Dependency parsing using Neural Network

1 Transition-based Dependency Parsing

A transition-based parser frames the problem of parsing as a series of action (shift/reduce) decisions given a configuration. Dependency parsing is an NLP parsing problem where the objective is to obtain a grammatical structure in the form of a parse tree. A parse tree contains directed edges going from a source (or, head) token to a target (or, dependent) token. Each edge is labeled with a dependency relation from a fixed set of relations. As an example, Figure 1 shows a dependency parse tree for the sentence “Mary lives in California”.

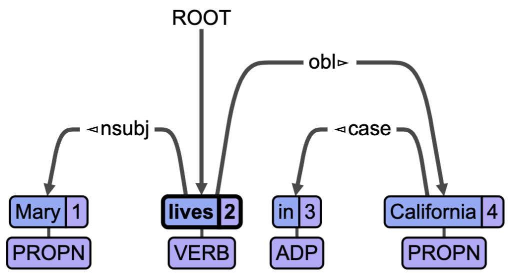

Figure 1: An example dependency parse tree for the sentence “Mary lives in California”.

The configuration of a transition-based parser is also known as a parse state. Transition-based dependency parser’s parse state comprises of three components – the stack, the buffer, and a set of dependencies. Initially, the stack and the set of dependencies are empty, while the buffer is initialized by the set of tokens in the sentence. Given a parse state, the best possible action needs to be predicted to update the parse state. There are three types of actions possible-

1. **Shift**: This moves the next element in the buffer to the top of the stack. Consequently, the stack and the buffer are changed in the parse state.

2. **ReduceLeft$_r$ or LeftArc$_r$**: This takes the top two elements of the stack. A dependency with relation r between the elements is added such that the topmost element of the stack is the head. The second element from the top of the stack is removed. Consequently, the stack and the set of dependencies are changed in the parse state.

3. **ReduceRight$_r$ or RightArc$_r$**: This takes the top two elements of the stack. A dependency with relation r between the elements is added such that the second element from the top of the stack is the head. The topmost element of the stack is removed. Consequently, the stack and the set of dependencies are changed in the parse state.

Action decisions are made till the stack and the buffer are empty. This configuration is known as the final state. At this point, the set of dependencies corresponds to the dependency parse of the sentence. See the lecture notes for a worked-out example.

Note that we haven't mentioned how the actions are decided. This is, typically, done using a multi-class classifier which takes a representation of the parse state as input and produces an action decision.

Evaluation. Dependency parsers are evaluated using two metrics– Unlabeled Attachment Score (UAS), and Labeled Attachment Score (LAS). UAS is the proportion of dependents that were assigned the correct head. LAS is the proportion of dependents that were assigned the correct head with the correct dependency relation label. Intuitively, LAS has to be less than or equal to UAS.

2 Parser for Mini Project

In this mini project, you will be implementing a simpler variation of a transition-based dependency parser proposed by Chen and Manning [2014]. We describe the data files, helper methods, and the format of the submission file below. Next, we will describe the model to be implemented and its various test configurations.

2.1 Data Files

We provide you with the following data files:

1. **data/{train.txt, dev.txt, test.txt}**: These are the data splits that need to be used for training the model, hyper-parameter tuning and testing. The three files are formatted in the same fashion. They contain the input tokens, their part-of-speech (POS) tags, and the gold shift-reduce actions. These three pieces are separated by `|||`.

Each piece is further space-separated. Each line in the file corresponds to a sentence. As an example, here is how a line looks: `‘‘The brown fox ||| DET ADJ NOUN ||| SHIFT SHIFT SHIFT REDUCE_L_amod REDUCE_L_det’’`. NOTE: All trees in the files are strictly projective.

2. **data/hidden.txt**: This is another split formatted similar to the files mentioned above except it does not contain the gold actions. You are expected to submit actions predicted by your model for these sentences (more on this later). Hence an example line in the file looks like: `‘‘The brown fox ||| DET ADJ NOUN’’`.

3. **data/tagset.txt**: This contains the set of 75 possible actions that a model can predict separated by line. Each left reduce action with a relation label r is written as `‘REDUCE_L_r’` and, consequently, each right reduce action with a relation label r is written as `‘REDUCE_R_r’`.

4. **data/pos_set.txt**: This contains the set of all 18 POS tags separated by line. While there are 17 official POS tags, we provide an addition `‘NULL’` tag that can be used for any padding tokens.

2.2 Helper Methods

We provide you with two files that contain helper methods/skeleton codes. While we do not require you to use these files, we believe these will save you time:

1. **scripts/state.py**: This file defines classes that might be useful to maintain state and implement state changes. Each method argument explicitly specifies the data type required. We provide three classes: (a) Token. This creates an instance with a token, its POS tag, and a unique ID. (b) DependencyEdge. This creates an edge instance defining the relation with the given relation label between a source (i.e., head) and target (i.e., dependent), both instances of class Token. (c) ParseState. This contains and maintains the parse state (stack, buffer, and dependency set) of the parser. In addition, this file contains skeleton code for methods that perform state changes. You can fill it with the necessary code.

2. **scripts/evaluate.py**: This contains the helper method (`compute_metrics()`) for evaluating the parser performance. It takes three lists of lists as input. These correspond to the token list, gold actions and predicted actions. Each inner list corresponds to a sentence. This method returns the UAS and LAS metrics for the data input. **NOTE**: This script relies on the methods in ‘state.py’. Please ensure that these methods are appropriately filled.

2.3 Output Format

You will be submitting a ‘results.txt’ file with the predicted actions for the sentences in the ‘hidden.txt’ file. Each action should be space-separated. The actions for different sentences must be in different lines. That is, the actions for a sentence on line k in ‘hidden.txt‘ should be on line k in ‘results.txt‘.

2.4 Model and Configuration

As mentioned before, you will be implementing a simpler version of the parser proposed in Chen and Manning [2014].

2.4.1 Model

As mentioned before, a multi-class classifier is required to decide the action to be taken given a parse state. You will be training this classifier. Remember that the classifier takes some representation of the parse state as the input and produces an action. You will be using information from just the tokens and the POS tags to compute this representation. More specifically, you need to take the top `c=2` elements of the stack and buffer and their corresponding POS tags. We denote these as:

$$w = [stack_{-1}, ..., stack_{-c}, buffer_0, ..., buffer_{c-1}]$$
$$p = [POS(stack_{-1}), ..., POS(stack_{-c}), POS(buffer_0), ..., POS(buffer_{c-1})]$$

Note that you may need to pad the stack and buffer appropriately. You can use the pad (`[PAD]`) token with a POS tag `NULL`.

Embeddings. Next, we need to convert these discrete words and POS tags to continuous embeddings. We will be using various versions of the pre-trained GloVe embeddings for the tokens. Note that GloVe embedding are static, i.e, they should not trained. For an embedding dimension d_{emb}, you'll get:

$$w_{emb}^{2c \times d_{emb}} = GloVe(w)$$

The superscript shows the size of the tensor. GloVe embeddings are implemented in the `torchtext` library [1]. For list of tokens, you can obtain the embeddings using the `get_vecs_by_tokens` method [2].

You need to consider **two types of input representations** for token embeddings:

(a) Mean. All the embeddings of the $2c$ tokens are averaged. In this case, the size of the tensor will be $w_{emb}^{1 \times d_{emb}}$.

(b) Concatenate. All the embeddings of the $2c$ tokens are concatenated. In this case, the size of the tensor will be $w_{emb}^{1 \times 2cd_{emb}}$

We'll explain the model using the 'Mean' representation going forward.

You need to embed the POS tags into continuous space as well. You need to use a trainable embedding to map POS tags to d_{pos} sized embeddings. You can use the `torch.nn.Embedding` to achieve these embeddings. Hence, we can obtain POS embeddings (p_{emb}) by:

$$p_{emb}^{2c \times d_{pos}} = E_{pos}(p)$$

As with tokens, you can take the mean or concatenate the POS embeddings. We will consider the mean for the sake of explanation.

[1] `https://pytorch.org/text/stable/vocab.html#torchtext.vocab.GloVe`

[2] `https://pytorch.org/text/stable/vocab.html#torchtext.vocab.Vectors.get_vecs_by_tokens`

Linear Layer. Next, we pass the embeddings through a linear layer in the following fashion:

$$h_{rep}^{1\times h} = ReLU(w_{emb}^{1\times d_{emb}}.W_{tok}^{d_{emb}\times h} + p_{emb}^{1\times d_{pos}}.W_{pos}^{d_{pos}\times h} + b) \tag{1}$$

where W_{tok} and W_{pos} are trainable weight matrices, and h is the hidden dimension.

Output. The hidden representation is then passed through another linear layer and softmax to obtain the distribution over the actions.

$$y^{1\times|T|} = softmax(h_{rep}^{1\times h}.W_{out}^{h\times|T|} + b_{out}) \tag{2}$$

where T is the set of actions.

2.4.2 Parameters

Here is a list of parameters and their values which

1. c=2. Top-c elements from stack and buffer should be considered for representation.
2. d$_{emb}$ = {50, 300}. You need to report results on four GloVE embeddings: {"glove.6B.50d", "glove.6B.300d", "glove.42B.300d","glove.840B.300d"}.
3. d$_{pos}$ = 50. Embedding dimension for POS tags.
4. h = 200. The hidden dimension.
5. |T| = 75. Number of distinct actions.
6. |P| = 18. Number of distinct POS tags.

2.4.3 Hyper-Parameters

Just like the previous mini project, the only tunable hyper-parameter is the learning rate. Train your model using the following learning rates: {0.01, 0.001, 0.0001}. Run the training loop for a maximum of 20 epochs. The best model across learning rate and epochs is the one that gets the highest dev LAS metric. Feel free to choose the batch size. **Please fix the random seed in your implementation.**

2.4.4 Notes on Parser Implementation

Here are a few things to keep in mind when you implement the parser:

1. Note that our final state condition that indicates the completion of our parsing process is slightly different than the convention. Conventionally, you check if the stack and buffer are empty and that marks the end of the parsing process. In our case, the final state will have one word on the stack and an empty buffer.

2. Following up on the previous point, the one word which remains on the stack at the final state will be the root word of the sentence. Note that we have not given any `REDUCE` action with the 'root' dependency label. This is due to the fact that, generally, reducing with a root relation is deterministic. NOTE: Do not add the `REDUCE_R_root` action in 'results.txt' as the final action. The `compute_metrics` method in 'evaluate.py' adds it during evaluation.

3. You need to handle cases when an illegal action is predicted. For example, you cannot `REDUCE` when there is only one word on the stack.

2.4.5 Benchmark Information

A training epoch roughly takes 30 seconds with a batch size of 64 on a Titan X machine. It requires roughly 8 GB of CPU RAM. You can specify it using the `--mem` flag in the Slurm script. This requires a maximum of 2GB GPU RAM. You can ask for a GPU by setting the `--gres=gpu:1` flag.

3 What to Report

1. [32 points] Implement the model and the parser. Submit the 'results.txt' on Canvas as specified above.

2. [32 points] Provide the results of your parser on the test split by filling up Table 1. Note that the model to compute these results on should be the model corresponding to the best learning rate and epoch as judged with respect to the dev split. The best learning rate and epoch should be determined by the LAS metric.

	Mean		**Concatenate**	
Embedding	**UAS**	**LAS**	**UAS**	**LAS**
GloVe 6B 50d				
GloVe 6B 300d				
GloVe 42B 300d				
GloVe 840B 300d				

Table 1: Experiment Results

3. [6 points] Summarize the trends that you observe in your results in a few sentences.

4. [15 points] Provide the dependency parse trees for the three sentences below. Use the best model from Table 1 to compute the parse.[3]

   (a) Mary had a little lamb . (POS tags: PROPN AUX DET ADJ NOUN PUNCT)

[3]You can use your favorite drawing tool (e.g., draw.io). There are more sophisticated tools for drawing dependency parses like UD Annotatrix for the curious.

(b) I ate the fish raw . (POS tags: PRON VERB DET NOUN ADJ PUNCT)

(c) With neural networks , I love solving problems . (POS tags: ADP ADJ NOUN PUNCT PRON VERB VERB NOUN PUNCT)

5. [15 points] Read Chen and Manning [2014]'s paper. We mentioned that we are using a simpler representation of the parse state as compared to the original paper. What are the differences between our representation and the representation used in the paper? Describe briefly.

4 Extra Credit: Using Dependency Labels in State Representations [20 points]

Dependency labels can be useful features when it comes to deciding actions. As an example, it is likely that a 'det' relation does not repeat for a particular word if it already has an existing 'det' relation in the dependencies set. For the extra credit, you need to find the leftmost and rightmost child of the top-2 elements of the stack. Include their respective embeddings in w_{emb}. Furthermore, extract the dependency labels corresponding to these child tokens. Construct an embedding for the labels with dimension $d_{lab} = 50$ which gives label embeddings l_{emb}. Consequently, you will require a separate weight matrix for the dependency labels, say, W_{lab}. Therfore, equation 3 will change to:

$$h_{rep}^{1\times h} = ReLU(w_{emb}^{1\times d_{emb}}.W_{tok}^{d_{emb}\times h} + p_{emb}^{1\times d_{pos}}.W_{pos}^{d_{pos}\times h} + l_{emb}^{1\times d_{lab}}.W_{lab}^{d_{lab}\times h} + b) \tag{3}$$

Report results for any one GloVe embedding and input representation combination (i.e., Mean or Concatenate) for the above mentioned state representation.

References

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