overcome the language barriers in conversations.

1.1 General Overview

The original problem of Natural Language Understanding (NLU) can be simply put. It models the natural language reasoning process through parsing and disassembling of some input. In other words, formal NLU attempts to show how one represents and learns the semantics (meaning) of natural language, to which there are only partial answers[1].

NLU should not be confused with NLP or Automatic Speech Recognition (ASR). Figure 1.1 provides a graphical comparison between NLP, NLU and ASR.

The challenges on the topic date from 1964 when Daniel Bobrow wrote STUDENT (Bobrow, 1964)- a system that could solve algebraic problems by understanding simple natural language inputs. His work was followed by other projects which were also based around the concept of pattern-matching through the adoption of small rule-sets - good example of which is (Weizenbaum, 1966). Thus, the work of Bobrow is known

[1]http://videolectures.net/icml2015_liang_language_understanding/

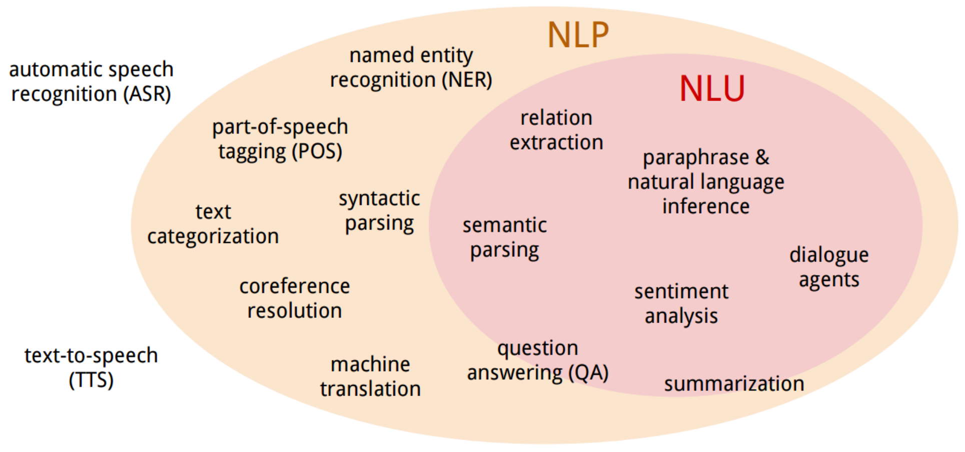


Figure 1.1: Relationship between the three fields[2].

to mark the first research conducted on the field. Similarly, the work of (Winograd, 1972), (Martin, 1973) or (Searle, 1980) during the 70s and 80s, addressed the challenge of understanding natural language through logic-driven, linguistically rich grounded systems[2]. The interest towards NLU suffered a drastic decrease during the 90s. That period, known as the "statistical revolution" in NLP, paid a lot of attention to the area of parsing as well as the fields of sentence processing and language acquisition (Johnson, 2009). Since 2010, however, NLU returned as a central interest, mixing the broad range of techniques introduced in the previous decades (Goldberg, 2015).

Moreover, the existing applications of neural networks (NN) have shown an improvement in many of the concepts laid out in Figure 1.1. Also known as non-linear classifiers, NNs can be roughly split in two main families, and namely into the fully connected feed-forward networks (which include multilayer perceptrons and convolutional neural networks with the associated pooling techniques); and the recurrent as well as recursive neural networks.

"Feed-forward systems have successfully managed to replace their linear alternatives" [3]. Being placed as immediate replacement of the linear models, they have managed to improve accuracy in topics like language modelling (Bengio et al., 2006) and syntactic parsing (Weiss et al., 2015), (Chen and Manning, 2014). Similarly, recursive (ReNN) and recurrent neural networks (RNN) have been largely successful in recognising regularities within sequentially structured data of arbitrary sizes such as sequences and trees (Goldberg, 2015). Good examples include the work of (Dyer et al.,

[2]http://nlp.stanford.edu/ wcmac/papers/20140716-UNLU.pdf
[3]T. Davchev's IRP report

2015) on dependency parsing and (Rocktäschel et al., 2015) on modelling entailment relations. Furthermore, convolutional neural networks (CNN) have proved very successful in tackling problems like sentence and document classification (Johnson and Zhang, 2015), (Kim, 2014) as well as question answering (Dong et al., 2015).

This project is focused on entailment modelling. The problem of Recognising textual entailment (RTE) is to determine whether the meaning of a natural language sentence is (i) in contradiction with another, (ii) not related with it at all, or whether (iii) the first sentence (called a premise) entails the second one (called a hypothesis) (Rocktäschel et al., 2015).

1.2 Motivations

The constantly growing impact of NLU has turned the field into an important and attractive research area. Moreover, it is facing a lot of interest not only from the academic world but from the industrial one too. The work of groups like SNLP[4] and ILCC[5] or products such as voice-driven assistants like Siri[6] or content summarization tools such as Summly[7] are currently driving the field forward, leading to a number of valuable and insightful discoveries (Reddy et al., 2016), (Chen et al., 2016), (Parikh et al., 2016).

The field of Natural Language Processing (NLP) uses entailment modelling in a variety of different problems such as machine translation, document summarisation and information extraction. These rely on the correctness of the RTE classification which itself marks the demand for as accurate textual entailment systems as possible.

In the light of these findings, we formulate our hypothesis. We believe that modelling entailment relations should be tackled as a classification problem. Thus, implying that CNNs can in fact introduce a simpler and yet at least as powerful solution as the RNN-based one proposed in (Rocktäschel et al., 2015).

We have proved that we can achieve higher results (81.26%) with the proposed in this thesis CNN architecture as opposed to a basic chain-structured long short-term memory (LSTM) network (Bowman et al., 2015) or even an Attention-based LSTM RNN (Rocktäschel et al., 2015) and this thesis aims to reveal how we did this.

[4]http://nlp.stanford.edu/
[5]http://web.inf.ed.ac.uk/ilcc
[6]http://www.apple.com/uk/ios/siri/
[7]http://summly.com/

In summary, there has been an increasing interest and constant progress in extracting information from unstructured data in the recent years. It is an exciting field for both the academic and industrial worlds as it gives the opportunity to be involved in the next state-of-the art solutions which will set the grounds for the forthcoming breakthroughs.

1.3 Realisation and Document Structure

In particular, we implement a solution in Theano. Moreover, we build a basic model which consists of a single CNN and use it to classify the relationship between two sentences. To ensure the correctness of our implementation, we use the MR dataset and compare the results obtained with those described in (Kim, 2014). We refer to this approach as our baseline. Then, we use the SNLI dataset and run it through the same neural network implementation. We analyse our results and compare them with the existing state-of-the-art results. We extend our basic model by implementing a Siamese-like architecture which consists of 3 CNNs in total. One for each of the two sentences and a third one which uses the output of the former two as input. Then, we implement a number of additive and multiplicative models which we use individually and build combinations of them. In all of these models we use a feed-forward layer for the final classification result.

The rest of the dissertation is divided into four sections. In Chapter 2, we provide a brief, yet sufficient overview of the background and related work to the research interests of this thesis. In Chapter 3, we define a detailed description of the work undertaken. Chapter 4 consists of the tests and evaluation conducted throughout this project. Finally, Chapter 5 consists of a conclusion and a discussion on the future work.

Chapter 2

Background

”You shall know a word by the company it keeps!”

(Firth, 1957)

Currently, the most successful approaches towards handling NLU tasks consider different variants of numerical representations of words and their relationship with other words. As already stated, the goal of NLU and NLP in general is to find ways of allowing computers to understand and reason about language.

Although linear combinations have useful analytical and computational properties, they fail to accurately classify complicated structures. The reason behind this is that they can easily fall under the curse of dimensionality. Thus, their application is limited to certain degrees. Fortunately, with the current advances in deep learning, the focus of NLP is no longer cast on using linear models (Goldberg, 2015). The main focus of this work is cast on refining the ways we use neural networks to classify entailment relations. Thus, the majority of this chapter provides a brief and yet sufficient explanation of the terminology required to understand the rest of the thesis. In addition, we adopt the writing consistency introduced in (Bishop, 2007).

2.1 Linear Models for Classification

To better explain the benefit of using neural networks, we will begin this chapter with an explanation of linear models. The term linear implies that the equation used to represent our model will always be a polynomial of degree 1. For example $y = 2x_1 + 3$ is a linear equation for y. We refer to x as a feature, something that describes our

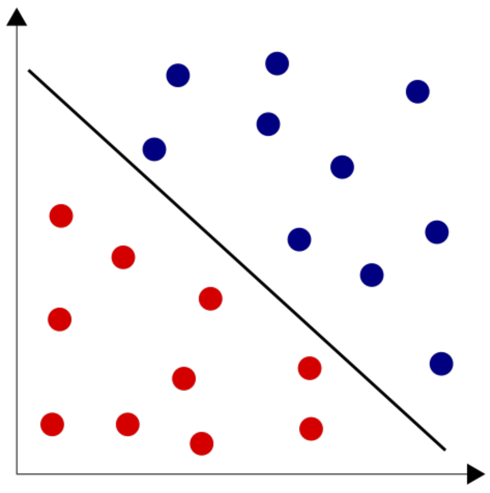


Figure 2.1: *Linearly separable data.*

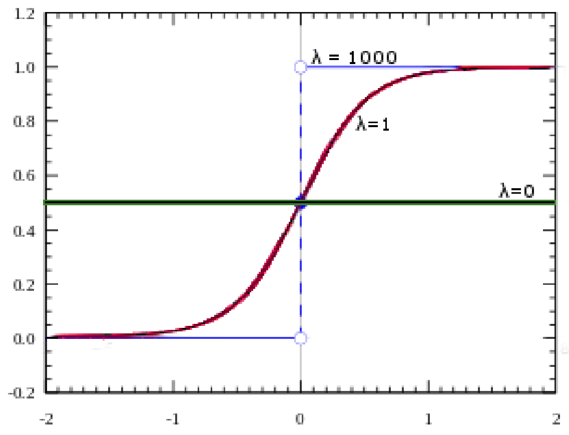


Figure 2.2: *Effect to what is learnt from varying λ.*

data. If the data was modelling a car, a potential feature would be its colour, or brand. The more features we use, the more descriptive our representation will be. In the example above we use 2 and 3 as constant values. In reality, however, we do not know what those values are. Thus, a common approach of writing them is as w_n. These values are referred to as weights. If we assumed that there is always an $x_0 = 1$ value which multiplies the free weight parameter (often referred to as bias), we can write the above equation as $y = w_1 \odot x_1 + w_0 \odot x_0$. Note that the bias parameter "shifts the position of the hyperplane, but does not alter the angle" (Williams, 2015a). We will represent the collection of features, i.e. the input to our linear classifier through a vector $X = [x_1, x_2, x_3, \ldots, x_n]$ and the collection of weights as another vector $W = [w_1, w_2, w_3, \ldots, w_n]$. Thus, we could represent the definition of y as:

$$y = W^T \odot X \tag{2.1}$$

The size of those vectors is the dimensionality D of our hyperparameters. Note that the equation presented here is both linear for the parameters W and the inputs X. Data sets which can have their content split with a straight line are called linearly separable (See Figure 2.1). Here, finding the weights is essential. This, will tell the classifier what is the best place to locate the straight line, i.e. this will define the maximum likelihood solution. Imagine we had a lot of values and more than one y. Then, X will be a matrix and not a vector. Thus, the closed-form maximum likelihood solution for the weight vector w using X and the vector y is given by:

$$\hat{w} = (X^T X)^{-1} X^T y\,, \tag{2.2}$$

We use y to refer to the target values and X is the matrix:

$$X = \begin{bmatrix} x_1^1 & x_2^1 & 1 \\ x_1^2 & x_2^2 & 1 \\ & \dots & \\ x_1^n & x_2^n & 1 \end{bmatrix} \tag{2.3}$$

Here, x_1^m is a feature x_1 for m rows from X and x_2^m is feature x_2 (Williams, 2015b). Appendix A.1.1 contains a detailed proof for the pseudo inverse.

In this model, the role of the bias is to neutralise the difference between the averages of the values in x or any other function that could accept x as a non-linearity measurement (Bishop, 2007). Appendix A.1.2 gives a detailed explanation of that statement.

The linear structure of the main equation can circumvent the limitations imposed by the linear structure of the equation with respect to the inputs. This can be done by introducing a function $\phi(.)$, which can accept the inputs X and hence deal with non-linearity. The equation will then be presented as:

$$y = W \odot \phi(X) = w_o + \sum_{i=1}^{N-1} w_i \odot \phi(x_i) \tag{2.4}$$

Functions of the form shown in Equation 2.4 are called linear models since they are still linear with respect to the parameters in W.

2.1.1 Activation Functions

The role of the activation functions is to transform the output from our classifier into a desired shape. Figure 2.3 shows a graphical representation of the output of the Sigmoid, TanH and ReLU functions as defined in this subsection. The Figure was taken from a Caffe tutorial[1]. The simplest and most commonly used function often used for the output of a regression problems is the identity activation function. It's range is between $-\infty$ and ∞ and maps the values used as input to the activation to themselves (see Equation 2.5). Quite often, we would like to control the range within which we get the output of our model. To achieve, this we would use one of the many activation functions. The most commonly used activation function is the Logistic function (Equation 2.6). The approach is also known as the 'logistic trick' (Williams, 2015a). It makes sure that the output will always be between 0 and 1. Hence, providing

[1]http://adilmoujahid.com/posts/2016/06/introduction-deep-learning-python-caffe/

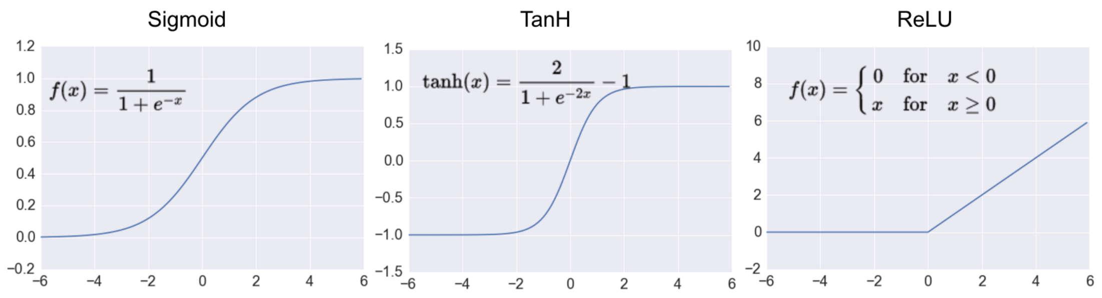


Figure 2.3: *The three main activation functions considered in this thesis.*

a probabilistic notion. A linear model, as defined in the previous subsection combined with the logistic trick results in the so called logistic regression.

However, in order to ensure that all the outputs will sum to one, we would need to use a Softmax function (Equation 2.9). This is especially needed in scenarios with many class options. Alternatively, it is sometimes useful to have output which ranges between -1 and 1 instead. For example, in a neural network setting, the very negative outputs will be forced to have close to zero representations if ran through a Sigmoid function. This will prevent the neural network from learning. Thus, we would need to use a hyperbolic tangent - TanH (see Equation 2.7). However, both functions will suffer from vanishing gradient problems (Nielsen, 2015). Then, the rectified linear unit function (Equation 2.8) can be used instead (Maas et al., 2013).

$$Iden(x) = x \tag{2.5}$$

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{2.6}$$

$$TanH(x) = \frac{2}{1+e^{-2x}} - 1 \tag{2.7}$$

$$ReLU(x) = \begin{cases} 0, & \text{if } x < 0 \\ x, & \text{otherwise} \end{cases} \tag{2.8}$$

$$Softmax(x) = \frac{e^{x_j}}{\sum_{n=1}^{N} e^{x_n}} \tag{2.9}$$

Note, that they will all differentiate differently. For the sake of argument, we will use the Sigmoid function for the proofs to follow, unless we state otherwise. Additionally, when using $\sigma(.)$ to describe a concept we would immediately assume that the same principle applies to all other functions, unless stated otherwise.

2.1.2 On-line parameter learning

The issue with the linear classifiers we have described so far is that they are very computationally expensive. Assuming a large data set, we could instead use a more sequential approach. Thus, we can use stochastic gradient descent. Also known as on-line learning algorithm. It iteratively decreases the value of the weights by a fraction of the derivative of the error with respect to that weight. The fraction is defined by a learning rate parameter η which can be empirically selected.

$$w^{\tau+1} = w^{\tau} - \eta \nabla E_D \tag{2.10}$$

Our work, however is based on an improved variant of this. Namely on Adadelta (Zeiler, 2012). That method dynamically updates the learning rate over time. We find Adadelta quite successful due to the fact that it explores both first and second order information. It accounts for the continual decay of the learning rate (η) during training and automating the process of choosing an actual value. Moreover, it does not introduce a significant overhead as opposed to quasi-Newton methods for optimisation, for example.

2.1.3 Regularisation

In classification tasks we use a data set which needs to be split in training, validation and testing parts which are used in three separate phases. The notion is quite intuitive. In the first phase, we use the training data set to train a network. We then verify that it learns properly through the validation set in the second phase. We repeat those two stages until we are confident enough that we have learnt all we can. Then, we use it in a third phase on an unseen data represented as the test set. Sometimes, the classifiers learn the training data too well, thus failing to determine the classes for the data presented in a test set, resulting in an over-fit over the training data. In order to prevent this, we can use different types of regularisation.

In linear classifiers, a regularisation term is added to the overall error function to prevent over-fitting. This term constitutes of a regularisation coefficient λ and an error function of the weights $E_W = \frac{1}{2}W^T W$. To better understand this concept, consider a scenario where we vary the value of λ in $\sigma(\lambda(w^T x + b))$, where $\sigma(.)$ is a Sigmoid function. The higher the value, the more we over-train the data. In other words, our classification turns into a step function. In contrast, the lower it is, the less we will learn (see Figure 2.2). Note, that there could be other types of regularisation. For instance, instead of multiplying w by itself (resulting in l_2 regularisation) we could take w to the power of 3,4 or else (Consider Figure 2.4). It is also easy to modify equation A.8 to result in:

$$W = (\lambda I + \Phi^T \Phi)^{-1} \Phi^T Y \tag{2.11}$$

Similarly, dropout is a simple technique where some of the weights are randomly assigned to 0, thus encouraging the actual deletion of features over training. In fact, (Kim, 2014) points out that this technique led to $2-4\%$ accuracy increase in his proposed solution for sentence classification.

Having considered the explained so far, we decided to use dropout and l_2 regularisation. In summary, dropout is known for preventing the co-adaptation of hidden units during forward propagation and l_2 regularisation is known for constraining the varying of the overall uncertainty while learning weights by maintaining proper error bars throughout training (see Fig 2.4).

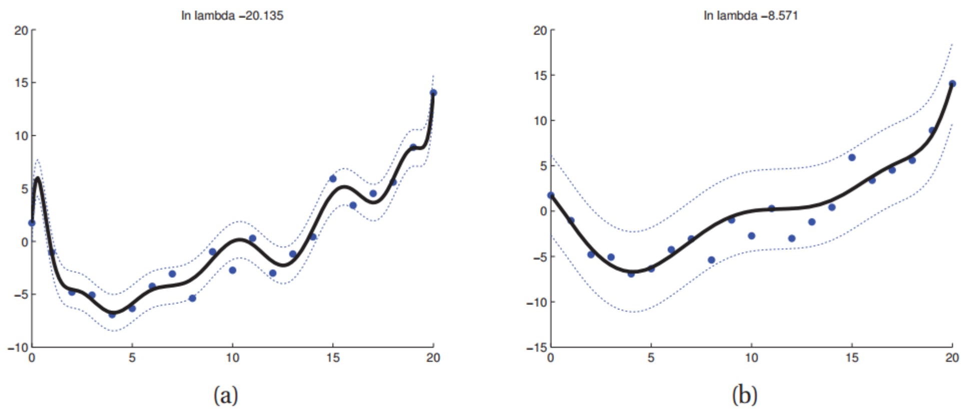


Figure 2.4: *"Degree 14 Polynomial fit to N = 21 data points with increasing amounts of l_2 regularization. Data was generated from noise with variance $\sigma_2 = 4$. The error bars, representing the noise variance σ_2, get wider as the fit gets smoother, since we are ascribing more of the data variation to the noise."* (Murphy, 2012).

2.1.4 Cross Validation

The purpose of cross validation is to ensure that the trained model generalises well. It is a technique which is commonly used when there is insufficient amount of training data. Moreover, it splits the data into smaller subsets (usually 10) and uses them to formulate training and validation subsets. Each subset is then used for validation in separate from each other trainings. Finally, we use the mean of all results as the final output.

2.2 Feed-forward Neural Networks

Although there are other approaches, like SVMs, for example, our work is influenced from the advances in Neural Networks. The most basic ones are the feed-forward neural networks. The main idea behind them is to fix a set of basis functions $\sigma(.)$ and allow for their parametric forms to be adapted during training (Bishop, 2007). In short, this model consists of multiple layers of logistic regressions with continuous non-linearities, thus implying the need to tackle non-convex scenarios. A drawback of this process is that it is very computationally expensive during its training phase. The reason for this is the cost of mathematical operations for large collections of data.

2.2.1 Concepts

If we think of a neural network as an approach of tackling some D-dimensional input as a linear combination of its D separate features, we can easily justify the purpose of a non-linear activation function. Every basis function in a neural network is itself a non-linear function of the inputs and the associated weights. We can thus refer to the linear combinations of those inputs as the variable parameters, which are in fact what we will continuously adapt during the training phase. In such a setting, the so called hidden units are namely the non-linear activation functions. Hence, we can represent this concept as:

$$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \tag{2.12}$$

where (1) represents the first 'layer' of a network and j = 1...M as defined in (Bishop, 2007). Then, we can represent a hidden unit as

$$z_j = h(a_j) \tag{2.13}$$

So far this definition is exactly the same as a simple logistic regression. A single non-linear function z_j defines a single hidden unit in a neural network. The non-linear functions $h(.)$ are usually chosen as the logistic Sigmoid or the hyperbolic tangent functions (Bishop, 2007). The outputs of those hidden units are linearly combined to produce the output unit activations, defined as follows:

$$a_k = \sum_{j=1}^{M} w_{kj}^{(2)} z_j + w_{k0}^{(2)} \tag{2.14}$$

where $k = 1, \ldots, K$ is the number of outputs. These are the outputs for the second layer of the network. Finally, we would like to use a Softmax (see Equation 2.9) to represent a set of N possible outputs. $y_k = Softmax(a_k)$ will always be between 0 and 1 and will sum up to 1 (Renals, 2015). Thus, suggesting a probabilistic structure where the output with the highest value will essentially be chosen as the correct class for a given input (see Figure 2.5). The process of propagating through a network starting from the inputs and ending at the outputs can be referred to as "forward propagation" of information. The continuous sigmoidal non-linearities in the hidden units $h(.)$ makes

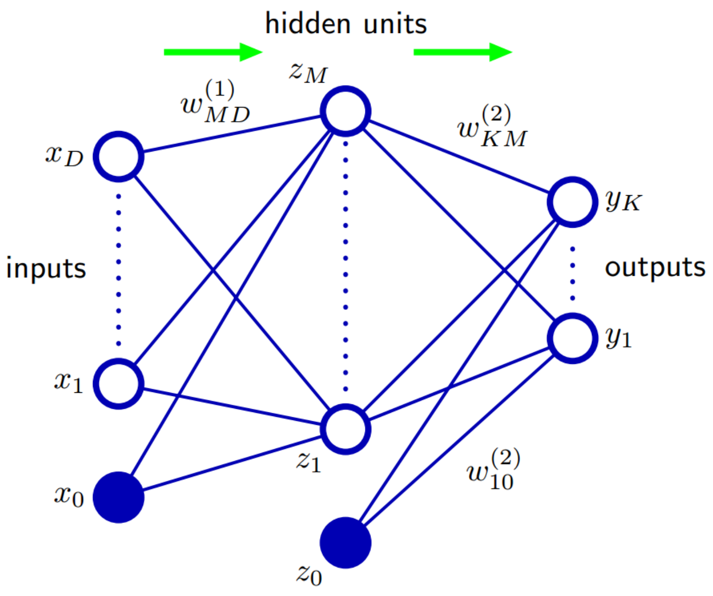


Figure 2.5: *A neural network with two hidden layers* (Bishop, 2007). *The image denotes the direction of the network with a green arrow. It also represents the inputs as $x_{1...D}$, the weights as w_{MD} where M refers to the neuron/unit the weight points to and D is the actual neuron the weight comes from. $z_{1...M}$ is the set of hidden units. Both x_0 and z_0 are bias terms while $y_{1...K}$ are the actual outputs.*

the neural network function differentiable with respect to the weights. This is essential to the training phase as it allows us to model the differences between the output and the targeted result. This difference can then be back propagated to the beginning of the network and used to update the weights. This process is then repeated a chosen amount of times which is in fact the definition of a training phase, also known as epoch.

Additionally, we define the network in Figure 2.5 as a two-layer neural network since there are two layers of adaptive weights. Figure 2.6 shows the classification problem solved by a feed-forward neural network. Note how it differentiates between the classes in a rather more complicated setting than the one presented in Fiugre 2.1.

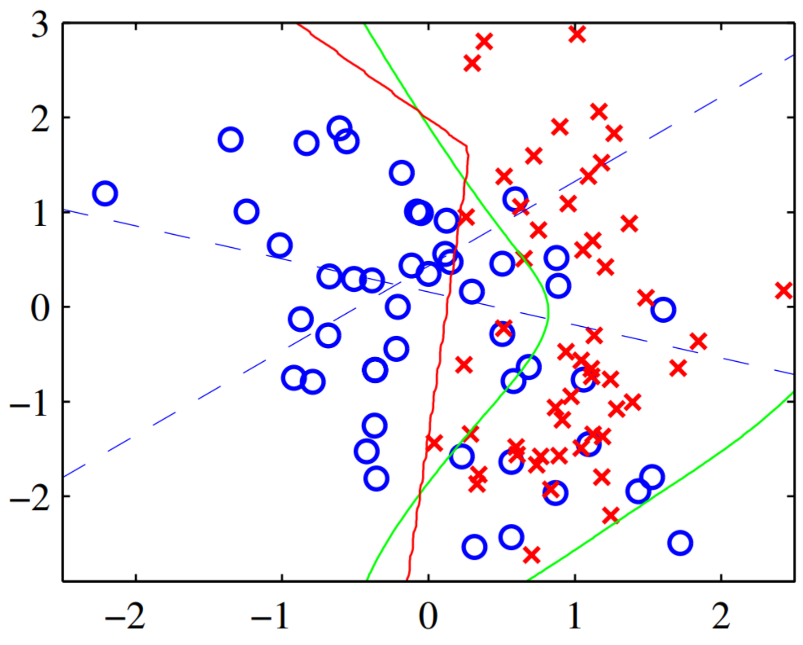


Figure 2.6: *Two-class classification with neural network. Green line represents the optimal decision boundary* (Bishop, 2007).

2.2.2 Network Training

In the last subsection, we briefly mentioned the concept of training a neural network. Here, we will explain that concept in a more detailed way. As already stated, the way neural networks work is through forward and backward propagation. To do that, however, we need to have some sort of way for measuring the difference between the output of a network and the targeted outcome. This can be achieved by training on a data set which contains a representation of some sort and its explanation - i.e. to what class a pair of sentences belongs to. Depending on that measurement, there exist different types of error functions - mean squared error, corss-entropy and others. Consider Equation A.10, we can state that there is a relationship between the error and the activation functions. However, to be more precise and show this, we should rewrite

A.10 as follows:

$$E(W) = \frac{1}{2}\sum_{m=1}^{M}(t_m - y_m)^2 \tag{2.15}$$

Hence, $y_m = a_m$, can be interpreted as the output equals the outcome of the final activation function. Thus, the sum-of-square errors (SSE) function is a result of taking the derivative of the error function with respect to the activation function.

$$\frac{\partial E}{\partial a_k} = t_k^2 - 2t_k y_k + y_k^2 = 2y_k - 2t_k \tag{2.16}$$

Thus:

$$\frac{\partial E}{\partial a_k} = y_k - t_k \tag{2.17}$$

Similarly, if we consider a Bernoulli classification problem, we can use Equation A.16 to define the cross-entropy (CSE) error function as follows:

$$E(W) = -\sum_{n=1}^{N}\{t_n ln(y_n) + (1-t_n) ln(1-y_n)\} \tag{2.18}$$

Note that in the equation above $y^n = \sigma(.)$, where $\sigma(.)$ is the term used in Equation A.16. Interestingly, (Simard et al., 2003) shows that using CSE for a classification problem leads to faster training and better generalisation.

We would also like to pay attention to the multiclass classification problem. This entails a classification where each input is associated with one of K different classes which are mutually excluding. Our project is using exactly this approach as described later in this thesis. The idea is that the target variables t_k have K possible outcomes. Thus, the probabilities 0-100 are split in K parts (in the case of this project 3) through the adoption of a Softmax activation function. Then, the outputs from the network are thought of as $y_k = p(t_k = 1|x)$, thus the error function:

$$E(W) = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_n^k ln(y_n^k n) \tag{2.19}$$

Given the definitions of an error function and the appropriate choice of one for a certain setting, the process of training consists of a continuous approach towards minimising the value of a chosen error function. This, as we already explained, is achieved through forward and backward propagation.

2.2.3 Parameter Optimisation

Minimising the error function essentially means finding the most suitable weights. The graphical representation (see Figure 2.7) of optimisation from (Bishop, 2007) can be particularly useful for visualising the explained so far.

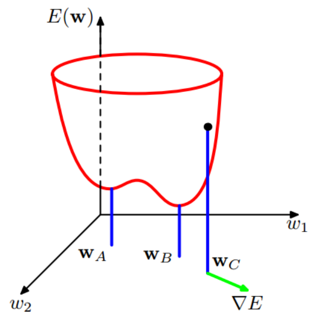


Figure 2.7

Geometrical view of the error function E(w) as a surface sitting over weight space (Bishop, 2007).

It is important to note that the vector $\nabla E(w)$ points towards the direction of increasing error and we are interested in following the exact opposite (i.e. $-\nabla E(w)$). Since the weight space is non-convex, a small step towards the wrong direction could lead to big variations. Although there are different ways for minimising the error (for example with random search), we do this through optimising our weights with gradient descent. Moreover, the highly nonlinear relationship between the weights and the error function leads to numerous places where the gradient becomes very small or in fact vanishes. Thus, using ReLU as an activation function can prevent the network from the vanishing gradients problem and also from constricting the output between 0 and 1 (Zhang and Wallace, 2015). There exist other ways of tackling this issue, however, they are dependent on the setting and the task.

There could be a number of local minimas. In fact, circumventing them is a major challenge in parameter optimisation. Bishop points out that in a standard two layer feed-forward neural network which consists of M hidden units, there are $M!2^M$ equivalent points for every existing point. Given the complexity of finding a global minima, it is commonly accepted that using a local minima is proved to be sufficient. However, it might be necessary to compare some number of local minimas before choosing one. A commonly accepted way of finding the most appropriate weights is namely through on-line parameter learning where we would initially set the weights to random values. In practice, it turns out that setting these values close to zero end up giving the best results.

2.2.4 Error Backpropagation

The process of sending information forward and backward throughout the training phase of a neural network is known as error backpropagation, or backprop (Bishop,

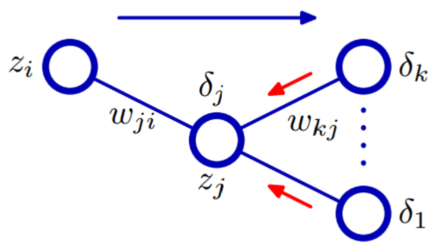


Figure 2.8 *"Illustration of the calculation of δ_j for hidden unit j by backpropagation of the δs from those units k to which unit j sends connections. The blue arrow denotes the direction of information flow during forward propagation, and the red arrows indicate the backward propagation of error information."* (Bishop, 2007).

2007). As already discussed, the training process aims at minimising the error (also known as cost) function. This can essentially be split into two stages. The first one consists of finding the derivative of the error function with respect to the weights. The second stage consists of the actual propagation of those derivatives backwards to the initial weights. Luckily, the chain rule - a concept that entails the differentiation of a function of a function, let us do that. This, essentially enables the actual update of the initial weights which will then be propagated forward until the new evaluation of y, which is iteratively compared with the targets. The simplest such technique involves gradient descent (Rumelhart et al., 1985). To give a better notion we encourage the reader to consider the SSE error function and a single hidden layer neural network. The gradient of this error function with respect to some weights was derived in Equation A.19. This can be thought of as a product between a given difference and the input itself. To address the second stage we adopt the chain rule as follows:

$$\frac{\partial E(w)}{\partial w_{ji}} = \frac{\partial E(w)}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}} \tag{2.20}$$

Hence, for expressiveness we can introduce $\frac{\partial E(w)}{\partial a_j}$ as δ_j and $\frac{\partial a_j}{\partial w_{ji}}$ as z_j. Thus, Equation 2.20 becomes:

$$\frac{\partial E(w)}{\partial w_{ji}} = \delta_j z_j \tag{2.21}$$

The above equation states that we can use the output representation of some weight and multiply it by the value of z which itself is the input representation of the weight. Note that if we were considering the bias, then $z = 1$ (Rumelhart et al., 1985), (Bishop, 2007) and (Renals, 2015).

As we already showed $\delta_j = y_j - t_j$. If we use δ's definition from above and hence make use of Equation 2.20 and also consider Equations 2.13 and 2.14, we will see that we can extend the backprop to the second stage of the previously defined algorithm exactly as shown in (Rumelhart et al., 1985), (Bishop, 2007) and in (Renals, 2015).

Namely:

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k \tag{2.22}$$

Therefore, we can apply the backpropagation algorithm to any feed-forward neural network, regardless of the number of hidden layers and topology. In Renals (2015) the error backpropagation algoritm is summarised very concisely into the follwoing steps:

1. Forward propagate given input vectors taken from the training set;
2. Compute the cost by comparing the outputs to the target values;
3. Evaluate the error signals δ_j for every output;
4. Evaluate the error signals δ for every hidden unit;
5. Obtain the overall derivatives through a basic summation over the evaluated derivatives for each training batch.

Backpropagation is considered as one of the breakthroughs in the modern representation of neural networks since it enables their wider adoption in a variety of domains. Among the main reasons for this is backprop's computational efficiency - $O(W)$. This follows naturally from the fact that the information is being passed through each layer once until it reaches the initial one and thus lead to the weight updates.

2.2.5 Model Architectures

The concepts defined so far can be used in many different ways as briefly mentioned in the previous subsection. Different number of layers can be combined in different ways, some weights can skip hidden activation frontiers and communicate with the ones after them. In addition, there could be different combinations of functions which can be used. The concept of designing a neural network is referred to as model building, where a given network architecture is called a model. The model optimisation consists of fine-tuning the model parameters, also known as hyper-parameters - type of activation function, what features are being used, the actual choosing of an architecture, how to regularise and many others. There are a variety of approaches which can be used, depending on the optimisation done. For example one could use grid search, randomised approach or even Bayesian parameter optimisation techniques to choose the most appropriate learning rate for the weights optimisation. The difficulty comes from the lack of structural definition of how to optimise a network for a specific task.

Thus, the difficulty of problem solving using neural networks. Essentially, different architectures perform differently for every task. The simple network in Figure 2.5 is an example of a very basic neural network model.

2.3 Word Vectors

Having covered all of the above however, does not explain how to apply it to words and sentences. To achieve this, we need to be able to represent words with numbers. This section will cover the main approaches used in support of this thesis.

To begin with, we need to point out that the meaning of a word or a sentence is very dependent on the context setting, as initially observed in (Harris, 1954). And quite intuitively, a word can mean different things in different contexts. In fact, the set of meaningful contexts can vary. For example, consider the word ”Cristina” in the following sentence ”Cristina is coming! Evacuate your homes!”. The second sentence suggests that the speaker refers to the hurricane. Similarly, ”Cristina is coming! Last time I saw her was on that family meeting two years ago.” suggests that Cristina is a human being, probably a relative the speaker has not seen for two years.

In the same sense, the goal of the word vectors is to provide the means for finding a successful way to represent words. This is achieved by embedding a word into a vector space (Jurafsky and Martin, 2014). Thus, we will use the term embedding to refer to the vector representation of a word.

2.3.1 Word Representations

There are different ways to represent a word as a vector. N-grams, for example model the probability of a word given its surrounding words. Essentially, modelling the distributions representing words.

$$p(W) = \Pi_i \, p(w_i | w_1, \ldots, w_{i-1}) \tag{2.23}$$

It is often simplified to trigrams as so:

$$p(W) = \Pi_i \, p(w_i | w_{i-2}, w_{i-1}) \tag{2.24}$$

Essentially, this model defines a probabilistic distribution of a sequence of words.

An alternative approach is to use one-hot vectors. This means that every word is represented as a vector which is as wide as the entire vocabulary. However, only the word that is currently considered is represented as a one, everything else is set to 0. The Bag-of-Words representation expands the previous idea. Here, each sentence is represented as a vector as wide as the entire vocabulary and each index represents the number of times the word corresponding to that index.

The issue with sparse vectors is their size where most of the values is zero. This makes them very inefficient in machine learning systems. Another advantage of the dense vectors is that their size, or more specifically, the lack of many zeros, means that they can generalise better (Jurafsky and Martin, 2014). Thus, our work is based on more complicated, yet dense models which are trained on neural network models as explained next.

2.3.1.1 CBOW and Skip-gram

The idea behind these models is that a neural network learns words representations by making the embeddings of neighbouring words similar to each other and less relevant to other words. This thesis makes use of a software package called word2vec. It implements a Continuous Bag-of-Words (CBOW) model and a skip-gram (Mikolov et al., 2013), (Mikolov and Dean, 2013).

The idea behind skip-gram is to predict the context L from a given word. It achieves it through learning two separate embeddings - namely one for the word projection and 2L for the context. If L=2 then a skip-gram will predict $[w_{i-2},\ w_{i-1},\ w_{i+1},\ w_{i+2}]$. Those two embeddings are encoded in separate matrices - W and C. Each row of which contains words each of which is $|V|$, where V is a given vocabulary. Every word is represented with a one hot vector. Given that definition, we can safely assume that the model is trying to predict $P(W_{context}|W_{target})$. Figure 2.9 shows the model for a context window of $L = 1$, $[w_{i-1},\ w_{i-2}]$. The input x contains one-hot vector representation for a given word w_t. The hidden layer, h is the so called projection layer. It represents the word embeddings after multiplying w_t by a randomly initialised weight matrix. In mathematical notation, we can use Equation 2.4 where x will correspond to the target word w_t and W will be the weights. In the same way y will be the actual word embeddings or the so called projection layer for w_t. Moreover, since we use one-hot vector representations, then the projection layer for word w_t is going to be v_j. Then, for each of the context words, we multiply them by the context matrix C for each of the 2L context words. The output is then normalised with a Softmax function (see Equation 2.9).

CBOW is the mirror image of skip-grams (Jurafsky and Martin, 2014). They are

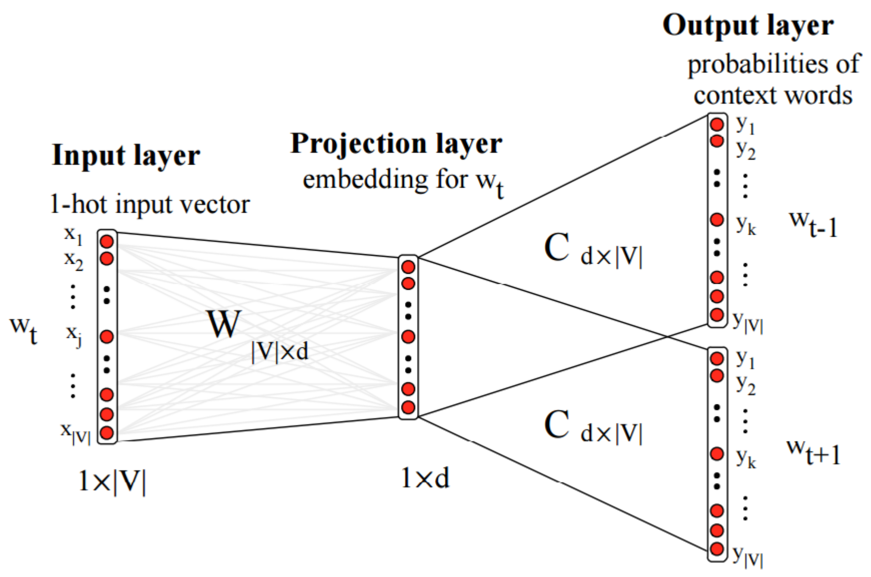


Figure 2.9: *The skip-gram model* (Mikolov et al., 2013), (Mikolov and Dean, 2013), (Jurafsky and Martin, 2014). *A word is denoted as w_t, while $x_{1...|V|}$ are indexes of the vector representation of w_t. $|V|$ is the vocabulary size. The size of the projection vector is d and $W_{|v|xd}$ and $C_{dx|V|}$ are embeddings matrices for the projection and the context respectively.*

modelling words based on the surrounding context. Depending on the task, each model might prove more useful. According to (Mikolov et al., 2013) and (Mikolov and Dean, 2013), skip-gram works better with smaller data sets, however takes longer to train than CBOW, which itself performs better for frequently occurring words. CBOW, for example, can be useful for text classification where the input consists of two sentences and the task is to classify then under certain criteria.

2.3.2 Semantic Composition

Having covered the notion of modelling words with numbers, it seems natural to attempt to model the semantic composition through the help of algebraic operators such as addition or multiplication. In fact, (Mitchell and Lapata, 2010) show how to do this. Logically, we split the means of connecting words' or sentences' compositions to two, and namely additive and multiplicative models. Table 2.10 shows a basic example of the relationships between a collection of words. We define an additive model as $p = Au + Bv$. Then, the following instances of additive models exist as defined in (Lapata, 2016):

	tune	answer	effectiveness	competence	build
reasonable	1	7	3	11	5
complexity	2	9	5	5	1
issue	1	14	6	8	0

Figure 2.10: *Word context matrix* (Lapata, 2016).

1. $p = u + v$, hence $reasonable + complexity = [3\ 16\ 8\ 16\ 6]$;

2. $p = u + v + \sum_i n_i$, thus $reasonable + complexity + issue = [4\ 30\ 14\ 24\ 6]$;

3. $p = \alpha u + \beta v$, therefore $0.4 * reasonable + 0.6 * complexity = [1.6\ 8.2\ 4.2\ 7.4\ 2.6]$;

4. $p = v$, where $complexity = [2\ 9\ 5\ 5\ 1]$.

Similarly, we define the multiplicative model as $p = Cuv$ and extend to the following instances:

1. $p = u \odot v$, hence $reasonable \odot complexity = [2\ 63\ 15\ 55\ 5]$;

2. $p = u \otimes v$, thus $reasonable \otimes complexity = \begin{bmatrix} 2 & 9 & 5 & 5 & 1 \\ 14 & 63 & 35 & 35 & 7 \\ 6 & 27 & 15 & 15 & 3 \\ 22 & 99 & 55 & 55 & 11 \\ 10 & 45 & 25 & 25 & 5 \end{bmatrix}$;

3. $p = u \circledast v$, therefore $reasonable \circledast complexity = [160\ 94\ 110\ 106\ 124]$.

The last item from the multiplicative models is a type of 1D convolution known as a circular convolution. For further details on how to compute it, we refer the reader to Appendix A.2.1.

2.3.2.1 Global Vectors

Even though the work proposed in (Mikolov et al., 2013) achieves very accurate analytical representation of words, it fails to use the statistical information embedded in texts. Using global vectors for word representations (GloVe) (Pennington et al., 2014), however proposes an approach which can achieve both good analytical and statistical representation of word vectors. Unlike CBOW and skip-gram, it uses dense vector representation as its input. GloVe uses the ratio between two probabilities of words to appear in a given context. It then uses these to compare the ratios between two words in the same context. (Pennington et al., 2014) shows a detailed definition of their model which is proved to use $log(1+X_{ik})$ for some word i under a context k. They train their model against a weighted least squares regression function, defined as:

$$E(W) = \sum_{i,j=1}^{V} \phi(X_{ij})(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - logX_{ij})^2 \tag{2.25}$$

Where (Pennington et al., 2014) defines ϕ as:

$$\phi(x) = \begin{cases} (\frac{x}{x_{max}}), & \text{if } x < x_{max} \\ 1, & \text{otherwise} \end{cases} \tag{2.26}$$

According to (Pennington et al., 2014) GloVe seems to perform better in most tasks than word2vec.

2.4 Convolutional Neural Networks

2.4.1 Motivation

Although feed-forward neural network are good at specific tasks yet they fail to achieve human-like or better accuracy. Among the main reasons for this is that they consider their input as equally important and assume that each dimension is equally related with the rest, or in other words it relies on dense weights representation. Thus, the size of the parameters becomes quickly infeasible.

Feed-forward networks do not consider the local regional importance in a given input. A rough notion about this can be created via a real-life example. Usually, in a book, the beginning often does not provide much information about how the story ends. Similarly, in a sentence, reading the first few words might not reveal enough information to adequately reason about its entire meaning. Thus, instead of thinking about the end of a book we can focus on its beginning and the immediate follow-ups. Thus discarding some of the unreasonable assumptions we make. Quite similarly, convolutional neural networks can adequately address that issue by adopting sparse weight relationship between the input and the output (LeCun et al., 1989). As a concept, they introduce a window (also known as kernel) through which they "convolute the context of a word with the word itself"[2]. A convolution is in fact quite similar to a cross

[2]Citation from T. Davchev's IRP report.

correlation. However, it reverts the second function and then applies cross correlation (for a more mathematical notion, see Appendix A.2.1). Moreover, we define a simple convolution as:

$$(x*g)(i) = \sum_{j=-\infty}^{\infty} g(j)(i-j) \tag{2.27}$$

Consider Figure 2.12 for an intuition of what a convolution is.

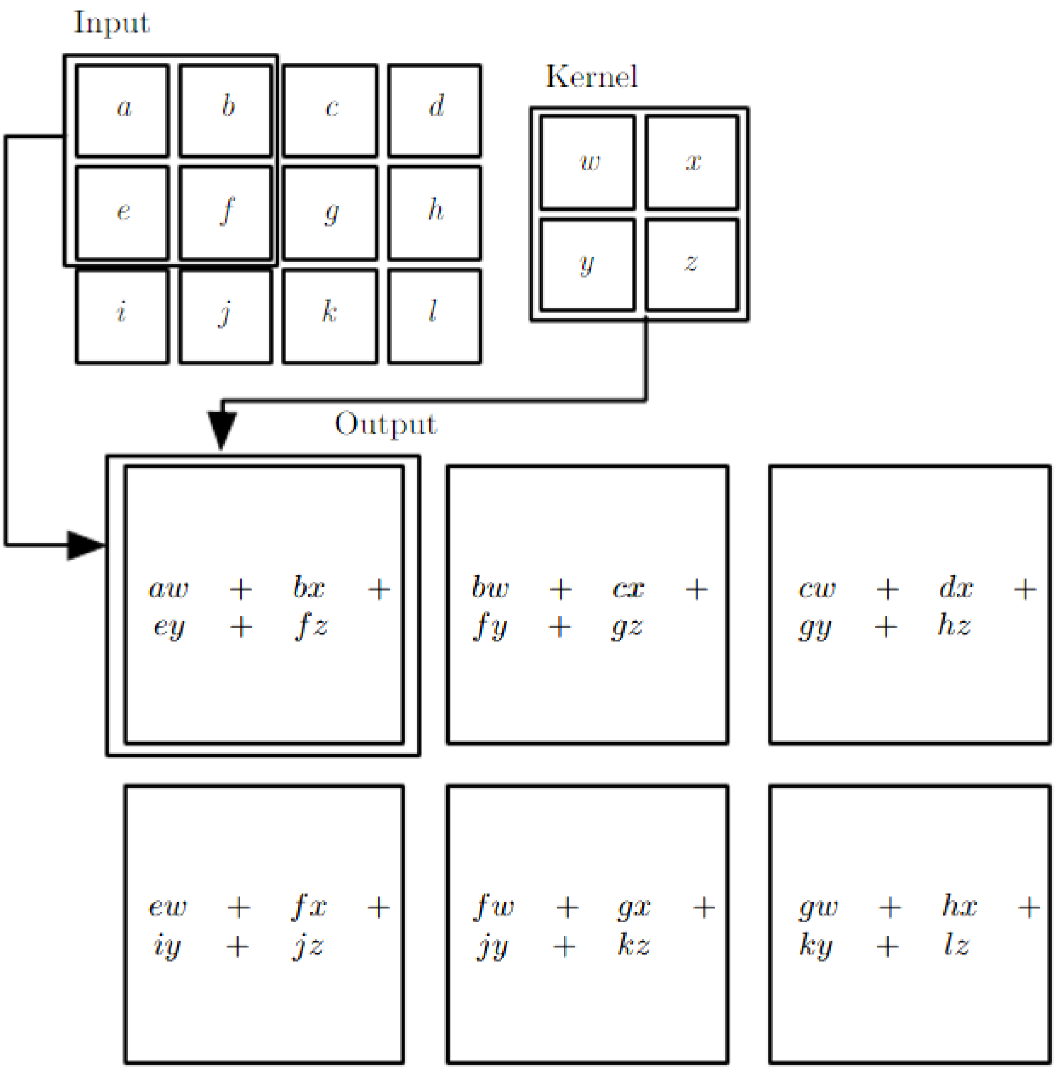


Figure 2.11: *"An example of 2-D convolution without kernel-ipping. In this case we restrict the output to only positions where the kernel lies entirely within the image, called validconvolution in some contexts. We draw boxes with arrows to indicate how the upper-left element of the output tensor is formed by applying the kernel to the corresponding upper-left region of the input tensor"* (Bengio and Courville, 2016)

2.4.2 A different perspective

(Bishop, 2007) defines three main mechanisms that enable CNNs to incorporate these relationships and namely: ”(i) local receptive fields, (ii) weight sharing, and (iii) pooling”. These three concepts come exactly from the sparsifying of the weights. We can well see these if we thought of a CNN as a feed-forward network for a moment (hinted from a lecture by Ali Ghodsi[3]). If we, then imagine a given set of inputs and a given set of outputs (as shown in Figure 2.12), we can then introduce the notion of locality. Namely, neighbouring inputs, perceived as local receptive fields produce a single output. Note the number of local parameters per single convolution will be defined by the actual kernel. The resulting parameters that connect all inputs with outputs belong to a single kernel, hence the notion of sharing. If we were considering a feed-forward neural network, the number of weights would have been 12 (i.e. 4 by 3). However, here the number of the parameters is in fact 6. Sparsity is a very interesting and actively researched area in the field of Machine Learning, it finds its applications in other topics like low-rank matrix approximation as discussed in the last chapter of this thesis.

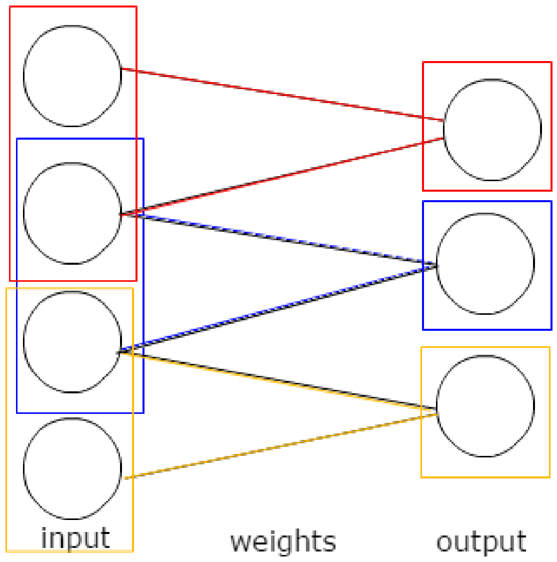


Figure 2.12: *Notion of how CNNs function.*

After a convolution is applied to the two functions (the input and the kernel), we apply a non-linearity function which results in the actual output in a similar way as with feed-forward neural networks where we applied a Sigmoid or a ReLU function, for example. This is called a Detector layer (Bengio and Courville, 2016). After this,

[3]https://goo.gl/bX8dgf

we apply the third mechanism from (Bishop, 2007) and namely pooling. In a way, pooling summarises the resulted outputs by looking for specific criteria. However, we leave the further details on that subject to a later subsection.

2.4.3 In depth analysis

Using CNNs requires the manual selection of a number of hyper-parameters. Thus, choosing the best for a given architecture can prove to be a tedious task. The work of Ye Zhang (Zhang and Wallace, 2015) is a good reference point for sentence classification in particular. Regardless, we will give a brief notion of the type of considerations that needs to be made throughout the fine-tuning of a CNN model. The structure of this subsection is influenced from (Britz, 2015).

2.4.3.1 Wide and narrow convolutions

In practice, it is common technique to pad sentences in order to ensure uniform length of the input as well as to account for striding through the first n sentences when the window size is larger than n. The reason for padding with zeros and not with the nearest neighbours, for example, is not entirely motivated in the literature. We think that using a zeros ensures that the padded cells will be ignored when multiplying all elements from a window. Although it should be also possible to pad with the nearest neighbours, there will be certain risk to double the padded neighbours' importance. Regardless, the common practice is to use zeros instead of anything else. Using zero-padding is often referred to as wide convolutions as opposed to not using zero-padding - narrow convolutions (Britz, 2015). The size of the output depends on the adoption

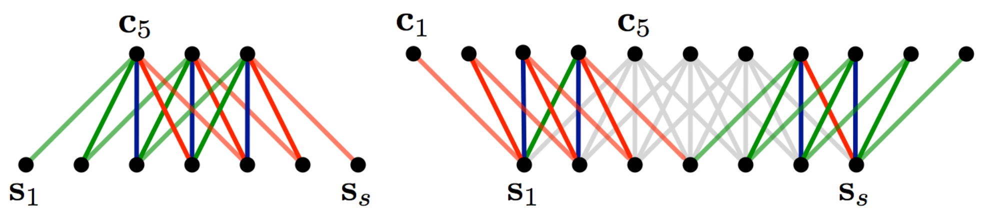


Figure 2.13: *Zero-padding example* (Britz, 2015).

of wide convolution as shown in Figure 2.13. More generally, the output size can be computed with the following formula:

$$|output| = (|x_{in}| + 2 \times |padding| - |filter|) + 1 \tag{2.28}$$

2.4.3.2 Strides

The shift of the filter when going through the input is defined by the stride size. The larger the stride, the more independent the features/local regions will be from one another. In other words, it leads to a smaller output size. Thus, a common stride size for NLP tasks in general is 1. Moreover, (Britz, 2015) points out that a larger stride size can make a CNN behave more like a Recursive Neural Network - id est behave more like a tree.

2.4.3.3 Pooling Layer

Pooling layers are best thought of as summarising the outputs of a convolutional layer through some function. They achieve this through various techniques. A widely adopted one is the k-max pooling, which selects only the k max outputs from a given filter output. In fact, it a commonly used approach is to pool over a small window instead of the entire feature map (see Figure 2.14 for an example of 1-max pooling[4]). Therefore, pooling can be thought of the means of reducing the dimensionality of some input. This, however would mean, that backpropagation will only update the values of those k selected max features, thus leading to loss of information. Although there are other pooling techniques, such as the L_p pooling or average pooling, we will use the 1-max pooling throughout this dissertation as this is the commonly accepted strategy (Zhang and Wallace, 2015).

Moreover, pooling layers are known to provide some useful properties. For example, it makes a feature invariant to local translation[3]. As an example consider Figure 2.15. Imagine that the content of the figure can be logically split in two parts- a left and a right one. Consider the left structure with four circled representations and three on top. Let us assume that the ones on the bottom are the contents of a feature map. Therefore, the three on top will be the results after applying 1-max pooling. Shifting the contents of that structure by one, results in the drawing from the right part of the figure. Notice how all of the circles in the bottom have now have completely different values compared to the ones in the left but the upper three have not changed that much

[4]http://cs231n.github.io/convolutional-networks/#pool

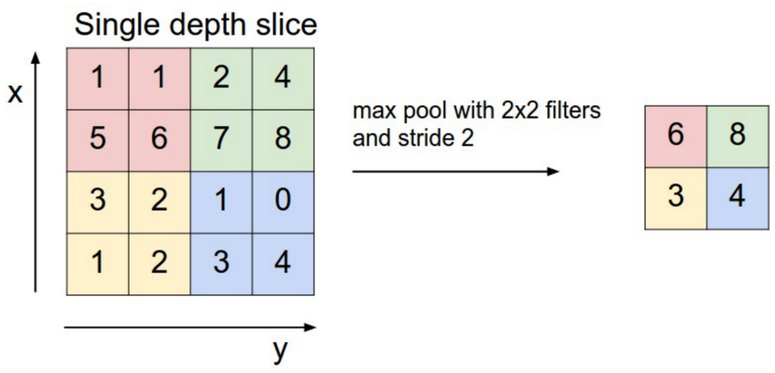


Figure 2.14: *1-max pooling.*

- only one of the three data points was in fact changed[3]. This property is in fact very useful for the purpose of our work because we are not as interested in learning if a given word is in the same position as before but in fact we would like to know if that word exists in a specific input we are looking at. The example in Figure 2.15 can be also used to show downsampling-i.e. the 4 inputs result in 3.

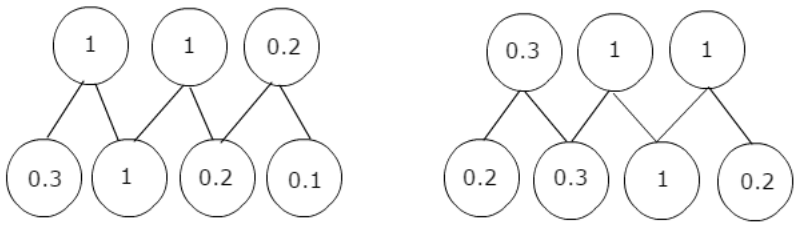


Figure 2.15: *Translation invariance.*

2.4.3.4 Channels

Channels enable the different representation of the input data. For example, in Vision, those often are the red, green and blue variants of an image. In NLP and NLU, these could be a word2vec and GloVe representations of the data, or have different ways of conveying the same information, for example a sentence translated in various languages (Britz, 2015).

2.4.4 Backpropagation

To train a CNN, we need to be able to compute the gradient with respect to the output as well as the gradient with respect to the kernel function given the gradient with respect to the output function. This, however, can be tricky as there is an itneresting property that needs to be considered. Namely, we need to bare in mind that we would essentially have to rotate the weight matrix in order to accurately compute the gradient. The reason for this is that the last element of a kernel striding over a given input results in the first element in the feature map as shown in Figure 2.16. Regardless, a more detailed, mathematical definition of backpropagataion is described in Appendix A.2.2.

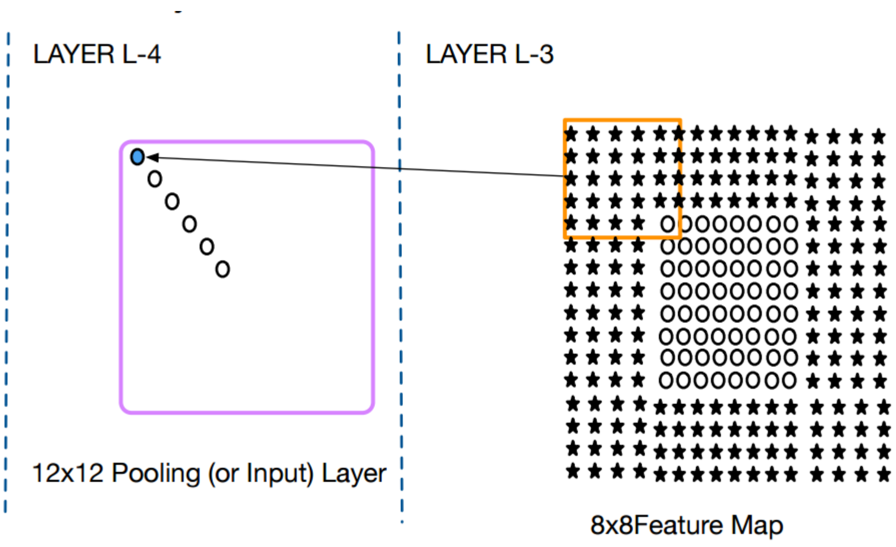


Figure 2.16: *The last element of a kernel window is the first element in a feature map. Here, the stars represent 0-padding and each dot is a value.*

2.4.5 Simple architecture of a CNN for NLU

This subsection aim at introducing the concept of CNNs in the light of NLU. Consider the word2vec or GloVe embeddings we discussed in the previous section. Essentially, we can use them to represent some text as a matrix. More specifically, consider the sentence ”Wait for the video and don’t rent it” (Kim, 2014). Each word can be represented as a $1 \times |V|$ vector, where $|V|$ is the dimensionality of the embedding - often set to 300. In practice, a sentence of length n is represented with the ids associated with the 300 dimensional vector representations of each word. The input sentence would thus look like:

$$M = \begin{matrix} 0 & 31 & 45 & 9 & 101 & 0 \\ 0 & 131 & 5 & 452 & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 623 & 1 & 0 & 0 \end{matrix} \tag{2.29}$$

Figure 2.17 shows a simple CNN architecture. The first matrix shows the vector representations of the same sentence.

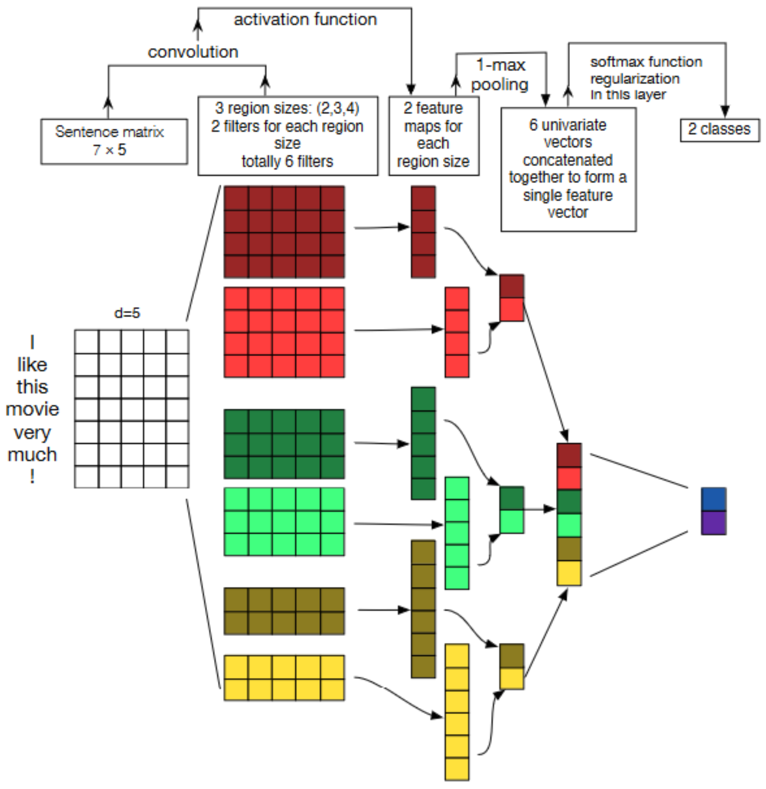

activation function
convolution
Sentence matrix 7 × 5
3 region sizes: (2,3,4) 2 filters for each region size totally 6 filters
2 feature maps for each region size
1-max pooling
6 univariate vectors concatenated together to form a single feature vector
softmax function regularization in this layer
2 classes
d=5
I
like
this
movie
very
much
!

Figure 2.17: *Model architecture with two channels for an example sentence* (Zhang and Wallace, 2015).

Consider the following notation from (Kim, 2014), we define a sentence $x_{1:n}$ as:

$$x_{1:n} = x_1 \oplus x_2 \oplus \ldots \oplus x_n \tag{2.30}$$

Convoluting a word x_i with its context is achieved through sliding a window, which size is defined empirically. To account for the full representation of every word we set this window to be always of the width of $k = |V|$. However, it's height can vary. Formally, a convolution is defined by the multiplication of a filter $w \in \mathbb{R}^{hk}$ by a window of words $x_{i:i+h-1}$, where h defines the height of the window. This results in a new feature. To compute it, we use a modification of Equation 2.4, we can introduce Equation 2.27 as:

$$c_i = f(W \odot x_{i:i+h-1} + b) \tag{2.31}$$

This operation is applied to a number of windows by sliding the window down by a given stride length until it reaches the end. In a cs224d lecture[5], Socher points out that it is useful to have different height sizes. Thus, forming a feature map $c = [c_1, \dots, c_2]$. Every feature map will then share the same weights with the rest. This also ensures a translation invariance, since it guarantees that regardless of where a given combination of words is located in a sentence, they will convey the same features. Moreover, every hidden layer can consist of multiple feature maps. Figure 2.17 shows this concept through the coloured rectangles. The outcome can be thought of some type of summary of the associated word compilation. That is usually achieved by passing the output through an additional layer, also known as a pooling layer. A very popular one is the 1-max pooling operation (Collobert et al., 2011).

2.5 Siamese Model

The Siamese architecture (Bromley et al., 1993) ranks the similarity between two inputs. It has mainly proved to be very useful in Vision tasks (Koch, 2015). Figure 2.18 is an example for such architecture. These models work by considering two distinct inputs through two separate neural networks (CNNs in the case of the figure referenced above) which are being trained against the same cost function. The network has two main features. Namely, it is symmetric and consistent. For example if a sentence S_1 is the hypothesis and a sentence S_2 is the premise, the network will learn to represent that pair in the same way even if S_2 was the hypothesis and S_1 the premise. In other words it will learn to represent an input which is split in two in the same way, regardless of which conjoining layer a part belongs to. Moreover, it guarantees that the relationship between the two sentences is going to be mapped in the same location in feature space since they are trained against the same cost function.

[5]http://cs224d.stanford.edu/lectures/CS224d-Lecture13.pdf

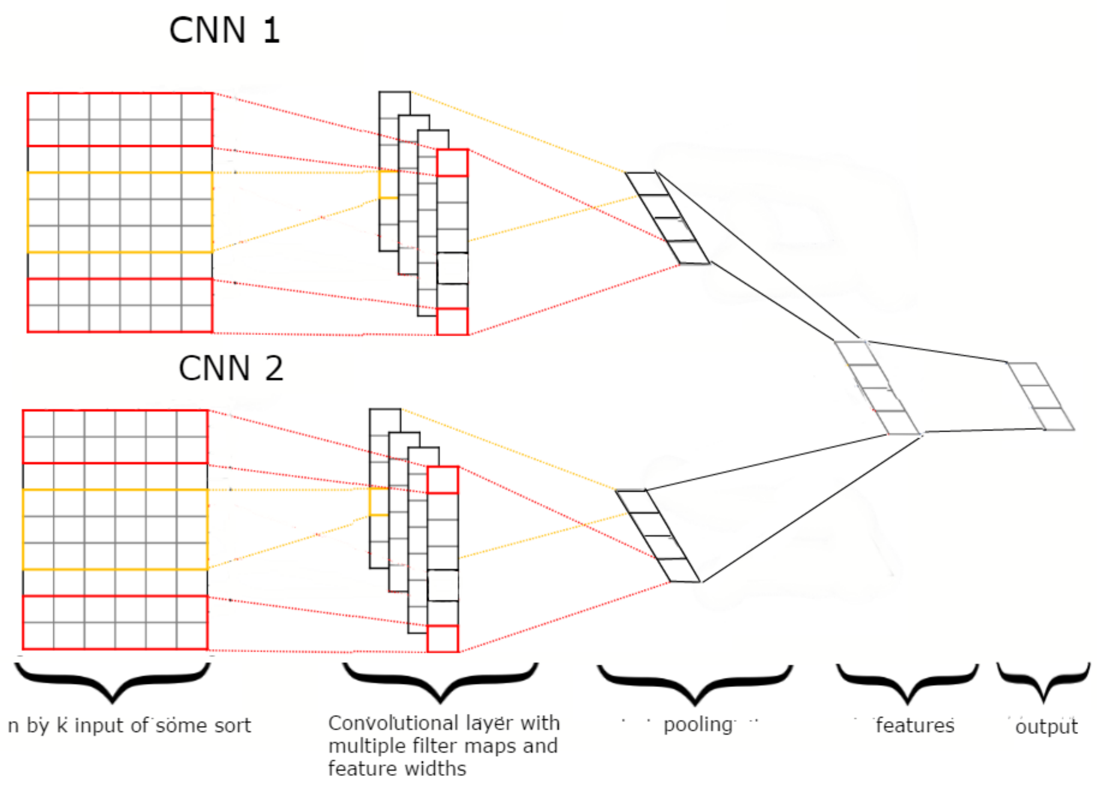

CNN 1
CNN 2
n by k input of some sort
Convolutional layer with multiple filter maps and feature widths
pooling
features
output

Figure 2.18: *Siamese Architecture which replicates the top and bottom sections to form twin networks. The first two matrices represent two inputs, regardless of their nature. The second layer of vectors are the feature maps, forming a CNN respectively with the pooling layers.*

Chapter 3

Modelling Entailment

The aim of this project is to find a simple yet successful way for sentence classification and more specifically for recognising entailment relations. We compare two neural network approaches and namely RNN-based architectures as proposed by Rocktäschel et al. (2015) and convolutional neural networks.

Throughout the course of work we begun with a thorough study of the SNLI data set. This helped us learn that the data set was comprised in such a way that it had more than one different hypothesis for every premise and that the data set was large enough to enable neural network training over GPUs rather than CPUs. This short study was followed by the actual implementation of the convolutional neural network this entire project is based on. That included the actual set up on the MSc students' cluster which turned out not to be a trivial task due to the type of the Unix operating system the cluster was running on, the permissions we had and the fact that we were the first from the MSc cohort who had to do that set up along with the numerous dependencies that had to be installed and configured.

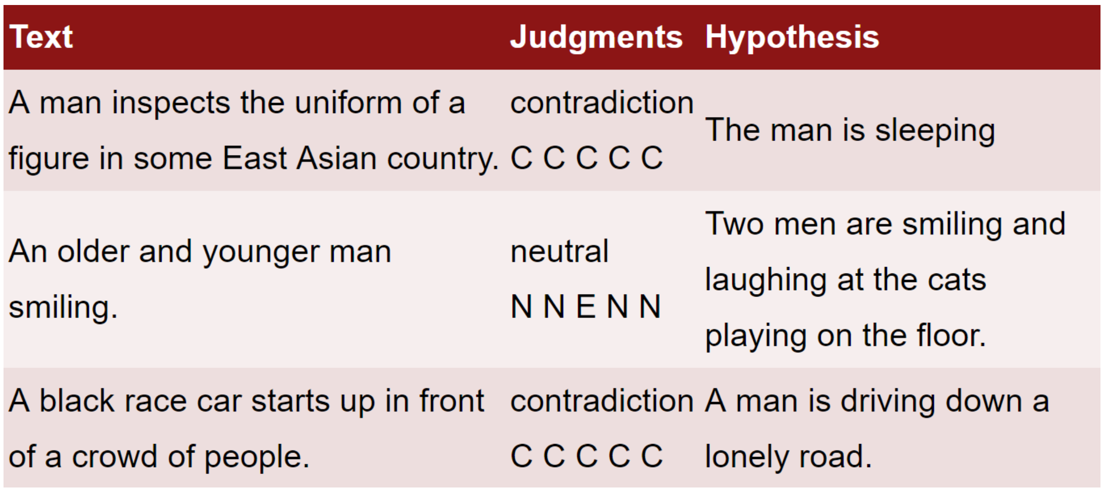
3.1 Entailment Relations with SNLI

The accurate processing of natural languages requires the ability to adequately determine the semantic relationship between a pair of sentences.

Currently, the largest known data set of labelled pairs is introduced through the Stanford Natural Language Inference corpus (SNLI) by Bowman et al. (2015). It consists of 570 000 pairs of sentences which have been annotated by humans.

Figure 3.1[1] consists of a few examples from SNLI that can help the reader to bet-

[1]http://nlp.stanford.edu/projects/snli/



Text	Judgments	Hypothesis
A man inspects the uniform of a figure in some East Asian country.	contradiction C C C C C	The man is sleeping
An older and younger man smiling.	neutral N N E N N	Two men are smiling and laughing at the cats playing on the floor.
A black race car starts up in front of a crowd of people.	contradiction C C C C C	A man is driving down a lonely road.

Figure 3.1: *Example pairs taken from the devtest portion of the SNLI corpus.*

ter understand the concept behind RTE. The sentences under 'Text' are the so called premises. These can be related in one of three ways with the given hypotheses (i.e. the sentences in the third column). Each row represents a labelled relationship between the given pairs. Another way of thinking about this is as a classification problem - the pairs can be one of three - contradicting, neutral or entailed. Therefore, the figure is said to illustrate 5 separate of each other examples. The relationship between the two sentences is defined by human annotators also known as turkers. In Figure 3.1, the column between the two sentences shows the judgements (each of which represented by a character) of five independent turkers as well as a final consensus judgement. For example, the first entry contains of a premise - "A man inspects the uniform of a figure in some East Asian country." and a hypothesis - "The man is sleeping". All five turkers have labelled the relationship between these two sentences with C (for contradiction). Thaking the average of their answers leads to labelling the overall judgements as being a "contradiction".

The field is constantly observing more and more precise ways of solving this problem. It has seen a number of state-of-the-art neural models including the one proposed by Rocktäschel et al. (2015) which achieves 80.9% accuracy with its baseline attention-based LSTM model on the Stanford Natural Language Inference (SNLI) data set. There are more complicated models which in fact achieve more than 87% (87.3% to be precise (Munkhdalai and Yu, 2016)). Their result comes very close to the currently known human accuracy on SNLI - 89%.

3.2 Simple Model

To ensure our work was correct, we used a model equivalent to the already published work by Kim (2014) (See Figure 3.2). The source code of their research was made available[2]. Thus, we used it as our starting point. In addition, we used the Movie

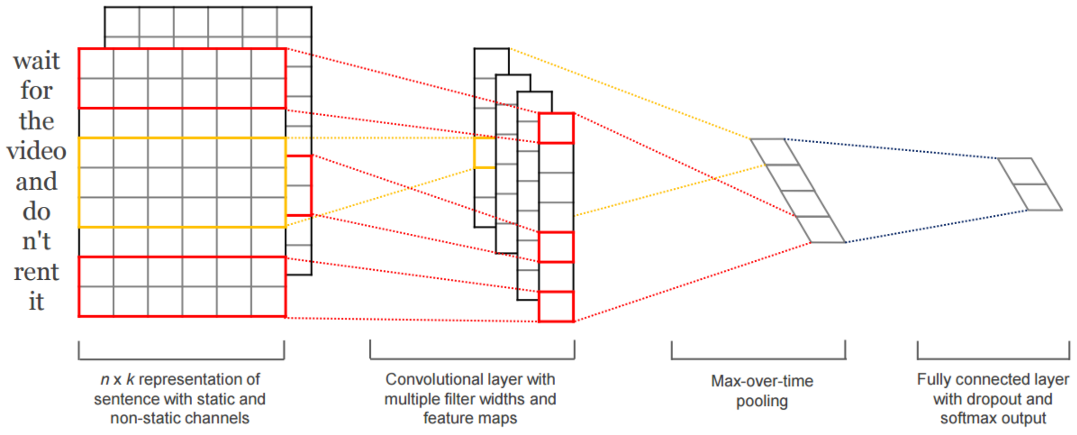

wait
for
the
video
and
do
n't
rent
it
n x k representation of sentence with static and non-static channels
Convolutional layer with multiple filter widths and feature maps
Max-over-time pooling
Fully connected layer with dropout and softmax output

Figure 3.2: *Model architecture with two channels for an example sentence* (Kim, 2014).

Review data set (Pang and Lee, 2005) and compared the results from our local set up with the ones reported in Kim (2014) and Zhang and Wallace (2015).

3.2.1 Implementation and Code Re-factoring

Their implementation, however, was not very suitable for our aims. The code was lacking a structure which is able to handle any further extensions. There were no help functions, hierarchical structure presented or in fact almost no logical separation in their implementation. For example, the architecture, mathematical operators, data pre-processing were all considered as a single train of thought and thus implemented in one method. The entire implementation was done in a single method which was called from the main function. There was, however, a good separation of the different layers which were implemented in a separate file and were modified primarily from deeplearning.net[3] and a couple of other sources which are referenced in the code itself. Therefore, we kept that single file intact.

[2]https://github.com/yoonkim/CNN_sentence
[3]http://www.deeplearning.net

However, we had to re-factor the majority of the code and make it scalable as well as easier to read. Moreover, we needed to be able to execute it multiple times with different settings. In addition, we needed something that is reusable. Thus, we referred to Richards (2015) as a guideline.

To achieve our goal, we could have used a Layered Architecture which would have essentially guaranteed the ability to separate the business logic - namely separating the adopted layers' class from the actual functionality. However, that seemed quite similar to what we already had- a cumbersome, difficult to modify and extend architecture which would not serve as a backbone to our future work but rather as a burden. On the other side, it would have enabled a system that is easier to test, something which turned out to be very important when working in Theano. Regardless, a monolithic, tightly coupled implementation would have had a negative impact on the overall scalability and usability of our work.

Using an Event-driven architecture, on the other hand, was closer to what we needed. We wanted to have a mediator which could essentially choose the correct model and run a training without having to change a lot. However, that architecture is more suitable for systems which have a number of steps and require some level of orchestration to guide the processes through. We did not have such requirements. If we were to choose that architecture, we would have had to introduce a number of very similar to each other functions. That, essentially, would have enabled us to call them from some sort of a host or an event-mediator, as per the definition of that model. Moreover, that implied the need for splitting the actual business logic of a single neural network model, thus promoting the design of an asynchronous system. This, immediately increases the difficulty of implementation and limits the testability of our work. Since we are using Theano, we had to make sure that testing is as easy and as accessible as possible. On the bright side, this architecture would have essentially allowed us to quickly implement new, reliable functionality and easily adapt our system to work with it. Regardless, as already stated, we needed something that promotes easy testing and this architecture does not.

We wanted to have a core system which could accept various models, different settings and yet enable the adoption of different data sets. Thus, the microkernel architecture seemed as something that can fit to our needs. Behind the very concept of a microkernel architecture is the so called plug-in components which contained their own features and custom code, extending the functionality of a system and producing additional business logic. That is exactly what we wanted to have. We needed tools for pre-processing different data sets, different types of embeddings, introduce new neural network models and yet in the core, use the same definitions of neural network layers, mathematical operators and means of testing. However, this architecture introduces an extra complication. The system must always keep track of what is available and how to access it. This would have essentially meant maintaining some XML or similar file which is kept updated with the currently available plug-ins, for example, and that affects scalability. In contrast, we need to be able to create a lot of models and easily change and pick between them. Therefore, the lack of scalability as well as the potential of over-complicating the task drove us away from this architecture.

Thus, after a series of trials, we ended up designing our own. We split the code in roughly three main parts, similar to the basic MVC architecture. However, instead of View, we had our different CNN models. In addition, the controller was split into subparts and namely an initialisation part responsible for handling the different settings available for a chosen view and data pre-processing part, which took care of processing and setting up the data sets as per the requirements for the model selected. It should be very clear that there is a subtle difference between the model from the Model-View Controller architecture and the models that we refer to in this thesis. The latter are architectural models which address the actual problem we solve in the context of neural networks. Thus, they model the structure of the NN that we build. The model of our system architecture itself addresses the data and the very logic behind it. In our case these were the SNLI and the MR data sets. From now on, we will refer to the architectural decisions related with the modelling of the neural networks as ’model’. Figure 3.3 shows a tree structure representation of the logic separation for the project.

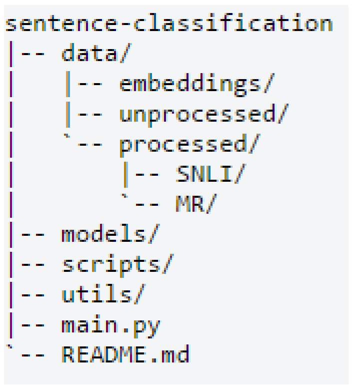

```
sentence-classification
|-- data/
|    |-- embeddings/
|    |-- unprocessed/
|    `-- processed/
|         |-- SNLI/
|         `-- MR/
|-- models/
|-- scripts/
|-- utils/
|-- main.py
`-- README.md
```

Figure 3.3: *Tree structure of the project.*

After re-factoring the code and splitting the logic so that it serves our purpose, we wanted to test it against results which were already listed in Kim (2014) and Zhang and

Hyper-parameter	Value
learning rate method	Adadelta
learning rate decay	0.95
hidden units	[100,2]
number of epochs	25
filter region	[3,4,5]
batch size	50
s^2	9
dropout rate	0.5

Figure 3.4: *Hyper-parameters as reported in* (Kim, 2014).

Wallace (2015). To achieve this, we used the MR data set[4]. We ran a 10-fold cross validation (CV) and used word2vec as embeddings. Moreover, we trained against different filter heights, namely 5 and 7, as adopted from the aforementioned publications. We fixed the hyper-parameters reported in Table 3.4 and compared our results to the ones reported in Zhang and Wallace (2015). We varied the filter heights and compared the outcome with the baseline reports. We tried $\{[3,4,5],[7,7,7],[7,7,7,7]\}$. We also compared our results against different activation functions: $[TanH, Iden, ReLU, Sigmoid]$ as well as both static and non-static word2vec input representations. We ran between 5 and 10 jobs per setting. The number varied as some of the jobs failed due to lack of memory allocated on the cluster's GPUs. For all of the tests described above, our results were within the boundaries reported in Zhang and Wallace (2015). Once we were sure the re-factored implementation from Kim (2014) worked correctly, we were ready to extend that to the purposes of our thesis.

3.3 Building Extended Models

3.3.1 Accommodating the model to work with SNLI

The next step towards our goal was to use the model described above with SNLI. To achieve this, we had to pre-process the SNLI data set. That meant creating a vocabulary with all words and embedding them with word2vec or GloVe. We concatenated the hypothesis and the premise sentences together and embedded them as a single in-

[4]https://github.com/yoonkim/CNN_sentence

put. In addition, we had to also account for representing the actual labels and think about conveying any additional information we might need in future. We decided to use [0,1,2] as the classification labels. We used 0 for pairs which were labelled as 'entailment', 1 for sentences which were 'neutral' and 2 for the contradicting ones. Moreover, we realised that it will be useful to keep track of the number of words existing in a pair. That would essentially help us when padding the data, for example. We also keep track of the type of data set the pair belongs to: i.e. train, test or validation. We also kept an index helping us refer to that particular text.

Unlike Kim (2014), we decided to use GloVe for the embeddings as global vectors tend to perform better than word2vec, in general (Pennington et al., 2014). In this scenario, we did not have to use cross-validation as SNLI has its own validation set which meant we had to account for that as well. This defined our baseline model which served as a comparison to our progress thereon. Visually, it looked very similar to Figure 3.2. However, instead of two channels as shown on that figure, it had only one. We used static vector representations as per the paper.

Overall, we did not expect to obtain high results. It was a basic CNN architecture and our baseline was built with much smaller data sets in mind. Moreover, the existence of the same premises for different labels was going to interfere with the correctness of the network. Instead, we could use that to our advantage and account for the existence of a premise and hypothesis. Thus, introducing the concept of the next subsection.

3.3.2 Extending the Model

We extend on the basic model above by introducing a Siamese-like architecture. In the core of our work is that we split the premise and the hypothesis and use them as inputs in two separate CNNs. Hence, taking into account the different roles of both sentences. We process their outputs and use their convolved representation as input to a third CNN which acts as the actual classifier to our task as described in Kim (2014). This is then followed by a single feed-forward layer and a Softmax function.

3.3.3 Siamese-like 3-CNN-wide structure

As already claimed, we do not think that the basic model will produce good results. Having the premise and hypothesis concatenated and used as a single input means that half of the input is the same regardless of the class under which the relationships fall into. This will essentially have a negative effect over the shared weights. Thus, splitting the two sentences and learning two separate weights for each of them made a lot of sense. However, this would still not be enough. In order to learn from both sentences simultaneously, we need to make sure that we train them against the same cost function. That way we would allow for our model to learn from both sentence representations. So far, this model inherits a lot from the Siamese architecture proposed in Bromley et al. (1993). Alternatively, we could have trained both sentences separately on separate cost functions and then use them as some sort of embedding representations in a third CNN. However, that would not learn anything as those sentences would have been trained on their own, thus ignoring the correlation between a premise and a hypothesis.

Regardless, the proposed by us structure so far, does not account for the combined/full representation of the sentence pairs. Even though the premise and hypothesis were trained against the same function they were still treated as entirely separate inputs. We wanted to represent the relationship between the two sentences with a mathematically expressed interaction between them. Thus, we needed a way of combining them through another layer. We used a third CNN which accepts as input the concatenation of the two outputs from the previous two CNNs. We then treated the problem as our initial model. Hence, we used a feed-forward layer and classify the output as before. Figure 3.5 shows a graphical representation of the resulted architecture.

3.3.4 Additive and Multiplicative Models

As we already hinted, a good way of modelling the relationship between two correlating sentences is through exploiting their semantic composition with basic mathematical operations (Mitchell and Lapata, 2010). Thus, we implemented a number of additive and multiplicative functions as described in this subsection.

3.3.4.1 Additive Models

The logical composition of languages enables the construction of complex structures through simpler ones. A property explained in Partee (1995) as "The meaning of a whole is a function of the meaning of the parts". To model this relationship successfully we assume that both sentences have the same dimensionality, are in the same language and use the same means of representation. That enables us to introduce simple compositional and additive models. We begin with element-wise, weighted and non-weighted addition and subtraction. This, however is based on very strong assumptions. As pointed out by Mitchell and Lapata, we are assuming that different parts of speech exist in the same dimensionality which itself implies the equality of the different parts of speech. However, we must take into account that we are not working with the words themselves but rather with their representations which were word2vec or GloVe embeddings that have been additionally processed through two separate CNNs. It is thus safe to assume that they can come from the same-dimensional space. Hence, we lose the strength of the assumptions made so far. If that is indeed the case, then weighing the vector representations should not impose any benefit whatsoever. In contrast, subtracting the two vectors will introduce a distinct representation between both vectors. Hence, it will improve the accuracy of the model. The work in Mitchell and Lapata (2010) summarises the functionality of additive models as "blending together the constituents they are composed from".

3.3.4.2 Multiplicative Models

All of the methods described above classify as additive models. We do consider multiplicative models as well. In other words, we will scale one vector with its relevance to the other. This, however, does not seem right for the purpose of our task. As already noted, the data set we observe considers the same premise with different hypothesis, leading to different classifications thereby. Scaling with relevance will deem poor results for those scenarios. Which will endorse a significant dose of randomness. Elementwise multiplication is a symmetric function, hence it disregards the word order and syntax (Mitchell and Lapata, 2010). Circular convolution is another member of the multiplicative models. It aims at correcting the length of the vector but does not affect its direction. This again seems unlikely to improve the performance for our task. However, we base this discussion on a number of assumptions. Thus, we decided to implement them all and analyse the outcome of using them in the next chapter.

Although most of the models described above were straightforward to implement given the project architecture we use and the correctness of our work, the circular convolution turned out to be not a trivial task. The main reason for this was that Theano did not support any 1D convolutions at all and the existing implementations in Lasagne did not seem to achieve what we needed. Therefore, we had to find a way of implementing our own function by making use of the already existing Theano functionality. The solution was to perform a 2D convolution over the two tensors. However we had to revert the second one and account for tensors with odd width.

Even though odd numbered tensors are unlikely for the purpose of our task (as a tensor is comprised of two equally sized vectors), we aim at re-using this function in future. Finally, we had to sum over the first axis, reshape and revert the output.

3.3.4.3 Extended Models

A logical extension of the models defined so far is a basic permutation of the combinations in between them. For example, we can populate a vector with n different outputs from applying n different operations. That, essentially, can increase the width of the input to the third CNN which will increase the number of parameters and thus potentially improve the performance as it extends the specificity of the problem learnt. Moreover, it meant that we can combine different outputs. This meant taking advantage of the benefits from using both addition and subtraction, for example. Using concatenation in collaboration with the two would allow for further increasing the size of the parameters, which can proof useful. In contrast, this will further complicate the task and hence increase the correlation between the hyper-parameters we will need to optimise.

3.4 Summary

The novelty of this model, even though it introduces a large number of parameters, is its simplicity. Essentially, we are using the same architecture for the 3 CNNs and connect them in a sensible way. If we compare this to the Attention-based LSTM model introduced in Rocktäschel et al. (2015), we attempt to show that modelling entailment as a task has been over-complicated to accommodate a model which does not perform as well in classification tasks. In the next chapter, we will evaluate our model through series of tests and hyper-parameter optimisations in attempt to show that our model is both simple and performs better than the one presented in Rocktäschel et al. (2015).

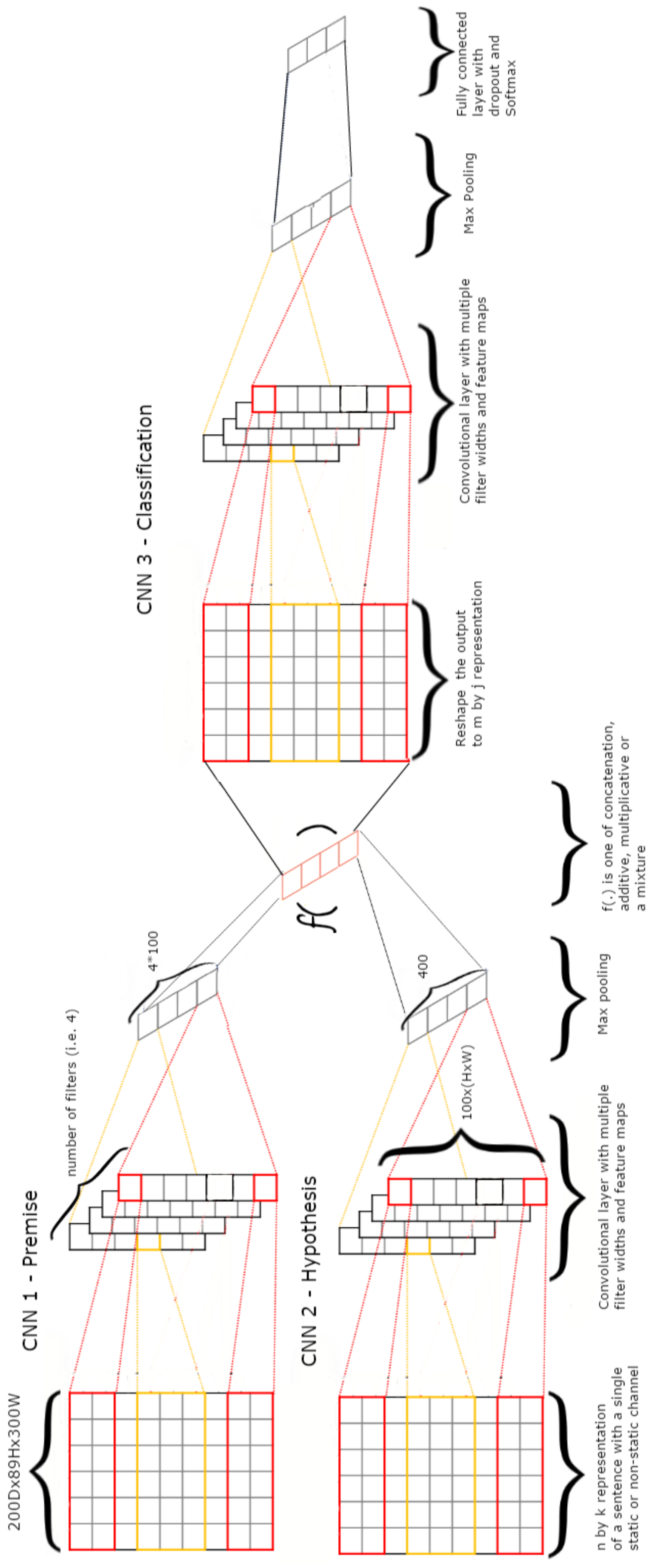

200Dx89Hx300W
CNN 1 - Premise
number of filters (i.e. 4)
4*100
CNN 2 - Hypothesis
100x(HxW)
400
f( )
n by k representation of a sentence with a single static or non-static channel
Convolutional layer with multiple filter widths and feature maps
Max pooling
f(.) is one of concatenation, additive, multiplicative or a mixture
CNN 3 - Classification
Reshape the output to m by j representation
Convolutional layer with multiple filter widths and feature maps
Max Pooling
Fully connected layer with dropout and Softmax

Figure 3.5: *Siamese-like 3-CNN-wide model.*

Chapter 4

Test and Evaluation

Having set a model to base our work on, we used the SNLI data set to obtain our first results. As already stated, we suspected that they will not be very good given the simplicity of the model. After improving the model to a Siamese-like 3-CNN-wide neural network and optimising the hyper parameters, we ended up achieving a state-of-the-art result of 81.26%. We used rectified linear units, filter windows (height) of [2,3,4,5] with 100 feature maps each, a dropout rate of 0.13, l_2 constraint (s in the code) of 3, learning rate decay of 0.95, a mini batch size of 200, non-static word2vec embeddings, 25 $\times$ 64 ratio of the *height to width* input for the third CNN.

4.1 Initial Tests

4.1.1 Baseline Model

As a baseline, we used the CNN-static model from Kim (2014) along with the same settings for the hyper-parameters as in their implementation (see Table 3.4). A CNN model is defined as static if it uses pre-trained word2vec embeddings where all words are kept static. Similarly, we define a model that fine-tunes the pre-trained vectors for each task as non-static (Kim, 2014). However, we set the output units to 3 instead of 2. We obtained an average result of 66.55% over the test data set (results varied between 66.11% - 66.99%).

Despite the low outcome, we used grid search to see if we can optimise it. And indeed we did, or at least to a certain degree. For example, given that the size of the data set was substantially larger than SST-2 (the data set used in Kim (2014)), reducing the dropout improved the results obtained by roughly 2% (See Table 4.3). In addition, increasing the batch size from 50 to 200 decreased the run time of each epoch from 313ms to about 210ms as shown in Table 4.2. It was, however, compulsory to use mini-batches given the size of the dataset and the limited resources we had.

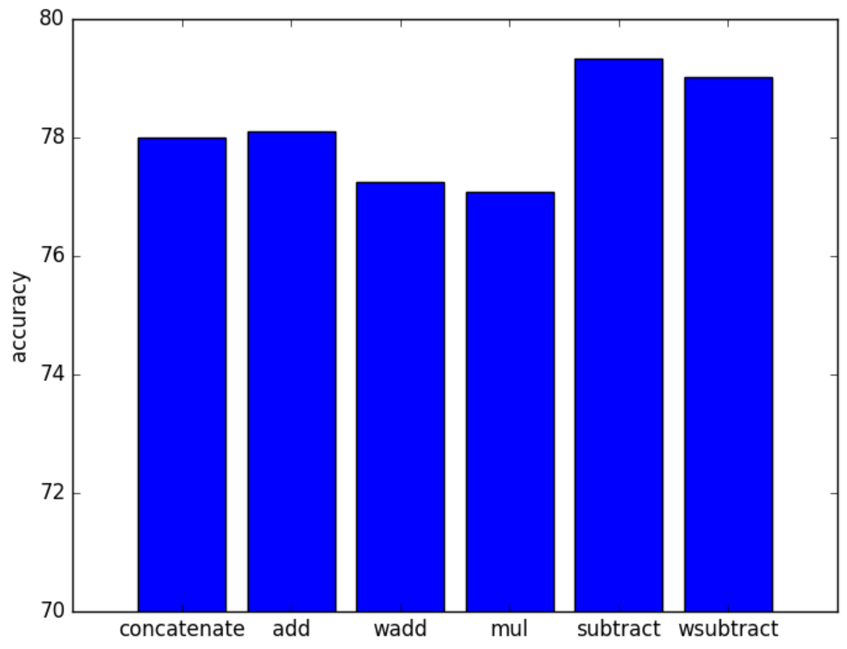


Figure 4.1: *Achieved results for different operators.*

In addition, given that we based our work on Kim's implementation, we are also using Adadelta. Thus, we didn't have to worry about decreasing the learning rate after increasing the batch size. Additionally, we used GloVe (Pennington et al., 2014) as opposed to word2vec (Mikolov et al., 2013) for the embeddings. This meant implementing scripts for preprocessing both the SNLI text files and embedding the result using GloVe. Overall, the best result we achieved was 68.11% after completely removing the dropout (See Table 4.3). In addition, we compared the performance between

Batch Size	Time per epoch
50	313ms
100	260ms
150	239ms
200	**210ms**

Figure 4.2: *Run time per epoch with respect to batch size.*

using a ReLU and Tanh as a non-linear function. As expected, the former performed better, TanH, in general is very difficult to adjust. For comparisons refer to Figure 4.3 and 4.4.

Dropout	Train	Validation	Test
0.5	69.24%	66.09%	66.55%
0.2	72.69%	66.91%	67.32%
0.1	73.71%	65.92%	66.55%
0.0	**75.25%**	**67.57%**	**68.13%**
0.7	66.44%	64.62%	64.64%

Figure 4.3: *ReLU as a non-linear function.*

Dropout	Train	Validation	Test
0.5	70.96%	68.05%	63.76%
0.2	74.00%	66.91%	58.48%
0.1	79.99%	67.56%	61.96%
0.0	75.86%	67.44%	62.04%
0.7	**70.84%**	**67.35%**	**64.60%**

Figure 4.4: *Tanh as a non-linear function.*

4.1.2 Extended Model

Next, we extended the model by implementing a Siamese network. we used separate CNNs for the premise and the hypothesis. However, the output of each of them was concatenated and used as an input to a third CNN which was then followed by a feed-forward neural network as per the initial model (see Figure 3.5). All three CNNs were trained against the same cost function simultaneously. This substantially improved the accuracy with about 10%. We achieved 77.22% with the adopted parameters used in Kim (2014) and 77.99% using 0.2 dropout as shown in Figure 4.5. In this section onwards we are assuming a fixed dropout of 0.2 unless otherwise stated. We are also using a batch size of 200 as well as ReLU as a non-linear activation function and 0.95 as a value for the learning rate decay. Although there is a good chance this won't end up being the most optimal setting as already noted, it allows for achieving reasonably high results to the point where we find an optimal model which we will further optimise. It should also be noted that we were merely concatenating the two

Dropout	non-linear function	Train	Validation	Test
0.5	relu	84.92%	75.05%	76.53%
0.2	**relu**	**88.20%**	**76.99%**	**77.99%**
0.2	tanh	90.71%	77.40%	69.7%

Figure 4.5: *Alternating dropout and non-linear function. Siamese-like Model.*

outputs from the first two CNNs before running them through the third one. However, that approach was not taking advantage of the actual dimensional similarities of the two outputs before processing them through a third CNN. Thus, we knew we should, in theory, be able to achieve more than what has so far been reported.

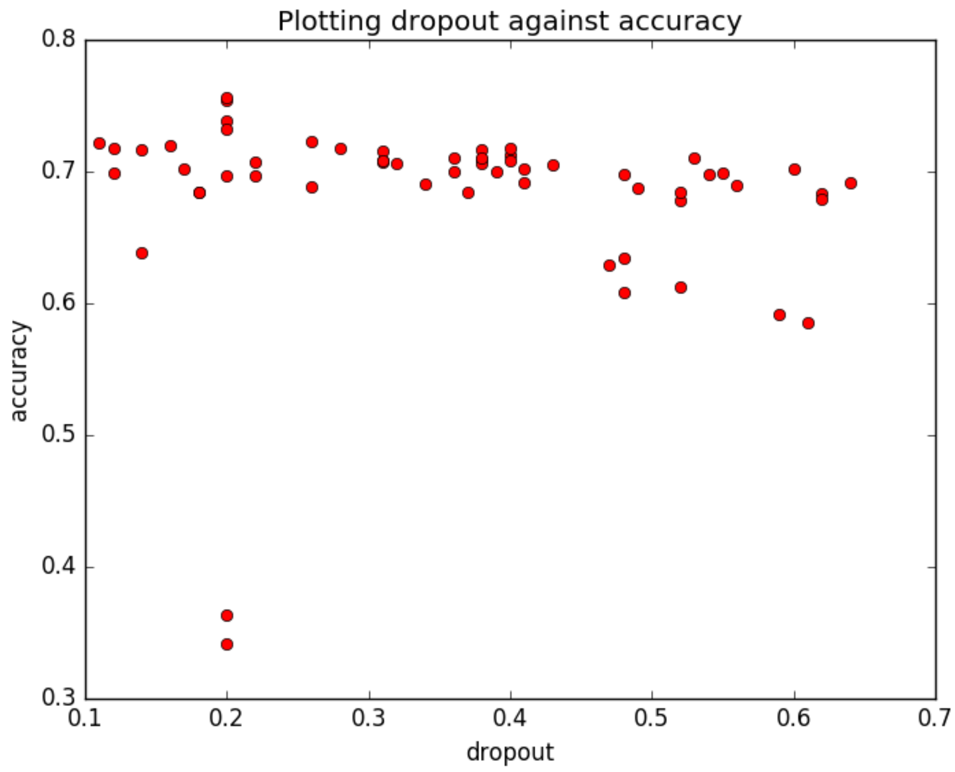


Figure 4.6: *2D relationship between accuracy and dropout.*

4.1.2.1 Linear operations

To improve the accuracy we implemented different linear operations which we used instead of concatenating the two outputs and later on in a combination with the concatenated outputs as well. We implemented the following additive and multiplicative models: addition, weighted addition, subtraction, weighted subtraction, multiplication and circular convolution. Then, we started optimising the hyper-parameters associated with the models.

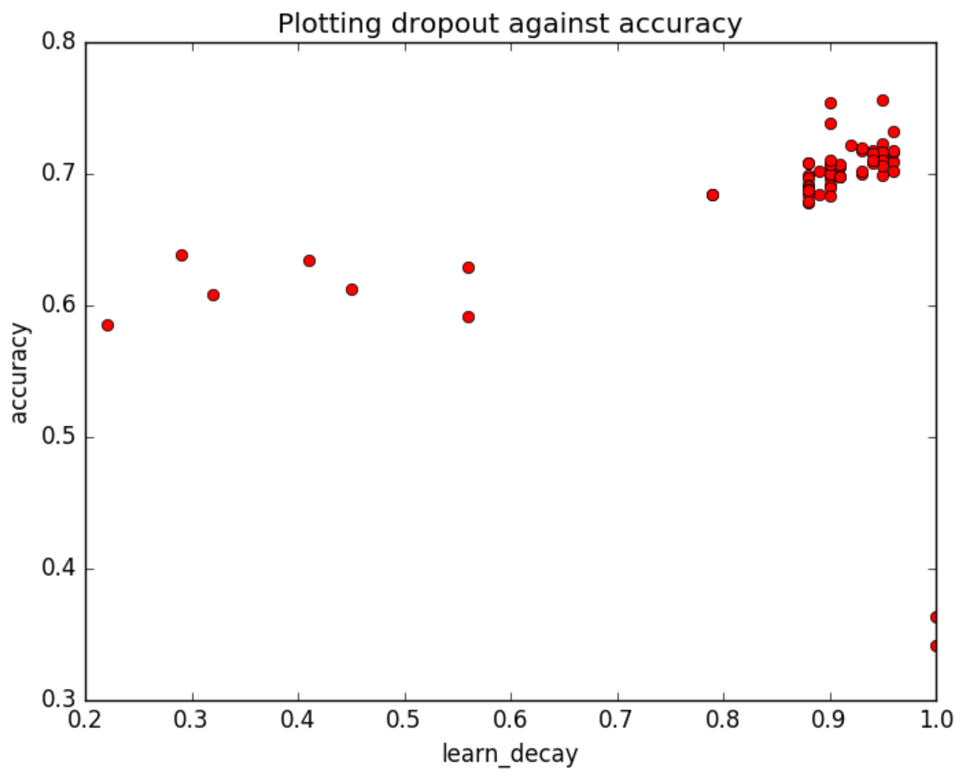


Figure 4.7: *2D relationship between accuracy and learn decay.*

4.1.3 Randomised Parameter Optimisation

To reduce the time required for hyper-parameter search we used a worker which continuously sampled random hyper-parameters and performed the optimisation on a smaller network comprising of 5 epoch training and smaller batch size. Although the smaller batch size increases the time per epoch it reduces the resources required for a network training which itself reduced the number of failed jobs on the MSc cluster. During the training, the worker kept track of the training and validation performance, the per-epoch time and wrote them down to a file as advised in Stanford University's cs231n [1].

4.1.3.1 Dropout and Learning Rate

The worker was sampling the most common hyperparameters in the context of Neural Networks (namely learning rate decay and dropout). It also took into account the relatively less sensitive hyper-parameters, for example the weights in weighted ad-

[1]http://cs231n.github.io/neural-networks-3/

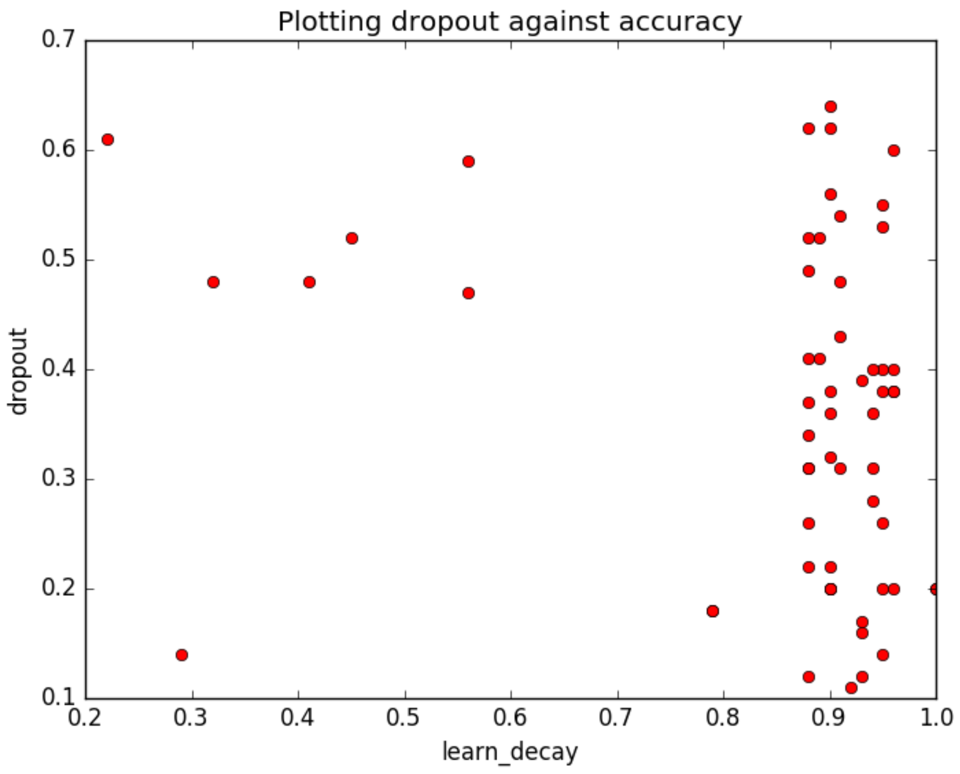


Figure 4.8: *2D relationship between dropout and learn decay.*

dition. The search was performed in hyper-parameter ranges as displayed in Figure 4.6

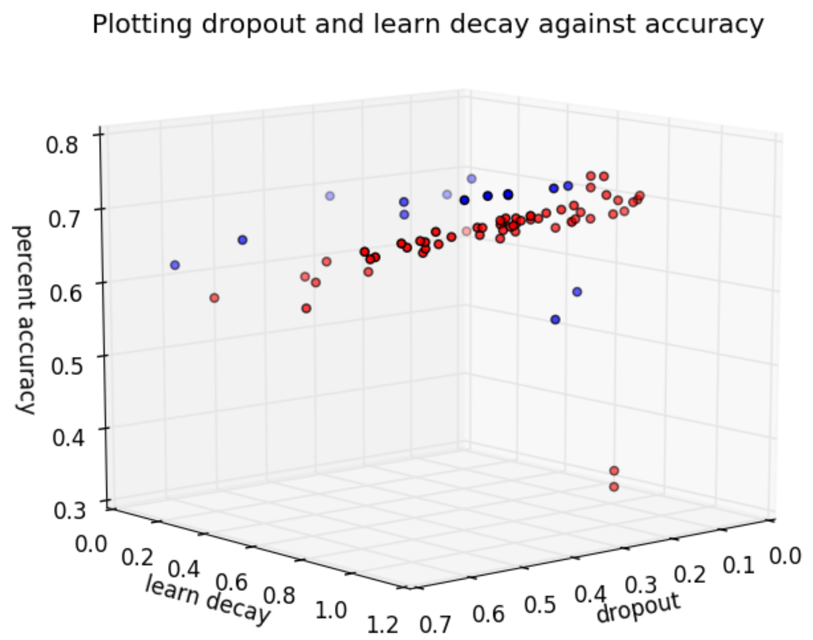


Figure 4.9: *3D Plot of the relationship between dropout, learning rate decay and accuracy. Red: mul, Blue: add. Batch size used is 50, hence the results obtained are low in general.*

and 4.7. We used 100 separate trainings for multiplication and addition. The two plots show the results obtained from the former model. Moreover, the two figures show the dependencies between dropout and accuracy as well as between learn decay and accuracy. Note that in Figure 4.7 most sampled points are between 0.88 and 0.96 as those seem to achieve the highest accuracy. Figure 4.6, however, shows a rather uniform sample of dropouts. It can be noticed that the lower dropout value yields higher performance. However, both dropout and learn decay were correlated, as shown on Figure 4.8. One can observe that the majority of the samples were taken for dropout which is smaller than 0.4 and for learning rate decay which is higher than 0.88.

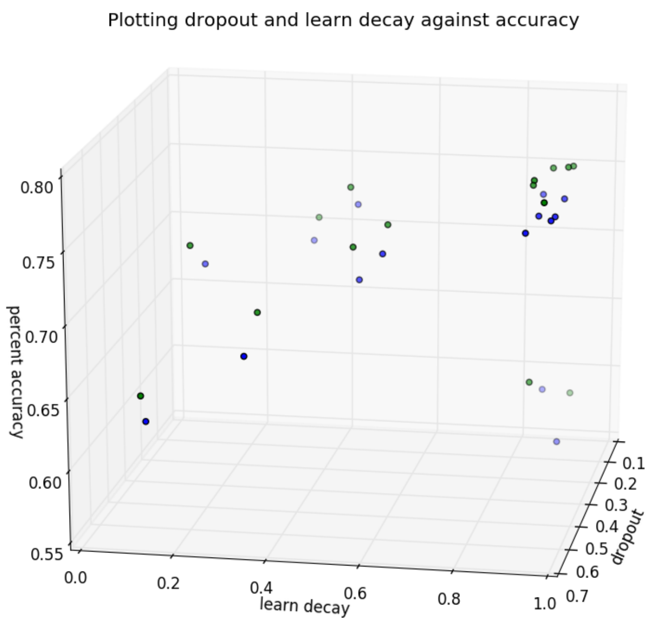


Figure 4.10: *3D Plot of the relationship between dropout, learning rate decay and accuracy. Green: subtr, Blue: add. Batch size used is 50, hence the results obtained are low in general.*

4.1.3.2 Simple Linear Operators

Figure 4.9 shows the relationship between the learning rate decay, dropout and the accuracy achieved. The red dots indicate the Siamese-like model ran using multiplication between the outputs of the first two CNNs and the blue dots indicate the same however using addition instead of multiplication. It can be noticed that addition performs slightly better than multiplication in all cases. An interesting observation at that moment was that neither basic addition nor multiplication could achieve results that differ by much from the simple concatenation (See Figure 4.1).

Similarly, subtraction outperformed addition by 1.5% on average as shown in Figure 4.10. This suggests that the assumption made in the previous chapter is not as strong just as anticipated and that we had a result which differed from the basic concatenation by a considerable margin of between 2-3%.

4.1.3.3 Weighted Addition

Figure 4.11 shows the results from sampling values for weighted addition. We define with α the addition weights associated with the premise and with β the ones associated with the hypothesis. It can be seen that weights do not lead to any improvement of performance since the best results stem from the cases where $\alpha = \beta = 0.5$. Note how two of the results are very low even though the weights are almost identical. The reason for this is that these runs have been performed with TanH non-linear activation function as opposed to ReLU.

4.1.3.4 Circular Convolution

Initially, we thought that circular convolution could highlight the importance of certain regions of the resulting vector. However, this turned out not to be the case. The 1D convolution was achieving 34% accuracy when ran on its own which could potentially mean it is not doing anything. We knew that tests performed on models that use circular convolution had to be split to mini-batches as opposed to the other operators which could use the full batch at a time, thus implying slightly worse performance. The reason for this is the memory constraints of the cluster's graphic power or in fact of any currently existing GPU configuration's limits. After testing the cyclic convolution along with some of the other operations we had implemented, we kept obtaining results which were still at least 2% on average lower than the cases which did not make use of 1D convolutions at all. Therefore, we concluded that circular convolution did

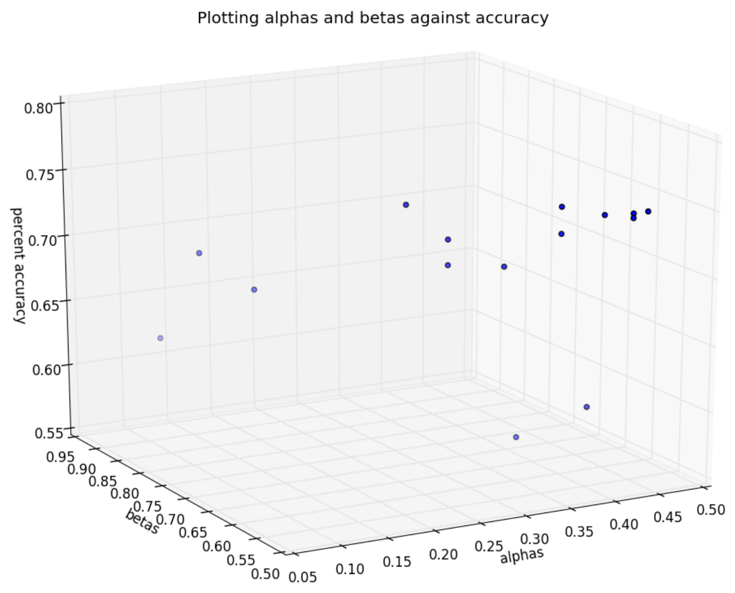


Figure 4.11: *3D Plot of the relationship between* α*,* β *and the associated accuracy.*

not work as we anticipated and thus we do not use it in the discussions to follow.

4.1.3.5 Static and non-static parameters

Applying the same tests we have discussed so far to a non-static model, yielded slightly better results - for example, the model which uses subtraction results in 79.90% accuracy as opposed to 79.24% when run against a larger batch size. Similar observation was made in both Kim (2014) and Zhang and Wallace (2015).

4.1.3.6 Window Size

As already explained, the kernel size in NLU tasks tends to only change in height and not width. The reason for this is that we are interested in modelling the relationship between neighbouring words as opposed to looking at neighbouring letters. To address this issue, we test different window shapes. More specifically, we look at filter heights comprised of three or four different sizes. To examine the performance we claimed that keeping a relatively small size of the window will improve the performance of our model. Therefore, we had to prove that increasing the heights will decrease the overall

Filter Height	Function	Train	Validation	Test
3,4,5	sub	92.80%	79.19%	79.41%
7,7,7	sub	93.52%	78.04%	78.95%
1,3,5,7	sub	93.86%	78.74%	79.94%
2,3,4,5	**sub**	**93.42%**	**79.57%**	**80.60%**
7,7,7,7	sub	93.15%	78.26%	80.05%

Figure 4.12: *Alternating filter heights.*

performance. To achieve this, we used substitution as the function which combines the outputs from the first two CNNs. First, we examined how our basic setting of [3,4,5] compares to larger windows of size [7,7,7]. This meant changing the padding to 7 for the latter scenario. Moreover, increasing the window size increased the memory requirements which did not allow for bigger windows. It turned out that on average the larger heights reduce performance, therefore by induction our claim was correct. To further examine this, we increased the size of the window filters by one. As reported in Table 4.12, we learnt that more filters and relatively smaller filter heights resulted in better performance. This, in fact, made a lot of sense. Increasing the number of filters meant increasing the number of feature maps which itself increased the number of parameters trained. Moreover, keeping the size of the kernels relatively small and varying it meant learning different convolutions of local regions and enabling more accurate pooling. In contrast, if the filter size was larger and there were not as many words in a sentence, the pooling would essentially summarise the actual sentence on a smaller scale since it will be looking primarily at zeros. This meant it will result in poorer performance. Moreover, varying the size of the kernels introduced the concept of learning different representations so that when summed up, the result will have a more accurate representation of the input that is being convolved with the kernels. An interesting observation is that the list of consecutively numbered window sizes deemed the best performance. We expected to show that a list of odd window heights performs better as there would always be a pair which is located in the middle of the window. However, Table 4.12 shows that this did not turn out to be true.

Hyper-parameter	Value
learning rate method	Adadelta
learning rate decay	0.95
hidden units	[100,3]
number of epochs	25
filter region	[3,4,5]
batch size	50
s^2	9
dropout rate	0.2
model	mix1

Figure 4.13: *Embeddings performance comparison.*

4.2 Further Work

4.2.1 Embeddings Evaluation

It is also important to note that we used Zhang and Wallace (2015) as a guideline while fine-tuning the variety of hyper-parameters.

The paper reports that word2vec outperforms by just a bit GloVe embeddings for sentence classification tasks. Thus, we wanted to learn if that was the case for our problem setting too. For that purpose, we used the reported in Table 4.13 hyper-parameters. We ran 10 tests with word2vec and as many with GloVe. Overall, obtained 79.96% when using word2vec as opposed to 79.04% with GloVe as reported in Table 4.14. Having recorded a difference of almost one percent helped us realise that using word2vec was more beneficial to our task. Thus, we will be using it from here on unless stated otherwise.

4.2.2 Mixed Models Performance

4.2.2.1 Settings

After we obtained a vague idea of what values might prove useful and of the problem itself, we decided to extend the size of the input for the third CNN. Thus, we designed 9 different mixtures which are reported in Table 4.15. An important observation is that the inputs to the third CNN will differ depending on the number of operations concatenated in a list as well as on the length of the filter window sizes. For

Embedding	Model	Dropout	Train	Validation	Test
word2vec	mix1	0.0	93.41%	78.94%	79.80%
GloVe	mix1	0.0	90.36%	78.75%	78.84%
word2vec	**mix1**	**0.2**	**92.95%**	**79.41%**	**79.96%**
GloVe	mix1	0.2	90.42%	78.74%	79.04%
word2vec	mix2	0.2	92.94%	79.40%	79.79%
GloVe	mix2	0.2	90.35%	78.77%	78.60%

Figure 4.14: *Hyper-parameters used for the embeddings comparison. Reported results are from the test set.*

Hybrid Model Name	Consists of
mix 1	[concatenation,addition,subtraction]
mix 2	[concatenation,addition,multiplication]
mix 3	[concatenation,subtraction,multiplication]
mix 4	[addition,subtraction]
mix 5	[addition,multiplication]
mix 6	[subtraction,multiplication]

Figure 4.15: *Hybrid models contents.*

every window we will add 100 entries (since that is the size of the feature maps) for each output of the first two CNNs. Thus, for a standard setting of $[3,4,5]$ filter window sizes, mix 1 will have input of size 1200 (600 from concatenating the two outputs and two times 300 from the addition and multiplication).

4.2.2.2 Evaluation

Now that we had a rough idea about the hyper-parameter values as well as the importance of the input ratios, we compared the performance of the different mixed models. So far, we know that subtraction yields the best performance, followed by addition, basic concatenation, multiplication and lastly performing a cyclic convolution. Thus, we suspected that the models which contained addition and subtraction will end up producing the best performance. And so they did. The increase of the input's size introduces a potential for more parameters. In order to circumvent this and examine the actual operators' performance, we reduced the width of the reshaped inputs to the third CNNs. We used about a fifth of the overall size of the inputs. Table 4.17 shows

Model	Train	Validation	Test
mix 1	**85.74%**	**78.93%**	**79.27%**
mix 2	79.57%	73.89%	75.85%
mix 3	87.97%	78.24%	78.47%
mix 4	85.27%	78.36%	78.98%
mix 5	83.46%	76.88%	77.47%
mix 6	**88.49%**	**78.36%**	**79.27%**

Figure 4.16: *Ratio from* 116×10. *Results are averaged. Note some of the models does not add up to 1600.*

Model	Train	Validation	Test
mix 1	**93.15%**	**79.67%**	**80.47%**
mix 2	90.86%	77.55%	78.05%
mix 3	93.42%	79.97%	80.36%
mix 4	93.46%	79.69%	80.10%
mix 5	91.39%	78.15%	78.59%
mix 6	93.34%	79.79%	80.18%

Figure 4.17: *Ratio from* 50×32. *Results are averaged. Note some of the models does not add up to 1600.*

the different results we obtained after running 10 separate tests with each of the new models. Noticeably, the differences between the results with and without the concatenation fraction were negligible. Hence, we concluded that the only benefit considering the actual concatenation of the outputs from the initial two CNNs is only to the extend where it enabled us to increase the number of parameters we would train overall. Similarly, we could have used two additions and two multiplications instead of having a concatenation with an addition and a multiplication. That is why, for example, when training with using a 1 to 1 ratio of the input to the third network we achieved about a 0.25% increase of the performance.

4.2.3 Third CNN's Input Shape

Another hyper-parameter we had to test was the shape of the input for the third CNN. So far we have been using values which were close to $300/81$ ratio - the same as the ratio excluding the padding for the first two CNNs. Thus, the values we used

Ratio	Train	Validation	Test
5×240	85.01%	77.68%	78.31%
10×120	85.56%	78.47%	78.87%
12×100	85.66%	78.31%	79.28%
15×80	86.77%	79.07%	79.49%
20×60	**87.08%**	**79.38%**	**80.06%**
100×12	86.46%	79.01%	79.87%

Figure 4.18: *Dependency of input ratio $height \times width$ to performance. Mix 1 with 0.2 dropout.*

were 12×50 for basic concatenation, 10×30 for the single additive and multiplicative models and 15×80 for the mixed models which consisted of 3 operations. This, however, did not mean that we had selected the most appropriate values. Having wider inputs meant introducing more parameters and the opposite meant more stride operations. Intuitively, wider inputs should introduce better performance. However, it turned out that without a sufficient height the size of the parameters did not matter as much. Tables 4.18 and 4.19 give a summary of the results we obtained showing that the best ratio was roughly one to three (20×60 and 25×64). However, the surrounding ratios with heights 15,100 and 16,50 do seem to perform well too. Regardless, we will continue to use one to three relationship unless otherwise stated.

Model	Train	Validation	Test
8×200	90.82%	78.01%	78.69%
10×160	87.21%	79.25%	79.09%
16×100	87.52%	79.55%	80.45%
25×64	**93.19%**	**79.82%**	**80.73%**
50×32	93.42%	79.51%	80.47%
100×16	93.17%	79.60%	79.96%

Figure 4.19: *Dependency of input ratio $height \times width$ to performance. Mix 1 with 0.2 dropout.*

4.2.4 Number of Epochs

An interesting observation is that the model stops learning anything roughly after the 10th epoch. Figure 4.20 shows this observation with all the considered models. Thus, there is in fact no need to run 25 epochs and 10 seem to be more than sufficient. We will from now on train our network with 10 epochs only. It is also interesting to note that these are visually split initially into three performance groups where sub, mix 1, 3, 4 and 6 fall under similar learning behaviour that achieves better than the rest.

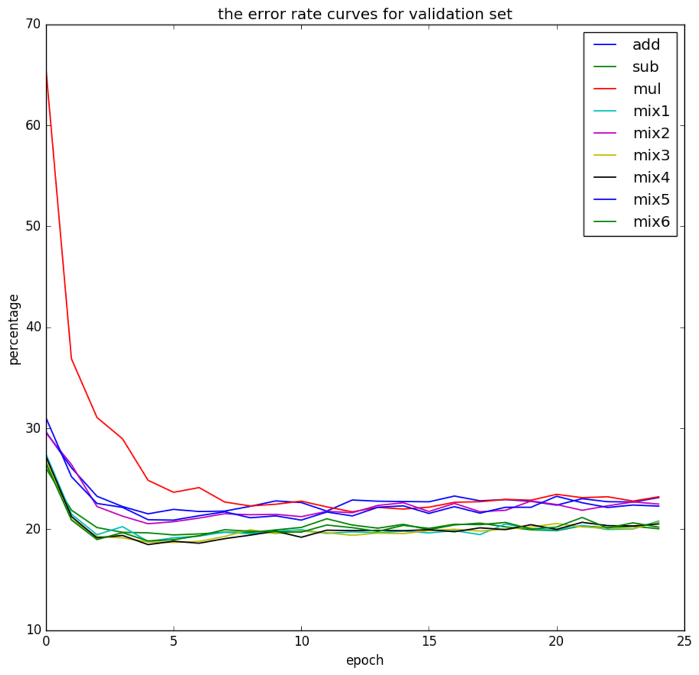


Figure 4.20: *Error rate curves for all models on validation set.*

4.3 Additional Hyper-parameter Optimisation

Finally, we wanted to further refine the hyper-parameters we had left, and namely dropout and learning rate decay. Table 4.22 shows the result from varying the learning rate decay within the established boundaries in the beginning of this chapter. The reported accuracy values are the average from running 5 jobs with every value. Table 4.21 shows the results we obtained for different dropouts and a learning rate set to 0.95. In fact, with the shown optimisation we achieve a state-of-the-art result of 81.26% outperforming the results reported in Rocktäschel et al. (2015) which was in

Dropout	Train	Validation	Test	Error
0.2	88.13%	79.82%	80.76%	8.26
0.17	88.02%	79.80%	80.13%	8.87
0.15	88.00%	79.92%	80.19%	8.81
0.14	87.87%	79.66%	80.17%	8.83
0.13	88.15%	**79.94**	**81.26%**	**7.74**
0.12	88.15%	79.96%	80.48%	8.52
0.1	88.19%	79.71%	80.55%	8.45
0.05	88.35%	79.69%	80.06%	8.94
0.0	88.25%	79.41%	80.45%	8.55

Figure 4.21: *State-of-the-art Result.*

fact our goal. The third column named 'Error' shows how far away our model is from the recorded human error on the SNLI data set, namely 89%. We are aware that the hyper-parameters in a CNN are very tightly correlated between each other and that these values might not be the same if we were to use different value for any other hyper-parameter.

Learning Rate Decay	Train	Validation	Test
0.95	**86.39%**	**79.27%**	**80.41%**
0.94	88.53%	80.02%	80.31%
0.93	86.66%	79.15%	80.00%
0.92	88.12%	79.53%	79.98%
0.91	88.08%	79.42%	80.00%
0.90	86.74%	79.40%	79.87%

Figure 4.22: *Improving learning rate decay.*

4.4 Conclusion

The results we achieved outperform the most basic RNN models used against the SNLI data set as defined by Bowman et al. (2015). Moreover, we achieve similar and even higher than the attention-based long short-term memory RNN proposed by Rocktäschel et al. (2015). To achieve this, we used a Siamese-like 3-CNN-wide architecture as shown in Figure 3.5. This new model led to an increase in the overall parameter size by introducing a new, larger input for the third CNN during training. We then used different mixtures between a number of mathematical operations. Regardless, we are still far from being close to the currently known human error of 89%.

Chapter 5

Discussion and Future Work

In this thesis we have shown that a lot simpler models can perform similar and in fact better than attention-based RNN solutions for the task of modelling entailment. We have achieved a state-of-the-art 81.26% performance as opposed to the 80.9% base model introduced in Rocktäschel et al. (2015). However, their solution, even though less than an year old, has only set the grounds for entailment modelling. They have been outperformed by new, more advanced models since then. Currently, the best known model achieves stunning 87.3% which is by far better than what we introduce here. Regardless, we would like to highlight that our work aimed to show that basic CNN implementations can in fact compete with and even outperform the massively adopted RNN LSTM structures for classifying entailment relationships with very small effort on feature engineering.

5.1 Further Experimentation

In order to further improve on our results, we would like to extend our hyper-parameter testing. More specifically, we would like to use an approach called Bayesian parameter optimisation Adams et al. (2012) to fine tune the values which will deem the best performance. This, uses a posterior distribution based on current results to pick the most appropriate hyper-parameter values. We did not use it for the purpose of this thesis as it can be very time-consuming and achieving results which exceed our results would have been unfeasible given the amount of people using the university's MSc cluster. Regardless, we think that this is a natural continuation of our work. Additionally, we would like to extend the experiments we did with the filter windows. Even though Zhang and Wallace (2015) show that the best pooling technique for sentence classification is in fact 1-max pooling, it might be useful if we did look into that too. We believe that there are a number of more options to be tested, some of which might result in better results than the ones received herein. Moreover, we wonder what would be the outcome if we treated the third CNN as if it models basic mathematical patterns and not as one used for an NLU task. Given that we saw how the input representation differs quite a bit from the actual syntactic representation observed initially, this might result in an interesting solution. Although, we think that this is highly unlikely, it is possible that a mixed model using RNNs and CNNs can turn out to be a better solutions.

5.2 Attention-Based CNNs

The work of Yin et al. (2015) uses a neural counterpart to alignment Bahdanau et al. (2014) which is more commonly used in LSTM-based architectures Rocktäschel et al. (2015), Cheng et al. (2016). The ABCNN model, however, does not use the SNLI corpus and does not report results obtained from using a basic CNN, thus it lacks a baseline for clear comparison of performance. Instead, it introduces a basic Siamese CNN. Moreover, the report describes the use of average-pooling and a TanH function for the non-linearity. We have, hereby shown that hyperbolic tangent is a rather difficult function to fit to a model and thus often results in low performance as reported in Chapter 4. In addition, Zhang and Wallace (2015) show that average pooling does not give the best results in general. All of these observations question the outcome of applying attention to CNNs. However, we are curious to see how and if similar approach can improve the performance of our model with the SNLI data set.

5.3 Parameter Reduction

Currently, our model has about 10M parameters (excluding the embeddings) which is very computationally expensive and in fact often prevented us from further increasing some of the hyper-parameters such as the batch size or the number of kernels used. Such restrictions, often affect not only the overall performance but the accuracy as well. Therefore, we would like to explore ways to reduce those in future. As a matter of fact Parikh et al. argue that current approaches focus too much on modelling the complex structures in a sentence and instead should model simple words alignment for RTE tasks.

An interesting study performed as part of a course work at Stanford's CS231n[1] suggests that the size of the parameters that are fed into the final fully-connected layer can be significantly reduced through low-rank approximation[2]. However, nothing concrete was proved in the student's study. Regardless, it will be interesting to consider ways of approximating the weights and hence significantly reduce the size of the parameters. In fact, we agree with Parikh et al. (2016) that the sparser a model, the better it will perform. We base this belief on the "bet on the sparsity" concept as discussed in Tibshirani (2014). Moreover, in support of this claim we reference some of the work of Hinton who proposes a neural network which reconstructs high-dimensional input vectors Hinton and Salakhutdinov (2006). Although their research is focused on feed-forward neural networks it can be applied to CNNs too. However, this would mean building and training a separate network which will probably not be feasible. In addition, we could use something simpler like PCA and ICA to cherry-pick only the most important vectors from the input feature space of the last fully connected layer. In fact, neural network factorisation is an actively researched field and there are a lot of opportunities for directing our future work that way. Regardless, all of these techniques will challenge the core concept of CNNs which assumes fixed architectures before learning. Thus, such an approach will challenge our architecture as well.

5.4 Domain Adaptation

Another interesting direction is that of Domain Adaptation. This is often confused with transfer learning. However, there is a subtle difference. Domain Adaptation tries to solve a given task in such a way that the solution can be applied to different priors as opposed to transfer learning which is interested in enabling the use of the same prior (data set) for different likelihoods (features). Regardless, given the little feature engineering one needs for our model, we believe that domain adaptation is indeed a feasible path. A big burden, however, would be to split the data in two logical sets (premises and hypothesis, for example). We think our model has the potential to contribute to areas like activity recognition. A current problem in that field, for example, is the lack of highly dimensional data set to cope with the complexity of the tasks of interest. However, currently existing approaches are quite good in recognising activities from simple signal processes Bhattacharya and Lane (2016). However, combined with

[1]http://cs231n.stanford.edu/reports/neckar.pdf

[2]http://nlp.stanford.edu/IR-book/html/htmledition/low-rank-approximations-1.html

depth images, for example, and fed into our model can in fact lead to state-of-the-art contributions to that field as well.

5.5 Conclusion

Overall, we believe we have shown the potential of using CNNs for modelling entailment relations through successfully achieving our goal and namely outperforming the work introduced in Rocktäschel et al. (2015) and Bowman et al. (2015). The little effort required for feature engineering and the relatively small error introduced in this thesis highlights an interesting perspective for using CNNs to achieve competitive results to current research. Moreover, the structure of our implementation can in fact enable the adaptation of other, enhanced models for any future work.

Appendix A

Appendix

A.1 Linear Regression

A.1.1 Pseudo-inverse

The probability density function (P.D.F.) of Y_i, given $Y \sim N(W^T X, \sigma)$, (i.e. Y is drawn from a normal distribution $N(.)$ with mean$W^T X$ and variance σ) is:

$$p(Y|X,w,\sigma) = \frac{1}{\sqrt{2\pi\sigma}} \odot exp(-\frac{1}{2\sigma^2} \odot (Y - w_1 X_{i1} - \cdots - w_n X_{in})^2) \tag{A.1}$$

Thus, assuming that data is drawn independently, the likelihood function is:

$$\prod_{i=1}^{n} L(Y_i) = (\frac{1}{\sqrt{2\pi\sigma}})^n \odot exp(-\frac{1}{2\sigma^2}(|Y - XW|)^2) \tag{A.2}$$

To maximise, we need to minimise $|Y - XW|^2$. Therefore, rewriting using scalar product as suggested in Panchenko (2006):

$$|Y - XW|^2 = (Y - \sum_{i=1}^{n} w_i X_i Y - \sum_{i=1}^{n} w_i X_i) = \tag{A.3}$$

$$(Y,Y) - 2\sum_{i=1}^{n} w_i (Y, X_i) + \sum_{i=1}^{n} \sum_{j=1}^{n} w_j w_i (X_j, X_i) \tag{A.4}$$

Then, setting the derivatives in each w_i equal zero follows:

$$-2(Y, X_i) + 2\sum_{j=1}^{n} w_j (X_i, X_j) = 0, \ therefore \tag{A.5}$$

$$(Y, X_i) = \sum_{j=1}^{n} w_j (X_i, X_j) \tag{A.6}$$

Which in matrix notation can be written as:

$$X^T Y = X^T X W \tag{A.7}$$

Matrix $X^T X$ is an nxn matrix. Moreover, it is invertible, given its rank and therefore Equation 2.2 holds. This proof was taken from my own coursework solution on MLPR[1]. This makes use of the so called *pseudoinverse* function Rao and Mitra (1971), Golub and Van Loan (1996) and namely resulting in:

$$W = (X^T X)^{-1} X^T Y \tag{A.8}$$

In order to analytically derive the above we will assume that Y is an $p \times 1$ vector and w is an nx1 (in our case $n = 3$) and X is thus $p \times n$ matrix. Denoting $X = (X_1, ..., X_n)$ and assuming the columns are linearly independent and the rank of the matrix is equal to n and that $n < p$.

A.1.2 Bias

If we define the logarithm of the probability as shown in Bishop (2007):

$$lnp(Y|X,w,\sigma) = \frac{N}{2} ln\sigma - \frac{N}{2} ln(2\pi) - \sigma \odot E_D(W) \tag{A.9}$$

Where we define an error function as:

$$E_D(W) = \frac{1}{2} \sum_{m=1}^{M} (y_m - w_0 - \sum_{i=1}^{N-1} w_i X_i))^2 \tag{A.10}$$

If we take the derivative with respect to w_0, we would get:

$$w_0 = \hat{Y} - \sum_{i=1}^{N-1} w_i \odot X_i \tag{A.11}$$

$$\hat{Y} = \frac{1}{M} \sum_{m=1}^{M} y_m \qquad X_i = \frac{1}{M} \sum_{m=1}^{M} x_m \tag{A.12}$$

A.1.3 Deriving ∇E_D

Deriving the derivative (∇E_D) for the error function is also straightforward. If we defined the $w^T \odot \phi(x)$ as a function, we would Consider the following Bernoulli scenario:

$$p(y^n|\phi(x)^n) = \sigma(w^T \phi(x) + bias) \tag{A.13}$$

[1]Derivation first done on MLPR Assignment

$$\Pi_{n=1}^{N} p(y^n|\phi(x)^n) = \Pi_{n=1}^{N} p(y^n = 1|\phi(x)^n)^{y^n}(1 - p(y^n = 1|\phi(x)^n)^{1-y^n} \quad (A.14)$$

We can thus define the log likelihood as:

$$\sum_{n=1}^{N} y^n \odot log(p(y = 1|\phi(x)^n)) + (1 - y^n) \odot log(1 - p(y^n = 1|\phi(x)^n)) \quad (A.15)$$

Defining $a = w^T\phi(x) + b$:

$$\sum_{n=1}^{N} y^n \odot log(\sigma(a)) + (1 - y^n) \odot log(1 - \sigma(a)) \quad (A.16)$$

$$\nabla wL = \sum_{n=1}^{N} y^n \frac{1}{\sigma(w^T\phi(x) + b)} \sigma`(w^T\phi(x) + b) + (1 - y^n) \frac{1}{\sigma(w^T\phi(x) + b)} (-\sigma`(w^T\phi(x) + b)) \quad (A.17)$$

$$\nabla wL = \sum_{n=1}^{N} \frac{y^n}{\sigma(a)} \sigma(a)(1 - \sigma)\phi(x) - \frac{(1 - y^n)}{1 - \sigma(a)} \sigma(a)(1 - \sigma(a))\phi(x) \quad (A.18)$$

Therefore,

$$\sum_{n=1}^{N} (y^n - y^n\sigma(a) - \sigma(a) + y^n\sigma(a))\phi(x) = \sum_{n=1}^{N} (y^n - \sigma(a))\phi(x) \quad (A.19)$$

A.2 Convolutional Neural Networks

A.2.1 Circular Convolution

The circular convolution model is not as straightforward to understand. Consider the first two rows from Table 2.10.

$$x = [1, 7, 3, 11, 5] \quad y = [2, 9, 5, 5, 1] \quad (A.20)$$

Then, if we revert vector y:

$$x = [1, 7, 3, 11, 5] \quad y = [1, 5, 5, 9, 2] \quad (A.21)$$

Then, we need to multiply the two vectors:

$$x \times y = 1 + 35 + 15 + 99 + 10 = 160 \quad (A.22)$$

Next step is to shift all elements in y to the left by one and situating the foremost ones to the end:

$$y = [5, 5, 9, 2, 1] \quad (A.23)$$

$$x \times y = 5 + 35 + 27 + 22 + 5 = 94 \tag{A.24}$$

We need to repeat that until we rotate through the entire vector y and get back to the initial state from Equation A.21.

A.2.2 Backpropagation

This work is based on Bengio and Courville (2016). Assume that we are presented with a kernel stack K, a sentence S and stride $stride$. Assuming that we have already computed a convolution $Y = c(K, S, stride)$, referred to as forward propagation, we would like to compute a backprop too, given some cost function $J(S, K)$. Therefore, we can write:

$$G_{i,j,k} = \frac{\partial}{\partial Y_{i,j,k}} J(S, K) \tag{A.25}$$

This, essentially equals:

$$\sum_{m,n} G_{i,m,n} S_{j,(m-1)\times stride+k,(n-1)\times stride+l} \tag{A.26}$$

If, however, the layer is not the first one, then we will need to compute the gradient w.r.t. the sentece S. This is achieved by:

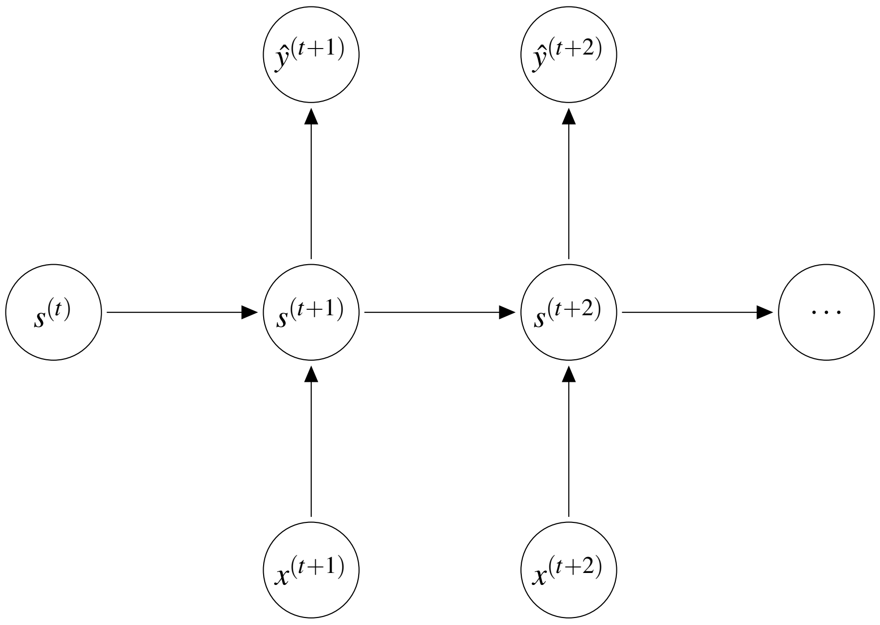
$$h(K, G, stride)_{i,j,k} = \frac{\partial}{\partial S_{i,j,k}} J(S, K) = \sum_{j=(l-1)\times stride+m} \; \sum_{k=(n-1)\times stride+p} \; \sum_{q} K_{q,i,m,p} G_{q,l,n} \tag{A.27}$$

A.3 Recurrent Neural Networks

A.3.1 Basic Model

Recurrent Neural Networks (RNN) are an alternative way of tackling complicated relationships between words and sentences. We will not intend to use them in this thesis. However, the majority of work against which we will compare ours makes use of them. Thus, we think it is important to simply highlight the concept. For a more detailed definition of the model we refer the reader to Mikolov et al. (2010), Mikolov et al. (2011) and Hochreiter and Schmidhuber (1997).

RNNs are known for being very good at modelling the sequential structure of a sentence. RNNs have only one hidden layer ($s^{(t)}$) that is being copied over time Keller (2016).



A.3.2 Backpropagation Through Time and LSTMs

Additionally, we can consider the concept of going back in time. It allows us to capture longer history information. This essentially means that we can have a better understanding of the general meaning of a given sequence of words. In other words ”the network ties the recurrent weights with each other guaranteeing a more concrete and direct adaptation of those weights in an observed sequence of words”[2]. Regardless, an important difference is the property of unfolding that BPTT enables us to account for the influence of n previously seen, 'older' words in a sequence. However, this leads to issues such as the vanishing/exploding gradients which can be solved with LSTMs (See Figure A.1[3]).

[2]This analysis is taken from Todor Davchev's work on Assignment 2 for NLU
[3]http://deeplearning4j.org/lstm.html

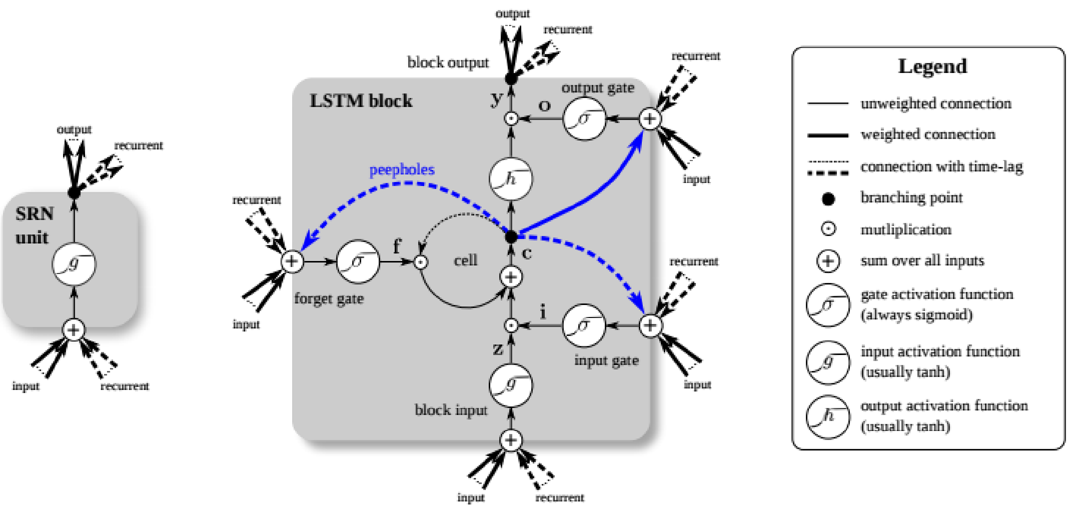


Figure A.1: *An LSTM architecture used for RNNs.*

Bibliography

Adams, R. P., Snoek, J., and Larochelle, H. (2012). Practical bayesian optimization of machine learning algorithms.

Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Bengio, I. G. Y. and Courville, A. (2016). Deep learning. Book in preparation for MIT Press.

Bengio, Y., Schwenk, H., Senécal, J.-S., Morin, F., and Gauvain, J.-L. (2006). Neural probabilistic language models. In *Innovations in Machine Learning*, pages 137–186. Springer.

Bhattacharya, S. and Lane, N. D. (2016). From smart to deep: Robust activity recognition on smartwatches using deep learning. In *2016 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)*, pages 1–6. IEEE.

Bishop, C. (2007). Pattern recognition and machine learning (information science and statistics), 1st edn. 2006. corr. 2nd printing edn.

Bobrow, D. G. (1964). Natural language input for a computer problem solving system.

Bowman, S. R., Angeli, G., Potts, C., and Manning, C. D. (2015). A large annotated corpus for learning natural language inference. *arXiv preprint arXiv:1508.05326*.

Britz, D. (2015). Understanding convolutional neural networks for nlp.

Bromley, J., Bentz, J. W., Bottou, L., Guyon, I., LeCun, Y., Moore, C., Säckinger, E., and Shah, R. (1993). Signature verification using a siamese time delay neural network. *International Journal of Pattern Recognition and Artificial Intelligence*, 7(04):669–688.

Chen, D., Bolton, J., and Manning, C. D. (2016). A thorough examination of the cnn/daily mail reading comprehension task. In *Association for Computational Linguistics (ACL)*.

Chen, D. and Manning, C. D. (2014). A fast and accurate dependency parser using neural networks. In *EMNLP*, pages 740–750.

Cheng, J., Dong, L., and Lapata, M. (2016). Long short-term memory-networks for machine reading. *arXiv preprint arXiv:1601.06733*.

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

Dong, L., Wei, F., Zhou, M., and Xu, K. (2015). Question answering over freebase with multi-column convolutional neural networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing*, volume 1, pages 260–269.

Dyer, C., Ballesteros, M., Ling, W., Matthews, A., and Smith, N. A. (2015). Transition-based dependency parsing with stack long short-term memory. *arXiv preprint arXiv:1505.08075*.

Firth, J. (1957). A synopsis of linguistic theory 1930-1955 in studies in linguistic analysis. philological society.

Goldberg, Y. (2015). A primer on neural network models for natural language processing. *arXiv preprint arXiv:1510.00726*.

Golub, G. H. and Van Loan, C. F. (1996). Matrix computations. 1996. *Johns Hopkins University, Press, Baltimore, MD, USA*, pages 374–426.

Harris, Z. (1954). Distributional structure. word 10: 146-162. reprinted in j. fodor and j. katz. *The structure of language: Readings in the philosophy of language*, pages 775–794.

Hinton, G. E. and Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507.

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

Johnson, M. (2009). How the statistical revolution changes (computational) linguistics. In *Proceedings of the EACL 2009 Workshop on the Interaction between Linguistics and Computational Linguistics: Virtuous, Vicious or Vacuous?*, pages 3–11. Association for Computational Linguistics.

Johnson, R. and Zhang, T. (2015). Semi-supervised convolutional neural networks for text categorization via region embedding. In *Advances in Neural Information Processing Systems*, pages 919–927.

Jurafsky, D. and Martin, J. H. (2014). *Speech and language processing*. Pearson.

Keller, F. (2016). Recurrent neural networks.

Kim, Y. (2014). Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*.

Koch, G. (2015). *Siamese neural networks for one-shot image recognition*. PhD thesis, University of Toronto.

Lapata, M. (2016). Semantic composition.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551.

Maas, A. L., Hannun, A. Y., and Ng, A. Y. (2013). Rectifier nonlinearities improve neural network acoustic models. In *Proc. ICML*, volume 30.

Martin, J. (1973). Design of man-computer dialogues.

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

Mikolov, T. and Dean, J. (2013). Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*.

Mikolov, T., Karafiát, M., Burget, L., Cernockỳ, J., and Khudanpur, S. (2010). Recurrent neural network based language model. In *Interspeech*, volume 2, page 3.

Mikolov, T., Kombrink, S., Burget, L., Černockỳ, J., and Khudanpur, S. (2011). Extensions of recurrent neural network language model. In *2011 IEEE International*

Conference on Acoustics, Speech and Signal Processing (ICASSP), pages 5528–5531. IEEE.

Mitchell, J. and Lapata, M. (2010). Composition in distributional models of semantics. *Cognitive science*, 34(8):1388–1429.

Munkhdalai, T. and Yu, H. (2016). Neural tree indexers for text understanding. *arXiv preprint arXiv:1607.04492*.

Murphy, K. P. (2012). *Machine learning: a probabilistic perspective*. MIT press.

Nielsen, M. A. (2015). Neural networks and deep learning. *URL: http://neuralnetworksanddeeplearning. com/.(visited: 01.11. 2014)*.

Panchenko, D. (2006). Multiple linear regression.

Pang, B. and Lee, L. (2005). Seeing stars: Exploiting class relationships for sentiment categorization with respect to rating scales. In *Proceedings of the 43rd annual meeting on association for computational linguistics*, pages 115–124. Association for Computational Linguistics.

Parikh, A. P., Täckström, O., Das, D., and Uszkoreit, J. (2016). A decomposable attention model for natural language inference. *arXiv preprint arXiv:1606.01933*.

Partee, B. (1995). Lexical semantics and compositionality. *An invitation to cognitive science: Language*, 1:311–360.

Pennington, J., Socher, R., and Manning, C. D. (2014). Glove: Global vectors for word representation. In *EMNLP*, volume 14, pages 1532–43.

Rao, C. R. and Mitra, S. K. (1971). *Generalized inverse of matrices and its applications*, volume 7. Wiley New York.

Reddy, S., Tackstrom, O., Collins, M., Kwiatkowski, T., Das, D., Steedman, M., and Lapata, M. (2016). Transforming dependency structures to logical forms for semantic parsing. *Transactions of the Association for Computational Linguistics*, 4:127–140.

Renals, S. (2015). Multi-layer networks.

Richards, M. (2015). Software architecture patterns.

Rocktäschel, T., Grefenstette, E., Hermann, K. M., Kočiskỳ, T., and Blunsom, P. (2015). Reasoning about entailment with neural attention. *arXiv preprint arXiv:1509.06664*.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1985). Learning internal representations by error propagation. Technical report, DTIC Document.

Searle, J. R. (1980). Minds, brains, and programs. *Behavioral and Brain Sciences*, 3:417–424.

Shapiro, S. C. (1992). *ENCYCLOPEDIA OF ARTIFICIAL INTELLIGENCE SECOND EDITION*. New Jersey: A Wiley Interscience Publication.

Simard, P. Y., Steinkraus, D., and Platt, J. C. (2003). Best practices for convolutional neural networks applied to visual document analysis. In *ICDAR*, volume 3, pages 958–962.

Tibshirani, R. J. (2014). In praise of sparsity and convexity. *Past, Present, and Future of Statistical Science (X. Lin, C. Genest, DL Banks, G. Molenberghs, DW Scott, and J.-L. Wang, Eds.). Chapman & Hall, London*, pages 497–505.

Weiss, D., Alberti, C., Collins, M., and Petrov, S. (2015). Structured training for neural network transition-based parsing. *arXiv preprint arXiv:1506.06158*.

Weizenbaum, J. (1966). Elizaa computer program for the study of natural language communication between man and machine. *Communications of the ACM*, 9(1):36–45.

Williams, C. (2015a). Classification.

Williams, C. (2015b). Data and models.

Winograd, T. (1972). Understanding natural language. *Cognitive psychology*, 3(1):1–191.

Yin, W., Schütze, H., Xiang, B., and Zhou, B. (2015). Abcnn: Attention-based convolutional neural network for modeling sentence pairs. *arXiv preprint arXiv:1512.05193*.

Zeiler, M. D. (2012). Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*.

Zhang, Y. and Wallace, B. (2015). A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification. *arXiv preprint arXiv:1510.03820*.