An Implementation of Standard Arc Neural Dependency Parser

Subject: CMSC 673: Introduction to Natural Language Processing

Graduate Assessment 3

Author: Kota Vinay Kumar (BM21496)

Dependency parsing is a technique where we get information about how words are related to each other in a sentence. By parsing a sentence into a tree containing different nodes and children, it would be helpful to understand what words to focus on. This can in turn help to generate a promising language model. My implementation here consists of the use of a shift-arc model. A standard arc dependency parser takes the help of graph-based representation to divide a word list into different dependencies.

The algorithm that I used was based on the works by (Chen & Manning, EMNLP 2014) and (Nivre, IWPT 2003). The neural model that I have implemented is a knock on what (Chen & Manning, EMNLP 2014) implemented in their paper.

According to the authors above, a transition-based dependency parser consists of three main elements. Given a training set that contains a pre-defined dependency tree, we should be able to extract a feature set of the form (V, A). Here V is the vertices and A is the action to be performed on the given state (that includes a stack, buffer, and dependency relations.).

Algorithm of Neural Arc Dependency Parser

The algorithm for this parser as stated in the 3SLP book starts at an initial configuration where the stack is empty, dependencies are empty and a buffer is the list of words. 3 operations are performed on the words, these operations are guided through a method called an oracle, which is the heart of the parsing. The 3 operations are shift, left arc and right arc respectively. A shift operation is done when we need to send an element from the buffer to the stack. A left arc is the technique of association with the elements to the left of the current word. A right arc is a technique of associating the current stack word with the right element of the stack.

At a given state, the oracle decides to choose between shift, left arc and right arc. As we already have a standard gold dependency tree available from the Universal Dependencies, our oracle refers to this gold parse tree to make a uniform decision. This is the first step of

feature extraction. With the actions as the labels and the configuration as the features, we need to create a neural model to make the model learn. For creating the rich feature set, as (Zhang & Nivre, ACL 2011) mentioned, we can choose the top elements from the stack as well as a buffer for each iteration and run it through a neural model. The learning can be done through cross-entropy loss.

Implementation of the Dependency Parser

First, I imported the necessary libraries (eg: conllu, Keras, pandas). I have created a function called "**get_info_deps**" which gives the overall information of the words, pos tags, dependency relation types etc. The training data will run over than function and give us the necessary outputs. This is helpful during the embedding process. I have not used any pre-trained embeddings like Glove. Next, I create a function called state, which is the main core of our application. The state function is the place where the transition operations occur and will in turn give us stack, buffer, and dep relations for a token list. The function "**process_token_list**" helps generate our training instances, where it returns a state and an operation to be performed. We will use the state objects from this list to generate our features.

Contrary to 3SLP (Chen & Manning, EMNLP 2014) I did not include the POS tags for the parsing (It is one of the challenges that I have encountered). I have selected 2 words from the stack and two words from the buffer. This is our input feature representation. The output labels are generated from the same "**process_token_list**" function.

Leveraging the knowledge, I have on TensorFlow/Keras for the projects during my undergrad and the M.P.S data science course, I have created a Keras dense model with three layers and a loss function of cross entropy. I found Keras to be easier to implement than PyTorch.

Next, I created two data frames with the inputs that are generated above and output labels. The outputs are encoded using Keras utils, and the training data frame was converted to a NumPy array and then fed into the model.

After multiple changes following the train and error method, the proposed solution is the best I could come up with. It chooses shift operation when it cannot perform left arc. It chooses the left arc when the right arc cannot be performed. Overall, the model has a bias towards applying the left arc.

The gold deps are created using a user-defined function, which gives the tuples in the format (head, token, deprel). This format is then compared to what we get from the state class and will be used to find the metrics.

As mentioned in the 3SLP book, I used UAS and LAS as the evaluation metrics for the parser. UAS stands for Unlabeled Attachment Score, tell us how much each word is linked to a correct head. LAS stands for Label attachment Score tells us how much per cent the label is correctly predicted (Metrics are mentioned below).

Challenges faced during development

The main challenge I faced was generating the feature set using the embeddings. I was unable to incorporate pos tags into the feature selection due to design constraints. The model here gives an accuracy of about 35 per cent, which is very low compared to many state-of-the-art models, but this is a baseline model which can be improved in the future. For this reason, I only used 4 features instead of 6 as mentioned in 3SLP or 18 as mentioned in (Chen & Manning, EMNLP 2014).

The time taken for the model to train is very less, but the time it took to predict and apply an action into the dev and test splits is more than one long hour.

Next, the English UD gave me a LAS score of 0.35, whereas the Estonian EWT gave me just a 0.06 LAS score.

I was not able to run the model through test set, since it took more than an hour couple of times and failed at that time, which made it very difficult to understand the root cause of the issue.

Below are the Model evaluation Metrics on the dev data

Metrics

```
Epoch 1/10
352/352 [==============================] - 6s 16ms/step - loss: 1.3175
Epoch 2/10
352/352 [==============================] - 5s 13ms/step - loss: 0.7002
Epoch 3/10
352/352 [==============================] - 5s 13ms/step - loss: 0.5692
Epoch 4/10
352/352 [==============================] - 5s 13ms/step - loss: 0.4977
Epoch 5/10
352/352 [==============================] - 5s 14ms/step - loss: 0.4514
Epoch 6/10
352/352 [==============================] - 5s 13ms/step - loss: 0.4196
Epoch 7/10
352/352 [==============================] - 8s 22ms/step - loss: 0.3951
Epoch 8/10
352/352 [==============================] - 5s 13ms/step - loss: 0.3774
Epoch 9/10
352/352 [==============================] - 5s 16ms/step - loss: 0.3595
Epoch 10/10
352/352 [==============================] - 5s 15ms/step - loss: 0.3485
<keras.callbacks.History at 0x7f5ee000c2b0>
```

Figure: The trained Model Loss rate for each epoch on the English EWT UD

The LAS score for the English dev data is:

1. 0.53 (when compared with the number of predictions made by the parser)
2. 0.35 (when compared with the gold dep relations structure)

The UAS score for the English dev data is:

1. 0.43 (when compared with the number of gold deps)

==========

```
Epoch 1/10
124/124 [==============================] - 3s 23ms/step - loss: 2.9182
Epoch 2/10
124/124 [==============================] - 2s 18ms/step - loss: 2.0412
Epoch 3/10
124/124 [==============================] - 2s 18ms/step - loss: 1.8519
Epoch 4/10
124/124 [==============================] - 2s 18ms/step - loss: 1.7606
Epoch 5/10
124/124 [==============================] - 3s 27ms/step - loss: 1.7082
Epoch 6/10
124/124 [==============================] - 2s 18ms/step - loss: 1.6794
Epoch 7/10
124/124 [==============================] - 2s 18ms/step - loss: 1.6607
Epoch 8/10
124/124 [==============================] - 2s 18ms/step - loss: 1.6476
Epoch 9/10
124/124 [==============================] - 2s 18ms/step - loss: 1.6378
Epoch 10/10
124/124 [==============================] - 2s 18ms/step - loss: 1.6293
<keras.callbacks.History at 0x7f5e7ae9a040>
```

Figure: The loss rates for each epoch on the Estonian EWT UD

The LAS score for the Estonian dev data is:

3. 0.12 (when compared with the number of predictions made by the parser)
4. 0.06 (when compared with the gold dep relations structure)

The UAS score for the Estonian dev data is:

1. 0.14 (when compared with the number of gold deps)

Summary

In my observation, I found out that the model was able to predict the embedding outputs correctly, but the difference occurred in the operations performed for each prediction. The model developed was able to predict correctly most of the time, with the limited amount of configuration I think the model did a good job. If we include another relevant feature such as the one mentioned in the 3SLP, then it could classify better and give the correct dependency parsing tree.

References

[1] Joakim Nivre. 2003. An Efficient Algorithm for Projective Dependency Parsing. In *Proceedings of the Eighth International Conference on Parsing Technologies*, pages 149–160, Nancy, France.

[2] Yue Zhang and Joakim Nivre. 2011. Transition-based Dependency Parsing with Rich Non-local Features. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 188–193, Portland, Oregon, USA. Association for Computational Linguistics.

[3]Danqi Chen and Christopher Manning. 2014. A Fast and Accurate Dependency Parser using Neural Networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 740–750, Doha, Qatar. Association for Computational Linguistics.