Fact Checking through Neural Language Inference

Eric Rossetto - (eric.rossetto@studio.unibo.it) - UNIVERSITY OF BOLOGNA
Salvatore Pisciotta - (salvatore.pisciotta2@studio.unibo.it) - UNIVERSITY OF BOLOGNA

Abstract

This document serves to show the methodologies and the results of the second assignment of the Natural Language Processing (NLP) course: a **Fact Checking** using Neural Language Inference (NLI) on a pre-processed version of FEVER dataset. Fact checking is an important NLP task that has the aim to assign a truth or false value to a claim in a particular context. Specifically, it has been asked to model the Fact Checking task as a binary classification problem. The FEVER dataset provides couples $(claim, evidence)$ that have to be fed into the network, to eventually learn the truth value associated. We tested several models that implement different merging techniques beneath the network, that is different models use different techniques that will be introduced in the paper. 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 is a pre-processed version of the FEVER dataset, in which all non-verifiable claims are filtered out and the reported evidence IDs have been replaced with the corresponding text from Wikipedia. Then the dataset has been split in train, validation and test set accordingly to the indications given. Furthermore, we decided to perform some pre-processing:

1. remove the **stopwords**, already collected in the NLTK library, excluding the negative ones in order to keep an hypothetically negative sense of a sentence,
2. transform the texts in lower case,
3. remove any special character,
4. remove any decimal digit, paying attention to keep important numbers like dates, etc,
5. remove possible duplicate words,
6. remove any extra left or right spacing (including carriage return) from text.

Given that we have to provide those sentences as input to different Neural Network models, we tokenized each pair, producing an unique vocabulary for both the context and the claims, and providing an appropriate representation for the terms. This operation has been done exploiting functionalities of the Keras tokenizer. We decided to use pre-trained GloVe word-embeddings to represent terms. 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 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 $[-0.1, 0.1]$. This choice is strengthened by the fact that the number of OOV terms is approximately 0.17% of total number of words in the corpus.

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 99% of the sentences, thus avoiding to discard too many terms (at testing time the length chosen was equal to 89). 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 Numpy arrays of size $N_{claims} \times$ *max_length* and $N_{evidences} \times$ *max_length* that will be passed to the network.

2.2 Model implementation

Four network configurations were implemented using the **Tensorflow/Keras** framework. Here we report a brief description of them:

1. **Last State-RNN model**: The first model is a neural network having two *Bidirectional* LSTM (Bi-LSTM) layers for each input, a *Merge* layer that takes the outputs of the twos Bi-LSTM and two FC layers.
2. **Mean of States-RNN model**: The second model is a variation of the first one but in this case the *Merge* layer process also the last hidden states produced by the Bi-LSTM layers.
3. **MLP model**: The third model has two FC layers that produces a representation to be fed into the *Merge* layer and as usual two FC layers to provide the final output.
4. **Bag of Vectors model**: The fourth model compute the sentence embedding as the mean of its token embeddings (**Bag of Vectors**) so it has just two reshape layers, the *Merge* layer and finally two FCs layer that, as usual, which provide the final output.

All the models have as first layers two *Input* layers and two *Embedding* layers, one for the claim and one for the correspondent evidence of the given input pair, which transforms the sentence in their corresponding embeddings by taking the already pre-defined embedding matrix M as input. Moreover, since we are modeling the problem as a binary classification task the last FC layer has a *Softmax* activation function; instead, the two FC layers in the **MLP model** have a *LeakyReLu* activation function. In order to improve the performance every model has, before the classification layer, two FC layers with the second one having the double numbers of units with respect to the first one and both having *LeakyReLu* as activation function and L2 regularizer. The aforementioned *Merge* layers are components of the model that the process the encoding produced by the previous layer to provide a proper sentence embedding. Say B is the current batch size and d the chosen embedding dimension, they are implemented in order to perform one of these operations:

- **Mean**: the classification input is given by the mean of evidence and claim embeddings. Its shape is $[B, d]$;
- **Sum**:the classification input is given by the sum of evidence and claim embeddings. Its shape is $[B, d]$;

- **Concatenation**: the classification input is given by the concatenation of evidence and claim sentence embeddings.Its shape is $[B, 2d]$.

A possible extension is to compute the *cosine similarity* metric between the claim and the associated evidence and concatenate it at the end of the obtained sentence embedding. In this way it is possible to see if some similarity information might be useful for the classification task. Since the cosine similarity is a scalar value the shape of the classification input is modified as:

- **Mean**:$[B, d + 1]$.
- **Sum**:$[B, d + 1]$.
- **Concatenation**:$[B, 2d + 1]$.

To prevent overfitting we have strongly relied on:

- *Dropout* layers by setting the chance of dropping input units.
- *Weight decay* technique , applied only on LSTM layers, 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 we have monitored the binary cross-entropy loss, accuracy, F1-score by additionally validating the data on a specific portion of the the FEVER dataset. The overall objective was to minimize the binary cross-entropy loss. Training was made for all the models for the same number of epochs.

2.3 Output

Given the already mentioned input, the output of every model is expected to be a bi-dimensional tensor since we are dealing with a binary classification task.

3 Training and evaluation

In order to find the best parameters for the training we decide to execute a *Keras Tuner* on all the models, this means that we looked for the best model that came out after letting the Keras tuner trying different combinations of hyper-parameters such as: regularizer impact, number of encoding and classification units, dropout rate, merge technique, kind of model, optimizer' learning rate and the presence of the cosine similarity.

We run the tuner in two different execution one enabling the concatenation of the cosine similarity term and one disabling it. We chosen by default to use a fixed embedding dimension of 300 and a batch size of 64.

The first model returned the best results, i.e., the *Last State-RNN model*, with the following configuration:

- **learning rate**: 0.001,
- **encoding units**: 300, **classification units**: 128,
- **merge mode**: sum,
- **dropout rate**: 0.3,
- **regularizer**: 0.0001,
- **cosine** applied.

In the table below we report the training results obtained applying these parameter found by the tuner to all the models:

	Train set		**Validation set**	
	Acc	**F1 score**	**Acc**	**F1 score**
Last State RNN	0.8626	0.8049	0.6987	0.6835
Mean of States RNN	0.8571	0.7925	0.6943	0.6793
MLP	0.8004	0.6729	0.6201	0.5844
BoV	0.7343	0.4234	0.5040	0.3351

Table 1: *Training results on the train set and validation set*

Once the training is completed, there are two ways to evaluate the performance of the classifiers:

- **Multi-input Classification Evaluation (MCE)** where we compute evaluation metrics such as accuracy, f1-score, recall and precision and we assess the performance of trained classifiers.
- **Claim Verification Evaluation (CVE)** where we want to give an evaluation that is related to the classification output of the claim itself. We consider pairs $(claim, evidence)$ having the same claim and through a majority voting we extract the the prediction on the claim. Once that we have extracted the prediction we compute the same classification metrics of the **Multi-input classification evaluation** method.

Here below we report the results obtained in the experimentation using both the evaluation methods.

	precision	**recall**	**f1-score**	**support**
SUPPORTS	0.63	0.92	0.74	3606
REFUTES	0.84	0.45	0.59	3583
macro avg.	0.68	0.68	0.68	7189
weighted avg.	0.73	0.68	0.67	7189

Table 2: *MCE Evaluation results on the test set for the best model.*

	precision	**recall**	**f1-score**	**support**
SUPPORTS	0.84	0.45	0.59	3304
REFUTES	0.63	0.92	0.74	3309
macro avg.	0.71	0.68	0.67	6613
weighted avg.	0.71	0.68	0.67	6613

Table 3: *CVE Evaluation results on the test set for the best model.*

4 Conclusions

We are quite satisfied by overall working pipeline followed and also by the results obtained. Though, by looking closely at both the architecture and the results we can say that:

- we could obtaine slightly better results with a larger model and more training epochs; also a more exhaustive search could be helpful,
- the addition of the attention mechanism could be beneficial, this is also supported by the fact that some of the State-of-The-Art techniques implement it,
- some data augmentation, more advanced pre-processing techniques (cfr. neural pre-processing) could also help in squeezing out some performances too.