XCS224N Assignment #3: Dependency Parsing

In this assignment, you will build a neural dependency parser using PyTorch. You will implement and train the dependency parser.

1. Neural Transition-Based Dependency Parsing (21 points)

In this section, you'll be implementing a neural-network based dependency parser, with the goal of maximizing performance on the UAS (Unlabeled Attachemnt Score) metric.

Before you begin please install PyTorch 1.0.0 or above from `https://pytorch.org/get-started/locally/` with the `CUDA` option set to `None`. Additionally run `pip install tqdm` to install the tqdm package – which produces progress bar visualizations throughout your training process.

A dependency parser analyzes the grammatical structure of a sentence, establishing relationships between *head* words, and words which modify those heads. Your implementation will be a *transition-based* parser, which incrementally builds up a parse one step at a time. At every step it maintains a *partial parse*, which is represented as follows:

- A *stack* of words that are currently being processed.
- A *buffer* of words yet to be processed.
- A list of *dependencies* predicted by the parser.

Initially, the stack only contains ROOT, the dependencies list is empty, and the buffer contains all words of the sentence in order. At each step, the parser applies a *transition* to the partial parse until its buffer is empty and the stack size is 1. The following transitions can be applied:

- `SHIFT`: removes the first word from the buffer and pushes it onto the stack.
- `LEFT-ARC`: marks the second (second most recently added) item on the stack as a dependent of the first item and removes the second item from the stack.
- `RIGHT-ARC`: marks the first (most recently added) item on the stack as a dependent of the second item and removes the first item from the stack.

On each step, your parser will decide among the three transitions using a neural network classifier.

*Note: **Please** do not use external python modules which are not specified in the `requirements.txt` file. This will cause the `autograder` to fail.*

(a) (6 points) Implement the `__init__` and `parse_step` functions in the `PartialParse` class in `parser_transitions.py`. This implements the transition mechanics your parser will use. You can run basic (non-exhaustive) tests by running `python parser_transitions.py`.

Note: You will find the `parser_transitions.py` file inside the `utils` folder

(b) (6 points) Our network will predict which transition should be applied next to a partial parse. We could use it to parse a single sentence by applying predicted transitions until the parse is complete. However, neural networks run much more efficiently when making predictions about *batches* of data at a time (i.e., predicting the next transition for any different partial parses simultaneously). We can parse sentences in minibatches with the following algorithm.

Implement this algorithm in the `minibatch_parse` function in `parser_transitions.py`. You can run basic (non-exhaustive) tests by running `python parser_transitions.py`.

Note: You will need `minibatch_parse` to be correctly implemented to evaluate the model you will build in part (c). However, you do not need it to train the model, so you should be able to complete

Algorithm 1 Minibatch Dependency Parsing

Input: `sentences`, a list of sentences to be parsed and `model`, our model that makes parse decisions

Initialize `partial_parses` as a list of PartialParses, one for each sentence in `sentences`
Initialize `unfinished_parses` as a shallow copy of `partial_parses`
while `unfinished_parses` is not empty **do**
  Take the first `batch_size` parses in `unfinished_parses` as a minibatch
  Use the `model` to predict the next transition for each partial parse in the minibatch
  Perform a parse step on each partial parse in the minibatch with its predicted transition
  Remove the completed (empty buffer and stack of size 1) parses from `unfinished_parses`
end while

Return: The `dependencies` for each (now completed) parse in `partial_parses`.

most of part (c) even if `minibatch_parse` is not implemented yet.

We are now going to train a neural network to predict, given the state of the stack, buffer, and dependencies, which transition should be applied next. First, the model extracts a feature vector representing the current state. We will be using the feature set presented in the original neural dependency parsing paper: *A Fast and Accurate Dependency Parser using Neural Networks.*[1] The function extracting these features has been implemented for you in `utils/parser_utils.py`. This feature vector consists of a list of tokens (e.g., the last word in the stack, first word in the buffer, dependent of the second-to-last word in the stack if there is one, etc.). They can be represented as a list of integers $[w_1, w_2, \ldots, w_m]$ where m is the number of features and each $0 \leq w_i < |V|$ is the index of a token in the vocabulary ($|V|$ is the vocabulary size). First our network looks up an embedding for each word and concatenates them into a single input vector:

$$\mathbf{x} = [\mathbf{E}_{w_1}, ..., \mathbf{E}_{w_m}] \in \mathbb{R}^{dm}$$

where $\mathbf{E} \in \mathbb{R}^{|V| \times d}$ is an embedding matrix with each row $\mathbf{E}_w$ as the vector for a particular word w. We then compute our prediction as:

$$\begin{aligned}
\mathbf{h} &= \mathrm{ReLU}(\mathbf{xW} + \mathbf{b}_1) \\
\mathbf{l} &= \mathbf{hU} + \mathbf{b}_2 \\
\hat{\mathbf{y}} &= \mathrm{softmax}(l)
\end{aligned}$$

where $\mathbf{h}$ is referred to as the hidden layer, $\mathbf{l}$ is referred to as the logits, $\hat{\mathbf{y}}$ is referred to as the predictions, and $\mathrm{ReLU}(z) = \max(z, 0)$). We will train the model to minimize cross-entropy loss:

$$J(\theta) = CE(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{3} y_i \log \hat{y}_i$$

To compute the loss for the training set, we average this $J(\theta)$ across all training examples.

(c) (9 points) In `parser_model.py` you will find skeleton code to implement this simple neural network using PyTorch. Complete the `__init__`, `embedding_lookup` and `forward` functions to implement the model. Then complete the `train_for_epoch` function within the `run.py` file.

[1] Chen and Manning, 2014, `https://nlp.stanford.edu/pubs/emnlp2014-depparser.pdf`

Finally execute `python run.py` to train your model and compute predictions on test data from Penn Treebank (annotated with Universal Dependencies). Make sure to turn off debug setting by setting `debug=False` in the `main` function of `run.py`.

Hints:

- When debugging, set `debug=True` in the `main` function of `run.py`. This will cause the code to run over a small subset of the data, so that training the model won't take as long. Make sure to set `debug=False` to run the full model once you are done debugging.
- When running with `debug=True`, you should be able to get a loss smaller than 0.2 and a UAS larger than 65 on the dev set (although in rare cases your results may be lower, there is some randomness when training).
- It should take about **1 hour** to train the model on the entire the training dataset, i.e., when `debug=False`.
- When running with `debug=False`, you should be able to get a loss smaller than 0.08 on the train set and an Unlabeled Attachment Score larger than 87 on the dev set. For comparison, the model in the original neural dependency parsing paper gets 92.5 UAS. If you want, you can tweak the hyperparameters for your model (hidden layer size, hyperparameters for Adam, number of epochs, etc.) to improve the performance (but you are not required to do so).

Deliverables:

- Working implementation of the neural dependency parser in `parser_model.py`. (We'll look at and run this code for grading).

Submission Instructions

1. **Please** do not use external python modules which are not specified in the `requirements.txt` file. This will cause the `autograder` to fail.
2. Run the `collect_submission.sh` script to produce your `assignment3.zip` file.
3. Upload your `assignment3.zip` file via the Gradescope link in the Assignment 3 block of your SCPD learning portal