# 

 Assignment 4:

Dependency Parsing

Natural Language Processing

Tzanis Nikolaos

mtn2217

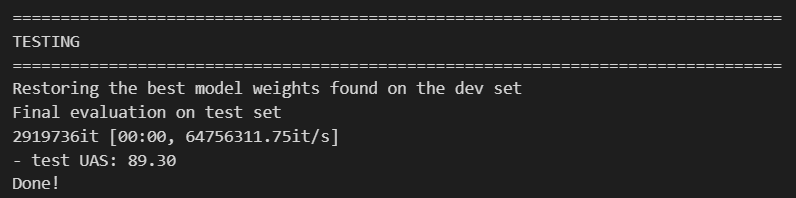
**Exercise A: Transition-Based Dependency Parser**

**Question 1**

The first task is to run the code with the given parameters without any adjustments and report the UAS (Unlabeled Attachment Score). The code was run locally with the command:

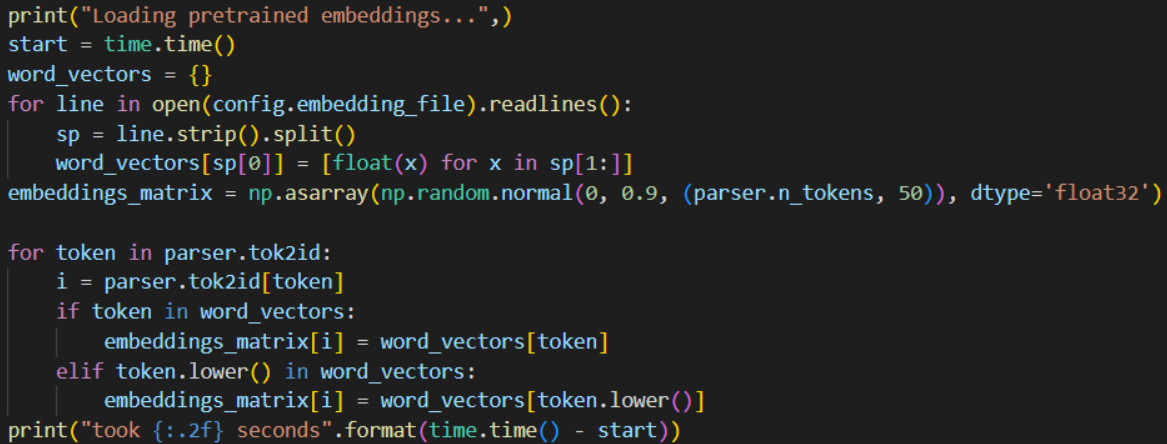


After 10 epochs of training, the UAS achieved was 89.3:

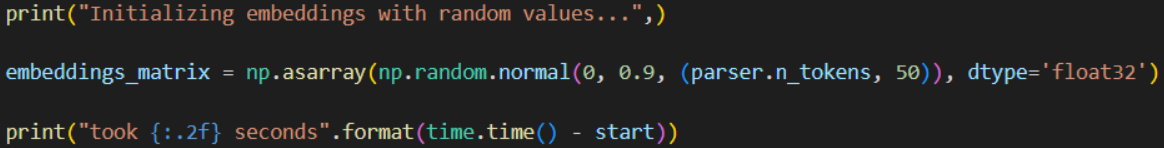


**Question 2**

The next task was to check the UAS while not using pretrained embeddings. All word embeddings are randomly initialized and then trained during the 10 epochs. The changes that achieve the above on the given code involve this part, located in the load\_and\_preprocess\_data function of the parser\_utils.py file:



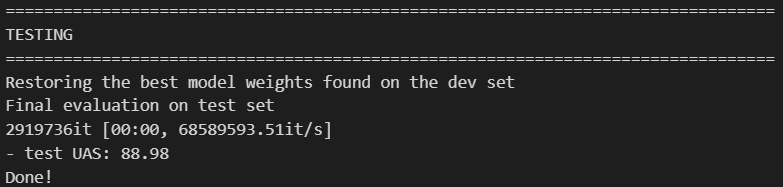
The above code was replaced with this one:



Then we run the code again with the command:



And get the following result:

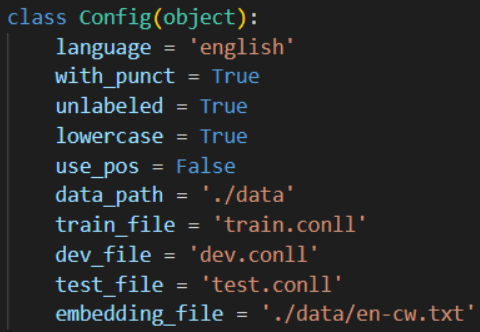


As we can see the above modifications slightly reduced the UAS to 88.98.

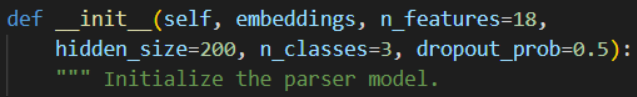
**Question 3**

For the next part all changed applied in the previous question were reverted and then, in order to stop using pos tags, we set the variable “use\_pos” to False in the parser\_utils.py file. Then we also have to change the n\_features variable to 18 from the original value that was 36.

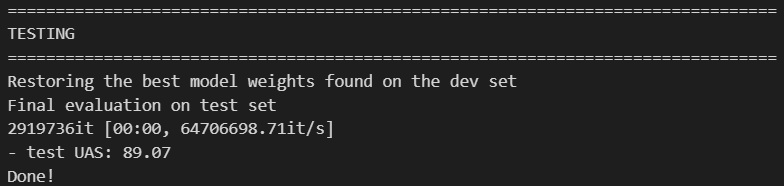
The 2 changes applied are seen below:



And:



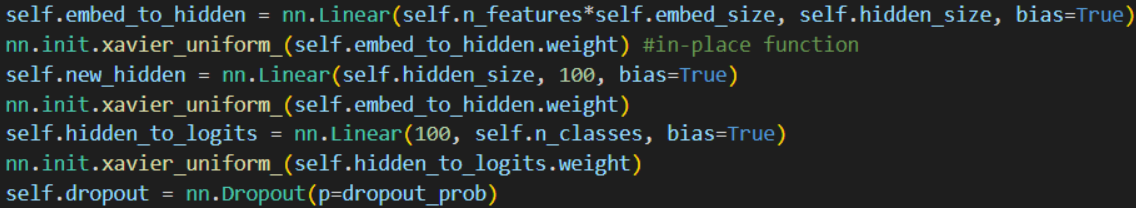
Executing the code again with the same command this time we get this result:



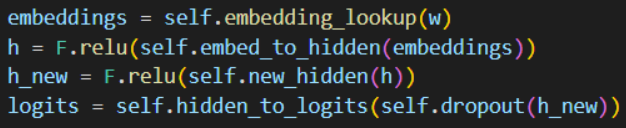
These adjustments improved the UAS slightly compared to the question 2, but the score is still lower than the one in question 1.

**Question 4**

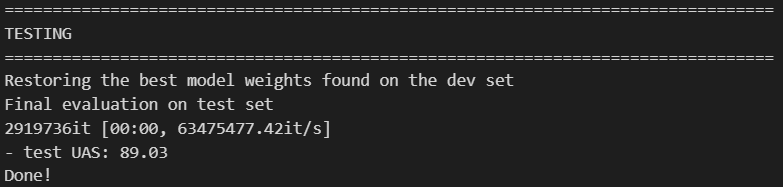
In this question we are adding a second hidden layer of size 100 over the first one. This new layer will take as input the output of the ‘embed\_to\_hidden” layer and output 100. The forwards function has to be adjusted too in order to use the relu activation function. The changes are seen below:



And



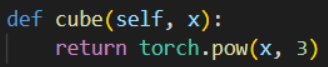
Running the code again with these changes we get this result:



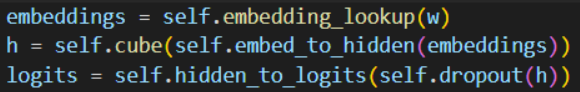
So, this new UAS is lower than the one in question 3 but higher than the one observed in question 2.

**Question 5**

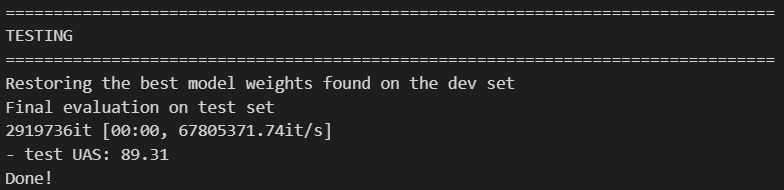
In the final question of Exercise A we have to define the cube activation function first:



Then we have to set this function to be used in the forward function:



Running the code again after the above implementation we get this result:



The final result is slightly better than the first one which makes it our best one yet.

**Exercise B: Graph-Based Dependency Parser**

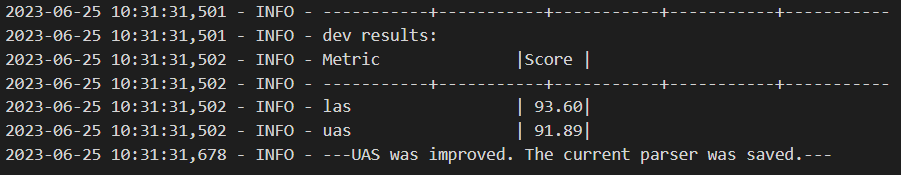
**Question 1**

This entire exercise requires the use of GPU in order to handle the quite length training times so all code was uploaded and ran on Kaggle’s platform for running code[[1]](#footnote-1). Also the number of epochs was changed from the default 5 to 3 as each epoch needed upwards of 40 minutes and I wanted to speed up the execution of the code.

For the first question the task was to run the code with 1 BiLSTM layer. For that purpose we have to provide a value for the n\_lstm\_layers param when executing the command line prompt. The following command sets the number of lstm layers to 1 and the number of epochs to 3.

!python /kaggle/input/exercise-b/B/main.py --train\_path /kaggle/input/exercise-b/B/data/train.conll --dev\_path /kaggle/input/exercise-b/B/data/dev.conll --test\_path /kaggle/input/exercise-b/B/data/test.conll --n\_lstm\_layers 1 --epochs 3

After executing the code with the above command, we get this result:



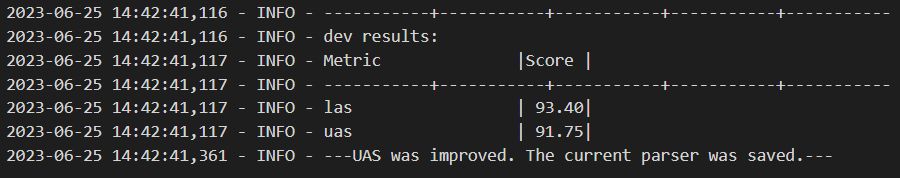
This result is a lot better than the ones observed on all models of Exercise A.

**Question 2**

For this question we are going to use the pretrained embeddings Glove-6B-100d. Thankfully these embeddings already exist as a dataset on Kaggle and that’s where they’re imported from. After adding them as a dataset to the current notebook, we run this command to use them in our code:

!python /kaggle/input/exercise-b/B/main.py --train\_path /kaggle/input/exercise-b/B/data/train.conll --dev\_path /kaggle/input/exercise-b/B/data/dev.conll --test\_path /kaggle/input/exercise-b/B/data/test.conll --n\_lstm\_layers 1 --ext\_emb /kaggle/input/glove6b100dtxt/glove.6B.100d.txt --epochs 3

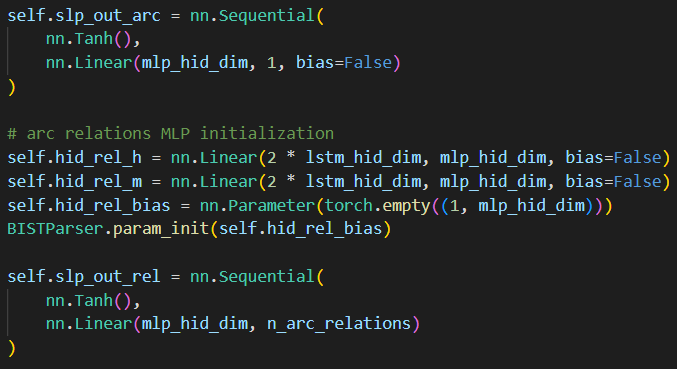
Using the above embeddings we get this result:



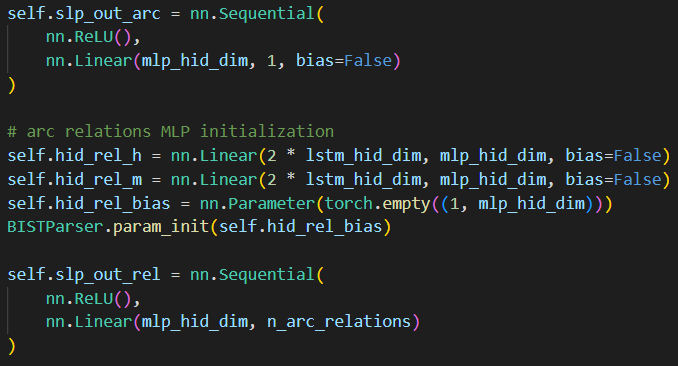
The UAS is higher than the models of Exercise A but slightly lower than the one we observed in Question 1.

**Question 3**

In this question we have to use the ReLU activation function instead of Tanh. In order to do this, we have to change the code in two points. Specifically, at the model.py file, at this point:



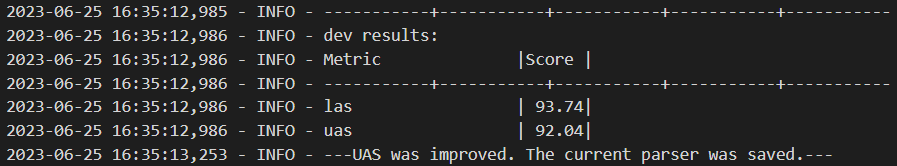
Both Tanh activation functions will be replaced with ReLU like this:



Then running this command again:

!python /kaggle/input/exercise-b-question-3/B/main.py --train\_path /kaggle/input/exercise-b-question-3/B/data/train.conll --dev\_path /kaggle/input/exercise-b-question-3/B/data/dev.conll --test\_path /kaggle/input/exercise-b-question-3/B/data/test.conll --n\_lstm\_layers 1 --epochs 3

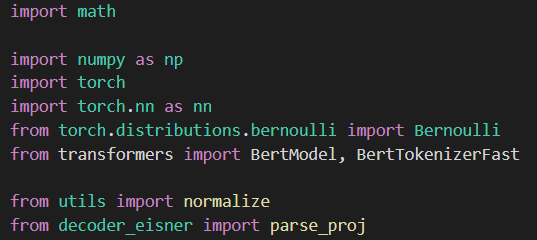
We get these results:



There is a slight improvement from the previous questions of Exercise B and still a lot higher score compared to Question A.

**Question 4**

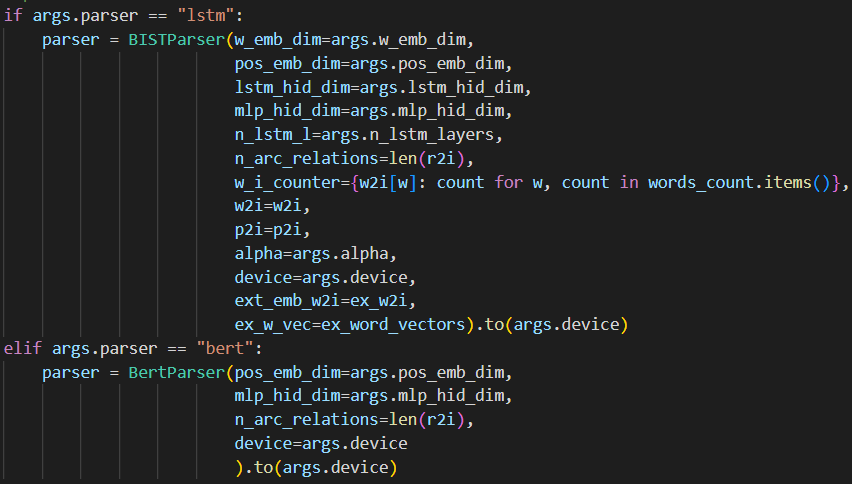
In order to use the BERT model in our code first of all we have to import it from the transformers library together with the BERT Tokenizer at the model.py file.



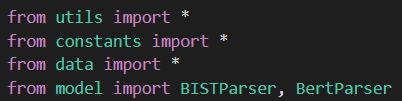
Then we are going to allow the choice of parser by the command line while running the code. For this the main.py file has to be adjusted:



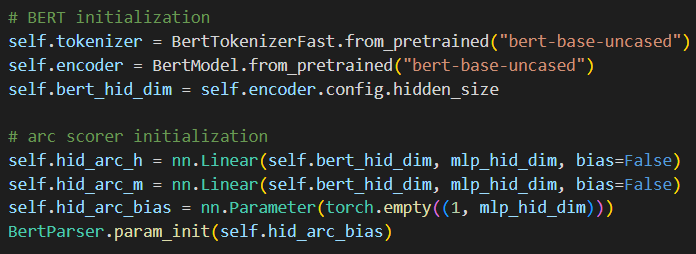
Then, still at the main.py file, depending on the parameter passed, the appropriate parser will be used:

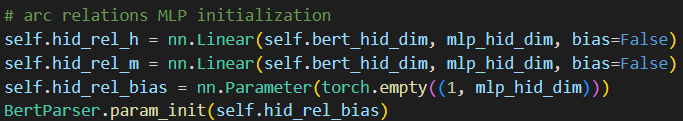


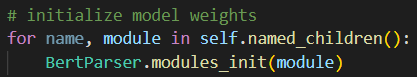
Of course, the BertParser has to be implemented in the model.py class and imported to the main.py file like this:



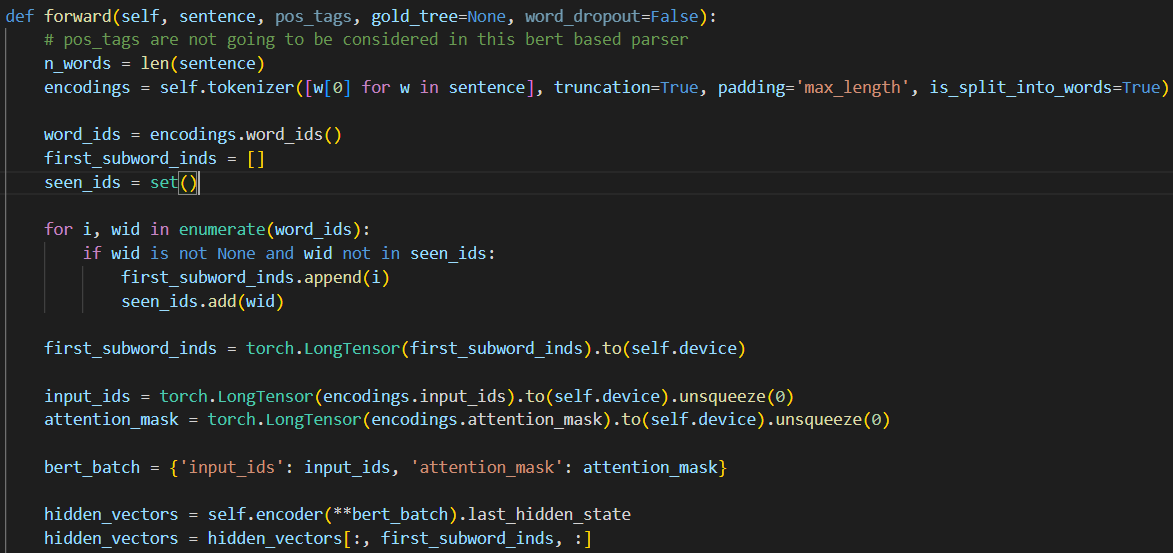
The following code implements the BertParser in the model.py file (only the differences from the BISTParser are included):







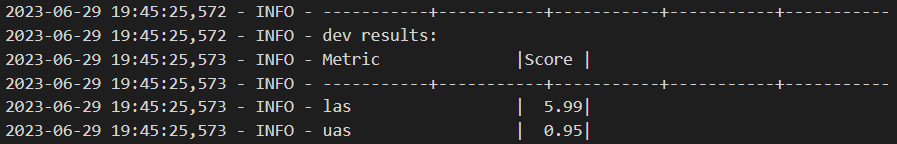
And at the forward method:



The next part is to run the code with the BERT model from the command line. By running this command:

!python /kaggle/input/exercise-b-question-4-bert/B/main.py --train\_path /kaggle/input/exercise-b-question-4-bert/B/data/train.conll --dev\_path /kaggle/input/exercise-b-question-4-bert/B/data/dev.conll --test\_path /kaggle/input/exercise-b-question-4-bert/B/data/test.conll --n\_lstm\_layers 1 --epochs 3 --parser bert

We get this result:



Which is the worst one yet by far. From the above we can see that even though the BERT model is especially useful in many cases, in this particular problem it probably requires a lot of fine-tuning in order to get good results.

**Question 5**

To begin with, we observed that the Graph-based dependency parsers in Exercise B performed better than the Transition-based dependency parsers in Exercise A, specifically in terms of UAS. Among the Transition-based models, the final (question 5) model achieved the highest performance with a UAS of 89.31. This means that replacing the ReLU function with the Cube activation was the only factor that managed to improve the results of the first model.

On the contrary, at the graph-based models of Exercise B, the replacement of Tanh with ReLU did not improve the UAS, while the pretrained embeddings in Question 3 provided some improvement. Substituting the LSTM encoder with the BERT encoder led to significantly worse results, indicating that this model may require more fine-tuning to perform well in this context.

Transition-based parsers aim to predict a sequence of actions (transitions) that can be used to construct a dependency tree. They have a relatively simple complexity, making them efficient, but they are susceptible to error propagation, as a wrong prediction early in the sentence can affect subsequent predictions.

On the other hand, Graph-based parsers aim to learn a scoring function for dependency trees and identify the highest scoring tree for a sentence. They analyze the entire sentence, allowing them to capture more complex dependencies. However, this complexity also makes the parser slower compared to the Transition-based parser.

1. <https://www.kaggle.com/code> [↑](#footnote-ref-1)