CS 224n Assignment #3: Dependency Parsing

Anthony Weng

Updated Sunday 29^{th} January, 2023 at 12:23am

In this assignment, you will build a neural dependency parser using PyTorch. For a review of the fundamentals of PyTorch, please check out the PyTorch review session on Canvas. In Part 1, you will learn about two general neural network techniques (Adam Optimization and Dropout). In Part 2, you will implement and train a dependency parser using the techniques from Part 1, before analyzing a few erroneous dependency parses.

Please tag the questions correctly on Gradescope, the TAs will take points off if you don't tag questions.

1. Machine Learning & Neural Networks (8 points)

(a) (4 points) Adam Optimizer

Recall the standard Stochastic Gradient Descent update rule:

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \alpha \nabla_{\boldsymbol{\theta}_t} J_{\text{minibatch}}(\boldsymbol{\theta}_t)$$

where $t+1$ is the current timestep, $\boldsymbol{\theta}$ is a vector containing all of the model parameters, ($\boldsymbol{\theta}_t$ is the model parameter at time step t, and $\boldsymbol{\theta}_{t+1}$ is the model parameter at time step $t+1$), J is the loss function, $\nabla_{\boldsymbol{\theta}} J_{\text{minibatch}}(\boldsymbol{\theta})$ is the gradient of the loss function with respect to the parameters on a minibatch of data, and α is the learning rate. Adam Optimization[1] uses a more sophisticated update rule with two additional steps.[2]

i. (2 points) First, Adam uses a trick called *momentum* by keeping track of $\mathbf{m}$, a rolling average of the gradients:

$$\mathbf{m}_{t+1} \leftarrow \beta_1 \mathbf{m}_t + (1-\beta_1) \nabla_{\boldsymbol{\theta}_t} J_{\text{minibatch}}(\boldsymbol{\theta}_t)$$
$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \alpha \mathbf{m}_{t+1}$$

where β_1 is a hyperparameter between 0 and 1 (often set to 0.9). Briefly explain in 2–4 sentences (you don't need to prove mathematically, just give an intuition) how using $\mathbf{m}$ stops the updates from varying as much and why this low variance may be helpful to learning, overall.

Answer:

Using $\mathbf{m}$ rather than $\nabla_{\boldsymbol{\theta}} J_{\text{minibatch}}(\boldsymbol{\theta})$ decreases update variance as each parameters update is determined largely by the parameters update in the previous timestep (given the typical value of $\beta = 0.9$) and only marginally incorporates information from the latest minibatch gradient. In practice, this method of computing parameter updates means parameters update vectors in future timesteps will always be, to some degree dependent on the number of eclipsed timesteps, constituted of the parameters update vectors in past timesteps – meaning all parameters update vectors are closer in distance, thereby limiting update variance.

Limiting update variance may be helpful to learning as it could lead to faster parameters convergence through decreased gradient "switching" – i.e., if two gradients in (near-)consecutive timesteps encourage model parameters to move in opposite directions, the model parameters will have changed very little overall and will hardly be closer to finding a local optimum solution. By constructing parameters updates to be more similar to each other, a lesser degree of this directional switching can take place and a local solution can be identified more quickly.

[1] Kingma and Ba, 2015, `https://arxiv.org/pdf/1412.6980.pdf`

[2] The actual Adam update uses a few additional tricks that are less important, but we won't worry about them here. If you want to learn more about it, you can take a look at: `http://cs231n.github.io/neural-networks-3/#sgd`

ii. (2 points) Adam extends the idea of *momentum* with the trick of *adaptive learning rates* by keeping track of $\mathbf{v}$, a rolling average of the magnitudes of the gradients:

$$\begin{aligned}
\mathbf{m}_{t+1} &\leftarrow \beta_1 \mathbf{m}_t + (1-\beta_1)\nabla_{\boldsymbol{\theta}_t} J_{\text{minibatch}}(\boldsymbol{\theta}_t) \\
\mathbf{v}_{t+1} &\leftarrow \beta_2 \mathbf{v}_t + (1-\beta_2)(\nabla_{\boldsymbol{\theta}_t} J_{\text{minibatch}}(\boldsymbol{\theta}_t) \odot \nabla_{\boldsymbol{\theta}_t} J_{\text{minibatch}}(\boldsymbol{\theta}_t)) \\
\boldsymbol{\theta}_{t+1} &\leftarrow \boldsymbol{\theta}_t - \alpha \mathbf{m}_{t+1}/\sqrt{\mathbf{v}_{t+1}}
\end{aligned}$$

where $\odot$ and / denote elementwise multiplication and division (so $\mathbf{z}\odot\mathbf{z}$ is elementwise squaring) and β_2 is a hyperparameter between 0 and 1 (often set to 0.99). Since Adam divides the update by $\sqrt{\mathbf{v}}$, which of the model parameters will get larger updates? Why might this help with learning?

Answer:

Since Adam divides by the update by $\sqrt{\mathbf{v}}$, model parameters **with smaller gradient magnitudes in the latest minibatch gradient computation** will receive larger updates. By amplifying the update the parameters with smaller (magnitude-wise) gradients receive, use of adaptive learning rates ensures that all model parameters (involved in a given minibatch of data) are updated to a similar extent. This form of update "equalization" helps with learning as the resulting parameter update vectors will guide the model parameters toward a configuration which is more of a local optimum with respect to all model parameters (and their associated features) rather than just some, meaning the final model will (hopefully) generalize better on data with different feature-wise prominence/presence than the training data.

(b) (4 points) Dropout[3] is a regularization technique. During training, dropout randomly sets units in the hidden layer $\mathbf{h}$ to zero with probability p_{drop} (dropping different units each minibatch), and then multiplies $\mathbf{h}$ by a constant γ. We can write this as:

$$\mathbf{h}_{\text{drop}} = \gamma \mathbf{d} \odot \mathbf{h}$$

where $\mathbf{d} \in \{0,1\}^{D_h}$ (D_h is the size of $\mathbf{h}$) is a mask vector where each entry is 0 with probability p_{drop} and 1 with probability $(1 - p_{\text{drop}})$. γ is chosen such that the expected value of $\mathbf{h}_{\text{drop}}$ is $\mathbf{h}$:

$$\mathbb{E}_{p_{\text{drop}}}[\mathbf{h}_{\text{drop}}]_i = h_i$$

for all $i \in \{1, \dots, D_h\}$.

i. (2 points) What must γ equal in terms of p_{drop}? Briefly justify your answer or show your math derivation using the equations given above.

Answer:

γ in terms of p_{drop}:

$$\gamma = \frac{1}{(1 - p_{\text{drop}})}$$

Justification:

$$\begin{aligned}
\mathbb{E}_{p_{\text{drop}}}[\mathbf{h}_{\text{drop}}]_i &= h_i \\
\mathbb{E}_{p_{\text{drop}}}[\gamma \mathbf{d} \odot \mathbf{h}]_i &= h_i \\
\gamma * \mathbb{E}_{p_{\text{drop}}}[\mathbf{d} \odot \mathbf{h}]_i &= h_i \\
\gamma * [0 * p_{\text{drop}} * h + 1 * (1 - p_{\text{drop}}) * h]_i &= h_i \\
\gamma * (1 - p_{\text{drop}}) * h_i &= h_i \\
\gamma * (1 - p_{\text{drop}}) &= 1 \\
\gamma &= \frac{1}{1 - p_{\text{drop}}}
\end{aligned}$$

[3]Srivastava et al., 2014, `https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf`

ii. (2 points) Why should dropout be applied during training? Why should dropout **NOT** be applied during evaluation? (Hint: it may help to look at the paper linked above in the write-up.)

Answer:

Dropout **should** be applied during training because it encourages the model to update parameters such that information can be extracted from individual features even in the absence of other features (i.e., features become useful on their own), helping the model to generalize when operating on data with different feature-wise prominence/presence (and combinations thereof) than the training data. Dropout **should not** be applied during evaluation because during evaluation, we want our model to leverage all of the information available in the test data to achieve the best possible result.

2. Neural Transition-Based Dependency Parsing (46 points)

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

Before you begin, please follow the README to install all the needed dependencies for the assignment. We will be using PyTorch 1.13.1 from `https://pytorch.org/get-started/locally/` with the `CUDA` option set to `None`, and the tqdm package – which produces progress bar visualizations throughout your training process. The official PyTorch website is a great resource that includes tutorials for understanding PyTorch's Tensor library and neural networks.

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:

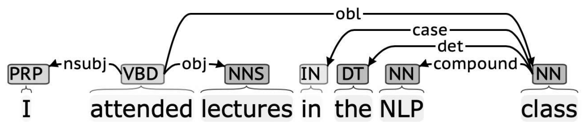
- A *stack* of words that are currently being processed.
- A *buffer* of words yet to be processed.
- A list of *dependencies* predicted by the parser.

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.

(a) (4 points) Go through the sequence of transitions needed for parsing the sentence *"I attended lectures in the NLP class"*. 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, attended, lectures, in, the, NLP, class]		Initial Config.
[ROOT, I]	[attended, lectures, in, the, NLP, class]		`SHIFT`
[ROOT, I, attended]	[lectures, in, the, NLP, class]		`SHIFT`
[ROOT, attended]	[lectures, in, the, NLP, class]	attended→I	`LEFT-ARC`

Answer:

Stack	Buffer	New dependency	Transition
[ROOT]	[I, attended, lectures, in, the, NLP, class]		Initial Config.
[ROOT, I]	[attended, lectures, in, the, NLP, class]		`SHIFT`
[ROOT, I, attended]	[lectures, in, the, NLP, class]		`SHIFT`
[ROOT, attended]	[lectures, in, the, NLP, class]	attended→I	`LEFT-ARC`
[ROOT, attended, lectures]	[in, the, NLP, class]		`SHIFT`
[ROOT, attended]	[in, the, NLP, class]	attended→lectures	`RIGHT-ARC`
[ROOT, attended, in]	[the, NLP, class]		`SHIFT`
[ROOT, attended, in, the]	[NLP, class]		`SHIFT`
[ROOT, attended, in, the, NLP]	[class]		`SHIFT`
[ROOT, attended, in, the, NLP, class]	[]		`SHIFT`
[ROOT, attended, in, the, class]	[]	class→NLP	`LEFT-ARC`
[ROOT, attended, in, class]	[]	class→the	`LEFT-ARC`
[ROOT, attended, class]	[]	class→in	`LEFT-ARC`
[ROOT, attended]	[]	attended→class	`RIGHT-ARC`
[ROOT]	[]	ROOT→attended	`RIGHT-ARC`

(b) (2 points) A sentence containing n words will be parsed in how many steps (in terms of n)? Briefly explain in 1–2 sentences why.

Answer:

A sentence containing n words will be parsed in $2n$ steps - for the parsing to be complete, each word will need to be `SHIFT`'ed from the buffer onto the stack exactly once (leading to $1 * n = n$ steps), and each word will also need to be `LEFT/RIGHT-ARC`'ed off of the stack exactly once (leading to an additional $1 * n = n$ steps, giving a total of $n + n = 2n$ steps). If we would like to count the parser initialization as a step, then $2n + 1$ steps would be required.

(c) (6 points) 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`.

Answer: **COMPLETED** – *see code submission.*

(d) (8 points) 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 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_d`.

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.

Answer: **COMPLETED** – *see code submission.*

(e) (12 points) 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*.[4] 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 $\mathbf{w} = [w_1, w_2, \dots, w_m]$ 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). Then our network looks up an embedding for each word and concatenates them into a single input vector:

$$\mathbf{x} = [\mathbf{E}_{w_1}, ..., \mathbf{E}_{w_m}] \in \mathbb{R}^{dm}$$

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:

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

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.

We will use UAS score as our evaluation metric. UAS refers to Unlabeled Attachment Score, which is computed as the ratio between 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 Penn Treebank (annotated with Universal Dependencies).

Note:

- For this assignment, you are asked to implement Linear layer and Embedding layer. Please **DO NOT** use **torch.nn.Linear** or **torch.nn.Embedding** module in your code, otherwise you will receive deductions for this problem.
- Please follow the naming requirements in our TODO if there are any, e.g. if there are explicit requirements about variable names you have to follow them in order to receive full credits. You are free to declare other variable names if not explicitly required.

[4] Chen and Manning, 2014, `https://nlp.stanford.edu/pubs/emnlp2014-depparser.pdf`

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 ($\mathbf{W}$, $\mathbf{b}_1$, $\mathbf{U}$, $\mathbf{b}_2$).
- 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 (e)` or `forward (f)` 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. 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 the training dataset, i.e., when debug mode is disabled.
- When 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).

Deliverables:

- Working implementation of the transition mechanics that the neural dependency parser uses in `parser_transitions.py`.
- Working implementation of minibatch dependency parsing in `parser_transitions.py`.
- Working implementation of the neural dependency parser in `parser_model.py`. (We'll look at and run this code for grading).
- Working implementation of the functions for training in `run.py`. (We'll look at and run this code for grading).
- **Report the best UAS your model achieves on the dev set and the UAS it achieves on the test set in your writeup**.

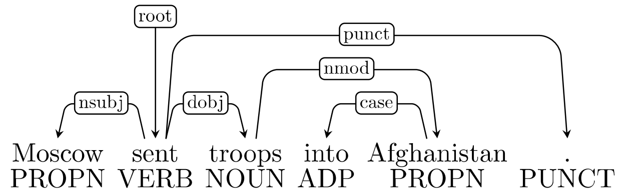
Answer:

- Best UAS achieved on dev. set: 88.63
- UAS achieved on test set: 88.94

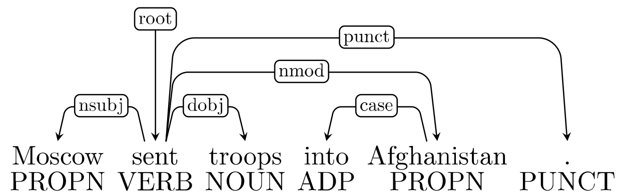
```
Epoch 10 out of 10
100%|
Average Train Loss: 0.06569825583520583
Evaluating on dev set
1445850it [00:00, 29509856.05it/s]
- dev UAS: 88.63
New best dev UAS! Saving model.

================================================================================
TESTING
================================================================================
Restoring the best model weights found on the dev set
Final evaluation on test set
2919736it [00:00, 28906619.36it/s]
- test UAS: 88.94
Done!
```

Figure 1: Screenshot of results.

(f) (12 points) 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, unless there are somehow weirdly some troops that stan Afghanistan). 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[5]. 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[6]. 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.*

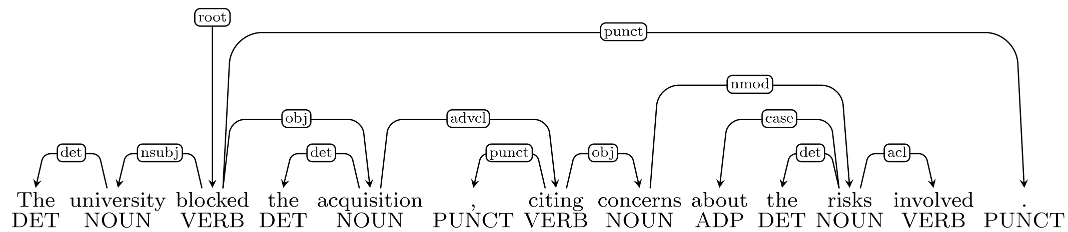
In this question are four sentences with dependency parses obtained from a parser. Each sentence has one error type, and there is one example of each of the four types above. For each sentence, state the type of error, the incorrect dependency, and the correct dependency. While each sentence should have 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

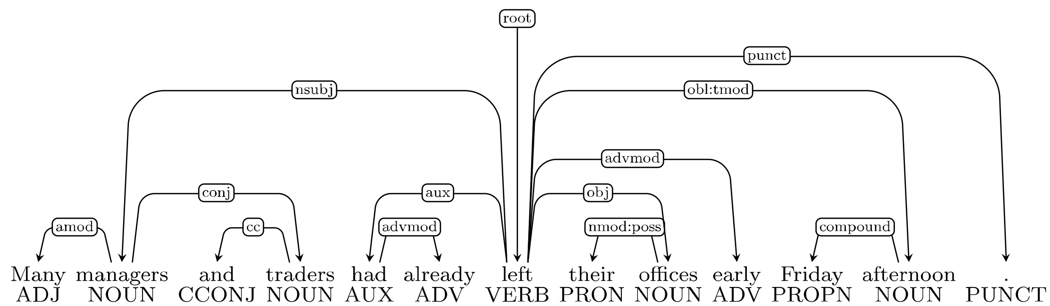
[5] For examples of prepositional phrases, see: https://www.grammarly.com/blog/prepositional-phrase/
[6] For examples of verb phrases, see: https://examples.yourdictionary.com/verb-phrase-examples.html

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`[7] *or the short introductory slides at:* `http://people.cs.georgetown.edu/nschneid/p/UD-for-English.pdf`*. Note that 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.*

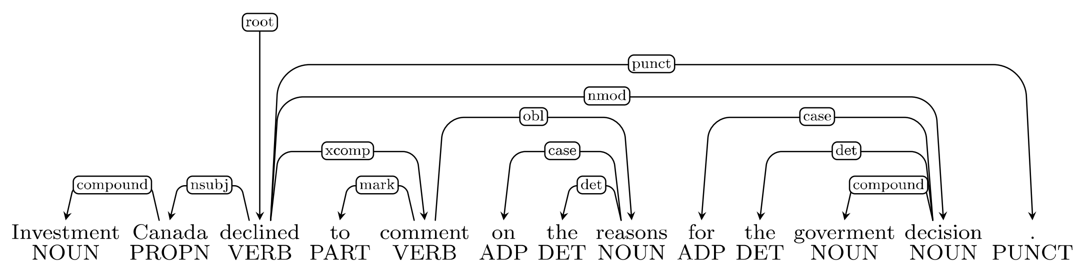
i.



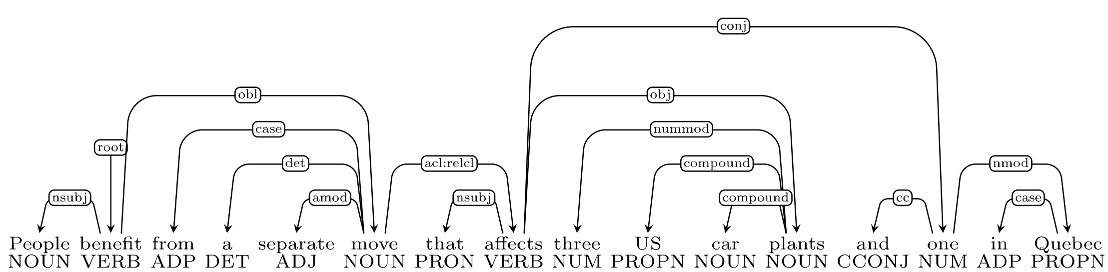
ii.



iii.



iv.



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

Answer:

i. Sentence: *"The university blocked the acquisition, citing concerns about the risks involved."*
   - **Error type:** Verb Phrase Attachment Error.
   - **Incorrect dependency:** acquisition $\rightarrow$ citing.
   - **Correct dependency:** blocked $\rightarrow$ citing.

ii. Sentence: *"Many managers and traders had already left their offices early Friday afternoon."*
   - **Error type:** Modifier Attachment Error.
   - **Incorrect dependency:** left $\rightarrow$ early.
   - **Correct dependency:** afternoon $\rightarrow$ early.

iii. Sentence: *"Investment Canada declined to comment on the reasons for the government decision."*
   - **Error type:** Prepositional Phrase Attachment Error.
   - **Incorrect dependency:** declined $\rightarrow$ decision.
   - **Correct dependency:** reasons $\rightarrow$ decision.

iv. Sentence: *"People benefit from a separate move that affects three US car plants and one in Quebec"*
   - **Error type:** Coordination Attachment Error.
   - **Incorrect dependency:** affects $\rightarrow$ one.
   - **Correct dependency:** plants $\rightarrow$ one.

(g) (2 points) Recall in part (e), the parser uses features which includes words and their part-of-speech (POS) tags. Explain the benefit of using part-of-speech tags as features in the parser?

Answer:

A given word may be able to be employed as multiple POS's, with the specific POS use case being dependent on the surrounding sentence construction (e.g., "swimming" can be used as a verb, as in "He is swimming," or as a noun, as in "Swimming is fun!"). Then, using POS tags as features can be beneficial for a parsing model since, for these multiple potential POS words, the model can learn specific parameters for each POS use case of a given word, thereby enabling the model to parse said word more accurately across all of its potential POS use cases.

Submission Instructions You shall submit this assignment on GradeScope as two submissions – one for "Assignment 3 [coding]" and another for 'Assignment 3 [written]":

1. Run the `collect_submission.sh` script to produce your `assignment3.zip` file.
2. Upload your `assignment3.zip` file to GradeScope to "Assignment 3 [coding]".
3. Upload your written solutions to GradeScope to "Assignment 3 [written]".