1. Model Implementation

A. Arc Standard method:

The arc standard method was implemented according to Nivre et al., where:

I. *Left Arc: When the transition string starts with 'L(', Left transition is appleid: the second to the top of stack is made dependent to the top of stack, and the second to the top of stack is removed.*

II. *Right Arc: When the transition string with 'R(', Right transition is applied: the top of the stack is made dependent to the second to the top of the stack, and the top of stack is removed.*

III. *Shift: If the transition string doesn't start with either it is shift action, hence the top of the buffer is popped out and added to the top of the stack.*

B. Feature Extraction:

The features were extracted based on the implementation of Manning et al., I used this repository as a reference to understand the process better and structure it.

1. Firstly, the direct tokens were extracted by putting the top 3 of the stack and the top 3 of the buffer into the feature list. (6)

2. The children tokens were extracted by extracting the following for the top of stack and the second top of the stack. (2)
(i) Left child and the right child. (2)
(ii) Second Left child and the second right child if the exist. (2)
(ii) Left child of left child and the right child of right child. (2)

3. Hence there are 6+12 items for which each of the following features were extracted.

*4. For each of the above the pos and the words were stored as features. Hence (6+12)*2 = 36.*

5. The labels were extracted only for the children token as features(12).

6. Putting those together we have 36+12=48 features.

7. We could actually have a million features, but this collection by itself although effective, is sparse. It will only make it even more sparse.

C. <u>The neural network architecture including activation function:</u>

The Neural Network was implemented the same way as implemented by Manning et al.,.

1. In init function (constructor), the embeddings were initialised using a random normal vector of dimensions [vocabulary size x embedding dimension]. I used a truncated normal distribution to not have the embeddings initialised far apart. The stddev of the normal distribution was set to 1/square root(vocabulary size). This provided better results as opposed to default value of 1.

2. Next the weights and the biases were initialised with the following setup:

a. weights(input - hidden): [feature len*embedding_dim x hidden_dim].
Init: Truncated normal distribution with stddev: 1/sqrt(feature len*embedding_dim).

b. weights(hidden - output): [hidden dim x output transitions].
Init: Truncated normal distribution with stddev: 1/sqrt(hidden dim).

c. Bias(hidden): [hidden dim], Init: Truncated normal distribution with stddev: 1/sqrt(hidden dim).

3. The call function is used for feed forward. The embedding lookup is used to extract the relevant inputs' embedding. It sent to the hidden layer by matrix multiplying with the hidden weights matrix and added with the bias.

4. The hidden layer is activated using the activation function that is called. This could be cubic, tanh, sigmoid.

5. The cubic activation is done by using raising the tensor to the power of three.

6. Finally the output layer logits are calculated by multiplying the hidden layer with the output layer weights.

7. If it is training period, the loss is computed by calling the loss_compute function.

D. <u>Loss Function:</u>

The loss function coded is the cross entropy on softmax.

1. The labels could be 1, -1 or 0. 1 represents the right transition, -1 represents invalid transition, 0 represents wrong transition.
2. To my understanding, learning -1 is to make the model aware of the transitions that need not be considered towards transitions at all. Hence we want the model to learn that there is 0 probability to that transition.

3. Firstly we vectorise the indices that have -1 or not. This is stored as a mask vector.
4. The labels are multiplied with the mask and is stored as actual mask.
5. The softmax values of the logits is calculated to compute cross entropy.
6. The numerator of thesoftmax is computed by exponentiating the product of mask and the logits.
7. The denominator is computed by summing up the numerator values.

8. Cross entropy is calculated by multiplying the masked labels with the log of softmax of logits. Reference
9. A sum across all dimensions is taken followed by a mean across all instances.

10. The l2 loss is taken for the weight vectors and is multiplied with the lambda val.

11. This is added to the loss computed and passed to the call.

<u>2. Experiments</u>

In all cases the no punc accuracy was a bit higher so was left out from this table.

A. <u>Activations:</u>

The activations used were cubic, tanh and sigmoid. The cubic activation as seen in the basic performed on par with the tanh and better than sigmoid on all measures. **Manning in his lecture lays emphasis on UAS metric, accuracy of getting the labels right.** Cubic performed the best across most metrics.

B. <u>Pretrained Embeddings:</u>

Without the embeddings, the model doesn't perform any good as the ones that use the pretrained embeddings on all measures.

C. <u>Tunability of Embeddings:</u>

Tuning the embeddings doesn't improve the performance of the model by any bit as compared to the use of embeddings with the cubic activation.

<u>Results:</u>

Experiment	**UAS**	**LAS**	**UEM**	**ROOT**
Basic (Cubic)	87.182	84.744	33.352	87.176
Sigmoid	85.744	83.194	28.941	86.0
Tanh	87.020	84.652	33.176	87.294
Without Glove	85.322	82.939	30.058	83.176
Word Embedding Tune	87.182	84.744	33.352	87.176

Table 1: Results

<u>3. Questions I had</u>

I. When I added another hidden layer, the loss started becoming NaNs after some time. Although I made sure to activate it with ReLU to avoid the log being the culprit, it still kept giving Nans.

II. I introduced a dropout of 0.1 to the hidden layer hoping it would generalise better, but it did not perform better in the validation set.

III. Well initialising the weights with truncated normal with std dev of 1/sqrt(dimensions) got my model to run as opposed to never ending NaNs it gave before I did that. I understand that the variances are lowered, but would like to understand why there is so much noticeable difference because of this. I have seen these implementations on few papers and that's why I gave it a shot. Reference