CSE 538 NLP Assignment - 3 Report
114707888, Shiva Sankeerth Reddy Yarradla

Implementation:

1. Arc-standard algorithm:

Transition-based dependency parsing draws on shift-reduce parsing, a paradigm originally developed for analyzing programming languages. It aims to predict a transition sequence from an initial sequence. Here, we'll have a stack on which we build the parse, a buffer of tokens to be parsed, and a parser that takes actions on the parse via a predictor called a guide.

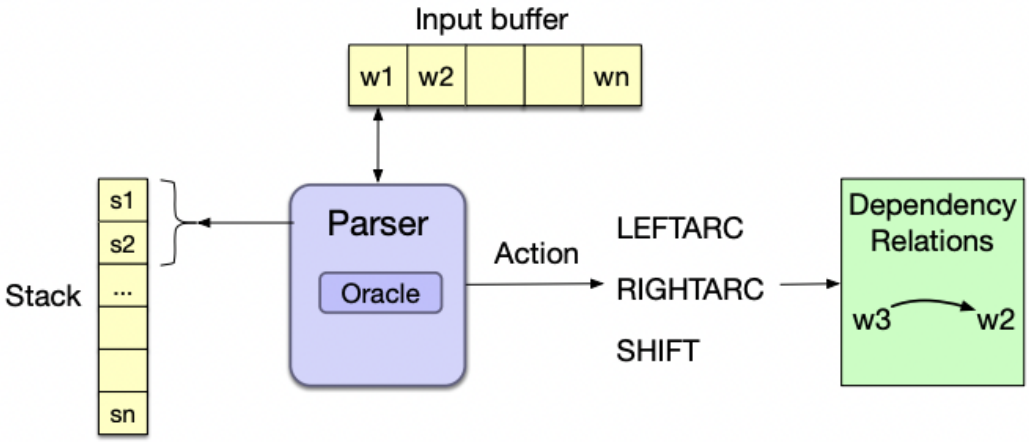

Figure-1 Basic Transition-based dependency parser. This figure was acquired from Speech and Language Processing. Daniel Jurafsky & James H. Martin.

The parser starts in an initial configuration. At each step, it asks the guide/oracle to choose between one of several transitions (actions) into new configurations. Parsing stops if the parser reaches a terminal configuration. The parser returns the dependency tree associated with the terminal configuration.

The arc-standard algorithm is a simple algorithm for transition-based dependency parsing. It is very similar to shift-reduce parsing as it is known for context-free grammars.

In **parsing_system.py**, there is an overall wrapper that begins with the initial state and asks the classifier to provide a distribution over the available actions, selects the next available legal action, and then changes the state by applying the action.

In **apply** method of the **ParsingSystem** class, I implemented the Arc-standard algorithm as given by Nirve using three types of transition actions,

1. LEFT-ARC: If the transition string starts with 'L(', we assert a head-dependent relation between the word at the top of the stack and the second word i.e. the second word on the stack is made

dependent on the top of the stack. And then remove the second word from the stack. This transition is called ‘Left-shift’.

2. RIGHT-ARC: If the transition string starts with ‘R(’, we assert a head-dependent relation between the second word on the stack and the word at the top i.e. the top of the stack is made dependent on the second word on the stack. And then remove the top word from the stack. This transition is called ‘Right-shift’.
3. SHIFT: If the transition string doesn’t start with either of ‘L(’ or ‘R(’, the shift action is implemented where the top of the buffer is popped and added as a top word in the stack.

2. Feature Extraction:

The features are from the state of the stack, buffer, and parse tree at that instance of the parsing of the sentence. What we wish to predict is the next transition of all the possible transitions at that instance. This will help with generating a better dependency parse tree as we make the best possible guess for adding an arc to the parse at any given state. We extract certain features, 48 in total, as mentioned in the paper, ‘**A fast and accurate dependency parser using neural networks**’, by Chen and Manning.

18 of these are for the words, another 18 for the PoS tags, and 12 features for dependency arcs.

In **data.py**, I implemented the method **‘get_configuration_features’** to extract the features according to the paper by Chen and Manning. The following calculation shows how the resultant number of features is calculated,

- Top 3 words from the stack and buffer (3+3=6).
- Child Tokens:
    - For each of the top 2 words on the stack,
        - First right child and first left child (2*2)
        - Second right child and Second left child (2*2)
        - Left-child of left-child and right-child of right-child child (2*2)
    - Total child tokens = 4 + 4 + 4 = 12
- There are (6+12) = 18 items for each of which PoS and words should be extracted.
    - So, (18 * 2) = 36
- We extract labels for the child tokens from step 2 i.e., excluding the top 3 words from stack and buffer.
    - So, total tokens = 36 + (18 - 6) = 36 + 12 = 48

3. Neural Network Architechture

Our model is a neural network for dependency parsing. We try to emulate the dependency parser as mentioned in the paper by Chen and Manning. In accordance with the model architecture in the paper, we build a 2 layer architecture.

For the first layer, we have a hidden_dim number of units. The first layer is followed by an activation function. The paper mentions the comparison of various possible activation functions, namely, cubic, hyperbolic tangent, and sigmoid. We try each as part of our experiments and analyze the results from each. The next layer contains num transitions number of units, with each unit corresponding to the probability of it being the next transition in constructing the parse tree. We get the probabilities by applying a softmax function to the second layer.

The neural network was implemented in **'model.py'**,

1. In the __init__ module,
    a. I initialized the embeddings between -0.01 and 0.01 using a uniform random initializer.
    b. I initialized the weights, **hidden_layer_weights** and **weight_sm**, randomly using the truncated_normal method using the standard deviation of **1/math.sqrt[size of the weight considered]**
    c. The bias, **hidden_layer_bias**, was also initialized in the same way as above.
2. In the forward method,
    a. The input to the first layer is a tensor of embeddings of the tokens. We reshape these inputs into a long one-dimensional vector to conform to the requirements of the input to the first layer of our model.
    b. We multiply the reshaped embeddings with that of our **hidden_layer_weights** and add bias **hidden_layer_bias**.
    c. We apply non-linear transformations to the above result using an activation function. The activation function can be anything from sigmoid, tanh, cubic.
    d. We get our final logits by multiplying the output of the above transformation with that of the weight **weight_sm**.

4. The Loss function:

When we are training the model, we also need to compute the loss for backpropagation which will update our trainable variables, **hidden_layer_weights, weight_sm, hidden_layer_bias,** and our embeddings when they are trainable. The loss we calculate has 2 components, the cross-entropy loss, and a regularization term, which are added to compute the final loss.

1. Labels, 1 represents correct transition, 0 incorrect transition, and -1 for invalid transition.
2. We have to find the illegal transitions and we remove logits corresponding to those impossible transitions.
3. Then, we convert the remaining values to probabilities using the softmax function.
4. Then, using correct labels, we get the probability values corresponding to that transition and we maximize them by taking a negative log of that value resulting us the **loss**.

5. The regularization term consists of L2 loss for our trainable variables. We multiply the sum of the L2 loss for our trainable variables with the regularization parameter (**_regularization_lambda**) to compute the regularization.
6. Then, we add regularization to the loss which results in the total loss.

Experiments:

Experiment	Activation	Loss	UAS	UASnoPunc	LAS	LASnoPunc	UEM	UEMnoPunc	ROOT
Basic	Cubic	0.14	84.929	86.74	82.249	83.711	28.47	30.882	85.411
Sigmoid	Sigmoid	0.25	77.62	79.952	74.427	76.386	17.294	18.411	72.411
tanh	Tanh	0.19	82.047	84.044	79.253	80.91	22.764	24.47	80.529
Without pre-trained	Cubic	0.09	84.762	86.604	82.316	83.883	27.941	30.352	82
Without tuning	Cubic	0.10	87.588	89.216	85.135	86.44	33.47	36.294	88.764

Table 1: Experiment Results

The results in Table-1 are for the 5 experiments carried out. The numbers reported are numbers after the 5 epochs run for each experiment. Below are the meanings of some of the relevant columns.

- Activation: The activation function used in layer 1 of our model.
- Loss: The average training loss for that epoch.
- UAS: Unlabeled Attachment Score - Percentage of words that get the correct head.
- UASnoPunc: Unlabeled Attachment Score (Without Punctuation) - Percentage of words that get the correct head. Here we disregard punctuation tokens.
- LAS: Labeled Attachment Score - Percentage of words that get the correct head and the label.
- LASnoPunc: Unlabeled Attachment Score (Without Punctuation) - Percentage of words that get the correct head and the label. Here we disregard punctuation tokens.
- UEM: Unlabeled Attachment Score with Exact Match - Percentage of exact match dependency parse trees.
- UEMnoPunc: Unlabeled Attachment Score with Exact Match (Without Punctuation)- Percentage of exact match dependency parse trees. Here we disregard punctuation tokens.

1. **Affect of Activation on the learning:**
    a. The activations used were cubic, tanh, and sigmoid. The cubic activation as seen in the **basic experiment** performed on par with the tanh and better than sigmoid on all measures. Cubic performed the best across most metrics.
    b. Cubic beats both these activations by almost 2% and 6% respectively when it comes to UASNoPunc.
2. **Affect of Pre-trained Embeddings on the learning:**
    a. The use of pre-trained embeddings is a positive factor. The experiment without any pre-trained embeddings reported the lowest scores on all fronts.
3. **Tunability of Embeddings:**
    a. For the last experiment, we use pre-trained embeddings but do not train/learn them over the course of the training. To my surprise, this model managed to get the best results on all fronts, even out-performing the cubic activation function.