Neural part-of-speech tagging

Bachelor’s Degree in Computer Engineering

Bachelor’s Thesis

Author:
Francisco de Borja Valero

Supervisors:
Juan Antonio Pérez Ortiz
Felipe Sánchez Martínez

November 2018
**Resumen**

En este proyecto proponemos la implementación de un sistema desambiguidor de categorías léxicas utilizando redes neuronales recurrentes. Para ello, inicialmente estudiamos los fundamentos teóricos de este tipo de redes neuronales. Posteriormente, proponemos tres arquitecturas diferentes para solventar la desambiguación de palabras ambiguas. Finalmente, obtenemos un sistema capaz de desambiguar con un 93.5% de acierto total y con un 83.2% de acierto en palabras ambiguas en la sección del Wall Street Journal perteneciente al corpus del Penn Treebank.

**Abstract**

In this work, we propose the implementation of a part-of-speech tagging system using recurrent neural networks. For that purpose, initially we study the theoretical fundamentals of that kind of neural networks. Next, we propose three different architectures in order to disambiguate ambiguous words. Finally, we achieve a system able to disambiguate with a 93.5% total accuracy and with 83.2% accuracy on ambiguous words with the section of Wall Street Journal that belongs to the Penn Treebank corpus.
## Contents

1. **Introduction** .......................................................... 1
   1.1. Overview .................................................................. 1
   1.2. Ambiguity classes .................................................. 2
   1.3. Outline ................................................................... 3

2. **Background** ................................................................. 5
   2.1. Neural Networks ..................................................... 5
       2.1.1. Activation functions ......................................... 7
       2.1.2. Loss functions ............................................... 8
   2.2. Training a Neural Network ....................................... 9
       2.2.1. Mini-batches .................................................. 10
       2.2.2. Dropout ....................................................... 11
       2.2.3. Early Stopping ............................................... 12
   2.3. Recurrent Neural Network ....................................... 13
       2.3.1. Vanishing gradient .......................................... 14
       2.3.2. Long Short Term Memory ................................. 15
       2.3.3. Gated Recurrent Unit ...................................... 16
       2.3.4. Bidirectional Recurrent Neural Networks .......... 17
       2.3.5. Stacked Recurrent Neural Networks ................. 17
       2.3.6. Variable Sequence Lengths ............................... 18
       2.3.7. Sequence to sequence ...................................... 19
       2.3.8. Beam Search ................................................. 20
       2.3.9. Attention ........................................................ 21

3. **Material and Methods** .............................................. 25
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1. Introduction</td>
<td>25</td>
</tr>
<tr>
<td>3.2. Framework</td>
<td>25</td>
</tr>
<tr>
<td>3.2.1. TensorFlow</td>
<td>26</td>
</tr>
<tr>
<td>3.3. Dataset</td>
<td>26</td>
</tr>
<tr>
<td>3.3.1. Penn Treebank - Wall Street Journal</td>
<td>26</td>
</tr>
<tr>
<td>3.4. Equipment</td>
<td>27</td>
</tr>
<tr>
<td>4. Architectures</td>
<td>29</td>
</tr>
<tr>
<td>4.1. Encode ambiguity classes</td>
<td>29</td>
</tr>
<tr>
<td>4.2. Architectures</td>
<td>30</td>
</tr>
<tr>
<td>4.2.1. RNN Architecture</td>
<td>30</td>
</tr>
<tr>
<td>4.2.2. Architecture in two phases</td>
<td>30</td>
</tr>
<tr>
<td>4.2.3. Seq2Seq with Attention Architecture</td>
<td>31</td>
</tr>
<tr>
<td>5. Experiments</td>
<td>33</td>
</tr>
<tr>
<td>5.1. Naive baselines</td>
<td>33</td>
</tr>
<tr>
<td>5.1.1. Tag level</td>
<td>33</td>
</tr>
<tr>
<td>5.1.2. Ambiguity level</td>
<td>34</td>
</tr>
<tr>
<td>5.1.3. Word level</td>
<td>34</td>
</tr>
<tr>
<td>5.2. Synthetic dataset</td>
<td>34</td>
</tr>
<tr>
<td>5.2.1. Automaton with backward dependencies</td>
<td>34</td>
</tr>
<tr>
<td>5.2.2. Automaton with forward dependencies</td>
<td>35</td>
</tr>
<tr>
<td>5.2.3. Automaton without dependencies</td>
<td>35</td>
</tr>
<tr>
<td>5.2.4. Complex automaton</td>
<td>36</td>
</tr>
<tr>
<td>5.2.5. Results</td>
<td>36</td>
</tr>
<tr>
<td>5.3. Wall Street Journal Dataset</td>
<td>40</td>
</tr>
<tr>
<td>5.3.1. Preprocessing</td>
<td>40</td>
</tr>
<tr>
<td>5.3.2. Results</td>
<td>41</td>
</tr>
<tr>
<td>6. Conclusions</td>
<td>47</td>
</tr>
<tr>
<td>6.1. Conclusion</td>
<td>47</td>
</tr>
<tr>
<td>6.2. Highlights</td>
<td>47</td>
</tr>
<tr>
<td>6.3. Future work</td>
<td>48</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Example of a MLP</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Activation functions</td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>Example of a MLP with dropout</td>
<td>12</td>
</tr>
<tr>
<td>2.4</td>
<td>Example of early stopping</td>
<td>13</td>
</tr>
<tr>
<td>2.5</td>
<td>Example of an RNN unfolded in time</td>
<td>14</td>
</tr>
<tr>
<td>2.6</td>
<td>Example of a bidirectional RNN</td>
<td>18</td>
</tr>
<tr>
<td>2.7</td>
<td>Example of an unidirectional RNN with 3 stacked layers</td>
<td>18</td>
</tr>
<tr>
<td>2.8</td>
<td>Example of a Seq2Seq architecture</td>
<td>20</td>
</tr>
<tr>
<td>2.9</td>
<td>Example of a beam search with beam width of 2</td>
<td>21</td>
</tr>
<tr>
<td>2.10</td>
<td>Example of an attention matrix</td>
<td>23</td>
</tr>
<tr>
<td>4.1</td>
<td>RNN architecture</td>
<td>30</td>
</tr>
<tr>
<td>4.2</td>
<td>Architecture in two phases</td>
<td>31</td>
</tr>
<tr>
<td>4.3</td>
<td>Seq2Seq with attention architecture</td>
<td>32</td>
</tr>
<tr>
<td>5.1</td>
<td>Automaton with backward dependencies</td>
<td>35</td>
</tr>
<tr>
<td>5.2</td>
<td>Automaton with forward dependencies</td>
<td>35</td>
</tr>
<tr>
<td>5.3</td>
<td>Automaton without dependencies</td>
<td>36</td>
</tr>
<tr>
<td>5.4</td>
<td>Toy language automata</td>
<td>36</td>
</tr>
<tr>
<td>5.5</td>
<td>Results of the architecture in two phases predicting the next ambiguity class using the synthetic dataset</td>
<td>37</td>
</tr>
<tr>
<td>5.6</td>
<td>Results of the architecture in two phases predicting PoS tag using the synthetic dataset</td>
<td>38</td>
</tr>
<tr>
<td>5.7</td>
<td>Results of the RNN architecture on the synthetic dataset</td>
<td>38</td>
</tr>
<tr>
<td>5.8</td>
<td>Results of the Seq2seq architecture on the synthetic dataset</td>
<td>39</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>5.9</td>
<td>Distributions of unambiguous, ambiguous and unknown words in the WSJ dataset, before (BF) and after (AF) the filter</td>
<td>42</td>
</tr>
<tr>
<td>5.10</td>
<td>Results of the architecture in two phases on the first phase using the WSJ dataset</td>
<td>44</td>
</tr>
<tr>
<td>5.11</td>
<td>Results of the architecture in two phases predicting PoS tag using the WSJ dataset</td>
<td>45</td>
</tr>
<tr>
<td>5.12</td>
<td>Results of the RNN architecture on the WSJ dataset</td>
<td>45</td>
</tr>
<tr>
<td>5.13</td>
<td>Results of the Seq2Seq architecture on the WSJ dataset</td>
<td>46</td>
</tr>
</tbody>
</table>
# List of Tables

3.1. PoS tag in the WSJ dataset corresponding to lexical categories. 27
3.2. PoS tags in the WSJ dataset corresponding to punctuation marks. 27
3.3. Characteristics of the GCP instance used. 28
3.4. Software requirements. 28

4.1. Relation between ambiguity class (rows) and PoS tags (columns). 30

5.1. NN setup used with the synthetic dataset. 37
5.2. Results of predicting the next ambiguity class with the synthetic dataset. 39
5.3. Results of predicting the PoS tag with the synthetic dataset. 39
5.4. Information about the sentences in the WSJ dataset, before (BF) and after (AF) the filter. 41
5.5. Information about words in the WSJ dataset, before (BF) and after (AF) the filter. 41
5.6. Baselines results with the WSJ dataset. 42
5.7. NN setup used with the WSJ dataset. 42
5.8. Results of predicting next ambiguity class on the WSJ dataset. 43
5.9. Results of predicting PoS tag on the WSJ dataset. 43
Acronyms

**BPTT** backpropagation through time.

**CUDA** Compute Unified Device Architecture.

**cuDNN** CUDA Deep Neural Network Library.

**DL** deep learning.

**GCP** Google Cloud Platform.

**GPU** graphics processing unit.

**GRU** gated recurrent unit.

**HMM** hidden Markov model.

**IE** information extraction.

**LSTM** long short term memory.

**MEMM** maximum entropy Markov model.

**ML** machine learning.

**MLP** multilayer perceptron.

**NER** name entity recognition.

**NLP** natural language processing.

**NLTK** natural language toolkit.
NN  neural network.

PoS  part-of-speech.

ReLU  rectifier linear unit.

RNN  recurrent neural network.

Seq2Seq  sequence to sequence.

SGD  stochastic gradient descent.

WSJ  Wall Street Journal.
Chapter 1

Introduction

This first chapter introduces the main topic of this work. First, we present an overview of the problem and comment on the approaches used in the past to tackle part-of-speech tagging (see section 1.1). Next, we describe ambiguity classes as a method to represent the words (see section 1.2). Finally, we give the structure of the project (see section 1.3).

1.1. Overview

Part-of-speech (PoS) tagging is the task of assigning a PoS tag (a lexical category) to each word in a sentence. This task belongs to the natural language processing (NLP) field, and provides very useful information to other NLP tasks, such as information extraction (IE) or name entity recognition (NER), among others [13].

In a natural language like English, a particular word can be assigned different PoS tags, depending on its context. Ambiguous words are frequently found in a sentence; about 1/3 of the words in running texts are ambiguous. The next two sentences: “My sister can speak three languages” and “I have a can of beer”, contain the same word “can”, but having different lexical categories. In the first one, “can” is a verb and in the second one is a noun; therefore in English the word “can” is ambiguous. The aim of a PoS tagger is, given a source sentence, disambiguate those words which are ambiguous by taking into account the context in which they appear.

First PoS tagging systems were rule-based [4]. Over the years, the rules of those systems became more complex [7], and on top of that, appeared stochastic models like
hidden Markov models (HMM) and maximum entropy Markov models (MEMM), which outperform the results of rule-based systems. At the end of the nineties, neural networks (NN) were emerging, and were used by some researchers obtaining interesting results; however the high computational cost of training them made the research community to focus mainly on stochastic models rather than NN models.

Nowadays, the use of this NN models are arising again; unlike before, today we dispose of high performance computers. In recent years, many researches in NN models has appeared. In this work we propose to retake those old architectures and update them to current models and programming frameworks. Moreover, new architectures are proposed.

1.2. Ambiguity classes

Usually, the input to PoS tagging is not an identifier or a representation of a word; typically it is a word class. In this specific problem, the word class is an ambiguity class that is the set of all PoS tags that a given word can be assigned. In order to get the ambiguity classes, first we have to process the relation between the words and their PoS tags in a training corpus. In this corpus, each word has assign its respective PoS tag (disambiguate corpus). In the training corpus, the same word can be found with different PoS tags. Therefore, the words that share the same set of PoS tags belong to the same ambiguity class as “the”:{DT} and “an”:{DT}. Also, when this ambiguity class has more than one PoS tag as “telecommunications”:{NN, NNS}; it is said to be ambiguous.

The approaches used above are useful when a given word has been seen in a training corpus. But there is a problem when a word is unknown: it has not been seen in the training corpus. The basic approach is to create an ambiguity class for the unknown words containing all PoS tags available in the tagset. In this way, the tagger decides which is the most promising PoS tag. An alternative to this approach consist of including in the ambiguity class for unknown words only those PoS tags that belong to open categories: nouns, verbs, adjectives and adverbs; because normally, when a new word is added to the vocabulary of a language it belongs to one of these open categories.
Another approach used to reduce the number of possible tags for unknown words consist of using a morphological guesser \cite{19} before running any tagging method. Also there are more complex methods taking into account the probabilities of the suffixes and ending of words as other features to restrict the set of PoS tags. Another approach is to count the occurrences of words along the training set, splitting them in regular (frequent) and rare word (less than five occurrences); in this way unknown words are treated as if they were rare words \cite{27}.

1.3. Outline

This dissertation is structured as follows. Chapter 2 provides the theoretical background required to understand recurrent neural networks (RNN). Chapter 3 shows the framework and resources used in this project. Chapter 4 explains the way of encoding ambiguity classes and the proposed architectures. Chapter 5 describes the naive baselines, shows results and other aspects related with experimentation. Finally, chapter 6 presents the conclusions of the project and future work.
Chapter 2

Background

In this second chapter, the online course “Deep Learning Specialization” by Andre Ng has been fundamental in order to clearly explain the basics of deep learning (DL): DL is a subset of machine learning (ML) whose models have several layers of neurons capable of learning tasks from data.

2.1. Neural Networks

Recurrent neural networks (RNN) are a neural model very popular for dealing with tasks involving sequence processing. Before explaining them, it is necessary to introduce first the feedforward model. These models tries to imitate the behaviour of neurons in the human brain.

Figure 2.1 shows a multilayer perceptron (MLP) which is constituted by an input layer with two neurons that represent the features of the input data. It is followed by two hidden layers with three and four neurons respectively, and an output layer with only one neuron whose activation value is the prediction \( \hat{y} \). A neural layer is represented by a matrix, whose elements are parameters, also know as weights. The shape of the weight matrix in layer \( l \) is \( W^l \in \mathbb{R}^{H^l \times H^{l-1}} \), where \( H^l \) and \( H^{l-1} \) are the number of neurons in the current and previous layer respectively. In all layers additional weights exist whose input is always 1. These weights are known as bias \( b^l \in \mathbb{R}^{H^l} \) and its size corresponds to the number of neurons in the corresponding layer. Its contribution is crucial for the training (see section 2.2), because it allows to shift the activation output

\[ \text{https://www.coursera.org/specializations/deep-learning} \]
in one direction or another for obtaining a successful training.

Equation below shows the operation performed by layer \( l \), where \( W^l \) and \( b^l \) are the weights of the current layer and \( z^{l-1} \) is the activation output of the previous layer. Linear combination of previous parameters are introduced in a nonlinear activation function \( f \). In section 2.1.1 some activations functions used by neural network layers will be explained with more detail.

\[
z^l = f(W^l z^{l-1} + b^l) \tag{2.1}
\]

In order to maximize the high performance of existing computers, it is very common to use vectorization principles, thus replacing loops with matrix operations; where \( m \) samples are processed simultaneously instead one. Therefore, the shape of input data will be \( X \in \mathbb{R}^{H^0 \times m} \), where \( H^0 \) is the number of features of the input data, and the shape of the expected output will be \( \hat{y} \in \mathbb{R}^{H^L \times m} \), where \( H^L \) is the number of neurons in the output layer.
2.1.1. Activation functions

Feedforward computation (see section 2.1) allows us to solve non-trivial pattern matching problems thanks to the use of nonlinear functions. In this section the most popular activation functions are described. They are applied to the output of the neural network layers; depending on the output, it will be more recommendable to use one or another.

The output value of the sigmoid is in \([0, 1]\); normally it is used in the output layer in problems like binary classification or regression where predictions must be in that domain. Rectifier linear unit (ReLU) is commonly used in hidden layer because of its special nonlinearity, as its output is always equal or higher than zero, therefore computationally it is simple. The last popular activation function is hyperbolic tangent (tanh), whose output is in \([-1, 1]\). Normally, it is used in hidden layers or when a prediction is necessary in that domain. Figure 2.2 shows the behaviour of the three nonlinear functions commented and their analytical forms are shown in the next equations:

\[
\sigma(x) = \frac{1}{1 + \exp(-z)} \quad (2.2)
\]

\[
\tanh(x) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)} \quad (2.3)
\]
\text{ReLU}(x) = \max(0, x) \quad (2.4)

None of the previous functions can be used in the output layer of a multiclass classification problems. For this reason the \textit{softmax} function is considered, which normalizes the output values. Then, it assigns a probability for each neuron, so that when the activation value of a neuron with respect to the others is high, it will be assigned a higher probability:

\[
\text{softmax}(x)_j = \frac{\exp(x_j)}{\sum_{i=0}^{C} \exp(x_i)} \quad (2.5)
\]

Parameter \(j\) corresponds to the index of output neuron. \(C\) is the size of the last layer which in classification tasks corresponds to the number of classes.

### 2.1.2. Loss functions

Loss function is used to compute the difference between the network prediction \(\hat{y}\) and the desired prediction \(y\). When the loss is small, it indicates that prediction is close to the expected one. There are many loss functions. Depending on the problem we want to solve one will be more recommendable than others. In this work, we will solve a sequence labelling problem. For this reason we will use cross entropy. This loss function is commonly used in classification problems.

In a binary classification problem, the output layer of a NN contains only one neuron, whose activation function is \textit{sigmoid} (see equation 2.2). When its output is higher than a threshold, normally 0.5, the prediction will be assigned one of the categories and the other otherwise. The cross entropy loss used in that case is:

\[
\mathcal{L}(\hat{y}, y) = - (y \log(\hat{y}) + (1-y) \log(1 - \hat{y})) \quad (2.6)
\]

In a multiclass classification problem, the last layer contains \(C\) neurons, where \(C\) is the total number of classes and each neuron corresponds to a different class. Also, the activation function is the \textit{softmax} in equation 2.5, therefore, each output node will be assigned a probability. The desired prediction \(y \in \mathbb{Z}_C\) is encoded using \textit{one-hot} representation, where the value of the column associated to the class being represented is set to one and the rest to zero. Then, the output node with higher probability will
be the more promising class. The cross entropy loss used in that case is:

$$L(\hat{y}, y) = -\sum_{i=0}^{C} y_i \log(\hat{y}_i)$$

As with feedforward computation (see section 2.1), where vectorization is used in order to compute the prediction for several samples in one pass, the computation of the cost is done in the same way. Therefore, the shape of the desired prediction would be $Y \in \mathbb{Z}^{C \times m}$, where $m$ is the number of samples, and the shape of the prediction $\hat{Y}$ is the same, but being real values $\mathbb{R}$. The equation below shows that the cost function is an average loss over all samples:

$$\text{Cost}(\hat{Y}, Y) = \frac{1}{m} \sum_{i=0}^{m} L(\hat{y}_i, y_i)$$

### 2.2. Training a Neural Network

Once explained the feedforward computation in section 2.1, where weights are involved in operations in each layer, these operations contribute to compute a final prediction $\hat{y}$, given an input data $X$. Ideally, whatever input data is introduced in the NN, it must always predict an output very close to the expected one. Therefore, the value of those weights are the key for a correct performance.

The weights of a feedforward model are initialized randomly. Common behaviour is that initial weights produce bad predictions. For this reason, the backpropagation algorithm appeared. This algorithm adjusts the weights of the model taking into account the loss produced (see section 2.1.2). This loss will be propagated from output layer (where it is computed) until first layer (where the computation begins). The aim of the backpropagation algorithm is to minimize the loss. The gradient descent optimizer is one of the methods used by the backpropagation algorithm for fitting the weights $W$, once their respective gradients $\nabla$ had been computed. Therefore, the chain rule is essential, in order to compute partial derivatives between the loss $L$ and all weights $W$ along the distinct neural layers that contribute to perform the prediction. The obtained gradients will be used with a learning rate $\lambda$ for fitting the correspondent weights through gradient descent:
\[ \Delta W = -\lambda \frac{\partial L}{\partial W} \] (2.9)

Nowadays, most machine learning (ML) frameworks automatically compute approximation of the gradients thanks to the use of automatic differentiation [29], [11].

Learning rate $\lambda$ is an hyperparameter and depending on its value the effect in the networks will be different. If its value is high, local minimum would be easily avoided, since changes in the weights are higher, but it does not converge, because it will be unstable. However, with small value, learning would be slow, but it can converge better. A technique exists called learning rate decay where value of $\lambda$ is inversely proportional to the number of training steps.

Currently, Adam [14] is the most popular optimizer. It computes an adaptive learning rate for each parameter. Actually, it is a combination of two previous optimizers: gradient descent with momentum and RMSprop [24]. Gradient descent with momentum computes the gradients using exponentially decaying average of past gradients. RMSprop performs the same computation but using squared gradients. Adam also incorporates a bias correction mechanism in order to counteract the initial steps where the bias value is close to zero.

2.2.1. Mini-batches

As seen before, vectorization is used to compute several predictions in one pass. Another important reason is that the average loss of several samples when fitting parameters allows for better generalization, thus avoiding overfitting. Overfitting happens when parameters of the model produce good performance in training set and bad ones with new data, producing the memorization of the training set. If weights are adjusted after every sample, the loss of the last samples will have higher influence in the model than the loss of the first samples.

There are three main ways of organizing samples in batches. Stochastic gradient descent (SGD) or online learning only uses one sample per batch. Batch gradient descent, where all samples are only in one batch; this is only useful when the number of training samples is very small. For this reason mini-batch gradient descent is used as usually the number of training samples is vast. This third method divides the training
samples in groups of $m$ samples.

In batch gradient descent at each step the cost decreases, but in mini-batch gradient descent small oscillations happen at each step; this is completely normal because in each batch the set of samples is different, unlike in batch gradient descent in which each batch contains the same data. However, in mini-batch gradient descent the cost along all epochs in the training is decreasing in the similar way as occurs in batch gradient descent.

The most appropriate training would have all samples in one batch, but nowadays, due to the vast amount of information available, it is impracticable. For this reason, the correct option is mini-batch gradient descent, because SGD would require much more time, since it fits the parameters taking into account only one sample per batch instead $m$.

In order to reduce the cost, it is necessary to consider all training set several times; a single pass including all the samples in the training set is know as an epoch.

### 2.2.2. Dropout

Dropout \[9\] is a regularization technique that tries to avoid the overfitting of the model. This technique works by deactivating different neurons in each time step, in order to achieve better generalization. A keep probability ($p_k$) of each layer must be indicated.

In figure 2.3 the input layer and output layer do not use dropout, therefore all neurons in both layers are activated, but this does not happen in both hidden layers with $p_k$ of 0.66 and 0.5 respectively. The value of non-activated neurons will be substituted by zero. For this reason in figure 2.3 no connections exist involving the deactivated neurons. Finally, the activation output of each layer will take into account the corresponding $p_k$:

\[
z_{\text{dropout}}[j] = \begin{cases} 
z'[j], & \text{with probability } p_k' \\
0, & \text{with probability } 1 - p_k' 
\end{cases} \tag{2.10}
\]
2.2.3. Early Stopping

*Early stopping* \[\textsuperscript{32}\] is another regularization technique for avoiding overfitting. It is based on using two datasets: one for training the network and another one which to decide when to stop. After each epoch of the training algorithm, the development set will be evaluated; if the cost obtained is smaller than the previous better cost obtained, that model will be saved. However, if the cost is worse in several consecutive epochs the training stops.

As can been see in figure 2.4, after epoch 400 the model is overfitting: the cost in the training set is stable, but on the development set the cost increases. For this reason, the training should be stopped at epoch 400, and the weights obtained at this epoch be used for testing.
2.3. Recurrent Neural Network

Recurrent neural network (RNN) is a neural model created for dealing with sequence data where at each time step a new input is introduced and the output for the current input may change depending on its context. This type of network has hidden states, weights and bias shared across different time steps.

Several variants of sequential information exist depending on the length of the input and output sequences. But in this work, we are only going to use many-to-many RNN where the input length $T_x$ is the same as output length $T_y$, $T_x = T_y$. In addition, there exist many-to-many RNNs in which the length of the input and the output sequence are different, as in speech recognition [10] or neural machine translation [11], many-to-one as in sentiment analysis [28] or text classification [28], and one-to-many as in image captioning [20].

Figure 2.5 shows an RNN unfolded in time in which the size of the hidden estate is 3, $S = 3$. In a RNN at each time step $t$ is computed a prediction $y_t$ that corresponds to the input $x_t$ fed in it. The great contribution offered by this sequence model is the use of a hidden state shared along all time steps. This hidden state $s_t$ allows us to capture dependencies along time, generating different output for the same input taking the context into account. In each time step a hidden state is computed that is sent to the next time step $t+1$.

In the first time step of equation 2.11 a previous hidden state does not exist; for this reason the initial hidden state of RNN is initialized with zeros.

This first RNN cell, computes the current hidden state $s_t \in \mathbb{R}^S$ in a similar way
to that of the NN layer (see equation 2.1), but considering two input sources instead of one, the previous hidden state \( s_{t-1} \) and the current input \( x_t \in \mathbb{R}^D \) where \( D \) is the number of the input features, therefore it uses two weights one for the current input \( W_{ss} \in \mathbb{R}^{S \times S} \) and other for the previous hidden state \( W_{sx} \in \mathbb{R}^{S \times D} \). After that, this current hidden state \( s_t \) will be used, in order to compute the current output \( \hat{y}_t \) in which another weights are used \( W_{\hat{y}s} \in \mathbb{R}^{C \times S} \).

\[
\begin{align*}
    s_t &= \begin{cases} 
        0, & \text{if } t = 0 \\
        g(W_{ss}s_{t-1} + W_{sx}x_t + b_s), & \text{otherwise}
    \end{cases} \\
    \hat{y}_t &= f(W_{\hat{y}s}s_t + b_{\hat{y}})
\end{align*}
\]

As presented in section 2.2 this sequence model is trained in the same way, but more parameters to optimize exist, as can be seen in equations 2.11 and 2.12. The training algorithm used is backpropagation through time (BPTT) [30]. In section 2.3.1 we explain the difficulties found by this algorithm when using RNN cells [21, 3].

### 2.3.1. Vanishing gradient

When the last prediction \( \hat{y}_T \) of input sequence \( \mathbf{x} \) is calculated, loss is computed. This loss is propagated backwards in time over all time steps in the RNN:

\[
\frac{\partial L(y_t, \hat{y}_t)}{\partial W_{t-k}}
\]
$W_{t-k}$ represents the weights of an RNN unfolded in time, where $k$ is the number of the backward steps. When the amount of backwards steps is high, the value of the computed gradient in the weight of the corresponding time step tends to zero, therefore the contribution of it to the error is irrelevant, preventing the model from captioning long dependencies. That is very common in long sequences. This problem is known as vanishing gradient.

In order to solve this adversity, two neural models were proposed, first long short term memory (LSTM) \cite{12} and later gated recurrent unit (GRU) \cite{1} both avoiding vanishing gradient problem. In sections \ref{sec:2.3.2} and \ref{sec:2.3.3} respectively we will explain both models with more detail.

\subsection{Long Short Term Memory}

LSTM \cite{12} is an extended type of neuron with the capacity to store information for a long time until it is useful. It includes a gate mechanism that forces to compute gradients whose value are very close to one, allowing a better propagation of the loss in very long sequences. In this way, it avoids the vanishing gradient problem and captures long distance dependencies. Therefore, the gate mechanism is a crucial improvement.

This model also includes a memory cell $c$. The candidate memory cell $\tilde{c}$ is computed as $s_t$ (see equation \ref{eq:2.11}) in the RNN. The gate mechanism is composed of three gates whose output values are in $[0, 1]$, allowing to maintain information through several steps or reset the memory cell, if necessary. The input gate $\Gamma_i$ regulates how much the new input change the memory cell value, the forget gate $\Gamma_f$ regulate how much the previous memory cell value is preserved and the output gate $\Gamma_o$ regulate how much information influences the output:

\begin{align*}
\tilde{c} &= \tanh(W_{\tilde{c}s} s_{t-1} + W_{\tilde{c}x} x_t + b_{\tilde{c}}) \quad (2.14) \\
\Gamma_i &= \sigma(W_{is} s_{t-1} + W_{ix} x_t + b_{i}) \quad (2.15) \\
\Gamma_f &= \sigma(W_{fs} s_{t-1} + W_{fx} x_t + b_{f}) \quad (2.16)
\end{align*}
\[ \Gamma_o = \sigma(W_{os}s_{t-1} + W_{ox}x_t + b_o) \] (2.17)

Current memory cell \( c_t \) is influenced by the new information added to the cell and by the degree to which the previous value is deleted. For obtaining that, two element-wise multiplications are computed (Hadamard product), one between the input gate \( \Gamma_i \) and the candidate memory \( \tilde{c} \) (new memory) and another between the forget gate \( \Gamma_f \) and the previous cell memory \( c_{t-1} \) (old memory); finally both products are added:

\[ c_t = \Gamma_u \ast \tilde{c} + \Gamma_f \ast c_{t-1} \] (2.18)

Finally, in order to compute current output \( s_t \) element-wise multiplication is used between output gate \( \Gamma_o \) and current memory cell value \( c_t \) which previously has been applied a tanh activation function. In this way, the output will be influenced by how much new information is added and how much old information is deleted.

\[ s_t = \Gamma_o \ast \tanh(c_t) \] (2.19)

The prediction at current time \( \hat{y}_t \) is computed using a NN layer (see equation 2.1) whose input is \( s_t \).

### 2.3.3. Gated Recurrent Unit

Time later LSTM, GRU [11] was created. It appeared to attach the problem of vanishing gradient in a similar way. In this type of neuron, the current memory cell \( c_t \) and the current output \( s_t \) are the same, \( c_t = s_t \). GRU uses two gates instead of three. These two gates are the update gate \( \Gamma_u \) and the reset gate \( \Gamma_r \). The gate mechanism and the candidate memory \( \tilde{c} \) are computed as in LSTM.

\[ \tilde{c} = \tanh(W_{\tilde{c}}c_{t-1} + W_{\tilde{c}x}x_t + b_{\tilde{c}}) \] (2.20)

\[ \Gamma_u = \sigma(W_{uc}c_{t-1} + W_{ux}x_t + b_u) \] (2.21)

\[ \Gamma_r = \sigma(W_{rc}c_{t-1} + W_{rx}x_t + b_r) \] (2.22)
However, the way of computing the current memory cell is influenced by how much input information is used and how much the old memory is kept. Both multiplications are element-wise too.

\[ c_t = r_t \odot \tilde{c} + (1 - u_t) \odot c_{t-1} \quad (2.23) \]

This second architecture computes less operations than LSTM, for this reason it is faster in inference mode. However, in some experiment it has been shown in GRU it get worse results [8].

### 2.3.4. Bidirectional Recurrent Neural Networks

In many tasks, bidirectional RNNs have surpassed unidirectional performance, because posterior information may help to better predict the current output, in comparison to using only past information. Normally, one RNN is used that processes the sequence from the first element to the last one. Therefore this RNN goes forward and will be represented as $\overrightarrow{\text{RNN}}$. When the first input of the network is the last element of the sequence and the last one is the first element of the sequence, this second RNN goes backward and will be represented as $\overleftarrow{\text{RNN}}$.

A bidirectional RNN is composed by two RNNs as can be seen in figure 2.6. Each RNN processes the input in a different direction. The output of both RNN generated at each time step are concatenated $[\overrightarrow{s}_t, \overleftarrow{s}_t]$ and this is the prediction of the bidirectional RNN:

\[ \hat{y}_t = g(W_y[\overrightarrow{s}_t, \overleftarrow{s}_t] + b_y) \quad (2.24) \]

### 2.3.5. Stacked Recurrent Neural Networks

Several problems exists whose complexity is very high like machine translation in the NLP field. Architectures with only one layer can not get a good performance. For this reason it is necessary to increment the number of layers in order to learn more abstract features. Figure 2.7 shows a RNN with three stacked layers.

In the first layer, the input at each time step is the corresponding element of the input sequence $x_t$, but in the remaining layers, the input is the hidden state of the
Chapter 2. Background

previous layer at the same time step $s_{l-1}^t$; obviously, the previous hidden state belongs to the same layer $l$ $s_{l-1}^t$:

$$s_l^t = g(W_{sx}s_{l-1}^t + W_{ss}s_{l-1}^t + b_s)$$  \hspace{1cm} (2.25)

2.3.6. Variable Sequence Lengths

In all human languages the lengths of the sentences are different. As has been explained in the previous sections, in order to attain fast computations it is crucial to follow vectorization methods, and therefore we must group sequences in mini-batches.

Figure 2.6: Example of a bidirectional RNN.

Figure 2.7: Example of an unidirectional RNN with 3 stacked layers.
(see section 2.2.1). Since sequences have different length, it is necessary to pad all sequences to have the same length. This length would be the maximum length existing in the corresponding mini-batch. Padding a sequence consists in adding a pad token \(<\text{pad}>\) to the end of the original sequence, until this padded sequence gets the expected length:

$$\mathbf{x}_0 = \begin{bmatrix} x_0^0 & x_0^1 \end{bmatrix}; \quad \mathbf{x}_1 = \begin{bmatrix} x_1^0 & x_1^1 \end{bmatrix}; \quad \mathbf{X} = \begin{bmatrix} x_0^0 & x_1^0 & \text{<pad>} \\ x_0^1 & x_1^1 & x_2^1 \end{bmatrix}$$

However, in order to compute the actual loss \(L_{\text{actual}}\) of the mini-batch, the loss of the padded tokens has to be omitted. For doing that, a mask \(\mathcal{M}\) is used with the same shape of the mini-batch input where zero elements correspond with padded elements:

$$\mathcal{M} = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

The padded input \(\mathbf{X}\) is introduced in a sequence model like RNN, generating a prediction \(\hat{Y}\):

$$\hat{Y} = \text{RNN}(\mathbf{X});$$

The total loss \(L_{\text{total}}\) is computed between the prediction of the RNN and the desired prediction \(Y\). After that, element-wise multiplication between the total loss and the mask is computed in order to obtain the actual loss:

$$L_{\text{total}} = L(\hat{Y}, Y) = \begin{bmatrix} L_0^0 & L_1^0 & L_2^0 \\ L_0^1 & L_1^1 & L_2^1 \end{bmatrix}; \quad L_{\text{actual}} = L_{\text{total}} \ast \mathcal{M} = \begin{bmatrix} L_0^0 & L_1^0 & 0 \\ L_0^1 & L_1^1 & L_2^1 \end{bmatrix}$$

**2.3.7. Sequence to sequence**

Sequence to sequence (Seq2Seq) \[26\] is an encoder-decoder architecture, where input sequence is encoded by an RNN, whose hidden state in the last time step is used as the initial state of an RNN decoder. The decoder predicts the current output \(\hat{y}_t\) given the previous one \(\hat{y}_{t-1}\) and the hidden state. As it does not exist a previous element for the
first prediction, an initial and end element are created for all sequences. Therefore, all output sequences share an initial element \(<\text{sos}>\) that means beginning of the sequence and is the first input introduced in the decoder, and also they share an end element \(<\text{eos}>\) too, that is the last element that must be predicted by the decoder. This architecture allows to tackle problems in which the input and output sequences have different length. Because of that, all elements produced by the decoder that appear after the end element are ignored.

Figure 2.8 shows a Seq2Seq where input sequence size is three and output is four, without taking into account the \(<\text{sos}>\) and \(<\text{eos}>\) elements.

![Figure 2.8: Example of a Seq2Seq architecture.](image)

### 2.3.8. Beam Search

As already discussed in section 2.3.7, the decoder predicts the current output \(\hat{y}_t\) given the previous one \(\hat{y}_{t-1}\) and the hidden state. Therefore, at each time step the network is fed with the more promising class; this is known as the greedy search approach to decoding. The score of an output sequence is the product of all the element probabilities. Therefore, the first element predicted with high probability does not necessarily belong to the output sentence with the highest score. For this reason the beam search technique [31] appeared, where \(n\) most promising classes are taken at each step, generating \(n\) different output sequences instead of one as occurs with the greedy search approach. Also, it is important to take into account the length of the sequence in order to choose the most promising sequence, because normally a long sequence has smaller probability than a shorter one, but a longer one could be a better output
sequence. In order to overcome this issue, the score is normalized taking into account the length of the predicted output too.

Figure 2.9 shows a beam search with a beam width of 2, where the most promising output sequence (BS) is not the expected output produced by the greedy search approach (G).

\[ P(c_0, c_1 | X) = 0.5 \times 0.3 = 0.15 \text{ (G)} \]

\[ P(c_2, c_0 | X) = 0.35 \times 0.6 = 0.21 \text{ (BS)} \]

Figure 2.9: Example of a beam search with beam width of 2.

### 2.3.9. Attention

The problem of Seq2Seq (see section 2.3.7) is that it uses a fixed encoded state for the whole source sequence, and it is very difficult to compress all meaning of the source sentence into a single vector. Attention [2] appeared to overcome this issue. This mechanism generates at each decoding time step a different encoded representation \( c \) (context) of the source sequence. For doing that, it requires to store all the hidden
states $h_i$ generated by the encoder.

$$a(s_{t-1}, h_j) = f(W_{as}s_{t-1} + W_{ah}h_j + b_a) \quad (2.26)$$

$$\alpha_{ij} = \frac{\exp(a(s_{i-1}, h_j))}{\sum_{k=0}^{T_x} \exp(a(s_{i-1}, h_k))} \quad (2.27)$$

$$c_i = \sum_{j=0}^{T_x} \alpha_{ij}h_j \quad (2.28)$$

Equation 2.26 computes the degree of attention between one element of input sequence $x_j$ and next element of the output sequence $\hat{y}_t$. For doing so it uses a NN layer between the corresponding hidden state of the encoder $h_j$ and the previous hidden state of the decoder $s_{t-1}$. This computation will be performed for all hidden states of the input sequence. Then, equation 2.27 computes a scalar value $\alpha_{ij}$ (energy) for each value obtained in equation 2.26; this value indicates how much attention has to be put into $x_j$ to predict $\hat{y}_i$. Finally, variable encoded state is computed in equation 2.28, by simply performing a weighted sum of attention states with the corresponding energy values.

Figure 2.10 uses an input and output sequence generated by the automaton shown in figure 5.4. This automaton contains backward and forward dependencies; therefore, the attention matrix shows an output that requires backward dependencies to predict it correctly like $c_2$ at $t_3$ where it has to put attention into the corresponding current input $a_2$ and the corresponding previous one $a_0$, and the same happens with $c_4$ at $t_5$ but requiring forward dependencies. The remaining outputs only require focusing on the corresponding current input to predict them correctly.
Figure 2.10: Example of an attention matrix.
Chapter 3

Material and Methods

3.1. Introduction

In a DL project there are three resources that are necessary to finish it with success. The first one is related to programming. We need to code complex neural networks easily and reduce as much as possible the time needed to code them and the likelihood of bugs. For this reason, the best option is choosing an open source specialized machine learning (ML) framework, such as TensorFlow (see section 3.2). The second one is data. It is advisable to use publicly available datasets in order to ensure reproducibility and make it easier the comparison with state-of-the-art approaches. In this way, it will be easy to compare them with our own results. The last one is related to the equipment used for training the models. Training deep neural networks requires dedicated hardware (GPUs) which if it is not available makes training very time consuming, which may prevent testing different configurations.

3.2. Framework

Thanks to the DL boom, many open source frameworks specialized in ML like CNTK\[^1\], PyTorch\[^2\], Keras\[^3\] and TensorFlow\[^4\] have appeared.

Prior to the development of this project, I had coded some NN toy models in Keras, and I had to understand the computation graph of TensorFlow, which is the framework  

[^2]: https://pytorch.org
[^3]: https://keras.io
[^4]: https://www.tensorflow.org
internally used by Keras. In this project I have gone deeper in understanding and programming NN in TensorFlow.

### 3.2.1. TensorFlow

TensorFlow [18] is an open-source library for numerical computation which uses data flow graphs. It is developed by Google Brain starting in 2015. The initial aim of this library was to encourage research in ML. The software developed with this library is easily deployed in a variety of hardware platforms like CPUs, GPUs and mobile devices, among others.

In this project, the main learning objective is to acquire the knowledge to develop NN models working with sequences of variable length, like natural language sentences. TensorFlow provides tf.data [5] which allows to easily preprocess a dataset composed of variable-length sequences and tf.contrib.rnn [6] which prototypes RNN in a simple way. TensorFlow also incorporates tf.contrib.seq2seq [7], which allows the development of Seq2Seq architectures with attention mechanism and beam search. For this reasons, TensorFlow is ideal for this work.

### 3.3. Dataset

There are many PoS tagging datasets available, like the Brown [8] corpus; however, there is one corpus that may be considered the standard dataset because it is used in most relevant research on part-of-speech tagging [9]. That dataset is the Penn Treebank [17].

#### 3.3.1. Penn Treebank - Wall Street Journal

The Wall Street Journal (WSJ): WSJ is a collection of English news; is a portion of the Penn Treebank composed of 25 sections (0-24). This dataset contains 45 PoS tags, of which 36 are lexical categories [10] and the rest are punctuation marks. Tables

[1] https://www.tensorflow.org/api/docs/python/tf/data
Table 3.1: PoS tag in the WSJ dataset corresponding to lexical categories.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
<th>Tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC</td>
<td>Coordinating conjunction</td>
<td>CD</td>
<td>Cardinal number</td>
</tr>
<tr>
<td>DT</td>
<td>Determiner</td>
<td>EX</td>
<td>Existential there</td>
</tr>
<tr>
<td>IN</td>
<td>Preposition or subordinating conjunction</td>
<td>FW</td>
<td>Foreign word</td>
</tr>
<tr>
<td>JJ</td>
<td>Adjective</td>
<td>JJR</td>
<td>Adjective, comparative</td>
</tr>
<tr>
<td>JJS</td>
<td>Adjective, superlative</td>
<td>LS</td>
<td>List item marker</td>
</tr>
<tr>
<td>MD</td>
<td>Modal</td>
<td>NN</td>
<td>Noun, singular or mass</td>
</tr>
<tr>
<td>NNS</td>
<td>Noun, plural</td>
<td>NNP</td>
<td>Proper noun, singular</td>
</tr>
<tr>
<td>NNPS</td>
<td>Proper noun, plural</td>
<td>PDT</td>
<td>Predeterminer</td>
</tr>
<tr>
<td>POS</td>
<td>Possessive noun</td>
<td>PRP</td>
<td>Personal pronoun</td>
</tr>
<tr>
<td>PRP$</td>
<td>Possessive pronoun</td>
<td>RB</td>
<td>Adverb</td>
</tr>
<tr>
<td>RBR</td>
<td>Adverb, comparative</td>
<td>RBS</td>
<td>Adverb, superlative</td>
</tr>
<tr>
<td>RP</td>
<td>Particle</td>
<td>SYM</td>
<td>Symbol</td>
</tr>
<tr>
<td>VBZ</td>
<td>Verb, 3rd person singular present</td>
<td>UH</td>
<td>Interjection</td>
</tr>
<tr>
<td>VB</td>
<td>Verb, base form</td>
<td>VBD</td>
<td>Verb, past tense</td>
</tr>
<tr>
<td>VBG</td>
<td>Verb, gerund or present participle</td>
<td>VBN</td>
<td>Verb, past participle</td>
</tr>
<tr>
<td>VBP</td>
<td>Verb, non-3rd person singular present</td>
<td>TO</td>
<td>to</td>
</tr>
<tr>
<td>WDT</td>
<td>Wh-determiner</td>
<td>WP</td>
<td>Wh-pronoun</td>
</tr>
<tr>
<td>WP$</td>
<td>Possessive wh-pronoun</td>
<td>WRB</td>
<td>Wh-adverb</td>
</tr>
</tbody>
</table>

Table 3.2: PoS tags in the WSJ dataset corresponding to punctuation marks.

<table>
<thead>
<tr>
<th>Tag</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>( $</td>
</tr>
<tr>
<td>)</td>
<td>.  #</td>
</tr>
<tr>
<td>:</td>
<td>“ ”</td>
</tr>
</tbody>
</table>

3.4. Equipment

We use the Compute Engine\footnote{https://cloud.google.com/compute/} service of Google Cloud Platform (GCP) for training our neural models. Table 3.3 shows the characteristic of the computing instance used. In order to be able to run TensorFlow in graphics processing units (GPU) it is mandatory to have installed the \textit{CUDA Deep Neural Network Library} (cuDNN) with the \textit{Compute Unified Device Architecture} (CUDA) driver. Therefore, I had to install the software listed in table 3.4.
Natural language toolkit (NLTK) will be used to tokenize the corpora. Each word will be substituted by the correspondent ambiguity class which will be the input to the neural network.

<table>
<thead>
<tr>
<th>Computing Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
</tr>
<tr>
<td>SSD</td>
</tr>
<tr>
<td>RAM</td>
</tr>
<tr>
<td>CPU’s cores</td>
</tr>
<tr>
<td>GPU</td>
</tr>
</tbody>
</table>

Table 3.3: Characteristics of the GCP instance used.

<table>
<thead>
<tr>
<th>Package</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>3.5</td>
</tr>
<tr>
<td>CUDA</td>
<td>9.0</td>
</tr>
<tr>
<td>cuDNN</td>
<td>7.2</td>
</tr>
<tr>
<td>TensorFlow</td>
<td>1.10.1</td>
</tr>
<tr>
<td>NLTK</td>
<td>3.3.0</td>
</tr>
</tbody>
</table>

Table 3.4: Software requirements.
Chapter 4

Architectures

This fourth chapter shows the way of encoding ambiguity classes (see section 4.1) as input to the NN models and describes the NN architectures we have explored (see section 4.2) to tackle the problem described in chapter 1.

4.1. Encode ambiguity classes

In this work we represent the ambiguity classes (see section 1.2) in two different ways. The first one uses one-hot encoding where each ambiguity class identifier is substituted by a vector $X^o \in \mathbb{Z}^A$ where $A$ is the size of the set of ambiguity classes and the element of the column associated to the ambiguity class being represented is set to one and the rest to zero. The second one uses multi-hot encoding where an ambiguity class is represented by a vector $X^m \in \mathbb{Z}^C$ where $C$ is the size of the tagset and the value of the columns associated to the tags in the ambiguity class being represented are set to one, and to zero otherwise. Multi-hot offers a more compact representation of the ambiguity classes than one-hot and also allows to incorporate new words belonging to ambiguity classes not seen in the training corpus.

Table 4.1 shows PoS tags that belongs to each ambiguity class in the automaton of figure 5.4. In this example the number of ambiguity classes is 6 and the size of the tagset is 7. Therefore, the one-hot representation of the ambiguity class $a_3$ would be: $[0,0,0,1,0,0]$; and its multi-hot representation would be: $[1,0,0,1,0,0,0]$. 
Chapter 4. Architectures

4.2. Architectures

We propose to evaluate three NN architectures for our PoS task: the first one uses RNN (see section 4.2.1); the second one is an architecture in two phases that uses RNN in the first phase and feedforward NN in the second phase [22] (see section 4.2.2); the third one is a Seq2Seq architecture with attention mechanism (see section 4.2.3).

4.2.1. RNN Architecture

Figure 4.1 shows the RNN architecture. The input to the RNN is a sequence of multi-hot representation of ambiguity classes $X^m$. The output value of the RNN $h$ (see section 2.3) is introduced to a NN layer (see equation 2.1). The values used to compute the cross entropy loss (see section 2.3.6) are the expected PoS tag $y$ and the output value of the NN layer $s$ in which a softmax function is applied, assigning a probability to each PoS tag $\hat{y}$; therefore the PoS tag with the greatest probability, it will be the most promising PoS tag and the prediction of the network to tag the word.

$X^m$ (ambiguity class)

```
<table>
<thead>
<tr>
<th></th>
<th>$c_0$</th>
<th>$c_1$</th>
<th>$c_2$</th>
<th>$c_3$</th>
<th>$c_4$</th>
<th>$c_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_0$</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_1$</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_2$</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_3$</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_4$</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_5$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
```

Table 4.1: Relation between ambiguity class (rows) and PoS tags (columns).

4.2.2. Architecture in two phases

The second architecture, shown in figure 4.2 learns two tasks. The first task consist of predicting the next ambiguity class using a RNN. The input to this phase is a sequence of one-hot representations of ambiguity classes $X^o$. The output value of the
RNN $h$ is introduced to a NN layer, generating the output value of the first phase $s$. The second task predicts the PoS tag using a feedforward NN and the input of this phase is $h$. In this way, the first task provides context information to the second one.

We propose training this architecture in two phases [22]. The first one minimizes the loss of predicting the next ambiguity class; computing the cross entropy loss (see section 2.3.6) between the output of the first phase in which $\text{softmax}_1$ is applied, assigning a probability to each ambiguity class $\hat{y}_1$ and the expected next ambiguity class $y_1$. Once this task converges, it is time to train the second one by minimizing the loss of predicting the PoS tag, computing the cross entropy loss between the output of the second phase that is the output of the feedforward NN $z$ in which $\text{softmax}_2$ is applied, assigning a probability to each PoS tag $\hat{y}_2$ and the expected PoS tag $y_2$. It is worth nothing that, in the second phase the loss is propagated to the parameters used in the first phase as well. This is done because those parameters contribute to the final prediction.

The greatest probability of $\hat{y}_1$ will be the most promising ambiguity class and prediction of the network in the first phase. The same happens with $\hat{y}_2$ whose greatest probability will be the most promising PoS tag and the prediction of the network in the second phase.

---

**Figure 4.2:** Architecture in two phases.

---

### 4.2.3. Seq2Seq with Attention Architecture

The last architecture, a Seq2Seq with attention mechanism, is shown in figure 4.3. The input to the Seq2Seq is a sequence of multi-hot representation of ambiguity classes $X^m$. The output value of the Seq2Seq with attention mechanism $\hat{y}$ (see sections 2.3.7
and \( \text{section 2.3.9} \) and the expected PoS tag \( y \) are used to compute the cross entropy loss (see section 2.3.6). The PoS tag with the greatest probability \( \hat{y} \), it will be the most promising PoS tag and the prediction of the network.

The encoder of the Seq2Seq provides context information to the decoder of the Seq2Seq, as it happens in the architecture in two phases (see section 4.2.2) in which the first phase provides context information to the second phase. Also, the input of this architecture is influenced by RNN architecture (see section 4.2.1), because it uses \textit{multi-hot} representation of ambiguity classes as input instead of \textit{one-hot} representation. For these reasons, this architecture combines the two previous architectures.

\[
\begin{align*}
X^n \quad &\text{(ambiguity class)} \\
\downarrow \\
\text{Seq2Seq + Attention} \\
\downarrow \hat{y} \\
\text{loss} \quad &\text{y (PoS tag)} \\
\end{align*}
\]

Figure 4.3: Seq2Seq with attention architecture.
Chapter 5

Experiments

In this fifth chapter we explain the naive baselines (see section 5.1). Next, we present the synthetic dataset (see section 5.2). Finally, we show the results of the experiments with the synthetic (see section 5.2.5) and the Wall Street Journal (see section 5.3.1) datasets. Also, the open source implementation of this Bachelor’s Thesis is available online in GitHub in order to ensure that all the experiments can be reproduced.

5.1. Naive baselines

We compare the performance of our approaches against three naive baselines: one working at the PoS tag level (see section 5.1.1), a second one working at the ambiguity level (see section 5.1.2) and a third one working at the word level (see section 5.1.3). They are based on counting PoS tags. The ambiguity class for unknown words counts all the PoS tags in the training corpus and chooses the PoS tag that belongs to the open lexical categories (see section 1.2) with the highest number of occurrences.

5.1.1. Tag level

This first naive baseline counts the number of occurrences of each PoS tag in the training set, and then, given an ambiguity class; chooses the PoS tag that belongs to that ambiguity class with a higher number of occurrences in the training corpus.

https://github.com/franborjavaleronpostagging
5.1.2. Ambiguity level

This second naive baseline counts the number of occurrences of each PoS tag associated with each ambiguity class in the training set, and then, given an ambiguity class chooses its PoS tag with a higher number of occurrences in the training corpus.

5.1.3. Word level

This third naive baseline counts the number of occurrences of each PoS tag associated with each word in the training set, and then, given a word chooses its PoS tag with a higher number of occurrences in the training corpus.

5.2. Synthetic dataset

Our approach to PoS tagging predicts a PoS tag given an ambiguity class (see section 1.2). Depending on the context, this PoS tag may change. For better understanding the problem, we have built different automata emulating the behaviour of the natural language in different scenarios. These small automata are then joint together (see section 5.2.4) to create synthetic data.

The three small automata, we have built are: one with backward dependencies (see section 5.2.1), a second one with forward dependencies (see section 5.2.2) and a third one without dependencies (see section 5.2.3).

5.2.1. Automaton with backward dependencies

Figure 5.1 shows the automaton with backwards dependencies. At time step $t$, the tagger has to assign a PoS tag to ambiguity class $a_2$, but this ambiguity class is ambiguous, as it can be assigned the PoS tag $c_2$ or $c_3$. In this scenario, it is crucial to know the ambiguity class at the previous time step $t - 1$ in order to correctly predict the PoS tag of ambiguity class $a_2$. If the previous ambiguity class is $a_1$, the current PoS tag assigned to $a_2$ must be $c_3$; otherwise, the previous ambiguity class is $a_0$, and the tagger must assign to $a_2$ the PoS tag $c_2$. This automaton generates an input sequence that is constituted by ambiguity classes and its corresponding output sequence that is constituted by PoS tags. For instance, the input sequence $[a_0, a_2]$ its corresponding
output sequence is $[c_0, c_2]$.

![Figure 5.1: Automaton with backward dependencies.](image)

**5.2.2. Automaton with forward dependencies**

Figure 5.2 shows the automaton with forward dependencies. It is very similar to the one in figure 5.1 but in the opposite direction, that is, for predicting correctly the PoS tag of ambiguity class $a_0$ it is necessary to know the ambiguity class in subsequent time step $t+1$. If in $t+1$ the ambiguity class is $a_1$, the current tag must be $c_0$; otherwise, when the subsequent ambiguity class is $a_2$, the current tag must be $c_1$.

![Figure 5.2: Automaton with forward dependencies.](image)

**5.2.3. Automaton without dependencies**

Figure 5.3 shows the automaton without dependencies. This automaton does not have to take into account backward and forward dependencies, therefore at time step $t$, the tagger has to assign PoS tag $c$ to ambiguity class $a$.
5.2.4. Complex automaton

The three simple automata described in previous sections (5.2.1, 5.2.2 and 5.2.3) can be combined in different ways to obtain a complex automaton (see figure 5.4). This complex automaton can be considered as a simplification of the real problem that we want to tackle. This automaton generates 4 different input sequences with their corresponding output sequences, also it contains 2 states that can take two different ways with the same probability.

5.2.5. Results

We use the complex automaton shown in figure 5.4 to ensure that our NN architectures learn from toy data. In these experiments, training, development and test sets contain the same 4 input and output sequences generated by the automaton, but with different distributions. These sets consist of 20,000, 200 and 200 sequences, respectively, chosen at random. Table 5.1 shows the NN setup used for training the models with synthetic data.

In all the experiments of this work, the accuracy is a score in [0,1] that indicates the correct predictions and the cost is computed using the equation 2.8. Figure 5.5 shows the results of the architecture in two phases (listed in table 5.1) predicting the next ambiguity class during the training. As can see in figure 5.5, the unidirectional RNN cannot learn perfectly the input sequences generated by the automata, because it only takes into account backwards dependencies, for this reason bidirectional RNN learns...
Table 5.1: NN setup used with the synthetic dataset.

<table>
<thead>
<tr>
<th>RNN</th>
<th>Type</th>
<th>Layers</th>
<th>Units</th>
<th>Model name</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSTM</td>
<td>1</td>
<td>8</td>
<td></td>
<td>lstm8</td>
</tr>
<tr>
<td>Bi-LSTM</td>
<td></td>
<td></td>
<td></td>
<td>blstm8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Two phases</th>
<th>Phase 1: RNN</th>
<th>Phase 2: feedforward NN</th>
<th>Model name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Layers</td>
<td>Units</td>
<td>Hidden layers</td>
</tr>
<tr>
<td>LSTM</td>
<td>1</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>Bi-LSTM</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Seq2Seq</th>
<th>Encoder</th>
<th>Decoder</th>
<th>Attention</th>
<th>Model name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Layers</td>
<td>Units</td>
<td>Type</td>
<td>Layers</td>
</tr>
<tr>
<td>LSTM</td>
<td>1</td>
<td>8</td>
<td>LSTM</td>
<td>1</td>
</tr>
<tr>
<td>Bi-LSTM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.2: Results in inference mode of predicting the next ambiguity class too, but once finished the training of predicting PoS tag. As can see in figure 5.5 and in table 5.2, the results do not match. This happens, because the weights of the second phase fit the weights of the RNN in order to minimize the loss of predicting PoS tag, producing worse predictions for predicting the next ambiguity class.

Figures 5.6, 5.7 and 5.8 show the values of the cost and the accuracy obtained for all the models (listed in table 5.1) during training of predicting PoS tag. Table 5.3 shows results on the test set. In the toy automaton language shown in figure...
5.4 the three naive baselines produce the same accuracy values, because each word correspond with a different ambiguity class, therefore there are not more than one word that belongs to the same ambiguity class. These results confirm bidirectional surpass unidirectional in this type of problem, because data contains forward dependencies.
Chapter 5. Experiments

too, not only backward dependencies. For this reason, in WSJ experiments we will use only bidirectional. However, in the Seq2Seq architecture with an unidirectional RNN encoder learns perfectly the prediction of PoS tag, because in this simple toy automata language with the variable encoded estate computed by the attention mechanism and the prediction in the previous time step are sufficient to predict correctly the PoS tag. Later, we will see if it happens the same with WSJ dataset.

<table>
<thead>
<tr>
<th>Model</th>
<th>Cost</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>lstm8-nn8</td>
<td>1.146</td>
<td>0.681</td>
</tr>
<tr>
<td>blstm8-nn8</td>
<td>0.226</td>
<td>0.856</td>
</tr>
</tbody>
</table>

Table 5.2: Results of predicting the next ambiguity class with the synthetic dataset.

<table>
<thead>
<tr>
<th>Model</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
</tr>
<tr>
<td>naive baseline</td>
<td>0.748</td>
</tr>
<tr>
<td>lstm8</td>
<td>0.936</td>
</tr>
<tr>
<td>blstm8</td>
<td>1.000</td>
</tr>
<tr>
<td>lstm8-nn8</td>
<td>0.936</td>
</tr>
<tr>
<td>blstm8-nn8</td>
<td>1.000</td>
</tr>
<tr>
<td>lstm8-lstm8-a8</td>
<td>1.000</td>
</tr>
<tr>
<td>blstm8-lstm8-a8</td>
<td>1.000</td>
</tr>
</tbody>
</table>

Table 5.3: Results of predicting the PoS tag with the synthetic dataset.
5.3. Wall Street Journal Dataset

In order to use the WSJ dataset, first we have to preprocess the corpus and make sure that the data introduced to the neural network is correct (section 5.3.1); after that, we conduct the experiments and evaluate the resulting neural networks (section 5.3.2).

5.3.1. Preprocessing

The WSJ dataset is composed of 25 sections, that we split in three sets. The first one is the training set and encompasses from section 0 to 18. The second one is the development set and encompasses from section 19 to section 21, and is used as a reference during training in order to apply early stopping (see section 2.4). The last one is the test set which encompasses from section 22 to section 24, and is used to evaluate the models. During the preprocessing of the original text files, we found several ambiguous words that were not disambiguated. As we follow a supervised training approach, sentences with non disambiguate words have been ignored.

As the development and test sets may contain unknown words not present in the training set, we created an ambiguity class for them. The PoS tags that belong to this ambiguity class are those corresponding to open lexical categories (see section 1.2).

After preprocessing the WSJ corpus, we analysed the distribution of unambiguous, ambiguous and unknown words in the three sets, and found out that about 2/3 of the words were ambiguous (including unknown words), which was wrong, because in English about 1/3 of the words in running texts are ambiguous (see section 1.2). As a consequence of that, we analysed the frequency of PoS tags for each word; for instance for the determiner “the” we obtained the next occurrences with the training set: { NNP:41, DT:48875, JJ:7, NN:1, CD:1, VB:1, VBP:1}; where several were incorrect, because “the” is not ambiguous and must always be assigned the PoS tag DT. In order to avoid the problem of words wrongly tagged, we only contemplate, for each word, the PoS tags with a frequency higher than 1%. The size of the vocabulary in training set is 39,471 words of which 381 words are assigned a PoS tags with a frequency lower than 1%. After applying this filtering, 237 words continued to be ambiguous and 144 ended not being so. The number of the ambiguity classes was reduced to 292 (before
applying the filtering there were 354). Table 5.4 shows the number of sentences in each
set: sentences containing at least one word tagged incorrectly were omitted if they
were ambiguous, otherwise they were substituted by the correspondent correct PoS
tag. Table 5.5 and figure 5.9 shows the distribution of unambiguous, ambiguous and
unknown word in each set, before and after the filter.

<table>
<thead>
<tr>
<th>Set</th>
<th>Sentences</th>
<th>Max length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BF</td>
<td>AF</td>
</tr>
<tr>
<td>Training</td>
<td>40,084</td>
<td>39,542</td>
</tr>
<tr>
<td>Develop</td>
<td>6,109</td>
<td>6,011</td>
</tr>
<tr>
<td>Test</td>
<td>7,025</td>
<td>6,915</td>
</tr>
</tbody>
</table>

Table 5.4: Information about the sentences in the WSJ dataset, before (BF) and after (AF) the filter.

<table>
<thead>
<tr>
<th>Set</th>
<th>Words</th>
<th>Ambiguous words</th>
<th>Unknown words</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BF</td>
<td>AF</td>
<td>BF</td>
</tr>
<tr>
<td>Training</td>
<td>968,458</td>
<td>951,005</td>
<td>606,272</td>
</tr>
<tr>
<td>Develop</td>
<td>147,883</td>
<td>144,930</td>
<td>94,745</td>
</tr>
<tr>
<td>Test</td>
<td>170,886</td>
<td>167,483</td>
<td>110,900</td>
</tr>
</tbody>
</table>

Table 5.5: Information about words in the WSJ dataset, before (BF) and after (AF) the filter.

5.3.2. Results

Table 5.7 shows NN setup of the trained model used in these experiments with
WSJ corpus. All of them use bidirectional LSTM, with the exception of the encoder
of one Seq2Seq experiment and both decoders of the Seq2Seq which use unidirectional
LSTM. Dropout is used with a keep probability of 0.5; it is applied in all LSTMs and
the feedforward NNs. The optimizer chosen is Adam with a learning rate of 0.003 and
the size of the batch is 256.

As can be seen in table 5.6 the results of the baselines on the development and
test sets are very close. The three baselines obtain the same accuracy with respect to
unknown, because they use the same method to tag them.

Table 5.8 shows the results of predicting the next ambiguity class; with a large
number of LSTM units, better results are obtained; however, in table 5.9, where the
results of predicting PoS tag are reported, it can be seen that a high number of neurons
Figure 5.9: Distributions of unambiguous, ambiguous and unknown words in the WSJ dataset, before (BF) and after (AF) the filter.

<table>
<thead>
<tr>
<th>Baseline</th>
<th>Accuracy</th>
<th>Total</th>
<th>Ambiguous</th>
<th>Unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Dev</td>
<td>Test</td>
<td>Dev</td>
<td>Test</td>
</tr>
<tr>
<td>Tag level</td>
<td>0.818</td>
<td>0.816</td>
<td>0.513</td>
<td>0.512</td>
</tr>
<tr>
<td>Ambiguity level</td>
<td>0.871</td>
<td>0.873</td>
<td>0.659</td>
<td>0.665</td>
</tr>
<tr>
<td>Word level</td>
<td>0.901</td>
<td>0.902</td>
<td>0.741</td>
<td>0.743</td>
</tr>
</tbody>
</table>

Table 5.6: Baselines results with the WSJ dataset.

<table>
<thead>
<tr>
<th>Architectures</th>
<th>RNN</th>
<th>Type</th>
<th>Layers</th>
<th>Units</th>
<th>Model name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RNN</td>
<td>Type</td>
<td>Layers</td>
<td>Units</td>
<td>Model name</td>
</tr>
<tr>
<td></td>
<td>Bi-LSTM</td>
<td>LSTM</td>
<td>64</td>
<td>128</td>
<td>blstm64-d0.5</td>
</tr>
<tr>
<td></td>
<td>Bi-LSTM</td>
<td>LSTM</td>
<td>64</td>
<td>128</td>
<td>blstm128-d0.5</td>
</tr>
<tr>
<td>Two phases</td>
<td>Phase 1: RNN</td>
<td>Phase 2: feedforward NN</td>
<td>Units</td>
<td>Units</td>
<td>Units</td>
</tr>
<tr>
<td></td>
<td>Type</td>
<td>Layers</td>
<td>Units</td>
<td>Units</td>
<td>Units</td>
</tr>
<tr>
<td></td>
<td>Bi-LSTM</td>
<td>LSTM</td>
<td>64</td>
<td>128</td>
<td>bilstm256-ml64-d0.5</td>
</tr>
<tr>
<td></td>
<td>Bi-LSTM</td>
<td>LSTM</td>
<td>64</td>
<td>128</td>
<td>bilstm512-ml64-d0.5</td>
</tr>
<tr>
<td>Seq2Seq</td>
<td>Encoder</td>
<td>Layers</td>
<td>Units</td>
<td>Type</td>
<td>Decoder</td>
</tr>
<tr>
<td></td>
<td>LSTM</td>
<td>64</td>
<td>LSTM</td>
<td>64</td>
<td>64</td>
</tr>
</tbody>
</table>

Table 5.7: NN setup used with the WSJ dataset.
in the RNN of the first phase, does not result in better accuracy in the second phase: the performance of both models are very close and the same happens with RNN architecture. However, the two models trained with Seq2Seq do not obtain similar results, because one uses a bidirectional encoder and the other uses an unidirectional encoder, unlike synthetic dataset (see section 5.2.5) in which both trained models obtained almost identical results. Therefore, with WSJ dataset a bidirectional Seq2Seq encoder surpasses an unidirectional Seq2Seq performance, as commented in section 2.3.4. The problem of these models is the accuracy obtained for the unknown ambiguity class, since the training corpus does not contain unknown words. However, the architecture in two phase obtains better results than the other two architectures (RNN and Seq2Seq architectures), because learning to predict the next ambiguity class helps to predict the PoS of unknown words. Furthermore, the results obtained on both sets (development and test) are very similar, guaranteeing does the absence of overfitting.

All the models shown in table 5.7 outperform results obtained by naive baselines, except Seq2Seq, possibly because we implemented a greedy search rather than a beam search, as it is usually done.

<table>
<thead>
<tr>
<th>Model</th>
<th>Cost</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Dev</td>
<td>Test</td>
</tr>
<tr>
<td>blstm256-nn64-d0.5</td>
<td>2.416</td>
<td>2.400</td>
</tr>
<tr>
<td>blstm512-nn64-d0.5</td>
<td>1.902</td>
<td>1.882</td>
</tr>
</tbody>
</table>

Table 5.8: Results of predicting next ambiguity class on the WSJ dataset.

<table>
<thead>
<tr>
<th>Model</th>
<th>Cost</th>
<th>Total</th>
<th>Ambiguous</th>
<th>Unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Dev</td>
<td>Test</td>
<td>Dev</td>
<td>Test</td>
</tr>
<tr>
<td>blstm64-d0.5</td>
<td>0.290</td>
<td>0.268</td>
<td>0.921</td>
<td>0.923</td>
</tr>
<tr>
<td>blstm128-d0.5</td>
<td>0.274</td>
<td>0.257</td>
<td>0.919</td>
<td>0.921</td>
</tr>
<tr>
<td>blstm256-nn64-d0.5</td>
<td>0.215</td>
<td>0.208</td>
<td>0.934</td>
<td>0.934</td>
</tr>
<tr>
<td>blstm512-nn64-d0.5</td>
<td>0.209</td>
<td>0.203</td>
<td>0.935</td>
<td>0.935</td>
</tr>
<tr>
<td>seq64-seq64-a64-d0.5</td>
<td>1.107</td>
<td>0.968</td>
<td>0.803</td>
<td>0.820</td>
</tr>
<tr>
<td>bseq64-seq64-a64-d0.5</td>
<td>0.606</td>
<td>0.547</td>
<td>0.887</td>
<td>0.893</td>
</tr>
</tbody>
</table>

Table 5.9: Results of predicting PoS tag on the WSJ dataset.

Figure 5.10 shows the cost and accuracy obtained for the first task by the architecture in two phases during training. As can be seen, the number of epochs needed when applying early stopping to learn the task with a high confidence is very low: two epoch in the model with 256 LSTM units and only one in the model with 512 LSTM units.
Figure 5.10: Results of the architecture in two phases on the first phase using the WSJ dataset.

As happens with results on the synthetic dataset (see section 5.2.5) the final accuracy obtained on the first task has been reduced in order to minimize the cost of learning the second task (see table 5.8).

Figure 5.11 shows the results obtained on the second task by the architecture in two phases during training. As can be seen in figures 5.12 and 5.13 the number of epochs needed by RNN architecture and Seq2Seq architecture are very large compared to architecture in two phases, which practically needs only one epoch or two epoch depending on the model trained, obtaining less cost and high accuracy than the models of the other two architectures.
Figure 5.11: Results of the architecture in two phases predicting PoS tag using the WSJ dataset.

Figure 5.12: Results of the RNN architecture on the WSJ dataset.
Figure 5.13: Results of the Seq2Seq architecture on the WSJ dataset.
Chapter 6

Conclusions

6.1. Conclusion

In this work, we have explored different approaches to PoS tagging. Additionally, we have explored different representations of the word classes in order to feed them as input into the model. Also, we have performed an exhaustive theoretical study of NN, specifically of sequence models like RNN and Seq2Seq. With all that background, we have proposed three architectures in order to tackle the problem of PoS tagging. Firstly, we experimented with a synthetic dataset generated by us that tries to imitate a real one, in order to ensure our implementations worked correctly. Finally, we experimented with different configurations of the proposed architectures on the WSJ corpus and we achieved a system able to disambiguate with a 93.5% total accuracy and with 83.2% accuracy on ambiguous words. Our results do not equal the state of the art in part-of-speech tagging on this dataset that is around 97% total accuracy \cite{16}. We use one of the first version of WSJ. As can be seen in section 5.3.1 the corpus contains words tagged incorrectly. Probably, the preprocessing of the corpus used by us has been affect to do not achieve a performance close to 97% total accuracy, also those systems use the last version of WSJ corpus in which contains less mistakes and use more elaborated representations of the words than ours.

6.2. Highlights

The highlights of the work are the following:
Chapter 6. Conclusions

- In-depth study of the neural models.

- Configuration of computing instances at GCP for DL research.

- Neural part-of-speech system implemented with TensorFlow. It allows to train whatever model that follows the proposed architectures described in this project, given a JSON file with the desired configuration. Once the models have been trained, they may be evaluated with a test set or used to disambiguate a raw text.

6.3. Future work

Due to the time constraints imposed on this project, many possible improvements were left out for future works. Here we summarize them:

- Complete the implementation of the Seq2Seq, since it actually uses a greedy search at inference mode instead of a beam search approach.

- Explore other approaches to represent unknown words, since the results obtained in this aspect are not good enough.

- Experiment with other hyperparameters and configurations.

- Calculate tests of statistical significance in order to know which is the best trained model.

- Include an unsupervised approach. In the training of this approach given an ambiguity class as input, its desired output will be its corresponding multi-hot representation in which all possible PoS tags have the same probability.
Bibliography


[16] Christopher D Manning. Part-of-Speech Tagging from 97 % to 100 %: Is It Time for Some Linguistics ?


