Natural Language Processing

Prodromos Kampouridis MTN2203
University of Piraeus
National Center for Scientific Research “DEMOKRITOS”
Athens, Greece

20 June 2023

Professor

Efstathios Stamatatos,

Dept. of Information and Communication Systems Engineering,

University of the Aegean

ABSTRACT

This project was assigned as part of the coursework for the Natural Language Processing course in the Master’s program in Artificial Intelligence at NCSR “Demokritos” and the University of Piraeus. This last assignment focuses on the task of Dependency Parsing which aims to detect dependencies between phrases of a sentence to determine its grammatical structure. Specifically, we will explore two kinds of Deep Learning models, the Transition-based, and Graph-based dependency parsers.

KEYWORDS

Transition-Based; Graph-Based; Dependency Parser; Unlabeled Attachment Score (UAS); Labeled Attachment Score (LAS)

A. TRANSITION-BASED DEPENDENCY PARSER

In this part of the assignment, we will explore a Transition-based dependency parsing model which is based on a feed-forward neural network architecture from the work of Chan & Manning (2014). This model detects unlabeled dependencies, meaning that the type of dependency is not considered. As part of the assignment, we will explore various modifications in the model’s architecture, and their effect in the performance of the model, in terms of UAS evaluation metric.

Initially, we modified the *parser_model.py* and *utils/parser_utils.py* files to implement the changes described in the questions. Moreover, we enriched the *run.py* script with additional arguments that control the behavior we want to achieve in each question.

1. First, we run the experiment in its original form, by executing **run.py** without any flags and the model’s performance on the test set is presented in the table below.

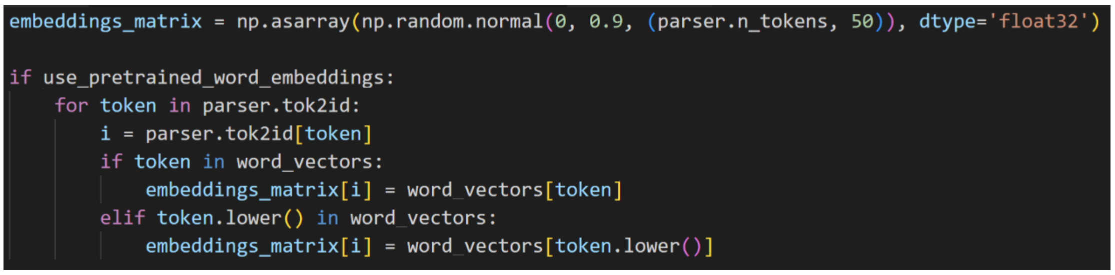
Command
python run.py

Table 1: The command we ran for the question A1

Test UAS
89.19

Table 2: Unlabeled Attachment Score (UAS) on the test data for the question A1

2. Thereafter, we run the experiment with random word embeddings rather than pretrained. To achieve this, we execute the *run.py* script with the flag **--disable_pretrained_embeddings**. Consequently, *__main__* calls the *load_and_preprocess_data* function with *use_pretrained_word_embeddings=False*. Inside the *load_and_preprocess data* function, we have modified the following snippet in the code:



```
embeddings_matrix = np.asarray(np.random.normal(0, 0.9, (parser.n_tokens, 50)), dtype='float32')

if use_pretrained_word_embeddings:
    for token in parser.tok2id:
        i = parser.tok2id[token]
        if token in word_vectors:
            embeddings_matrix[i] = word_vectors[token]
        elif token.lower() in word_vectors:
            embeddings_matrix[i] = word_vectors[token.lower()]
```

In this snippet the *embeddings_matrix* is initialized with random vectors. Then for each token in the vocabulary, if there exists a word embedding for this token, the corresponding vector is replaced by the pretrained vector. Since POS tags do not appear in the *word_vectors* vocabulary, they remain randombly initialized. We modified the code by adding the *if use_pretrained_word_embeddings* condition. If this condition is False, the usage of pretrained embeddings is skipped and all the embeddings (word and pos) are randomly initialized.

Command
python run.py --disable_pretrained_embeddings

Table 3: The command we ran for the question A2

After running the experiment, the performance of the model on the test is presented in the following table.

Test UAS
87.98

Table 4: Unlabeled Attachment Score (UAS) on the test data for the question A2

Using random word embeddings instead of pretrained ones leads in a decline in performance compared to the previous question.

3. Next, to disable the usage of POS features, we execute the run.py script with the flag **--disable_pos_features**. We have modified the *load_and_preprocess_data* function to receive the *use_pos_features* parameter. Internally, *load_and_preprocess_data* replaces the value of the *config.use_pos* variable, with the value of the *use_pos_features*. The same change in Config also happens during the creation of the *Parser* object inside the *load_and_preprocess_data* method, by again passing a *use_pos_features* parameter. In addition, inside *main* we create the *ParserModel* by explicity setting *n_features=parser.n_features* which is 18 when POS features are disabled, instead of 36 when enabled.

Command
python run.py --disable_pos_features

Table 5: The command we ran for the question A3

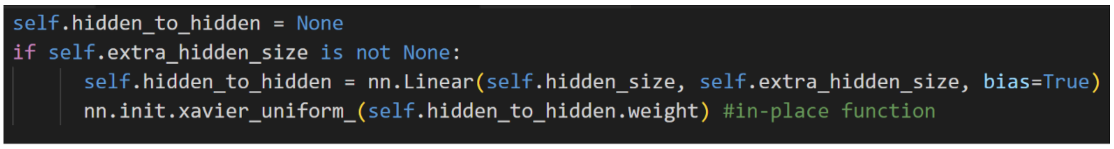
Now, after running this experiment, the performance of the model is presented below, respectively.

Test UAS
87.40

Table 6: Unlabeled Attachment Score (UAS) on the test data for the question A3

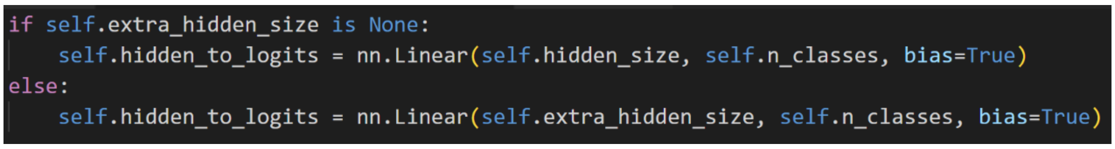
Compared to the result of the first question (UAS: 89.19), the performance declines when the model does not use the POS features.

4. To complete this question, we modified the *ParserModel* class in *parser_model.py* file, so that its constructor receives one extra parameter, the *extra_hidden_size*. If its value is None, the model remains in its initial form. Otherwise, an extra hidden layer is created, named *hidden_to_hidden*, with input size equal to the output size of the previous hidden layer (*embed_to_hidden*) and output size equal to *extra_hidden_size*, as shown in the following code.



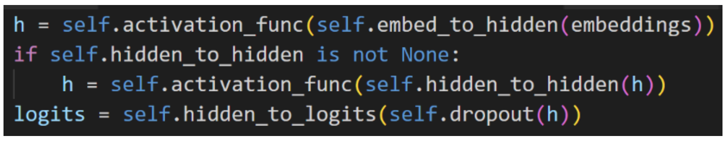
```
self.hidden_to_hidden = None
if self.extra_hidden_size is not None:
    self.hidden_to_hidden = nn.Linear(self.hidden_size, self.extra_hidden_size, bias=True)
    nn.init.xavier_uniform_(self.hidden_to_hidden.weight) #in-place function
```

Additionally, we change the input size of the output layer to *extra_hidden_size*, i.e. 100, as shown in the else clause below.



```
if self.extra_hidden_size is None:
    self.hidden_to_logits = nn.Linear(self.hidden_size, self.n_classes, bias=True)
else:
    self.hidden_to_logits = nn.Linear(self.extra_hidden_size, self.n_classes, bias=True)
```

Finally, we apply the extra hidden layer (*hidden_to_hidden*), in the forward function, and apply the activation function, which is ReLU for this question.



```
h = self.activation_func(self.embed_to_hidden(embeddings))
if self.hidden_to_hidden is not None:
    h = self.activation_func(self.hidden_to_hidden(h))
logits = self.hidden_to_logits(self.dropout(h))
```

To activate the extra hidden layer in the experiment, we execute the *run.py* script with the flag **--enable_extra_hidden** , which consequently creates a ParserModel with *extra_hidden_size=100* inside the *__main__* method.

Command
python run.py --enable_extra_hidden

Table 7: The command we ran for the question A4

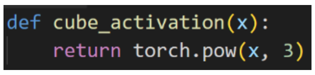
Now, after running the experiment the performance of the model is presented in the following table.

Test UAS
89.19

Table 8: Unlabeled Attachment Score (UAS) on the test data for the question A4

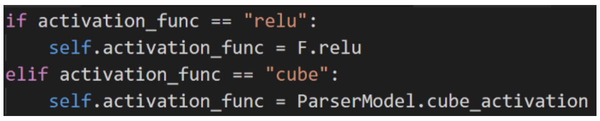
Compared to the first question, the model's performance remains unchanged with the addition of an extra hidden layer.

5. To answer the last question, we implement the cube activation function inside the ParserModel class, with the following definition.



```
def cube_activation(x):
    return torch.pow(x, 3)
```

We have modified the constructor of the *ParserModel* class, to accept an extra parameter *activation_func*. Inside the constructor, the *self.activation_func* property of the class is set to either *F.relu* or *cube_activation, depending* on the value of *activation_func* as shown below



```
if activation_func == "relu":
    self.activation_func = F.relu
elif activation_func == "cube":
    self.activation_func = ParserModel.cube_activation
```

Next, in *forward()*, instead of calling directly relu or cube, we use the *self.activation_func* property of the class, which contains the desired activation function.

To use the *cube* function in this experiment, we execute the *run.py* script with the argument **--activation_function=cube**.

Command
python run.py --activation_function=cube

Table 9: The command we ran for the question A5

Lastly, after running the experiment we present the performance of the model in the table below.

Test UAS
87.38

Table 10: Unlabeled Attachment Score (UAS) on the test data for the question A5

The model's performance is lower when using the cube activation, compared to the question 1 with UAS equal to 89.19.

B. GRAPH BASED DEPENDENCY PARSER

In this part of the assignment, we will explore a Graph based dependency parser based on the work of Kiperwasser and Goldberg (2016). The model is based on a bidirectional LSTM encoder and two MLPs. One for predicting a score for each possible dependency, and one to detect the type of the dependency. As part of the assignment, we will explore various modifications in the model's architecture, as well as the change of the Bi-LSTM encoder with a BERT encoder, and the effect of these changes in the performance of the model, in terms of UAS and LAS evaluation metrics.

Initially, we have modified the *model.py* and *main.py* files to implement the changes described in the questions. Moreover, we have created a new *model_bert.py* file to implement the BERT based model on question B4.

1. Firstly, we run the experiment with 1 BiLSTM layer, by executing *main.py* with the argument **--n_lstm_layers 1**. To evaluate the model's performance on the test set, we then execute the *main.py* script with the **--do_eval** and **--model_dir** arguments. As the *model_dir*, we pass the path of the saved model.

Command
python main.py --n_lstm_layers 1
python main.py --n_lstm_layers 1 --do_eval --model_dir "results/ds=ptb_epochs=5_lr=0.001_seed=1234_extEmb=False_wDim=100_pDim=25_lstmDim=125_mlpDim=100_lstmN=1_date=06_20_2023"

Table 11: The commands we ran for the question B1

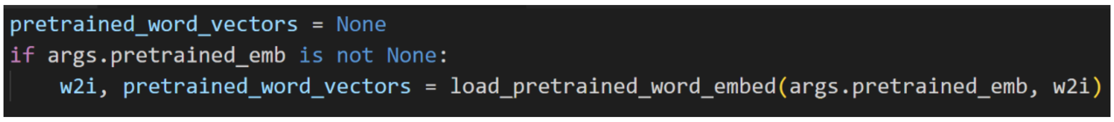
The model's performance on the test set is presented in the table below.

Test UAS	Test LAS
93.56	92.05

Table 12: The Model's performance on the test set for question B1

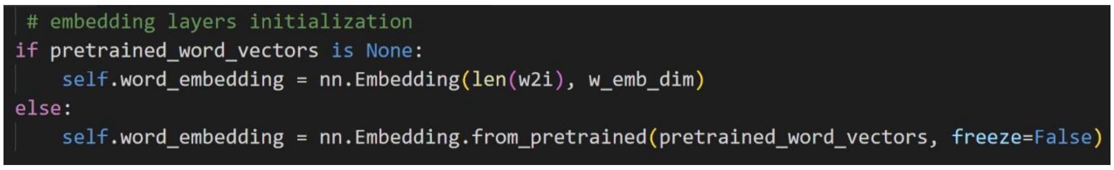
This model performs significantly better compared to the best models of part A (A1 and A4) which both achieved UAS 89.19.

2. To replace the randomly initialized embeddings with pretrained embeddings, we first added an extra command line argument in *main.py* named **--pretrained_embed**. It is important to note that *main.py* has already available the **--text_emb** argument, which however does not replace the randomly initialized embedding with the ones in the given file, but rather concatenates the given embeddings along with the randomly initialized ones and the POS embeddings. We modified *main.py* so that when the --pretrained_embed is used, the *load_pretrained_word_embed* util function is called with input the given pretrained embedding file *glove.6b.100d.txt* and the already calculates word-index mapping. We use the returned w2i value of the function to replace the previous w2i variable, and thus restrict the vocabulary to contain the words with available embeddings, as follows.



```
pretrained_word_vectors = None
if args.pretrained_emb is not None:
    w2i, pretrained_word_vectors = load_pretrained_word_embed(args.pretrained_emb, w2i)
```

Then, we create the BISTParser with an extra parameter *pretrained_word_vectors* that contains the embeddings matrix. Finally, we modified the BISTParser so that when the *pretrained_word_vectors* is passed, the *word_embedding* layer is initialized with the pretrained embeddings rather than random ones:



```
# embedding layers initialization
if pretrained_word_vectors is None:
    self.word_embedding = nn.Embedding(len(w2i), w_emb_dim)
else:
    self.word_embedding = nn.Embedding.from_pretrained(pretrained_word_vectors, freeze=False)
```

Command
!python main.py --n_lstm_layers 1 --pretrained_emb ./glove.6B.100d.txt
!python main.py --n_lstm_layers 1 --pretrained_emb ./glove.6B.100d.txt --do_eval --model_dir "results/ds=ptb_epochs=5_lr=0.001_seed=1234_extEmb=False_wDim=100_pDim=25_lstmDim=125_mlpDim=100_lstmN=1_pretrained_emb=True_activation=tanh_encoder=lstm_date=06_21_2023"

Table 13: The commands we ran for the question B2

The model's performance on the test set is presented in the following table.

Test UAS	**Test LAS**
93.71	92.22

Table 14: The Model's performance on the test set for question B2

Again, this model outperforms the models of part A, that achieved UAS 89.19, by an even larger margin.

3. We modified the BISTParser class in the *model.py* file to take *activation_function* as a parameter in the constructor. Then, we create the *self.slp_out_arc* and *self.slp_out_rel* layers with *nn.Tanh* if the name of the given activation function is *tanh*, or with *nn.ReLU* if the name is relu. We have also added an extra command line argument *activation_function* in the *main.py* file, which passes the given activation function (default: tanh) to the BISTParser constructor during the parser initialization. To train and evaluate the model with ReLU activation function, we run the following commands:

Command
python main.py --n_lstm_layers 1 --activation_function relu
python main.py --n_lstm_layers 1 --activation_function relu --do_eval --model_dir="results/ds=ptb_epochs=5_lr=0.001_seed=1234_extEmb=False_wDim=100_pDim=25_lstmDim=125_mlpDim=100_lstmN=1_pretrained_emb=False_activation=relu_encoder=lstm_date=06_21_2023"

Table 15: The commands we ran for the question B3

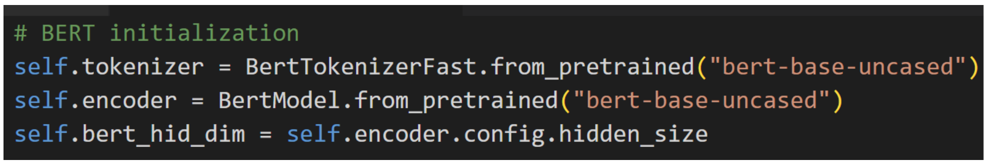
The model's performance on the test set is presented in the table below.

Test UAS	**Test LAS**
93.66	92.16

Table 16: The Model's performance on the test set for question B3

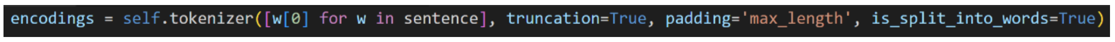
The model performs significantly better than the models of Part A that achieve UAS 89.19.

4. To implement the BERT based parser, we created a new file *model_bert.py* with a BertBISTParser inside. In the constructor we create a *BertTokenizerFast* and a *BertModel*, both based on the *bert-base-uncased* version of BERT.



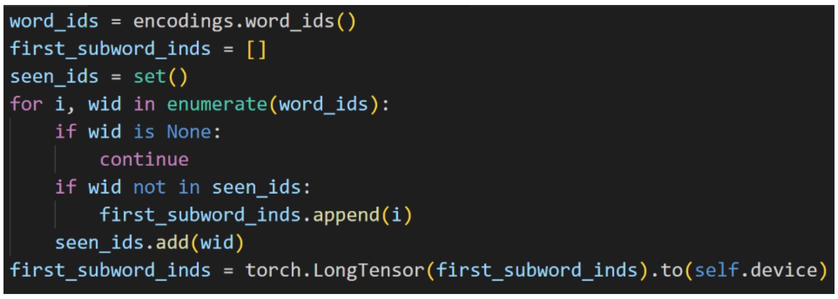
```
# BERT initialization
self.tokenizer = BertTokenizerFast.from_pretrained("bert-base-uncased")
self.encoder = BertModel.from_pretrained("bert-base-uncased")
self.bert_hid_dim = self.encoder.config.hidden_size
```

We also modify the MLPs so that their input size matches the hidden layer size of BERT.



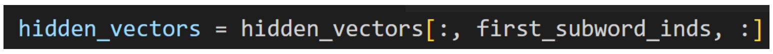
```
encodings = self.tokenizer([w[0] for w in sentence], truncation=True, padding='max_length', is_split_into_words=True)
```

In the *forward* method, we tokenized the (already split into words) sentence, using the bert tokenizer. At this point, we also keep the positions of the first sub-words of each word, as follows:



```
word_ids = encodings.word_ids()
first_subword_inds = []
seen_ids = set()
for i, wid in enumerate(word_ids):
    if wid is None:
        continue
    if wid not in seen_ids:
        first_subword_inds.append(i)
    seen_ids.add(wid)
first_subword_inds = torch.LongTensor(first_subword_inds).to(self.device)
```

After passing the encoded sentence through bert, we keep the last hidden states of BERT that correspond to the first sub-words of each word:



```
hidden_vectors = hidden_vectors[:, first_subword_inds, :]
```

Finally, these hidden states pass through the classification MLPs.

To enable the bert model, we added an argument **--encoder bert** in main.py. We also set the learning rate to 1e-5 since the default one did not provide good results. It's important to note that due to limitations in Google colab GPU availability, the BERT-based model was trained for 2 epochs.

Command
python main.py --encoder bert --lr 1e-5
python main.py --encoder bert --do_eval --model_dir="results/ds=ptb_epochs=5_lr=1e-05_seed=1234_extEmb=False_wDim=100_pDim=25_lstmDim=125_mlpDim=100_lstmN=2_pretrained_emb=False_activation=tanh_encoder=bert_date=06_22_2023"

Table 17: The commands we ran for the question B4

The model’s performance on the test set is presented in the table below.

Test UAS	Test LAS
96.19	94.52

Table 18: The Model’s performance on the test set for question B4

Interestingly, the Graph-based dependency parser with BERT encoder outperforms both the models of part A and the LSTM based models of Part B.

5. To conclude, we observe that Graph-based dependency parsers of Part B outperformed the Transition-based dependency parsers of Part A, in terms of UAS. Regarding the Transition-based models, the initial model achieved the best performance (UAS: 89.19). Adding one extra layer in the MLP provided the same results, whereas the other changes performed worse. Regarding the Graph-based model with LSTM encoder, the use of ReLU instead of tanh provided a small improvement over the model of B1 (UAS: 93.66 vs 93.56) and using pretrained embeddings instead of randomly initialized provided an even larger but still relatively small improvement over B1 (UAS: 93.71 vs 93.56). Finally, changing the LSTM encoder with the BERT encoder, had the largest impact in performance, by achieving UAS 96.19 vs 93.71 which is the next best performance. The Transition based parsers aim to predict a sequence of actions (transitions), based on which a dependency tree can be produced. They have relatively simple complexity and thus are efficient, but they are prone to error propagation since an early wrong prediction in the sentence can affect the subsequent predictions. On the other hand, the Graph-based parsers try to learn a dependency tree scoring function and detect the highest scoring dependency tree for a sentence. They analyze the whole sentence and thus can extract more complex dependencies. However, this makes the parser more complex and thus slower than the Transition based parser.