Homework 3

Amirhesam Abedsoltan
University of Southern California
abedsolt@usc.edu

Abstract

This document contains our implementation details and results on the dependency parser based on arc-standard parsing method. Our model is based on the Neural dependency parser proposed by (Chen and Manning, 2014). Our results show that our model is able to correctly predict 86% of the arcs correctly(without label, 80% with label) .

1 General framework

The final goal of this paper is that for any given sentence our algorithm generates the head(or parent) of each word and its relationship with its heads, namely its arc-label. As a general procedure for any classification task we need to first generate features from each sentence and use those features to predict both head and arc-label of each word. We should be careful about the input of our neural network. The input is in fact the configuration or state of the sentence which is based on Arc-Standard Parser. We first briefly summarize the idea behind Arc-Standard Parser and feature generating in section 2 then in secretion 3 we fill in the details about the algorithm and our neural network. In section 4 we talk about the details of parsing using the trained model. Finally in section 5 we talk about our extra mile efforts.

2 Data pre-processing

In the first sub-section, we first talk about Arc-Standard Parser to clarify what are inputs and outputs of the neural network and in the second subsection we talk in details about the chosen features extracted from each configuration.

2.1 Arc-Standard Parser

Arc-Standard(Nivre et al., 2006) is one of the well-known transition based parsing methods to generate dependency parsing tree (example in figure 1) from the given sentence.

The way algorithm works is as below: (Consider figure 2 from the course notes as an example)

1. We define a configuration as a tuple of $c = (s, b)$, where S is stack initialized as $S = [\text{"ROOT"}]$ b is buffer initialized with all words in the sentence. For example in the figure 2 it would be b = [1,2,3,4,5], where each number is equivalent to its corresponding word. For instance 2 is for "the".

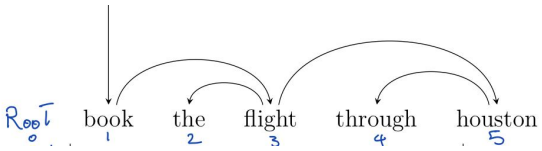

Figure 1: An example of parsing

2. As long as buffer is not empty and stack has members other than "ROOT", do as below: (S[i] mans i^{th} top element of stack same for the buffer)
   (a) If $|S| \geq 2$ and s[0] is head of s[1] with label a, then do a Left arc. Meaning eliminate s[1] from the stack.
   (b) If $|S| \geq 2$ and s[1] is head of s[0] with label a and no dependent of s[0] is in the buffer, then do a Right arc. Meaning eliminate s[0] from the stack.
   (c) If $|S| = 1$ or neither of cases above were true then do a shift. Meaning eliminate b[0] from the buffer and ad it to the top of the stack.

   Basically every time that we do one of these actions the configuration, meaning $C = (s, b)$, changes. So, we want to propose a neural network such that given a configuration, as input,

it gives us the correct action, meaning one of the a, b or c, along with the correct arc-label, meaning the relation between head and its dependent. To sum up, the neural network gets some features extracted from a specific configuration and outputs a (a,t) where "a" is the relation between head and its dependent and t is either left, right or shift. Note, that if the action is shift then we do not have any a. Therefore, if we assume there are L possible relation between a head and its dependent then there are $2 \cdot L + 1$ possibilities for (a,t). In our code we assigned an integer to each possibilities as below: if it's a shift action then the neural network should output 0. if it is left action then the neural network should outputs a number between 1 to L depending on the relation between head and its dependent and if it's a right action it should outputs a number between L+1 to 2*L depending on the relation between head and its dependent. See example in picture 3 from course notes for more details.

1	ROOT	book the flight through houston	SHIFT
2	ROOT book	the flight through houston	SHIFT
3	ROOT book the	flight through houston	SHIFT
4	ROOT book the flight	through houston	LEFT
5	ROOT book flight	through houston	SHIFT
6	ROOT book flight through	houston	SHIFT
7	ROOT book flight through houston		LEFT
8	ROOT book flight houston		RIGHT
9	ROOT book flight		RIGHT
10	ROOT book		RIGHT
11	ROOT		Done

Figure 2: An example of parsing(full version)

The only part left to be explained and it is unclear above is what sort of features should we extract from a given configuration. We discuss this in the next sub-section.

2.2 Generating features from a configuration

according to Danqi paper we generate these features from each configuration:

1. Top 3 words on the stack. 3 features.
2. First three words in the buffer. 3 features
3. The first and second leftmost/rightmost children of top two words on the stack. 8 features
4. The leftmost of leftmost/rightmost of rightmost children of top two words on stack. 4 feature
5. POS tags of category 1,2,3,4. Total (3+3+8+4=18) features.
6. arc-label(if it's available) for category 3 and 4. Total (8+4=12)

In total we have words(3+3+8+4)+POS(18) + arc-labels(12) = 48 features per configuration. It's clear that there are many cases which these children or words do not exist. Thus, we need to put aside 3 embedding for each word, POS and arc-label for empty cases when words do not exist. We used embedding of size 50. Thus the final input of the neural network would be a vector of size $50 * 48 = 2400$ for each configuration. Some important considerations:

1. In the code we have named only the first 4 categories for the features because basically 5 and 6 are the same words but with POS and arc-labels.
2. The way we train the model is important, we should be careful that even we have access to all heads and arc-labels in the training we should not use them as features when they have not been yet identified. For instance, in the example of figure 2 when we are in state of line 5. we should not add features of "through" as the leftmost child of "Houston". Because, we do not know this yet.
3. In addition to 2, we use the the embedding features based on what model predicts about the most probable head of each word otherwise we just use the empty embedding.
4. For each sentence with l words, it take $2 \cdot l$ steps of changing configuration to be completely parsed. Because each word should once get shifted from buffer to stack and then it should get eliminated from the stack. Thus, each sentence generates $2 \cdot l$ sample for our neural network for training.

3 Training

As promised, in this section we will fill in the detail about the model and algorithm. We should be careful that the algorithm uses the neural network model to keep updating the configuration with a greedy approach. In subsection 3.1 we talk about the neural network model and in the subsection 3.2 we talk about the non-projecive trees and how we eliminate them from the training data set. Finally,

in subsection 3.3 we talk about how we trained the model.

3.1 Neural Network model

As mentioned earlier, the NN model gets a vector of size $50 * 48 = 2400$ and predicts one of possible $2 * L$ labels as output. You can see the model in figure 3 from the original paper. We discussed how

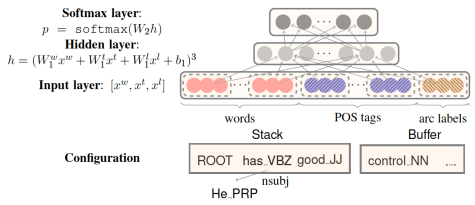

Figure 3: Neural network model

to generate features from a configuration and what are the important considerations extensively in the previous section. The hidden layer was chosen 200 using the same number as the original paper. Also, we tried both cubic and tangh activation function and we talk about it in the "extra mile" section.

3.2 None-projective trees

The arc standard assumes that the dependency parse tree of the sentence is projective, meaning no two edges cross each other assuming all the tokens are placed in a 2D plane with the same order as they appear in the sentence. Figure 4 shows an example of a non-projective parsing, where its arcs are obviously crossing each other no matter one draws them.

So, in the code we checked two possible ways

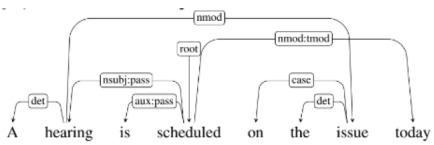

Figure 4: a non-projective example from course notes

where non-projective can happen:

1. for each arc we checked whether some outer word had a child between these two words or not. This is the case of figure 4.

2. Also, we need to check whether a word between the arc has a child outside the the words within the arc.

The total number of non-projective sentence in the Train-set were 120 which is about 0.03% of total samples in the Train set. Also, there were only 5 non-projective samples in the DEV set which is again about 0.03% of total samples in DEV set.

3.3 Training

As mentioned in subsection 2.2, each sentence with l words generates $2 \cdot l$ configuration with its corresponding label. We use these generated configuration along with their labels to train our neural network by using cross-entropy loss along with l_2-norm regulizer as below:

$$L(\theta = -\sum_i log(p_{t_i}) + \frac{\lambda}{2}||\theta||^2 \qquad (1)$$

Where p_{t_i} is the probability of true label predicted by neural network. Also, for the optimization part we used Adagrad with learning rate 0.01. For the embeddings we trained all word, POS and arc-label embedding from scratch initialized uniformly random in [-0.01,0.01]. We talk more about embedding in "extra-mile" section.

4 Parsing and results

After training the model we use the neural network to parse DEV and Test data set. We do as below:

Algorithm 1: Parsing DEV(Test)

```
Result: parsed CONLL file
for sample in DEV(TEST) do
    s = ["ROOT"];
    b = [1,2,...,number of words in sample]
    while s!=["ROOT"] or b!=[] do
        config = gen(s,b)
        output = model(config)
        l = arg max_{feasible labels} output
        update s and b based on the label:
        s,b = update(s,b,l)
    end
    Write the exatracted information in
     CONLL file
end
```

The final result that we got on DEV and Test set is as below:

Data set	UAS	LAS
DEV	86%	82%
Test	80%	76%

Table 1: Results.

Also, we attached the human readable file for DEV set as DEV.vocab. It was helpful for debugging process.

5 Extra mile

Four important observation has been done for this part:

1. The role of initialization of embedding is really important. We tried 3 kind of initialization:

   (a) All start from zero vector. This was really slow ad takes a lot to converge. The results after 10 epochs are as below: The reason is that it takes more time for the model to differentiate different words, POS and labels.

Data set	UAS	LAS
DEV	63%	59%
Test	60%	57%

   Table 2: all zero embedding Results.

   (b) using the same values as the paper, meaning uniform random in [-0.01,0.01] interval. This was really fast and converged eve after one or 2 epochs. We showed the results i the last section.

   (c) We wanted to see what happens if we assign values from a broader range like [-10,10] or [-1,1]. It turned out the bigger the interval the slower the convergences. It seems it takes more time for the model to converge from large values to optimal and final values. for [-10,10] the result was as below:

Data set	UAS	LAS
DEV	65%	62%
Test	63%	60%

   Table 3: Huge initilization results.

2. We also tried cubic activation. In figure 5 you can see both Tangh and cubic function. As you can see the cubic derivitive near 0 is really small and this makes the progress really slow since the wights are mostly small and near zero. Thus, Tangh is expected to converge much faster. Indeed this was the case and although we got almost same result using cubic activation but we had to run the model for more epochs to get acceptable results shown below in Ta le 4.

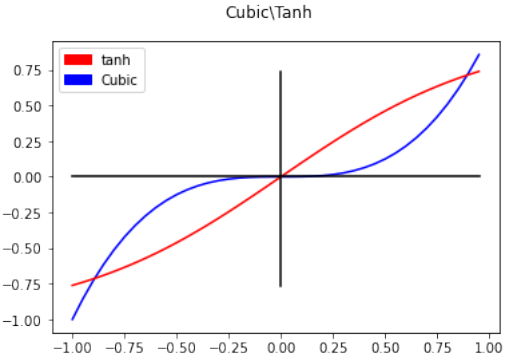

Figure 5: Cubic Vs. Tangh

Data set	UAS	LAS
DEV	85%	82%
Test	78%	75%

Table 4: Cubic Vs tangh results

3. We tried l_2-norm with different λs from 0 to 1. However, the results were not that different. without l_2-norm meaing $\lambda = 0$ there was a small reduction in generalization accuracy. It became 76% which was 4% reeducation. For huge values though the results started to become really low. The reason was that even the training now cannot fit properly because we are putting a lot of penalty for making weights big. Thus, it cannot fit to the training data set.

4. we tried to add glove pre-trained embedding using spacey library. If you uncomment line: 16,23,24,31 and 44 to 66 and comment line:15,20,28,42,74 in "modeltorch.py" it will use pre-trained model. Unfortunately, due to lack of time and google cloud credits we could not run it thoroughly.

References

Danqi Chen and Christopher D. Manning. 2014. A fast and accurate dependency parser using neural networks.

Joakim Nivre, Johan Hall, and Jens Nilsson. 2006. A data-driven parser-generator for dependency parsing.