Natural Language Processing **Tel Aviv University**

Assignment 4: Dependency Parsing

Due Date: *January 22, 2019* Lecturer: Jonathan Berant

In this home assignment we will implement and train a neural dependency parser using PyTorch, and analyze a few erroneous dependency parses. You are provided with the data and supporting code. **Note that the Penn Treebank dataset is licensed! It is illegal to make it public in any way!**

0 Preliminaries

Submission Instructions The assignment files can be downloaded from `https://drive.google.com/open?id=1P6sKWaBKDEAscTxIAH_GXoF7QCDBOTK-`. Submit your solution through Moodle. Your submission should consist of a single zip file named `<id1>_<id2>.zip` (where `<id1>` refers to the ID of the first student). This zip file should include **all the code files (including the ones you did not modify)** and a written solution in a **PDF format** named `answers.pdf`. **Don't include the data**. Only one student needs to submit. If you are comfortable with LaTeX, feel free to use the supplied `written_solution_template.tex` as a basis for you written solution.
Your code will be tested with Python 3.7 on the School of Computer Science operating system, installed on nova and other similar machines. Please make sure your code runs there. If your code does not run on nova, it will not be graded. Avoid any code updates, besides in sections marked by `YOUR CODE HERE`. Solution which includes other modifications (such as function signature modification or redundant logs) will not be graded.

Acknowledgements This assignment was adapted from Stanford's CS224n course. Additional background material was adapted from slides by Joakim Nivre. Their contributions are greatly appreciated.

1 Neural Transition-Based Dependency Parsing

Dependency parsing is the task of analyzing the syntactic dependency structure of a given input sentence $\mathcal{S}$. A dependency parser analyzes the grammatical structure of a sentence, establishing relationships between *head* words, and words which modify those heads. It outputs a dependency tree where the words of the input sentence are connected by typed dependency relations. Formally, the dependency parsing problem asks to create a mapping from the input sentence with words $\mathcal{S} = w_0 w_1 \dots w_n$ (where w_0 is the ROOT) to its dependency tree graph G (see Figure 1 for an example). $G = (\{w_0, w_1, \dots, w_n\}, A)$, where $A \subset \{(w_i, l, w_j) | 0 \le i, j \le n, \; l \text{ is a type of a dependency relation}\}$. It is a well-formed tree if and only if:

- every node has at most one incoming arc (single head)
- the graph is (weakly) connected (no dangling nodes)
- there are no cycles (no word is dependent on itself)

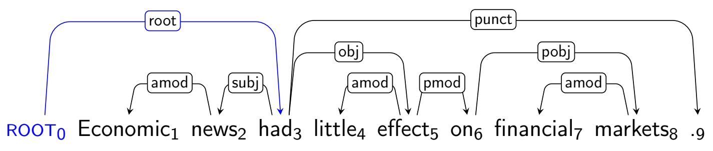


Figure 1: A dependency tree.

Refer to CS224n's lecture notes for more background.

In this section, you'll be implementing a neural-network based dependency parser, with the goal of maximizing performance on the UAS (Unlabeled Attachemnt Score) metric.[1]
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.

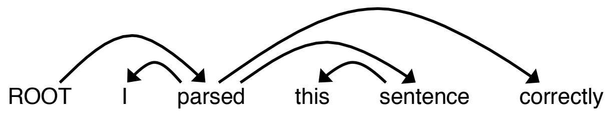
Initially, the stack only contains ROOT, the dependencies list is empty, and the buffer contains all words of the sentence in order. At each step, the parser applies a *transition* to the partial parse until its buffer is empty and the stack size is 1. The following transitions can be applied:

- `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.

(a) Configure an environment with Python 3.7, torch==1.3.1 and tqdm (produces progress bar visualizations throughout your training process).

(b) Go through the sequence of transitions needed for parsing the sentence *"I parsed this sentence correctly"*. The dependency tree for the sentence is shown below. At each step, give the configuration of the stack and buffer, as well as what transition was applied this step and what new dependency was added (if any). The first three steps are provided below as an example.



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→I	`LEFT-ARC`

(c) A sentence containing n words will be parsed in how many steps (in terms of n)? Briefly explain why.

(d) Implement the `__init__` and `parse_step` functions in the `PartialParse` class in `parser_transitions.py`. This implements the transition mechanics your parser will use. You can run basic (non-exhaustive) tests by running `python parser_transitions.py part_d`.

(e) 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

[1]UAS is the percentage of words that get the correct head, without considering the label.

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 `parser_transitions.py`. You can run basic (non-exhaustive) tests by running `python parser_transitions.py part_e`. *Note: You will need `minibatch_parse` to be correctly implemented to evaluate the model you will build in part (e). However, you do not need it to train the model, so you should be able to complete most of part (e) even if `minibatch_parse` is not implemented yet.*

(f) 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.*[2] 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 $[w_1, w_2, \ldots, w_m]$ 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). First our network looks up an embedding for each word and concatenates them into a single input vector:

$$\mathbf{x} = [\mathbf{E_{w_1}}, \ldots, \mathbf{E_{w_m}}] \in \mathbb{R}^{dm}$$

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:

$$\mathbf{h} = \text{ReLU}(\mathbf{xW} + \mathbf{b}_1)$$
$$\mathbf{l} = \mathbf{hU} + \mathbf{b}_2$$
$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{l})$$

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.

[2]Chen and Manning, 2014, http://cs.stanford.edu/people/danqi/papers/emnlp2014.pdf

In `parser_model.py` you will find skeleton code to implement this simple neural network using PyTorch. Complete the `__init__`, `embedding_lookup` and `forward` functions to implement the model. Then complete the `train_for_epoch` and `train` functions within the `run.py` file.

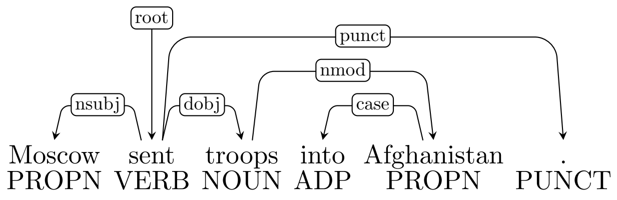
Finally execute `python run.py` to train your model and compute predictions on test data from Penn Treebank (annotated with Universal Dependencies).

Hints:

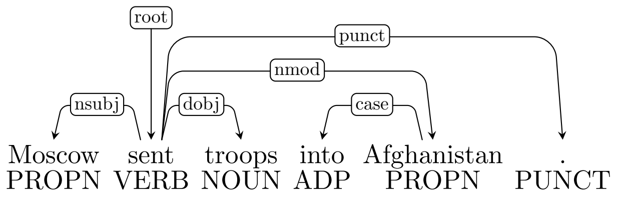
- When debugging, execute `python run.py --debug`. This will cause the code to run over a small subset of the data, so that training the model won't take as long.
- When running with `--debug`, you should be able to get a loss smaller than 0.2 and a UAS larger than 65 on the dev set (although in rare cases your results may be lower, there is some randomness when training).
- With CPU, it should take about **1 hour** to train the model on the entire the traing dataset. Alternatively, you can use Google Colab to complete training in a few minutes by taking the following steps:
  - i Go to https://colab.research.google.com/drive/1q6zN3LcreDR9TUXJR5yWn3uWi1Af8jnE and copy it to your drive (`File → Save a copy in Drive`).
  - ii Open the left pane, go to the `Files` tab, and upload the files `for_colab.zip` and your **modified** `parser_transitions.py`, `parser_model.py` and `run.py`. If you need to update and reupload files, restart the runtime for the changes to take effect.
- When running without `--debug`, 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, **without changing hyperparameters**. 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).

(g) Report the best UAS your model achieves on the dev set and the UAS it achieves on the test set.

(h) We'd like to look at example dependency parses and understand where parsers like ours might be wrong. For example, in this sentence:


root
punct
nmod
nsubj
dobj
case
Moscow sent troops into Afghanistan .
PROPN VERB NOUN ADP PROPN PUNCT

the dependency of the phrase *into Afghanistan* is wrong, because the phrase should modify *sent* (as in *sent into Afghanistan*) not *troops* (because *troops into Afghanistan* doesn't make sense). Here is the correct parse:


root
punct
nmod
nsubj
dobj
case
Moscow sent troops into Afghanistan .
PROPN VERB NOUN ADP PROPN PUNCT

More generally, here are four types of parsing error:

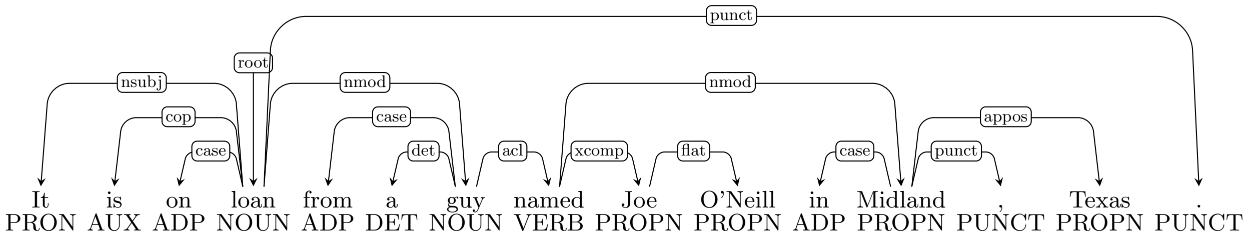
- **Prepositional Phrase Attachment Error**: In the example above, the phrase *into Afghanistan* is a prepositional phrase. A Prepositional Phrase Attachment Error is when a prepositional phrase is attached to the wrong head word (in this example, *troops* is the wrong head word and *sent* is the correct head word). More examples of prepositional phrases include *with a rock*, *before midnight* and *under the carpet*.
- **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 $\rightarrow$ Afghanistan
- **Error type**: sent $\rightarrow$ Afghanistan

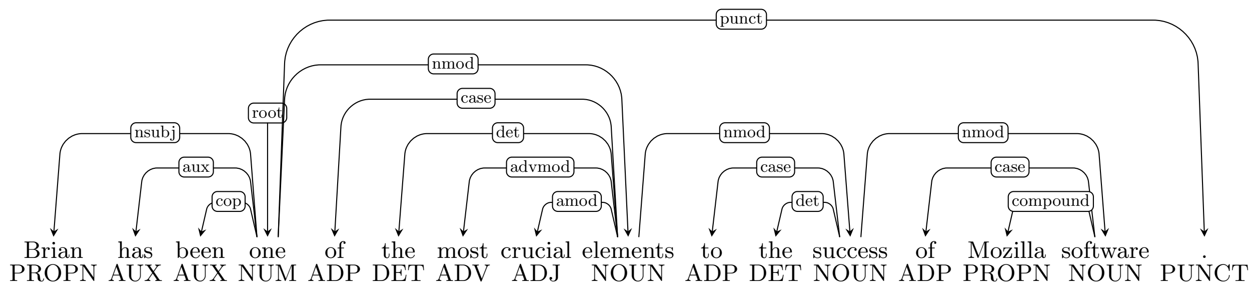
Note: *There are lots of details and conventions for dependency annotation. If you want to learn more about them, you can look at the UD website:* http://universaldependencies.org.[3] *However, you* ***do not*** *need to know all these details in order to do this question. In each of these cases, we 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.

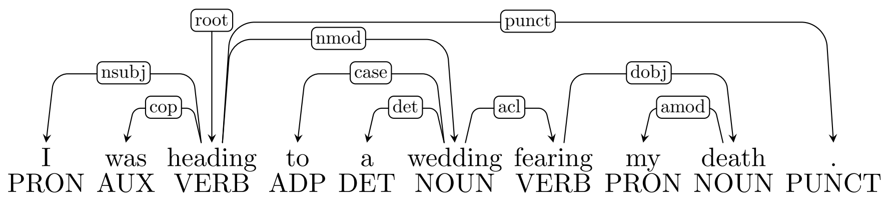


[3]But note that in the assignment we are actually using UDv1, see: http://universaldependencies.org/docsv1/

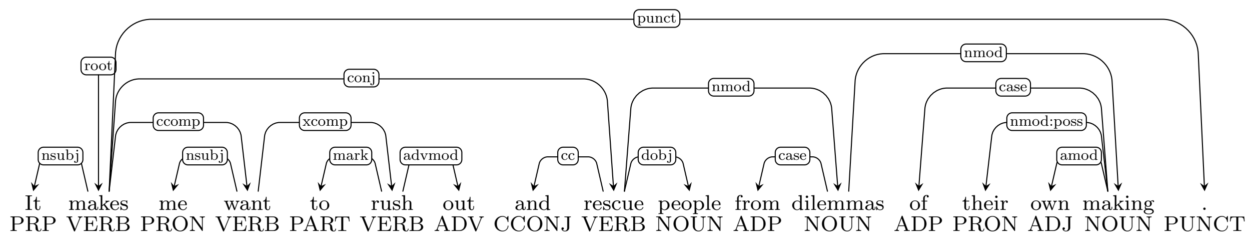
ii.


punct
nmod
case
root
nsubj
det
nmod
nmod
aux
advmod
case
case
cop
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

iii.


root
punct
nmod
nsubj
case
dobj
cop
det
acl
amod
I was heading to a wedding fearing my death .
PRON AUX VERB ADP DET NOUN VERB PRON NOUN PUNCT

iv.


punct
root
nmod
conj
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