University of Toronto, Department of Computer Science
**CSC 485/2501F—Computational Linguistics, Fall 2018**

# Assignment 3

**Due date:** 23:59, Thursday 6 December 2018, at 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.*

- 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. (Don't turn in a copy of this handout.)
- Please 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.

# 1. Transition-Based Dependency Parsing (40 marks)

In this assignment, you'll implementing a neural-network-based dependency parser. Dependency grammars posit relationships between "head" words and their modifiers, much like the function-argument relations that are required by lexical entries in categorial grammar. 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.

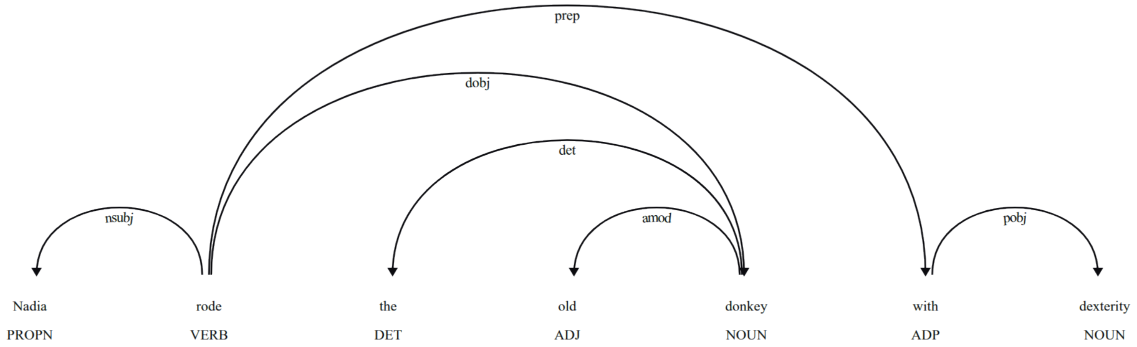
This is very much like a shift-reduce parser. Initially, the stack only contains ROOT, the dependencies lists is empty, and the buffer contains all words of the sentence in order. At each step, the parser advances by applying a "transition" to the partial parse until its buffer is empty and the stack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

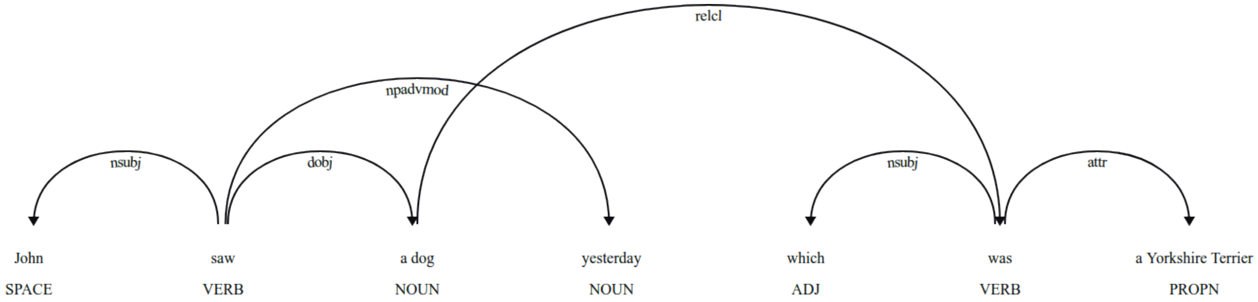
(a) (6 marks) Go through the sequence of transitions needed for parsing the sentence "Nadia rode the old donkey with dexterity." The dependency tree for the sentence is shown below (without ROOT). At each step, provide the configuration of both the stack and the buffer, as well as which transition to 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 $\overset{nsubj}{\to}$ Nadia | LEFT-ARC |



(b) (2 marks) A sentence containing *n* words will be parsed in how many steps (in terms of *n*)? 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 (about 99%) of the sentences in our data-set have projective dependency trees.

(d) (7 marks) Implement the `complete` and `parse_step` functions in the PartialParse class in `/h/u2/csc485h/fall/pub/deptrans/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 (not-exhaustive) tests by running `python 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 batches of data at a time, i.e., predicting the next transition for a 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 (not-exhaustive) tests by running `python 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 unlabeled 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` function of `parser.py`. Once again, you can run basic tests by running `python 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 `parser_utils`. 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). Our network should 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. We then compute our production as:

$$h = \max\left((x^w W_1^w + x^t W_1^t + x^l W_1^l + b_1), 0\right)$$
$$\hat{y} = \mathrm{softmax}(hU + b_2).$$

The rectifier for $h$ should be called with the function `tf.nn.relu`.

We evaluate using 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) (5 marks) In order to avoid neurons becoming too correlated and ending up in a poor local minimum, it is often helpful to randomly initialize parameters. One of the most frequent initialization strategies is called Xavier initialization[1].

Given a matrix $A$ of dimension $m \times n$, Xavier initialization selects values $A_{ij}$ uniformly from $[-\varepsilon, \varepsilon]$, where

$$\varepsilon = \frac{\sqrt{6}}{\sqrt{m+n}}$$

Implement the initialization as `xavier_weight_init` in `/h/u2/csc485h/fall/pub/deptrans/initialization.py`. You can run basic (nonexhaustive tests) by running `python initialization.py`. This function will be used to initialize $W$ and $U$.

(b) (20 marks) In `/h/u2/csc485h/fall/pub/deptrans/model.py` implement the neural network classifier governing the dependency parser by filling in the appropriate sections. We will train and evaluate our model on a modified version of the Penn Treebank that has been annotated with universal dependencies. Run `python model.py` to train your model and compute predictions on the test data. Make sure to turn off debug settings when doing your final evaluation!

**Hints:**

[1] This is also referred to as Glorot initialization and was initially described in http://jmlr.org/proceedings/papers/v9/glorot10a/glorot10a.pdf

- When debugging, pass the keyword argument `debug=True` to the main method (`model.py` does this by default). This will cause the code to run over a small subset of the data, so that training the model will take less time.
- This code should run within 1 hour on a CPU.
- When running with `debug=False`, you should be able to get a loss smaller than 0.11 on the train set (by the end of the last epoch) and a Labeled Attachment Score of at least 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, hyperparameters for Adam, number of epochs, etc.) to improve the performance, but you are not required to do so.

(c) **Bonus** (1 mark). Add an extension to your model (e.g., L2 regularization or an additional hidden layer) 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.

## 0.1 What to submit

### 0.1.1 On paper

Your answers to (1a), (1b) and (1c). For questions (2b) and (2c), write a report in which you discuss what you attempted. When addressing (2b), be certain to report the best LAS and UAS that your model achieved on the dev set and the LAS and UAS that your submission achieves on the test set.

### 0.1.2 Electronically

Submit the entirety of the following files: `initialization.py, model.py`, and `parser.py`. Do not make changes outside the BEGIN and END boundaries given in the comments that tell you where to add your code. Do not remove the boundary comments either.

Also submit a file `q2btest.txt` that lists the labels you predict for the test set in (2b). If you submit an answer for (2c), submit a separate file `q2ctest.txt` for that.

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

```
% submit -c <course> -a A3 <filename-1>...<filename-n>
```

where `<course>` is either `csc485h` or `csc2501h`, and `<filename-1>` to `<filename-n>` are the *n* files you are submitting. Make sure every file you turn in contains a comment at the top that gives your name, your login ID on `teach.cs`, and your student ID number.

# Appendix A Tensorflow

You will be using Tensorflow to implement components of your neural dependency parser. Tensorflow is an open-source library for numerical computation using dataflow graphs. Nodes in the graph represent mathematical operations, while the graph edges represent multidimensional data arrays (tensors) that are communicated between them. This architecture makes it simple to construct rather complex models, and distribute the computation across multiple CPUs, GPUs and physical machines.

We recommend reading through the 'Getting Started' guide (`https://www.tensorflow.org/get_started/get_started`) to get up to speed on Tensorflow's mechanics. It will introduce you to the data-structures used to construct data-flows, such as *placeholders* and *variables*. Understanding these mechanics will give you a better idea of what's going on in the second question.

If you want to run this code on your own machine, we recommend installing the **CPU** version of Tensorflow using the instructions found at `https://www.tensorflow.org/install/`. If you're feeling adventurous and have a compatible graphics card, you can try installing the GPU version. Use Tensorflow version 1.3 or 1.4.

# CSC 2501 / 485, Fall 2018: Assignment 3

Family name: ____________________ First name: ____________________

Staple to assignment this side up

# CSC 2501 / 485, Fall 2018: Assignment 3

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