NLP Assignment 3: Transition Parsing with Neural Networks

Utkarsh Garg

November 16, 2019

1 Transition Parsing System

We use the Arc Standard system to implement the Dependency Parser, which involves 3 components: A stack s , a Buffer b and a set of dependency arcs A.

The initial configuration for a sentence $w_1, ..., w_n$ is s = [ROOT]; b = $[w_1, ..., w_n]$,A $=\phi$. A configuration c is terminal, if the buffer is empty and the stack contains the single node ROOT, and the parse tree is given by A_c.

2 Implementation

2.1 Dependency Parser:Arc-Standard Algorithm:

- We start by Implementing the Arc-Standard algorithm in parsing_system.py file. This involves creating 3 transitions viz. Left-Arc, Right-Arc and Shift.

- We're given transitions as inputs in the format $L(label)$ or $R(label)$ in case of Arc operations, otherwise simply S for Shift operation .

- For Arc transitions, we identify the type of arc operation, plus the label from the transition input. We also extract the 1st two words from the existing stack in order to add an arc with label ,and eventually pop one item from the stack, making one word-token a dependent on another.

- For **Shift** operations, we simply push the next input token w_i into the stack .

- For **Left Arc** transition, we add an arc $s_0 \rightarrow s_1$ with label and remove 2^{nd} top element $s1$ from the stack. Whereas, for **Right Arc** transitions, we add an arc $s_1 \rightarrow s_0$ with label and remove stack top $s0$ from the stack.

2.2 Features extraction:

We follow the section 3.1 from the paper **The choice of** $S_w; S_t; S_l$ in order to extract our feature-token ids.

- We are given a Vocabulary and a Configuration file with defined modules for performing operations and retrieving words/ids for Stack and Buffer operations.

- We select 18 words from our Stack totally, in the following order: (1) The top 3 words on the stack and buffer: $s_1; s_2; s_3; b_1; b_2; b_3$; (2) The first and second leftmost / rightmost children of the top two words on the stack: $lc_1(s_i); rc_1(s_i); lc_2(s_i); rc_2(s_i), i = 1; 2$. (3) The leftmost of leftmost / rightmost of rightmost children of the top two words on the stack: $lc_1(lc_1(s_i)); rc_1(rc_1(s_i)), i = 1; 2$.

- We use the corresponding POS tags for S_t ($n_t = 18$), and the corresponding arc labels of words excluding those 6 words on the stack/buffer for $S_l(n_l = 12)$.

2.3 Neural Network Architecture:

We start by defining our Dependency Parser, which makes use of a classifier powered by a neural network. The neural network accepts distributed representation inputs: dense, continuous representations of words, their part of speech tags, and the labels which connect words in a partial dependency parse.

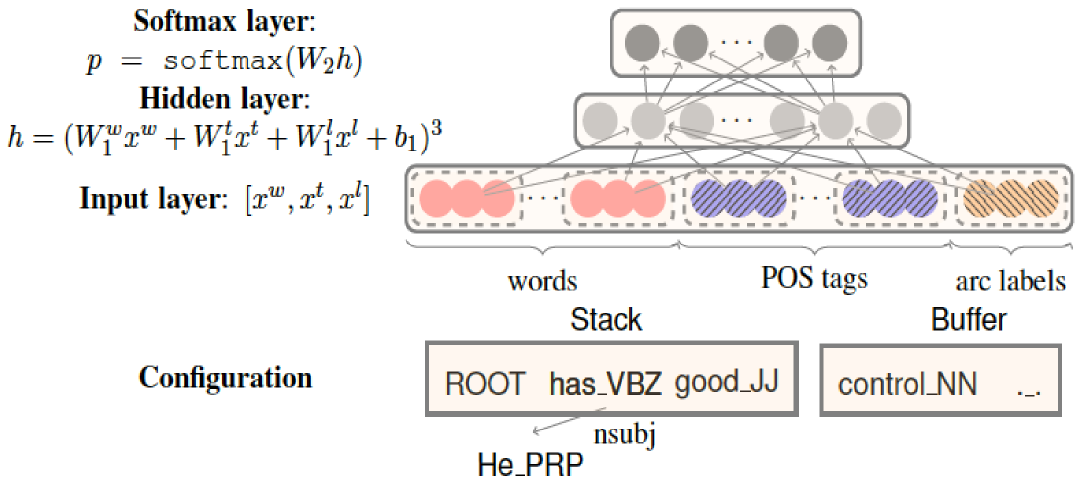


Figure 2: Our neural network architecture.

- In our init call to our Model, we set our activation functions, along with other Trainable variables. We also initialise our embeddings and weights and biases for our subsequent hidden and output layers.

- For our embeddings, we declare a tensor of shape [vocab_size,embedding_dim] with truncated_normal values in the range [-.01,.01].

- For weights and biases for the hidden layer, we do the similar work , this time with a standard deviation defined as $sqrt(2/fan\_in)$, where fan_in represents no. of input features.

- Similarly for the output (classification) layer, we define another weight matrix to be multiplied by output value from the hidden layer.

- In the method call, we then define our Forward Pass for the dependency Parser. We extract the tokens from the embedding matrix initialised earlier, followed by matrix multiplication of hidden layer weights from $weights1$ adding the bias.

- This is further converted to non-linear form by using Cubic activation function defined earlier.

- The output from the hidden layer is then subjected to classification layer , using the weights2 matrix initialised earlier.

- This gives us the logits which are then passed to the compute_loss function for further processing.

2.4 Loss Function:

- For the loss function, we make use of the logits computed from Forward pass implemention, with an aim to obtain the probability of all transitions, incl. feasible and infeasible.

- We define these transitions as follows:

- 1: 'correct' transition according to the oracle. There is only one 'correct' move per configuration state
- 0: feasible, but 'incorrect' transition according to the oracle. A move that can be made according to the current state, but would be 'incorrect'
- -1: infeasible transition. This means that the configuration is not in a state where this transition could be performed.

Since the logits returned from our Model class, represent 3 types of transitions, we need to filter out [-1] transitions in order to compute the correct softmax probabilities.

- For this purpose we create a mask to filter out incorrect transitions/labels [-1] from our computed logits. We have used tf.cast on condition labels¿=0 to get the corresponding float values as 0 and 1 for our mask, identifying infeasible and feasible labels respectively. The result is multiplied with logits to obtain 'feasible_logits'.

- we compute $exp^{feasible\_logits}$ to compute the numerator for the softmax.

- Since,taking exponent converts labels 0[infeasible] from e^0 value to 1. We therefore need to filter those out . So we reuse the created 'label_mask' to filter only correct $exp^{feasible\_logits}$. We thus obtain 'masked_exp_feasible_logits'

- Similarly, for the denominator, we compute 'sum_masked_exp_feasible_logits' by taking summation of 'masked_exp_feasible_logits' across y-axis to get a $[batch_size, 1]$ shaped tensor. We then perform a matrix multiplication with 1's tensor of shape $[1, num\_transitions]$ to obtain the 'sum_masked_exp_feasible_logits' of shape $(batch\_size, num\_transitions)$

- We then use the below formula to obtain our final softmax logits(probabilities):

$$softmax(x) = \frac{\exp x_i}{\sum_i \exp x_i}$$

- The final objective is to minimize the cross-entropy loss, plus a l2-regularization term,given by:

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

Here:
$p_{t_i} \rightarrow$ softmax probability for input i.
$\lambda \rightarrow$ regularization_lambda
$\theta \rightarrow$ set of parameters $\left\{W_1^w, W_1^t, W_1^l, b_1, W_2, E^w, E^t, E^l\right\}$

- Following above loss function, we compute our log_prob by taking log on our softmax_logits.

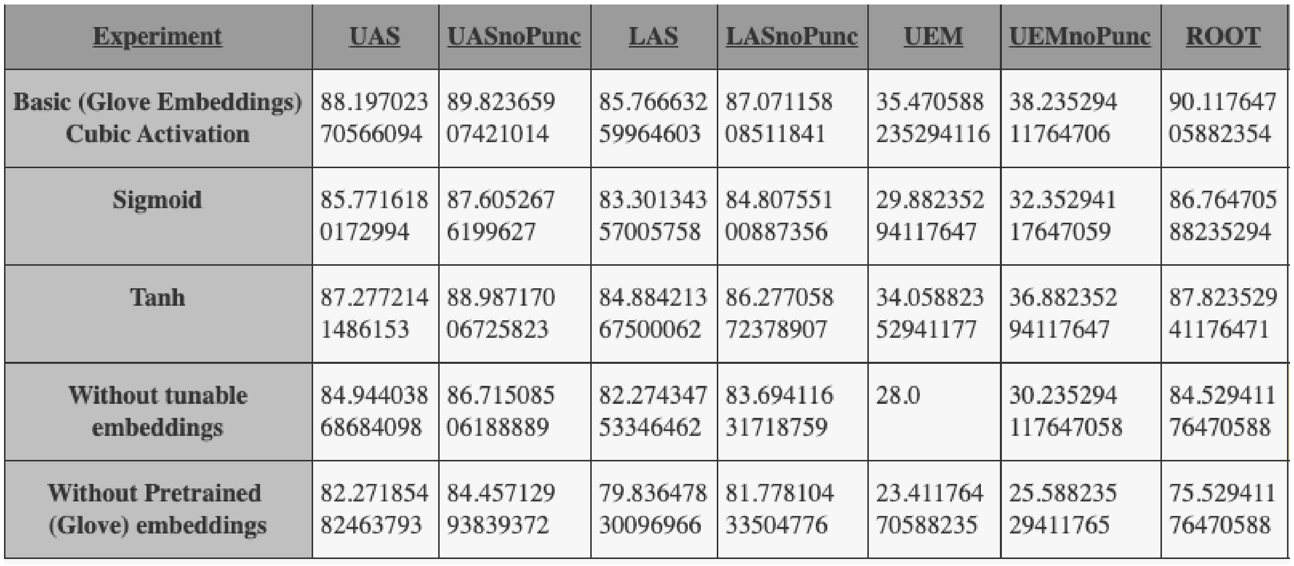
- We create a mask to filter just the correct transitions/labels [+1] from our computed log(softmax-logits), to compute 'correct_logits'.

- We then sum this over entire batch inputs, by taking summation of correct_logits, to get 'batch_loss'.

- Our total loss is computed by taking mean of this batch_loss along axis=1.

- Regularization term is computed by taking sum of L2-loss over set of parameters ($weights1, weights2, bias, embeddings/tokens$), multiplied by regularization_lambda value provided .

3 Analysis



Experiment	UAS	UASnoPunc	LAS	LASnoPunc	UEM	UEMnoPunc	ROOT
Basic (Glove Embeddings) Cubic Activation	88.19702370566094	89.82365907421014	85.76663259964603	87.07115808511841	35.470588235294116	38.23529411764706	90.11764705882354
Sigmoid	85.7716180172994	87.6052676199627	83.30134357005758	84.80755100887356	29.88235294117647	32.35294117647059	86.76470588235294
Tanh	87.2772141486153	88.98717006725823	84.88421367500062	86.27705872378907	34.05882352941177	36.88235294117647	87.82352941176471
Without tunable embeddings	84.94403868684098	86.71508506188889	82.27434753346462	83.69411631718759	28.0	30.235294117647058	84.52941176470588
Without Pretrained (Glove) embeddings	82.27185482463793	84.45712993839372	79.83647830096966	81.77810433504776	23.41176470588235	25.58823529411765	75.52941176470588

Fig:1 : Performance Evaluation for different Experiments.

We can see that our best model is the Basic one with Pre-trained GLOVE Embeddings , activated by Cubic function. We obtain the ROOT accuracy of 90% , the highest amongst all our experiments. Tanh model is the next closest and has comparable accuracy to Basic.

Going with the best model ,we have thus used the **Basic** Model to generate predictions for the test data (inside test_predictions.conll).