HW1 - Report
NLP 2019/2020

Donato Crisostomi, 1754001

Abstract

In this document I present a Pytorch implementation of a bidirectional LSTM model with a CRF layer based on the work of (Lample et al., 2016) that exploits POS-tags and pre-trained word embeddings to reach state-of-the-art performance on an english corpus.

1 Introduction

Named Entity Recognition, like many other problems in NLP, was once dominated by ad-hoc approaches based on domain knowledge. Recently, it has been succesfully tackled with neural approaches. The most simple of these is to treat each token as a single decision problem, so the model could for example output the most probable tag for each input token. This model would nevertheless suffer from the possible many meanings of words. To mitigate this problem, context is added to the tokens; nevertheless, reasoning only about the word-tag relations is still not sufficient in case of sequence classification such as NER. For example, we would expect our model to give a poor score to the pair of subsequent tags (Person, Location) due to the english language syntax. In such scenarios where labeling decisions are intertwined, the tags can't be produced by independent decisions. What we can do is treat the whole sequence of tags as a tag itself, and pick the most probable sequence according to some score function. This brings us to *Conditional Random Fields*, which are explained in section 2.2.

2 Model

2.1 BiLSTM

Recurrent Neural Networks are the most used architectures for sequences; these summarize the context of a certain token m with a recurrently updated vector

$$\mathbf{h}_m = g(\mathbf{x}_m, \mathbf{h}_{m-1}), \quad m = 1, 2, \ldots, m$$

where $\mathbf{x}_m$ is the vector embedding of the token w_m and g defines the recurrence. Nevertheless, RNNs often fail to capture long-time dependencies; for this reason, LSTMs are often used. These employ a more complex recurrence, in which a memory cell goes through a series of gates, in fact avoiding repeated applications of non-linearity (Eisenstein, 2019). The hidden state $\mathbf{h}_m$ accounts for information in the input leading up to position m, but it ignores the subsequent tokens, which may also be relevant to the tag y_m; this can be addressed by adding a second LSTM, in which the input is reversed. This architecture is called *Bidirectional LSTM* and is the model I use in this work. See Implementation for the implementation details.

2.2 CRF layer

As stated in the introduction, *CRF* allows to reason over the tags of a sequence jointly. Therefore, we can obtain the tags sequence as

$$\hat{\mathbf{y}} = \underset{\mathbf{y} \in Y(\mathbf{w})}{\arg\max}\, \Psi(\mathbf{w}, \mathbf{y}) \tag{1}$$

where $\Psi(\mathbf{w}, \mathbf{y})$ is some scoring function. Since there are exponentially many sequences of tags, enumerating the score for each sequence is infeasible; to avoid this, we can decompose the score as a sum of scores over pairs of subsequent tokens.

$$\Psi(\mathbf{w}, \mathbf{y}) = \sum_{m=1}^{M+1} \psi(\mathbf{w}, y_m, y_{m-1}, m) \tag{2}$$

In order to capture relations among tags, we model the score of transitioning from a tag i to a tag j in addition to the emission scores $(word, tag)$. We put the transition scores in a matrix T. Given the local scores, we can compute eq. (2) via the *Viterbi* algorithm, which exploits the fact that the equation can be rewritten as a recurrence and thus be computed via dynamic programming. To obtain the probability for a given sequence $\mathbf{y}$ we can apply the softmax to obtain

$$P(\mathbf{y}|\mathbf{w}) = \frac{e^{\Psi(\mathbf{w},\mathbf{y})}}{\sum_{\hat{y}\in Y} e^{\Psi(\mathbf{w},\hat{\mathbf{y}})}} \tag{3}$$

The parameters can now be obtained by maximizing this probability, which is equal to minimizing its negative logarithm. The CRF implementation I used is based on pytorch-crf.

2.3 Embeddings

Pretrained word embeddings immediately boosted the model performance over random initialized ones. Among these, *Fasttext* (Bojanowski et al., 2016) performed best. In particular, the embeddings that gave best coverage were obtained from a wikipedia crawl and are 300-dimensional. To reduce this dimension I implemented the post-processing algorithm 1 suggested in (Raunak, 2017); this reduced significantly the complexity of the model, lowering the embeddings dimension of a multiplicative factor 3.

The final embeddings are also enriched with *Part Of Speech* information, concatenating the result of the word embedding with the one-hot representation of the word POS tag. These are obtained using the pretrained POS-tagger from Stanza.

2.4 Dataset

The dataset is highly unbalanced, as can be seen in fig. 3. The labels 'O' and 'ORG' are the hardest to discriminate, take for example the following sentence from the dataset: "*She also developed a friendship with Ian McCulloch of Echo and the Bunnymen.*" Here '*and*' and '*the*' from the band name should be both classified as 'ORG', but should be classified 'O' almost anywhere else.

2.5 Parameterization

In the training process, input sequences as tensors of indices are fed into an embedding layer, which is obtained as explained in section 2.3. The result is then passed through a dropout layer and is then concatenated to the one-hot representation of the sequence POS tags, thus becoming a tensor of size $B \times D \times N_p$ where B is the batch size, D the embedding dimension and N_p the number of POS tags. The resulting tensor is packed before being fed to the BiLSTM, which returns the concatenation of the forward and backward LSTM result. The final layer is a dense layer that maps the results to the tag space. The obtained tensor contains for each sentence in the batch, the score between any $(word, tag)$ pair. These will be the emission scores used by the CRF layer.

The CRF layer also keeps the transition matrix T introduced in section 2.2, which consists of $N_t \times N_t$ weights that will be the learned during the training phase, where N_t is the number of labels, plus two added tags for the start and end of the sentence. Furthermore, the model has to learn the weights of the dense layers.

2.5.1 Regularization

In order to prevent the model from overfitting, two kinds of regularizers have been employed, namely dropout and weight decay. The former consists of randomly setting some computation nodes to zero during training to prevent feature co-adaptation, while the latter prevents the size of the learned weights from growing too large.

3 Experiments

Adam resulted in the best performance among the optimizers. Check table 1 for the used parameters. A dropout layer with $p = 0.5$ has been applied to the output of the embedding layer, to the hidden layer of the LSTM and also to its output. The chosen LSTM architecture consists of 2 layers with hidden dimension 128, larger models tended to overfit. Weight decay λ has been set to 0.0001, bigger values slowed significantly the convergence. On average, a training epoch took 100 seconds, all the computations have been carried locally on a Dell XPS with 16GB ram, using a Nvidia GTX 1650 GPU.

3.1 Results

The presented model obtained an F_1 score of 0.92, the remaining scores are listed in fig. 1. The main boost in performance has come from the word embeddings, which boosted a poor F_1 to 0.84, then CRF got it from 0.84 to 0.89; the last 3 centesimals came with the combination of parameter tuning and information from the POS tags. As can be seen in the confusion matrix in fig. 2, the model is weaker in the minority classes, finding it most difficult to classify 'ORG', which is often confused with 'O'.

Resources

Implementation

The employed implementation is the one provided by Pytorch, in which for each element in the input sequence, each layer computes the following function:

$$\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
g_t &= tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot tanh(c_t)
\end{aligned}$$

The chosen LSTM architecture has 2 layers with hidden dimension 128.

Embedding post processing algorithm

Input: Word embedding matrix X, threshold parameter D
Output: Post-processed word embedding matrix X.
1. Subtract the mean:

$$X = X - mean(X)$$

2. compute the PCA components:

$$u_i = \text{PCA}(X), \text{ where } i = 1, 2, \ldots, D$$

3. Eliminate the top D components:
$\forall v \in X$:

$$v = v - \sum_{i=1}^{D}(u_i^T \cdot v)u_i$$

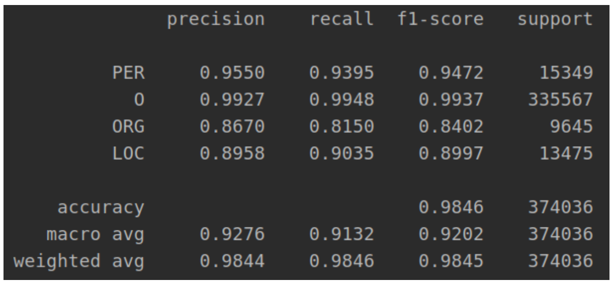
Algorithm 1: Post-Processing Algorithm

Parameters

amsgrad	True
betas	(0.9, 0.999)
eps	1e-08
lr	0.0005
weight	0.0001

Table 1: Optimization parameters.

Scores



```
              precision    recall  f1-score   support

         PER     0.9550    0.9395    0.9472     15349
           O     0.9927    0.9948    0.9937    335567
         ORG     0.8670    0.8150    0.8402      9645
         LOC     0.8958    0.9035    0.8997     13475

    accuracy                         0.9846    374036
   macro avg     0.9276    0.9132    0.9202    374036
weighted avg     0.9844    0.9846    0.9845    374036
```

Figure 1: Classification report.

Confusion matrix

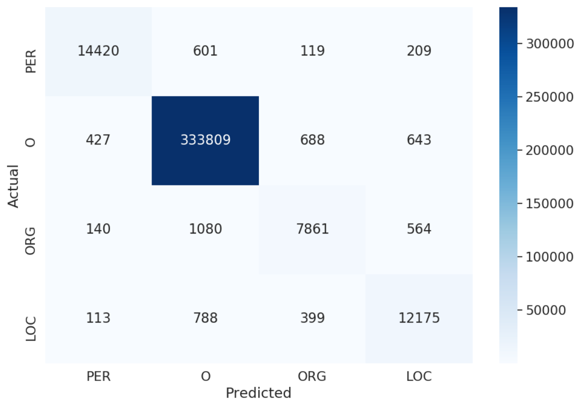


Figure 2: Confusion matrix.

Labels distribution

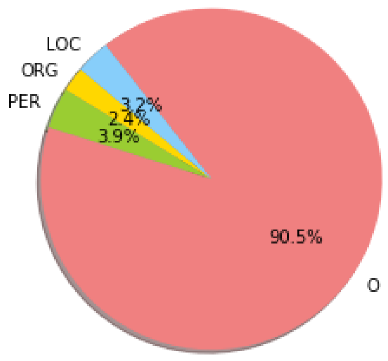


Figure 3: Labels distribution pie chart.

References

Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2016. Enriching word vectors with subword information.

Jacob Eisenstein. 2019. *Introduction to natural language processing*. The MIT Press.

Guillaume Lample, Miguel Ballesteros, Sandeep Subramanian, Kazuya Kawakami, and Chris Dyer. 2016. Neural architectures for named entity recognition.

Vikas Raunak. 2017. Simple and effective dimensionality reduction for word embeddings.