TRANSITION PARSING WITH NEURAL NETWORKS

112685069, Abdullah Mitkar

1. Model Implementation

a. Arc Standard algorithm.

- The Arc Standard algorithm is implemented from the paper 'Incrementality in Dependency Parsing (2004, Nirve)'
- The apply function takes in transition, configuration as input and return configuration based on the changes made by following the algorithm.
- The algorithm defines three types of transitions based on the input transition:

  Input: Transition, Configuration

  - Transition starts with L: LEFT-ARC (l)- It adds an arc from the top most element(s1) on the stack to the second element(s2) on the stack with a label 'L' and pops the top element(s1) from the stack.
  - Transition starts with R: RIGHT-ARC (r)- It adds an arc from the second element on the stack(s2) to the first element on the stack with a label 'R' and removes the second element(s2) from the stack.
  - Otherwise: SHIFT() - Configuration has a method shift that moves b1 from the buffer to the stack.

b. Feature Extraction.

- Features are extracted based on the paper "A Fast and Accurate Dependency Parser using Neural Networks (2014, Danqi and Manning)"
- A set of elements are chosen based on the position in the stack and buffer for each type of information (word, POS tag or arc label) or (S_w, S_t, S_l) which might be useful for making predictions.
- S_w: 18 elements. (3 + 3 + 4 + 4 + 2 + 2)
  I. 3 elements from the stack (3)
  II. 3 elements from the buffer (3)
  III. The first and second leftmost children of the top two elements of stack (4)
  IV. The first and second rightmost children of the top two elements of stack (4)
  V. The leftmost of the leftmost of the top two words on the stack (2)
  VI. The rightmost of the rightmost of the top two words on the stack (2)
- S_t: 18 elements.
  I. POS tags for the elements mentioned for Sw.
- S_l: 12 elements.
  I. Arc labels for elements mentioned in III, IV, V and VI.

c. Neural Network Architecture.

- The neural network consists of 1 hidden layer, 1 activation layer, and the softmax layer as suggested in the paper "A Fast and Accurate Dependency Parser using Neural Networks (2014, Danqi and Manning)".
- Initialisation:
    - $weight_1$ [num_tokens, hidden_dim]: Random initialization with std dev= 1/sqrt(hidden_dim)
    - $weight_2$ [hidden_dim, num_transitions]: Random initialization with std dev=1/sqrt(num_transitions)
    - $bias_1$ [hidden_dim]: Zero initialisation
- Input Layer
    - Should be: [x_w, x_t, x_l] (Suggested in paper); Actual: [batch_size, num_token]
    - Do embedding lookup. Output[batch_size, num_token, embedding_dim]
        - We find the embedding of the feature words that we receive as input.
    - What to do? Re-shape to: [batch_size, num_token * embedding_dim]
    - Pass through first hidden layer ($W_1X + B_1$), Output:[batch_size, hidden_dim]
    - Pass through activation function $f(x)^3$, Output:[batch_size, hidden_dim]
    - Pass though second hidden layer (W_2X) , Output:[batch_size, num_transition]
    - Pass to loss function, Output:[batch_size, 1]

d. Loss Function.

- Get Cross Entropy loss
    - Feasibility Mask <- Create a mask for filtering labels which are -1. [Unfeasible labels]
    - Find stable logits for stable softmax. [logits -max(logits)]
        - e^x for stable logits
        - Apply feasible mask to remove unfeasible exponentials [not to consider]
    - Cross Entropy loss on softmax for positive labels(1).
- Add L2-Regulariser for weights, biases, embeddings (when trainable)

2. Experiments

	With Pretrained Cubic	With Pretrained Tanh	With Pretrained Sigmoid
UAS	86.41224	86.893337	85.90373
UASnoPunc	88.09416	88.6254451	87.6957
LAS	83.79241	84.452975	83.48082
LASnoPunc	85.13254	85.8616402	84.96015
UEM	31.41176	32.2941176	29
UEMnoPunc	34.11765	35	31.29412
ROOT	87.52941	87.17647	86.11765

	Without Pretrained Embeddings	Tunability of Embeddings
UAS	82.94239	84.12643
UASnoPunc	85.05906	85.98598
LAS	80.42725	81.39193
LASnoPunc	82.23591	82.90284
UEM	24.88235	27.41176
UEMnoPunc	26.88235	29.17647
Root	77.76471	83.47059

The finding by the experiments are in sync with what the paper states.

Cubic activation outperforms tanh activation function and other activation functions. The cubic function models the relationship between three input elements and therefore is better.

Using pretrained embedding increases the accuracy as compared to training the embeddings from scratch. Allowing the embedding to learn from scratch will take more time compared to using pretrained state-of-the-art embeddings like glove. Hence, pretrained embedding perform better because they are trained on larger dataset for more time.

Allowing the embeddings to further learn the weights also increases the accuracy as opposed to freezing the embeddings and stopping them to learn weights. The embedding can also correct themselves for this specific task as opposed to their initial values in which they are trained for generic natural language task. Hence, training the embedding further improves accuracy as the embedding get the learn the embedding for the elements in this specific task.

The best configuration hence is Pretrained glove with Cubic activation function with trainable embeddings allowed.