NLP ASSIGNMENT 3

Sakshi Gupta -112552239

Section 1: Model Implementation:

1.1 Arc-standard system

As is mentioned in the given paper, the Arc-Standard system consists of three basic actions:

1. Shift: I implemented shift using configuration.shift() which is pre-written in the code. The pre-written code basically fetches the first element from the buffer and pops it and appends it to the stack while checking the mandatory condition for 'shift' operation to be valid.
2. Left-arc:I fetched the top two stack elements and used configuration.add_arc to make a left arc from the top element to the second top element while marking the arc with the appropriate label and then I popped the second element from the stack.
3. Right-arc: I fetched the top two stack elements and used configuration.add_arc to make a right arc from the second top element to the top element while marking the arc with the appropriate label and then I popped the top element from the stack.

1.2 Feature Extraction

As mentioned in the paper:
(1) The top 3 words on the stack and buffer: s1, s2, s3, b1, b2, b3: So I fetched these 6 elements and appended them to a list li0.
(2) The first and second leftmost / rightmost children of the top two words on the stack: lc1(si), rc1(si), lc2(si), rc2(si), i = 1, 2: doing this I got 8 elements and appended it to a new list li.
(3) The leftmost of leftmost / rightmost of rightmost children of the top two words on the stack: lc1(lc1(si)), rc1(rc1(si)), i = 1, 2: doing this I got 4 elements and appended it to the previous list li.

1. All this is implemented using the pre-written functions get_left_child and get_right_child and at the end we get 18 elements in list li1.
2. Then I fetched 18 POS tags using the above 18 elements and get_pos() and get_pos_id() and stored it in a list. (St (nt = 18) as mentioned).
3. Then I fetched 18 words using the above 18 elements and get_word() and get_word_id() and stored it in a list. (Sw (nw = 18) as mentioned).
4. Then I fetched 12 arc labels using the second list li (to exclude the first 6 elements) and get_label() and get_label_id and stored it in a list. (Sl (nl = 12))

At the end I combined the 3 lists to get the features vector which has 48 elements in total.

1.3 Neural Network Architecture

1. Activation Function: I used tf.pow() to implement the cubic activation function which returns the cube of the input vector. Sigmoid and Tanh were implemented beforehand in the given code.
2. Neural Model: The DependencyParser class has some input parameters which I used to set the shape of the 4 tensors- embeddings, W1, W2 and b1(bias).

__init__():

- embeddings:I used vocab_size and embedding_dim for defining the shape of 'embeddings' where vocab_size is the number of words in the vocabulary and embedding_dim is the dimensions of the embedding. For random sampling, I used np.random.normal and provided standard deviation as 1/sqrt(embedding_num). I set 'trainable' here as the provided Boolean value of 'trainable_embeddings' for both the cases.

- W1: I used tf.truncated_normal() for getting the distribution and I defined the shape as (num_tokens*embedding_dim, hidden_dim) where num_tokens is number of words or tags and hidden_dim is the dimension of the neural network. I have used standard deviation as 0.005 and mean as 0.0 here.

- W2: Random sampling is done in the same way as W1 here. The shape here is (hidden_dim,num_transitions) where num_transitions is the total number of transitions we have for the configuration.

- b1: It is a (1, hidden_dim) tensor initialized to all zeros. It has this dimension as it needs to be added to the above three tensors.

__call__():

- I performed embedding lookup on the 'embeddings' tensor computed earlier using 'inputs' as the ids for the lookup.
- Then I multiplied this resultant matrix with W1 and added b1 to it and then passed it through the activation function (cubic, tanh or sigmoid based on the parameters passed).
- The output of this layer is multiplied with W2 and later softmax function will be applied on this multiplication.

1.4 Loss Function

- I filtered the labels where labels>-1 i.e taking only 0 and 1 values and created a mask out of it. I used reduce_max to get 1's from the columns and then I multiplied it with logits to get the numerator for the softmax function.
- In the denominator, we want to take these values of the labels i.e 0 and 1.

- Then I computed the softmax by dividing the above two and computed the loss parameter as per the formula given in the paper. I also added some noise to my loss value to enhance the accuracy.
- Then I computed the regularization parameter (multiplied with lambda) based on the value of 'trainable_embeddings' and add it to the loss which is returned by compute_loss().

Section 2: Experimental Analysis:

- ### Observations:

1. Scores Table:

	Basic	Tanh	Sigmoid	wo_glove	wo_emd_tune
Pretrained Emb	glove.6B.50d	glove.6B.50d	glove.6B.50d	-	glove.6B.50d
UAS	87.94	86.52	85.22	86.35	84.78
UASnoPunc	89.58	88.35	87.16	88.01	86.53
LAS	85.46	84.08	82.75	83.92	82.11
LASnoPunc	86.77	85.58	84.35	85.27	83.53
UEM	35.94	32.41	30.11	32.35	29.05
UEMnoPunc	38.88	35.17	32.23	35.0	31.17
ROOT	89.82	86.82	84.52	86.17	85.23

In the above table I have only considered the Un-labelled and Labelled Attachment Scores, UEM and ROOT scores to measure the accuracy of the final models with different configurations such as with different activation functions (Cubic, Tanh and Sigmoid) and using the pre-trained model also.
The Unlabeled and Labelled Attachment Scores are the most widely used scores and give a measure of the words that have given the correct label. These scores have been considered both with and without punctuations. Unlabeled Exact Match (UEM) scores are used for measuring sentence parsing accuracy. This score has also been considered with both with and without punctuations.

2. Average Training Loss Table:

	Basic	Tanh	Sigmoid	wo_glove	wo_emb_tune
Final loss	0.11	0.15	0.18	0.11	0.15

In the above table, I have noted down the average training loss values after each epoch for all the 5 configurations.

- ## Analysis:

1. Different Activation Functions- Cubic VS Tanh VS Sigmoid:

1. There is a clear difference of 1.4% in the accuracies of the basic configuration which uses cubic and the configuration which uses tanh. There is also a difference of 1.2% between configurations of tanh and sigmoid.
2. We can see from the above point that Cubic outperforms Tanh and Sigmoid. Tanh comes in the second place and Sigmoid turns out to be the worst performer amongst the three.
3. The reason could be as mentioned in the paper that cubic activation function helps the interaction between three parameters xi, xj and xk easy, which is basically what is done in each hidden unit before adding the bias.
4. Tanh is said to be the logistic version of sigmoid. Both seem to suffer from saturating and killing the gradients (vanishing gradient problem) as the layer's activation saturates at either tail of 0 or 1 in case of sigmoid and -1,0 and 1 in case of Tanh. Sigmoid and Tanh are also computationally expensive.
5. Tanh performs better than sigmoid as Tanh outputs are zero-centered whereas Sigmoid outputs are not. Having this makes the learning easier for the next layer. Also, The derivatives of tanh are higher than sigmoid and so it is faster to converge and minimize the calculated loss.
6. From the average training loss table also we can see that the basic configuration performs better and has a lesser loss than Tanh and Sigmoid. (Sigmoid performing worse than Tanh).
7. All the configurations apart from the last one have been executed using tunable embeddings whereas the last configuration wo_emd_tune does not have tunable embeddings and hence performs worse than all the other configurations.

2. Pretrained VS Without Pretrained model:

1. We ran the Basic, Tanh and Sigmoid configurations using a pretrained model. W0_glove runs without a pretrained model and uses random initializations done in the code.

2. As we can see from the above table, the scores of w0_glove are lesser than those of the basic model with a difference of around 1.5% in the accuracy of both even though both of them use cubic function as the activation function.
3. One of the possible reasons can be that when we are using pre-trained models we start our model training using some learned weights that are produced by the pre-trained model whereas without any pre-trained model we start from random initializations. This helps in capturing the underlying relationships between the parameters of our model.
4. This difference in initialization can have an impact on quick convergence and overall accuracy of the model.
5. From the average training loss table, we see that the both the models have the same final training loss.
6. Wo_emb_tune obviously has a worse average training loss than the basic configuration which is expected because of the absence of tunable embeddings.

- References:

1. http://cs231n.github.io/neural-networks-1/
2. https://heartbeat.fritz.ai/pre-trained-machine-learning-models-vs-models-trained-from-scratch-63e079ed648f
3. https://pdfs.semanticscholar.org/6008/3a8515dca99079df7ceea3ede1abbb4f1ba4.pdf