University of Toronto, Department of Computer Science
CSC 485F: Computational linguistics, Summer 2020

Assignment 1

Due date: 23:59 on Friday, May 22, 2020.
Late assignments will not be accepted without a valid medical certificate or other documentation of an emergency.
This assignment is worth 45% of your final grade.

- **Read the whole assignment carefully.**
- 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 Discourse forum for the class. 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/summer/pub/deptrans/`. In this handout, code files we refer to are located in that directory.
- When implementing code, make sure to **read the docstrings carefully** as they provide important instructions and implementation details.
- Fill in your name, student number, and `teach.cs` login username on the relevant lines at the top of each Python file that you submit. (Do not add new lines; just replace the `NAME`, `NUMBER`, and `USERNAME` placeholders.)

1. Transition-based dependency parsing (11 marks)

Dependency grammars posit relationships between "head" words and their modifiers. 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:

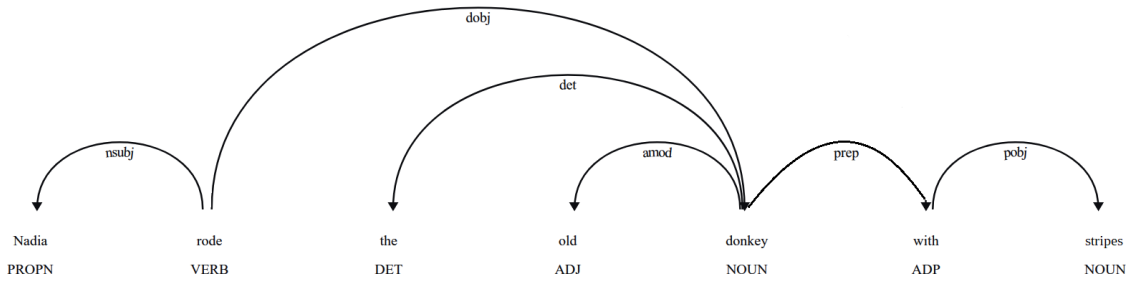
- A stack of words that are currently being processed.
- A buffer of words yet to be processed.
- A list of dependencies predicted by the parser.

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.

In this assignment, you'll be implementing a dependency parser based on the above mechanism. But first, let's practice and think about this parsing mechanism.

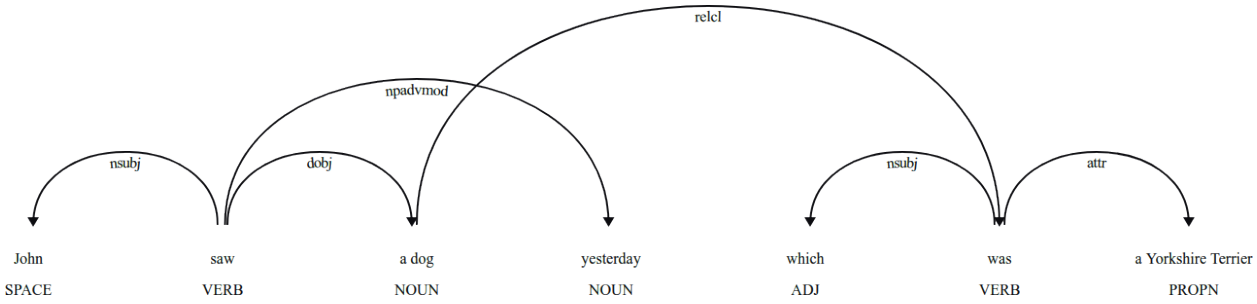
(a) (6 marks) Go through the sequence of transitions needed for parsing the sentence "Nadia rode the old donkey with stripes." The dependency tree for the sentence is shown below (without ROOT). At each step, provide the configuration of both the stack and the buffer, as well as which transition to apply 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, stripes]		Initial Config
[ROOT, Nadia]	[rode, the, old, donkey, with, stripes]		SHIFT
[ROOT, Nadia, rode]	[the, old, donkey, with, stripes]		SHIFT
[ROOT, rode]	[the, old, donkey, with, stripes]	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) (3 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.



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

The *gap-degree* of a word in a dependency tree is the least k for which the substring consisting of the word and its descendants is entirely comprised of $k+1$ contiguous substrings. The gap-degree of a dependency tree is the greatest gap-degree of any word in the tree.

What is the gap-degree of the above non-projective tree? How do you know?

2. Implementing a transition-based dependency parser (28 marks)

Your parser will use a neural network as a classifier that decides which transition to apply at a given *partial parse* state. A partial parse state is collectively defined by a sentence buffer as above, a stack as above with any number of items on it (including zero), and a set of correct dependency arcs for the sentence.

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

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`.

(b) (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 according to Algorithm 1.

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

(c) (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 will use the model's per-transition accuracy (per partial parse) as a proxy for the parser's attachment score, so we will need the correct transitions. But the data set just provides sentences and corresponding dependency arcs. You must therefore implement an *oracle* that, given a partial parse and a set of correct, final target dependency arcs, provides the next transition to take to advance the partial parse towards the final solution set of dependencies. The classifier will later be trained to try to predict the correct transition by minimizing the error in its predictions versus the transitions provided by the oracle.

Implement your oracle in the `get_oracle` method of `parse.py`. As above, you can run basic tests by running `python3 parse.py`.

3. Neural dependency transition prediction (15 marks)

The last step is to construct and train a neural network to predict which transition should be applied next, given the state of the stack, buffer, and dependencies. 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 pre-trained *word embeddings* (i.e., word vectors), our network will first look up the 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 and the semicolon represents concatenation. 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:

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

The function for h is called the rectified linear unit (ReLU) function: $\text{ReLU}(z) = \max(z, 0)$.

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) (2 marks) For effective, stable training, especially for deeper networks, it is important to initialize weight matrices carefully. A standard initialization strategies for ReLU layers is called Kaiming initialization[1]. For a given layer's weight matrix W 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, Kaiming initialization samples values W_{ij} from a Gaussian distribution with mean $\mu = 0$ and variance $\sigma^2 = 2/m$.

[1] Also referred to as He initialization.

However, it is common to instead sample from a *uniform* distribution $U(a,b)$ with lower bound a and upper bound b. Derive the values of a and b that yield a uniform distribution $U(a,b)$ with the mean and variance given above.

(b) (13 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 `model.py`, 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 around 10 minutes or less—again, with some variation depending on the specific GPU. The code we've provided for this assignment will automatically use the GPU if it's available; you should never have to specify the device of a `Tensor` or call `.clone().detach()` on one.

- Via ssh to `teach.cs.toronto.edu`, you can run your model on a GPU-equipped machine. The starter code directory includes a `gpu-run-model` script that will run the relevant command to run your `model.py` file, assuming the latter is in your current working directory (CWD; i.e., assuming that you run your the command while in the same directory as `model.py`). Note that the GPU machines are shared resources, so there may be limits on how long your code is allowed to run.

  The `gpu-run-model` script should be sufficient; it takes care of submitting and running the job to the cluster-management software. If you like, you are free to manage this yourself; see the documentation online at the relevant page on the Teaching Labs site. The partition you can use for this class is `csc485`.

- With everything correctly implemented (and with `debug=False`), you should be able to get a Labelled Attachment Score of *around* 0.88 on both the validation and test sets (with the best-performing model out of all the epochs at the end of training). If you want, you can tweak the hyperparameters for your model (hidden layer size, number of hidden layers, number of epochs, optimizer hyperparameters, etc.) to try to improve the performance, but you are not required to do so, and it should not be necessary to achieve the above LAS.

Keep in mind that your mark is based on your implementation, *not* (directly) on the LAS your implementation achieves. An LAS of 0.88 doesn't guarantee full marks, nor does an LAS less than 0.88 guarantee that marks will be docked. **The end result can vary based on differences in environment** (hardware or software) so this number is approximate! (But the further away your LAS, the more likely it is that you have an error in your implementation.)

- In your final submission, make sure that all of your changes occur within `BEGIN` and `END` boundary comments. **Do not add any extra package or module imports and do not modify any other parts of the files** other than to fill your details in at the top of the file as instructed on the front page of this assignment handout.
- 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. Also, make sure your submitted files are importable in Python (i.e., make sure that `'import model.py'` and `'import parse.py'` don't yield errors).

What to submit

Submission is electronic and can be done from any Teaching Labs machine using the `submit` command:

```
$ submit -c csc485h -a A1 <filename-1> ... <filename-n>
```

where `<filename-1>` to `<filename-n>` are the *n* files you are submitting. The files you are to submit are as follows:

- `a1written.pdf`: a PDF document containing your answers to questions (1a), (1b), (1c), and (3a). As well, for (3b), provide the best LAS and UAS that your model achieved on the validation and test sets. If you attempted some of the optional hyperparameter tweaking or ran into trouble somewhere, you can mention this here.

  This PDF must also include a typed copy of the Student Conduct declaration as on the last page of this assignment handout. Type the declaration as it is and sign it by typing your name.

- `model.py`: the (entire) `model.py` file with your implementations filled in.
- `parse.py`: the (entire) `parse.py` file with your implementations filled in.
- `weights.pt`: the weights file produced by your model's final run (i.e., the one that corresponds to the scores you present in `a1written.pdf`).

Again, in your code submissions, ensure that there are no changes outside the `BEGIN` and `END` boundary comments provided other than filling in your details at the top of the file; **you will lose marks if there are**. Do not change or remove the boundary comments either.

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 (such as this one) to get up to speed on PyTorch'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. Make sure to use PyTorch version 1.5. You will also need to install the `nltk` package.

Make sure to consult the PyTorch documentation as needed. The API documentation for `torch`, `torch.nn`, `torch.nn.functional`, and `torch.Tensor` will be useful while implementing the code in this assignment.

CSC 485, Summer 2020: 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: _____