Natural Language Processing, Homework 2 Report

Dorjan Hitaj, Matricola 1740478

May 7, 2017

1 Abstract

LSTM have shown to be very effective for tagging sequences tasks, and Word embeddings have shown to be very useful in representing the statistical properties of natural language. In this homework we had to implement a part of speech tagger based on recurrent neural networks, more specifically, a network build upon LSTM with word embeddings in order to classify the tokens in a corpus of sentences, into one of the 17 universal part of speech tags they belong to. For this task, the model I built achieved a F1-score of **94.29%** when tested on the test set.

2 Model Description

In here I will briefly describe the topology of the neural network I constructed to get the best model for this task. Complete reasoning of how and what led me to this model is described in the Optimization and Experiments section.

My neural network consists of 2 LSTM layers, the first one with 160 neurons, **tanh** activator, **sigmoid** recurrent activator, **uniform** kernel initializer and a recurrent dropout of 10%, and also set the dropout rate to 30%. The input shape is defined on the first layer and the shape is (window*2+1, 300), the first showing the total length of a single word context(***right,word,left***) and the 300 stands for the length of a single word vector. The second LSTM has 75 neurons, **tanh** activator, **uniform** kernel initializer and a dropout rate of 10%. In the end I have a **Dense** layer with 17 nodes and **softmax** activation function. The optimizing function I have is **SGD** with initial learn rate of 0.1, decay of 1e-5, momentum of 0.9 and nesterov set to True. Initial learn rate because I have used **Keras callbacks** to tune the learning rate while learning. *(in details in section 3.2)* **Nesterov** and **momentum** usage is suggested in this paper: [**Ilya Sutskever et al**]. The **loss** function is **categorical_crossentropy** since this is a multiclass classification task. The batch size I have chosen is 160 and the number of epochs is 30.

3 Optimization

3.1 Preprocessing

Before feeding the neural network with input, the data had to be transformed in the format that the neural network requires. By reading the files line by line I built 2 separate lists: One list contains all the words of the file organized in their sentences, the other contains the word's part of speech tag index. By means of **gensim** library I load the Google pretrained vectors and also the italian pretrained vectors. Since we have to handle also the unknown words I create a specific vector of length 300 by means of gensim word2vec, and use it anytime a word is not in the pretrained models.

Using Google's pretrained model to represent the words as vectors is an optimization step, because the Google pretrained model captures the semantics of the words and also has a reduced dimensionality so it will function similar to an Embedding layer. Iterating through each sentence, for each word in the sentence I built its **right** and **left** context according to a specific pre-selected windows size. The window size works like this: A window size of 3 means that for a specific word at position **i** in the sentence, the corresponding word context will be composed of the words(*represented as vectors*) in the pos [**(i-1), (i-2), (i-3), i(word itself), (i+1), (i+2), (i+3)**] The word and its right and left neighbours are transformed into vectors of dimensionality 300 by means of the pretrained models. If there are not enough words in the left or right side to complete the length of the window size, then zero vectors are added to the word context to serve as padding. In the end the whole word context is appended to a final array and the next word in sentence is taken into consideration. After processing every sentence I convert the list to numpy array since that is the input type the Keras Sequential model expects. This is the **X** vector.

I build **Y** vector by using **np_utils.to_categorical()** function, which transforms the tag index to an array of length 17 composed of all zeros and a 1 in the position that will resemble to the index of the correct tag that that word has.

3.2 Hyperparameter optimization

Neural networks are difficult to train due to the presence of many tuning parameters. To optimize some of the crucial parameters, like the number of neurons, optimizing function, learn rate, number of epochs, dropout rate and more, I performed GridSearching by means of the scikit-learn. **GridSearch** is a model hyperparameter optimization technique. It builds a model for each combination of tuning parameters, perform tests on them and in the end shows the best performing combination. This library provides **KerasClassifier** class that can be used to wrap Keras models and then perform Grid Searching that model with a dictionary of most common values of one or more tuning parameters. This method was really helpful in giving insights about the approximate values of the parameter values I had to use but I could not use it to get the most optimal ones due to memory limitations on my machine. The insights that this optimization technique gave me, narrowed my search space for the optimal parameters but I performed many individual experiments on various tuning parameters to get to best possible ones.

From Grid searching I learned that the most effective **optimizer** to use were:**RMSprop, Adamax**. But from further experiments explained on Section 4, the best performance was reached by **SGD**, which was also very efficient to tune against overfiting while the others were very fast learners but would result in overfiting the data after a few epochs.

Optimizer **learn rate** is crucial in the learning process. To help the network perform better I used a dynamic update of the learn rate. When after a certain number of epochs, loss was increasing on validation data I reduced the learning rate. I did that by means of the **Keras callbacks**. In the call back **ReduceLROnPlateau()** function I monitor loss on validation data(dev set) and if loss increases compared to the previous epoch the learn rate is reduced by a factor of 0.8. I also prevent the learn rate to become very low which would cause stagnation in the process of learning by setting a minimum learning rate of 0.001. From GridSearching I learned also that the suitable dropout rate to set was between 20 and 30%, which I confirmed through further experiments.

Through experiments I tried to optimize the values for the following parameters: **Window size**, **Dropout rate, Activation function, Optimizer, Learn rate, Number of neurons in LSTM, Number of epochs**.

4 Experiments

4.1 Optimization experiments

I performed many experiments with the various tuning parameters. Models were trained on train set and tested on the dev set. To find what was the most suitable for a parameter I performed tests like this:

I kept all the other network parameters fixed and change only the parameter whose best value I was trying to approximate. I performed also more complex experiments but this setup was a baseline for future experiments.

Batch size is the number of items you show to the neural network before the weights are updated. According to the literature, batch size has a big impact on network performance. From the experiments it was surprisingly not very significant in the overall accuracy and the results obtained were close.

Window size was a crucial parameter in accuracy. Feeding the network with the right length of word context is very important to learn the accurate part of speech tag for that word. By performing tests with window size from 0 up to 6, I noticed that after size 2 the performance is fairly good, but sizes of 4,5,6 were close. In overall, bigger window was always better. The small change in accuracy between windows 4,5 and 6 is because in the majority of the sentences do not have more than 4 words to the left and to the right to approximate the POS tag of the word in the middle, so the rest of the gaps are just padding with zero vectors. But for longer sentences we might need. This explains why a bigger window was giving higher accuracy. Test results for tuning the window size are displayed on **figure 2**. The best performing window size for my model was 6.

Number of neurons in the LSTM layer is important to tune. Too few and the network will not be able to learn and too much and the network will overfit by memorizing the data. The key was finding a value in between. Experiments on **figure 4** show the impact of neurons on learning. In that experiment I learned that a number close to 160 neurons was good enough for my model. From that I headed on to try complex networks with more than 1 LSTM layer.

Dropout rate was crucial in preventing overfiting. I used both ***normal*** dropout and also ***recurrent*** dropout on the LSTM layer. The best values were at 0.3 for the ***normal*** dropout and 0.1 for the ***recurrent*** dropout. By performing experiments with various dropout rates from 5% up to 50%, I got that the network was performing better with a dropout rate of 0.3 (**figure 5**). Some more tests on the two dropout types resulted that the combination of 30% dropout in the output neurons and 10% in the recurrent state was the most optimal for the first layer.

Activation function in the LSTM layer had a big impact on the performance of the network. The default activation function for the LSTM layer is **tanh** and after trying other activation function the default one proved to be the most suitable. On the other hand the recurrent activation function by default was **hard_sigmoid**, but after many tests the **sigmoid** function was the best to use as the recurrent activator. As for the output layer it had to be used a **softmax** activator in order to get the results as probabilities.

Number of Epochs This parameter is also dependent on the size of the corpus and the optimizer function. In the baseline experiment (**figure 3**), as the number of epochs increases so does the accuracy, but more than 30 epochs was not necessary for this task.

Optimizer, function was crucial in the learning process. From the baseline test (**figure 1**) the best performing ones were Adamax and RMSprop. But in complex networks and higher number of epochs SGD outperformed them and gave the highest accuracy. Also the **Keras callbacks** were very helpful in making progress.

Keras provided a *Bidirectional* wrapper for the LSTM so I experimented also with them. The performance was sightly better, but close to the regular LSTM.

4.2 Results and Analysis

4.2.1 English

The best model I built for the English dataset had a F1 score of 94.29% on the test set. This performance was achieved by using the Google pretrained vectors to represent the words as vectors. From the **confusion matrix** we see that the model is performing fairly well. We see in the matrix that the most common part of speech tags like NOUN and VERB are the easiest to predict by the model, thus having the highest number of correct predictions. The most common mis-predicted POS-tags are the pairs (PROPN, NOUN), (PUNCT, CONJ), (NUM, PUNCT). The pair (PROPN, NOUN) is easy to be mis-predicted by the model because the PROPN is a NOUN itself. The two other pairs that are interchanged by the model have happened because these POS tags are very close to each other in the sentences, thus they will be very close in each others context window, which sometimes causes the model to make mistakes in predicting one in place of the other.

4.2.2 Multilingual

The best model I built for the Multilingual dataset had a F1 score of 94.91% on the multilingual test set. This performance was achieved by using the Google pretrained vectors to deal with the English words and the provided Italian vectors for the Italian words. When testing the model and checking its confusion matrix, as expected the same pairs that were mis-predicted for the English model are also mis-predicted mostly by the multilingual model.

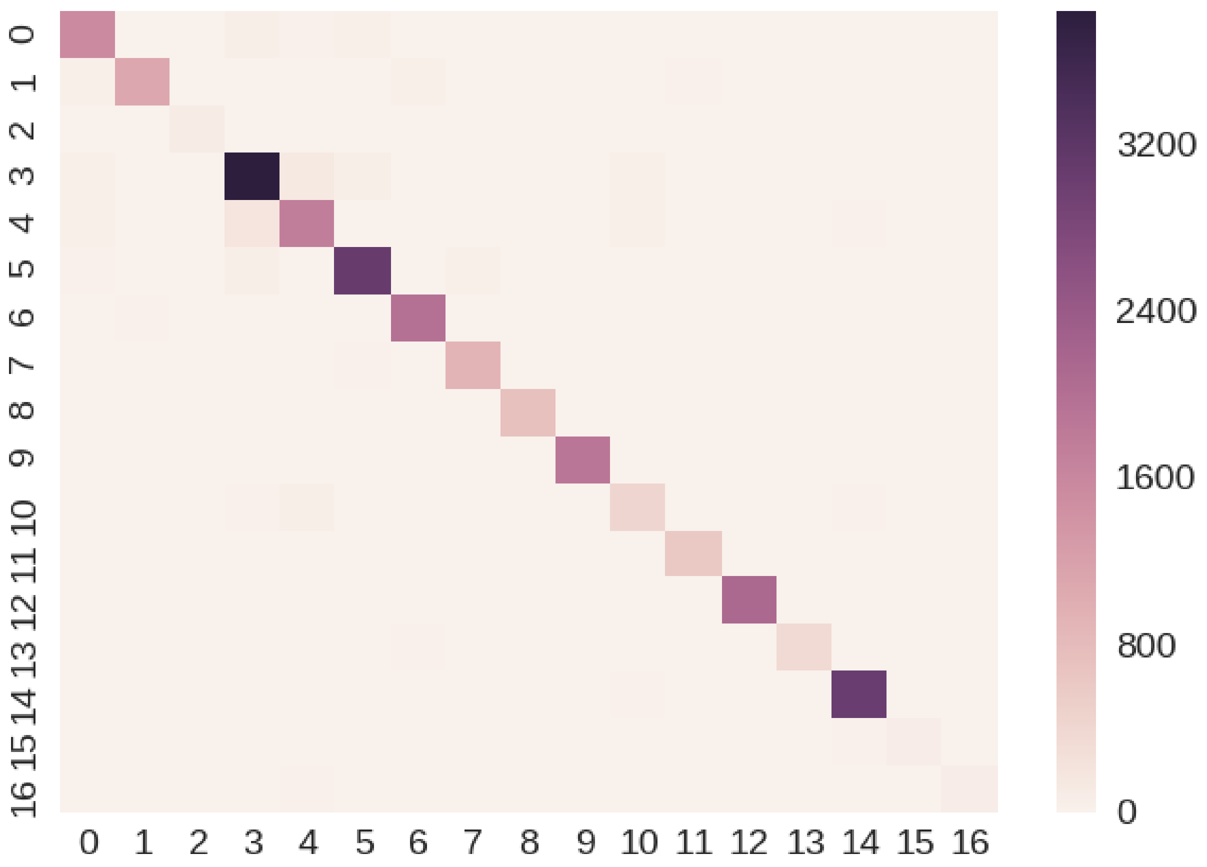
5 Conclusion

This homework was about building a POS tagger based on the LSTM-RNN. Neural networks are hard to optimize due to large number of tuning parameters. In this report I presented the strategies I followed to approximate the best tuning parameter values. They were: **GridSearch**, **Word Embeddings**, **Dynamic update of learning rate** and many more. All the parameters are tuned using the development set. The best F1 score I got was 94.29% on the English dataset and 94.91% on the multilingual dataset.

6 Note

The resource names for testing the code are as follows: **w2v-word-vectors-english** , **word-vectors-italian** , **word-vectors-italian.syn0.npy** , **word-vectors-italian.syn1neg.npy** , **word-vectors-italian.table.npy**

7 Confusion Matrix Heat Map



8 Confusion Matrix

	ADJ (0)	ADV (1)	INTJ (2)	NOUN (3)	PROPN (4)	VERB (5)	ADP (6)	AUX (7)	CONJ (8)	DET (9)	NUM (10)	PART (11)	PRON (12)	SCONJ (13)	PUNCT (14)	SYM (15)	X (16)
ADJ (0)	1553	14	0	57	19	35	5	0	0	0	3	0	1	0	6	0	0
ADV (1)	37	1095	1	10	0	8	38	0	1	6	1	19	3	3	3	0	0
INTJ (2)	4	4	98	3	4	0	1	0	1	1	0	0	1	1	1	0	1
NOUN (3)	33	8	1	3839	131	56	5	2	0	1	32	0	3	2	10	1	8
PROPN (4)	41	3	1	190	1760	4	5	1	1	1	39	0	2	0	17	1	10
VERB (5)	23	1	0	58	4	3088	2	30	0	0	3	0	0	0	2	0	1
ADP (6)	1	27	0	1	1	1	1971	0	0	0	1	5	0	9	0	1	0
AUX (7)	0	0	0	0	1	19	0	916	0	0	0	0	1	0	0	0	0
CONJ (8)	0	2	0	2	0	0	2	0	731	1	0	0	0	0	1	0	0
DET (9)	0	2	0	0	0	0	0	0	1	1885	1	0	7	0	0	0	0
NUM (10)	0	0	0	24	45	2	0	0	0	0	430	0	4	0	17	0	14
PART (11)	0	5	0	0	1	1	2	0	0	0	0	619	0	0	1	0	1
PRON (12)	0	2	0	0	0	0	1	0	0	7	0	0	2141	7	0	0	0
SCONJ (13)	0	3	0	0	0	1	27	2	0	0	1	0	3	348	2	0	0
PUNCT (14)	5	0	0	12	7	0	0	0	0	0	16	1	0	0	3055	4	6
SYM (15)	0	0	0	0	0	0	0	0	0	0	2	0	0	0	16	73	1
X (16)	4	1	1	12	25	3	2	0	0	2	13	0	0	0	13	0	63

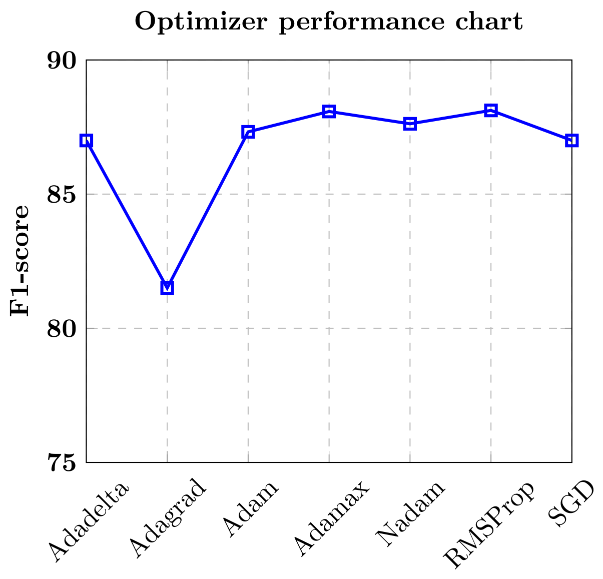


fig 1. Single LSTM layer with 100 neurons, 10 epochs, window size 3

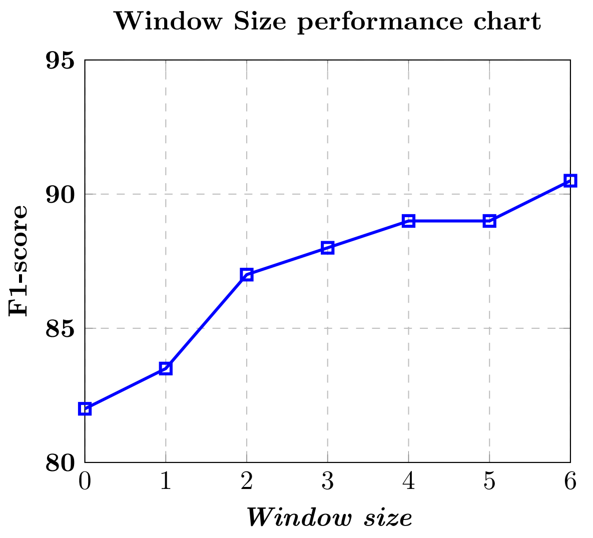


fig 2. Single LSTM layer with 100 neurons, 10 epochs, SGD optimizer

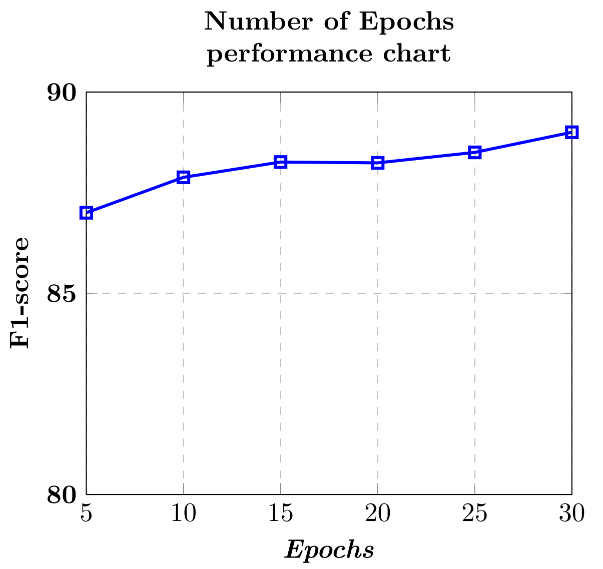


fig 3. Single LSTM layer with 100 neurons, window size 3, optimizer SGD

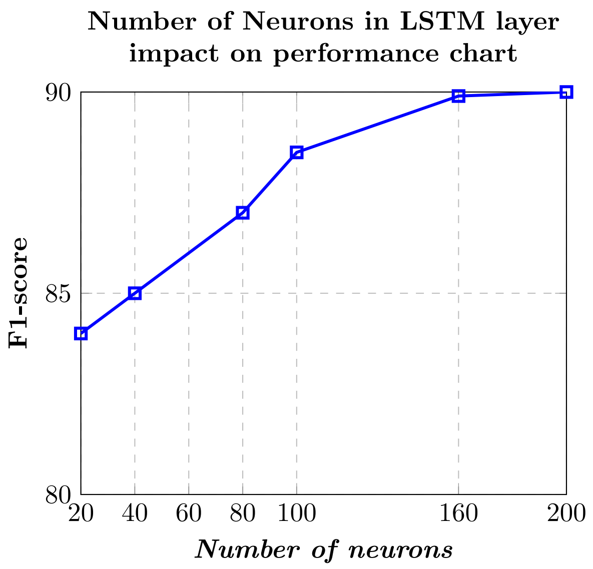


fig 4. Single LSTM layer window size 3, optimizer SGD

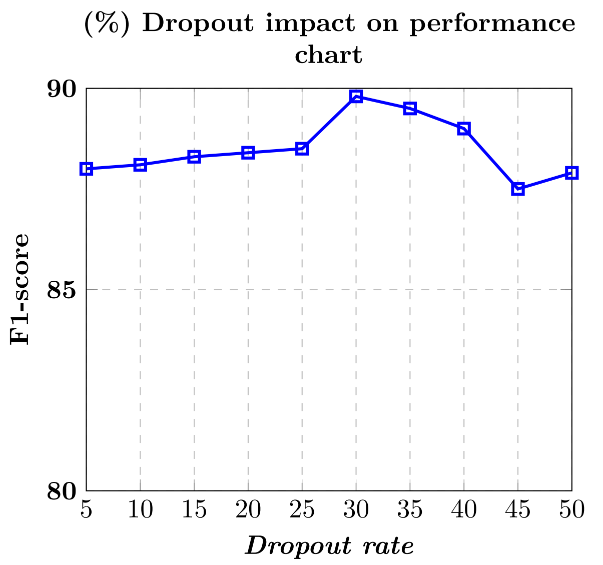


fig 5. Single LSTM layer with 100 neurons, 10 epochs, optimizer SGD

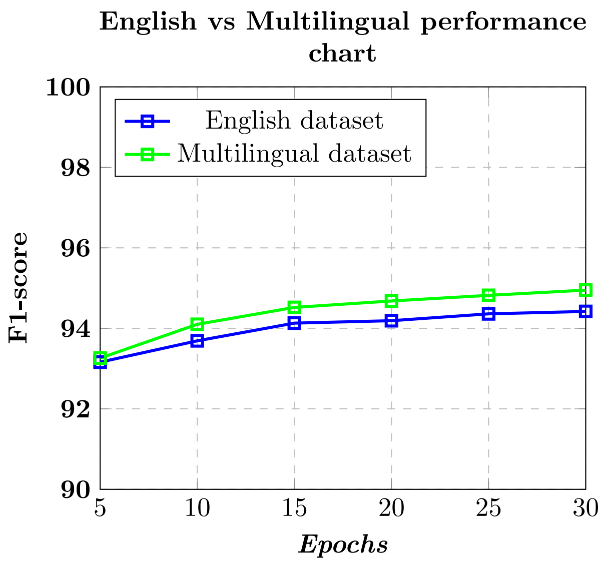


fig 6. Best Performing Model 2 LSTM layers, 30 epochs, optimizer SGD (*other params are as described in section 2*)