NLP Assignment 3: Transition Parsing with Neural Networks
CSE 538 Fall 2019

Anmol Shukla, SBU ID - 112551470

15^{th} November 2019

1 Model Implementation

1.1 Arc-standard Algorithm

As described in the paper by Nivre, we use a straight-forward parsing system that has the following operations -

- Left-reduce - For this transition, I have combined the top two stack elements. First I retrieved the top (right-child) and the second topmost element (left-child) of the stack. Then I popped the second topmost element as part of reduce step and added a left arc from right-child to left-child.
- Right-reduce - Similarly, I retrieved the topmost and second topmost elements of the stack. But this time, I popped out the stack top for the reduce step and added a right arc that goes from left-child to right-child using the configuration.add_arc() method.
- Shift - This transition basically pushes the next input token w_i onto the stack. For this, I used the shift() method provided in $Configuration$ class.

1.2 Feature Extraction

After generating the configuration, we want to generate features for our neural model as described by Chen et al. The features of our model are a concatenation of three sets - S^w, S^t, S^l where S^w is the set of words, S^t is the set of POS tags, S^l is the set of labels. We generate these features for a given configuration using the vocabulary inside get_configuration_features() method of data.py.

- S^w - As described in the paper, S^w consists of 18 elements as follows -
  - Top 3 words on the stack: I retrieved them using the get_stack(k) method which returns the token index of k^{th} word on the stack, and added the word_id of each of the word to the list "features".
  - Top 3 words on the buffer: Similarly, I retrieved the top 3 buffer elements using get_buffer() method and added to "features".
  - The first and second leftmost / rightmost children of the top two words on the stack: Added the four elements to "features" using the get_left_child() and get_right_child () methods for the top two words on the stack.
  - The leftmost of leftmost / rightmost of rightmost children of the top two words on the stack: For the top two words on the stack I found the leftmost of leftmost using config.get_left_child(config.get_left_child(....)) and rightmost of rightmost using nested calls to config.get_right_child() in a similar manner.
- S^t - I used the vocabulary.get_pos_id() method and added the POS ID to S^t while adding the corresponding word to S^w. Therefore, S^w contains 18 POS tag IDs corresponding to the 18 words in S^w.

- S^l - Excluding the top 3 words on stack and buffer, I added the labels for the last 12 words in S^w and for each of them, I found their arc labels using vocabulary.get_label_id() method.

Then a list "features" containing all the above 48 features as described above was returned from get_configuration_features() by concatenating S^w, S^t and S^l.

1.3 Neural Network Architecture and Cubic Activation Function

1.3.1 __init__() method in DependencyParser

In this method, I have defined the following class members that will be used in call() method -

- **self.embeddings**: these refer to the word embeddings which can be either pre-trained glove embeddings or randomly set between -0.01 to 0.01 as described in the paper. I have used truncated_normal for calculating these. There are assigned by train.py in the case we are using pre-trained embeddings otherwise they contain the values assigned by truncated_normal. The shape of the embeddings is $(vocab\_size, embedding\_dim)$.
- **self.weights_input**: this refers to W^1, W^2 and W^3 from the paper but we use one weight matrix instead of three different matrices which essentially is the concatenation of all three. For initializing the input weight matrix, I have used Xavier initialization to set the standard deviation as follows -

$$\pm \frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}}$$

  where n_i is the number of incoming network connections, or "fan-in," to the layer and n_{i+1} is the number of outgoing network connections or "fan-out" to the layer. I used truncated_normal for assigning the values from a truncated normal distribution. The dimensions for this tensor are $(num\_tokens * embedding\_dim, hidden\_dim)$.
- **self.weights_output**: This is the weight matrix/tensor for output weights which gets multiplied by the output of hidden layer to produce the logits. They are initialized similar to input weights (using Xavier initialization) and their dimension is num_transitions x hidden_dim.
- **self.bias**: This is a zero tensor of size hidden_dim x 1.

1.3.2 call() method in DependencyParse

In this method, I have defined the forward propagation step of our neural model. We first get the embeddings for the given input using *tf.nn.embedding_lookup* and reshape it into a tensor of dimension (batch_size, num_tokens * embedding_dim). Then, I multiplied the embeddings of input with the weights for input layer and applied the activation function on it. After this, I multiplied the hidden layer output with the weights of the output layer to get the logits.

1.3.3 compute_loss() method in DependecyParser

The method used to computer the loss according to the paper is defined as follows -

$$L(\theta) = -\sum_i \log p_{t_i} + \frac{\lambda}{2}\|\theta\|^2$$

As described by Chen et al., we compute the cross-entropy loss over the softmax probabilities of feasible transitions. Therefore, we first need to calculate the softmax probabilities over feasible transitions as follows -

- Created a mask for masking out infeasible transitions from logits.
- I have used the exp-normalization trick to implement numerically stable Softmax. For this, I have subtracted the maximum logit score from all other logits in that row.
- Multiplied the exponentiated logits with the mask before taking the softmax.
- Calculated the softmax using the following formula -

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

- Calculated cross-entropy by multiplying the log of softmax probabilities with the masked values of labels. I added a small value 10^{-9} before taking the log so as to avoid NaN values in case the input is 0 for the log function.
- Calculated regularization term as defined in the paper by Chen et al. I used tf.nn.l2_loss() function to calculate l2 norm of input weights, bias, output weights and embeddings. I multiplied the sum of these with the regularization lambda to get the regularization term.
- Returned the final loss as cross entropy loss + regularization term.

2 Experiments

2.1 Base Configuration

Configuration
Used pre-trained embeddings (glove.6B.50d.txt)
Activation layer: Cubic
Trainable embeddings
experiment name: basic
Num epochs: 5

Average training loss	0.11
UAS	86.30
UASnoPunc	88.02
LAS	83.73
LASnoPunc	85.12
UEM	31.29
UEMnoPunc	33.64
ROOT	87.11

2.2 tanh Activation

Used pre-trained embeddings (glove.6B.50d.txt)
Activation layer: tanh
Trainable embeddings
experiment name: tanh
Num epochs: 5

Average training loss	0.13
UAS	87.21
UASnoPunc	88.84
LAS	84.67
LASnoPunc	85.97
UEM	32.53
UEMnoPunc	34.94
ROOT	87.76

2.3 Sigmoid Activation

Used pre-trained embeddings (glove.6B.50d.txt)
Activation layer: tanh
Trainable embeddings
experiment name: sigmoid
Num epochs: 5

Average training loss	0.16
UAS	85.99
UASnoPunc	87.74
LAS	83.51
LASnoPunc	84.94
UEM	29.47
UEMnoPunc	31.82
ROOT	87.47

2.4 Without Pretrained GLoVE Embeddings

No pre-trained embeddings
Activation layer: cubic
Trainable embeddings
experiment name: wo_glove
Num epochs: 5

Average training loss	0.08
UAS	84.00
UASnoPunc	86.00
LAS	81.59
LASnoPunc	83.27
UEM	26.64
UEMnoPunc	28.76
ROOT	79.17

2.5 Without trainable embeddings

Used pre-trained embeddings
Activation layer: cubic
Embeddings **not trainable**
experiment name: wo_emb_tune
Num epochs: 5

Average training loss	0.14
UAS	84.34
UASnoPunc	86.14
LAS	81.72
LASnoPunc	83.18
UEM	28.23
UEMnoPunc	30.23
ROOT	83.52

From the above experiments, I observed that for 5 epochs, the tanh activation unit performed slightly better than cubic function. And cubic activation unit performed better than sigmoid activation. However, these are the results for 5 epochs. We would have to train for a higher number of epochs to conclude the performance with every activation function.

Without using the pre-trained embeddings, my model did slightly worse than the base configuration described in 2.1. However, surprisingly the model did very slightly better when we froze the embeddings as compared to when we did not using the embeddings at all. This highlights that having some pre-trained embeddings for the task is always a good idea. But, freezing them might not be helpful as evident in our experiments.

2.6 Best Configuration

Activation unit: Cubic
Used pre-trained, trainable embeddings
Num epochs: 10
experiment name: best_config

Average training loss	0.07
UAS	87.25
UASnoPunc	88.91
LAS	84.14
LASnoPunc	85.39
UEM	35.35
UEMnoPunc	37.58
ROOT	88.35