Natural Language Inference with Recurrent Neural Networks

David Torrejon Moya

Thesis submitted for the degree of
Master of Science in Artificial
Intelligence, option Engineering and
Computer Science

Thesis supervisor:
Prof. dr. Marie-Francine Moens

Assessors:
Prof. dr. Ir. Hendrik Blockeel,
Ir. Geert Heyman

Mentor:
Ir. Geert Heyman

Academic year 2015 – 2016

Preface

I would like to thank Geert Heyman for his infinite patience over all my questions and for answering emails even when he is on vacation. Also thanks, because he told me about this research topic at beginning of the year. Thanks to professor Marie-Francine Moens, for her tips and insights. I would also like to thank professors David Vernet and Francesc Alias, from La Salle URL, to help me get into KU Leuven, and encouraging me to keep working and developing my own ideas. I would like to thank all the exceptional friends that I have met along this year, the coffees and beers drank to keep us all sane are priceless. To Victor Baiges, for being a long standing friend, and always challenging me to go one step further. Also and more importantly, he knows the good restaurants in Barcelona. To my family, for helping and supporting me, even when I do not deserve it. And also because they pretend to understand something about neural nets, but they don't. And finally, to Ari. There is no way I would have gotten here without her. Thanks

David Torrejon Moya

Contents

Abstract

Natural Language Inference is a key part of Natural Language Understanding. NLI task is based on determining if an hypothesis can be derived from a premise. In the past year a new approach using Recurrent Neural Networks was very successful in solving the task. In the same work, a dataset was introduced with pairs of sentences labeled as entailment, contradiction or neutral to train NLI models. However there is a lack of understanding on why the model can't learn better, and is still far behind human accuracy in the task. The work done in this thesis explores the errors and provides insight on why we are still not fully able to achieve human accuracy on the task. It is shown that the recently proposed neural network architectures have more problems with sentences containing ambiguous words than humans. When working with sentences that have easily to spot distinct meanings, our models perform much better. The thesis also proposes a new method of combining the sentence representation of premise and hypothesis based on vector multiplication to achieve better accuracy. This method is based on element-wise multiplication to force the model to better understand whether two sentences contain the same meaning or not, helping the classifier to decide if there is an entailment relationship between the two sentences.

List of Figures and Tables

List of Figures

List of Tables

List of Abbreviations and Symbols

Abbreviations

AI	Artificial Intelligence
BoW	Bag-of-Words
BP	Backpropagation
BPTT	Backpropagation through time
CNN	Convolutional Neural Network
GPU	Graphics Processing Unit
IR	Information Retrieval
QA	Question Answering System
LM	Language Model
LSTM	Long Short Term Memory Network
MLP	Multi-layer perceptron
NLI	Natural Language Inference
NLP	Natural Language Processing
NLU	Natural Language Understanding
NMT	Neural Machine Translation
RNN	Recurrent Neural Network
RTE	Recognizing Textual Entailment
SQL	Structured Query Language
SNLI	Stanford Natural Language Inference

Symbols

σ	Sigmoid function
$\otimes$	Element-wise vector multiplication

Chapter 1

Introduction

Inference is the process to draw conclusions from premises we accept to be true. It has been a key topic in Artificial Intelligence (AI) research, and has taken many forms, logic inference, using logical rules to derive conclusions such $A \rightarrow B$, where B is our hypothesis to be tested, and A our premise that we take as true, or statistical inference, where we try to acquire knowledge about certain data population that has some uncertainty in it. Lately a new approach has been successful at predicting text entailment. This approach is the main focus of this work. The approach is based in Neural Networks, specifically Recurrent Neural Networks (RNN). RNNs allow us to model variable length sequences in an elegant way, which is precisely what we need to do for the task. Also they provide with two advantages, they do not require domain specific rules and they are more broadly applicable than the previous approaches, but this comes at a cost. The cost comes in the form of our data size. These models require much more data than any other previous models where some hand-crafted features could be derived from a few examples. These models learn directly from raw data, or representations of the raw data, rather than manually extracted features.

In this introductory chapter, I will try to express how important and complex Natural Language Inference is. In this chapter I will also contextualize the work done, expose the goals of the thesis and a brief presentation of the following chapters.

1.1 Natural Language Inference

Solving Natural Language Inference (NLI) is specially interesting in the context that we live in, with huge amounts of data flowing, and the needs of correct answers to any query we use against a database. We are moving to a world where retrieving information is less and less and expert thing (an SQL expert for example), and more and more someone at a department of a company needing some information. Old models of Question Answering for example, expected a question A, and only A, to answer B. But if a question meant the same as A, but was not asked as the model was expecting, the question would fail. Shall we ask the person to rewrite the question? What if he asks for "cakes" and we only have "pies" in the database? Shall we not provide any answer despite of both cakes and pies being synonyms. NLI steps in this direction, allowing to extract *meaning* from sentences, and checking whether another sentence in the same language is plausible from the original sentence.

- SELECT * FROM Food INNER JOIN Chef
  ON Chef.id_bakery = Food.id_bakery
  WHERE Food.Cake=1 AND Food.Cheese = 1 AND Chef.Location = 'Leuven'
  SQL
- I want to know where I can get a cheese cake and the chef is from leuven
  Natural Language

Obviously the previous example is an exaggeration, nowadays in general search engines you can search for virtually everything using plain English for example. The previous example pretends to illustrate how important is to deal with Natural Language, and how more easy would be to extract knowledge if everyone could participate of the process. The above example also illustrates one of the biggest challenges for NLP. PROLOG, a logic programming language used in classical approaches, is very useful when we want to explain certain closed domains, but it is not very good at generalizing or making assumptions. And, of course, when we use natural language we are making assumptions all the time (inferring). Apart from that, it and contains many features, it contains uncertainty, ambiguity, figures of speech, irony, and many others that make natural language a complex task to deal with. To link it with the *SQL* example before, it is also noticeable that when to a general search engine you use some exact words, the search results are much more accurate, but it does not perform well enough if we do not know the exact thing that we are looking for. Obviously there is much more work to do in querying via Natural Language (IR), and NLI works in the way of improving this task, and many others.

The specific case of NLI is to infer whether a human could accept a hypothesis h drawn from the premise p, repeating a structure similar to $p \rightarrow h$. Whereas logic inference has a process to detect whether $p \rightarrow h$, called the transformation rules, in NLI the structure might be similar, but the path to the goal is very different. Although expressing NLI as a process of inferring h from p might seem easy to see it, is once we explore natural language that we see is not. In logic, the first step is always to convert natural language to logic, to a formal language like first-order logic with no ambiguities. To provide the reader with an example of how complex the problem is, allow me to introduce a pair of sentences, extracted from the data set, without providing a proper solution.

- Two boys, one wearing green, the other in yellow, are playing with sidewalk chalk, drawing pictures on the ground. *premise*
- Two boys are drawing pictures with pink chalk. *hypothesis*

The annotators for this particular case, out of five annotators, three said neutral, and two said entailment.

I hope the example given above illustrates the complexity of the problem. If the previous example does not seem complex enough, allow me to propose another example found in the test set as an extreme case, that will help illustrate further more the complexity of the task.

- A man is painting a picture outside behind a crowd. *premise*
- A man is punching a picture to show great anger and rage outside to people. *hypothesis*
- contradiction neutral entailment contradiction neutral *Manual Labels*

In this example, no ground truth was set, meaning that there is no human agreement between the five human annotators that worked on this sample, making this particular case, extremely difficult to classify. And this is not the only one, just in the test set that I used, there were 112 cases similar to the presented above.

NLI can be used in a nearly unlimited amount of topics relating NLP, such as information retrieval (IR), or Question Answering Systems (QA). NLI is specially interesting if we can make us of automatic summarizing tools, in which from a text we can extract the key points, and then use these key points to perform any kind of task. At the same time, automatic summarizing tools could be optimized to write the key points in a closer way than we humans do, accept analogies, synonyms, or any other kind of semantic relation that can be captured.

1.2 Goals and Context of the Thesis

In 2009, all approaches to NLI known, were not related to Deep Learning techniques. In fact all approaches were based in hand crafted features and logic models. It is late 2015 that Bowman et al. [2] come with a new idea for NLI applying deep learning techniques like RNN and also word embeddings. Since then, during this past year, a few other approaches have appeared. Some similar than the one cited before, some others bringing different models of RNNs like Bidirectional LSTMs [12], and some others making use of what is called Attention models [21]. Models using Attention Mechanisms have been found to be particularly successful to the task at hand.

All the above mentioned papers have a common approach, on improving the results for task. All of the previous approaches try to improve the network model, rather than understanding if there is something lacking in the way we represent words. We have accepted word embeddings as the base representation model, but it is not easy to find discussion around the model itself. It is because of this research, that I've also begin wondering if word embeddings are really that good, and part of the thesis work, aims at the study of this way of encapsulate word information. As of today, I'm not aware of any study about meaning and word embeddings, a topic that is particularly interesting in the quest of not only helping to solve NLI, but Natural Language Understanding.

The original goal of this master thesis began reproducing the results from [2]. After doing so, the next goal was to improve the baseline model presented by the previously cited paper. But as explained above, there is a lack of understanding of the errors and limitations that appear in these systems, thus missing some possible improvements that are at a different level than the Neural Network model. We don't know how to incorporate common sense in our models, forcing each model to learn independently from another model. This is a huge difference in how us, humans, deal with Natural Language because we are able to allocate knowledge gained in one task to another one, helping to solve the second much easily than if we had to start all over. From the latest papers released on NLI, we are trying to solve inference without questioning the way we represent words and meaning, and only improving on the Neural models. This thesis work starts from one of the best performing models at dealing with inference, and provides insights on the results, trying to explore if they way we encode words affects in the task. It also shows how can this model be improved by abusing the dissimilarity relationship that NLI has inherent.

1.3 The Thesis

The thesis contains 4 chapters, with conclusions on each chapter, and in the end a fifth chapter, being the overall conclusions of this thesis.

To begin the thesis, Chapter 2, the reader will be provided with all the relevant background necessary to understand the work, and some insight on the reasoning on why Natural Language Inference has seen interest boosted recently. Chapter 2, more specifically contains information on Distributed representations and word embeddings, Recurrent Neural Networks (RNN) and its improvement Long-Short Term Memory networks (LSTM). A brief comment on deep learning is an absolute must, since it introduces both concepts mentioned before.

After presenting all the necessary background information, in Chapter 3 I will present the previous datasets and previous approaches to solve NLI. It is important that the reader gets the idea of what has been done before, specially because NLI has shifted completely, like many other tasks, from a paradigm where the features were hand crafted (acting as prior knowledge), to a representation learning paradigm, where in contrast for the hand crafted data, what it is expected is to have lots of data, so the network lets its own representations.

Unlike previous chapters, Chapter 4 is more a hands-on approach, detailing all the practical details of the work, allowing all the ideas and theory to remain in the first two chapters. This chapter details decisions involved in data prepossessing, and how I designed my model. It also introduces a few changes done to the original model. This modification of the model in [2] improves by 3% the results in the NLI task.

Chapter 5 presents the evaluation on the NLI task, using the SNLI corpus, recently developed to facilitate the use Neural Networks models. In this chapter I provide my results on building a neural network model similar to the one in [2], and some conclusions of my own, about why is so hard to achieve good results in the task. Before providing the reader with the results, I introduce the SNLI corpus, and also briefly mention the framework used to build the model, *Keras*.

The conclusions will give an overview of the thesis, as well as the general conclusions from the work and provide some future work guidelines that I think are interesting to explore.

Chapter 2

Word Embeddings and Deep Learning

In this chapter, several techniques will be explained, from what word embeddings are, what do they have to offer, and how to generate them to a brief overview on what is deep learning, and how it is related to NLI. Then we will dive into to RNNs, an elegant technique used to deal with sequences, being able to learn representations of these sequences (sentences for example), to a more complete view on LSTMs, which are a modification to standard RNNs, allowing them to deal with longer time spans.

2.1 Word Embeddings

In the following section we will explain an unsupervised learning technique that has helped several NLP tasks to beat state-of-the-art systems. This technique in conjunction with Deep Learning models such as RNNs have helped boost accuracies across the board in many NLP techniques.

Distributed Representations

Before diving into Word Embeddings itself, it is worth explaining the problem of modeling data using techniques of such as Bag of Words models. In many classification problems we could find vectors with ones and zeros describing features. In our case these techniques are not much useful. Imagine we need to describe cakes and, with the resulting descriptions, classify unseen cakes. A binary model to rate the attributes, a Bag of Words or, in our cake classifier, a Bag of Features model, we will list all our features and the length of this list is going to be the dimension of our vectors. Then we will describe with a '1' or a '0' whether the cake contains or uses this ingredient (feature). This can be seen in Table 2.1

As the reader can see, this leads to a nearly infinite possibility of combinations (curse of dimensionality), even in just our dataset, that may not represent all possible combinations. Thus making this kind of representation hard to deal with, and with a higher likelihood to cause overfit in the training data. Distributed Representations

Feature	Sugar	Cheese	Chocolate	...	Iogurt
Sugar and Cheese	1	1	0	0/1	0
Sugar and Chocolate	1	0	1	0/1	0

Table 2.1: Example of Bag of Features Representation

Vector Representation	dim1	dim2	dim3
Sugar and Cheese	0.3	0.84	-0.12
Sugar and Chocolate	0.4	0.81	-0.16
Iogurt and Mint	-0.9	-0.5	0.63

Table 2.2: Example of Distributed Representations

[8] solve this problem. Word embeddings are a type of distributed representations. This kind of representations maintain that each feature is meaning something in particular, but our concepts (words) are represented by combinations of features, and a feature can be in many concepts. There is a relation many-to-many between features and concepts. Generating this many-to-many relationship we can represent concepts (words), in spaces with lower dimensionality than when represented with BoW, having better generalization and less problems with overfitting. What we have then is a table like Table 2.2.

It can appreciated, we moved the possible infinite feature binary map, into the possible values of these new vectors, allowing us to learn and represent infinite words (or anything we want to represent) into a fixed dimension vector. As we can see, the sugar and cheese vector and the sugar and chocolate vector are quite similar, but they are very different from iogurt and mint. These representations may produce some errors because of having similar vectors. Although this problem is partially solved by adding more dimensions, it is never fully solved. In the case of word embeddings, as we will see in the following section, vectors representing "he" and "she" are similar, but "she" and "cake" are not. To end the section on distributed representations, it is worth commenting that distributed representations are fairly good approximations on how our brain works. This is why sometimes, when we want to retrieve a word from our own dictionary, we end up saying a close word that may not be the right one. And, as in the training process of word embeddings, if the word that we came out with and the expected word are not the same, we will update our mental model to make sure that the next time we want to use that word the wrong one will not be "as close as" the expected word.

Now that the reader is familiar with the distributed representation concept, we can tackle word embeddings.

As expressed in the previous section, word embeddings are nothing else than vectors representing words, and I briefly mentioned that vectors containing the words "he" and "she" are probably closer than "he" and "cake". Now we will try to explain the most important question on the topic, how can we get these vectors.

GloVe model

The GloVe model is one of the three popular word embeddings models [20], being the other two in the word2vec [17] framework. In the word2vec framework we can fin the Continuous Bag-of-Words(CBOW), and the Skip-gram models(SG). The GloVe model is the one that I will base my explanation on how to obtain word embeddings. I do so because the GloVe model is the one it's used in [2] to train their models, making it the most appropriate to work with since I am going to compare performances of different models, all of the with the same word embedding model.

GloVe model is based on word counting, rather than the word prediction models used in the word2vec framework, CBOW and SG. GloVe model word-embeddings come from a word co-occurrence matrix. A co-occurrence matrix computes how many times a word is found in the *context* of some other words.

$$X = \begin{bmatrix} 0 & 4 & 0 & 2 \\ 4 & 0 & 1 & 0 \\ 0 & 1 & 0 & 3 \\ 2 & 0 & 3 & 0 \end{bmatrix}$$

The structure of the word co-occurrence is always of the shape $(|V|, |V|)$, where $|V|$ is the length of our vocabulary (unique words). The context is a free parameter to be defined, and refers to how many words on each side we are considering to be co-occurring. This example of a word co-occurrence matrix already provides us with some interesting statistics, for example by examining X, we would already know that $word_1\ and\ word_2$ happen in the same context, meaning that they most likely have some relation between them, and $word_1\ and\ word_3$ never happen together, meaning that they have no relation. This co-occurrence matrix is then translated into a co-occurrence probability matrix, shifting from a word-count to a word-probability.

Where X_{ij} is the number of times word j occurs in the context of word i, and X_i be the sum over all times X_{ik}, being $\sum_{k=1}^{|V|} X_{ik}$. From this point forward, the corpus is no longer relevant, and we only make use of the co-occurrence probability matrix. To begin computing the probabilities we need to introduce another point made by the authors. GloVe model is built upon ratios of co-occurence probabilities, rather than the probabilities themselves [20].

$$F(w_i, w_j, \vec{w_k}) = P_{ik}/P_{jk} \tag{2.1}$$

Notice that the above equation contains three word vectors, being w vectors words, and $\vec{w}$ being context words. So at this point we already know that the model will generate for every word, a context word vector, that later we will decide what to do.

The equation below is a simplification of equation 2.3, where the term b_i is a bias term that comes from $\log X_i$, and the term b_j is added to keep the symmetry.

$$w_i{}^{\top} \vec{w_k} + b_i + \vec{b_k} = \log X_{ik} \tag{2.2}$$

In this equation we can see all the parameters of the model, from these we are interested in W and $\vec{W}$. The first one represents the vectors with the word embeddings

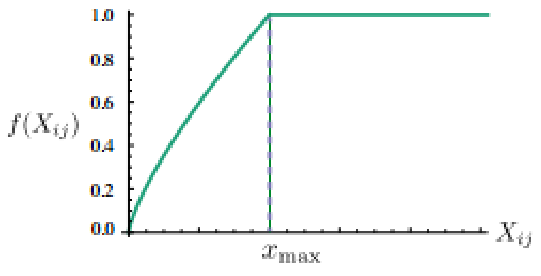

Figure 2.1: Weight function with $\alpha = 3/4$ from [20]

and the second one represents the context word vectors. Since there is a symmetry relation between W and $\vec{W}$, they are different vectors because of the random initialization of the parameters in them. The model itself carries one major drawback, which is that the it weights all co-occurrences equally, including those that barely happen, carrying noise with it. To deal with rare co-occurrences that barely happen, the GloVe model uses a least square regression model, with a weighthing function, equation 2.4. Figure 2.1 shows the behaviour of this function. This weighting function provides us with a couple of interesting points. It is a function that is always increasing, meaning that the more frequent co-occurrences have higher weights. If the function would decrease at some point, rare co-occurrences would have higher weight than others that happen more often and this is completely undesired. The second point is that in cases were the co-occurrence ratio is high, we maintain the weight at a maximum of 1, not overweighting frequent co-occurence terms. The α parameter needs to be tunned manually, and there is no theoretical background about the tuning of this parameter. In [20] they use $\alpha = 3/4$ and $x_{max} = 100$.

$$P_{ij} = P(j|i) = X_{ij}/Xi \tag{2.3}$$

$$f\left(X_{ij}\right) = \begin{cases} \left(\frac{X_{ij}}{x_{max}}\right)^{\alpha} & if X_{ij} < x_{max} \\ 1 & otherwise. \end{cases} \tag{2.4}$$

Using the above equation in our cost function, we finally get J

$$J = \sum_{i,j=1}^{|V|} f\left(X_{ij}\right)\left(w_i^T \vec{w}_j + b_i + \vec{b_j} - \log X_{ij}\right)^2 \tag{2.5}$$

As explained before W is the word embeddings parameters that we are looking for, however, as suggest in [20], what it is used is $W + \vec{W}$, which usually achieves a small boost in performance in the tasks that word embeddings are evaluated. To wrap the explanation we say that the GloVe model is a count-based model, that its goal is to perform a dimensionality reduction from the probabilities matrix to a lower dimension that captures most of the variance in the data.

2.2 Deep Learning

Deep Learning is the evolution of Artificial Neural Networks. It attempts to learn representations of the inputs in order to perform a task like classification or clustering. Through different layers, these networks learn meaningful representations of the inputs. The term deep is used because this different representations are learnt in different layers, making the network deep. The models that we will explore in this section, Recurrent Neural Networks, fall under the category of deep learning.

Deep Learning is not a novel technique, in fact dates from 1965 [9]. but is in the last few years where this technique has had a huge impact. Deep Learning has been able to shift the pattern recognition paradigm while improving its results drastically. Deep learning shifts from experts helping train the model, to directly train the models with no experts needed. To perform tasks in computer vision few years ago, as an example, we needed some expert that could explain which features are relevant for each task, and then train a model using these features. These models worked because of the human knowledge on the domain, and they were also useful because of the lack of proper datasets. Data nowadays is collected easier than it was few years ago, this has helped create larger datasets that are used in the training process of the new deep models.

The difference with traditional neural networks models, besides the depth of the models (hence the name of deep learning, referring to deep architectures) is that we can train this models without the need of any knowledge of the particular domain of the problem. We can conclude that there have been two major key factors in the deep learning auge, namely the access to more, and better data, and the access to better hardware, the use of GPUs, that can speed up the training process by large orders of magnitude. Deep Learning techniques are seen in all three learning paradigms, supervised, unsupervised and reinforcement learning. Recent success in challenging tasks like the game GO have increased the visibility of these models. In this thesis we will only explore supervised learning in the form of Recurrent Neural Networks, outside of word embeddings trained in an unsupervised fashion explained before.

Having introduced what deep learning is, and before going deep into the network architectures, techniques and problems they present, I must explain the link between Deep Learning, and Natural Language Understanding (NLU). The relation between AI and NLU, has always been via feature extraction, or via logic representations of the language. Models before the deep learning auge, were based in techniques such as Part-of-Speech tagging, or more importantly, Bag-of-Words (BoW) models only. Deep learning models, via word embeddings allow us to not use features based on the domain, but more generalized concepts. For example, while building the model for the thesis solution, I made use of pre-trained word embeddings, properly fine-tuning them for the specific task. These word embeddings were trained in a corpus not specific to the task, making them usable as a starting point for any task that requires them.

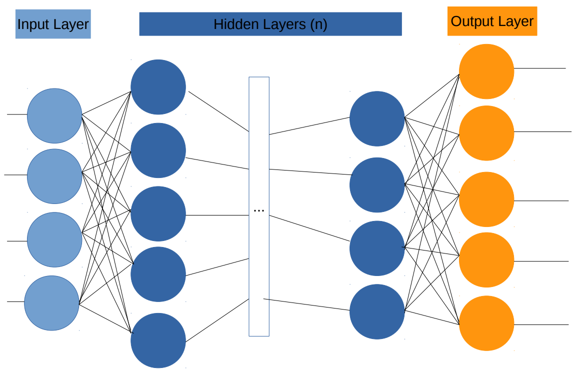

Figure 2.2: Classical Feedforward Network architecture

In the following sections we will discuss the main Network Architecture for supervised learning with sequenced data and its purposes, this type of network is called a Recurrent Neural Network(RNN).

2.2.1 Deep Neural Networks

In this section we will explain both, RNNs, and their modification LSTMs. Allow me to illustrate the reader with Figure 2.2, were we can observe the classical architecture of a feedforward network architecture. As any feedforward network, has three types of layers, the input and output, and then n-hidden layers. The objective remains the same as in classical approaches, train the network via backpropagation, to learn patterns and classify according to these patterns. In case of RNNs, backpropagation algorithm is slightly different than the original one in order to deal with time depth. Backpropagation through time is the name that the algorithm used to train RNNs recieves and will be explained in this chapter. After presenting RNNs, the *vanishing gradient* problem will be explained. A natural solution to this problem is the use of LSTM networks, that help in memorizing long term relationships.

There is another type of deep neural architecture called Convolutional Neural Networks (CNN). CNNs are widespread used to solve computer vision problems, and RNN are mostly used for language oriented problems. RNNs, as we will see, capture dependencies through time, making it a perfect tool for language modelling problems. It is worth mentioning that some recent publications have shown that CNNs could also work with language [10] but in this thesis we will not explore this option.

2.2.2 Recurrent Neural Networks

Before getting *deep* in the RNNs and their effectiveness, I must first say that RNNs, and specially their variation LSTMs, are used successfully in all types of sequential data problems, such as speech recognition or neural machine translation (NMT). For

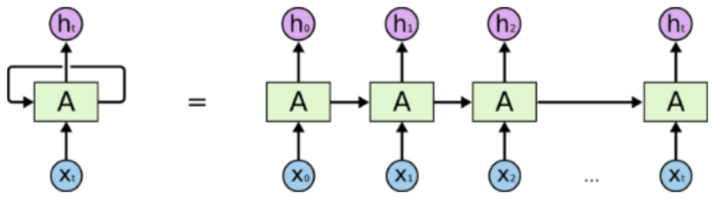

FIGURE 2.3: Rolled to unrolled RNN (figure from[18])

more information about the success of RNN/LSTM I suggest reading this post from Andrej Karpathy [11].

So far the reader has been introduced to the concept of neural networks its deep architectures, and how they can help solving pattern recognition problems. In many tasks that humans perform daily, there is one key aspect to be considered that so far we have ignored: **time**. Time is present in most of our problem solving, and time is present in many NLP tasks. When we are trying to compute the probabilities of a language model (LM), time is present, for example:

- There is a pie.
- A pie there is.

The concept of time in these sentences is implicit in the sense that before is, we have there, and before there, we can have pie or nothing, there is more detailed explanation in [6]. Classic architectures of neural networks do not deal with the time, because they have no memory, and that is far from how humans act. We know that after a word comes another one, and the fact that there was a word at a certain point in time, makes plausible that some other word occurs, and diminishes the possibility that some other word occurs. This problem is solved using RNN, which are networks that are capable of remembering what happened in the past, maintaining dependencies with its previous states.

Figure 2.3 represents the basic principles of RNNs. On the left side we have the rolled model for an RNN, in which like a classical architecture at a time t it has an input x and an output h. The main difference is this self loop, that does not exists in a feedforward network, and in an RNN establishes the "memory" part. Watching figure 2.3, it is quite intuitive that if we unroll the network we get the same network over time. We can see that at a time t, the state of the network is affected by its input x_t and its previous state at time *t-1*.

To explain in more detail how RNNs work, I will make use of figure 2.4. While training, RNNs still work in a feedforward net fashion, with two steps. First step is the forward pass, and second step is adjusting all the weights, making use of the backpropagation (BP) algorithm, with some modifications that we will go through soon, called *backpropagation through time (BPTT)*.

First lets give an overview of all the parameters that we have in Figure 2.4.

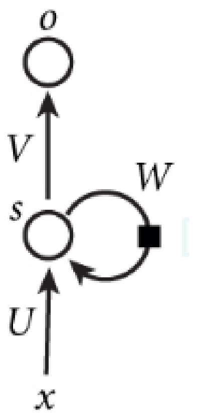

Figure 2.4: RNN with all the parameters (figure from [3])

- x: Input vector
- o: Output vector
- $U,\ W,\ V$: Parameters to learn
- s: network state *(memory)*

$x,\ o,\ s$ are all directly affected by time, so from now on we will refer to them as x_t, o_t, s_t

Forward Pass

The forward pass is quite similar to the one in any MLP, where we have input to hidden product, and product to output. But in this case, our hidden will be the memory state s, and will be affected by W.
All multiplications are matrix multiplications, if otherwise will be explained in each particular case.

To compute the memory state we will use:

$$s_t = f(Ux_t + Ws_{t-1}) \tag{2.6}$$

$$o_t = softmax(Vs_t) \tag{2.7}$$

Where f is the non-linearity function. Depending on the literature we can find either tanh or σ. The reader can appreciate that at $t = 0$, we already need to solve s_0, needing s_{-1}. In this case usually s_{-1} is initialized to all zeroes.

One key detail is that this type of network shares the weight matrices $U,\ W,\ V$. If we look again at Figure 2.3, we will see that A, in the formulas our s, is always the same. It is the memory of the network, and all the weight matrices are shared as we unroll the network through time, making it an elegant solution to represent sequences.

To put this into perspective for the reader, and make it closer to my NLI task, I will explain in terms of language modelling (LM) what this RNN is supposed to do. Our input vector x_t is what is called a one-hot encoding vector (1-N), meaning that we will have a vector of length $|V|$ with all zeros, and only one position at one, meaning that at time t the word that we have at our input is just the word with that position. If our vocabulary contains three words:

- Vocabulary: ['I', 'like', 'cakes']
- Indexes:
  - 1 = I
  - 2 = like
  - 3 = cakes
- The sentence "I like cakes" can be represented as [1, 2, 3], and the sentence "cakes I like" as [3, 1, 2].
- To represent represent "I like cakes" and be able to input the sentence in our network, we will create the following matrix: $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

This type of representation raises a concern, and begs to keep our vocabulary as small as possible. The output o_t is going to be a probability distribution at time t over our vocabulary, so it will output a vector of length $|V|$ and in each position the probability of the next word.

Computing the error

After the forward pass and obtaining our probability distribution on the next word, we need to compute the error, and check if our output word matches the expected word. Computing the error, and obviously, minimizing it, it is equivalent that computing the cost function, also known as objective or loss function, and minimizing it, meaning that a high value on this function means high error, and the lower the cost function, the lower our error.

$$J^t(\theta) = -\sum_{j=1}^{j=|V|} y_{t,j} \times \log(o_{t,j}) \tag{2.8}$$

Equation 2.8 is called cross-entropy, and is a classical loss function used on multiple classification problems. It compares two probability distributions, $y_{t,j}$, that is the correct labels, a one-hot encoding vector with the "1" being the correct answer and, $o_{t,j}$, being the output of our RNN, a probability distribution over all the words. The specific name for this cost function is categorical cross-entropy, and usually is used on top of a softmax function. Usually you find a softmax function as the activation function of the last layer of a neural network.

Softmax

Softmax is a function that flattens a vector of real values into a vector of values where the sum over those values is 1. Basically the softmax functions ends up providing a probability distribution for a vector, making the values of the vector more interpretable, ranging from 0 to 1.

$$O(t)_j = \frac{e^{o_j}}{\sum_{k=1}^{K} e^{o_k}} \tag{2.9}$$

What this equation does is quite simple. The denominator is the sum over all elements of the vector, and the numerator is the position j of the vector o (remember, o is our output vector), that will contain a real value number, and will be flattened into a number between $(0, 1)$.

Backpropagation through time (BPTT)

On this section I will briefly present backpropagation (the delta rule more specifically) to quickly explain BPTT. Backpropagation is the key algorithm for gradient descent (GD). Many modifications have been proposed for gradient descent algorithm, but we will stick with the vanilla one, explaining it with a small modification. Backpropagation algorithm is the derivative chain-rule applied to neural networks, and its goal is to compute the partial derivatives of our cost function (eq 2.8), but it can be any other cost function like mean squared error (MSE), with respect of all the parameters of our network, our weights and biases. I will introduce the formulas, and then move to BPTT.

$$\Delta w_k j(n) = \alpha \delta_j y_k + \eta \Delta w_k j(n-1) \tag{2.10}$$

The left part of the equation represents the increment on a weight going from a layer k to the layer j. The n represents the *epoch*, meaning that this wieght change affects the weights at the *epoch n*, and as we can see in the formula, in the second term of the sum, we make use of the Increment of the weight at the previous epoch. In the above formula, we have 2 parameters α and η, which are free parameters that range between $(0, 1]$ and $[0, 1)$ accordingly, being α the learning rate and η the momentum. The first affects on how big we want the changes on the weights to be, and the momentum term encourages us on making bigger changes when we are in the correct direction of our gradient. y_k represents the output of a neuron of the layer k The only term that is left of explaining is δ_j. The δ_j, from here comes the name, is what is called the *error term.* To better explain the computation of the δ term, we will imagine that our network only uses sigmoid σ activation functions. Remember that the derivative of the sigmoid is:

$$\frac{dy}{dx} = y(1-y) \tag{2.11}$$

Knowing this, we have that the computation of the error is 2.12 for the output layer, and 2.13 for all hidden layers. y_j represents as before, the target vector, and o_j, represents the real output of our network.

$$\delta j = (y_j - o_j)o_j(1 - o_j) \tag{2.12}$$

$$\delta j = (\sum_{i=1}^{I_j} \delta_j w_{ji})o_j(1 - o_j) \tag{2.13}$$

A more in depth explanation that I found useful in order to explain the back-propagation algortihm can be found in [15], where all the derivatives are given.

Now backpropagation through time has an significant change respect to the original algorithm although the rest remains similar. Our cost function remains also the same. As in usual backpropagation we have to compute the gradients with respect of V, W, U, seen in figure 2.4. For V we have:

$$\frac{dE_t}{dV} = \frac{dE_t}{do_t}\frac{do_t}{dV} = \frac{dE_t}{do_t}\frac{do_t}{dz_t}\frac{dz_t}{dV} \tag{2.14}$$

It is important to have figures 2.3 and 2.4 in mind. The weights between A in Figure 2.3, represent our W, our V are weights from A to h, and our U are the weights between x and A. Equations 2.14 z_t is 2.6 withtout the softmax. This part remains the same, compute the error in the output, and propagate it backwards. In the case of W and U, they share the same modification. The equation is fairly similar to the one introduced before.

$$\frac{dE_t}{dW} = \frac{dE_t}{do_t}\frac{do_t}{ds_t}\frac{ds_t}{dW} \tag{2.15}$$

The same chain rule is applied, but, instead of using dz_t we use ds_t which carries a huge difference from the previous equation. z_t depends solely on the output at a certain time step, but s_t does not. We have seen s_t before, in 2.6. At a certain time step, s depends on W, which is directly affected by s_{t-1}. This helps explaining the *depth* of the network, at each time step, we need to consider all previous time steps. The modification to compute the gradient is the following.

$$\frac{dE_t}{dW} = \sum_{i=0}^{t} \frac{dE_t}{do_t}\frac{do_t}{ds_t}\frac{ds_t}{ds_i}\frac{ds_i}{dW} \tag{2.16}$$

In the end we add all W modifications at each time step up until time step t. A good explanation with all the derivatives and operations can be found in [7]. The original paper can be found in [26]

Vanishing Gradient

Having presented BPTT, or BP for the case that matters, I will introduce the Vanishing Gradient problem. The problem it is not only found in Neural Networks,

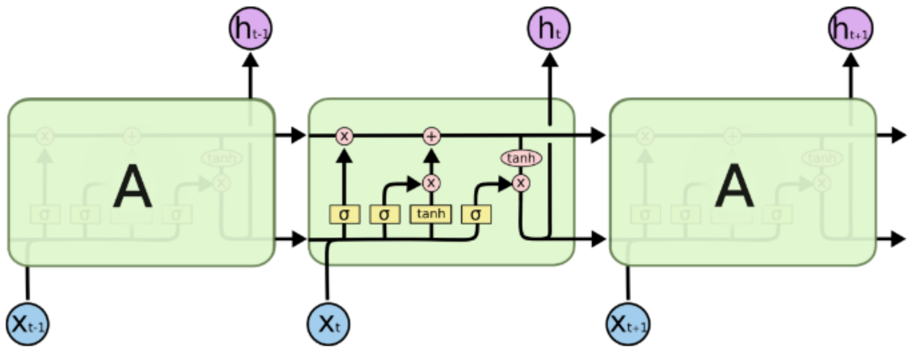

Figure 2.5: LSTM module (figure from[18])

but it is a problem inherent of training models via gradient methods. Remember that when training our networks via gradient descent, we end up doing so via the chain rule. This means that the gradient of the cost function is passed backwards in form of multiplication. Our gradient, is computed from the derivative of the cost function, usually a value between $(-1, 1)$ is we use tanh activation function. So when we are passing our gradient to our previous layers we are doing a multiplication. If the gradients are small, the more we keep passing them backwards, the smaller they will get, therefore affecting our learning rate in the early layers of our network. This problem is known as the vanishing gradient, because gradients get so small that they do not affect or affect so little in the early layers of our network. The opposite problem also exists, called *Exploding Gradients*, in which the gradients are very large. This problem is easily solved by clipping back the gradients into a predefined range. A deeper analysis on this topic and how it affects RNN training can be found in [19]

Next section we will explore LSTMs, a modification of the standard RNN that deals with the problem, allowing the network to memorize longer time sequences.

2.2.3 LSTMs

LSTMs follow the same principle as RNN, with units repeating over and over. And just like standard RNN, the cell state s at time t, is directly affected by the cell state s_{t-1} at time $t-1$. However as we can see in 2.5 the module inside differs quite a lot from Figure 2.4.

To establish a notation and to be accurate at the image, we will use h as our input, and x as our input. The cell state is going to be represented by our s as before. In 2.5 the s is represented by the black straight line that goes through the multiplication and the sum. To further simplify the explanation, we will explain from left to right what all the operations represent. The explanation given in this work is based in the work done by Christopher Olah, and can be found in [18].

The full idea of the cell state s, is to determine which information remains from the previous output, and which information we are going to store in it from the current input. This is done via the forget gate, and the input gate that later generates a vector candidate that can be added in the cell state coming from the previous time step. Finally, the output is decided by the output gate, that depends in the current state. Let's see the equations. The first equation introduces the forget gate, which decides which information between the previous output, and the current input stays.

$$f_t = \sigma(W_f(h_{t-1}, x_t) + b_f) \tag{2.17}$$

The result is counter intuitive, since it is called forget gate, we would expect that a result close to 1 would mean forget, instead a 1 represents do not forget. The next equation (2.18) represent the input gate, as we see both equations are equal, the only modifications are the matrices W that represent the weights to be learned and the b that reprent the biases.

$$i_t = \sigma(W_i(h_{t-1}, x_t) + b_i) \tag{2.18}$$

Equation 2.19 represents the generation of the candidate.

$$\widetilde{s_t} = \tanh(W_s(h_{t-1}, x_t) + b_s) \tag{2.19}$$

Right now, we already know which old information want to keep, which new information want to integrate, but still we haven't do it. Equation 2.20 does so. Note that $\otimes$ denotes element-wise multiplication.

$$s_t = f_t \otimes s_{t-1} + i_t \otimes \widetilde{s_t} \tag{2.20}$$

With this we concluded the cell state, but we still have not presented any output for the network. The output is going to be very similar than the operations presented before, in fact, the output gate is equal than the forget and input gate. Equations 2.21 and 2.22 represent the output gate, and the output.

$$o_t = \sigma(W_o(h_{t-1}, x_t) + b_o) \tag{2.21}$$

$$h_t = o_t \otimes \tanh(s_t) \tag{2.22}$$

LSTMs were first introduced in 1997 by Hochreiter and Schmidhuber. The original paper can be found in [22].

2.3 Conclusion

In this first chapter, we have introduced word embeddings and Recurrent Neural Networks, and their improvement Long-Short Term Memory networks. In reality, there is very few practical use of the RNNs, as we have seen they deal poorly with long time relations, instead LSTMs, deal better with this long-term relations.

We have seen how word embeddings are obtained, using unsupervised learning, going from a high dimensional probability matrix, to vectors of lower dimensionality. Word embeddings have amazing properties, they capture semantic and syntactic relations, as we have seen when we represent them in 2-dimensional plots.

In the deep learning part, we have explained the shift on the paradigm, going from hand-crafted features and small models, to loads of data, with no hand-crafted features, and huge models. This is possible through the backpropagation algorithm, and the computational power gained in the last years. RNNs have been presented, they are an elegant way to represent sequences (sentences in our case), and we have also seen their major drawback, the *vanishing gradient*, that prevents them from learning long-term relations. To solve the long-term relationship problem, we have introduced LSTMs, a modification of standard RNNs. Their design makes them highly valuable because they solve the long-term relationship problem, thus providing us with a method of encoding sequences with long time dependencies.

Chapter 3 will explain in more detail the NLI problem. It will present previous approaches to solve NLI, approaches that are not deep learning based. After this, current approaches based on word embeddings and LSTMs will be explained.

Chapter 3

Problem definition

During this chapter a more detailed explanation of the NLI task will be defined, aswell as presenting previous approaches to the problem. At the end of the chapter it will be introduced a technique that has seen a lot of success this past 6 months, called attention mechanisms.

3.1 The Problem

Natural language inference (NLI) is a part of a much broader topic, called Natural Language Understading(NLU). The core problem that tries to solve NLI is whether a sentence h entails, contradicts or it is neutral to another sentence p.

$$p \rightarrow h \neq h \rightarrow p$$

This also entails one key feature of the problem, which is that our hypothesis is derived without previous knowledge on the premise. This is present in our approach, and can be seen in [2], both the premise and the hypothesis are two separate sentence models, that are first met in the first layer of the classifier.

The problem that enables this thesis was set by [2], where given a pair of sentences and no extra information, a system could determine if these sentences were incurring in entailment, contradiction, or were neutral to each other. [2] making use of RNNs and word embeddings achieved a result of 77.8%, on a dataset build solely for the purpose of solving NLI.

3.2 Previous (and current) Approaches

In this section previous approaches will be presented first. Two different types of approaches will be introduced, none of them related to Neural Networks. After, a brief introduction on the current approaches based on Neural Networks will be presented.

3.2.1 Shallow Approaches

Shallow approaches like Bag-of-Words (BoW) models have been used since the early days to try to solve NLI. BoW models try to line up all the words in the premise with all the words from the hypothesis. These models have two main issues, that will be presented as examples, examples that we have seen before.

- David likes cakes & Cakes likes David.
- David does not like cakes & David likes cakes

In a BoW model, the first example will match all the words from sentence 1 to all the words in sentence 2, and will say that the inference is correct, that is entailment. In the second example, again a BoW model will match nearly all the words, not paying attention to the importance of the negation, and inferring again entailment. Although this can be partially solved, more precise approaches are needed.

3.2.2 Deep (Logic) Approaches

This type of approach try to extract all the semantic from the premise and hypothesis and try to match it via logic approaches, via formal representations of the sentences. This type of approach solves the 2 problems from the previous approach, but generates a much bigger problem. We are trying to solve a Natural Language problem, meaning that transforming sentences into Logic might be quite difficult, if not impossible. We need information of the domain. Many words in natural language have no direct translation into logic approaches. And the biggest of all the problems, natural language is full of ambiguity, while logic can't afford to have.

This part is based on [14], section 1.4 Previous approaches to NLI. The cited work is the doctoral dissertation from B. MacCartney. It contains detailed information on previous approaches (current in 2009), and how they perform in different setups.

3.2.3 Current Approaches

So far I've mentioned approaches before deep learning started tackling the job, and word embeddings took over NLU tasks.

NLI is no different than other NLU tasks, and word embeddings and Neural Network models are taking over. The very first approach using Neural Networks was [2]. In this paper, aswell as in this thesis, we use word embeddings to represent words, and LSTMs to encode sentences. We know that word embeddings have some nice properties while encoding words, and that LSTMs are the best resource available to model sequenced data, therefore it is appropriate to use both to tackle the task. In the next chapter, figure 4.1 will be used to show the model used to tackle the NLI problem.

Since [2] few other papers have come out with different approaches to solve the task, or improve the accuracy on it. Some of the new approaches, that improve over the original paper are centered in *Attention Mechanisms* like the one we can see in

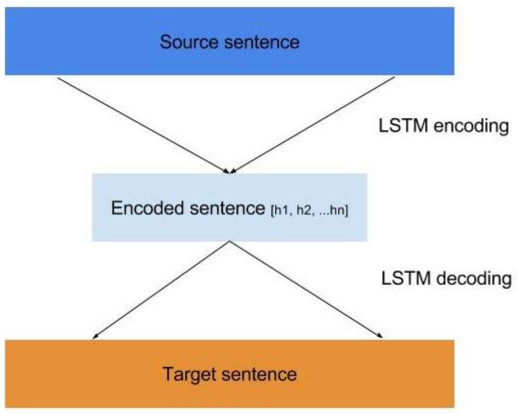

FIGURE 3.1: Neural Machine Translation classic model

[21]. Attention mechanisms are specially used in Neural Machine Translation (NMT), a task highly related to NLI.

Attention Mechanisms

NLI and NMT tasks work in a similar way, NMT tries to go from a source sentence to a target sentence, that the system is able to produce. In NLI, the question is more like, is our model likely to "generate" the target sentence from the source sentence? Traditionally, these tasks as we have seen in the previous section for NLI, relied on extracting features (BoW), or generating logic models. With RNNs, this tasks removed partially the need for feature extraction and became more of a vector encoding task. In NLI we have two vectors coming from RNN/LSTMs networks, that encapsulate the meaning of both sentences, and then a classifier on top of it to determine what relation these sentences have. In a NMT task we have a source sentence encoded in a vector, and then a decoder than using the encoded source vector, decodes the target vector, producing a sentence in another language (Figure 3.1). This presents a challenge, which is encoding the full meaning of sentence in a high dimensional vector, no matter if the sentence has 8 words, or 50, one high dimensional vector needs to encapsulate all the information. LSTMs solve the long term relationship, they can memorize events that happened some steps ago in time, but they can not memorize that much back in time. Attention mechanisms, aim at solving this problem, by shifting the paradigm of encoding a sentence into a vector, and working with only one vector, apart from generating some amazing insight on what our model is doing. Figure 3.2 shows an Attention Model for NTM [1]. The shift of paradigm is that, instead of encoding all the information into one vector, we led our model translate more on a word by word level. X represents the words in our source language, and Y, represent the words in our target language. The trick here is in the α values, that the system will need to learn. The translation will happen one word at a time, but we will have to look at all source words every time we output

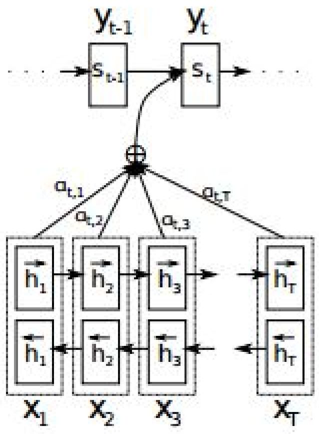

Figure 3.2: Attention Mechanism from [1]

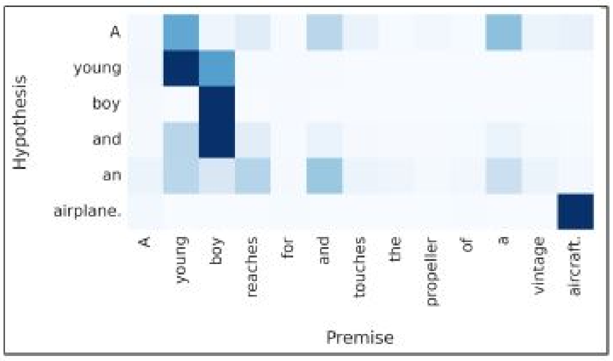

Figure 3.3: Attention Mechanism weights from [21]

a target word. This cost is the major drawback for attention mechanisms, needing to recompute all α values at each word that we need to output. Figure 3.3 is an amazing feature of attention models, allowing us to see what our model is looking at, by checking α values. The stronger the blue square, means that the network is paying more attention to that word.

3.3 Conclusion

This chapter has been used to present a more detailed problem definition as long as some previous work, and some current work being done. This chapter must serve the reader as a way to see the paradigm shift, from hand-crafted features and small models, to loads of data and big models.

We have seen some previous approaches to solve the problem, shallow and logic approaches, that due to the difficulty of the task, yelled not good enough results.

To finish this chapter, we have seen some current approaches not used in this thesis work, such as attention mechanisms, that aim to learn which words are relevant in every context, so that the model knows at where to pay attention to solve the task.

The next chapter will present the approach to meet the goals explained in the introduction 1.2. It will also provide a more detailed explanation of the model used, its architecture, and the use of word embeddings. It will also explain the possible modifications that we can apply to the model. Finally it will also detail the modification to the model proposed to increase the accuracy of it.

Chapter 4

Approach

In this chapter I will try to explain the ideas applied in the thesis, the decisions chosen while implementing and other relevant aspects on the more practical part of the work.

The implemented thesis model has revolved around three sub-models. This sub-models will help the study of possible limitations of the task using this or similar approaches. As a brief introduction on what is going to follow, I will describe this sub-models. The first model is related to words and their representation using word embeddings. The second model is how to encode the information of a sentence once we have encoded the different words. This is done via the use of LSTMs, that were detailed in chapter 2, section 2.2.3. The last model is a classifier on top of the sentence model, that relies on the sentence vectors as inputs, and outputs whether the pair of sentences are *entailment, neutral, contradiction.*

4.1 Word Representations

In this short section I will introduce all kinds of data used to perform the tests. Note that the dataset will be properly presented in the next chapter.

4.1.1 GloVe Model

As we have seen in Chapter 2, GloVe vectors are high dimensional vectors, that contain semantic and syntactic information about the token that they encode. Different corpora size pretrained word vectors are published by the authors of the GloVe model, but we know from [20] that the bigger the corpora, the better results this vectors yield. Therefore, the embeddings used are trained in a 42 billion tokens corpora. Using this size of corpora not only provide us with "better" vectors, but also diminishes the chances of not finding a word from our dataset. In smaller datasets of pre-trained word embeddings, it is common that some words are missing, this is completely undesired in our work.

In case we have some unknown word never seen in training, highly unlikely, I used a common approach of having an unknown vector to assign to that token.

This unknown vector is a vector of the same dimension as the word embeddings, in our case 300 dimensions, with all its initial weights randomly initialized between $[-0.05, 0.05]$. Since finding an unknown word is highly unlikely, I used the same vector for all the unknown tokens, when I trained using GloVe pre-trained vectors. This can also be done, because during the training process I decided to train the embeddings along with all the other network parameters, meaning that this randomly initialized vectors would get proper values, according to the context the tokens are being used. Training the embeddings, specially when having tokens not found in the GloVe model is specially interesting because the only embeddings that will remain random, will be the ones representing unknown tokens found only in the test set, thus generating more accurate vectors for each particular task. Since part of the thesis is setting different training setups, and then studying the results, a different setup that was used was to not use the glove word embeddings. In this case, a randomly initialized vector was assigned to each token, same weight setup as previously mentioned ($[-0.5, 0.5]$), and as the previous setup, the word embeddings were trained as part of the process.

As part of the work done that specifically interacts with the GloVe model, that later in the next chapter will be presented, it is worth explaining that I also tested using the embeddings as direct input for the network, making the embeddings not a parameter to be trained during the training process. Instead of inputting a vector representing the sentence, I tried using as input the embeddings itself or each sentence. Notice that this changes considerably the first part of the network, as we go from a 1-D array with sentence length, to an 2-D array, of shape (|S|, 300), where the $|S|$ is the length of the sentence. This parameter |S|, is something that has had some influence in the modelling of the first layer of the network. Note that in the first and second setups presented, when we train the embedding parameters as part of the network, the length of the sentence is less relevant. It is important to realize that when we use as input, a 1-D array with the token indexes, as the ones presented in 2.2.2, the first operation, is just selecting which vectors, representing the word embeddings, we are going to use. If we have 0s in the sentence, we can deal with these zeros as padding, having less relevance in the model. On the other hand, if we do not train the embeddings, and we treat them as input instead of parameters, we need to input the network with a bunch of extra zero vectors to have all the sentences the same length. This might not seem relevant, but, in our case, the SNLI dataset the hypothesis have an average of 14 words, but the longest sentence has 78 words, meaning that we would have to pad with the sentence with 64 zeros. In case we decided to stick with the embeddings as input for the network, there is another solution that it is called "bucketing". Bucketing consists of grouping same lengths sequences, and then feed them into the model. This might seem a better idea, but remember, we can't reorder our sentences freely, because each sentence has a pair. That would mean, that we would have to pick a method of ordering the sentences at the same time, to pairs of sentences of the same size. The bucketing solution does not seem to be very useful in this particular case, so as many cases, a mix of padding and bucketing would seem a better solution. Still this solution carries with the sorting problem, so it has never been used as part of the project. In the end I got

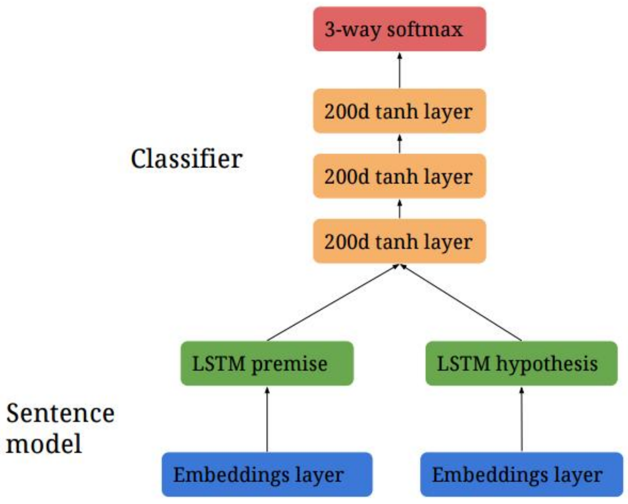

FIGURE 4.1: Baseline model (Image based on [2])

stick to padding in the case of not training the embeddings as part of the network.

4.2 Baseline Model

In this section I will introduce the best performing model presented in [2]. We can see the model in figure 4.1.

The model implemented for the task has 3 key components, a first embedding layer, followed by the sentence representation layer (the LSTMs), ending with a classifier. In the case of figure 4.1 the classifier is represented by the three hidden layers of 200 units, ending in a 3-way softmax layer. As a remainder, the 3-way softmax layer represents entailment, neutral and contradiction, meaning that the baseline accuracy would be at 33% , random guessing. The sentence model is an LSTM network with 300 dimension vectors of inputs, and 100 dimension vectors as output. In the figure, the sentence model consists of the embedding layer and the RNN/LSTM network. I also decided to have two separate embedding layers, with the same initialization parameters, the GloVe vectors, but they are tuned separately. Since we encode the premise and the hypothesis separate, to be coherent with the model the embedding layers needed to be separated aswell.

4.2.1 Influence of Hyper-parameters

This subsection will try to give an overview of modifications of this network ant its motivations, choices that are independent of the input that we choose to have.

There are three clear choices in which we can modify the network, and see what the performance looks like. Two of these choices relate to the classifier, first we can modify the number of layers, see if stacking makes the classifier perform better, or if it learns meaningful parameters in order to distinguish the three classes. The second clear modification is to modify the 200 dimensions of each layer. This may affect directly to the dimension of the embeddings. With the current setup, the first layer is straight the sum of the length of the previous layer. Having different dimensions on each layer can also be a possibility, although we won't explore that in this work, since as we will see, the classifier has less impact that we can expect.

The third clear choice to modify, has a greater impact, than the modifications introduced before. It has the same idea as the dimensionality of our word embeddings. Theoretically, depending on the length of the vocabulary, we need more dimensions to properly embed all the words. In practice, the quality of the embeddings rely more on the size of the corpora, than in the dimensions of the vectors. Nevertheless, we do not use fifty dimensions for our vectors, nor we use 2000 dimensions. A common choice for the embeddings is the embeddings that we used, 300 dimensions. This leads to think, that when we are encoding a sentence using LSTMs, the dimensionality of the vector may affect on the information is able to capture. So another modification applied, that turned to be also useful, was to make the vectors resulting from encoding the bigger. In particular I tried 200 and 300 dimension vectors. The major drawback on doing this, is a longer training time. When using 300 dimension vectors to encode the sentences, the training time was three times bigger longer.

4.3 Network Modification

A term that has not been introduced before is the nature of the relation between the premise and hypothesis. This relation is not symmetric, meaning that:

$$p \to q \neq q \to p \tag{4.1}$$

There is no negation that while this is true, when solving this task, we are not solving a real inference task. We are encoding two different sentences, and then if the encoding is similar, the network will likely predict entailment, otherwise will predict contradiction or neutral. In an effort to not interfere with the symmetric nature of this relation, we concatenate the resulting vectors of the LSTMs, and feed them into our classifier.

Interestingly enough, when working with word embeddings, more concrete studying the results of the famous analogies, in which we compute something similar to $\vec{king} - \vec{man} + \vec{woman} = \vec{queen}$, it was suggested as multiplication as a powerful method to compute these analogies. Although clearly multiplication does not respect the symmetry presented before, it introduces some interesting ideas. The idea is also inspired on attention mechanisms introduced before, but in my case, the weights are not learnt, it is a simply one-to-one relation.

Therefore, I decided to make a change that has reported an increase of accuracy over the best model in [2]. So far, after encoding the premise and the hypothesis, we concatenate the vectors and feed them into the classifier. The change that I propose is that instead of concatenating the resulting sentence embedding vectors, use an element-wise multiplication. While doing this, I also decided to encode

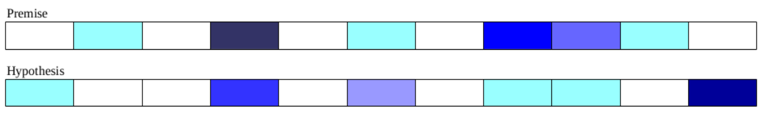

FIGURE 4.2: Encoding of premise and hypothesis

more information in the sentence model, going from a 100 dimensional vectors to 200 dimensional vectors, the result of the multiplication element-wise, is still a 200 dimensional vector, hence we do not have to modify the classifier.

The most interesting idea behind multiplying the two sentences is, that we will empower the weights that are shared, and diminish weights that are not in common. Imagine figure 4.2 as the result of encoding two sentences, our premise and our hypothesis. The darker the blue, the higher the value it is, otherwise, so the resulting vector would be a vector with some values way higher than the others in case of two sentences with similar meaning. This, theoretically endorses the model to find more entailment relations than it should, but in reality, at least in the SNLI dataset, improves the accuracy significantly.

4.4 Conclusion

This chapter is more a hands-on approach chapter. Having seen the theory behind the models used, and the amazing capabilities of all the techniques learnt, this chapter helps understand a little bit more all the work done.

It first introduces ideas to work with the word embeddings, such as using them as input in the network or allowing them to be part of the training process, modifying them along the training process.

Then it introduces modifications that are straight on the model, and their implications, like changing layer size, dimensionality of the layer, or the number of layers.

At the end of the chapter, it is introduced the symmetric concept behind the Natural Language Inference task. More importantly it introduces a change that, while not respecting the symmetry of the task, has improved the results significantly.

In the next chapter we will see the results of the proposed approach. It will follow a similar schema as in this chapter. The results obtained are gathered from the different modifications that are proposed. A proper explanation on the dataset will be given, aswell as a brief introduction to the framework used to implement the different models.

Chapter 5

Evaluation

In this section I will present with the results achieved on the data set. All the evaluations have been done using the Stanford Natural Language Inference dataset [2]. The base model, and all its modifications have been implemented using Keras [4].

5.1 Data Set

The Stanford Natural Language Inference corpus contains approximately 570.000 pairs of sentences. Each pair is manually labeled as Entailment, Contradiction or Neutral. There are two type of labels, the gold label, also known as ground truth, which is the one used to determine whether a pair is correctly labeled in the training or testing process, and also a set of up to five manually annotated labels by annotators. The gold label is captured by majority vote, meaning that there needs to be at least three votes of the five in the same direction. This is key in my research, and will be seen in this section. To further illustrate this label situation we can check Table 5.1

The data set is already prepared for Natural Language tasks, and it is split into three parts, development, training and testing, with 10k, 550k, and 10k sentence pairs each. The data set also contains the Part-of-Speech tags of each element of the sentence, for each sentence, in case needed.

Another key factor that has been explored in this work is the agreement between the author of the sentences and the ground truth. As we saw in table 5.1, Manual

Premise	Hypothesis	Manual Labels	Ground Truth
Premise 1	Hypothesis 1	[E E E E E]	E
Premise 1	Hypothesis 2	[E N E N N]	N
Premise 1	Hypothesis 3	[C C C N C]	C
Premise 2	Hypothesis 1	[E E C N E]	E

TABLE 5.1: SNLI corpus format

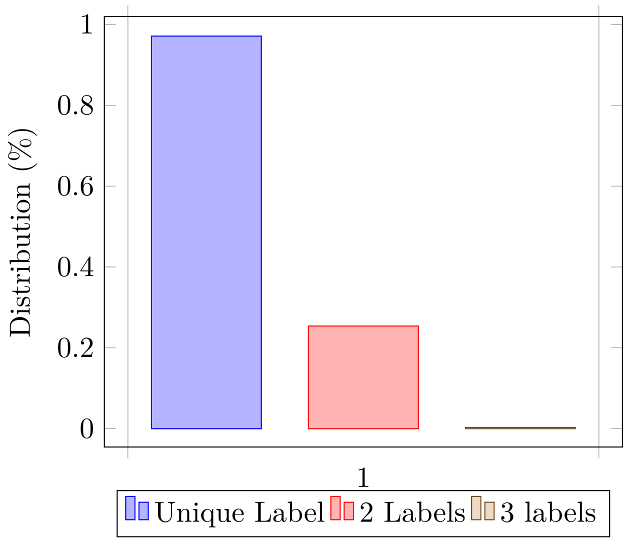

Figure 5.1: Data distribution for training set

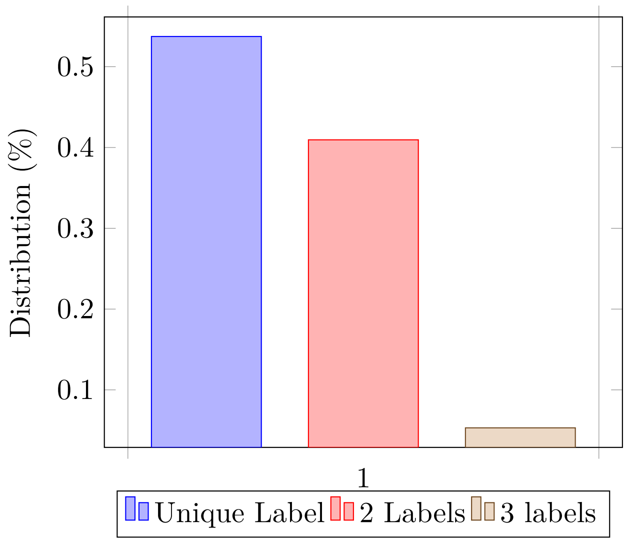

Figure 5.2: Data distribution for test set

Labels do not always agree 100% with the Ground Truth label. This begs the question, can the model perform better on a task that humans not always agree? In figure 5.1 we see the distribution on the agreement of the labels in the training set. Figure 5.2 shows the same distribution for the test set. As we can see there is a huge discrepancy in both data sets distributions. We will see the impact of this on the results section.

5.2 Implementation and Keras

As part of the thesis work was recreating the results from [2], but no code was released for that publication, I implemented all the code needed for the thesis. In appendix A there are a couple of code snippets of the implementation. The implementation is done in Python, using pandas [16] to work with the dataset, and numpy [25] to work with vectors. I used the framework Keras [4] to implement all the Neural Network models. Keras is a Neural Network library implemented in Python and works on top of Theano [23] or the newly released Tensorflow library. In my case we used as back end Theano, because at the time of the initial implementation it was the only back end possible, since Tensorflow was publicly released early November. One key advantage on choosing Keras as a library to work with, was the high level abstraction it provides, helping researchers focus on producing and testing models faster.

5.3 Results

In this section we will explore the results obtained from the model and its small modifications. We will start from the test accuracy on the best model presented by [2] and go on exploring the results that try to validate whether human agreement on the task matters.

5.3.1 Baseline Model

As stated during the thesis, the first important aspect was being able to reproduce the the results from [2]. In the paper the authors report an accuracy of 77.8% on the test set using a model like figure 4.1 .

The first thing that was done was generate a vocabulary out of both the training and the test set. The data sets came in form of pairs of sentences, its labels and their Parts-of-Speech tags, so before generating the vocabulary, I first had to tokenize all the sentences. The vocabulary consists of all tokens without stemming or any other kind of preprocessing. This was done in order to extract GloVe embeddings more efficiently. The embedding layer is of the size $(|V|, nb\_dimensions)$ where $|V|$ is the length of the vocabulary and $nb\_dimensions$ is the length of the vector that represents the word embedding. The way I dealt with words that are in the vocabulary but are not represented in the GloVe dictionary, as explained in the previous section 4.1.1, is by generating a 300 dimensional vector with small weights, between $(-0.05, 0.05)$ and then using this vector for all the unknown words in the vocabulary. This vector that represents the unknown tokens is also trained in the embeddings layer as any other word embedding.

Having explained the embeddings, we use two LSTMs to encapsulate into a single vector both the premise and the hypothesis. The key at this point is that the premise has no information about the hypothesis and the hypothesis has no information about the premise. At this point we have two vectors that contain the information about the premise and the hypothesis, and we proceed to feed them at the same time to our classifier, by concatenating both vectors. The results of this can be seen

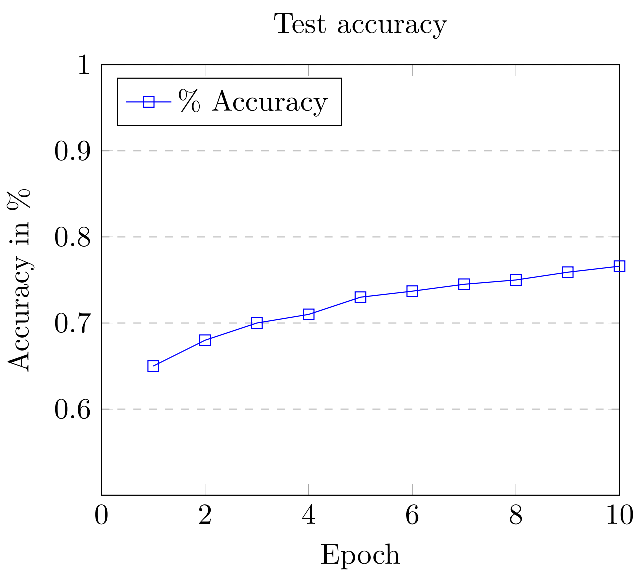

Figure 5.3: Accuracy over the test set, having trained all the training set

in figure 5.3. The accuracy resulting from this model is close to the 77.8% achieved by the authors of [2], being **76.6%** in my best test, and averaging over 76.2% over all the training and testing sessions, achieving close to state of the art results.

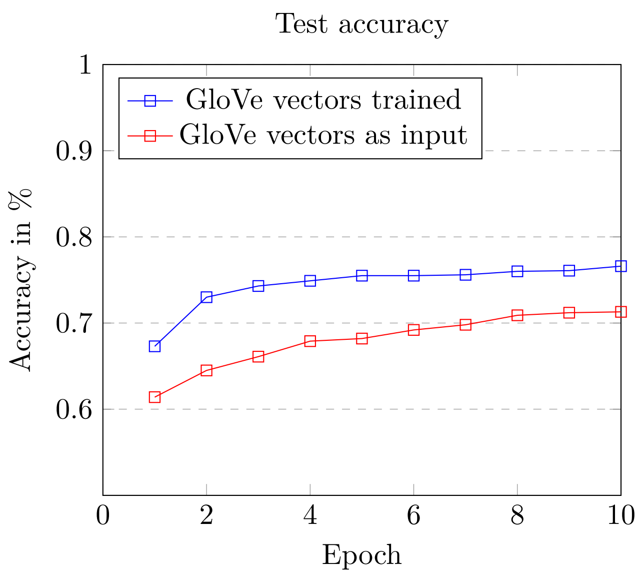

FIGURE 5.4: Train vs no-training on the embeddings

5.3.2 Word Embeddings Related Tests

My next focus was on exploring whether pre-trained word embeddings would affect the output of the model, and to study how much would affect had training the embeddings as part of the network, or leaving them purely as inputs of the network.

Since [2] uses GloVe vectors, in order to distort the least possible the results, I trained my models using GloVe vectors also.

First step is studying whether it impacts in the model if we train the embeddings as network parameters. The idea of training the embeddings with the model has an small concern. When we train the embeddings with a non-general dataset, e.g Wikipedia data, the embeddings are more specific to the specific task, and less generalist, so they are less portable to other tasks. It has also been expressed by the authors of [21] that training the embeddings may lead to overfitting. Nonetheless, as we will see in the next figure, the results produced between training or not training the parameters of the embeddings are quite relevant. Training the embeddings produced a bigger difference than initializing them as random. This can lead to think that training the embeddings is more important than actually having a good start with the embeddings.

Nonetheless the model achieves the best performance when we tune the parameters of the word embeddings during the training phase, using the pretrained word vectors as initialization of the embedding layer. We can see the results in Figure 5.4

In the next figure (5.5) we can observe the difference between initializing the word embeddings using the GloVe vectors, or random vectors of the same dimension (300) and weights between $(-0.05, 0.05)$. Each token (word) had a randomly initialized vector, that was later fine-tuned during the training process of the network, meaning

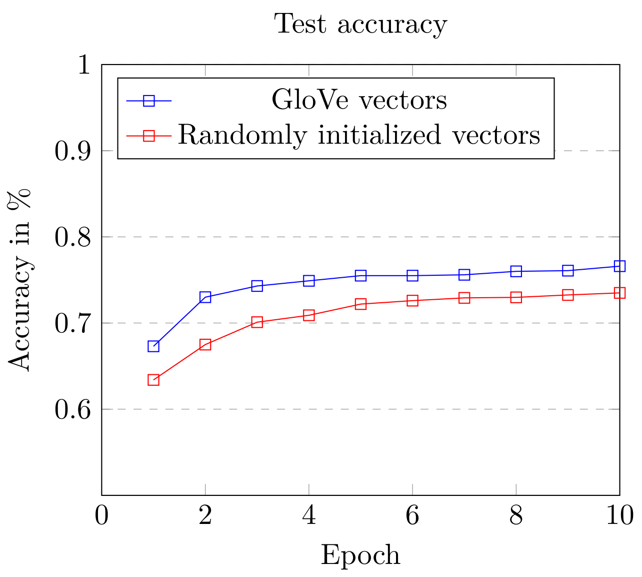

Figure 5.5: GloVe vs random initialization

that the process did not change whether I used GloVe vectors or randomly initialized vectors. Figure 5.5 illustrates the difference between between both initializations. As expected GloVe vectors outperform in the task randomly initialized vectors by a $3 - 4\%$ at every epoch of training. Although it is surprising that randomly initialized vectors perform as good as they are performing in the task. At the mark of 76% on accuracy using GloVe vectors, randomly initialized vectors had an accuracy of 73%. This leads to think that for this particular task, with a corpus of length 33550, using pre-trained vectors provides us with a good head start.

On the other hand, this leads to think that within domains that have a low word count $|V|$, training the embeddings from scratch is a viable solution, because the model can achieve notable results without the needs of using pre-trained vectors to further fine-tune them in the process.

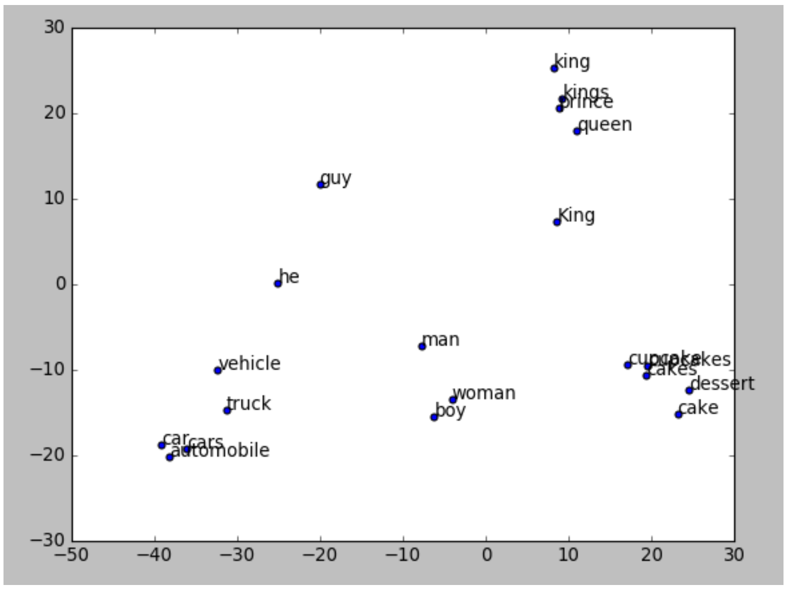

FIGURE 5.6: Word embeddings visualization

Word Embeddings concern

While working on the thesis a concern on how we acquire word embeddings and how we use them has raised. There is no doubt that word embeddings have delivered a major leap in NLU related tasks, and clearly being able to work with words as vectors has many fantastic features The concern that I expose does not deny at all how well they perform, it will try to expose a limitation that I feel we will find in a not so much distant future while trying to solve other NLU tasks, or while trying to squeeze the accuracy in our models. The concern also raises in the representation process of the data, not on the neural models that we use. One of the major differences between language and images is that for images we already have good representations to input our models, basically, we input the images itself, making images also easier to deal with. For language, word embeddings represent the best approach that we currently have, but will it be enough?

In order to express my concern, first I had to visualize the word embeddings, which I used t-SNE [24] a technique based on Principal Component Analysis (PCA) of the relevant embeddings [1]. Figures 5.6 and 5.7 show how this word embeddings can be seen in a 2D space. Figure 5.6 shows the classical example with man and king, it has also been added words *cake and car* to see how clusters of words are formed. Interestingly enough, when represented this way, *car* is closer to *man*, than to *kings*, which makes perfect sense. And *man or woman* are in between *cake or car*. The next figure, 5.7, expresses one of my major concerns while doing the thesis, and one my major concerns regarding the information captured by word embeddings that I will try to explain as best as possible, I will focus a bit on word embeddings

[1]Visualizing Word Embeddings with t-SNE

and by the end we will see the link to NLI task.

The state-of-the-art methodology of producing word embeddings is based on capturing as much information as possible in a vector. As we saw in the GloVe model, the idea is to perform a dimensionality reduction of the word probabilities co-occurrences matrix. In other cases, the idea is to predict a word w.r.t a context of n words, in the end, we are trying to encapsulate some information in a vector, and this information is always based on a context. In reality, what we do, is capture some relations between some *tokens*, not words. What does this mean? It means that when we are computing the co-occurrence of the word, let's say *plant*, it does not matter whether the text is about plants and seeds, or about plants and factories. We simply encapsulate a context into this token. Interestingly enough, the GloVe model produces two vectors (eq 2.2), the first vector is what we know as our word embedding, but the second vector, is what is called context vector. These two vectors, perform similarly in several tasks. Interestingly enough, as the authors of [20] mention, is when they add these two vectors that the accuracy of, lets say, the analogy task is further improved. We encapsulate information into vectors over and over, the GloVe model that was used for this thesis was computed on over 42 billions of tokens. The tokens were not preprocessed, meaning that the token *plant* and *Plant*, may or may not encapsulate the same information. Actually, they do (see figure 5.7), but that is not the real point. Whether *plant* or *Plant*, or *sick* and *Sick* capture the same information is not relevant. The relevant point is that these, not so random chosen words, have several meanings. To give a number, in the top 500 most common words in English, they have an average of 23 meanings. This means that when we find a token in our corpus, lets say, *book*, we will encapsulate under the same vector all its meanings, no matter if book is for example a noun, *"To read a book."* or if it is a verb *"I am going to book this book."* or used in figure of speech like *"I read him like a book"*, and out of these 3 sentences, we used the token *read* in two different scenarios, but still we are going to try to capture as much syntactic and semantic information in the same vector. This in fact, might not be like a bad idea, none of these sentences in Natural Language Inference might produce any troubles to the model. Now let me give a pair of sentences that might produce a problem.

- Plants produce oxygen, they are really good for the environment.
- Plants produce too much pollution, they are really bad for the environment.

And remember, *Plants* vector contains as much semantic and syntactic information as possible. So when a model reads the word *Plants*, it can either be good, because they produce oxygen, or be bad, because they produce too much pollution. Now we can check figure 5.7, where we can find the following words used to generate the clusters, plant, factory, houseplant and cake. The cake cluster, is quite clear, but the plant cluster? *plant* is in between *factories*, and *flowering*, and has roughly the same distance to *houseplant* than to *factory*. This is quite sensitive when dealing with Natural Language Inference, because, same words, with same spelling, might have different meaning in roughly the same context. The word *sick*, can be used as either something that is really good, specially in the Internet slang, or meaning that something is "disgusting". I believe the current way of generating word embeddings leads to some confusion during the learning process by my models, or any other models.

To further illustrate this concern I will present some sentences extracted from the test set, in which the model is not able to correctly classify. In these sentences the words *play or show* are used in different contexts.

- Play
  - A group of kids play on a colorful structure.
  - Two men in cultural garb play the drums for a crowd.
  - A black man is performing in a play.
- Show
  - A man and woman show off their baby.
  - The men are hosting a news show.
  - A man is punching a picture to show great anger and rage outside to people.

In all these examples presented, the error is similar, either it predicted Entailment or Contradiction, and was expecting Neutral, or was expecting Neutral and predicted Entailment or Contradiction.

Despite the concern expressed, word embeddings are a much valued resource, that have proven to help beat all state-of-the-art tasks, and that have much room of improvement. It provides us with a great way of representing words, thus enabling us to further improve in NLU tasks.

5.3.3 Learning from the Labels

This subsection is going to explore the results and my concerns about the model, that were introduced previously in 5.3.2. All the tests are going to be done with the same setup, GloVe embeddings and training the parameters of the embeddings during the training process. When we are building any kind of AI system, it is good to know how humans perform at the task. It forms for a base on how good a system can be to be "acceptable". The idea behind the next test was to see how the model performed in three different setups.

- Train with all the dataset (the training set), accepting cases were 3 labels were assigned to a sentence pair.
- Train with only sentence pairs that had maximum two different labels, no matter the label combination, for instance [entailment, neutral] or [entailment, contradiction] were accepted.
- Train with instances that all the annotator labels were the same, full agreement on the ground truth.

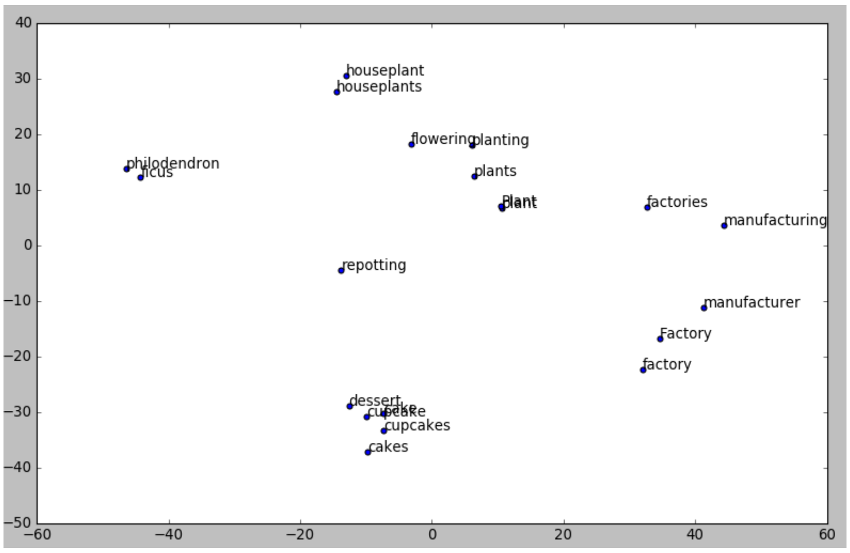

Figure 5.7: Word embeddings visualization

The distribution of the data was shown in Figures 5.1 and 5.2. As we could see in the figures, the distribution of the data is not particularly balanced, specially between all match label and two labels. The next Figure 5.8 reflects the test accuracy on the three options. Interestingly enough, when we look at pairs of sentence with pure human agreement, the model performs better. The other options are hard to discriminate, due to how small is the size of the pairs of sentences with three labels on it. This is consistent with the idea that the model is able to perform inference in cases that there is little ambiguity, as such all human annotators agreed on the label.

Similarly to Figure 5.8, I studied the results according to the outputs, meaning that it was studied if there was any relation between the three possible outputs. In order to do so, I decided to study what happened when *neutral* output was ignored, still training sentences with neutral labels, but the output was set to 0, and as output it was chosen the highest of the two *entailment or contradiction.* As an example:

- A woman with a green headscarf, blue shirt and a very big grin. *premise*
- The woman is very happy. *hypothesis*
- Output results: [0.46199772 0.52680892 0.01119339] (a)
- Output expected: [1 0 0]
- The woman has been shot. *hypothesis*
- Output results: [0.47479814 0.44156361 0.08363828] (b)
- Output expected: [0 0 1]

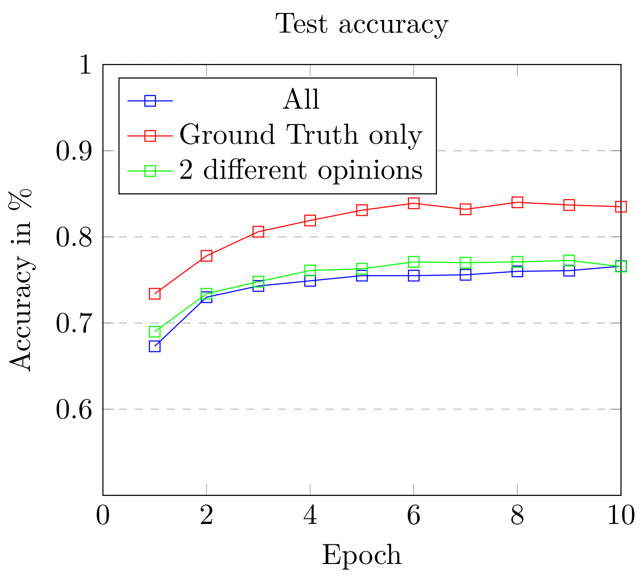

FIGURE 5.8: Comparision on human agreement

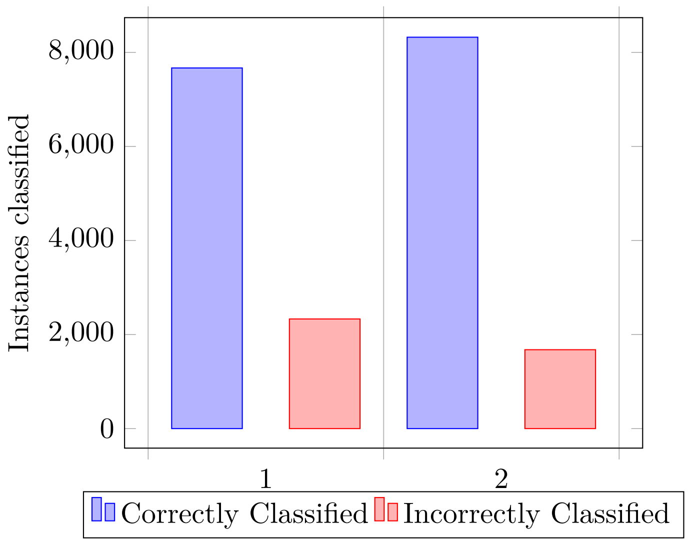

FIGURE 5.9: Data distribution for test set

In this particular example, the model predicted *neutral (a)*, when in reality, it was expecting to be *entailment*. To study this phenomenon, neutral was turned to 0, meaning that what we have is [0.46199772 0 0.01119339]. There is no need to normalize it, since we are only looking for the index of highest remaining value, and the index will not change even if we normalize the remaining values. To do so I used the 2331 wrongly classified sentences, as the one presented before. After modifying the outputs Figure 5.9 shows the changes, where the first two columns

Expected \ Output	Entailment	Contradiction	Percentage
Entailment	-	389	0.235
Neutral	472	428	0.53
Contradiction	388	-	0.235
Percentage	0.512	0.487	1

Table 5.2: Results from removing neutrals

represent the accuracy previously to this modification, and the second, represent the accuracy after. As expected accuracy increases, correctly classifying nearly 30% of the wrongly classified instances. With the remaining 1677 instances that are still wrongly classified, we will see where the classification error is.

Table 5.2 considers all the test samples that are wrong, therefore if an entailment is expected it can only be contradiction, because we also removed neutrals from the output labels. The second row, shows all the expected neutrals, where are really classified, falling under entailment or contradiction. It shows two notable results on the study. The columns represent the output from the model, and the rows represent the expected output. As we can see, I already removed the *Neutral* output according to the removal of the neutrals as explained before. The remaining pairs of sentences that are wrongly classified, produce similar output errors, meaning that there is no bias towards any of the two remaining outputs. This can be seen when summing over the columns. The results are roughly 50% for each, Entailment or Contradiction.

On the other hand, we have another "expected" result. When we look at the rows, the one with more misclassified classes is still *neutral*. Again, rows of entailment and contradiction are roughly the same. The meaning of the rows, to make it more clear is, the number of expected entailment classifications that were classified as contradictions. This means that there is roughly the double of times in which a sentence that has to be classified as neutral is classified as any other of the two.

As a result of this, I decided to study what classification accuracy rate would the model have if there were no neutral labels. This has two main ideas behind it. The first as expressed in section 5.3.2, is what information is actually encapsulated in the embeddings, and therefore, in the sentence embeddings of the network. The second one directly derives from figure 5.2 previously presented. Neutral classifications are hypothesis that could potentially be true, but are not directly derived from the premise, hence, they are neutral. As we saw before, there is roughly the double of neutrals misclassified than entailment or contradiction. To further study this fact, I removed the neutral labels, and changed the output of the network to be a 2-way prediction. Figure 5.10 shows the results, in comparison with the standard model shown in figure 5.3

The results achieved by the network are completely surprising. First is important to mention that the dataset went down 33% of its size, remember that each premise has 3 matching hypothesis, one for each possible label, but the ratio of sentences and labels remains the same, because the dataset is perfectly balanced for each label, so

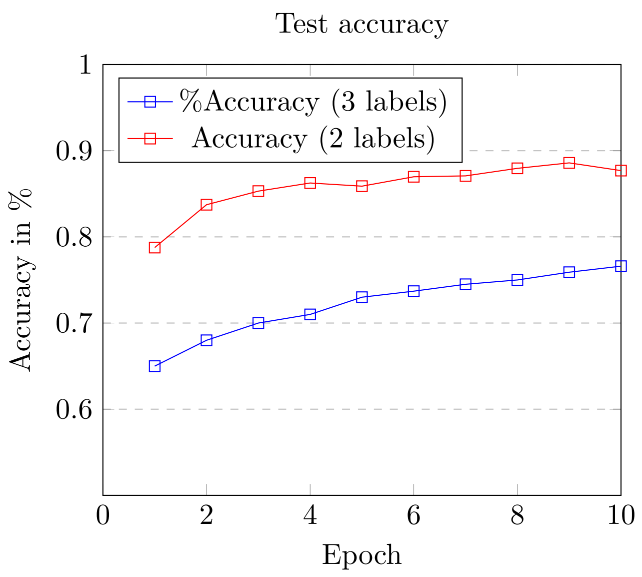

FIGURE 5.10: 3 labels vs 2 labels

this reduction on size of the dataset has no effect in the learning process. Without the neutral label, the baseline accuracy raises to 50%, making the problem much easier to deal with. An 88% accuracy was achieved by the network. This is close to an 11% better accuracy than the model with three labels.

Along this subsection we have seen how *neutral* pairs of sentences bring an element of uncertainty that our model is not particularly good at. Further research should head into studying how to deal with ambiguity and similar linguistic phenomena. On the other hand, using the SNLI data set we can achieve good accuracy results on entailment and contradiction.

5.3.4 Breaking the Symmetry

In this last subsection I will present the results achieved when instead of concatenating the premise and the hypothesis, we multiply them, removing the asymmetric condition in the inference task. I found this approach particularly interesting, even if it breaks the asymmetric relation $p \to q$ it provides us with a way of dealing with the sentence vectors that have some practical results. The results presented in this section will follow the same idea than the previous subsection when we studied the labels result. We will study what results we get when we have 3 possible labels to classify, aswell as when we only have 2 possible labels.

In this particular case, the network is slightly different in dimensions. When we concatenate sentences, we need to sum their length in order to use them in our classifier. As we see in 4.1, from 100 dimensional vectors, we obtain a vector of 200 dimensions. When multiplying element by element, the length of the vector remains unaffected, hence the network would have a first layer of 100 dimensions.

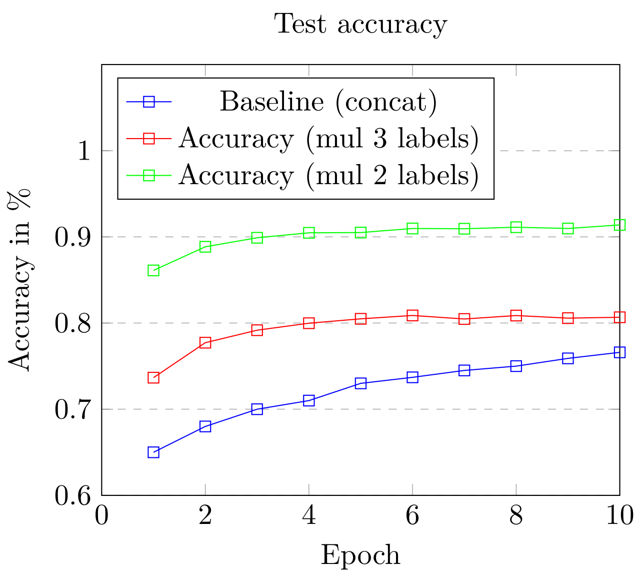

Figure 5.11: Multiplication vs Concatenation

This presented the question whether 100 dimensions would be good enough. To fight against a possible loss of information in the classifier, in this subsection our sentence models have 200 dimensions, remaining the first layer of our classifier to be also 200 dimensional. This, of course, alters they way our sentences are encoded, making it a little bit harder to compare with previous results. Nevertheless to keep track of our first approach, I will keep the baseline accuracy in the figures shown.

In the figure above (5.11) we can see the results of the modification on the network. From the results some interesting conclusions can be drawn. Multiplication works as expected, enforcing bigger differences between the vectors, helping the classifier identify which cases are which. We still are stuck in the 80% -81% accuracy when classifying with three labels. On the other hand, when using the multiplication with 2 labels (entailment, contradiction), our accuracy goes to 90%. This reinforces the idea that neutral pairs of sentences are not properly dealt with. It also reinforces the premise that when multiplying two vectors, we are providing the classifier with vectors that help properly classify entailment or contradiction. Entailment vectors are expected to have a larger amount of higher valued parameters, meanwhile contradiction vectors are expected to have a lower amount of lower valued parameters.

5.4 Conclusion

In this chapter we have seen the results of the work on the thesis. We first introduced the results of our baseline model. From there we have seen the results of choosing between training our word embeddings as part of the network, or using them as input of the network. We have seen how training the embeddings with the network produce better results, despite some [21] implying that training the embeddings with the network may induce to overfitting.

After studying this, we have seen more relevant concept. We have seen how the sentences labeled as neutral are more difficult to learn by our model, and how models that have to choose only between entailment and contradiction are least prone to errors.

To end the evaluation chapter, we have seen how introducing the concept of multiplication instead of concatenation of vectors helps the classifier perform better. This concept breaks the asymmetry in the inference task, but helps the classifier improve its accuracy. This has been empirically demonstrated by training on the three labels, or on only entailment and classification. It is important to remark that, despite breaking the asymmetry of the inference task, when training our model with only two labels, we have achieved an accuracy over 90%.

Chapter 6

Conclusion and future work

6.1 Conclusions

In this thesis we have seen what Natural Language Inference (NLI) is, as well as some state-of-the-art approaches.

I have introduced the problem facing in the NLI task in chapter 3. Remember the setup of the problem is to infer whether our hypothesis can be derived from our premise. This has be done using word embeddings and Recurrent Neural Networks (RNNs), specifically, Long Short Term Memory Networks (LSTMs), the theory about them can be found in chapter 2.

The thesis began by implementing the best performing model in [2]. This was the starting point for the two main contributions. The first one has been to explore the results obtained by the implementation of the previous paper, and the influence of several aspects of this implementation. The second one has been introducing a new model network model, allowing use to obtain better results in the task.

- From the first goal we can conclude that:
  - Training word embeddings as part of the network parameters provide better results. Also initializing the word embeddings using pre-trained embeddings yield a boost in accuracy.
  - Neutral labels cause the most troubles to our classifier.
  - It is significant that where humans have most agreement, the model performs significantly better than when we add sentences with disagreement.
- The second goal was achieved by modifying the concatenation into an element-wise multiplication, between the sentence model layer, and the first classification layer.
  - Breaking the symmetry relation yield better results by providing the classifier with more distinct features in the vectors.

Natural Language tasks contain huge challenges yet to be solved. Common sense allows us to perform a huge number of tasks, like word sense disambiguation, or like inferring an hypothesis from a premise. It is a huge challenge to improve how we represent words to fully solve this tasks.

6.2 Future Work

I have made clear along the thesis work a path that I think its worth exploring. I am not sure the way we are dealing with meaning is efficient nor good enough. NLU tasks are significantly harder than other tasks, one of the reasons is the ambiguity of a word, or even a sentence, making context and common sense a key feature while dealing with these tasks. Word embeddings represent a major advance in NLP tasks, that have improved significantly accuracy in many tasks, being able to deal with context much better than previous approaches. Nonetheless, further refinement on the way we generate word embeddings needs to be explored. RNN/LSTMs work very well at sequence encoding, and during this year modifications and improvements over [2] have proved to be very useful at boosting the accuracy of many NLU related tasks.

- Study how to generate word embeddings that have more usefulness. Currently there is no meaning differentiation in the token embedding. This needs to be studied and solved if we want to keep improving in NLP tasks. Possibly add in the word embeddings Part-of-Speech differentiation.
- Study figures of speech. In our daily life, we use figures of speech constantly, and we all know that the words used have different meanings out of the context, maybe not even related to the figure of speech used. It is worth to see if we can group figures of speech into special tokens also represented by vectors.
- Explore more deeply why neutral sentences generate so many troubles to the classifier. During this thesis we have seen how much better accuracy the model can achieve when there are no neutrals. Again, I'm sure it is worth revisiting all the encoding of the meaning in word embeddings.
- The last two lines of study for future work are quite related.
  - Deep Learning techniques are as limited as their data. Because word embeddings can be trained in "real data" such as wikipedia, we should be able to find ways to train NLI models in data that is not specifically prepared for the task. Right now, generating the SNLI dataset is a heavy work done manually, therefore we can only train our models using this dataset. Easier generation of datasets should be explored, since supervised learning is really limited to the datasets that we can get.
  - In line with this, we should explore ways to go from the SNLI dataset, to real data. It would be interesting to study how viable a system that has learnt from the SNLI dataset acts in a real world application.

Appendix A

Appendix A

In this appendix snippets of the code used to built the model will be provided. The final iteration of the code will be published soon in my Github account. The code can be found in the NLI_Code[1] repository. Note that this repository will not contain the GloVe vectors, nor the snli dataset. Right now, there is a working version of the code, but several things are going to be modified to make the external use easier, such as selecting the layers dimension, or the way the sentence model is fed to the classifier.

- The GloVe pretrained vectors can be found in the following address GloVe Vectors[2]
- The snli dataset can be found in the following address SNLI project[3]

A.1 A little bit of Keras Code

It would be a bit unfair to not present a little bit of the Keras code used to do most of my thesis work.

The following snippet defines our baseline model.

Listing A.1: Building the baseline model

```
from keras.layers import recurrent
from keras.models import Sequential
from keras.layers.core import Activation, Merge,
    Dense, Dropout, Flatten, Reshape
from keras.layers.embeddings import Embedding
import numpy as np

premise_model.add(Embedding(output_dim=300,
    input_dim=input_dim,
```

[1]https://github.com/tmdavid/NLI_Code
[2]http://nlp.stanford.edu/projects/glove/
[3]http://nlp.stanford.edu/projects/snli/

```
    mask_zero=True,
    weights=[emb_init]))
premise_model.add(Dropout(0.1))
premise_model.add(self.RNN(100, return_sequences=False))
premise_model.add(Dropout(0.1))

hypothesis_model.add(Embedding(output_dim=300,
    input_dim=input_dim,
    mask_zero=True,
    weights=[emb_init]))
premise_model.add(Dropout(0.1))
hypothesis_model.add(self.RNN(100, return_sequences=False))
premise_model.add(Dropout(0.1))

self.nli_model = Sequential()
self.nli_model.add(Merge(
    [premise_model, hypothesis_model], mode='concat',
        concat_axis=1))
self.nli_model.add(Dense(input_dim=200, output_dim=200,
    init='normal',
    activation='tanh'))
for i in range(1, self.stacked_layers -1):
    self.nli_model.add(Dense(input_dim=200,
        output_dim=200, init='normal',
        activation='tanh'))

self.nli_model.add(Dense(input_dim=200,
    output_dim=3, init='normal', activation='tanh'))

# 3 way softmax (entail, neutral, contradiction)
self.nli_model.add(Dense(3, init='uniform'))
self.nli_model.add(Activation('softmax'))
```

As we can see, Keras is a fantastic library, that helps implementing Neural Networks.

The last snippet presented is used to compile and run our model. We can see that by using *fit*, we train our model, and with *evaluate and predict*, we either test our data with our test set, or we get predictions for our test set that later we can use to evaluate. The later is the one used to compute the accuracy, specially because we get information about every single test, and we can study the mistakes easily, but numerically has the same results as evaluate.

```
self.nli_model.compile(loss='categorical_crossentropy',
    optimizer='rmsprop')

if LOAD_W:
    print('loading weights...')
    self.nli_model.load_weights(self.weights_path)
"""
X, Y contain training data while Y contains the
    labels related to the data
X_t, Y_t, contain the test data and labels.
"""
history = self.nli_model.fit(X, Y,
    batch_size=64, nb_epoch=1, verbose=1,
    sample_weight=None, show_accuracy=True)
score = self.nli_model.evaluate(X_t, Y_t,
    batch_size=64, show_accuracy=True, verbose=1)
predictions = self.nli_model.predict(X_t, batch_size=64,
    verbose=1)
```

Appendix B

Previous Datasets

Previously to the paper [2], there were two other datasets built to solve this task, the FraCaS, and the most known Recognizing Textual Entailment (RTE) competition. Both datasets try to achieve the same goal as the dataset introduced by [2].

- FraCaS was introduced in 1996 and contains a set of 346 problems. The dataset is diferent as the one in [2], there is no premise, hypothesis, label setup, and instead there is a setup with premises, questions regarding the premises, and then the answer of the questions, acting as hypothesis.
  - One example of the FraCaS dataset
    - *Premise 1* John went to Paris by car.
    - *Premise 2* Bill by train.
    - *Question* Did Bill go to Paris by train?
    - *Hypothesis* Bill went to Paris by train.
  - For extra information in the dataset check [13]
- RTE was first introduced in [5]. The RTE challenge resembles much more as the dataset released by [2], in which we have a pair of sentences, a premise and a hypothesis, and a label, and our model needs to guess whether the hypothesis can be inferred from the premise. The RTE data set specially differs in the length of the premise, where in some cases, instead of being a sentences, we can even have a short text as a premise. In the RTE contest of 2008, the average length of the premise was about 39 words, meanwhile in [2] it has an average of 14.
  - The results in successive RTE challenges, have been successively incremented up to an 80% accuracy. It is also worth saying that during the first competition the best accuracy achieved in the dataset was of only 58.6% by Glickman et al. In successive competitions, between 2006 and 2008, the average accuracy reported is ranges between 60-62%.

To put things in perspective, the average accuracy on the RTE task for humans is between 90-95%.

Bibliography

[1] D. Bahdanau, K. Cho, and Y. Bengio. Neural Machine Translation by Jointly Learning to Align and Translate. https://arxiv.org/pdf/1409.0473v7.pdf, Sept. 2014.

[2] S. R. Bowman, G. Angeli, C. Potts, and C. D. Manning. A large annotated corpus for learning natural language inference. http://nlp.stanford.edu/pubs/snli_paper.pdf, 2015.

[3] D. Britz. Recurrent neural networks part 1, introduction to rnns. http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/, 2015.

[4] F. Chollet. Keras. https://github.com/fchollet/keras, 2015.

[5] I. Dagan, O. Glickman, and B. Magnini. The pascal recognising textual entailment challenge. http://u.cs.biu.ac.il/~dagan/publications/RTEChallenge.pdf, 2005.

[6] Y. Goldberg. The unreasonable effectiveness of character-level language models. http://nbviewer.jupyter.org/gist/yoavg/d76121dfde2618422139, 2015.

[7] J. Guo. Backpropagation through time. http://ir.hit.edu.cn/~jguo/docs/notes/bptt.pdf, 2013.

[8] G. Hinton. Learning distributed representations of concepts. http://www.cogsci.ucsd.edu/~ajyu/Teaching/Cogs202_sp12/Readings/hinton86.pdf, 1986.

[9] A. Ivakhnenko. *Cybernetic Predicting Devices*. 1965.

[10] S. S. W. Jozefowicz, Vinyals. Exploring the limits of language modeling. https://arxiv.org/pdf/1602.02410.pdf, 2016.

[11] A. Karpathy. The unreasonable effectiveness of recurrent neural networks. http://karpathy.github.io/2015/05/21/rnn-effectiveness/, 2015.

[12] Y. Liu, C. Sun, L. Lin, and X. Wang. Learning natural language inference using bidirectional lstm model and inner-attention. https://arxiv.org/pdf/1605.09090v1.pdf, 2016.

[13] B. MacCartney. The fracas textual inference problem set. http://www-nlp.stanford.edu/~wcmac/downloads/fracas.xml, 2005.

[14] B. MacCartney. Natural language inference. http://nlp.stanford.edu/~wcmac/papers/nli-diss.pdf, 2009.

[15] J. Makin. Backpropagation. https://inst.eecs.berkeley.edu/~cs182/sp06/notes/backprop.pdf, 2006.

[16] W. McKinney. pandas: a foundational python library for data analysis and statistics. http://pandas.pydata.org/index.html.

[17] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. https://arxiv.org/pdf/1310.4546v1.pdf, 2013.

[18] C. Olah. Understanding lstm networks. http://colah.github.io/posts/2015-08-Understanding-LSTMs/, 2015.

[19] R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks. http://www.jmlr.org/proceedings/papers/v28/pascanu13.pdf, 2013.

[20] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. http://www.aclweb.org/anthology/D14-1162, 2014.

[21] T. Rocktaschel, G. Edward, and K. e. a. Moritz Hermann. Reasoning about entailment with neural attention. http://arxiv.org/pdf/1509.06664v4.pdf, 2015.

[22] J. S. Sepp Hochreiter. Long short-term memory. http://deeplearning.cs.cmu.edu/pdfs/Hochreiter97_lstm.pdf, 1997.

[23] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. http://arxiv.org/abs/1605.02688, May 2016.

[24] L. van der Maaten and G. Hinton. Visualizing data using t-sne. https://lvdmaaten.github.io/publications/papers/JMLR_2008.pdf, 2008.

[25] S. van der Walt, S. C. Colbert, and G. Varoquaux. The numpy array: A structure for efficient numerical computation. https://arxiv.org/pdf/1102.1523.pdf, March 2011.

[26] P. J. WERBOS. Backpropagation through time: What it does and how to do it. http://deeplearning.cs.cmu.edu/pdfs/Werbos.backprop.pdf, 1990.

Master thesis filing card

Student: David Torrejon Moya

Title: Natural Language Inference with Recurrent Neural Networks

Dutch title: Natural Language Inference with Recurrent Neural Networks

UDC: 681.3*I20

Abstract:

This thesis presents a new method of studying Inference in Natural Language using Recurrent Neural Networks and an study of the possible limitations of the task and its current approaches. The new method presented is based on breaking the symmetry that any Natural Language Inference task is entailed to.

Thesis submitted for the degree of Master of Science in Artificial Intelligence, option Engineering and Computer Science

Thesis supervisor: Prof. dr. Marie-Francine Moens

Assessors: Prof. dr. Ir. Hendrik Blockeel,
Ir. Geert Heyman

Mentor: Ir. Geert Heyman