Unification of Lexical/Structural and Semantic Short Text Similarity Analyses

Seyithan Dag

MASTER THESIS

enrolled in
Master of Science in

Computer Science

in Rheinische Friedrich-Wilhelms-Universität Bonn

in August 2019

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Rheinische Friedrich-Wilhelms-Universität Bonn, August 31, 2019

Seyithan Dag

University of Bonn

Master thesis

Unification of Lexical/Structural and Semantic Short Text Similarity Analyses

Author:
Seyithan Dag

Supervisor:
Prof. Dr. Jens Lehmann

Second Supervisor:
Dr. Giulio Napolitano

Advisor:
Dr. Diego Esteves

Contents

List of Figures

List of Tables

Abstract

In this work, we present a novel method for incorporating structural similarity analysis to artificial neural networks. Our method involves treating words as relations and crafting relation matrices out of these relations. We introduce a novel structural string similarity metric which we teach to the neural network consisting of an encoder, a convolutional layer and a multilayer perceptron that is meant to execute the structural similarity analysis between two short text pieces. We turn the metric teaching into simple regression problem thanks to the finiteness induced by the relational encoding scheme we develop. We also propose a neural network architecture for semantic similarity analysis which is used as a gauging mechanism to judge the effectiveness of the inclusion of structural similarity analysis, as we later on unify both architectures. We present test results on semantic entailment recognition, paraphrase identification and semantic textual similarity detection. Through unification, we achieve up to 13% increase in test accuracy for semantic entailment recognition and semantic textual similarity detection.

Chapter 1

Introduction

What makes a piece of text similar to another? Are the frequencies of each of the letters appearing in a piece of text what makes that text structurally unique? Or, is the context that lies behind the only key to a piece of text to serve its purpose of communication? When given a simple thought, text is just an end result of certain permutations of letters, and at the end, what makes this text unique, or gives a meaning to it is what these permutations point to and how they are placed.

The compliment of what makes a text unique, by definition, is what makes it similar; and everything that makes it unique, or in other words, every measurement/-calculation made for the purpose of determining a text's uniqueness can technically also be used to determine if that text is similar to another. With this idea in mind, similarity comparison of two or multiple pieces of text could be carried on.

Determining the similarity of a text piece to another surely has many parameters to be looked on. After all, it is the meaning behind what makes a text ultimately useful as a tool of conduct, and because of that, the importance of the context cannot be underestimated. However, it is not always the case where the context to which a piece of text belongs is/must be taken into consideration. There are many algorithms that compute the similarity of two text pieces based solely on the lexical/structural features. The team of Maharjan et. al. [45] achieved the first place in SemEval 2017 Task-1 for semantic similarity using just alignments, sentence level embeddings and Gaussian mixture model output. In this respect, it could be seen that, as previously mentioned, the context might not always play a role in determining the similarity of text, therefore, it would rather be more practical to think of both the structure and the context as two hyper-parameters, involvement and usage of which could be decided on according to the current need. Another very familiar case where just the context of a text has the utter importance, that is, without the importance of the structure, is the named entity recognition process. Since difference in structure of two words is apparent, what makes Berlin and London similar is the context that lies behind, that is, they are the names of two cities where city is an abstract concept hooked onto the communicative purpose of these letter permutations by the context. This very context is commonly referred to as the semantics.

Semantic similarity of two text pieces is a heavily studied subject in the literature.

With their exponentially growing evolution and lately-obtained popularity, neural networks are currently the first choice for the determination of semantic similarity of two text pieces. With the help of the word embeddings [47], neural networks achieve up to 90% accuracy on certain text similarity tasks. It is often the case that while semantic similarity scoring is sought, lexical similarity is either discarded or obtained and added into the whole equation through hand-crafted text features. That is, to the best of our knowledge, there is yet a neural network model to fully combine lexical/structural and semantic features of text as the sole working entity.

With the light of the explanations above, in this paper, we propose a novel technique to incorporate lexical/structural similarity analysis to a neural network, inspired by the Braille encoding for the blind [7]. We turn words into relations, then to so called relation matrices. We introduce a novel structural similarity metric based on these relation matrices that is used to teach structural string distances to the neural network we propose for structural similarity analysis in a supervised fashion. We also propose a neural network architecture for semantic similarity analysis that is used mainly as a gauging mechanism to express the beneficial outcomes of including structural similarity while performing tasks meant for semantic similarity analysis of short text, as we in the end unify both architectures and produce test results for various sub-tasks of short text similarity analysis done by only the semantic network alone and also by the unified network. We use terms lexical and structural interchangeably throughout this paper.

The organization of chapters is as follows. In chapter 2, we scan through the related work in the literature, then in chapter 3 we explain the technologies used. After these, we explain our proposed model and the peripheral findings in chapter 4. In chapter 5, we present and discuss the results of our experiments, after which we conclude our work.

Chapter 2

Related Work

Thanks to the abundance of works, text similarity analysis gets its fair share in the literature. Due to the exponentially-rising popularity of neural networks, the recent work in the area revolves heavily around varieties of artificial neural networks. Since text matching comes with many sub-tasks ranging from paraphrase identification to semantic entailment recognition, the models in the literature are usually benchmarked for these different sub-tasks. Therefore, in this chapter, we also follow a mixed approach for conveying the literature. It is also extremely common to see word embeddings being utilized to an extend that almost every model in the literature uses word embeddings.

The first usage of convolutional neural networks for textual entities dates back to 2014 with Kim's work [38]. In this paper, Kim uses convolutional neural networks for sentence classification. With the expressive power of word embeddings, the author uses 1-dimensional convolution on word embeddings to produce pseudo n-word-grams of a given sentence. The model achieves then-state of the art results for sentiment analysis and question classification. Another study that makes use of convolutional neural networks is Hu et. al. [33]. Through letting convolutionally extracted n-word-gram features of two sentences, they were able to capture rich interaction based patterns in given to sentences so as to be utilized for text matching. They achieved then-state of the art results in sentence completion, response matching and paraphrase identification. Another early example for the usage of neural networks for text similarity is Socher et. al.'s paper [55]. They introduce a method using recursive autoencoders that follow a novel unfolding objective and learn feature vectors for phrases existing in syntactic trees. Their main intention for their model is paraphrase detection, however they also show the expressive power of mentioned feature vectors when it comes to word-wise and phrase-wise similarity of sentences. They also introduce a novel dynamic pooling layer that standardizes the outcome of the model so that length invariance among sentences is achieved. The frequency of the usage of convolutional architectures in text similarity analysis cannot be discarded, In their 2018 dated work, Chen et. al. [11] uses both local interactions of sentences and feature engineering followed by convolution and pooling operations for text matching. Just like aforementioned works, they also make use of n-word-grams,

which, in fact, shows the power of this automatically generated feature. Yet one more interesting study done by Dai et. al. [17] uses convolutional neural networks for generating n-grams of various lengths and then soft matches them in a unified embedding space. They utilize their model for ad-hoc search involved in query to document matching.

It is common in the reviews of literature to see the distinction being made whether a method is strong-interaction based or not. We will not be grouping our scan of literature according to this. However, when needed, this feature might be pointed out such as this another example of a strong interaction model in Wan et. al.'s paper [60] where they utilize long short-term memory networks to create their model called MV-LSTM. The model uses LSTMs to match two sentences with multiple positional sentence representations. Specifically, each positional sentence representation is a sentence representation at this position, generated by a bidirectional long short term memory (Bi-LSTM). After which, the aggregated representations are given to a multilayer perceptron to produce a final score. Of course, this work is not the first to introduce the usage of LSTMs. With the publishing of SNLI (Stanford Natural Language Inference) corpus, Bowman et. al. [6] used LSTMs to recognise textual entailment an achieved then-favourable results. Following this, many papers investigated the expressive power of LSTMs for text similarity analysis [43, 49, 51, 64], either via specialization of the LSTM architecture or interconnecting them in different ways such as parallel or stacked LSTMs. In their collective work [49], Prijatelj et. al. extensively investigate the power of combinations of different interconnections of LSTMs and multilayer perceptrons for the task of semantic textual similarity detection. Their work even uses 10 stacked LSTMs to showcase whether the amount of LSTMs have a direct effect on the performance. They, however, prove that this is not the case, as the most successful model in their experiments turn out to be two LSTMs stacked back-to-back that is used for embedding sentences followed by another two stacked LSTMs for similarity score production. Yao et. al's [64] work on utilization of LSTM-encoder architecture also shows the comprehensive power of LSTMs for the similarity analysis of unlabelled short text. They show that the architecture surpasses the performance of various short text similarity measurement algorithms. Another example for using deep neural networks for ad-hoc operations such as retrieval is Guo et. al.'s paper [26]. In this work, they employ feature engineering for creating histograms of two text pieces and use a multilayer perceptron network to teach the underlying properties of these histograms.

It is also very frequent to see hybrid architectures being implemented. That is, models that use both hand-made features as the results of feature engineering and also generate their own features. These features are usually familiar natural language tools such as POS (part of speech) tagging or NER (named entity recognition). Study done by Wu et. al. [62] benefits from prior knowledge of texts as the main engineered feature used in conjunction with a neural network by the model named KEHNN (knowledge enhanced hybrid neural network). They achieved then-state of the art results on answer and response selection. Another different hybrid approach taken by Duong et. al. [19] tries to leverage the advantages of the com-

bination of feature engineering and neural networks for paraphrase identification. In this work, they combine named entity recognition and word embeddings to map sentences to another embedding space. They also make use of the currently-very-popular attention mechanism. DeepHAN model presented by Song et. al. [57] also harnesses the success of word-level and sentence-level attention. They use GRUs (gated recurrent units) coupled with the attention mechanism to hierarchically extract meaningful matching patterns and rich contextual features of query-document pairs. They present the performance of their model via tests done on the WikiQA dataset [1].

Siamese networks [8], on the other hand, gained a lot of popularity in recent years. The usage of these type of networks started with Baldi and Chauvin's work on fingerprint comparison [3]. After which, the usage continued with the work done by Yih et. al. for short text comparison [65]. Of course, in between there were studies that also used siamese architecture, however they mainly dealt with the area of computer vision. The DSSM (deep semantic similarity model) [34] is also a siamese architecture that consists of two deep neural network models where one is specialised for handling the queries and the other for documents. In this paper, the authors turn the texts (document and query) into bags-of-character trigraphs. They train their model with clickthrough data, such that each sample contains a query, a positively associated document and a negatively associated document. In the literature, it is very common to see that models implementing the siamese architecture make use of triplet, contrastive or hinge losses as their minimization objective during training. Another paper that uses siamese architecture is by Shen et. al. [54]. In their paper, authors present a series of new latent semantic models based on a convolutional neural network (CNN) to learn low dimensional semantic vectors for search queries and web documents. By using the convolution-max pooling operation, authors extract local contextual information at the word-n-gram level. Then, salient local features in a word sequence are combined to form a global feature vector. Finally, the high-level semantic information of the word sequence is then further extracted to form a global vector representation. They evaluate their model on a web document ranking task using a large-scale, real-world dataset and achieve then-state of the art results.

Approaches to leveraging lexical features of text have also been around in the recent years. Brychcin and Svoboda [9] ranked first in SemEval 2016 Task-1: Semantic Short Text Similarity. In their model, they extract many lexical features of text such as lemma, POS and character n-grams. Not only this, but they also extract semantic features through both conventional machine learning techniques and deep neural network architectures. Mitra et. al. [48] hypothesize that matching with distributed representations complements matching with traditional local representations, and that a combination of the two is favourable. In their paper, they propose a novel document ranking model composed of two separate deep neural networks, one that matches the query and the document using a local representation, and another that matches the query and the document using learned distributed representations.

[1] https://www.microsoft.com/en-us/download/details.aspx?id=52419

The two networks are then jointly trained as part of a single neural network. They evaluate their method with web page ranking test and achieve then-state of the art results.

Lastly, distance metric leaning can be shortly explored. Gouk. et. al. [23] proposes an algorithm where through an embedder neural network architecture, Euclidean distances are transformed to a new inner product space in which the proximity of vectors resemble Jaccard similarity in the original space. They show that the learned distance improves the results on instance-based multi-label classification.

Chapter 3

Technical Background

This chapter explains the fundamental technologies used as building blocks of the model that is going to be presented in chapter 4. There are two main parts: first part dwells on selected string similarity/distance measures, and the second part provides fundamental but not so deep explanation of neural networks in general, and the specialized versions thereof.

3.1 String Similarity/Distance Measures

In this section, conventional string similarity/distance measures will be introduced. These methods have actively been used for pattern matching, clustering, classification and many more [13, 14]. In this paper, these methods are used either as baseline for performance evaluation, or as building blocks for further similarity/distance measures that are composed of several methods.

Jaccard Similarity
Jaccard similarity is a statistic used for measuring the similarity and diversity of two finite sets [35]. The value of Jaccard similarity is defined through dividing the size of the intersection by the size of the union of these sets. Formally, for two finite sets A and B:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|} \tag{3.1}$$

Strings can be thought of as ordered sets of characters for the application of this metric.

Levenshtein Distance
Levenshtein distance, which is a metric of the class of edit distances, is a string metric for measuring the difference between two sequences [42]. Informally, the Levenshtein distance between two strings is the minimum number of single-character edits required to change one string into the other. These edits can be insertions, substitutions or deletions. The Levenshtein distance between the first i characters

of string a and the first j characters of string b that are of lengths $|a|$ and $|b|$ respectively is given by the following formula:

$$lev_{a,b}(i,j) = \begin{cases} max(i,j) & \text{if } min(i,j) = 0 \\ min \begin{cases} lev_{a,b}(i,j-1)+1 \\ lev_{a,b}(i-1,j)+1 \\ lev_{a,b}(i-1,j-1)+\mathbf{I}_{a_i \neq b_j} \end{cases} & \text{otherwise} \end{cases} \tag{3.2}$$

where $\mathbf{I}_{a_i \neq b_j}$ is the inequality indicator function that outputs 1 if $a_i \neq b_j$, and 0 if $a_i = b_j$.

Jaro-Winkler Similarity

Jaro–Winkler similarity is also a metric of the class of edit distances, measuring the edit distance between two strings (that is, $1 - similarity$). It is the modified version of the Jaro distance [36] metric and uses a prefix scale p which gives more favourable ratings to strings that match from the beginning for a set prefix length l [61]. The formula for this similarity metric for two strings a and b is as follows:

$$sim_{jw} = sim_j + lp(1 - sim_j) \tag{3.3}$$

where sim_{jw} is the Jaro-Winkler similarity, sim_j is the Jaro similarity, l is the length of common prefix at the start of the string up to a maximum of four characters and where p is the constant scaling factor for $0 < p \leq 0.25$. The Jaro similarity sim_j is, then, given by

$$sim_j = \begin{cases} 0 & \text{if } m = 0 \\ \frac{1}{3}\left(\frac{m}{|a|} + \frac{m}{|b|} + \frac{m-t}{m}\right) & \text{otherwise} \end{cases} \tag{3.4}$$

where $|a|$ and $|b|$ are the lengths of strings a and b respectively. Here, m denotes the number of matching characters. The value of the t is the half number of transpositions. That is, each character of a is compared with all its matching characters in b and then the number of matching characters that are of different orders (i.e. having different positions than their counterparts) is halved, which gives the number of transpositions. Lastly, the Jaro-Winkler distance d_{jw} is then

$$d_{jw} = 1 - sim_{jw} \tag{3.5}$$

Hamming Distance

Hamming distance between two strings of equal length is the number of positions at which the corresponding symbols are different [27]. Hamming distance is also a metric of the class of edit distances that is mainly used for character and bit strings. That is, as previously stated, it measures the minimum number of substitutions required to change one string into the other, or, additionally, the minimum number of errors that could have transformed one string into the other. For two strings a and b, the hamming distance d_{ham} is calculated as follows:

$$d_{ham}(a,b) = \sum_{n=0}^{k-1} \left[y_{a,k} \neq y_{b,k} \right] \tag{3.6}$$

where k is the index of the respective character reading y out of the total number of characters n. The Hamming distance itself gives the number of mismatches between the characters paired by k. Note that Hamming distance assumes that both strings are of equal length, which is hardly the case for real world applications. This complication however can be got over via adding empty spaces to the string that has fewer characters.

3.2 Artificial Neural Networks

An artificial neuron is a biologically inspired computational unit, whose structure and mechanism of action resembles that of an actual neuron in a brain. The first proposal of an artificial neuron-alike structure was made by [46], which then took the form commonly known as **perceptron** [52]. A perceptron can be thought of as the simplest form of a neural network that has only one neuron and binary output. A neuron, however, is a generalization of the idea of a perceptron. Hence, it would not be too inappropriate to think of a perceptron unit as a primitive single neuron. Therefore, henceforth, the term (single) neuron will mean a generalized version of a perceptron and will not be used interchangeably.

A single neuron basically has four components: inputs, weights, bias and an activation function. A simple graphical depiction of a neuron can be seen in figure 3.1. The computation carried on by a neuron unit with N number of inputs is executed by the following equation:

$$y = f(\sum_{i=1}^{N} w_i x_i + b) = f(\boldsymbol{W}\boldsymbol{X} + b) \tag{3.7}$$

where $\boldsymbol{X} \in \mathbb{R}^N$ is row vector of the inputs, $\boldsymbol{W} \in \mathbb{R}^N$ is the column vector of the parameters, i.e. weights, $b \in \mathbb{R}$ is the bias term, $f\colon \mathbb{R} \to \mathbb{R}$ is the non-linear activation function and $y \in \mathbb{R}$ is the output. As can be seen from the equation above, a neuron basically applies some affine linear transformation to the inputs, which are then mapped to some scalar value by the activation function. This value is, then, used for the purpose of classification or regression, depending on the task. There exists variety of activation functions, Sigmoid, tanh, ReLU etc., to name a few. Sigmoid and tanh activations are characterized by the following equations:

$$sigmoid(x) = \frac{1}{1+e^{-x}} \qquad tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}, \tag{3.8}$$

and softmax is defined as

$$p(C_k|\mathbf{x}) = \frac{e^{\,a_k(\mathbf{x})}}{\sum_{j=1}^{K} e^{\,a_j(\mathbf{x})}} \tag{3.9}$$

, where K is the number of probabilities in the given input vector $\mathbf{x}$ of probabilities. The role of the bias term in a neuron can simply be though of as a necessary shifter of the activation function. For simplicity and to stay within the boundaries

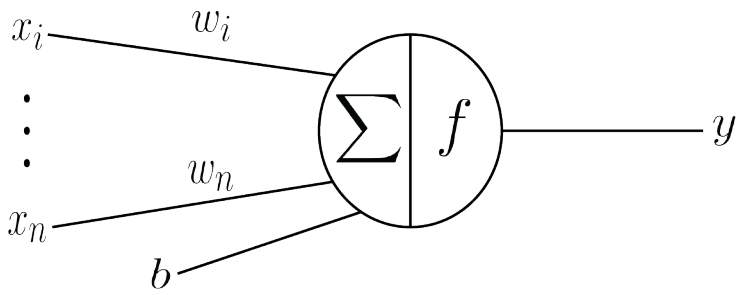

Figure 3.1: A basic diagram of a single neuron unit.

of this work, these three activation functions will be on focus, as they are the only activation functions that are used throughout the proposed model in chapter 4. However, though, ReLUs are currently the default recommended first-choice activation functions [22]. A rectified linear unit is characterized by

$$ReLU(x) = max\{0, x\} = \begin{cases} 0 & x < 0 \\ x & \text{otherwise} \end{cases} \tag{3.10}$$

As ReLUs are almost linear functions, they allow interconnected neural architectures, i.e. networks of neurons generalise well using the training data, and therefore perform well on the actual data [22].

Through interconnecting many neurons, one naturally obtains a so called network of these neurons. As the name of this chapter suggests, these structures are called **artificial neural networks**. When many neurons are put together, they are organized in a layered fashion. This topology allows the networks to successfully shatter the input space such that even linearly inseparable spaces become separable through back to back addition of neurons. Although one still possibly *could* achieve this with just one layer of neurons via careful selection of the activation function (that is of an unconventional nature)[4], this will not be dwelt on. A "layer" of neurons ultimately means that all of the neurons in the same set of formation possess the same type of activation function and inputs; these inputs might be the initial inputs fed to the network or can be coming from the previous layer of the network, in which case this very layer is named as **hidden**. Having given the necessary introduction to the concept of a single neuron and the networks thereof, it is time to move on to the next section where types of artificial neural networks, their capabilities and how they learn on data are going to be explored.

3.2.1 Feedforward Neural Networks

Feedforward neural networks, or multilayer perceptrons (MLPs), are the mostly used and popular form of artificial neural networks. The main purpose of a feedforward neural network is to approximate some function $g(x)$, for which a mapping $y = f(x;\theta)$ is induced and the values of parameters θ are learned such that $f(x;\theta) \approx g(x)$. After all, artificial neural networks are universal function approximators [16, 18, 32].

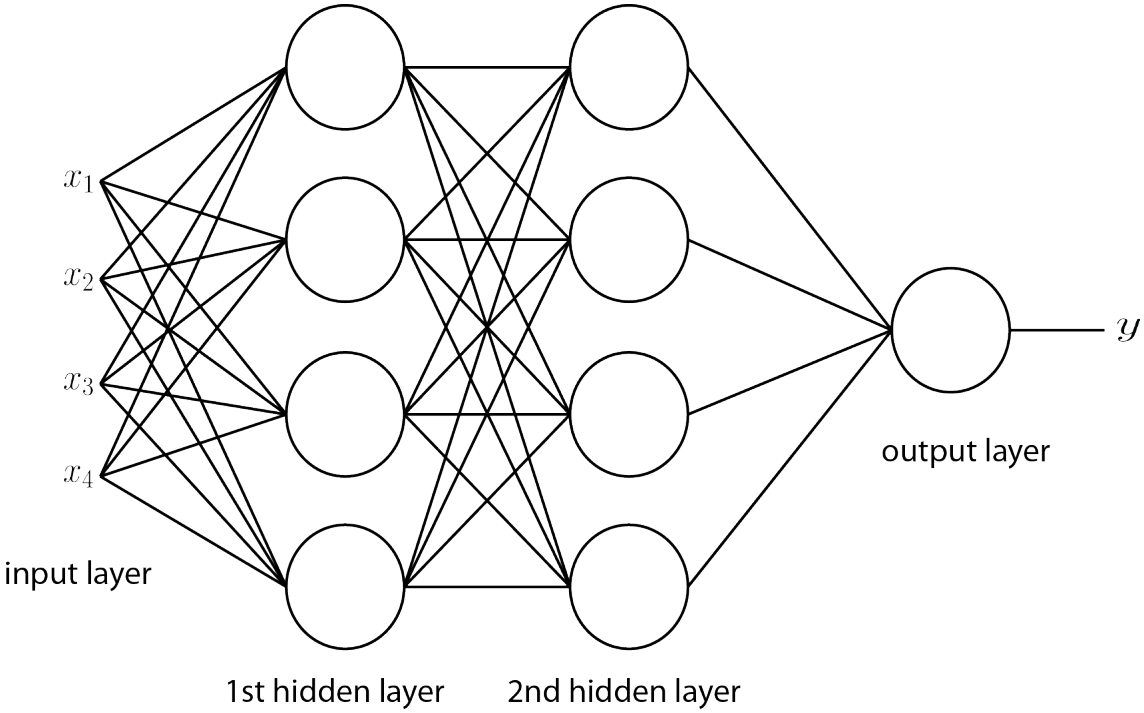

Figure 3.2: Feedforward neural network with four inputs and two hidden layers with four neurons in each layer. Bias terms are omitted for simplicity.

As stated in the previous section, feedforward neural networks are constructed by stacking layers of neurons back to back of each other. Following this convention, the first layer of a feedforward neural network is called the input layer, the last layer is called the output layer, and all the layers that lie in between are called the hidden layers. Figure 3.2 shows an example diagram of a feedforward neural network. Note that the output layer does not have to bear only one neuron, it could be multiple according to the need, as could the number of hidden layers. In fact, the overall number of hidden layers are called the **depth** of the network. Feedforward neural networks are called feedforward, because the information flows from the inputs to the outputs, without the network having any feedback connections of any sort. If this is the case, i.e. if the network has feedback connections where at any stage of computation any one of or multiple immediate outputs are fed back to the network, then this network is called a **recurrent neural network**, which will be explained in section 3.2.2.

If the network in figure 3.2 is taken as an example, the transformation induced by this network is formulated as

$$y = (f^{(2)}(f^{(1)}(\boldsymbol{X}\boldsymbol{W}^{(\mathbf{1})} + b^{(1)})\boldsymbol{W}^{(\mathbf{2})} + b^{(2)}))\boldsymbol{W}^{(\mathbf{3})} \tag{3.11}$$

where $f^{(l)}$ is the activation function in the lth layer, $\boldsymbol{W}^{(\boldsymbol{l})}$ is the weights vector and $b^{(l)}$ are the biases for layers $l = \{1, 2, 3\}$ respectively. As previously stated, the goal of a feedforward neural network is to approximate some function $g(x)$ for which

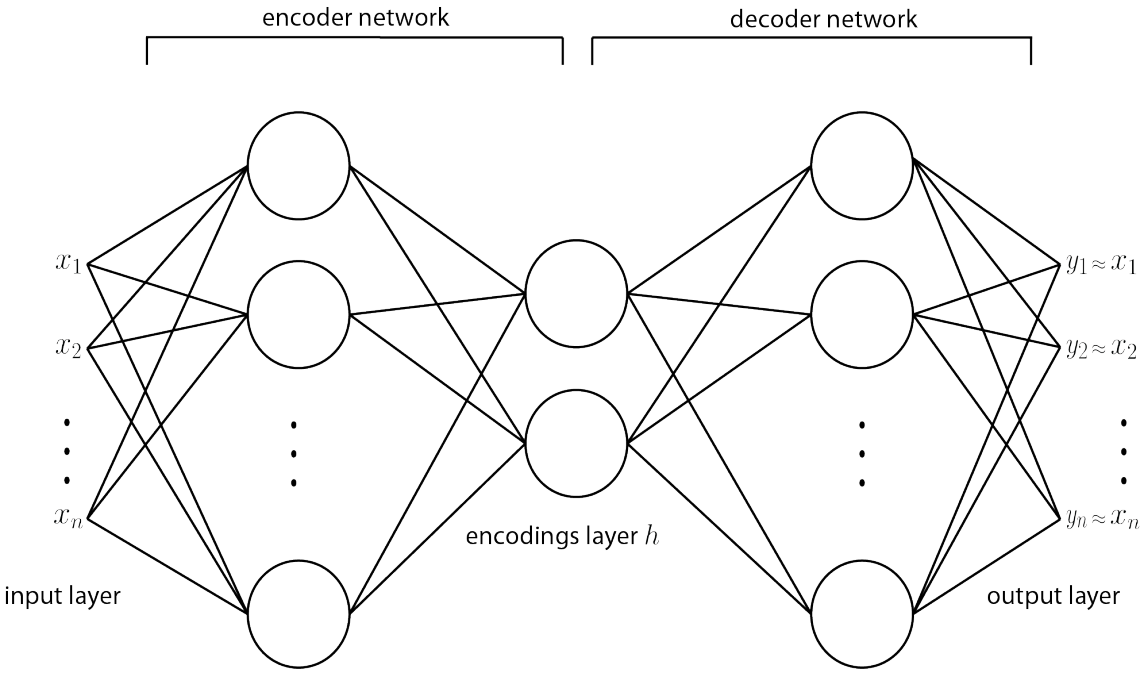

Figure 3.3: An autoencoder network with $d = 2$ as the encodings dimension.

the parameters of the network are adjusted so that the resulting outputs match the data, i.e. $f(x_i) = y_i \approx g(x_i)$ for each x_i presented during training as close as possible. It is the objective of the learning algorithm to decide how each layer of the network produce values for the final or intermediate outputs, so that this goal is achieved. The training of a feedforward neural network can be of supervised or unsupervised fashion. The details of the learning process will be explored in section 3.2.3. Before digging more into the learning, two specialized versions of feedforward neural networks will be explained. These are namely autoencoders and convolutional neural neural networks.

3.2.1.1 Autoencoders

Autoencoders are basically feedforward neural networks that are trained in an unsupervised manner for the purpose of copying the input to its output. The earliest definition and novel formalism of an autoencoder (although with different terminology) dates back to 1986 [53]. The internal structure of an autoencoder carries a hidden layer $\boldsymbol{h}$ that ultimately describes an embedding (encoding), whose dimension is ideally smaller than the original input data. Such autoencoders where the dimension of the encoding is less than the input are called **undercomplete**. The traditional usage of autoencoders was for the purpose of dimensionality reduction or feature learning, however latest advancements have also put autoencoders to the focus of generative modelling [22]. A graphical representation of an example autoencoder network can be seen on figure 3.3. An autoencoder consists of two sub-networks: an encoder network that maps the input to the latent encoding on layer $\boldsymbol{h}$, and a

decoder network that tries to reconstruct the original input based on the learned latent encoded representation. There are also many specialised types of autoencoders, such as denoising autoencoders, sparse autoencoders, convolutional autoencoders and so on. However, again, for the purposes of staying within the boundaries of this work, these will not be explored, as the proposed model in chapter 4 (albeit having a convolutional layer) only makes use of this basic undercomplete autoencoder paradigm.

3.2.1.2 Convolutional Neural Networks

Convolutional neural networks (CNNs) are specialized type of feedforward neural networks that take advantage of patterns in the data that is organised in a grid-like topology [22, 41]. Since images are of this fashion, convolutional neural networks are especially popular choice in the vision community, especially after their success at recognising and classifying objects in an image invariant to their position was shown [40]. The name convolutional comes from the fact that at least one layer in this special kind of network employs the mathematical operation known as **convolution** instead of a general matrix multiplication. Mathematically, convolution operation between two functions is defined as

$$s(t) = (x * w)(t) = \int x(a)w(t-a)da \tag{3.12}$$

which produces function $s(t)$ that describes how the shape of function $x(t)$ is modified by $w(t)$. The asterisk is used to denote the convolution operation. However, since the interest of data in the field of neural networks is of discrete nature, it is necessary to move on to the discrete version of convolution operation, which is

$$s(t) = (x * w)(t) = \sum x(a)w(t-a). \tag{3.13}$$

The first argument in this equation, i.e. the function $x(t)$, is the **input**, and the second argument is the **kernel**. The output produced by this operation is called a **feature map**. In the terminology of neural networks the kernel is the same set of parameters reused by the neurons, and all neurons that rely on the same kernel form a **convolutional filter**. The set of outputs produced by the same convolutional filter is then a feature map. The type of data that the convolutional neural networks deal with with are usually multidimensional arrays, which are commonly referred to as **tensors**. For example, a colour image that has 28×28 pixel is a tensor of shape $(28, 28, 3)$, where the last dimension belongs to the 3 colour channels of the image, and which, ultimately, is the depth of this tensor. Hence the dimensions are often referred to as height, width and depth respectively. The mathematical way of convolution involves flipping of the kernel so as to be able to make use of the commutative property of the convolution operation. In neural networks, however, this is usually not the case, and actually the operation used is a close relative of convolution called cross-correlation [22]. That is why, in the rest of this paper cross-correlation will be referred to as convolution. Let T be the function depicting a two

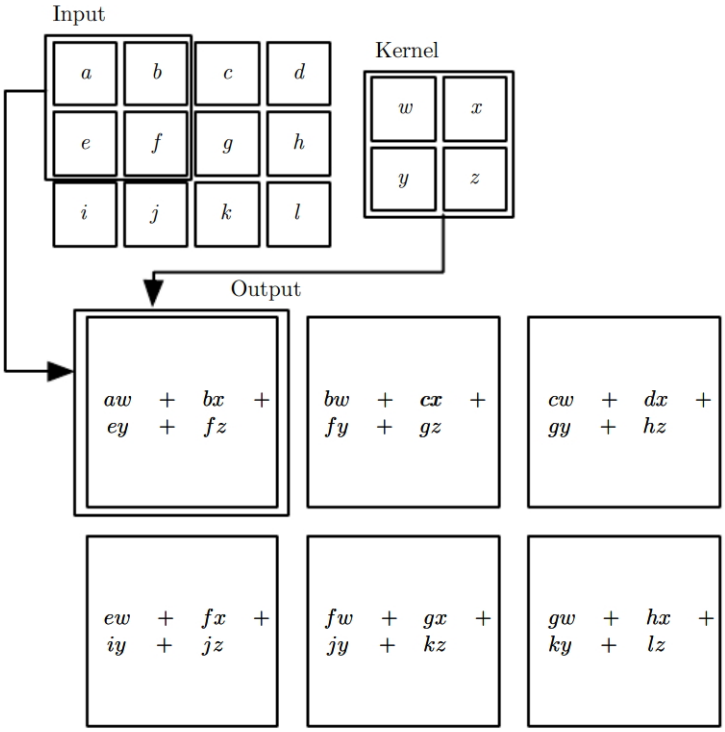

Figure 3.4: 2-D convolution operation with stride=1 and the resulting output (feature map). Each square below the label output denotes one entry in the feature map. Image obtained from [22].

dimensional tensor and K a two dimensional kernel with $k_1 \times k_2$ dimensions, the convolution operation is defined as

$$S(i,j) = (T * K)(i,j) = \sum_{m}^{k_1-1} \sum_{n}^{k_2-1} T(i+m, j+n) K(m,n) \tag{3.14}$$

where the kernel K literally "travels" over the given tensor T. The amount of the increment of i and j (which is always the same for both) is called the **stride**. In order to be able to equate the dimension of the resulting feature map with the input tensor, sometimes the input is padded with either zeroes or ones across height and width dimensions. In convolutional neural networks, convolution operation is executed by convolutional layers that contain a collection of filters (kernels) K and biases b associated with each filter. For an input tensor $T \in \mathbb{R}^{H \times W \times D}$, F number of filters $K \in \mathbb{R}^{k_1 \times k_2 \times D \times F}$ with $k_1 \times k_2$ dimensions and F number of biases $b \in \mathbb{R}^F$, the output of the convolution operation is yielded by

$$(T * K)_{ij} = \sum_{m=0}^{k_1-1} \sum_{n=0}^{k_2-1} \sum_{d=0}^{D-1} T_{i+m,j+n,d} \cdot K_{m,n,d} + b_f \tag{3.15}$$

where b_f is the bias of the fth filter and $T \cdot K$ denotes the dot product. After convolution is computed, the resulting value is run through a non-linear activation

function

$$f((T * K)_{ij}) = O_{ij} \tag{3.16}$$

which gives the final output of the whole operation. Activations are also sometimes referred to as having their own layer, however that notation will not be used in this paper.

Pooling

Following the creation of feature maps, it is often part of the practice to expose them to an operation called pooling. The layer that performs this operation is called a **pooling layer**. Pooling operation is used to replace the output of the convolution operation at a certain location with the statistical summary of the neighbouring outputs. The goal of the pooling layer is to reduce the dimension of the output of the convolutional layer while keeping statistical heuristics about the data itself, which in turn allows these outputs to become invariant to small translations of the input. Pooling layers do not go through learning [22, 41]. Among a few others, there are two mostly-used pooling types, namely max-pooling and average pooling. For window dimensions of $k \times k$, max-pooling layer selects the maximum value among $k \times k$ neighbouring data of the output O_{ij} centered at i, j. Similarly, average pooling behaves the same way where the only difference is that instead of selecting the maximum, it averages the values of all $k \times k$ neighbouring data. Pooling layers also have the option of operating based on a defined stride value, which is by default equal to 1.

3.2.2 Recurrent Neural Networks

Artificial neural networks that adopt a topology in which neurons have feedback connections are called recurrent neural networks (RNNs) [20]. Recurrent neural networks are usually picked for usage when the data is of sequential fashion. There are many variants of these networks, as one can achieve recurrence in a network as per many feedback connections as possible. Therefore, in order to be able to give the general idea of how a recurrent neural network functions and what kind of topology it has, a generalised explanation will be followed. Figure 3.5 depicts a simple, enclosed version of a recurrent neural network. Recurrent networks have loops. That is, each input data presented at time step t is handled by parameters that are shared recurrently across the network from the previous time step $t - 1$. This allows information to be passed from one step to the next, ultimately persisting this very information. This phenomenon can easily be seen on figure 3.6. For each time step t, the following equations govern the internal operations in a recurrent neural network:

$$\begin{aligned} h^{(t)} &= f(\boldsymbol{b} + \boldsymbol{W} h^{(t-1)} + \boldsymbol{U} x^{(t)}) \\ o^{(t)} &= \boldsymbol{c} + \boldsymbol{V} h^{(t)} \end{aligned} \tag{3.17}$$

where vectors $\boldsymbol{b}$ and $\boldsymbol{c}$ are biases, and $f \colon \mathbb{R}^{\mathbb{N}} \to \mathbb{R}^{\mathbb{M}}$ is a non-linear activation function. Looking at the equations, it is easy to see that the value of the hidden state at time

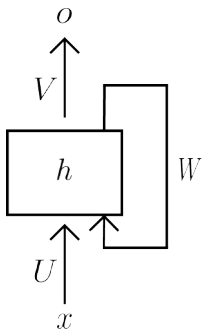

Figure 3.5: A simple RNN diagram. Input x is mapped to output o. The middle square represents the hidden units h in the network. Matrix U is the weights matrix between the inputs and hidden units. V is the weights matrix that parametrises the connections between hidden units to the outputs. W is the recurrent weight matrix that interacts only with hidden units.

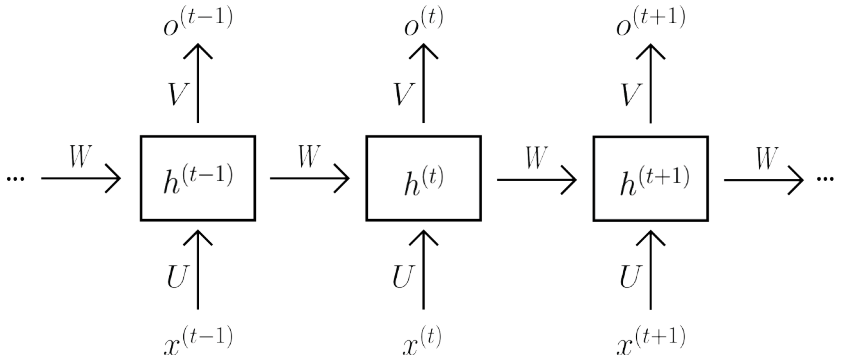

Figure 3.6: Unrolled version of the simple RNN diagram in figure 3.5.

t, depends on the previous value obtained at time step $t-1$, hence the recurrence. Initially, the hidden state is usually all zeroes. Parameter sharing allows recurrent neural networks to construct and maintain a single model that is able to operate on all time steps, which in turn allows generalization. After this adequate introduction to recurrent neural networks, it is time to introduce a more specialized version of them called long short-term memory, which takes direct part in the proposed model in chapter 4.

3.2.2.1 Long Short-Term Memory

Although highly useful, recurrent neural networks are often challenged by a phenomenon called the vanishing gradient problem that arises from the long-term dependencies [28, 29]. As can be seen in section 3.2.3, parameter learning in artificial neural networks involves calculation of gradients of an error function, and, in recurrent neural networks, these gradients that are propagated through the network over many stages tend to unfortunately vanish, i.e. approach to zero, especially when the used activation functions have derivatives that converge to zero really fast. In

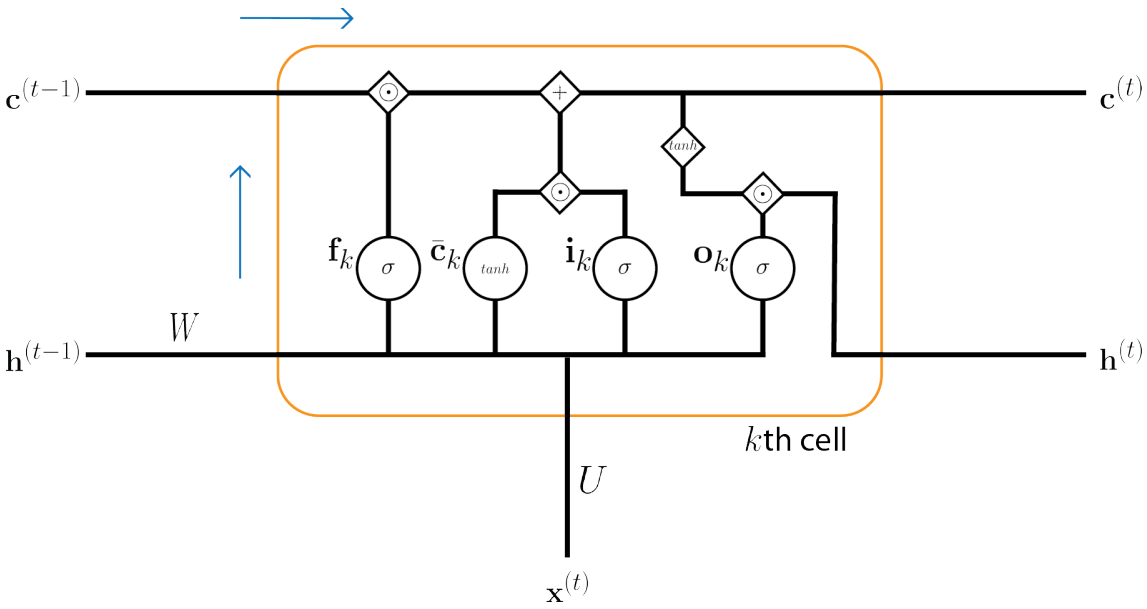

Figure 3.7: Gates and junctions inside an LSTM cell unit. Blue arrows depict the direction of information flow.

order to be able to overcome this problem, one solution is to use **gated RNN**s. Gated RNNs bare gates and paths that prevent the gradients from vanishing via having derivatives that persist over time. Two popular architectures of this fashion are gated recurrent unit (GRU) [12] and long short-term memory (LSTM) [30] architectures. Because of their direct involvement in the model presented in chapter 4, long short-term memory networks will be the focus of this section.

As previously stated, LSTMs consist of gates that are basically specialised junctions of neural connections. These junctions make LSTMs very capable of remembering long-term dependencies of data. From speech recognition, handwriting recognition to machine translation, LSTMs have been shown to be very successful in many applications [2, 24, 25]. A diagram depicting the gates and junctions existing in a single LSTM cell can be seen in figure 3.7. There are a total of three gates in an LSTM cell, namely forget gate $\mathbf{f}_k$, output gate $\mathbf{o}_k$ and input gate $\mathbf{i}_k$ for the kth cell. These gates are basically nothing other than separate single-layered neural networks with dedicated activation functions. In addition to a conventional (also called vanilla) RNN architecture, LSTMs additionally have these inner recurrences on the top of the outer recurrence. Apart from the hidden state vector $\mathbf{h}$, the vector variable $\mathbf{c}$ introduces an information called the cell state, which is, just like the hidden states, carried through the cells over the time steps and can be thought of as the variable that adds memory capabilities to an LSTM unit. The inner neural network (as in the "gates") $\bar{\mathbf{c}}$ is called the candidate layer and controls the flow of information for the cell state. Equations governing the internal calculations for the kth LSTM cell

are as follows:

$$
\begin{aligned}
f_k^{(t)} &= \sigma\left(\sum_j x_j^{(t)} U_{k,j}^f + \sum_j h_j^{(t-1)} W_{k,j}^f + b_k^f\right) \\
i_k^{(t)} &= \sigma\left(\sum_j x_j^{(t)} U_{k,j}^i + \sum_j h_j^{(t-1)} W_{k,j}^i + b_k^i\right) \\
o_k^{(t)} &= \sigma\left(\sum_j x_j^{(t)} U_{k,j}^o + \sum_j h_j^{(t-1)} W_{k,j}^o + b_k^o\right) \\
\bar{c}_k^{(t)} &= \tanh\left(\sum_j x_j^{(t)} U_{k,j}^{\bar{c}} + \sum_j h_j^{(t-1)} W_{k,j}^{\bar{c}} + b_k^{\bar{c}}\right) \\
c_k^{(t)} &= f_k^{(t)} \odot c^{(t-1)} + i_k^{(t)} \odot \bar{c}_k^{(t)} \\
h_k^{(t)} &= \tanh(c_k^{(t)}) \odot o_k^{(t)}
\end{aligned}
\tag{3.18}
$$

where $\mathbf{x}^{(t)}$ is the current input vector for time step t. The variables $\mathbf{b}$ are the biases, $\mathbf{U}$ are the input weights, $\mathbf{W}$ are the recurrent weights; the values of these vectors differ for each gate. σ denotes the sigmoid activation where tanh denotes the hyperbolic tangent activation. After all these calculations, vectors $c_k^{(t)}$ and $h_k^{(t)}$ are passed on to the $(k+1)$th cell, to which the kth cell is recurrently connected to.

3.2.3 Learning in Artificial Neural Networks

Learning in artificial neural networks, as in many machine learning practices, revolves around optimization. That is, finding the parameters θ of the neural network that minimizes a cost function $J(\theta)$. In a typical setting, the cost function is written as

$$J(\theta) = \mathbb{E}_{(x,y)\sim p_{data}} L(f(x;\theta), y) \tag{3.19}$$

which gives the average error over the training set where x is the input, $f(x;\theta)$ is the predicted output, and L is the loss function per example that calculates how far apart the predicted output is from the given input quantitatively. Here, p_{data} is the data generating distribution. The value of the function $J(\theta)$ is called the **risk**. In an ideal situation where the data generating distribution $p_{data}(x,y)$ was known, the problem itself would just be a simple optimization task. However, in a machine learning environment, the training data at hand gives an empirical overview of the true distribution, that is, rather than the generating distribution itself, what is known is the empirical distribution $\hat{p}_{data}(x,y)$ of data defined by the very training set. Therefore, the form that the cost function takes is

$$J(\theta) = \mathbb{E}_{(x,y)\sim \hat{p}_{data}}[L(f(x;\theta), y)] = \frac{1}{N}\sum_{i=1}^{N} L(f(x_i;\theta), y_i) \tag{3.20}$$

which is called the **empirical risk** where N is the total number of training examples. Since the goal of a neural network is to approximate the function $f(x;\theta)$ that is

parametrized by θ, finding the best values $\hat{\theta}$ out of all possible values of θ that minimizes $L(f(x;\theta), y)$, and which, in turn, minimizes $J(\theta)$ is the ultimate objective that can now be formulated as

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}}\, J(\theta) = \underset{\theta}{\operatorname{argmin}}\, \frac{1}{N} \sum_{i=1}^{N} L(f(x_i;\theta), y_i) \tag{3.21}$$

which is called **empirical risk minimization**. Through empirical risk minimization, it is hoped that the learned (optimized) parameters $\hat{\theta}$ will obtain an adequate generalization so that the risk will have also been minimized as well. However, this optimization problem has no closed-form solution and finding optimal set of parameters $\hat{\theta}$ is an NP-hard problem [1]. The most commonly used method for calculating this quantity involves the usage of gradient information [22].

3.2.3.1 Gradient Descent

Gradient descent methods are used to find good parametrization $\hat{\theta}$ of a neural network. The simplest approach to using gradient information involves choosing the value of the parameters θ in the direction of the negative gradient, such that

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_\theta J(\theta^{(t)}) \tag{3.22}$$

where t denotes the time step and η is called the **learning rate**. Practically, gradient descent methods iteratively update parameters θ at each time t via taking a step of magnitude η towards the direction of a better solution pointed by the (negative) gradient information. As indicated by vector calculus, this direction is a vector which is a collection of the partial derivatives $\nabla_\theta J(\theta) = \{\frac{\partial J}{\partial \theta_i}, \ldots, \frac{\partial J}{\partial \theta_N}\}$ of function $J(\theta)$ at point $\theta^{(t)}$ for all parameters $\theta = \{\theta_1, \theta_2, ..., \theta_N\}$. As can be seen, gradient descent requires both the loss function L and thus the function f to be differentiable.

Stochastic Gradient Descent with Minibatches

Computing the expectation in equation 3.21 through 3.22 could be computationally very expensive, especially when the dataset is large [4]. In order to cope with this complication, one can randomly sample a small number of examples from the dataset and then by averaging these can obtain an unbiased estimation of the exact gradient. Algorithms that use single one or less than all of the training examples are called **stochastic** or **online**. In the realm of neural networks, usually a small subset of all of the training examples that is drawn i.i.d from the dataset is presented to the model. These subsets are named **minibatches**. That is, sampling m number of training examples $\{x_1, x_2, ..., x_m\}$ from the entire dataset with the corresponding target values y_i and then computing the gradient of the loss with respect to that minibatch:

$$\hat{g} = \frac{1}{m} \nabla_\theta \sum_{i=0}^{m} L(f(x_i;\theta), y_i) \tag{3.23}$$

where updating the parameters θ in the direction of $\hat{g}$ would be performing stochastic gradient descent on the generalization error. In order to get an unbiased estimation of the gradient, a repeated random sampling of the same minibatch however would not be sufficient, as after the first pass over all of the training examples, the sampling process would produce the same minibatches which ultimately would drive the estimation to be biased [22]. Thus, a good rule of thumb would be to reshuffle the the entire collection of training examples at the end of every pass, so as to avoid bias and increase the randomization of the sampling process.

3.2.3.2 Backpropagation

An efficient (and currently the most popular) way of evaluating the gradient of an error function is the backpropagation algorithm [53]. The work principle of backpropagation algorithm is as follows: The inputs x given to a network provide the initial information which flows through the network and finally reaches the output producing the output $\hat{y}$. This procedure via which the internal and the final output is computed is called the **forward pass**. After the final output is produced, the value of the cost function $J(\theta)$ is calculated. From this point on, the information obtained by the value of the cost function can be backpropagated though the network in order to compute the gradient, which is called the **backward pass**. Note that backpropagation is not a stage of learning but rather a way of calculating the gradient of any differentiable multi-variate function [4, 22]. That is why backpropagation algorithm is and *can be* used for any type of neural network architecture for the purpose of evaluating partial derivatives.

3.2.3.3 Adam Algorithm

There exist many strategies/algorithms for the purpose of reducing the training time of gradient descent based algorithms. These algorithms generally deal with adjusting the learning rate, as learning rate is a hyperparameter of utter importance owing this to its impact on the model performance. One of these strategies is to adjust learning rates adaptively, which is adopted by a few algorithms. One of the algorithms that execute this is Adam (adaptive moments) [39].

Adam works by keeping track of the first and second moment estimates of the gradient. That is, after computing the gradient $\hat{g}^{(t)}$ as

$$\hat{g}^{(t)} = \frac{1}{k}\nabla_\theta \sum_{i=0}^{k} L(f(x_i;\theta), y_i) \tag{3.24}$$

at time step t for k samples, two variables $m^{(t)}$ and $v^{(t)}$ are updated through

$$\begin{aligned} m^{(t)} &= \beta_1 m^{(t-1)} + (1-\beta_1)\hat{g}^{(t)} \\ v^{(t)} &= \beta_2 v^{(t-1)} + (1-\beta_2)\hat{g}^{(t)} \odot \hat{g}^{(t)} \end{aligned} \tag{3.25}$$

where β_1 and β_2 are called the exponential decay rates for the moment estimates, and where $m^{(t)}$ and $v^{(t)}$ are called the biased first and second moment estimates

respectively. $\odot$ is the element-wise multiplication, also known as the Hadamard product. After the biased moments are calculated, Adam moves on to correct the bias in these terms, such that

$$\begin{aligned} \hat{m}^{(t)} &= m^{(t)}/(1-\beta_1^i) \\ \hat{v}^{(t)} &= v^{(t)}/(1-\beta_2^i) \end{aligned} \tag{3.26}$$

where β_1^i and β_2^i are exposed to time dependent regularization at every iteration i. Having computed all the intermediate variables, the final value of the operation, which is the value of the update, can now be calculated. That is,

$$\Delta\theta = -\frac{\eta\hat{m}^{(t)}}{\sqrt{\hat{v}^{(t)}}+\epsilon} \tag{3.27}$$

where ϵ is a small constant used for numerical stabilization that prevents the divisor from getting too close to zero. With this value at hand, the gradient update can be rewritten as

$$\begin{aligned} \theta^{(t+1)} &= \theta^{(t)} + \Delta\theta \\ \theta^{(t+1)} &= \theta^{(t)} - \frac{\eta\hat{m}^{(t)}}{\sqrt{\hat{v}^{(t)}}+\epsilon} \end{aligned} \tag{3.28}$$

which concludes the working mechanisms of Adam.

Chapter 4

Proposed Model

In this chapter, we show our proposed model. Firstly, we explain the preliminary technologies/methods adopted. Then, we move on to presenting the building blocks of the model, which is organised in three sections. In the first section, we introduce our novel approach to integrating the analysis of structural similarity of texts into neural networks. In the second part, we demonstrate the neural network architecture we construct for the analysis of semantic similarity of texts. In the last section, we talk about the last form of the model through which the unification of both structural and semantic similarity analyses takes place.

4.1 Preliminaries

4.1.1 Word Embeddings: word2Vec to fastText

Word embeddings is the collective name given for the process of mapping words to vectors or real numbers. It is used to define the set of natural language techniques that are used for feature learning/engineering and language modelling.

The breakthrough of word embeddings happened by the word2vec method proposed by Mikolov et. al. [47]. With the help of simple neural networks, they were able to map words to a new high dimensional space where the positions of words were decided on based on their semantic relatedness. Naturally, this ultimately means that each word is converted to a vector. Therefore, the cosine similarity between for example the word "Berlin" and "London" would be higher than "Berlin and "car". The positions of the vectors were learned by the neural network through corpuses containing billions of words, based on their frequency of co-occurrence. we will not be going into the details of this process.

fastText [5], on the other hand, is also a word embedder that takes what word2vec started even further. The key difference between word2vec and fastText is that fastText uses word character n-grams to map the words to the high dimensional space. This in turn allow the model to be more robust and less sensitive to words that either do not exist in the corpus used for training or are very rar. To cut it short, one can say that fastText is the drastically improved version of word2vec that

do not alienate unknown words, which is one of the reasons we stick to only fasTtext throughout model evaluation.

4.1.2 Siamese Architecture

A Siamese network [8] is an architecture for non-linear metric learning with similarity information. The word "siamese" comes from the biological term used to explain twins who are born stuck together sharing various body parts and organs. Just like this, each sample in the dataset runs through a siamese network one by one and gets exposed to same parameters. In other words, data share the parameters of the network while being processed. By exposing different data to same parameters, a siamese network naturally learns representations that embody the invariance and selectivity of the data. The term different data here means that the pairs of data, whose relationship among themselves are tried to be learned/explored. This paradigm allows a network to pinpoint the key aspects of data that makes it either unique or similar to others. Thus, a siamese network can be though of as an unsupervised feature engineering mechanism that automatically deciphers the intricate relationships among data.

4.2 The Model

4.2.1 Neural Network for Structural Similarity Analysis

For the explanation of integration of structural similarity to neural networks, we start with core ideas and definitions behind, and then build up to the last state of the integration. Terms *string* and *word* are used interchangeably. Unlike the models in the following two sections, whose training phases are explained in chapter 5, the details of training this network will be included in this chapter.

4.2.1.1 Treating Words as Relations

For the first stage, we employ a relation based algebra, and turn each word into relation matrices. For this, we treat each word as a relation. We start by defining the concept of relation matrix [31].

Definition 1. *If R is a binary relation between the finite indexed sets X and Y such that $R \subseteq X \times Y$, then R can be represented by the relation matrix M whose row and column indices index the elements of X and Y respectively where the entries of M are of the form:*

$$M_{i,j} = \begin{cases} 1 & (x_i, y_j) \in R \\ 0 & (x_i, y_j) \notin R \end{cases}$$

for $0 < i < |X|$ and $0 < j < |Y|$.

In order to turn words intro relations, we equip two sets C and P, where $C = \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, r, s, t, u, v, w, x, y, z, ä, ö, ü, ß(ss)\}$ is the set

of characters in the alphabet with $|C| = 30$, and $P = \{p \,|\, p \in \mathbb{N} < 18\}$ is the set of positions of these characters as we set 18 to be the maximum character length of the longest word that the model can accommodate. Accordingly, a word can be defined by the relation $R_w \subseteq C \times P$ for any word w whose characters are a subset of C and whose maximum length is less than or equal to $18 = |P|$ characters. For example, the relation for word `gehen` would then be $R_{\texttt{gehen}} = \{(g,0),(e,1),(h,2),(e,3),(n,4)\}$, for which the relation matrix is defined as

$$\boldsymbol{M}_{\texttt{gehen}} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \\ \vdots \\ n \\ \vdots \\ ss \end{array} \begin{array}{c} \begin{array}{cccccccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & \cdots & 17 \end{array} \\ \left[\begin{array}{cccccccccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \end{array}\right] \end{array}$$

As can be seen, this matrix is very sparse. However, this sparsity is a natural result of the size of the alphabet and the maximum word length allowed. We will be going into the detail of how we deal with this sparsity and the size of these matrices shortly. Before that, we introduce our novel similarity metric.

4.2.1.2 Relational Similarity Metric

Relational similarity metric we propose is a novel string similarity metric that utilizes relation matrices of strings (in this case, words) to produce a scalar value between 0 and 1 to indicate how similar two given strings are. We start by defining this metric.

Definition 2. *Let w_1 and w_2 denote two strings that correspond to two words. Let w_1^r denote the reverse of the string w_1, and w_2^r of the string w_2 respectively. Let $\boldsymbol{M}_w$ define the relation matrix for word w. Relational similarity $sim_{rel}(w_1, w_2) \in [0, 0.99]$ is*

$$sim_{rel}(w_1, w_2) = \frac{\sum_{i,j}\Big[(\boldsymbol{M}_{w_1} \odot \boldsymbol{M}_{w_2})_{i,j} + (\boldsymbol{M}_{w_1^r} \odot \boldsymbol{M}_{w_2^r})_{i,j} + \big|(\boldsymbol{M}_{w_1} \odot \boldsymbol{M}_{w_2})_{i,j} - (\boldsymbol{M}_{w_1^r} \odot \boldsymbol{M}_{w_2^r})_{i,j}\big|\Big]}{\sum_{i,j}\Big[(\boldsymbol{M}_{w_1} + \boldsymbol{M}_{w_2})_{i,j}\Big] + \epsilon} \tag{4.1}$$

where ϵ is a very small number for protection against division by zero.

The driving motivation behind the invention of this similarity metric is because of the fact that, naturally, there exists no similarity metric for binary matrices that

s_1	s_1	$sim_{rel}(s_1, s_2)$
bearbeitung	ableitung	0.699
abteilung	ableitung	0.777
gesellschaft	freundschaft	0.499
gesellschaft	ffreundschaft	0.479
gesellschaft	fffreundschaft	0.461
gesellschaft	fffreundschaftt	0.148
gesellschaft	fffreundschafttt	0.071
freundschaft	ffreundschafttt	0.074
freundschaft	ffreund	0.105
freundschaft	schaft	0.666
freundschaft	shcaft	0.444
freundschaft	freund	0.666
freund	freund	0.999

Table 4.1: Relational similarity score of example string pairs.

would in the end give a meaningful outcome as to whether two strings are similar or not. That is, depending on our method of converting words into relation matrices, we needed a metric that would not only be lightweight but also robust enough for the next stages of integration of structural similarity into neural networks. This need will become much clearer as this section progresses. Relational similarity metric favours matching prefixes, matching suffixes, and orders of the matching characters all together. Table 4.1 showcases a few examples for the value of relational similarity metric for some word/string pairs.

4.2.1.3 Neural Network Integration

The main idea of turning words into relations and then to relation matrices serves the very purpose of being able to feed words to a neural network as data for training and, ultimately, for inference. That is, we require a neural network architecture which, when given two words, is able to produce a similarity score between these words. However, due to the aforementioned sparsity/dimensionality complication, it is theoretically and technically impossible to do so, as the training of such neural network would be impractical because of one's possibility of having 31^{18} unique relation matrices (with columns consisting entirely of zeros included). In order to cope with this, we execute a method similar to divide & conquer strategy. Since relation matrices are composed of nothing other than zeroes and ones, instead of feeding the relation matrices of words as a whole, we propose a method where we separate a relation matrix belonging to a word to its 3×3 sub-matrices $\in \mathbb{R}^{3\times3}$, and then feed these one by one as pairs to a neural network that produces a score matrix $\boldsymbol{S} \in \mathbb{R}^{\frac{30}{3}\times\frac{18}{3}}$ whose entries in the end bare the similarity score between each of the corresponding sub-matrices of two words. After this, the whole matrix S is

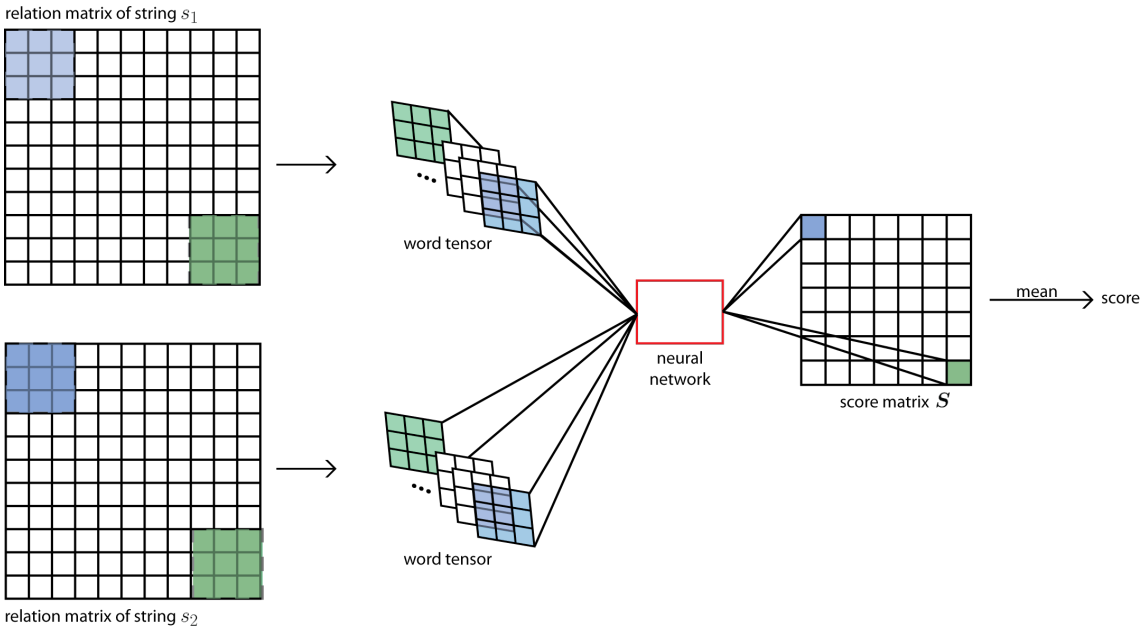

Figure 4.1: Diagram depicting an overview of the workflow of relational similarity calculation through a neural network after two strings (words) s_1 and s_2 have been converted to their corresponding relation matrices. Sub-matrices in word tensors are paired with each other according to their positional correspondence, and go through the network pair by pair.

reduced to its mean, which, finally, gives the similarity score of two words (strings). Separating a relation matrix to its sub-matrices ultimately means converting them to tensors, which we name as word tensors. These tensors are of shape $(3, 3, 60)$. Figure 4.1 shows this process. This is the single core idea behind the integration of structural similarity metric to neural networks. That is, through the introduction of similarity metric calculation for sub-matrices (just as in receptive fields in a convolutional neural network), we are able to achieve **language invariant** string similarity analysis done by an artificial neural network. At this point, the need for the newly introduced relational similarity metric becomes clear, as it is the metric used for teaching the neural network distances/similarities of these sub-matrices.

Converting Binary Relation Matrices to Vectors

Before digging more into the details of the main neural network for structural similarity, we first explain our strategy of further dimensionality reduction. This is also essential because if each unique 3×3 sub-matrix were to be treated as a class, one would in total end up with 2^9 classes. This is not exactly the case for our model, because, depending on the way the relation matrix is constructed, it is technically not possible for one column to have more than one ones (1s), which reduces the number of classes. However, this number is, then, 2^6. Furthermore, given the fact that the neural network for calculating structural similarity is expected to process pairs of these sub-matrices, it would mean that this network has to learn $2^6 \times 2^6 = 4096$

different combinations, which can still be considered a lot. Therefore, in order to ease the work for the main network and reduce the training time, we first employ an autoencoder that maps the sub-matrices to vectors of dimensions of four. That is, we seek an encoder function $\Phi(\boldsymbol{X})$: $\boldsymbol{X} \in \mathbb{R}^9 \rightarrow \hat{\boldsymbol{X}} \in \mathbb{R}^4$ to be used as an encoding (embedding) layer.

The autoencoder we construct for this purpose is a simple undercomplete autoencoder with two three-layered multilayer perceptron networks where one serves the purpose of encoding and the other decoding as per usual. The amount of neurons for each layer $l_i \in \{l_{input}, l_1, l_2, l_3, l_{output}\}$ of the encoding network is $l_{input} : 9, l_1 : 90, l_2 : 30, l_3 : 4$, and $l_{input} : 4, l_1 : 30, l_2 : 90, l_{output} : 9$ for the decoding network respectively. We use ReLU activation in all layers, and Adam as the optimizer for the network with the default parameters suggested in [39]. We set the starting learning rate to be 0.001 and reduce it to $0,0001$ when a plateau is encountered. We also clip the values of gradients to be equal to 1.0 maximum, so as to fasten the learning. Note that we first flatten (vectorize) sub-matrices ($\mathbb{R}^{3\times3} \rightarrow \mathbb{R}^9$) and turn them into simple bit strings. As dataset, we use all of the 2^6 unique sub-matrices and set the number of epochs to be 1000. However we also apply early stopping, such that the training stops when the loss reaches zero, which is achieved around 140th epoch. We use mean squared error as the loss function. Unlike conventional view of generalization, it is important for the loss to reach zero as we want this network to map all sub-matrices perfectly to their corresponding vectors in the newly induced latent space. As long as the loss reaches zero, we are sure that the perfect mapping is achieved and thus do not actually care about the latent space.

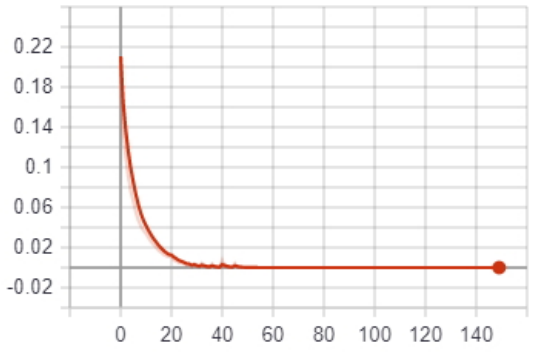

Figure 4.2: Training loss (left) vs. epoch (bottom) graph for the autoencoder network.

The Main Network

For the main network for determining structural similarity of words we employ partial siamese architecture where the encoder we introduced before is also a part of this network. Diagram of the entire network can be seen in figure 4.3. The whole network consists of an encoding layer, a convolutional layer and a multilayer perceptron. The siamese part corresponds to the encoding layer and the convolutional layer. That is, one by one each sub-matrix in a tensor go through the same encoder and convolutional layers. After this, the output of the convolutional layer for each sub-matrix is vectorized, then both of the vectors are concatenated and fed to a multilayer perceptron to produce a similarity score.

The architectural properties of the network is as follows: We employ $50 = F$

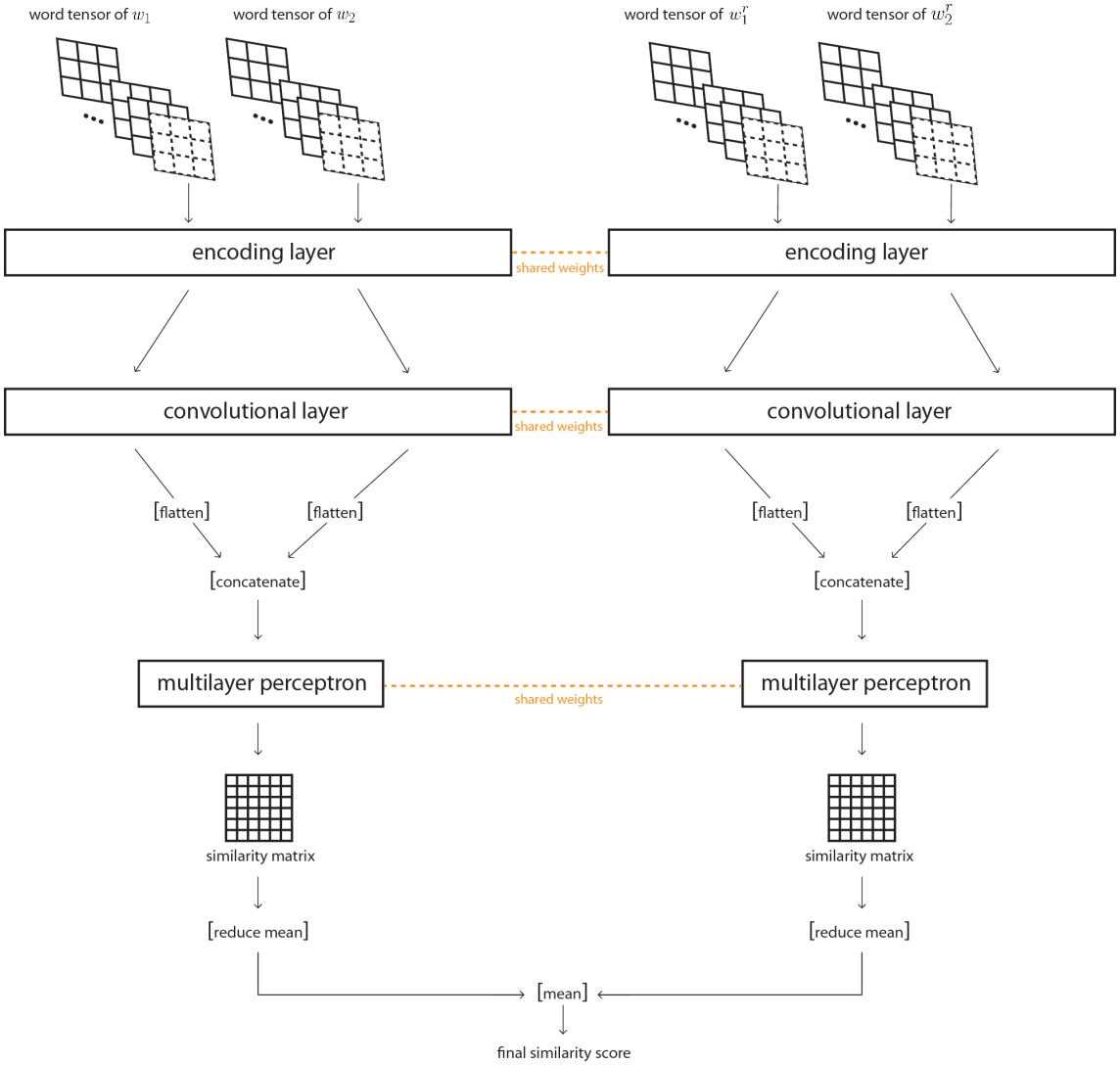

Figure 4.3: Complete diagram of the main neural network architecture for structural similarity analysis. Words (i.e. strings, and their respective reverses) are shown in their tensor forms. Shared weights denote the multiple usage of the same subnetwork.

filters in the convolutional layer, each with dimensions of 2×2 and do not apply any padding. The dimensionality of the tensor produced by the convolutional layer is $1 \times 1 \times 50$, i.e. each filter produces a single scalar value. The amount of neurons for each layer $l_i \in \{l_{input}, l_1, l_2, l_3, l_{output}\}$ of the multilayer perceptron network is $l_{input} : 100, l_1 : 50, l_2 : 25, l_3 : 5, l_{output} : 1$. For digging further into the details, we start by explaining the workflow of the network for two sub-matrices.

Let $\boldsymbol{T}^{w_i} \in \mathbb{R}^{3\times3\times60}$ denote the word tensor of the word w_i, and $\boldsymbol{t}_k^{w_i} \in \mathbb{R}^{3\times3}$ denote the kth sub-matrix belonging to the this tensor where $0 \leq k < 60$. Let $vec(\cdot)$ denote the flattening (vectorization) of a matrix or a tensor, and let $concat(x, y) =$

$[x, y]$ denote concatenation. For two sub-matrices $(\boldsymbol{t}_k^{w_1}, \boldsymbol{t}_k^{w_2})$, the steps of similarity calculation are as follows:

1. Vectorize through $vec(\boldsymbol{t}_k^{w_1})$ and produce $\boldsymbol{v}_k^{w_1}$.
2. Encode $\Phi(\boldsymbol{v}_k^{w_1}) = \hat{\boldsymbol{v}}_k^{w_1}$.
3. Feed $\hat{\boldsymbol{v}}_k^{w_1}$ to convolutional layer with parameters $\boldsymbol{\theta}_c$ and produce a tensor of feature maps $\boldsymbol{z}_{k,f}^{w_1} \in \boldsymbol{Z}_{\boldsymbol{k}}^{\boldsymbol{w_1}}$ for each filter $f = 1, 2, ..., F$.
4. Vectorize feature maps $vec(\boldsymbol{Z}_k^{w_1}) = \boldsymbol{m}_k^{w_1}$.
5. Run steps 1-4 for $\boldsymbol{t}_k^{w_2}$ and produce $\boldsymbol{m}_k^{w_2}$ in the end.
6. $concat(\boldsymbol{m}_k^{w_1}, \boldsymbol{m}_k^{w_2}) = \boldsymbol{x}_k$
7. Feed $\boldsymbol{x}_k$ to the multilayer perceptron with parameters $\boldsymbol{\theta}_m$ and produce a similarity score $s_k \in [0, 1)$.

As a successor to these sub-steps, each score s_k is inserted to its respective position in the similarity matrix $\boldsymbol{S}^{w_1,w_2}$, after which, all entries of non-zero pairs are averaged to produce the intermediate similarity score belonging to w_1 and w_2. As the second main step, the same process takes place as a whole for the reverse of the words ($w^r{}_1$ and $w^r{}_2$) as well, producing the second intermediate similarity score via averaging of $\boldsymbol{S}^{w^r{}_1,w^r{}_2}$. In the third main step, both intermediate scores are averaged to produce the final similarity score.

Training the Main Network

Our goal with training the main network is to ultimately introduce a new space, where the vectors that are constructed by the mapping $\Omega(\cdot)$ induced by the network are placed according to the similarity metric being taught. However, it should be noted that since our training set is finite, we are not interested in teaching this metric directly to the network, which would require certain properties of metric learning to be fulfilled if one seeks a generalization making the model work for any vector size. The finiteness of our dataset comes from the fact that there is finite number of combinations of zeroes and ones that construct a 3×3 matrix. As previously stated, this number is 2^6, and the number of possible combinations of pairs of two 3×3 binary matrices is 4096, which is the very size of our training set. That is, the target values of our training set D can be expressed as a collection of distances such that $D_{target} = \{d_0(\boldsymbol{t}_0, \boldsymbol{t}_0), d_1(\boldsymbol{t}_1, \boldsymbol{t}_0), ..., d_n(\boldsymbol{t}_i, \boldsymbol{t}_j)\}$ where $0 \leq n \in i \times j < 4095 = N$ and $0 \leq i, j < 2^6 - 1$ for all combinations of every binary matrix pair $(\boldsymbol{t}_i, \boldsymbol{t}_j)$. Here $\times$ denotes Cartesian product. It is because of this finiteness that we are not interested in the inner workings of the mappings, nor do we care about the positions of the vectors so long as all of the distances in the training set are learned (approximated) as successfully as possible by the network. The metric $d(\cdot, \cdot)$ is nothing other than the relational similarity we introduced in section 4.2.1.2, such that

$$d(\boldsymbol{t}_i, \boldsymbol{t}_j) = sim_{rel}(\boldsymbol{t}_i, \boldsymbol{t}_j) \tag{4.2}$$

for any binary matrix $\boldsymbol{t} \in \mathbb{R}^{3\times3}$. Let $f(\boldsymbol{t}_i, \boldsymbol{t}_j; \boldsymbol{\theta}_c, \boldsymbol{\theta}_m)$ denote the function we are trying to approximate, we define our loss function accordingly as

$$L(\boldsymbol{\theta}_c, \boldsymbol{\theta}_m) = \frac{1}{N}\sum_{i}^{2^6-1}\sum_{j}^{2^6-1}(d_{n=i\times j}(\boldsymbol{t}_i, \boldsymbol{t}_j) - f(\boldsymbol{t}_i, \boldsymbol{t}_j; \boldsymbol{\theta}_c, \boldsymbol{\theta}_m))^2, \tag{4.3}$$

which is basically mean squared error.

By turning the problem into a regression problem, thanks to the finite number of target values, we bypass the need to get to know the mapping $\Omega(\cdot)$ or any properties of the latent space(s) and the vectors existing therein. Note that the encoding layer does not take part in the training process; rather, it is attached as a readily-trained network that fastens the training process through reducing dimensionality of the input.

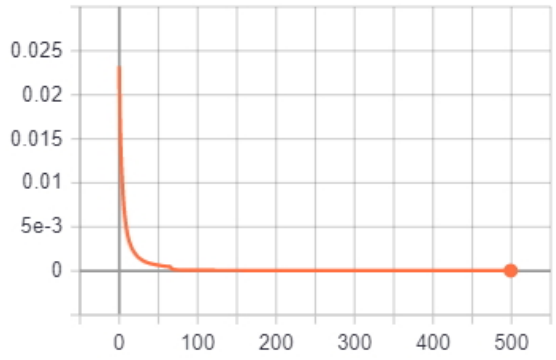

Figure 4.4: Training loss (left) vs. epoch (bottom) graph for the main network. Loss reaches ~ 0.02 after 500 epochs.

We set number of training epochs to be 500 and use Adam as the optimizer with the default parameters suggested in [39]. We employ Xavier (also known as Glorot uniform) initialization of weights for the convolutional layer [21]. We set *ReLU* as the activation function in all layers of the network and the starting learning rate to be 0.001, which is further reduced to 0.0001 if/when a plateau is encountered. We also clip gradients to be 1.0 of maximum value. The network reaches a training loss of 0.02 after 500 epochs, which we take as good enough. The architecture of the whole network was decided using grid search. We require this network to be as deterministic as possible, which ultimately means seeking a training loss value that is as low as possible. Training loss of this network is a valid measurement as to whether the network would also perform well on the test data, since the training data itself is exactly the same as the test data. After numerous trainings, possibly due to the amount of "classes" (i.e. each matrix pair) being very high, 0.02 seems to be the greatest minimum training loss achievable for our model.

4.2.2 Neural Network for Semantic Similarity Analysis

The network we propose for semantic similarity analysis is also a novel model, yet it is not expected from the network to produce state of the art results, as it is mainly used for gauging the effectiveness of the network for structural similarity. The basic idea behind this network is gathering a few select useful parts from the various models in the literature and combining them to achieve favourable-enough results.

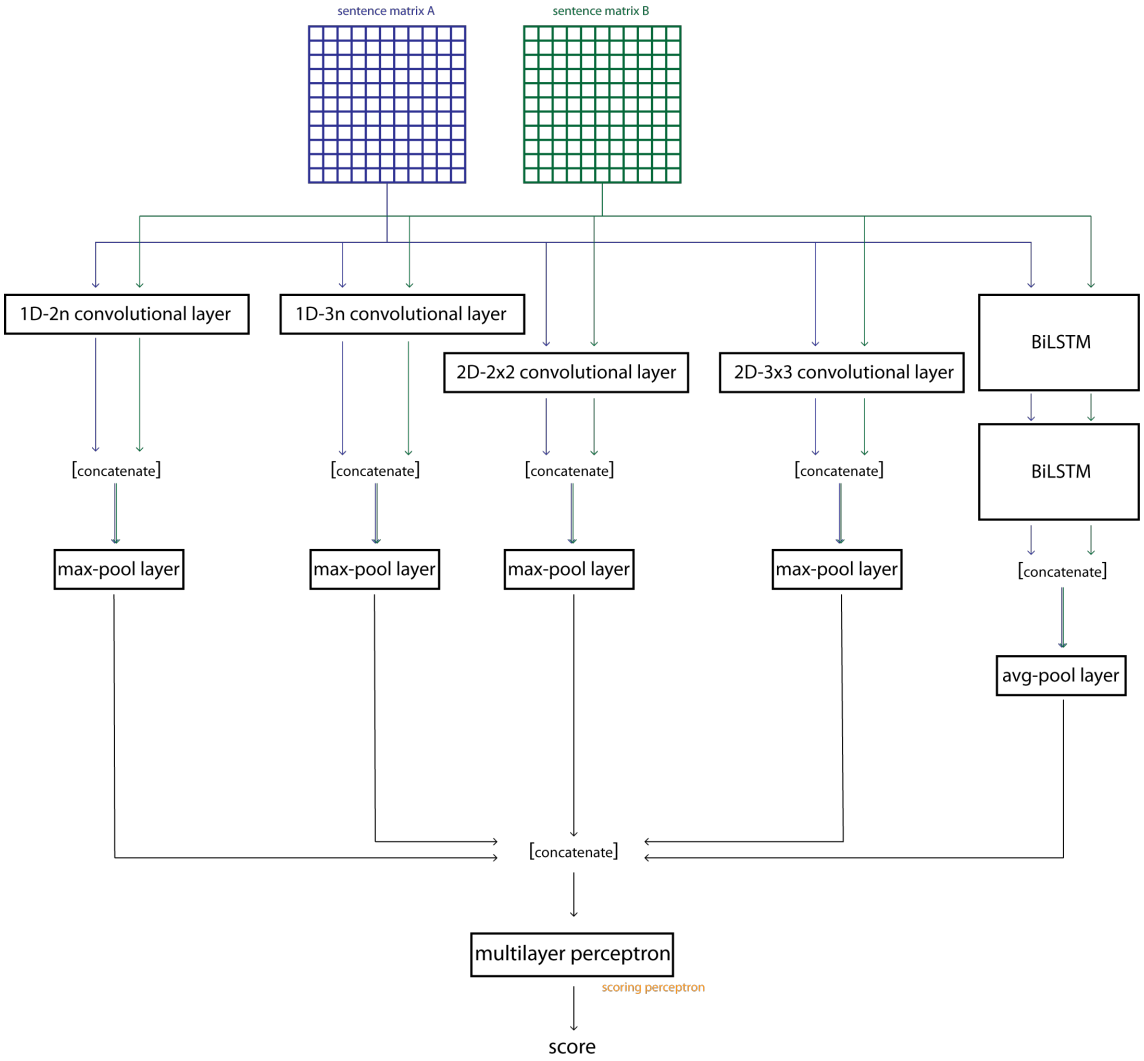

Figure 4.5: Diagram depicting the network for semantic similarity analysis. Sentence matrices go one by one through the blocks of the network until pooling layers.

For this network, we make use of convolutional neural networks, multilayer perceptron networks and recurrent neural networks, specifically long short-term memory networks. Figure 4.5 shows the diagram depicting this network. This network also adopts siamese architecture where it accepts two sentences as inputs and produces a score $\in [0, 9.99]$ indicating the semantic similarity of these sentences. Before being fed to the network, each sentence is first converted to its matrix representation, which is basically the collection of word vectors of each word in the sentence, preserving the word order. The network consists of four convolutional layers, four max-pooling layers, two stacked bidirectional LSTMs (BiLSTMs), one average-pooling layer and one multilayer perceptron network. After the first sentence matrix has been received as the input, it first simultaneously goes through five components: 1-dimensional convolutional layer with window size of 2, 1-dimensional convolutional

layer with window size of 3, 2-dimensional convolutional layer with kernel size of 2×2, 2-dimensional convolutional layer with kernel size of 3×3 and two stacked bidirectional LSTMs with 50 and 100 units respectively. Each convolutional layer has 10 filters. 1-dimensional convolutional layers can be though of as components extracting 2 and 3-grams of the sentence on word by word basis. The first BiLSTM with 50 units is immediately connected to the other BiLSTM with 100 units, hence the name "stacked". BiLSTMs both have tanh as their activation. After the outputs of these components are received, the second sentence matrix also goes through the same components, Following to this, the output vectors of each of these components for the two sentence matrices are concatenated according to their source, i.e., the outputs of each layer are concatenated with their respective counterpart. Then, each of the concatenated vectors is fed to max-pooling layer(s), except for the concatenated output of the stacked LSTMs. The output of LSTM block goes through an average-pooling layer. As a successor to these steps, the outputs of each of the MLPs are concatenated and fed to the last MLP named scoring perceptron, which have five layers with $l_i \in \{l_{input} : 5400, l_1 : 1800, l_2 : 900, l_3 : 300, l_{output} : 1\}$, whereby it ultimately produces the sought similarity score. Except for the last neuron of the scoring perceptron, which has sigmoid as its activation, all neurons in this network employ ReLU as their activation. Note that we alter the number of neurons and the type of the activation on the last layer of the scoring perceptron according to the task. For regression, we employ sigmoid, and for classification, softmax. We test this network alone and also unified with the structural similarity network, details of which we explain in chapter 5.

4.2.3 Unification of Structural and Semantic Similarity Analyses

For the unification of structural and similarity analyses, we basically combine the two above explained methods through altering a few of the existing components and adding a few others. We make use of transfer learning, that is, we do not change the weights of the structural similarity network and hook it onto the semantic similarity network as a new component. The diagram of the unification network can be seen in figure 4.6.

The aim of the structural similarity network in the unification is to aid the semantic similarity network via producing a word-to-word interaction matrix with similarity scores. For this certain output, i.e. the word-to-word interaction matrix, one does not need to change the structure of the structural similarity network. Instead, as a preprocessing step, we repeat and tile the word tensors accordingly, so that the comparison calculations made by the structural similarity network will naturally be of word to word for each combination of the pairs of words. For two sentences $S_1 = \{w_0^{S_1}, w_1^{S_1}, ..., w_n^{S_1}\}$ and $S_2 = \{w_0^{S_2}, w_1^{S_2}, ..., w_k^{S_2}\}$ with n and k number of words respectively, the tiling process produces the input tensors of following fashion:

$$
\begin{matrix}
w_0^{S_1} & w_0^{S_2} \\
w_0^{S_1} & w_1^{S_2} \\
w_0^{S_1} & w_2^{S_2} \\
w_0^{S_1} & w_3^{S_2} \\
w_0^{S_1} & w_4^{S_2} \\
w_0^{S_1} & w_5^{S_2} \\
\vdots & \vdots \\
w_1^{S_1} & w_0^{S_2} \\
w_1^{S_1} & w_1^{S_2} \\
w_1^{S_1} & w_2^{S_2} \\
w_1^{S_1} & w_3^{S_2} \\
w_1^{S_1} & w_4^{S_2} \\
w_1^{S_1} & w_5^{S_2} \\
\vdots & \vdots \\
w_n^{S_1} & w_{k-1}^{S_2} \\
w_n^{S_1} & w_k^{S_2}
\end{matrix}
$$

Basically, each collection of word tensors goes through tiling and repetition in by the amount of maximum number of words (max_words) in a sentence allowed by the model, which we set to be 30. In the tiling process, each word of S_1 is repeated max_words times, where S_2 is repeated max_words times as a whole. Note that the same tiling operation applies to both the sentence tensors themselves and the tensors of their reverses.

The architectural changes to this network involve the removal of the scoring perceptron from the semantic similarity network and addition of it to this network as the main last component. That is, the scoring perceptron is now the score producer of the unified network, where the semantic similarity network now produces a vector that consists of concatenation of the outputs of its pooling layers. Furthermore, we add another LSTM block with 30 units that accepts the output of the structural similarity network, i.e. the word-to-word interaction matrix, size of which is $\in \mathbb{R}^{max\_words \times max\_words}$. The LSTM block here encodes the similarity scores further into collection of vectors, which are then vectorized (flattened) and concatenated with the the outputs of the semantic similarity network. After this concatenation, the resulting "collection" vector is fed to the scoring perceptron to produce the resulting similarity score. During the training phase of this network, the weights of the components of the whole semantic similarity network and plus the newly added LSTM block are optimized, where, as previously stated, the weights of the structural similarity network are kept static.

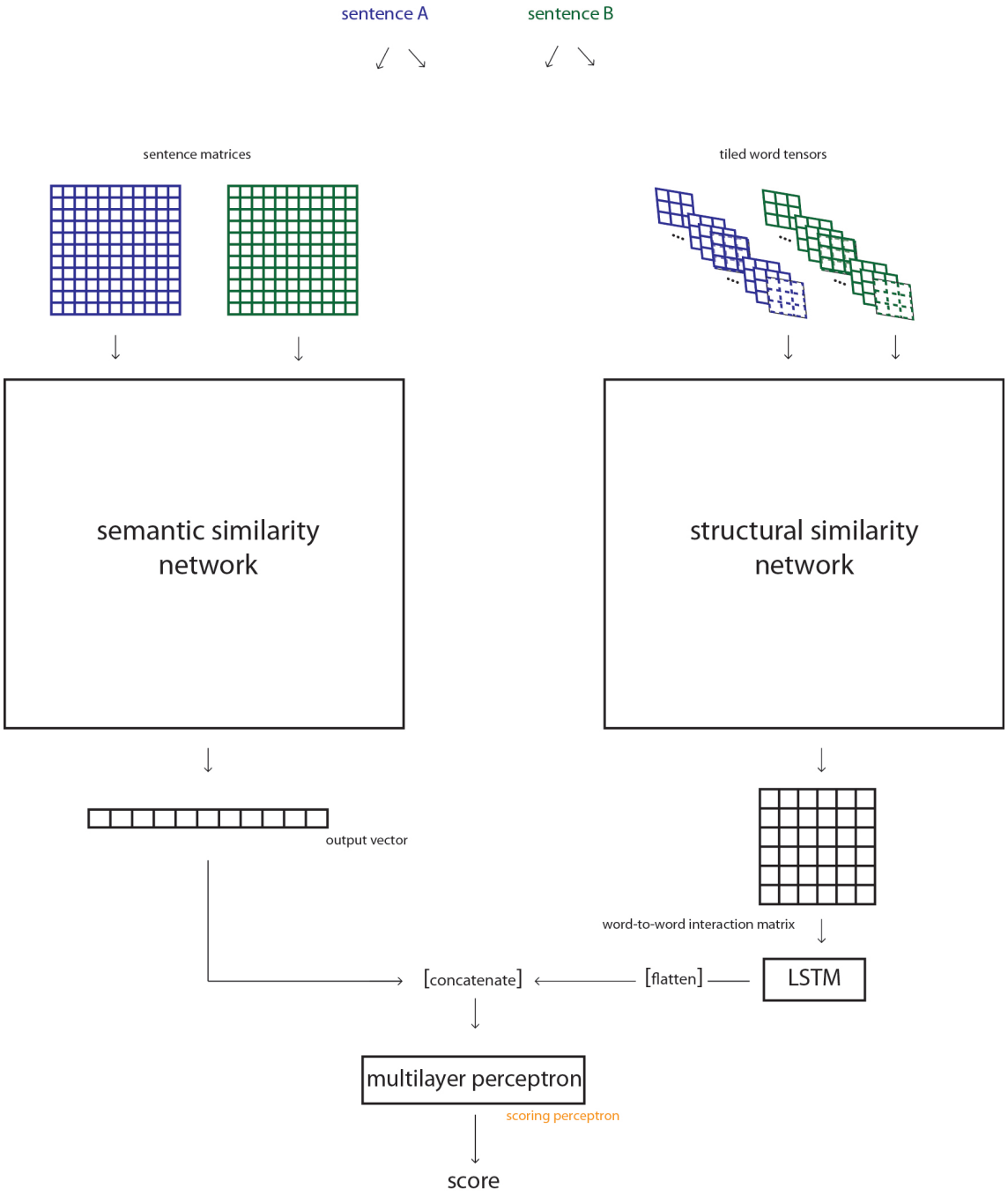

Figure 4.6: Diagram depicting the network for unification of short text similarity analyses.

Chapter 5

Experiments and Discussion

In this chapter, we present the results of our experiments done in four different areas of natural language processing. These are namely semantic entailment recognition, paraphrase detection, semantic textual similarity detection and, lastly, short text matching. Our **main goal** is to test whether the addition of structural similarity information done by a neural network aids the semantic similarity analysis or vice versa. That is why, as previously stated in chapter 4, we do not expect the semantic similarity network to produce state of the art results, rather, it is sufficient for it to be a favourable gauging baseline.

Constraints, Preprocessing and Preliminaries We set maximum length of sentences to be 30 words. As previously stated, we set the maximum character length of words to be 18, and the alphabet size to be 30 characters. We apply padding of zeroes if a sentence or a word is shorter than the corresponding set lengths, so as to ensure size invariance. Should a word or a sentence have lengths greater than set, we simply discard the exceeding parts. As preprocessing, we turn all characters in a given sentence to lower case, remove all of the punctuation and lastly separate any numerals stuck to words, i.e. put a blank in between the word and the numeral. In all our experiments, we use 300-dimensional fastText vectors and Adam algorithm as the optimizer with starting learning rate of 0.001, which is later reduced up to 0.0001 if/when a plateau is encountered.

Implementation Details We use Python 3.6 and the latest version of Keras (with tensorflow backend) as of date. The machine that we run the tests on has 1 x NVidia Tesla P100 GPU, 150GB of RAM and 16 x 1.7 Ghz Intel cores.

Legend We denote our semantic similarity network as SEMN and structural similarity network as STRN. For the unification network, we simply use SEMN + STRN.

SICK-E		
Premise	**Entailment**	**Hypothesis**
The young boys are playing outdoors and the man is smiling nearby.	Entailment	The kids are playing outdoors near a man with a smile.
Two dogs are fighting.	Neutral	Two dogs are wrestling and hugging.
Two dogs are wrestling and hugging.	Contradict.	There is no dog wrestling and hugging.
A boy is hitting a baseball.	Entailment	A child is hitting a baseball.

Table 5.1: Example sentence pairs and their corresponding labels from the SICK-E dataset.

5.1 Semantic Entailment Recognition

Understanding sentence semantics is a core ability that we homo sapiens possess. By arranging and connecting the semantics of sentences, we infer the very natural language we speak. This fact makes semantic entailment recognition one of the most popular natural language processing tasks. In NLP, semantic entailment presents itself as a classification task. That is, given two sentences, semantic entailment tries to find out if one sentence can be inferred from the other. More specifically, it tries to identify the relationship between the meanings of pair of sentences (one named the "premise", and the other "hypothesis"). These relationships are:

- Entailment: the hypothesis is a sentence with a similar meaning as the premise
- Contradiction: the hypothesis is a sentence with a contradictory meaning
- Neutral: the hypothesis is a sentence with mostly the same lexical items as the premise

In order for a neural network to succeed at semantic entailment recognition, it needs to identify lexical entailment, lexical coreference, tense, intention parsing (belief and modality of a speaker), and syntactic ambiguity, which are what happens in our brain.

To test our networks for semantic entailment recognition, we use SICK-E (Sentences Involving Compositional Knowledge) dataset provided by the Center for Mind/Brain Sciences of the University of Trento [1]. In this dataset, there are approx. 10,000 sentence pairs, each labelled according to their entailment (as explained above). For training, we use 9,000 of the sentence pairs, and for testing, we use the remaining 1000. We split the test set via random sampling. We set batch size to 120, and the number of epochs to 50. We set the number of neurons in the last layer of the scoring perceptron to 3 and use softmax as the activation in this layer. We employ cross-entropy loss as our loss function, that is, for two distributions $p(x)$ and

[1] https://wiki.cimec.unitn.it/tiki-index.php?page=CLIC

Classification accuracy, by model name	
MV-RNN [56]	58.14%
DT-LSTM [59]	83.11%
LSTM [59]	76.80%
Bi-LSTM [59]	82.11%
SEMN (ours)	61.48%
SEMN+STRN (ours)	74.02%
2-layer LSTM [59]	78.54%
2-layer Bi-LSTM [59]	79.66%
InferSent [15]	86.62%
Mix-model [58]	88%

Table 5.2: Accuracy results of various models on SICK-E test set.

$q(x)$ where the first is the true distribution and the latter is the distribution being estimated, cross-entropy can be calculated as

$$H(p, q) = -\sum_{\forall x} p(x) \log(q(x)). \tag{5.1}$$

For N number of classes where $\boldsymbol{y}$ is the true labels and $f(x; \boldsymbol{\theta})$ is the function being estimated with parameters $\boldsymbol{\theta}$, the cross-entropy loss is then given by

$$J(\boldsymbol{\theta}) = -\frac{1}{N}\left(\sum_{i=1}^{N} \boldsymbol{y_i} \cdot \log(f(\boldsymbol{x}_i; \boldsymbol{\theta}))\right). \tag{5.2}$$

We present our test results in table 5.2 where accuracy is used as the metric and the results are per-cent based. We also include various results from the literature. As can clearly be seen, the addition of structural information drastically improves the performance. By this addition, the unified model achieves $\approx 13\%$ increase in accuracy. Also the unification model SEMN + STRN converges much more faster, as can be seen on the validation accuracy plot in figure 5.1. Thus, it would not be out of place to say that, when it comes to semantic entailment recognition, structural properties of the given text plays a role. This experiment clearly demonstrates that if more attention is also paid to the structural similarity along with the semantic, there exists an incontrovertible possibility for models to perform better.

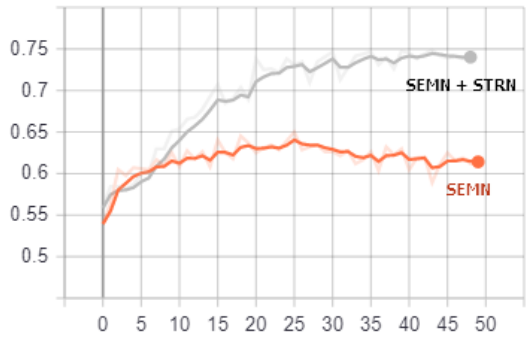

Figure 5.1: Validation accuracy (left) vs. epoch (bottom) graph for the SEMN and SEMN+STRN models on the SICK-E test set.

MSR		
Sentence A	**P.phrase?**	**Sentence B**
Amrozi accused his brother, whom he called "the witness", of deliberately distorting his evidence.	TRUE	Referring to him as only "the witness", Amrozi accused his brother of deliberately distorting his evidence.
Around 0335 GMT, Tab shares were up 19 cents, or 4.4%, at A$4.56, having earlier set a record high of A$4.57.	FALSE	Tab shares jumped 20 cents, or 4.6%, to set a record closing high at A$4.57.
But he added group performance would improve in the second half of the year and beyond.	TRUE	De Sole said in the results statement that group performance would improve in the second half of the year and beyond.

Table 5.3: Example sentence pairs and their corresponding labels from the MSR dataset.

5.2 Paraphrase Identification

Paraphrase identification (or detection) is also another challenging task existing in the NLP universe. This task basically involves distinguishing whether given two sentences are paraphrases of one another. A paraphrase of sentence is another sentence that convey the exact same meaning through using different words and word orders.

For this task, we use MSR dataset provided by Microsoft [2]. This dataset contains 5800 pairs of sentences extracted from news sources from the web. The labels in the dataset are human annotations indicating whether each pair captures a paraphrase (semantic) equivalence relationship. As the dataset comes already split, which approx. 4000 sentence pairs for training and approx. 1700 for testing, we do not apply random sampling and use the dataset as it is. For training our model, we again change the number of neurons in the scoring perceptron's last layer to two and employ softmax as activation. We use cross entropy as the loss function, and again utilize accuracy as the metric. We set the batch size to 120, and the number of epochs to 50. The results of this test for our models and for various other models from the literature can be seen on table 5.4. The unification this time does not really help the semantic similarity network, which is, when given a fair bit of thought, not so surprising, By definition, paraphrasing involves usage of different words/word phrases and word orders to convey the same meaning. Technically, in general, it is only natural that the addition of structural information has little to no effect on the accuracy of the semantic model. There is approx 2% increase in the accuracy, however this is highly negligible. Therefore, it could be concluded that for paraphrase

[2]https://www.microsoft.com/en-us/download/details.aspx?id=52398

Classification accuracy, by model name	
XLNet-Large [63]	90.7%
MT-DNN-ensemble [44]	90.3%
Snorkel MeTaL(ensemble) [50]	88.5%
GenSen [58]	78.6%
SEMN (ours)	63.8%
SEMN+STRN (ours)	65.8%
InferSent [15]	76.2%

Table 5.4: Accuracy results of various models on MSR test set.

detection, structural similarity of words in given two sentences has no effect on the outcome, at least based on the test we run.

5.3 Semantic Textual Similarity Detection

Semantic textual similarity is another interesting task for natural language processing jobs. As can be understood from the title, it involves producing a semantic relatedness score between given two sentences. The organization who is the pioneering task producer for this topic is the International Workshop on Semantic Evaluation. Each year, this workshop puts out a series of tasks gathered under the name SemEval [3] that are used as baseline performance evaluation tools. We use the collection of datasets provided by SemEval over the years for semantic textual similarity detection.

Our dataset for testing the performance of our models for the task of determining semantic similarity of two short texts/sentences has approx. 5000 traning and approx. 2000 test samples. Example data from the dataset can be seen on table 5.5. The dataset is a mixture of collection of sentence pairs from image and video captions, news articles, headlines and forum entries. In the dataset, basically, each two sentence pair is given a score $\in [0, 5]$ that reflects their semantic similarity, obtained via human annotation. For training our networks, we first squeeze and map these scores to a value between $[0, 1]$, and reduce the number of neurons in the last layer of scoring perceptron to a single one. Since the output we expect is now within the range of zero and one, we change the activation to sigmoid on this neuron. Following the convention of the literature, we now also change our metric to Pearson Correlation Coefficient, also known as Pearson's R and defined by

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}} \tag{5.3}$$

where x_i is the ith sample and y_i is the corresponding label. In our case, label is a scalar $\in [0, 1]$. The values $\bar{x}$ and $\bar{y}$ denote the means of the samples respectively

[3] http://alt.qcri.org/semeval/

SemEval STS		
Sentence A	**Score**	**Sentence B**
A man is playing a guitar.	1.714	A man is playing a trumpet.
A man is playing a guitar.	1.714	A man is playing a trumpet.
A man is cutting an onion.	5.000	A man cuts an onion.
A man is cycling.	0.600	A man is talking.
A man is slicing open a fish.	4.400	A man is cutting up a fish.
A man is slicing a tomato.	2.000	A man is slicing a bun.
A man is playing a guitar.	1.800	A man is playing a keyboard.
A baby panda goes down a slide.	4.400	A panda slides down a slide.
A man is singing and playing a guitar.	3.600	A man is playing a guitar.

Table 5.5: Example sentence pairs and their corresponding labels from the SemEval test set.

Classification accuracy, by model name	
fastText (vectors only) [37]	65.3%
Word2vec skipgram [47]	70.1%
Doc2Vec [10]	61.6%
LSTM [59]	75.0%
BiLSTM [59]	76.0%
SEMN (ours)	64.0%
SEMN+STRN (ours)	74.1%
InferSent [15]	80.1%

Table 5.6: Accuracy results of various models on STS test set.

for total number of n samples. The results of the test can be seen on table 5.6. The addition of structural similarity information adds a 10% accuracy to the outcome of the semantic similarity network. Based on this results, it would be okay to say that the inclusion of structural similarity while determining the semantic similarity of two sentences improves the overall outcome. It is not surprising, as can be seen from the dataset, that the sentences bearing a high semantic similarity score often contain more than a few words that exactly match not only character-wise, but also position-wise. Therefore, we suggest that leveraging this effective heuristic should be included in model designs.

Chapter 6

Conclusion

In this paper, we proposed a novel method to incorporate structural similarity analysis to neural networks. In order to achieve this, we present a relational encoding scheme, based solely on which we also develop a novel string similarity metric formula. We then use this metric as part of a dimensionality reduction approach through which we are able to teach a neural network to execute structural similarity analysis in a language invariant fashion. In the composition of structural similarity network, we make use of autoencoders, convolutional neural network and multilayer perceptron architectures.

We also propose a lightweight neural network architecture for semantic similarity analysis. This network utilizes 1-dimensional and 2-dimensional convolutions along with 2 stacked bidirectional LSTMs. This network was not expected to output state-of-the art results, rather it is a gauging tool to showcase the power of including structural similarity analysis along with semantic. We achieve this via unifying these both architectures.

The test results clearly suggest that the addition of structural similarity definitely aids the semantic similarity analysis. Especially for semantic textual entailment recognition and semantic textual similarity detection, the 13% and 10% respective increments in test accuracy validate this premise. Until now, structural similarity has been a hand-crafted/engineered feature. However, with our novel method, the possibility of incorporating this to a neural network such that it would no longer be needed to be hand-crafted is shown. Which, in turn, based on the increment in accuracies, demonstrates that such inclusion should always be considered to be included when designing a model for semantic similarity analysis, if one wants to achieve enhanced results.

References

[1] A. Anandkumar and R. Ge. “Efficient approaches for escaping higher order saddle points in non-convex optimization”. In: *Conference on Learning Theory* (2016), pp. 81–102 (cit. on p. 19).

[2] D. Bahdanau, K. Cho, and Y. Bengio. “Neural machine translation by jointly learning to align and translate”. In: *arXiv preprint:1409.0473* (2014) (cit. on p. 17).

[3] P. Baldi and Y. Chauvin. “Neural networks for fingerprint recognition”. In: *Neural Computation* 5.3 (1993), pp. 402–418 (cit. on p. 5).

[4] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006 (cit. on pp. 10, 19, 20).

[5] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov. “Enriching word vectors with subword information”. In: *Transactions of the Association for Computational Linguistics* 5 (2017), pp. 135–146 (cit. on p. 23).

[6] S. R. Bowman, G. Angeli, C. Potts, and C. D. Manning. “A large annotated corpus for learning natural language inference”. In: *arXiv preprint arXiv:1508.05326* (2015) (cit. on p. 4).

[7] L. Braille. “Method of writing words, music, and plain songs by means of dots, for use by the blind and arranged for them”. In: *Paris, France: l’Institution Royale des Jeunes Aveugles* (1829) (cit. on p. 2).

[8] J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah. “Signature verification using a "siamese" time delay neural network”. In: *Advances in Neural Information Processing Systems*. 1994, pp. 737–744 (cit. on pp. 5, 24).

[9] T. Brychcin and L. Svoboda. “UWB at SemEval-2016 Task 1: Semantic Textual Similarity Using Lexical, Syntactic, and Semantic Information”. In: (cit. on p. 5).

[10] D. Cer, M. Diab, E. Agirre, I. Lopez-Gazpio, and L. Specia. “Semeval-2017 task 1: Semantic textual similarity-multilingual and cross-lingual focused evaluation”. In: *arXiv preprint arXiv:1708.00055* (2017) (cit. on p. 42).

[11] H. Chen, F. X. Han, D. Niu, D. Liu, K. Lai, C. Wu, and Y. Xu. "Mix: Multi-channel information crossing for text matching". In: *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM. 2018, pp. 110–119 (cit. on p. 3).

[12] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. "Learning phrase representations using RNN encoder-decoder for statistical machine translation". In: *arXiv e-prints:1406.1078* (2014) (cit. on p. 17).

[13] S. S. Choi, S. H. Cha, and C. C. Tappert. "A survey of binary similarity and distance measures". In: *Journal of Systemics, Cybernetics and Informatics* 8 (1 2010), pp. 43–48 (cit. on p. 7).

[14] W.W. Cohen, P. Ravikumar, and S.E. Fienberg. "A comparison of string distance metrics for name-matching tasks". In: *IIWeb* (2003), pp. 73–78 (cit. on p. 7).

[15] A. Conneau, D. Kiela, H. Schwenk, L. Barrault, and A. Bordes. "Supervised learning of universal sentence representations from natural language inference data". In: *CoRR* abs/1705.02364 (2017) (cit. on pp. 39, 41, 42).

[16] G. Cybenko. "Approximations by superpositions of sigmoidal functions". In: *Mathematics of Control, Signals, and Systems* 2 (4 1989), pp. 303–314 (cit. on p. 10).

[17] Z. Dai, C. Xiong, J. Callan, and Z. Liu. "Convolutional neural networks for soft-matching n-grams in ad-hoc search". In: *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*. ACM. 2018, pp. 126–134 (cit. on p. 4).

[18] C. Debao. "Degree of approximation by superpositions of a sigmoidal function". In: *Approximation Theory and its Applications* 9 (3 1993), pp. 17–28 (cit. on p. 10).

[19] P. H. Duong, H. T. Nguyen, H. N. Duong, K. Ngo, and D. Ngo. "A hybrid approach to paraphrase detection". In: *2018 5th NAFOSTED Conference on Information and Computer Science (NICS)*. IEEE. 2018, pp. 366–371 (cit. on p. 4).

[20] J. L. Elman. "Finding structure in time". In: *Cognitive Science* 14 (2 1990), pp. 179–211 (cit. on p. 15).

[21] X. Glorot and Y. Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics* (2010), pp. 249–256 (cit. on p. 31).

[22] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016 (cit. on pp. 10, 12–15, 19, 20).

[23] H. Gouk, B. Pfahringer, and M. J. Cree. “Learning distance metrics for multi-label classification”. In: *8th Asian Conference on Machine Learning*. Vol. 63. 2016, pp. 318–333 (cit. on p. 6).

[24] A. Graves and N. Jaitly. “Towards end-to-end speech recognition with recurrent neural networks”. In: *International Conference on Machine Learning* (2014), pp. 1764–1772 (cit. on p. 17).

[25] A. Graves and J. Schmidhuber. “Offline handwriting recognition with multi-dimensional recurrent neural networks”. In: *Advances in Neural Information Processing Systems* (2009), pp. 545–552 (cit. on p. 17).

[26] J. Guo, Y. Fan, Q. Ai, and W. B. Croft. “A deep relevance matching model for ad-hoc retrieval”. In: *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. ACM. 2016, pp. 55–64 (cit. on p. 4).

[27] R. W. Hamming. “Error detecting and error correcting codes”. In: *The Bell System Technical Journal* 29 (2 1950), pp. 147–160 (cit. on p. 8).

[28] S. Hochreiter. “The vanishing gradient problem during learning recurrent neural nets and problem solutions.” In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6 (2 1998), pp. 107–116 (cit. on p. 16).

[29] S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber. *Gradient flow in recurrent nets: the difficulty of learning long-term dependencies*. 2001 (cit. on p. 16).

[30] S. Hochreiter and J. Schmidhuber. “Long short-term memory”. In: *Neural Computation* (1997), pp. 1735–1780 (cit. on p. 17).

[31] L. Hogben. *Handbook of Linear Algebra (Discrete Mathematics and Its Applications)*. Boca Raton: Chapman and Hall/CRC, 2006. Chap. 31.1 (cit. on p. 24).

[32] K. Hornik. “Approximation capabilities of multilayer feedforward networks”. In: *Neural Networks* 4 (2 1991), pp. 251–257 (cit. on p. 10).

[33] B. Hu, Z. Lu, H. Li, and Q. Chen. “Convolutional neural network architectures for matching natural language sentences”. In: *Advances in Neural Information Processing Systems*. 2014, pp. 2042–2050 (cit. on p. 3).

[34] P. Huang, X. He, J. Gao, L. Deng, A. Acero, and L. Heck. “Learning deep structured semantic models for web search using clickthrough data”. In: *Proceedings of the 22nd ACM International Conference on Information and Knowledge Management*. ACM. 2013, pp. 2333–2338 (cit. on p. 5).

[35] P. Jaccard. “Étude comparative de la distribution florale dans une portion des alpes et des jura”. In: *Bull Soc Vaudoise Sci Nat* 37 (1901) (cit. on p. 7).

[36] M. A. Jaro. “Advances in record linkage methodology as applied to the 1985 census of tampa florida”. In: *Journal of the American Statistical Association* 84 (406 1989), pp. 414–420 (cit. on p. 8).

[37] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov. “Bag of tricks for efficient text classification”. In: *CoRR* abs/1607.01759 (2016) (cit. on p. 42).

[38] Y. Kim. “Convolutional neural networks for sentence classification”. In: *arXiv preprint arXiv:1408.5882* (2014) (cit. on p. 3).

[39] D. P. Kingma and J. Ba. “Adam: A method for stochastic optimization”. In: *arXiv e-prints:1406.1078* (2014) (cit. on pp. 20, 28, 31).

[40] A. Krizhevsky, I. Sutskever, and G. E. Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in Neural Information Processing Systems* (2012), pp. 1097–1105 (cit. on p. 13).

[41] Y. LeCuna, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. “Backpropagation applied to handwritten zip code recognition”. In: *Neural Computation* 1 (4 1989), pp. 541–551 (cit. on pp. 13, 15).

[42] V. I. Levenshtein. “Binary codes capable of correcting deletions, insertions, and reversals”. In: *Soviet Physics Doklady* 10 (8 1966), pp. 707–710 (cit. on p. 7).

[43] P. Liu, X. Qiu, J. Chen, and X. Huang. “Deep fusion LSTMs for text semantic matching”. In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2016, pp. 1034–1043 (cit. on p. 4).

[44] X. Liu, Pengcheng He, Weizhu C., and J. Gao. “Improving multi-task deep neural networks via knowledge distillation for natural language understanding”. In: *arXiv preprint arXiv:1904.09482* (2019) (cit. on p. 41).

[45] N. Maharjan, R. Banjade, D. Gautam, L. J. Tamang, and V. Rus. “DT_Team at SemEval-2017 Task 1: Semantic similarity using alignments, sentence-level embeddings and gaussian mixture model output”. In: *Proceedings of the 11th International Workshop on Semantic Evaluation (SemEval-2017)*. 2017, pp. 120–124 (cit. on p. 1).

[46] W. S. McCulloch and W. H. Pitts. “A logical calculus of the ideas immanent in nervous activity”. In: *The Bulletin of Mathematical Biophysics* 5 (4 1954), pp. 115–133 (cit. on p. 9).

[47] T. Mikolov, K. Chen, G. Corrado, and J. Dean. “Efficient estimation of word representations in vector space”. In: *arXiv preprint arXiv:1301.3781* (2013) (cit. on pp. 2, 23, 42).

[48] B. Mitra, F. Diaz, and N. Craswell. “Learning to match using local and distributed representations of text for web search”. In: *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee. 2017, pp. 1291–1299 (cit. on p. 5).

[49] D. Prijatelj, J. Kalita, and J. Ventura. “Neural networks for semantic textual similarity”. In: *Proceedings of the 14th International Conference on Natural Language Processing (ICON-2017)*. 2017, pp. 456–465 (cit. on p. 4).

[50] A.r Ratner, B. Hancock, J. Dunnmon, F.c Sala, S. Pandey, and C. Ré. "Training complex models with multi-task weak supervision". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33. 2019, pp. 4763–4771 (cit. on p. 41).

[51] T. Rocktäschel, E. Grefenstette, K. M. Hermann, T. Kočisky, and P. Blunsom. "Reasoning about entailment with neural attention". In: *arXiv preprint arXiv:1509.06664* (2015) (cit. on p. 4).

[52] F. Rosenblatt. "The perceptron: A probabilistic model for information storage and organization in the brain." In: *Psychological Review* (1958), pp. 386–408 (cit. on p. 9).

[53] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. "Learning internal representations by error propagation". In: *Parallel Distributed Processing* 1 (1986) (cit. on pp. 12, 20).

[54] Y. Shen, X. He, J. Gao, L. Deng, and G. Mesnil. "Learning semantic representations using convolutional neural networks for web search". In: *Proceedings of the 23rd International Conference on World Wide Web*. ACM. 2014, pp. 373–374 (cit. on p. 5).

[55] R. Socher, E. H Huang, J. Pennin, C. D. Manning, and A. Y. Ng. "Dynamic pooling and unfolding recursive autoencoders for paraphrase detection". In: *Advances in Neural Information Processing Systems*. 2011, pp. 801–809 (cit. on p. 3).

[56] R. Socher, B. Huval, C. D. Manning, and A. Y. Ng. "Semantic compositionality through recursive matrix-vector spaces". In: *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*. Association for Computational Linguistics. 2012, pp. 1201–1211 (cit. on p. 39).

[57] M. Song, Q. Liu, and E. Haihong. "Deep hierarchical attention networks for text matching in information retrieval". In: *2018 International Conference on Information Systems and Computer Aided Education (ICISCAE)*. IEEE. 2018, pp. 476–481 (cit. on p. 5).

[58] S. Subramanian, A. Trischler, Y. Bengio, and C. J. Pal. "Learning general purpose distributed sentence representations via large scale multi-task learning". In: *CoRR* abs/1804.00079 (2018) (cit. on pp. 39, 41).

[59] K. S. Tai, R. Socher, and C. D. Manning. "Improved semantic representations from tree-structured long short-term memory networks". In: *arXiv preprint arXiv:1503.00075* (2015) (cit. on pp. 39, 42).

[60] S. Wan, Y. Lan, J. Guo, J. Xu, L. Pang, and X. Cheng. "A deep architecture for semantic matching with multiple positional sentence representations". In: *Thirtieth AAAI Conference on Artificial Intelligence*. 2016 (cit. on p. 4).

[61] W. E. Winkler. “String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage”. In: *Proceedings of the Section on Survey Research Methods* (1990), pp. 354–359 (cit. on p. 8).

[62] Y. Wu, W. Wu, C. Xu, and Z. Li. “Knowledge enhanced hybrid neural network for text matching”. In: *Thirty-Second AAAI Conference on Artificial Intelligence*. 2018 (cit. on p. 4).

[63] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. Salakhutdinov, and Q. V. Le. “XLNet: Generalized autoregressive pretraining for language understanding”. In: *arXiv preprint arXiv:1906.08237* (2019) (cit. on p. 41).

[64] L. Yao, Z. Pan, and H. Ning. “Unlabeled short text similarity with LSTM encoder”. In: *IEEE Access* 7 (2018), pp. 3430–3437 (cit. on p. 4).

[65] W. Yih, K. Toutanova, J. C. Platt, and C. Meek. “Learning discriminative projections for text similarity measures”. In: *Proceedings of the Fifteenth Conference on Computational Natural Language Learning*. Association for Computational Linguistics. 2011, pp. 247–256 (cit. on p. 5).