CS114 (Spring 2020) Programming Assignment 6
Neural Transition-Based Dependency Parsing*

Due May 13, 2020

Introduction

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

Note: You are **not** allowed to use any specialized neural network or deep learning libraries, including, but not limited to, CNTK, Keras, MXNet, PyTorch, TensorFlow, Theano, etc. If you have any questions about what libraries you are allowed to use, please ask. It is possible to do the assignment using only those packages imported for you in the starter code.

Background

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.

---

*This assignment is adapted from the CS 224N course at Stanford.

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, adding a *first_word → second_word* dependency to the dependency list.

- 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, adding a *second_word → first_word* dependency to the dependency list.

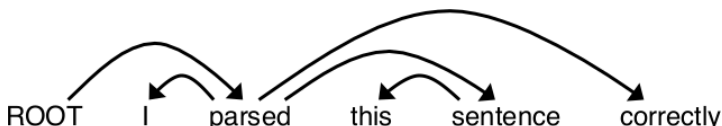On each step, your parser will decide among the three transitions using a neural network classifier.

Assignment

(a) Go through the sequence of transitions needed for parsing the sentence "*I parsed this sentence correctly*". The dependency tree for the sentence is shown below. At each step, give the configuration of the stack and buffer, as well as what transition was applied this step and what new dependency was added (if any). The first three steps are provided below as an example.

ROOT   I   parsed   this   sentence   correctly

Stack	Buffer	New dependency	Transition
[ROOT]	[I, parsed, this, sentence, correctly]		Initial Configuration
[ROOT, I]	[parsed, this, sentence, correctly]		SHIFT
[ROOT, I, parsed]	[this, sentence, correctly]		SHIFT
[ROOT, parsed]	[this, sentence, correctly]	parsed → I	LEFT-ARC

(b) A sentence containing n words will be parsed in how many steps (in terms of n)? Briefly explain why.

2

(c) 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 part_c`.

(d) 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.

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 `PartialParse`s, 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`.

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 part_d`.

Note: You will need `minibatch_parse` to be correctly implemented to evaluate the model you will build in part (e). However, you do not need it to train the model, so you should be able to complete most of part (e) even if `minibatch_parse` is not implemented yet.

(e) 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 $\mathbf{w} = [w_1, w_2, ..., w_m]$ where m is the number of features and each $0 \leq w_i \leq |V|$ is the index of a token in the vocabulary ($|V|$ is the vocabulary size). Then 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 E_w as the vector for a particular word w. We then compute our prediction as:

$$\mathbf{h}_1 = \text{ReLU}(\mathbf{x}\mathbf{W_1} + \mathbf{b}_1)$$
$$\mathbf{h}_2 = \text{ReLU}(\mathbf{h_1}\mathbf{W_2} + \mathbf{b}_2)$$
$$\mathbf{y} = \text{softmax}(\mathbf{h_2}\mathbf{U} + \mathbf{b}_3)$$

where $\mathbf{h}_1$ and $\mathbf{h}_2$ are referred to as the hidden layers, $\hat{\mathbf{y}}$ is referred to as the predictions, and $\text{ReLU}(z) = \max(z, 0)$). 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.

We will use UAS score as our evaluation metric. UAS refers to Unlabeled Attachment Score, which is computed as the ratio between the number of correctly predicted dependencies and the number of total dependencies, ignoring the labels (our model doesn't predict these).

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

In `parser_model.py` you will find skeleton code to implement this simple neural network using Numpy. Complete the `relu`, `__init__`, `embedding_lookup`, and `forward` functions to implement the model. Then complete the `d_relu` and `train_for_epoch` functions within the `run.py` file.

Finally execute `python run.py` to train your model and compute predictions on test data from Penn Treebank (annotated with Universal Dependencies).

Hints:

- Once you have implemented `embedding_lookup` (`e`) or `forward` (`f`) you can call `python parser_model.py` with flag `-e` or `-f` or both to run sanity checks with each function. These sanity checks are fairly basic and passing them doesn't mean your code is bug free.

- When debugging, you can add a debug flag: `python run.py -d`. 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 remove the `-d` flag to run the full model once you are done debugging.

- When running with debug mode, you should be able to get a loss smaller than 0.24 and a UAS larger than 65 on the dev set (although in some releases your results may be lower; there is some randomness when training).

- It took about **12 minutes** (using a 2.5 GHz quad-core processor) to train the model on the entire training dataset, i.e., when debug mode is disabled. Your mileage may vary, depending on your computer. That being said, if your model takes hours rather than minutes to train, your code is likely not as efficient as it can be. Make sure your code is fully broadcasted!

- When debug mode is disabled, you should be able to get a loss smaller than 0.09 on the train set and an Unlabeled Attachment Score larger than 85 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, learning rate, number of epochs, etc.) to improve the performance (but you are not required to do so).

(f) We'd like to look at example dependency parses and understand where parsers like ours might be wrong. For example, in this sentence:
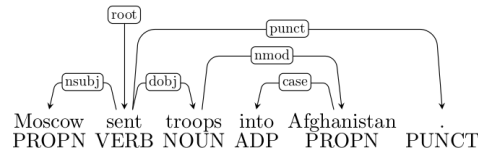
the dependency of the phrase *into Afghanistan* is wrong, because the phrase should modify *sent* (as in *sent into Afghanistan*) not *troops* (because *troops into Afghanistan* doesn't make sense). Here is the correct parse:
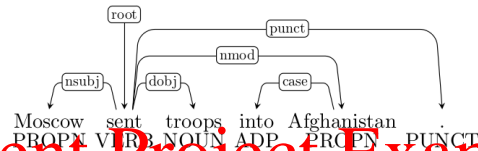
More generally, here are four types of parsing error:

- **Prepositional Phrase Attachment Error**: In the example above, the phrase *into Afghanistan* is a prepositional phrase. A Prepositional Phrase Attachment Error is when a prepositional phrase is attached to the wrong head word (in this example, *troops* is the wrong head word and *sent* is the correct head word). More examples of prepositional phrases include *with a rock*, *before midnight*, and *under the carpet*.

- **Verb Phrase Attachment Error**: In the sentence *Leaving the store unattended, I went outside to watch the parade*, the phrase *leaving the store unattended* is a verb phrase. A Verb Phrase Attachment Error is when a verb phrase is attached to the wrong head word (in this example, the correct head word is *went*).

- **Modifier Attachment Error**: In the sentence *I am extremely short*, the adverb *extremely* is a modifier of the adjective *short*. A Modifier Attachment Error is when a modifier is attached to the wrong head word (in this example, the correct head word is *short*).

6

- **Coordination Attachment Error**: In the sentence *Would you like brown rice or garlic naan?*, the phrases *brown rice* and *garlic naan* are both conjuncts and the word *or* is the coordinating conjunction. The second conjunct (here *garlic naan*) should be attached to the first conjunct (here *brown rice*). A Coordination Attachment Error is when the second conjunct is attached to the wrong head word (in this example, the correct head word is *rice*). Other coordinating conjunctions include *and*, *but*, and *so*.

In this question are four sentences with dependency parses obtained from a parser. Each sentence has one error, and there is one example of each of the four types above. For each sentence, state the type of error, the incorrect dependency, and the correct dependency. To demonstrate: for the example above, you would write:
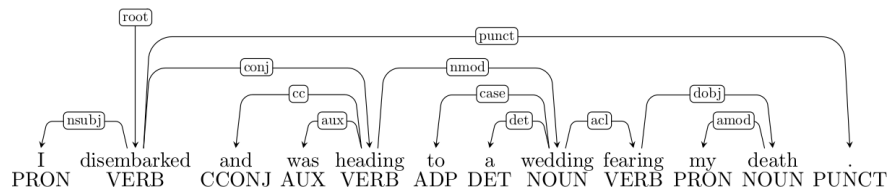
- **Error type**: Prepositional Phrase Attachment Error
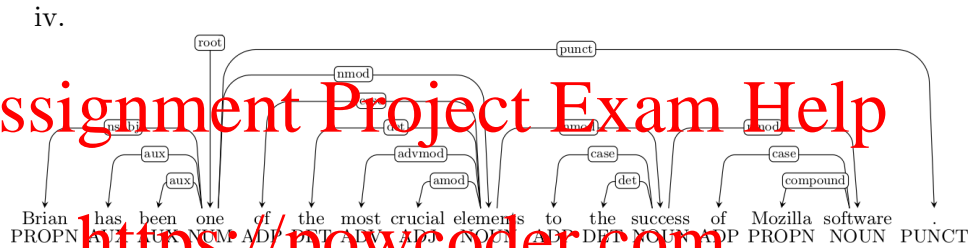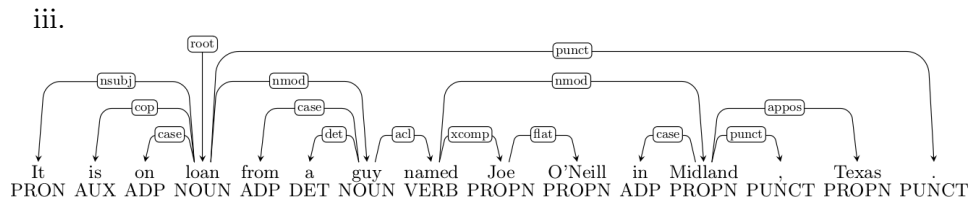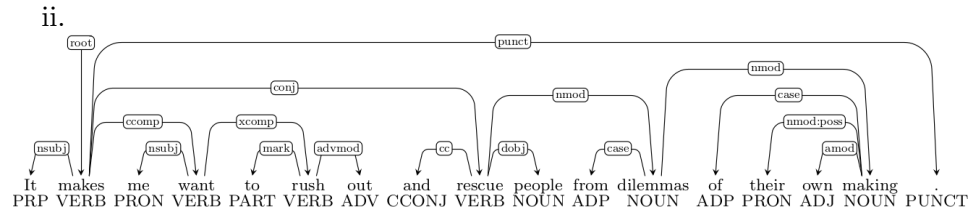- **Incorrect dependency**: troops → Afghanistan
- **Correct dependency**: sent → Afghanistan

Note: There are lots of details and conventions for dependency annotation. If you want to learn more about them, you can look at the UD website: `http://universaldependencies.org`[2] or the short introductory slides at: `http://people.cs.georgetown.edu/nschneid/p/UD-for-English.pdf`. However, you **do not** need to know all these details in order to do this question. In each of these cases, we are asking about the attachment of phrases and it should be sufficient to see if they are modifying the correct head. In particular, you **do not** need to look at the labels on the the dependency edges—it suffices to just look at the edges themselves.

i.

---

[2]But note that in the assignment we are actually using UDv1, see: `http://universaldependencies.org/docsv1/`

ii.

It makes me want to rush out and rescue people from dilemmas of their own making .
PRP VERB PRON VERB PART VERB ADV CCONJ VERB NOUN ADP NOUN ADP PRON ADJ NOUN PUNCT

iii.

It is on loan from a guy named Joe O'Neill in Midland , Texas .
PRON AUX ADP NOUN ADP DET NOUN VERB PROPN PROPN ADP PROPN PUNCT PROPN PUNCT

iv.

Brian has been one of the most crucial elements to the success of Mozilla software .
PROPN AUX AUX NUM ADP DET ADV ADJ NOUN ADP DET NOUN ADP PROPN NOUN PUNCT

Write-up

You should also prepare a short write-up that includes at least the following:

- The answers to the questions in parts (a), (b), and (f).

- The best UAS your model achieves on the dev set and the UAS it achieves on the test set.

- **Important**: If you are graduating this semester, please make a note of this, so we know to grade your assignments first!

Submission Instructions

Please submit four files: your write-up (in PDF format), `parser_model.py`, `parser_transitions.py`, and `run.py`. You don't need to include any data or any of the other code that was provided to you.