Natural Language Processing:
Assignment 3 – Report

Name: Payal Mehta
SBU id: 112715351
Date: 15th Nov 2019

Section1: Model Implementation

Arc-standard:

In the arc-standard configuration, there is a stack, a buffer and a set of dependency arcs given to us as part of the boilerplate code. Below are the possible transitions that can be made, and these have been implemented in the following way.

Right-Reduce:
A configuration where an arc is added from the second topmost element of the stack to the topmost element of the stack. We then pop the topmost element from the stack.
This is implemented by simply calling on the configuration.add_arc () method and passing the elements of the stack. The top most element of the stack is removed by calling the configuration.remove_top_stack () method.

Left -Reduce:
A configuration where an arc is added from the topmost element of the stack to the second topmost element of the stack. We then pop the second topmost element from the stack.
This is implemented by simply calling on the configuration.add_arc() method and passing the elements of the stack. The top most element of the stack is removed by calling the configuration.remove_second_top_stack() method.

Shift:
An element is simply moved from the buffer to the stack. I implemented this by calling configuration.shift() (which is defined in the boilerplate code)

Feature extraction:

Following steps have been followed to extract features: -

1. First, I have taken the top 3 elements of the stack and top 3 elements of the buffer, so current count is 6.
2. Then, for the topmost and second topmost element, I have selected the left and right children of both, and then their left and right so current sum is 6 + 8 = 14
3. Then I have selected the leftmost and rightmost children of these so current sum is 14 + 4 = 18.

4. Next, for these 18 elements, I have retrieved the words and parts of speech (POS) tags from the vocabulary by querying the configuration.get_word() and configuration.get_pos_id(). so total sum is now 18 + 18 = 36.
5. For the elements above, barring the 6 stack and buffer elements (3 each), I have then got the labels by querying vocabulary.get_label_id() method.
6. Combining these 18, 18 and 12 elements, we get our feature list of 48 features.

Neural network architecture including activation function:

- The default activation provided in the method call is cubic, which I have implemented by simply returning the cube of the input vector. Using g(x) = x^3, we can model the product terms for any three different elements at the input layer directly.
- This model defines a transition-based dependency parser which uses a neural network classifier taking as input; the representation of words, their part of speech tags, and the labels which connects words in a dependency parse.
- The size of the vocabulary, number of tokens, dimension of embeddings, dimensions of hidden layer, number of transitions, trainable embeddings (whether True or False) and the regularization loss fraction lambda is given to us.
- Making use of above, I have then initialized the embedding matrix, weights and biases.

For the random model,

- The embeddings are formed by creating a random distribution of the size of vocabulary and embedding dimensions. A small standard deviation is added equal to the reciprocal of the square root of the embedding dimensions for creating a uniform distribution.
- To obtain the weights W1 and W2, I have created a TensorFlow variable of shape equal to product of the dimensions of the embeddings and number of tokens and the hidden layer dimensions.
- The TensorFlow command to create variables on which operations can be performed is tf.variable(). I have used a standard deviation of 0.005 for both the weights.
- For bias, I have again created a TensorFlow variable whose shape equals the hidden layer dimensions. Since this vector will be used with other two-dimensional ones in addition operations, I have used 1 as the shape for the second dimension.
- To now lookup the embeddings, I have passed the inputs to the tf.nn.embedding_lookup method to look in the embeddings matrix.
- The vector matrix is calculated as the matrix multiplication of embeddings and first weight. And then I have added bias to it. The output of the hidden layer is the cubic activation of the vector matrix.

This output is further multiplied with the weights for second layer to obtain logits.

Loss function:

The logits returned as an output to the neural architecture is then used to compute the loss. I have calculated the SoftMax by creating two masks where one mask contains only 1s and one contains 0 and 1, ignoring the -1s. The softmax is simply the division of the numerator and denominator where the numerator is the labels > 0 multiplied with the logits, and the denominator is the labels including 0 and 1 (second mask). The negative of this division is the loss.

Section 2: Experiments

Below are the experimental results: -

	Basic(default)	**Tanh**	**Sigmoid**	**wo_glove**	**wo_emb_tune**
Activation	Cubic	Tanh	Sigmoid	Cubic	Cubic
Pretrained Embeddings	Yes	Yes	Yes	No	Yes
Loss	0.11	0.16	0.18	0.11	0.15
UAS	87.98	86.75	85.63	87.23	84.35
LAS	85.55	84.26	83.13	84.77	81.68
UASnoPunc	89.64	88.47	87.54	88.65	86.12
LASnoPunc	86.89	85.64	84.72	86.09	83.12
UEM	34.58	32.11	29.82	34.05	27.0
UEMnoPunc	37.47	34.82	32.47	36.52	29.05
Root	89.52	87.11	86.05	88.0	84.76

Table 1

Observations:

1. The performance of the model depends on the **Activation Function** being applied:
    a. Cubic
    The basic model uses the cubic activation. As we can see from (table1) the basic model performs better in both unlabeled (UAS) and labelled attachment scores (LAS). The paper by Manning also mentions their experimental results verifying the success of the cube activation function empirically and the same is evident from above results.

    b. Tanh
    - The tanh function performs better than the sigmoid function, however not as good as the cubic one, theoretically. This can be confirmed from looking at the results of table1.

    Tanh is nothing but a scaled sigmoid function. However, the gradient in case of tanh is stronger, meaning the derivatives are steeper. This also means that the cost function can be minimized faster if you use tanh as an activation function instead of sigmoid.

- The tanh function ranges from -1 to 1, that means it is centered around 0; while the sigmoid function ranges from 0 to 1, meaning centered around 0.5. One observation is that the output increases but at a very slow rate. This can be equated to the mathematical reason that as the input increases, the value of exponent terms in sigmoid becomes infinitesimally small and hence the output becomes equal to one.
- Models with sigmoid activation function are usually slow learners and these models generate predictions which have lower accuracy.
- **We can see that the loss in tanh function is 0.02 lesser than sigmoid but 0.05 more than cubic. Also, the accuracies for Labelled and Unlabeled differ by 1.2% when compared with cubic and sigmoid. (Refer table1)**

Our experiments clearly show that tanh performs better than sigmoid in terms of accuracy.

c. Sigmoid
   - The sigmoid function is the worst of all the three activation functions and our experiments clearly show that (Refer table1).
   - The derivates in the case of a sigmoid activation function are not as steep as in the tanh function. The tanh function takes into consideration the range between -1 to 1, whereas sigmoid just sticks to the positive end, 0 to 1.
   - Models with sigmoid activation function are usually slow learners and these models generate predictions with lower accuracy.
   - **The loss in sigmoid is the highest at 0.18. The accuracies for labelled and unlabeled differ by 2.3% when compared with cubic and by 1.1% when compared with tanh.**

Tanh activation functions are usually preferred over sigmoid activation functions.

2. The performance of your model also significantly depends on whether you are **Using a pretrained model v/s not using one**
   - The wo_glove model is the one without the pretrained embeddings, while all the other ones are with the pretrained embeddings.
   - If we just pick two models, the basic one and the wo_glove one, we can see that the one with the pretrained embeddings performs better than the one without it.
   - One possible reason is that the embeddings are already trained at a point called as the initialization point which might have been decided based on several parameters, whereas the model in which we do not have a pretrained load, are random.
   - We might have to do a lot of analysis and work on a lot of parameters to decide which are the ones we want to use in our model and also how to optimize them.
   - The best part of using a pretrained model is that the hard work of optimizing the parameters has already been done and hence we can directly get to the part of tuning the hyperparameters.

- **Hence, the accuracy in case of a pretrained model is usually better. (also evident from columns 1 and 4 – there is a difference of 0.5 – 1% in accuracies between a pretrained model and a random one.)**

3. The performance of our model depends on whether we are **using tuned embeddings v/s not using them.**
    - From the above table (table1), we can see that the wo_emb_tune, which is infact our model without the tuning, performs so much worse than the models where we have used tunable embeddings.
    - Tuning is important as it directly controls the behavior of the training algorithm and has a significant impact on the performance of the model being trained.
    - The tunable embeddings are created with a good choice of hyperparameters and this plays a huge role in the success of our neural model architecture, as it directly affects the learned model.
    - One hyperparameter that makes a huge impact on the performance of our model is the learning rate; if we choose a rate too low, the model will miss important patterns in the data as it will take a long time to reach the minima.
    - **The accuracies in this case differ by 3.6% when compared with the tuned embeddings, the former being less accurate.**

It can be seen from our experiments, that the model where we haven't used tunable embeddings (wo_emb_tune) performs much worse than the ones where we have used such embeddings where the hyperparameters are stored and kept for us.