

# XCS224N Assignment 3 Dependency Parsing

---

**Due Sunday, March 30 at 11:59pm PT.**

## Guidelines

1. If you have a question about this homework, we encourage you to post your question on our Slack channel, at <http://xcs224n-scpd.slack.com/>
2. Familiarize yourself with the collaboration and honor code policy before starting work.
3. For the coding problems, you must use the packages specified in the provided environment description. Since the autograder uses this environment, we will not be able to grade any submissions which import unexpected libraries.

## Submission Instructions

**Coding Submission:** Some questions in this assignment require a coding response. For these questions, you should submit **all files indicated in the question** to the online student portal. For further details, see Writing Code and Running the Autograder below.

## Honor code

We strongly encourage students to form study groups. Students may discuss and work on homework problems in groups. However, each student must write down the solutions independently, and without referring to written notes from the joint session. In other words, each student must understand the solution well enough in order to reconstruct it by him/herself. In addition, each student should write on the problem set the set of people with whom s/he collaborated. Further, because we occasionally reuse problem set questions from previous years, we expect students not to copy, refer to, or look at the solutions in preparing their answers. It is an honor code violation to intentionally refer to a previous year's solutions. More information regarding the Stanford honor code can be found at <https://communitystandards.stanford.edu/policies-and-guidance/honor-code>.

## Writing Code and Running the Autograder

All your code should be entered into the `src/submission/` directory. When editing files in `src/submission/`, please only make changes between the lines containing `### START_CODE_HERE ###` and `### END_CODE_HERE ###`. Do not make changes to files outside the `src/submission/` directory.

The unit tests in `src/grader.py` (the autograder) will be used to verify a correct submission. Run the autograder locally using the following terminal command within the `src/` subdirectory:

```
$ python grader.py
```

There are two types of unit tests used by the autograder:

- **basic:** These tests are provided to make sure that your inputs and outputs are on the right track, and that the hidden evaluation tests will be able to execute.
- **hidden:** These unit tests are the evaluated elements of the assignment, and run your code with more complex inputs and corner cases. Just because your code passed the basic local tests does not necessarily mean that they will pass all of the hidden tests. These evaluative hidden tests will be run when you submit your code to the Gradescope autograder via the online student portal, and will provide feedback on how many points you have earned.

For debugging purposes, you can run a single unit test locally. For example, you can run the test case `3a-0-basic` using the following terminal command within the `src/` subdirectory:

```
$ python grader.py 3a-0-basic
```

Before beginning this course, please walk through the [Anaconda Setup for XCS Courses](#) to familiarize yourself with the coding environment. Use the env defined in `src/environment.yml` to run your code. This is the same environment used by the online autograder.

## Test Cases

The autograder is a thin wrapper over the python `unittest` framework. It can be run either locally (on your computer) or remotely (on SCPD servers). The following description demonstrates what test results will look like for both local and remote execution. For the sake of example, we will consider two generic tests: `1a-0-basic` and `1a-1-hidden`.

### Local Execution - Hidden Tests

All hidden tests rely on files that are not provided to students. Therefore, the tests can only be run remotely. When a hidden test like `1a-1-hidden` is executed locally, it will produce the following result:

```
----- START 1a-1-hidden: Test multiple instances of the same word in a sentence.
----- END 1a-1-hidden [took 0:00:00.011989 (max allowed 1 seconds), ???/3 points] (hidden test ungraded)
```

### Local Execution - Basic Tests

When a basic test like `1a-0-basic` passes locally, the autograder will indicate success:

```
----- START 1a-0-basic: Basic test case.
----- END 1a-0-basic [took 0:00:00.000062 (max allowed 1 seconds), 2/2 points]
```

When a basic test like `1a-0-basic` fails locally, the error is printed to the terminal, along with a stack trace indicating where the error occurred:

```
----- START 1a-0-basic: Basic test case.
<class 'AssertionError': {'a': 2, 'b': 1} != None <----- This error caused the test to fail.
File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/grader.py", line 23, in test_0 <----- In this case, start your debugging
    submission.extractWordFeatures("a b a") <----- in line 23 of grader.py.
File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
----- END 1a-0-basic [took 0:00:00.003809 (max allowed 1 seconds), 0/2 points]
```

## Remote Execution

Basic and hidden tests are treated the same by the remote autograder. Here are screenshots of failed basic and hidden tests. Notice that the same information (error and stack trace) is provided as the in local autograder, now for both basic and hidden tests.

### 1a-0-basic) Basic test case. (0.0/2.0)

```
<class 'AssertionError': {'a': 2, 'b': 1} != None <----- Just like in the local autograder, this error caused the test to fail.
File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
File "/autograder/source/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
File "/autograder/source/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
File "/autograder/source/grader.py", line 23, in test_0 <----- Just like in the local autograder, start your
    submission.extractWordFeatures("a b a") <----- debugging in line 23 of grader.py.
File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
```

## 1a-1-hidden) Test multiple instances of the same word in a sentence. (0.0/3.0)

```
<class 'AssertionError': { 'a': 23, 'ab': 22, 'aa': 24, 'c': 16, 'b': 15} != None
File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
File "/autograder/source/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
File "/autograder/source/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
File "/autograder/source/grader.py", line 31, in test_1
    self.compare_with_solution_or_wait(submission, 'extractWordFeatures', lambda f: f(sentence))
File "/autograder/source/graderUtil.py", line 183, in compare_with_solution_or_wait
    self.assertEqual(ans1, ans2)
File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEquals
    assertion_func(first, second, msg=msg)
File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
```

This error caused the test to fail.

Start your debugging in line 31 of grader.py.

Finally, here is what it looks like when basic and hidden tests pass in the remote autograder.

## 1a-0-basic) Basic test case. (2.0/2.0)

## 1a-1-hidden) Test multiple instances of the same word in a sentence. (3.0/3.0)

# 1 Neural Transition-Based Dependency Parsing

In this assignment, you will build a neural dependency parser using PyTorch. You will implement and train the dependency parser. You'll be implementing a neural-network based dependency parser, with the goal of maximizing performance on the UAS (Unlabeled Attachment Score) metric.

This assignment requires PyTorch without CUDA installed. GPUs will be necessary in the next two assignments (via CUDA), but are not necessary for this assignment.

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.

- [6 points (Coding)]** Implement the `__init__` and `parse_step` functions in the `PartialParse` class in `src/submission/parser_transitions.py`. This implements the transition mechanics your parser will use.
- [6 points (Coding)]** 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 `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 `src/submission/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 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*.<sup>1</sup> The function extracting these features has been implemented for you in `src/submission/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  $[w_1, w_2, \dots, w_m]$  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:

$$\mathbf{x} = [\mathbf{E}_{\mathbf{w}_1}, \dots, \mathbf{E}_{\mathbf{w}_m}] \in \mathbb{R}^{dm}$$

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:

$$\begin{aligned}\mathbf{h} &= \text{ReLU}(\mathbf{x}\mathbf{W} + \mathbf{b}_1) \\ \mathbf{l} &= \mathbf{h}\mathbf{U} + \mathbf{b}_2 \\ \hat{\mathbf{y}} &= \text{softmax}(\mathbf{l})\end{aligned}$$

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  $\text{ReLU}(z) = \max(z, 0)$ . 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.

- (c) **[9 points (Coding)]** In `src/submission/parser_model.py` you will find skeleton code to implement this simple neural network using PyTorch. Complete the `__init__`, `reset_parameters`, `embedding_lookup` and `forward` functions to implement the model. Then complete the `train_for_epoch` function within the `src/submission/train` file.

Finally execute `python run.py` within the `src/` subdirectory 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 `src/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).

<sup>1</sup>Chen and Manning, 2014, <https://nlp.stanford.edu/pubs/emnlp2014-depparser.pdf>

## Deliverables

For this assignment, please submit all files within the `src/submission` subdirectory. Update files **without directory structure**.

This includes:

- `src/submission/__init__.py`
- `src/submission/parser_model.py`
- `src/submission/parser_transitions.py`
- `src/submission/parser_utils.py`
- `src/submission/train.py`

We provide a script `src/collect_submission.sh` to collect these files and create a zip file for submission. You can run the script on Linux/Mac/Windows(using Git Bash), then submit the zip file to the assignment.

```
cd src
bash ./collect_submission.sh
```

## 2 Quiz

**[13 points (Online)]** This remainder of this homework is a series of multiple choice questions related to the dependency parsing algorithm. Please input your answers into the Gradescope Online Assessment **A3 Online Assessment**.

This handout includes space for every question that requires a written response. Please feel free to use it to handwrite your solutions (legibly, please). If you choose to typeset your solutions, the `README.md` for this assignment includes instructions to regenerate this handout with your typeset L<sup>A</sup>T<sub>E</sub>X solutions.

---

THERE IS NO WRITTEN SUBMISSION FOR THIS ASSIGNMENT.

YOU ARE NOT REQUIRED TO SUBMIT ANYTHING.