Natural Language Processing Assignment 1: POS Tagging

Eric Rossetto (eric.rossetto@studio.unibo.it)
Ganesh Pavan Kartikeya Bharadwaj Kolluri (ganeshpavan.kolluri@studio.unibo.it)
Salvatore Pisciotta (salvatore.pisciotta2@studio.unibo.it)

Abstract

This document serves to show the methodologies and the results of the first assignment of the NLP course: a **Part-of Speech** (POS) tagging as Sequence Labelling task on a dependency tree bank dataset using pre-trained GloVe embeddings. The POS tagging is an important NLP process where each word of a sentence is labeled with a particular tag that reflects the role of a particular term with respect to the context of the sentence. In particular, it has been asked to use four different neural architectures to predict a tag for each word. The relative performances has been analyzed by considering the overall qualities of the dataset and to possibly introduce further improvements.

1 Data preparation

The dataset provided was splitted in train, validation and test set accordingly to the indications given. Furthermore, each document has been divided into sentences by using as separator punctuation marks like ".", "?", "!". Given that we have to provide those sentences as input to a Recurrent Neural Network (RNN), we have to produce a tokenization for each sentence and provide an appropriate representation for the terms. In order to do so, we have used word-embeddings to represent terms in our dataset. A key step is to define a procedure to build an embedding matrix $M \in R^{|V| \times d}$, where V is the set containing all the terms present in the corpus and d is the dimension associated to the word-embedding vectors. The vocabulary provided by GloVe contains 400k unique words and an handy associated embedding, but anyway there may be words that are not present in such dictionary but that appear in our dataset. Such words, called *Out-of-Vocabulary* (OOV) words, need to be handled carefully. Indeed, since the pre-trained embedding model has associated a vector to each word in its vocabulary, we have decided to assign to each OOV word a randomly generated vector of dimension d sampled from a uniform distribution ranging in $[-0.5, 0.5]$. This is choice is strengthened by the fact that the number of OOV terms is approximately the 0.06% of total number of words in the corpus. Then, in order to build the matrix M, we have followed closely these steps:

1. Build vocabulary **V1**, which is made by the words present in both in the corpus and the pre-trained GloVe model.
2. Compute random embeddings for terms out of vocabulary **V1** (**OOV1**) of the training split.
3. Add the computed embeddings to the vocabulary, so to obtain vocabulary **V2 = V1 + OOV1**.
4. Compute random embeddings for terms **OOV2** of the validation split.
5. Add embeddings to the vocabulary, so to obtain vocabulary **V3 = V1 + OOV1 + OOV2**.
6. Compute random embeddings for terms **OOV3** of the test split.
7. Add embeddings to the vocabulary, so to obtain the final vocabulary **V4 = V1 + OOV1 + OOV2**.

At the end of the aforementioned procedure we obtained the embedding matrix M and two complete vocabularies (one for the terms and one for the tags) built sequentially, i.e. each OOV word taken from the validation set will be only added if and only if that word was not present in the training set (the same reasoning needs to be applied to the test and validation set).

2 Models

2.1 Input

Until now, the sentences got different lengths and since they have to be fed in input to the network they all need to be padded to be all equal in length. The maximum length is chosen in order to cover approximately 98% of the sentences, thus avoiding to discard too many terms (at testing time the length chosen was equal to 114). In this way, too long sentences were truncated at the maximum length whereas too short ones were filled with zeros up to the maximum length. The padding value has been added to the word and tag vocabulary accordingly. The resulting sentences were then one-hot encoded, resulting in a tensor of size $N \times \textit{max\_length} \times C$ that will be passed to the network. In the training phase, $N = N_{train}$ is the number of phrases and $C = C_{train}$ the number of unique tags present in the training split.

2.2 Model implementation

Alongside a *baseline* model other three networks were implemented using the **Tensorflow/Keras** framework. Here we report a brief description of them:

1. **Baseline model**: The baseline model is a neural network having two layers: a *Bidirectional LSTM* layer and a *Fully Connected* (FC) Dense layer.
2. **Bi-GRU+FC**: The second model is a variation of the baseline, in this case we have beforehand a *GRU* layer instead of a *LSTM* one.
3. **2xBi-LSTM+FC**: The third model tries to add more complexity to the baseline model by adding another *Bidirectional LSTM* layer on top the first one.
4. **Bi-LSTM+2xFC**: Similarly, the fourth model takes the baseline model and it adds on top of the *Bidirectional LSTM* a *FC* layer.

All the models have as first layer a non-trainable *Embedding* layer which transforms the words in their corresponding embeddings by taking the already pre-defined embedding matrix M as input. Moreover, each FC layer has as activation function a *softmax*, save the first layer in the **model 4** which has a *leaky relu* function. To prevent overfitting we have strongly relied on:

- *Dropout* layers by setting the chance of dropping input units.
- *Weight decay* technique by setting α in the rescaling factor $\frac{\lambda_i}{\lambda_i + \alpha}$.
- *Early stopping* by setting the minimum change **min_delta** in the validation loss and the **patience** which instead control the number of epochs with no improvements after which the training will be stopped.

During training it was monitored the both validation loss and the accuracy on both the training and validation sets. The loss function chosen was the *categorical crossentropy* with the objective to maximize the accuracy. Although the preferable metric to take care of is the F1 score, during training for the early stopping there was the necessity to use accuracy since it is not possible to distribute and aggregate F1 across batches. Training was made for all the models for N epochs.

2.3 Output

Due to the application of one-hot encoding on the input, the output of the models are expected to be tensors of dimension $N \times$ *max_length* $\times C$ where C is the number of different tags present in the dataset on which you compute the predictions.

3 Evaluation and results

Once the training was completed, the two best models are chosen by looking on the *macro-F1* scores on the validation set. The two mentioned models are then evaluated on the test set. The *macro-F1* score is computed on the ground truth labels and the predicted ones by looking token by token (after applying a flattening method on the output). In the evaluation, all the punctuation tags and symbols tags are not taken in consideration.

Here below we report the best results on *accuracy* and *macro-F1* obtained so far in the experimentation, even though they can noticeably vary between runs due to the random nature of the methods applied.

	Validation set		**Test set**	
	Acc	**Macro F1**	**Acc**	**Macro F1**
Baseline Model	0.8745	0.5164	0.8817	0.5454
Bi-GRU+FC	0.8788	0.4266	–	–
2xBi-LSTM+FC	0.8739	0.5151	–	–
Bi-LSTM+2xFC	0.8729	0.5632	0.8786	0.5871

Table 1: *Evaluation results on the validation and test set*

3.1 Evaluation pipeline

These scores are obtained as a result a fine tuning on the dropout rate and the regularization term. Indeed the we have looked both at the loss and the accuracy curves before applying parameters because in the first trials there were a strong signs of overfitting on the training set.Given that, we have decided to increment slightly the dropout rate in the range $[0.2, 0.7]$ with steps of 0.5, whereas the regularization term values are in $\{1, 1e^{-1}, 1e^{-2}, 1e^{-3}, 1e^{-4}\}$ following some hints on the literature. Modifying these values caused the greatest variations on the F1 scores.

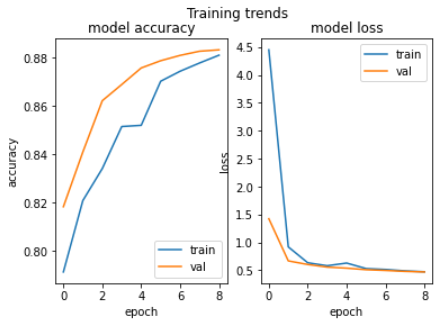

Figure 1: *Training trends of accuracy(left) and loss(right) of baseline model*

The other parameters were mostly tuned on the baseline model as reference for the remaining models. The size of the batches was chosen by trial and error on our model and dataset by choosing sizes between $8, 32, 64, 128$. Nevertheless, we have even tried to run our baseline model on batches of unit size with no success, even though with such size it was possible to compute the monitor the macro F1 during training.The number of hidden nodes was chosen in a similar manner, trying values like $64, 128$ and 256; fixating the same number of units on both the *LSTM* layers on the third model whilst in the fourth one we have cut in half the number of nodes in the second *FC* layer. Both the number of units and the size of batches could cause severe issues if not chosen properly.

The optimizer chosen is Adam with Nesterov Momentum (*Nadam*) with a learning rate of $1e^{-3}$ and a clipping of the gradient norm set to 10, in order to prevent the issue of exploding gradients (LSTMs only addresses the problem of vanishing gradients). The training was executed up to fifteen epochs with early stopping callback with 5 epochs of patience a minimum accepted delta of 0.1. The latter is a key element in preventing the mentioned overfitting issues and in our case to stop the execution whenever the loss started to increase unexpectedly.

An other important parameter to fix was the dimension of the word embedding vectors. The chosen value is 300 and it was empirically chosen because, among all other values (50,100 and 200), it was the one that at training and evaluation time that provided the best results.

Finally, by looking at results of our experimentation, the models that obtained the two best f1 scores on the validation were the baseline and the fourth one, whereas the best model on the test set is the model 4.

4 Error analysis and possible improvements

Even though, as already told in the Evaluation and Result section 3, results are quite influenced by randomness, still it is to draw some conclusions on some of the possible mistakes made:

- The model tuning process can be improved, provided more computational effort and time of execution.Also, increasing the epochs could work. There is handy tool from *Keras* called *Keras Tuner* that can be used to find the optimal hyperparameters combination.
- The bad predictions can be also be the result of a training done on a imbalanced and not so insightful dataset. For this reason, providing the networks with more data could lead to significant improvements on score results.
- Another possible reason is that the data maybe needs to be preprocessed a bit more.

We also tried to solve part of the intrinsic unbalance in the dataset by assigning more weight to less frequent classes, and distribute those weights to the words present in the corpus. The resulting vector was then passed as input to the baseline model to look for any improvements. Another possible approach is to exploit some of the parameters offered by *Keras*, like dropping some of the connections between recurrent units (see *recurrent dropout*).

5 Conclusions

To conclude, we are quite satisfied by overall working pipeline followed and also by the results obtained. We are aware of the fact that it is possible to obtain better results looking for one of the solutions already mentioned or search for other approaches to follow, but in general this will not affect the followed procedures.