University of Toronto, Department of Computer Science
CSC 485/2501F—Computational Linguistics, Fall 2019

Assignment 1

Due date: 23:59 on Friday, October 11, 2019, in the course drop-box in BA 2220.
Late assignments will not be accepted without a valid medical certificate or other documentation of an emergency.
This assignment is worth either 25% (CSC 2501) or 33% (CSC 485) of your final grade.

- **Read the whole assignment carefully before you begin.**
- Fill out both sides of the assignment cover sheet, and staple together all answer sheets (in order) with the cover sheet (sparse side up) on the front. **Do not turn in a copy of this handout.**
- Type your reports in no less than 12pt font; diagrams and tree structures may be drawn with software or neatly by hand.
- What you turn in must be your own work. You may not work with anyone else on any of the problems in this assignment. If you need assistance, contact the instructor or TA for the assignment.
- Any clarifications to the problems will be posted on the course bulletin board. You will be responsible for taking into account in your solutions any information that is posted there, or discussed in class, so you should check the page regularly between now and the due date.
- The starter code directory for this assignment is accessible on Teaching labs machines at the path `/h/u2/csc485h/fall/pub/deptrans/`. In this handout, code files we refer to are located in that directory.

1. Transition-Based Dependency Parsing (40 marks)

In this assignment, you'll be implementing a neural-network-based dependency parser. Dependency grammars posit relationships between "head" words and their modifiers, as you have seen in class. These relationships constitute trees, in which each word depends on exactly one parent: either another word or, for the head of the entire sentence, a dummy root symbol, ROOT. You will implement a transition-based parser that incrementally builds up a parse one step at a time. At every step, the state of the (partial) parse is represented by:

- 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.

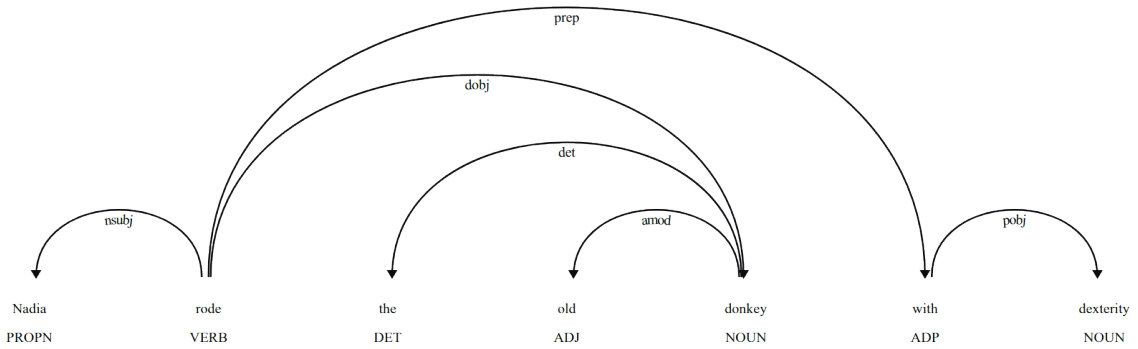
This is very much like a shift-reduce parser. Initially, the stack only contains ROOT, the dependencies lists is empty, and the buffer contains all words of the sentence in order. At each step, the parser advances by applying a "transition" to the partial parse until its buffer is empty and the stack is of size 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.

Your parser will use a neural network as a classifier that decides which transition to apply at each state. But first, you must implement the partial parse representation and transition functions.

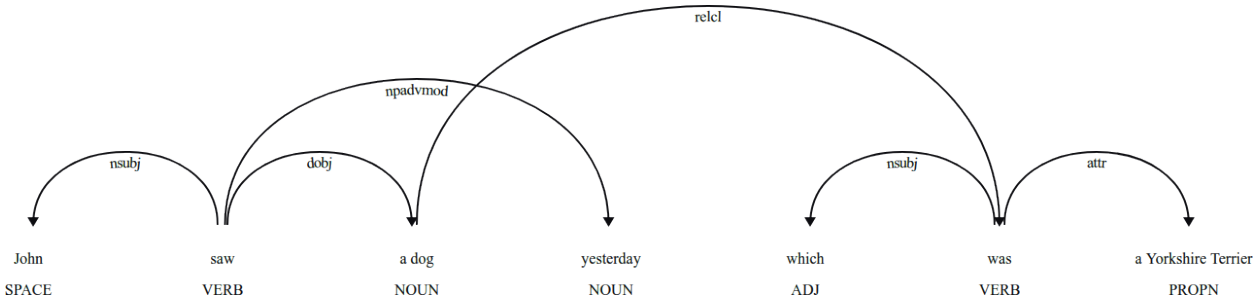
(a) (6 marks) Go through the sequence of transitions needed for parsing the sentence "Nadia rode the old donkey with dexterity." The dependency tree for the sentence is shown below (without ROOT). At each step, provide the configuration of both the stack and the buffer, as well as which transition to apply at this step, including what, if any, new dependency to add. The first three steps are provided below to get you started.

stack	buffer	new dependency	transition
[ROOT]	[Nadia, rode, the, old, donkey, with, dexterity]		Initial Config
[ROOT, Nadia]	[rode, the, old, donkey, with, dexterity]		SHIFT
[ROOT, Nadia, rode]	[the, old, donkey, with, dexterity]		SHIFT
[ROOT, rode]	[the, old, donkey, with, dexterity]	rode $\overset{nsubj}{\rightarrow}$ Nadia	LEFT-ARC



(b) (2 marks) A sentence containing n words will be parsed in how many steps, in terms of n? (Exact, not asymptotic.) Briefly explain why.

(c) (4 marks) A *projective dependency tree* is one in which the edges can be drawn above the words without crossing other edges when the words, preceded by ROOT, are arranged in linear order. Equivalently, every word forms a contiguous substring of the sentence when taken together with its descendants. The above figure was projective. The figure below is not projective.



Why is the parsing mechanism described above insufficient to generate non-projective dependency trees?

Fortunately, most ($\sim$99%) of the sentences in our data set have projective dependency trees.

(d) (7 marks) Implement the `complete` and `parse_step` methods in the `PartialParse` class in `parser.py`. These implement the transition mechanism of your parser. Also implement `get_n_rightmost_deps` and `get_n_leftmost_deps`. You can run basic (non-exhaustive) tests by running `python3 parser.py`.

(e) (6 marks) Our network will predict which transition should be applied next to a partial parse. In principle, we could use the network to parse a single sentence simply by applying predicted transitions until the parse is complete. However, neural networks run much more efficiently when making predictions about "minibatches" of data at a time; in this case, that means predicting the next transition for many different partial parses simultaneously. We

can parse sentences in minibatches with the following algorithm:

Algorithm 1: Minibatch Dependency Parsing

input : a list of `sentences` to be parsed and a `model`, which makes parser decisions.

Initialize a list of `partial_parses`, one for each sentence in `sentences`;
Initialize a shallow copy of `partial_parses` called `unfinished_parses`;
while `unfinished_parses` is not empty **do**
    Use 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 those parses that are completed from `unfinished_parses`;
end
return The `arcs` for each (now completed) parse in `partial_parses`.

Implement this algorithm in the `minibatch_parse` function in `parser.py`. You can run basic (non-exhaustive) tests by running `python3 parser.py`.

(f) (15 marks) Training your model to predict the right transitions will require you to have a notion of how well it performs on each training example. The parser's ability to produce a good dependency tree for a sentence is measured using an **attachment score**. This is the percentage of words in the sentence that are assigned as a dependent of the correct head. The unlabelled attachment score (**UAS**) considers only this, while the labelled attachment score (**LAS**) considers the label on the dependency relation as well. While this is ultimately the score we want to maximize, it is difficult to use this score to improve our model on a continuing basis. Instead, we use the model's per-transition accuracy (per partial-parse) as a proxy for the parser's attachment score.

To do this we will build an oracle that, given a partial parse and a final set of arcs, predicts the next transition towards the solution given by the final set with perfect accuracy. The error of our model's predictions versus the oracle's will back-propagate and thereby optimize our model's parameters. Implement your oracle in the `get_oracle` method of `parser.py`. Once again, you can run basic tests by running `python3 parser.py`.

2. A Neural Dependency Parser (25 marks)

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. The function that extracts the features that we will use has been implemented for you in `data.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). Using one-hot encodings, our network will first look up the semantic embedding for each word and concatenate them into a single input vector:

$$x_w = [L_{w_0}\,;\, L_{w_1}\,;\, \ldots\,;\, L_{w_m}] \in \mathbb{R}^{dm}$$

where $L \in \mathbb{R}^{|V| \times d}$ is an embedding matrix where each row L_i is the vector for the i-th word. The embeddings for tags (x_t) and arc-labels (x_l) are generated in a similar fashion from their own embedding matrices. The three vectors are then concatenated together:

$$x = [x_w\,;\, x_t\,;\, x_l]$$

From the combined input, we then compute our prediction probabilities as:

$$h = \max\left((xW_h + b_h), 0\right)$$
$$\hat{y} = \text{softmax}(hW_o + b_o).$$

The function for h is called the rectified linear unit (ReLU) function.

The objective function for this network (i.e., the value we will aim to minimize) with parameters θ is the cross-entropy loss:

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

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

(a) (3 marks) In order to avoid neurons becoming too correlated and ending up in a poor local minimum, as well as for stable training, it is important to initialize parameters carefully. One of the most frequent initialization strategies for ReLU layers is called He initialization. Given a layer's weight matrix A of dimension $m \times n$, where m is the number of input units to the layer and n is the number of units in the layer, He initialization selects values A_{ij} from a Gaussian distribution with mean 0 and standard deviation σ, where

$$\sigma = \sqrt{\frac{2}{m}}$$

Implement this method as `he_initializer` in `initialization.py`. You can run basic (non-exhaustive) tests by running `python3 initialization.py`. This function will be used to initialize W_h, W_o, the tag embedding matrix, and the arc-label embedding matrix.

(b) (22 marks) In `model.py`, implement the neural network classifier governing the dependency parser by filling in the appropriate sections; they are marked by `BEGIN` and `END` comments. You will train and evaluate the model on a modified version of the Penn Treebank that has been annotated with universal dependencies. Run `python3 model.py` to train your model and compute predictions on the test data.

Notes:

- When debugging, you may find it useful to pass the argument `debug=True` to the main function. This will cause the code to run over a small subset of the data so that training the model will take less time. Debug mode also does the featurizing of the training set during each of the training epochs rather than all at once before starting, so that you can get into the training code faster. Remember to change the setting back prior to submission, and before doing your final run.
- This code should run within 2–3 hours on a CPU, but may be a bit faster or slower depending on the specific CPU. A GPU is much faster, taking less than 10 minutes, again with some variance depending on the specific GPU.
- With everything correctly implemented (and with `debug=False`), you should be able to get a loss less than 0.11 on the training set by the end of the last epoch and a Labelled Attachment Score of at least 0.88 on the dev set (with the best-performing model out of all the epochs). If you want, you can tweak the hyperparameters for your model (hidden layer size, number of epochs, optimizer hyperparameters, etc.) to try to improve the performance, but you are not required to do so.
- In your final submission, make sure that all of your changes occur within `BEGIN` and `END` boundary comments. **Do not modify other parts of the files.**
- You will be better off with partially complete implementations that work but are sometimes incorrect than you will with an almost-complete implementation that Python can't run. Keep this in mind for your final submission.

(c) **Bonus** (1 mark). Add an extension to your model (e.g., L2 regularization, an additional hidden layer, etc.) and report the change in both LAS and UAS on the dev set. Briefly explain what your extension is and why it helps (or hurts!) the model. Some extensions may require tweaking the hyperparameters in `Config` to make them effective.

What to submit

---

1 On paper

Your answers to questions (1a), (1b), and (1c). Also submit a short report—1 paragraph should be plenty—for (2b) where you provide the best LAS and UAS that your model achieved on the dev and test sets and discuss any other things you may have attempted, such as the optional hyperparameter tweaks. If you ran into trouble, you can mention this as well. Submit a similar report for (2c) as well if you attempt it, making sure to provide details about your improvements.

2 Electronically

Submit the entirety of your `initialization.py`, `model.py`, and `parser.py` files. If you attempt question (2c), submit it in a separate file called `model_ext.py`. Again, in your final submission, ensure that there are no changes outside the `BEGIN` and `END` boundaries given in the comments that tell you where to add your code; you will lose marks if you do. Do not change or remove the boundary comments either.

From your final run (i.e., the one that corresponds to results you mention in your report), submit the weights output file as a file called `weights.pt` (rename it to this filename if necessary). If you submit an answer for (2c), also submit its weights as a separate file called `weights_ext.pt`.

Submit all required files using the `submit` command on `teach.cs`:

```
$ submit -c <course> -a A1 <filename-1> ... <filename-n>
```

where `<course>` is `csc485h` or `csc2501h` depending on which course you're registered in, and `<filename-1>` to `<filename-n>` are the *n* files you are submitting.

Appendix A PyTorch

You will be using PyTorch to implement components of your neural dependency parser. PyTorch is an open-source library for numerical computation that provides automatic differentiation, making the back-propagation aspect of neural networks easier. Computation is done in units of *tensors*; the storage of and operations on tensors is provided in both CPU and GPU implementations, making it simple to switch between the two.

We recommend trying some of the tutorials on the official PyTorch site (e.g., `https://pytorch.org/tutorials/beginner/pytorch_with_examples.html`) to get up to speed on Tensorflow's mechanics. It will introduce you to the data structures used and provide some examples.

If you want to run this code on your own machine, you can install the `torch` package via `pip3`, or any of the other installation options available on the PyTorch site (`https://pytorch.org/get-started/locally/`). The code we've provided for this assignment will automatically use the GPU if it's available. Make sure to use PyTorch version 1.2.

CSC 2501 / 485, Fall 2019: Assignment 1

Family name: _____

First name: _____

Course (485 or 2501): _____

Teaching labs username: _____

Staple to assignment this side up

CSC 2501 / 485, Fall 2019: Assignment 1

Family name: _____

First name: _____

Student #: _____

Date: _____

I declare that this assignment, both my paper and electronic submissions, is my own work, and is in accordance with the University of Toronto Code of Behaviour on Academic Matters and the Code of Student Conduct.

Signature: _____

Grade:

1. _____ / 40
2. _____ / 25

TOTAL_____ / 65