CS542 (Fall 2023) Programming Assignment 4 Neural Transition-Based Dependency Parsing[*]

Due November 18, 2023

Introduction

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

Background

A dependency parser analyzes the grammatical structure of a sentence, establishing relationships between *head* words, and words which modify those heads. For example, if an adjective modifies a noun, the noun is the head and the adjective is the *dependent*. The relationship between these is the *dependency*. 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:

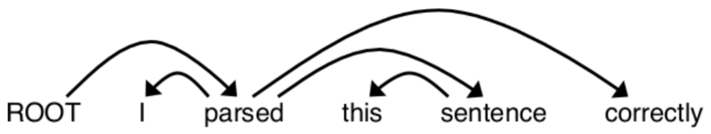
- A *stack* of words that are currently being processed.
- A *buffer* of words yet to be processed.
- A list of *dependencies* predicted by the parser.

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:

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

- 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* $\rightarrow$ *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* $\rightarrow$ *first_word* dependency to the dependency list.

For example, the sequence of transitions needed for parsing the sentence "*I parsed this sentence correctly*" is shown below, with the dependency tree. Each step shows the configuration of the stack and buffer, as well as what transition was applied this step and what new dependency was added (if any).



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 $\rightarrow$ I	LEFT-ARC
[ROOT, parsed, this]	[sentence, correctly]	•	SHIFT
[ROOT, parsed, this, sentence]	[correctly]	•	SHIFT
[ROOT, parsed, sentence]	[correctly]	sentence $\rightarrow$ this	LEFT-ARC
[ROOT, parsed]	[correctly]	parsed $\rightarrow$ sentence	RIGHT-ARC
[ROOT, parsed, correctly]	[]	•	SHIFT
[ROOT, parsed]	[]	parsed $\rightarrow$ correctly	RIGHT-ARC
[ROOT]	[]	ROOT $\rightarrow$ parsed	RIGHT-ARC

A sentence containing n words will be parsed in $2n$ steps. For each word, two transitions are needed: one to push it from the buffer onto the stack (SHIFT), and one to pop it off the stack (either LEFT-ARC or RIGHT-ARC).

1 Assignment

On each step, your parser will decide among the three transitions using a neural network classifier. The functions you need to implement are described below. For each function, the template code provides step-by-step instructions and the approximate number of lines of code you need to write. Therefore if you find yourself writing way more code than is indicated, stop and reconsider how you are approaching the problem.

(a) Complete the `parse_step` function in the `PartialParse` class in `parser_transitions.py` by filling in the `if` statements according to the description of each transition above. This implements the transition mechanics your parser will use. You can run basic (non-exhaustive) tests by running `python parser_transitions.py part_a`. These same tests are run by the automated grader.

(b) Your network will predict which transition should be applied next to a partial parse. You 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 1.

```
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.
```

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_b`. These same tests are run by the automated grader.

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

(c) You 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:

$$\begin{aligned}
\mathbf{h} &= \mathrm{ReLU}(\mathbf{x} \cdot \mathbf{\Theta_1}) \\
\mathbf{l} &= \mathbf{h} \cdot \mathbf{\Theta_2} \\
\hat{\mathbf{y}} &= \mathrm{softmax}(\mathbf{l})
\end{aligned}$$

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.

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 PyTorch. Complete the `embedding_lookup` and `forward` functions to implement the model. Then complete the `train_for_epoch` and `train` functions within the `run.py` file. If you are not familiar with PyTorch, links to the PyTorch docs are provided in code comments with examples of how to do this.

Finally execute `python run.py` (*without* `-d`) to train your model and compute predictions on test data from Penn Treebank (annotated with Universal Dependencies).

Note: Please **do not** use `torch.nn.Linear` or `torch.nn.Embedding` modules in your code. You will be effectively implementing these yourself in `parser_model.py`.

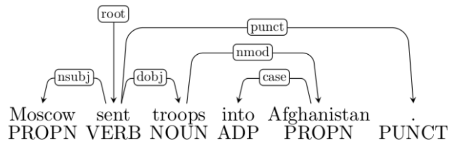
Hints:

- 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 on the test data. Be sure to test on more than the toy data as shown below!
- When debugging on non-toy data, 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.
- It took about **18 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 0.03 on the train set and an Unlabeled Attachment Score of at least **88** 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. You are not required to do so, but extra

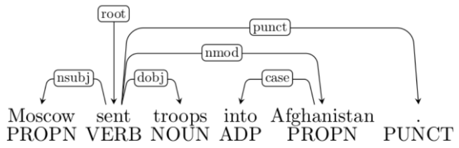
credit may be assigned for performance on the dev or test sets reliably above 88 UAS.

1.1 Analyzing Parse Errors

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* isn't a verb in this sentence. 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* (or *like an arrow* from HW3).

- **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*).

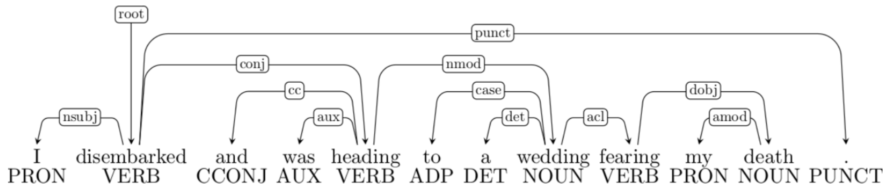
- **Coordination Attachment Error**: In the sentence *Would you like brown rice or garlic naan?*, the phrases *brown rice* and *garlic naan* are both conjuncts and the word *or* is the coordinating conjunction. The second conjunct (here *garlic naan*) should be attached to the first conjunct (here *brown 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

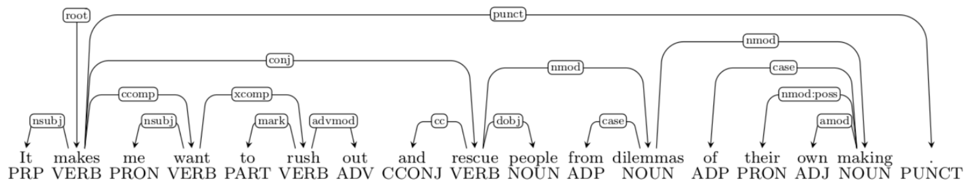
Note: There are lots of details and conventions for dependency annotation. If you want to learn more about them, you can look at the UD website: `http://universaldependencies.org`[2] or the short introductory slides at: `http://people.cs.georgetown.edu/nschneid/p/UD-for-English.pdf`. However, you **do 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.

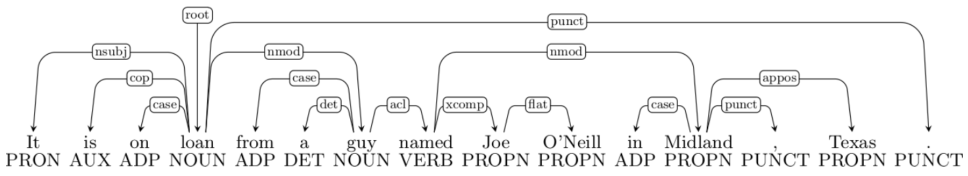


[2]But note that in the assignment we are actually using UDv1, see: `http://universaldependencies.org/docsv1/`

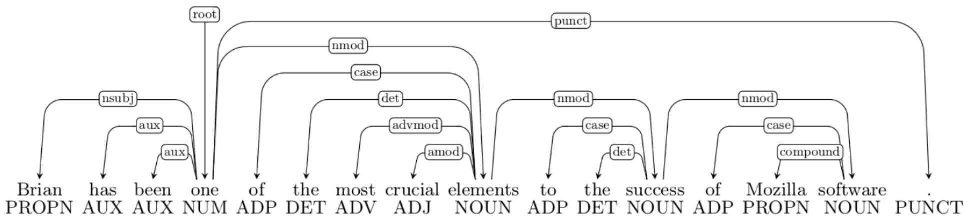
ii.


root
punct
conj
nmod
nmod
case
ccomp
xcomp
nmod:poss
nsubj
nsubj
mark
advmod
cc
dobj
case
amod
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.


root
punct
nsubj
nmod
nmod
cop
case
appos
case
det
acl
xcomp
flat
case
punct
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.


root
punct
nmod
case
nsubj
det
nmod
nmod
aux
advmod
case
case
aux
amod
det
compound
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

Data and Resources

You are given three template code files: `parser_transitions.py`, `parser_model.py`, and `run.py`. You should make changes in the functions mentioned above, in the designated places. Doing otherwise is buying yourself a headache. The sole exception to this is optionally changing hyperparameters to increase performance on the dev and test sets, which can be done in *main* in `run.py`. You also have `pa5_grader.py` so you can continually test your code.

The `utils` folder contains utility classes that you should not modify.

`data` contains the following:

- `en-cw.txt`: the pretrained word vectors.
- `train.conll`: the training data.
- `dev.conll`: the dev-test data to report results on.

- `test.conll`: the final test data to report results on.

{`train,dev,test`}`.conll` are just plain `.txt` files minus the extension that you can open in a text editor if you want to examine the data. Since loading and preprocessing data is already done for you by the provided utility classes, you can complete this assignment without ever opening the data file, but you may want to for your own interest.

Submission

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

- The answers to the questions in Section 1.1.
- The best UAS your model achieves on the dev set and the UAS it achieves on the test set. If you changed any of the preset hyperparameters to increase performance, note your changes here.

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.

Grading

Grades will be determined as follows:

- The automated grader is provided to you run the full battery of sanity checks on your code. These tests are non-exhaustive, so just because your code passes these sanity checks doesn't mean it is bug free or fully correct. Therefore simply passing the autograder is not enough! There are 6 sanity checks run though the autograder that apportion points as outlined below:
    1. **5 points** for loading your code error-free. If you do not change the template code outside of the prescribed areas, this should be no problem.
    2. **10 points** for passing all 3 sanity checks on `test_parse_step`.
    3. **10 points** for passing the sanity check on `test_parse`.
    4. **10 points** for passing all 3 sanity checks on `test_minibatch_parse`.
    5. **5 points** for passing the `embeddings_lookup` sanity check.

6. **5 points** for passing the `forward` sanity check.

- You will receive **up to 20 points** for the final test in the autograder, the full debug run. This tests that `train` and `train_for_epoch` were implemented correctly in `run.py`, and also tests the correctness of the rest of your implementation in the process. A fully-correct implementation will receive a UAS above 70 on the reduced dev data. Therefore at the end of the run a *performance penalty* PP will be assessed for UAS under 70.

  $$PP = \begin{cases} (.7 - UAS) * 20, \text{if } UAS < .7 \\ 0, \text{otherwise} \end{cases}$$

  (UAS is calculated as a decimal within the code). PP will be subtracted from the 20 points for this test.

- **20 points (5 for each answer)** for the questions in Section 1.1. Note that these parses don't necessarily come from the assignment dataset; these answers are to show you can analyze specific known examples.

- The final **15 points** come from reporting your model's best results on the dev and test sets. 7.5 points will be deducted for each score you do not report. If you optionally change hyperparameters to improve performance, report those changes here. Extra credit may be assigned if your UAS on the dev or tests sets is reliably above 88 after tuning hyperparameters.