Word-in-Context Disambiguation

Giuseppe Masi
Sapienza Univesity of Rome
Matricola 1962771
masi.1962771@studenti.uniroma1.it

1 Introduction

Since many words can be interpreted in multiple ways depending upon the context of their occurrence, this homework addresses a Word-in-Context Disambiguation task in which we are required to determine if a target word has the same meaning or not in two different sentences, without relying on a fixed inventory of word senses.

2 Word Embedding

I used the pre-trained word vectors GloVe [1] to perform word embedding. It allows us to represent each word of the sentence in a multi-dimensional space, in particular, I used the 300-dimensional variant.
By inspecting the provided datasets, I notice that the missing words are about 0.2% of the total words. Since this value is not that high, I addressed the missing words issue just ignoring that words.
I also tried to represent the *unknown words* with a (fixed, i.e. the same for all of them) random vector, obtaining approximately the same performances.

3 Pre-processing

The task is a binary classification task on a balanced dataset, so class balancement is not needed.
Before calculating the latent representation of the sentence, I perform on it the standard steps like lowercasing, removing punctuation, and removing stop-words.
Once the sentence is cleaned up by punctuation and stop-words, I discard instances in which (at least) one of the two sentences contains fewer words than a threshold.

4 Model Architecture

Both approaches share the overall architecture (Figure 1) in which the first layer is the word embedding layer.
Successively there is the layer in which the latent representation of the pair of sentences is calculated. Even if this is the layer in which the two approaches differ, both of them compute the latent representation of the sentences independently of each other, and then they are concatenated. On top of this layer, there is the classification layer (MLP) with input the latent representation of the sentences and output the *True/False* value. The classifier is also composed of a dropout layer, proven to be an effective technique for prevent overfitting [2]. The loss function used is the *Binary Cross Entropy*.
So, the two approaches differ on how the latent representation of the single sentence is computed.

4.1 First Approach

The first approach uses a word embedding aggregation function to get a prediction. In particular, once each word is represented by a vector we can represent the sentence performing an aggregation function on the set of vectors of the words in the sentence.
The aggregation function is performed component by component of the vectors, so it returns a vector in the same space in which the words are represented.
Once obtained the latent representation of the two sentences, I concatenated them and I used a classification layer to get a prediction.
The classifier is composed of 2 linear layer and a dropout layer. I also tested more complex architecture for the classifier, with additional linear layers, but the performances are not better w.r.t. the simpler architecture. I think this is because the key to obtain high performance is not the classifier on top of the model, but the hidden representation of the sentences.
All the hyper-parameters of this approach are shown in Table 2.
I implemented the following aggregation function:

mean, sum and weighted mean.

Weighted mean This variant of the mean aggregation function is performed by using a specific set of coefficients to be multiplied by words. It is computed ad-hoc for each sentence to give greater weight to words close to the target word, and less weight to words distant from the target word, following an exponential distribution.

4.2 Second Approach

The second approach makes use of an LSTM (*Long Short-Term Memory*) to obtain a hidden representation of the sentence. In general, RNNs exploit sequence level semantics and it is useful to obtain a contextual representation of each word in the input sentence.
The rest of the model is about the same as the first approach, so concatenating the two hidden representation of the sentences and using a classifier to get a prediction.
In this case, the missing words are represented by a (fixed) random vector and the sequences are padded (with a special token corresponding to a zeros vector) to perform batch computing.
First I implemented the many-to-one strategy, taking the output of the LSTM corresponding to the last word of the sentence.
Successively, with the purpose of exploiting the contextual information of the target word as much as possible, I implemented a Bidirectional LSTM. It allows us to consider the words in the sentence sequentially both from first to last and vice versa.
I tested two strategies about which hidden representation of sentence consider for the classification purpose, in particular:

- **Last-First summary**: concatenate the forward summary corresponding to the last word in the sentence and the backward summary corresponding to the first word in the sentence.
- **Target summary**: concatenate the forward and the backward summary both corresponding to the token word in the sentence.

Table 3 show all the hyper-parameters of this approach.

5 Results

First Approach: Mean It is the simplest idea of all but it points out a well hidden presentation of the sentence. With this approach, I achieved an accuracy of 67% on the Development Dataset.

First Approach: Sum Almost the same performances are obtained by using the *sum* aggregation function, achieving an accuracy of 66%.

First Approach: Weighted Mean The *weighted mean* aggregation function achieves significantly worse performance, about 60%. I think it is because words that are far from the target word still have relevance in the meaning of it (and of the sentence).

Second Approach: LSTM I expected to obtain better results by using a sequence encoder like LSTM, but the accuracy achieved is lower w.r.t. the first approach results. In fact, this approach reaches 64% of accuracy, which is also the best performance among the approaches that use a sequence encoder.

Second Approach: Bi-LSTM This experiment makes use of a Bi-LSTM taking the *Last* forward summary and the *First* backward summary, achieving an accuracy equal to 64%.

Second Approach: Bi-LSTM, target summary This experiment makes use of a Bi-LSTM taking the forward summary and the backward summary corresponding to the target word token. It achieves significantly worse performance w.r.t. the previous sequence encoder approaches, about 55%.

All the trends of the accuracies on the Development Dataset, across the training epochs, are shown in Figure 2 for the first approach and in Figure 3 for the second approach, and summarized in the Table 1.

6 Conclusions

I experimented that simplest approaches like the first one achieve better performances than the LSTM-using approaches.
An issue of using the Bi-LSTM could be that the pad tokens are taken into account when the LSTM computes the backward summary. I tried to address this problem by using a zeros vector to represent the pad token and set the *bias* hyper-parameter of the LSTM to *False*. I also tried to set the *batch size* to 1 (so no pad is added) obtaining quite similar performances.
One improvement could be to assign a POS (Part Of Speech) tag to each word in the sentence to improve the performance.

References

[1] Jeffrey Pennington, Richard Socher, and Christopher Manning. “GloVe: Global Vectors for Word Representation”. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1532–1543. DOI: 10.3115/v1/D14-1162. URL: https://www.aclweb.org/anthology/D14-1162.

[2] Nitish Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

7 Figures

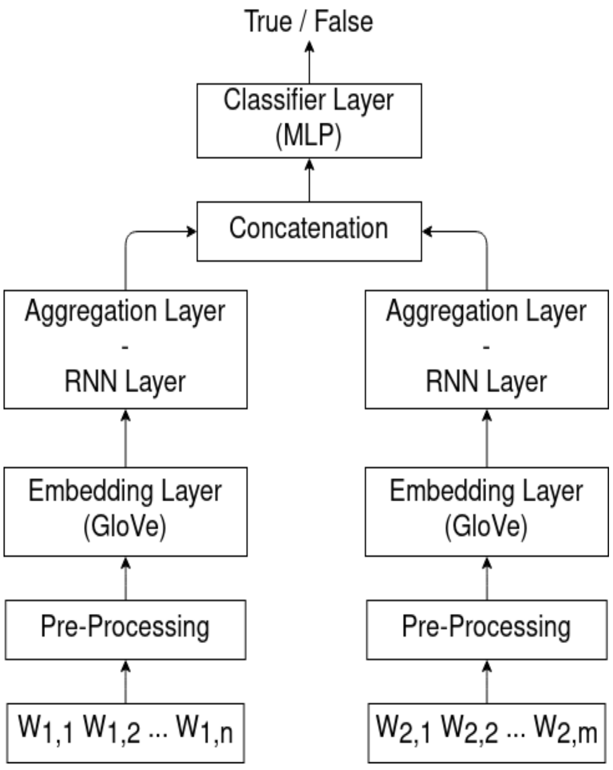
True / False
Classifier Layer (MLP)
Concatenation
Aggregation Layer - RNN Layer
Aggregation Layer - RNN Layer
Embedding Layer (GloVe)
Embedding Layer (GloVe)
Pre-Processing
Pre-Processing
$W_{1,1}$ $W_{1,2}$ ... $W_{1,n}$
$W_{2,1}$ $W_{2,2}$ ... $W_{2,m}$

Figure 1: The overall model architecture.

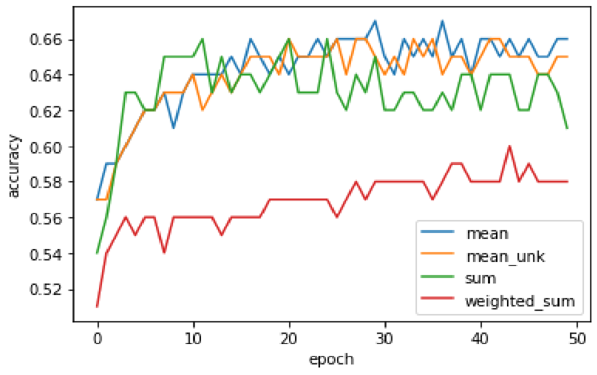

Figure 2: Trend of the accuracy by using the first approach with different aggregate functions across the epochs.

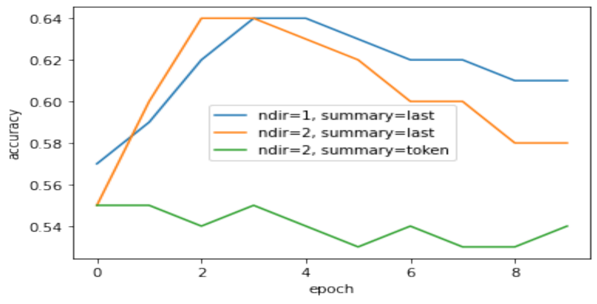

Figure 3: Trend of the accuracy by using the second approach with different LSTM settings across the epochs.

8 Tables

Model	**Accuracy**	**Epoch**
Mean	**67%**	29
Mean_unk	66%	20
Sum	66%	11
Weighted Mean	60%	43
LSTM	64%	3
Bi-LSTM	64%	2
Bi-LSTM, target	55%	1

Table 1: Max accuracies obtained from each model.

Hyper-parameter	**Tested values**
Base	**2**/e
Min length	3/**4**/5
Dropout	0.2/**0.3**
Optimizer	**Adam**/SGD
Learning rate	$1e-1/1e-2$/**1e-3**/$1e-4$

Table 2: First approach hyper-parameter tested (in bold that one that give best performance).
The *Base* hyper-parameter is the base used to calculate the exponential distribution of the coefficients for the *weighted mean*.

Hyper-parameter	**Tested values**
Min length	3/**4**/5
Hidden size	**512**
Num layer	**1**/2
Bias	**True**/$False$
Dropout	**0.2**/0.3
Optimizer	**Adam**/SGD
Learning rate	$1e-1/1e-2/1e-3$/**1e-4**

Table 3: Second approach hyper-parameter tested (in bold that one that give best performance).