Neural sequence labelling

Natural Language Processing and Deep Learning

Francisco Martínez Lasaca (frml@itu.dk)

July 26, 2020

1 Introduction

Parts of speech (PoS; also known as *word classes* or *syntactic categories*) are the different categories in which words that behave similarly within a sentence can be classified. Most of them rely on the following categories: noun, verb, pronoun, preposition, adverb, conjunction, participle, and article. Identifying the parts of speech of the words in a sentence is useful, as we can infer which are the most likely types of the surrounding words, and it eases information retrieval and coreference resolution [3].

A program that, given a sentence, tags their words appropriately is called a *PoS tagger*, and there are multiple ways to implement them. In this report, we present the methodology and analysis of an implementation using a Bi-LSTM or *Bidirectional Long Short Term Memory* neural network.

2 LSTM

An *(Artificial) Neural Network* or NN is a network of different processing units united by weights. Neural networks can either have cycles or not. *Recurrent Neural Networks* (or RNN) are networks that exploit the cycles in a network ([2]). A representation of a simple RNN is presented in Figure 1. Their advantage over networks that do not allow cycles is that the recurrent connections allow a 'memory' of previous inputs to persist in the network's internal state, and thereby influence the network output.

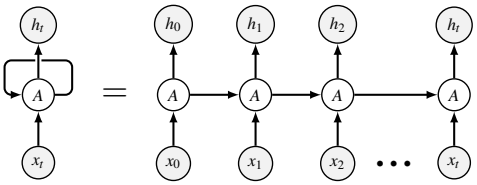


Figure 1: Recurrent Neural Network[1]

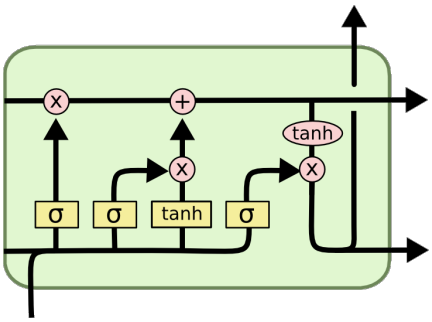


Figure 2: LSTM memory cell[2]

The problem with this simple approach is that is has a limited memory: the influence of a given input on the hidden layer, and therefore on the network output, either decays or blows up exponentially as it cycles around the network's recurrent connections. This is known as the *vanishing gradient problem* ([1]).

An LSTM (*Long Short Term Memory*) is a special type of RNN that solves this vanishing gradient problem. Their architecture consists of a set of recurrently connected subnets, known as *memory cell* (see Figure 2). Each cell contains one or more self-connected memory cells and three multiplicative units —the *input*, *output* and *forget* gates.

3 Data

Recall our mission: PoS tagging. In order to train our neural network to output parts of speech given sentences, we need annotated sentences: our data. We use an extensive PoS-tagged corpora called Univeral Dependencies (UD).[3] In particular, we

[1] Source: https://tex.stackexchange.com/questions/494139/how-do-i-draw-a-simple-recurrent-neural-network-with-goodfellows-style

[2] Source: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

[3] https://universaldependencies.org

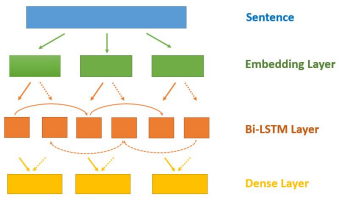


Figure 3: Neural network model

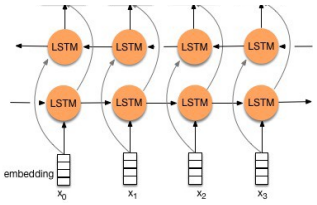


Figure 4: Bidirectional LSTM[10]

are going to use GUM,[4] the Universal Dependencies syntax annotations from the GUM corpus in English.

The GUM corpus is made of 5961 sentences (113,374 words), which is already divided into a training, a test and a validation corpus. For simplicity of the code, we are *not* going to respect this separation of the data. Instead, the training and validation corpora are put together and later divided dedicating an 80 % of this joint corpus to training, and 20 % for validation.

At a later step we will also use UD German-HDT[5]— a corpus for the German language.

4 Implementation

The implementation is available in a Google Colab notebook,[6] and it is built on top of the code of an article called *Build a POS tagger with an LSTM using Keras*.[7] We distinguish five implementation steps:

Loading the data The data come as form of `*.conllu` files which contain sentences tagged with parts of speech and other information. In this step, those files are parsed and sentences words and tags are extracted separately.

Preprocessing the data Both input and output data have to be converted to numbers, which is the language that neural networks speak. First, inputs —sentences— are represented assigning an integer per unique word. Two extra integers are reserved for padding and for *out of vocabulary* words (or OOV). In total, $|\mathcal{V}|+2$ integers. For outputs —tags— we follow the same procedure (without the OOV category), and then represent each tag using one-hot-encoding. Each tag is represented as a vector of the same size as the total number of tags plus one, where every position is 0 except for a value of 1 in the actual tag index. Finally, we pad all the data with the previously reserved word and tag *pad* so that all of them are of the same size. The reason for this is that the neural network has a fixed size. We use the length of the largest sentence; that is, the length of $\arg\max_{s \in S} |s|$.

Creating a neural network model The network architecture is formed of stacked layers which can be thought as different computation stations that progressively transform the input (see Figure 3):

1. The first is an *Embedding* layer. This layer converts the input represented as integers (e.g. `[25, 782, 42]` for a sentence of 3 words) to dense vectors of a fixed size. Remains as an extension of the project using other embeddings, such as word2vec[8] or GloVe.[9]
2. The second layer is a bidirectional LSTM or Bi-LSTM. They are formed of two independent LSTMs one above the other —see Figure 4. While a LSTM can propagate the state from one input to another (i.e. left to right or right to left), with Bi-LSTM, we gain both backward and forward information of the sequence. Their use is very useful to solve this problem, as they retain more information of the context and yield better results.
3. The last one is a *dense* or fully-connected layer. With softmax as the activation function, this layer will finally solve the classification problem: determining to which tag the inputted word is associated.

[4] https://github.com/UniversalDependencies/UD_English-GUM

[5] https://github.com/UniversalDependencies/UD_German-HDT/tree/master

[6] https://colab.research.google.com/drive/1Mxr5i9XeELrfgaoHB4RY6GPEnFkPY3PP?usp=sharing

[7] https://nlpforhackers.io/lstm-pos-tagger-keras/

[8] https://radimrehurek.com/gensim/models/word2vec.html

[9] https://nlp.stanford.edu/projects/glove/

[10] Source: https://towardsdatascience.com/understanding-bidirectional-rnn-in-pytorch-5bd25a5dd66

Training the model We split the preprocessed data into three datasets: the training, the validation and the test datasets. We use the training dataset to train the neural network, and —once the model is trained— we evaluate it on the test dataset, which is data that has never been seen before.

We also reserve some data for tuning hyperparameters: the validation dataset. One problem neural network suffer is called *overfitting* —when a model is too tight to the training data and it is not useful for similar but unseen cases. To solve that, we can use *early stopping*, which monitors the validation dataset loss to stop training once the model has got the best accuracy on the validation model and starts to get worse. An example can be seen in Figure 5: a plot of the losses of both the training and the validation datasets over epoch. If an early stopping algorithm had been applied to this training of the neural network, the training would have stopped around the 14th epoch, when the validation loss hits its minimum.

Another benefit of early stopping is that we do not have to specify a number of epochs: the training will automatically finish when the evaluation dataset hits its minimum loss.

Finally, to train our model we also need to specify how big our batches are and which optimizer to use. There are different out-of-the-box algorithms (fixed learning rate, RMSProp, ...); we are going to use Adam. According to [4]: the method is "computationally efficient, has little memory requirement, invariant to diagonal rescaling of gradients, and is well suited for problems that are large in terms of data/parameters". Its learning rate is called α, and we will vary it along with the batch size in the following section.

Evaluating the performance We use *metrics* to evaluate the performance of the model. We use a modified version of the regular *accuracy*, where padding tags are not taken into account. We will also compute the per-sentence accuracy.

5 Analysis

5.1 Changing batch size and α

We start the analysis by examining how the neural network responds to varying the batch size and the learning rate of the optimizer.

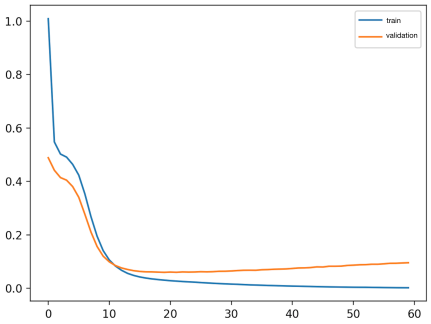


Figure 5: Losses of the training and validation datasets over epoch

As shown in Appendix A, a value α of 0.01 or 0.001 produces the best per-token and per-sentence accuracies, with a batch size of 32. With both too high or too low values of α, the neural network shows an erratic behaviour. Empirically, for high values of α we have seen that the optimizer may be lucky enough to take a "golden step" in the error surface —like in (128, 0.1)— or simply do not find anything, like in the rest of cases with $\alpha = 0.1$. For low values of α, the learning process is so slow that is stopped by the early stopping mechanism. In other cases, where the loss is constantly reduced per epoch, even if slightly, it can reach similar accuracies —see (64, 0.0001)— but spending unnecessarily more time.

Concerning the number of epochs, if we only focus on the cases where there is a good stability (when $\alpha = 0.01$ or $\alpha = 0.001$), they increase a bit as the learning rate gets smaller.

In the rest of the analysis, we are going to use the $(\text{batch size}, \alpha) = (32, 0.001)$ setup shown in Table 2.

5.2 Classification performance

Once the network is trained, it is interesting seeing how good the model learns to classify the different tags. The results are presented in Figure 6a.

Most of the categories are tagged correctly, except for `X`, `INTJ` and `SYM`. This is probably due to the fact that they are the tags that appear least in the dataset —272, 130 and 126 out of a total of 113,374 tokens.

5.3 Changing language

We have optimized the network using English as the language of the dataset. But what happens if we use another one? While English is a SVO language

—that is, sentences are formed of the subject, then a verb and then an object— we are going to test the network with German, which follows an SOV structure.

Using the data described in Section 3, we obtain a 93.44 % token accuracy and a 40.69 % sentence accuracy. This good result can be explained because the dataset is significantly larger (around 30 times) than the English one. The training time was also higher (around 60 minutes). The confusion matrix is plotted in Figure 6b. The results are satisfactory for all the tags, except for some nouns and proper nouns. In any case, those tags of speech are very similar and may be even ambiguous for a native speaker to identify —is *Dama* a proper noun or a noun? And it is particularly more difficult in German, where most nouns are capitalized, as proper nouns in many other languages.

5.4 Changing recurrent layer

Instead of using LSTMs, we can use other recurrent layers like Gated Recurrent Units (GRU), and also check if having a bidirectional recurrent layer or not has an impact on the accuracy.

	LSTM	**GRU**
Regular	83.35 / 16.85	84.11 / 17.86
Bidirectional	86.56 / 22.92	86.17 / 21.12

Table 1: Token and sentence accuracy (%)

The results in the previous Table 1 show that the bidirectional layers surpass the regular ones, and that there is not a clear advantage of using GRU instead of LSTM.

References

[1] Y. Bengio, P. Simard, and P. Frasconi. "Learning long-term dependencies with gradient descent is difficult". In: *IEEE Transactions on Neural Networks* 5.2 (1994), pp. 157–166.

[2] Alex Graves. "Supervised sequence labelling". In: *Supervised sequence labelling with recurrent neural networks*. Springer, 2012, pp. 5–13.

[3] Dan Jurafsky and James H Martin. *Speech and language processing. Vol. 3*. 2014.

[4] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. eprint: `arXiv:1412.6980`.

A Learning rates and batch sizes

Here follows how the per-token and per-sentence accuracies, and the necessary number of epochs vary as the neural network with a bidirectional LSTM is trained with the English corpora while varying the optimizer's learning rate α and the batch size. See tables 2, 3 and 4.

	1.0000	**0.1000**	**0.0100**	**0.0010**	**0.0001**
32	26.64	0.00	85.99	**86.56**	16.87
64	18.54	0.00	86.36	85.71	84.82
128	19.80	67.70	82.30	85.41	0.00
256	8.69	0.00	86.19	19.69	0.00
512	4.96	0.00	85.52	16.70	0.00

Table 2: Token accuracy (in %) of the Bidirectional LSTM NN with batch size over ADAM's α.

	1.0000	**0.1000**	**0.0100**	**0.0010**	**0.0001**
32	1.24	0.00	20.90	**22.92**	0.00
64	1.12	0.00	22.69	21.34	19.10
128	1.24	6.97	16.07	21.01	0.00
256	0.00	0.00	22.34	0.00	0.00
512	0.00	0.00	22.47	0.00	0.00

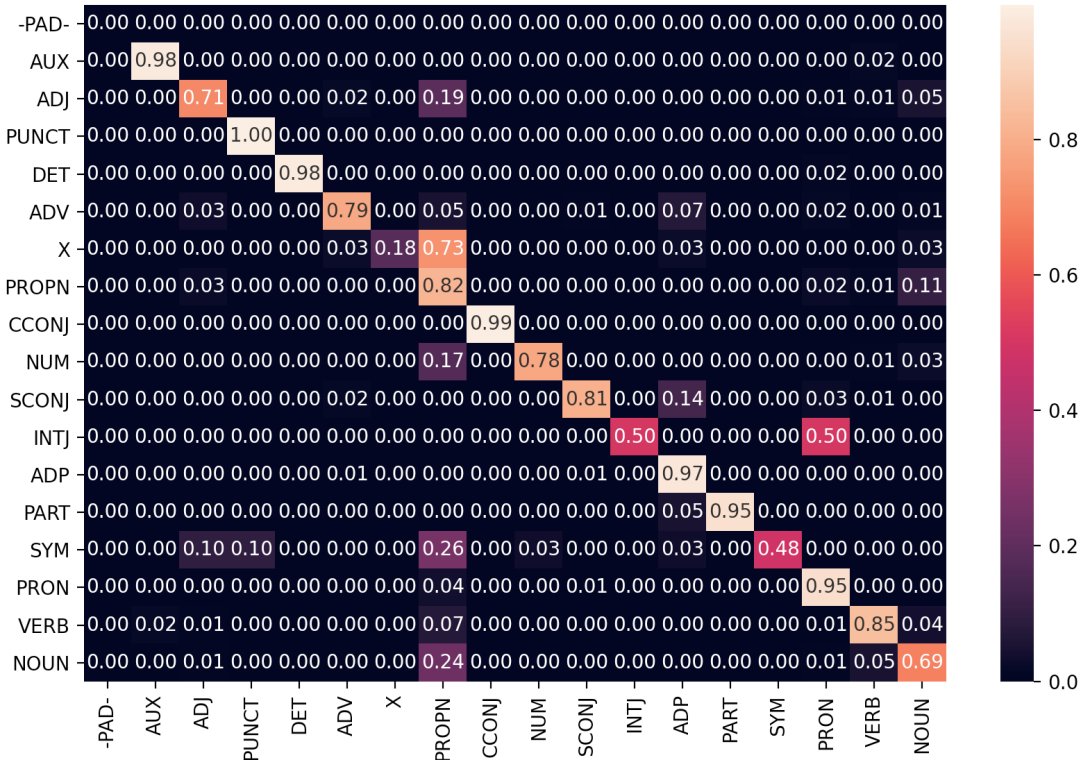
Table 3: Sentence accuracy (in %) of the Bidirectional LSTM NN with batch size over ADAM's α.

	1.0000	**0.1000**	**0.0100**	**0.0010**	**0.0001**
32	5	4	6	11	5
64	6	4	7	14	50+
128	6	41	12	19	4
256	4	4	13	6	4
512	7	4	17	9	4

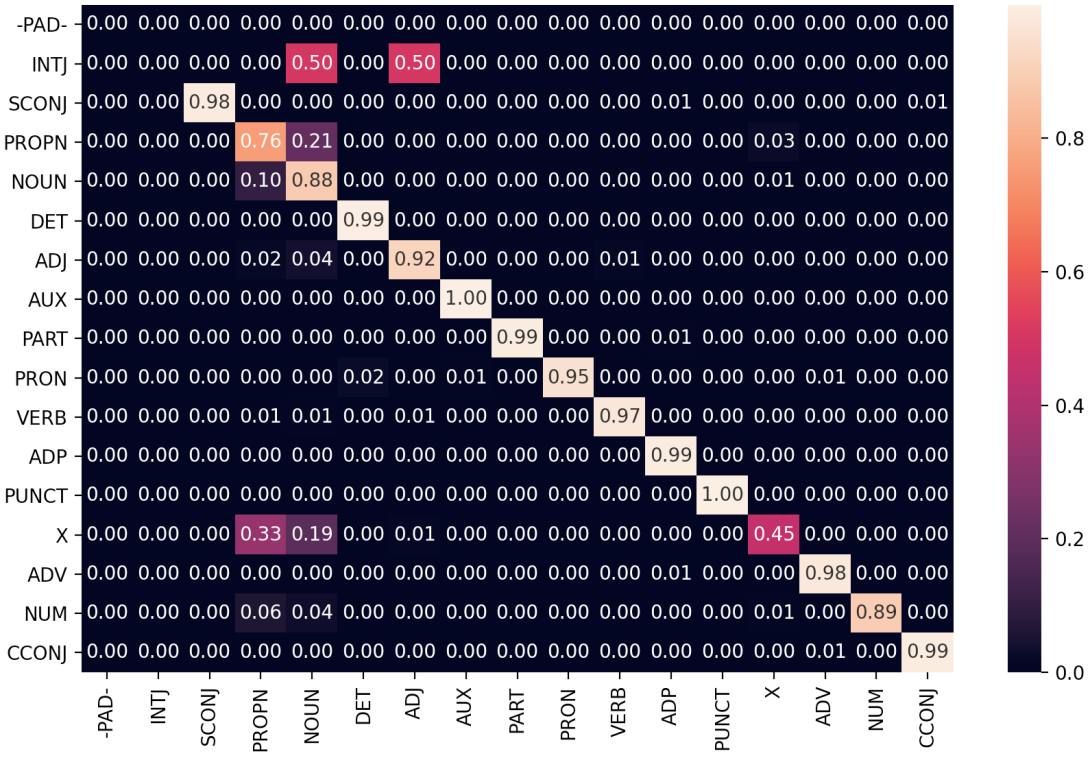
Table 4: Number of epochs of the Bidirectional LSTM NN with batch size over ADAM's α.

B Confusion matrices

Confusion matrices (Figure 6) are presented for English (Figure 6a) and German (Figure 6b). Both of them use Bidirectional LSTM, a batch size of 32 and $\alpha = 0.001$.



(a) English



(b) German

Figure 6: Confusion matrices. True tag over predicted tag.