CSE 538: NLP Assignment 3 Report
Transition-based dependency parsing

By
Rutvik Parekh
SBU ID: 112687483

1) Model Implementation

a) Arc standard algorithm:

The algorithm used for obtaining a dependency parse tree is the arc-standard algorithm which is described in the paper "Incrementality in deterministic dependency parsing." The arc-standard algorithm has the following components in a configuration:

a) A stack,
b) A buffer, and
c) A set of dependency arcs.

The arc-standard algorithm is implemented in the parsing_system.py file in the apply() method. First, I get the top element and the second top element from the stack. If the transition is right-arc, I add the dependency arc from the second top element to the top element and remove the top element from the stack. If the transition we get is left-arc, then I add the dependency arc from the top element to the second top element and remove the second top element from the stack. If we get neither left-arc or the right-arc as the transition, then it must be a shift transition, and I perform the shift action to shift one element from the input buffer to the stack. In the end, the updated configuration is returned. I used various methods defined in configuration.py to implement the algorithm.

b) Feature extraction:

The feature extraction part is performed in the data.py file in the method get_configuration_features(). This method is implemented as given in the paper "A fast and accurate dependency parser using neural networks."

As described in the paper, first I create empty lists for words, postags, and labels. Then, I first iterate through the stack to get the top three elements (s1, s2, and s3) of the stack. I add the words and postags of those elements to the corresponding lists.

Then, I iterate through the buffer to get the top three elements of the input buffer (b1, b2, and b3) and I add the words and postags of those elements to the corresponding lists.

Then, I iterate through the stack again to get the first and the second leftmost and rightmost child of the top two elements of the stack. I add the words, postags and the labels of those elements to their corresponding lists. I also get the leftmost of the leftmost and the rightmost of the rightmost child of the

top two elements of the stack and put the words, postags and the labels of those elements in the corresponding lists. In the end, I append the filled words, posTags and labels lists to the features list. I used various functions from configuration.py and vocabulary.py to implement feature extraction.

c) Cubic activation function:

As described in the paper, cubic activation function just takes the tensor and returns the cube of the tensor vector. I simply used tf.pow(vector, 3) to implement it.

d) Neural Network architecture:

The neural network architecture is implemented in model.py in the class DependencyParser. In the '__init__()' method, I initialize the embeddings, weight1, and weight2 as described in the paper.

For weight1, the dimensions will be ((num_tokens*embedding_dim) x hidden_dim). It is calculated using the function np.random.standard_normal() to get the normal distribution. The distribution is also divided by the square-root of (num_tokens*embedding_dim*hidden_dim) to get the values in range. Weight1 is created using the function tf.Variable(). Finally, it is stored in the variable self._W1.

Similarly, for weight2, the dimensions will be (hidden_dim x num_transitions). It is also calculated using the function np.random.standard_normal() to get the normal distribution. The distribution is also divided by the square-root of (hidden_dim*num_transitions) to get the values in range. Weight2 is also created using tf.Variable(). Finally, it is stored in the variable self._W2.

Embeddings are of the shape (vocab_size x embedding_dim) which will contain the embeddings for all the words in the vocabulary, along with the embeddings of postags and labels, as described in the paper. I use the function tf.random.uniform() to get the embeddings in the uniform distribution in the range of -0.01 to 0.01. Embeddings are also created using tf.Variable(). Finally, it is stored in the variable self.embeddings.

In the call() function, I get the embeddings for the 'inputs' vector using the tf.nn.embedding_lookup() function. Then, we need to reshape the input embeddings to the size of (batch_size x (num_tokens*embedding_dim)) to multiply it with the weight1 tensor. We get the weight1 tensor through the variable self._W1. The result of this multiplication is a tensor of size (batch_size x hidden_dim). This tensor is then multiplied with self._W2 (weight2 tensor) to get the logits. These logits are stored in output_dict["logits"]. Next, we compute the loss if labels are present using the compute_loss () function.

e) Loss function:

Firstly, I apply the mask for the labels. For values which are -1 (infeasible labels) in labels, I simply put zeros in its place. Then, I calculate the exponential of the logits using tf.exp() function for calculating the softmax. Then, I calculate the softmax by using the function

tf.divide_no_nan(). I divide the exponentiated tensor by the sum of the exponentiated tensor. (The sum is calculated using the tf.reduce_sum() function). Then, I calculate the log by using the function tf.math.log(). I also used the function tf.clip_by_value() to clip the values between 1e-10 & 1 to prevent the problem of NaN. Then, I multiply the logartihms with the labels to get the loss vector, which is then reduced to sum and then reduced to mean using the functions tf.reduce_sum() and tf.reduce_mean() respectively. Then, I multiply the final output with -1 using the function tf.scalar_mul(). Finally, regularization term is calculated by multiplying the summation of l2_loss of W1 and W2 to self._regularization_lambda. The summation of loss and regularization term is returned as the training loss.

2) Experiments and Analysis

1) **Activations:**
   a) Cubic: UAS score: 87.72
   b) Tanh: UAS score: 86.84
   c) Sigmoid: UAS score: 85.52

   As we can see, the highest UAS score of **87.72** is achieved by using the cubic activation function described in the paper. The tanh activation function performs worse than the cubic function at **86.84.** The worst performing of the three is clearly the sigmoid activation function at **85.52**. The cube function achieves approx. 0.88% improvement over the next best nonlinear function tanh() over the development dataset. This proves the effectiveness of the cube activation function empirically. The other metrics UASnoPunc, LAS & LASnoPunc were also **1-2%** lower than the one with cubic activation.

2) **Pretrained Embeddings:**
   UAS score for without pretrained embeddings: **86.05**

   The activation function used here is cubic, which is the default activation function. So, on the development dataset, the accuracy achieved is **86.05** which is around 1.67% lower than the accuracy of **87.72** which was achieved with pretrained embeddings using the cubic activation function. The other metrics UASnoPunc, LAS & LASnoPunc were also **1-2%** lower than the one with pretrained embeddings.

3) **Tunability of embeddings:**
   UAS score for without tunability of embeddngs: **85.01**

   Using frozen embeddings lead to a UAS score which is 85.01, 2.71% lower than with tunability of embeddings. This score is also 1.04% lower than 86.05, which we get by

using no pretrained embeddings. The other metrics UASnoPunc, LAS & LASnoPunc were also **1-2%** lower than the one with tunability of embeddings.

Hence, we get the best UAS score by using the cubic activation function and with tunability of embeddings and by using the pretrained embeddings.

3) Model used for generating test_predictions.conll

As we get the best accuracy for the cubic activation function, I have used the **basic model** for generating the test predictions by using the following command:

python predict.py serialization_dirs/basic data/test.conll --predictions-file test_predictions.conll