Deep Learning for NLP

Assignment 2: Dependency Parsing

1 Introduction

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

We will be using the `torch` and the `tqdm` package – which produces progress bar visualizations throughout your training process. Remember that the official PyTorch website is a great resource that includes tutorials for understanding PyTorch's Tensor library and neural networks.

2 Dependency parsing

A dependency parser analyzes the grammatical structure of a sentence, establishing relationships between head words, and words which modify those heads. There are multiple types of dependency parsers, including transition-based parsers, graph-based parsers, and feature-based parsers. 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:

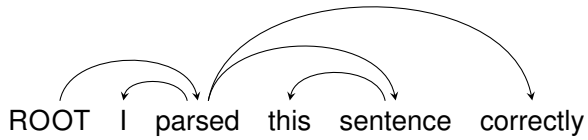
- `SHIFT`: removes the first word from the buffer and pushes it onto the stack.
- `LEFT-ARC`: marks the second (second most recently added) item on the stack as a dependent of the first item and removes the second item from the stack, adding a `first_word` $\rightarrow$ `second_word` dependency to the dependency list.
- `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, adding a `second_word` $\rightarrow$ `first_word` dependency to the dependency list.

On each step, your parser will decide among the three transitions using a neural network classifier.

3 Understanding

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

4 Implementation

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_c
```

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 multiple samples of data at a time i.e., predicting the next transition for different partial parses simultaneously. We can parse sentences in this *mini-batch mode* with the following algorithm:

Algorithm 1: Minibatch Dependency Parsing

Result: The `dependencies` for each (now completed) parse in `partial_parses`
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

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_d
```

Note: You will need `minibatch_parse` to be correctly implemented to evaluate the model built in the next part. However, you do not need the function to train the model, so you should be able to complete most of the next part even if `minibatch_parse` is not implemented yet.

5 Training

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*[1]. 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 m integers $\mathbf{w} = [w_1, ..., w_m]$ and each entry w_i is the index of a token in the vocabulary, so that $0 \leq w_i < |V|$ (zero-based indicies), where $|V|$ is the vocabulary size. Then our network looks up an embedding for each word at the according index and concatenates them along a single dimension into a (flat) input vector:

$$x = [\mathbf{E}_{w_1}; ...; \mathbf{E}_{w_m}] \in \mathbb{R}^{m \cdot d}$$

with $\mathbf{E} \in \mathbb{R}^{|V| \times d}$ as an embedding matrix where each row is the vector for a particular word. We then compute our prediction as:

$$\mathbf{h} = \text{ReLU}(\mathbf{x}\mathbf{W} + \mathbf{b_1}) \tag{1}$$

$$\mathbf{o} = \mathbf{h}\mathbf{U} + \mathbf{b_2} \tag{2}$$

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{o}) \tag{3}$$

where $\mathbf{h}$ is referred to as the hidden layer, $\mathbf{o}$ is referred to as the output or logits of the network, $\hat{\mathbf{y}}$ is referred to as the predictions, and ReLU(z) = max($z, 0$)). We will train the model to minimize the cross-entropy loss:

$$J(\theta) = CE(y, \hat{y}) = -\sum_{i=1}^{3} y_i \, log \, \hat{y}_i$$

To compute the loss for the training set, we average this $J(\theta)$ across all training examples. We will use UAS score as our evaluation metric. UAS refers to *Unlabeled Attachment Score*, which is computed as the ratio between the number of correctly predicted dependencies and the number of total dependencies despite of the relations (our model doesn't predict this).

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 the Penn Treebank corpus (annotated with Universal Dependencies).

[1] `https://www.aclweb.org/anthology/D14-1082.pdf`

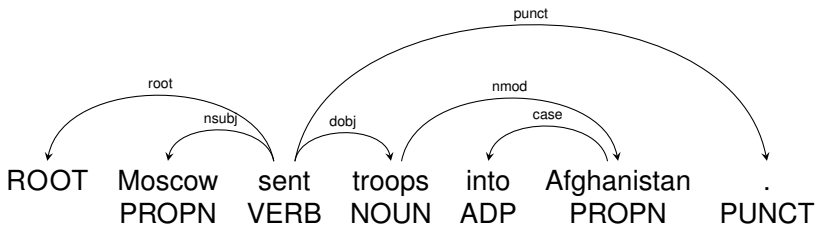
Please follow the naming requirements in the `TODO` if there are any, e.g., if there are explicit requirements about variable names you have to follow them. You are free to declare other variable names if not explicitly required.

Hints:

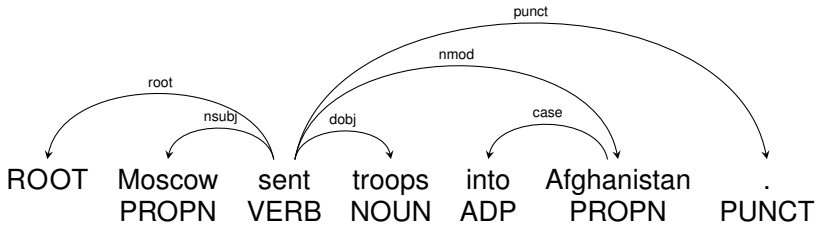
- Each of the variables you are asked to declare (`self.embed_to_hidden_weight`, `self.embed_to_hidden_bias`, `self.hidden_to_logits_weight`, `self.hidden_to_logits_bias`) corresponds to one of the variables above (**W, b1, U, b2**).
- It may help to work backwards in the algorithm (start from $\hat{\mathbf{y}}$) and keep track of the matrix/vector sizes.
- Once you have implemented `embedding_lookup` or `forward` you can call `python parser_model.py` with flag `-e` or `-f` or both to run sanity checks with each function. These sanity checks are fairly basic and passing them doesn't mean your code is bug free.
- When debugging, you can add a debug flag: `python run.py -d`. This will cause the code to run over a small subset of the data, so that training the model won't take as long (dry-run). Make sure to remove the `-d` flag to run the full model once you are done debugging.
- When running with debug mode, 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 training dataset, i.e., when debug mode is disabled.
- When the debug mode is disabled, 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).

6 Analysis

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



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:



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[2]. 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).

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[3]. 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*.

Here below are three sentences with dependency parses obtained from a parser. Each sentence has one error type from one of the four types above. For each sentence, state the type of error, the incorrect dependency, and the correct dependency. While each sentence should have a unique error type, there may be multiple possible correct dependencies for some of the sentences.

To demonstrate: for the example above, you would write:

- Error type: Prepositional Phrase Attachment Error
- Incorrect dependency: troops $\rightarrow$ Afghanistan
- Correct dependency: sent $\rightarrow$ Afghanistan

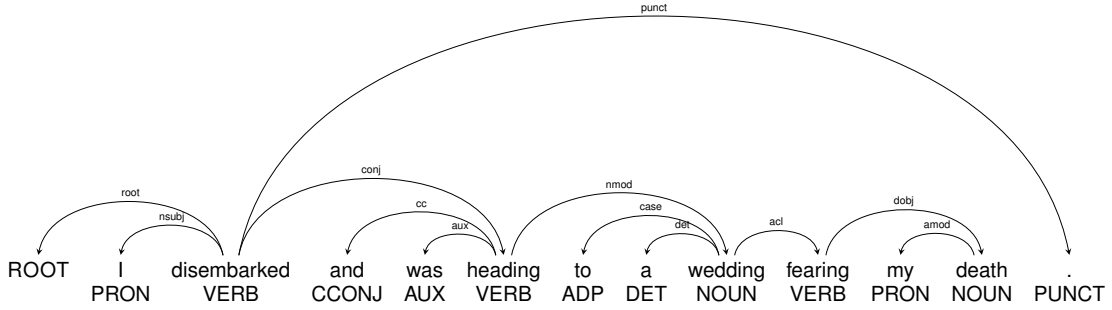
Note: There are lots of details and conventions for dependency annotation. If you want to learn more about them, you can look at the UD website: `http://universaldependencies.org`. You don't need to know all these details in order to answer this exercise. In each of these

[2]For examples of prepositional phrases, see: `https://www.grammarly.com/blog/prepositional-phrase/`

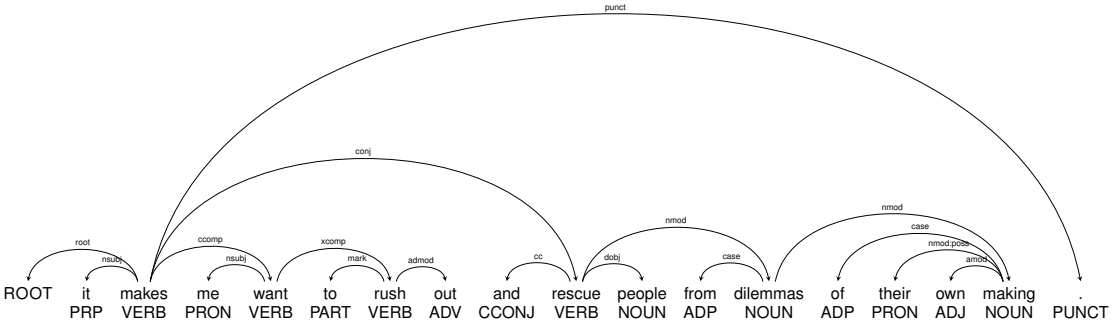
[3]For examples of verb phrases, see: `https://examples.yourdictionary.com/verb-phrase-examples.html`

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 don't need to look at the labels on the the dependency edges – it suffices to just look at the edges themselves.

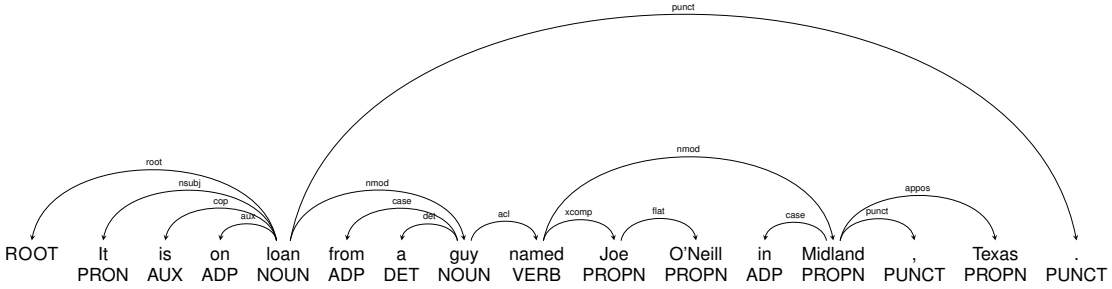
1.



2.



3.



7 Submission

- Upload your files `src/parser_transitions.py`, `src/parser_model.py`, and `src/run.py` file in a single compressed file on Moodle.
- In addition, upload your answers for sections 3 and 6 in a separate PDF file.