NLP Homework: Dependency Parsing with Feed-Forward Neural Network

Submission Deadline: Monday Dec. 11th, 5 pm

1 Background on Dependency Parsing

Dependency trees are one of the main representations used in the syntactic analysis of sentences.[1] One way to recover a dependency tree for a sentence is through transition-based methods. In transition-based parsing, a tree is converted to a sequence of shift-reduce actions and the parser decides between possible actions depending on the current configuration of the partial tree.

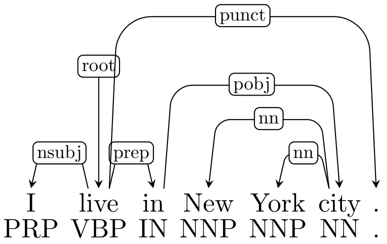

Figure 1: An example dependency tree from the Wall Street Journal treebank.

In this homework, you are going to train a dependency parsing model based on the "arc-standard" system [2]. In the arc-standard system, each parse state is a *configuration* $\mathcal{C} = \{\sigma, \beta, \alpha\}$, in which σ is a stack of processed words, β is an ordered list of unprocessed words and α is a set of already recovered dependency relations $j \xrightarrow{l} i$ (ith word is headed by the jth word with a dependency label l). The initial configuration is

$$\mathcal{C}^{(0)} = \{\sigma = [root_0], \beta = [w_1, \cdots, w_n], \alpha = \{\}\}$$

where *root* is the dummy root node of the sentence and $[w_1 \dots w_n]$ are the words in the sentence. One can theoretically show that the arc-standard parser can

[1] The definitions and explanations in this section are meant to be very brief. This is mainly because you will not really implement the parsing actions by yourself and we have already provided that for you. If you are interested, you can look at the corresponding chapter in the NLP textbook: `https://web.stanford.edu/~jurafsky/slp3/14.pdf`.

$$\begin{array}{ll}
\textbf{Initialization} & \{\sigma = [root], \beta = [w_1, \cdots, w_n], \alpha = \{\}\} \\
\textbf{Termination} & \{\sigma = [root], \beta = [], \alpha = \{j \xrightarrow{l} i; 0 \leq j \leq n, \forall i \in [1, \cdots, n]\}\} \\
 & \\
\textbf{Shift} & \{\sigma, i|\beta, \alpha\} \rightarrow \{\sigma|i, \beta, \alpha\} \\
\textbf{Left-Arc}^{label} & \{\sigma|ij, \beta, \alpha\} \rightarrow \{\sigma|j, \beta, \alpha \cup \{j \xrightarrow{label} i\}\} \\
\textbf{Right-Arc}^{label} & \{\sigma|ij, \beta, \alpha\} \rightarrow \{\sigma|i, \beta, \alpha \cup \{i \xrightarrow{label} j\}\}
\end{array}$$

Figure 2: Initial and terminal states and actions in the arc-standard transition system. The notation $\sigma|ij$ means a stack where the top words are $\sigma_0 = j$ and $\sigma_1 = i$.

Action	σ	β	$h \xrightarrow{l} d$
Shift	[root$_0$]	[I$_1$,live$_2$, in$_3$, New$_4$, York$_5$, city$_6$, .$_7$]	
Shift	[root$_0$, I$_1$]	[live$_2$, in$_3$, New$_4$, York$_5$, city$_6$, .$_7$]	
Left-Arc^{nsubj}	[root$_0$, I$_1$, live$_2$]	[in$_3$, New$_4$, York$_5$, city$_6$, .$_7$]	$2 \xrightarrow{nsubj} 1$
Shift	[root$_0$, live$_2$]	[in$_3$, New$_4$, York$_5$, city$_6$, .$_7$]	
Shift	[root$_0$, live$_2$, in$_3$]	[New$_4$, York$_5$, city$_6$, .$_7$]	
Shift	[root$_0$, live$_2$, in$_3$, New$_4$]	[York$_5$, city$_6$, .$_7$]	
Shift	[root$_0$, live$_2$, in$_3$, New$_4$, York$_5$]	[city$_6$, .$_7$]	
Left-Arc^{nn}	[root$_0$, live$_2$, in$_3$, New$_4$, York$_5$, city$_6$]	[.$_7$]	$6 \xrightarrow{nn} 5$
Left-Arc^{nn}	[root$_0$, live$_2$, in$_3$, New$_4$, city$_6$]	[.$_7$]	$6 \xrightarrow{nn} 4$
Right-Arc^{pobj}	[root$_0$, live$_2$, in$_3$, city$_6$]	[.$_7$]	$3 \xrightarrow{pobj} 6$
Right-Arc^{prep}	[root$_0$, live$_2$, in$_3$]	[.$_7$]	$2 \xrightarrow{prep} 3$
Shift	[root$_0$, live$_2$]	[.$_7$]	
Right-Arc^{punct}	[root$_0$, live$_2$, .$_7$]	[]	$2 \xrightarrow{punct} 7$
Right-Arc^{root}	[root$_0$, live$_2$]	[]	$0 \xrightarrow{root} 2$
Terminal	[root$_0$]	[]	

Figure 3: A sample action sequence with the arc-standard actions (Figure 2) for the tree in Figure 1.

reach the *terminal state* after applying exactly $2n$ actions where n is the number of words in the sentence. The final state will be

$$\mathcal{C}^{(2n)} = \{\sigma = [root], \beta = [], \alpha = \{j \xrightarrow{l} i; 0 \leq j \leq n, \forall i \in [1, \cdots, n]\}\}$$

There are three main actions that change the state of a configuration in the algorithm: shift, left-arc and right-arc. For every specific dependency relation (e.g. subject or object), the left-arc and right-arc actions become fine-grained (e.g. RIGHT-ARC:nsubj, RIGHT-ARC:pobj, etc.). Figure 2 shows the conditions for applying those actions. Figure 3 shows a sequence of correct transitions for the tree in Figure 1.

2 Learning a Parser from Transitions

It is straightforward to train a model based on transitions extracted from trees in each sentence. A python script is provided for you to convert dependency trees to training instances, so you need not worry about the details of converting dependencies to training instances.

2.1 Input Layer

We follow the approach of [1, 3] and use three kinds of features:

- Word features: 20 types of word features.
- Part-of-speech features: 20 types of POS features.
- Dependency label features: 12 types of dependency features.

The data files should have 53 columns (space separated), where the first 20 are word-based features, the next 20 are the POS features, the next 12 are dependency features and the last column is the gold label (action).

2.2 Embedding Layer

Based on the the input layer features $\phi(D_i)$ for data D_i, the model should use the following embedding (lookup) parameters:

- Word embedding (E_w) with dimension d_w. If the training corpus has N_w unique words (including the $< null >$, $< root >$, and $< unk >$ symbols), the size of the embedding dictionary will be $\mathbb{R}^{d_w \times N_w}$.
- Part-of-speech embedding (E_t) with dimension d_t. If the training corpus has N_t unique POS tags (including the $< null >$ and $< root >$ symbols), the size of the embedding dictionary will be $\mathbb{R}^{d_t \times N_t}$.
- Dependency label embedding (E_l) with dimension d_l. If the training corpus has N_l unique dependency labels (including the $< null >$ and $< root >$ symbols), the size of the embedding dictionary will be $\mathbb{R}^{d_l \times N_l}$.

Thus the output from the embedding layer e, is the concatenation of all embedding values for the specific data instance.

$$d_e = 20(d_w + d_t) + 12d_l$$

2.3 First Hidden Layer

The output of the embedding layer e should be fed to the first hidden layer with a rectified linear unit (RELU) activation:

$$h_1 = RELU(W^1 e + b^1) = max\{0, W^1 e + b^1\}$$

where $W^1 \in \mathbb{R}^{d_{h_1} \times d_e}$ and $b^1 \in \mathbb{R}^{d_{h_1}}$.

2.4 Second Hidden Layer

The output of the first hidden layer h_1 should be fed to the second hidden layer with a rectified linear unit (RELU) activation:

$$h_2 = RELU(W^2 h_1 + b^2) = max\{0, W^2 h_1 + b^2\}$$

where $W^2 \in \mathbb{R}^{d_{h_2} \times d_{h_1}}$ and $b^2 \in \mathbb{R}^{d_{h_2}}$.

2.5 Output Layer

Finally, the output of the second hidden layer h_2 should be fed to the output layer with a softmax activation function:

$$q = softmax(V h_2 + \gamma)$$

where $V \in \mathbb{R}^{A \times d_{h_2}}$ and $\gamma \in \mathbb{R}^A$. A is the number of possible actions in the arc-standard algorithm.

Note that in decoding, you can find the best action by finding the arg max of the output layer without using the softmax function:

$$y^* = \arg \max_{i \in [1 \ldots A]} l_i$$

and

$$l_i = (V h_2 + \gamma)_i$$

The objective function is to minimize the negative log-likelihood of the data given the model parameters ($-\sum_i \log q_{y^i}$). It is usually a good practice to minimize the objective for random mini-batches from the data.

3 Step by Step Guideline

This section provides a step by step guideline to train a deep neural network for transition-based parsing. You can download the code from the following repository: `https://github.com/rasoolims/nlp_hw_dep`.

3.1 Vocabulary Creation for Features

Deep networks work with vectors, not strings. Therefore, first you have to create vocabularies for word features, POS features, dependency label features and parser actions.

Run the following command:

```
>> python src/gen_vocab.py trees/train.conll data/vocabs
```

After that, you should have the following files:

- **data/vocabs.word**: This file contains indices for words. Remember that $< null >$ is not a real word but since there are some cases that a word feature is null, we also learn embedding for a null word. Note that $< root >$ is a word feature for the root token. Any word that does not occur in this vocabulary should be mapped to the value corresponding to the $< unk >$ word in this file.

- **data/vocabs.pos**: Similar to the word vocabulary but this is for part-of-speech tags. There is no notion of unknown in POS features.

- **data/vocabs.labels**: Similar to the word vocabulary but this is for dependency labels. There is no notion of unknown in dependency label features.

- **data/vocabs.actions**: This is a string to index conversion for the actions (this should be used for the output layer).

3.2 Data Generation

The original tree files cannot be directly used for training. You should use the following command to convert the training and development datasets to data files:

```
>> python src/gen.py trees/train.conll data/train.data
```

```
>> python src/gen.py trees/dev.conll data/dev.data
```

Each line shows one data instance. The first 20 columns are for word-based features. The second 20 columns are for the POS-based features. The next 12 features are for dependency labels. The last column shows the action. Remember that, when you want to use these features in implementation, you should convert them to integer values, based the vocabularies generated in the previous subsection.

3.3 Neural Network Implementation

Now it is time for implementing a neural network model. We recommend you to implement it with the *Dynet* library[2] because we can help you with details but if you are more familiar with other libraries such as Pytorch, Chainer, TensorFlow, Keras, Caffe, CNTK, DeepLearning4J, and Thiano, that is totally fine. Feel free to use any library or implementation of feed-forward neural networks. You can mimic the same implementation style as in the POS tagger implementation in the following repository: `https://github.com/rasoolims/ff_tagger`.

[2] `https://github.com/clab/dynet`: take a look at its sample codes in the paper: `https://arxiv.org/pdf/1701.03980.pdf` and its API reference: `https://github.com/clab/dynet/blob/master/examples/jupyter-tutorials/API.ipynb`.

3.4 Decoding

After you are done with training a model, you should be able to parse sentences directly from tree files. We provided wrappers for you to be able to run a decoder. Bellow is a container for your decoder in the “depModel.py” file. You can change the content of the class (especially the score function), to make it work as a correct decoder.

```
import os,sys
from decoder import *

class DepModel:
    def __init__(self):
        '''
            You can add more arguments for examples actions and model paths.
            You need to load your model here.
            actions: provides indices for actions.
            it has the same order as the data/vocabs.actions file.
        '''
        # if you prefer to have your own index for actions, change this.
        self.actions = ['SHIFT','LEFT-ARC:cc',...,'RIGHT-ARC:root']
        # write your code here for additional parameters.
        # feel free to add more arguments to the initializer.

    def score(self, str_features):
        '''
        :param str_features: String features
        20 first: words, next 20: pos, next 12: dependency labels.
        DO NOT ADD ANY ARGUMENTS TO THIS FUNCTION.
        :return: list of scores
        '''
        # change this part of the code.
        return [0]*len(self.actions)

if __name__=='__main__':
    m = DepModel()
    input_p = os.path.abspath(sys.argv[1])
    output_p = os.path.abspath(sys.argv[2])
    Decoder(m.score, m.actions).parse(input_p, output_p)
```

If you just run the code as-is, it will generate default trees with very low accuracies (because it only generates zero scores for every possible action).

```
>> python src/depModel.py trees/dev.conll outputs/dev.out
```

4 Homework Questions

This section provides details about the questions. Please put your code as well as required outputs in the corresponding directories. A blind test set is also provided for you (*trees/test.conll*). You do not have access to the correct dependencies in the blind testing data but your grade will be affected by the output from that blind set.[3]

Part 1

Table 1 provides details about the parameters you need to use for this part. Except the ones mentioned, every other setting should be the default setting of the neural network library that you use.

Parameter	Value
Trainer	Adam algorithm
Training epochs	7
Transfer function	RELU
Word embedding dimension	64
POS embedding dimension	32
Dependency embedding dimension	32
Minibatch size	1000
First hidden layer dimension (d_{h_1})	200
Second hidden layer dimension (d_{h_2})	200

Table 1: Parameter values for the experiments.

1. After training the model with 7 epochs on the training data, run the decoder on the file $trees/dev.conll$ and put the output in $trees/dev\_part1.conll$.[4]
2. Run the following script to get the unlabeled and labeled dependency accuracies. Report this number in your document.

   ```
   >> python src/eval.py trees/dev.conll outputs/dev_part1.conll
   ```

3. A blind test set is also provided for you ($trees/test.conll$). Run the decoder with your trained model on the blind test set and output it in $trees/test\_part1.conll$.

Part 2

This part is roughly the same as part one but with the following differences: $d_{h_1} = 400$ and $d_{h_2} = 400$.

Similar to part 1, do the following steps:

1. After training the model with 7 epochs on the training data, run the decoder on the file $trees/dev.conll$ and put the output in $trees/dev\_part2.conll$.
2. Run the following script to get the unlabeled and labeled dependency accuracies. Report this number in your document.

[3] Because of the randomness in deep learning, we do not expect a particular performance but your accuracies should not be far bellow the numbers that we produce.

[4] Remember to shuffle you training data at each epoch.

```
>> python src/eval.py trees/dev.conll outputs/dev_part2.conll
```

3. Run the decoder with your trained model on the blind test set and output it in *trees/test_part2.conll*.

4. Why do you think the accuracy changed?

Part 3

This part is open-ended so you are free to change the parameters. Try at least one variation in of model and report the accuracy on the development data. Use the best model to output *trees/dev_part3.conll* and *trees/test_part3.conll* files.

The variations can come from the following (though you are not bounded to these variations):

- Changing the activation function. For example, using the leaky RELU instead of the original RELU activation function.
- Initializing word embeddings with pre-trained vectors for example from `https://nlp.stanford.edu/projects/glove/`.
- Changing dimensions of the hidden or embedding layers.
- Dropout during training.
- Changing the mini-batch size.
- Changing the training algorithm (e.g. SGD, AdaGrad) or changing the updater's learning rate.
- Changing the number of training epochs.

In addition to reporting the accuracies, you should explain your observations and reasons for the improvement that you achieve.

References

[1] Chen, D., and Manning, C. A fast and accurate dependency parser using neural networks. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)* (2014), pp. 740–750.

[2] Nivre, J. Incrementality in deterministic dependency parsing. In *Proceedings of the Workshop on Incremental Parsing: Bringing Engineering and Cognition Together* (2004), Association for Computational Linguistics, pp. 50–57.

[3] Weiss, D., Alberti, C., Collins, M., and Petrov, S. Structured training for neural network transition-based parsing. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)* (2015), Association for Computational Linguistics, pp. 323–333.