1. Model Implementation

1.1. Arc-standard algorithm

Left-Reduce: Add an arc with stack's top element as head and stack's second top element as a dependent node, also remove the second top element of the stack.
Right-Reduce: Add an arc with stack's second top element as head and stack's top element as a dependent node, also remove the top element of the stack.
Shift: Use the ***configuration#shift*** function that is already defined in codes.

1.2. Feature extraction

Use the method described in the paper [Chen and Manning, 2014]:

The choice of S^w, S^t, S^l

Following (Zhang and Nivre, 2011), we pick a rich set of elements for our final parser. In detail, S^w contains $n_w = 18$ elements: (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 ($n_l = 12$). A good advantage of our parser is that we can add a rich set of elements cheaply, instead of hand-crafting many more indicator features.

- Add the top 3 words on the stack and buffer(***s1, s2, s3, b1, b2, b3***) to ***word_features***.
- Add the stack's top two words(i.e. ***s1 and s2***)'s first leftmost children(***lc1***), second leftmost children, rightmost children(***lc2***), and second rightmost children to ***chd_features***.
- Add the leftmost and rightmost children of ***lc1*** and ***lc2*** to ***chd_features***.
  - Also add all the ***chd_features*** to ***word_features***.
- Transform ***word_features*** to ***pos_features*** using ***configuration#get_pos*** and ***vocabulary.get_pos_id*** functions.
- Transform ***chd_features*** to ***label_features*** using ***configuration#get_label*** and ***vocabulary.get_label_id*** functions.
- Transform ***word_features*** using ***configuration.get_word*** and ***vocabulary.get_word_id*** functions.
- Then the output is the stack of ***word_features***, ***pos_features***, and ***label_features***.

1.3. Neural network architecture

- **DependencyParser#__init__**:
  - define ***self.embeddings***(which will then be override if 'pretrained-embedding-file' is true), using ***'truncated_normal'*** to initialize embedding matrix within (-0.01, 0.01) for Ew, Et, and El which are combined as a (***vocab_size***, ***embedding_dim***) matrix, and ***freeze=trainable_embeddings***.
  - define ***self._W1*** with size (***num_tokens*** * ***embedding_dim***, ***hidden_dim***)
  - define ***self._W2*** with size (***hidden_dim***, ***num_transitions***) and ***bias=False***.
- **DependencyParser#forward**:
  - Apply embeddings on the inputs, which will give an embedding of size (batch_size, num_tokens, embedding_dim), then reshape as (batch_size, num_tokens * embedding_dim).
  - For every batch, apply self._W1, self._activation, self._W2 sequentially, which gives the `logits` for the output.
- **CubicActivation**: Simply apply power of 3 to the input vector.

1.4. Loss function

- **DependencyParser#compute_loss**:

According to the paper Chen and Manning, 2014:

> The final training objective is to minimize the cross-entropy loss, plus a l_2-regularization term:
>
> $$L(\theta) = -\sum_i \log p_{t_i} + \frac{\lambda}{2}\|\theta\|^2$$
>
> where θ is the set of all parameters $\{W_1^w, W_1^t, W_1^l, b_1, W_2, E^w, E^t, E^l\}$. A slight variation is that we compute the softmax probabilities only among the feasible transitions in practice.

  - **Comput l2-regularization**: Find params from self.named_parameters with name in ['embeddings.weight', '_W1.weight', '_W1.bias', '_W2.weight'], apply l2-norm on each of them and then apply self._regularization_lambda on their sum and divided by 2.
  - **Compute cross entropy loss**: Find feasible logits according to feasible labels. Then apply negative LogSoftmax on feasible logits, then multiply by feasible labels, which gives the probabilities of the correct output. The cross entropy loss is the sum of these.
  - Thus the loss is the sum of the l2-regularization and the cross entropy loss.

2. Experiments

2.1. Default configuration

device	CPU
training data	train.conll
validation data	dev.conll
pre-trained embeddings	glove.6B.50d.txt
epochs	5
embedding_dim	50
regularization_lambda	1e-08

2.2. Test results

Name	**basic**	**wo_glove**	**wo_emb_tune**	**tanh**	**sigmoid**
activation	cubic	cubic	cubic	tanh	sigmoid
tunnable	❌	❌	✅	❌	❌
pre-trained	✅	❌	✅	✅	✅
UAS	85.223	71.520	88.327	82.097	84.682
UASnoPunc	86.958	74.334	89.792	83.963	86.435
LAS	82.753	67.024	85.784	79.203	82.030
LASnoPunc	84.166	69.524	86.912	80.716	83.442
UEM	29.118	11.118	35.706	23.176	28.882
UEMnoPunc	31.529	11.823	38.706	24.471	30.647
ROOT	84.706	63.353	90.588	80.706	84.411

2.3. Analysis

- Among different activations within cubic, tanh and sigmoid, under the same other configurations, cubic performs the better than the others. Which verifies the results given in the paper that "cubic can achieve 0.8% ~ 1.2% improvement in unlabeled attachment scores (UAS) over tanh and other functions". Reasons are also stated in the paper, with cubic activation, the input

product term of three elements could come from different dimensions of three embeddings, which is benificial for dependency parsing.

- For cubic activation, tunnable pre-trained word embeddings performs the best, while non-tunnable randomly intialized embeddings within (-0.01, 0.01) gives the worst results. Assumptions are that these word2vec embeddings captures amonts of information of words which could relate and benifit to the dependency analysis. With tunnable word2vec, the training error derivatives will be backpropagated to these embeddings during the training process, thus we can see that the results are improved by at least around 3%.