CS 224n Assignment #3: Dependency Parsing

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.

1. The momentum term $\mathbf{m}$ behaves as an expotiential moving average of the gradients, which reduces the variance of the parameter updates and helps to maintain a consistent direction towards the minimum.

2. SGD with momentum helps learning by smoothing out the updates to the model parameters. This smoothing effect helps the optimizer to avoid getting stuck in local minima and saddle points, and to converge faster to the global minimum of the loss landscape.

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:

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

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?

---

[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

1. **Question 1:** The parameters with samller $\sqrt{v_{t+1}}$ will get larger updates. In other words, the parameters with smaller moving average of the squared gradients will get larger updates.

2. **Question 2:** Adam uses adaptive learning rates to adaptively adjust the learning rate for each parameter. This helps to speed up the learning process by making larger updates to the parameters with smaller gradients, and smaller updates to the parameters with larger gradients. And because of the adaptive parameter updates, Adam usually converges faster than SGD with momentum, and is less sensitive to the choice of hyperparameters.

(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.

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

Therefore, $\gamma = 1/(1 - p_{\text{drop}})$. This ensures that the expected value of the activations is the same during training and evaluation.

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.)

1. Dropout should be applied during training to prevent overfitting and improve the generalization performance of the model.

2. By randomly dropping out units in each training iteration, dropout forces the remaining units to learn more robust and diverse features that are not dependent on the presence of any single unit.

3. Dropout also acts as a form of model averaging and ensembling, since the model is trained on different subsets of the units in each iteration.

4. Dropout should not be applied during evaluation because it would cause the model to make inconsistent and non-deterministic predictions for the same input.

---

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

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 → 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 → 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.
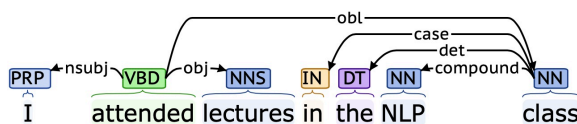
Stack	Buffer	New dependency	Transiti...
[ROOT]	[I, attended, lectures, in, the, NLP, class]		Initial C
[ROOT, I]	[attended, lectures, in, the, NLP, class]		SHIFT
[ROOT, I, attended]	[lectures, in, the, NLP, class]		SHIFT
[ROOT, attended]	[lectures, in, the, NLP, class]	attended→I	LEFT-A
[ROOT, attended, lectures]	[in, the, NLP, class]		SHIFT
[ROOT, attended]	[in, the, NLP, class]	attended→lectures	RIGHT-
[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-A
[ROOT, attended, in, class]	[]	class→the	LEFT-A
[ROOT, attended, class]	[]	class→in	LEFT-A
[ROOT, attended]	[]	attended→class	RIGHT-
[ROOT]	[]	ROOT→attended	RIGHT-

(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.

A transition-based parser needs to perform $2n$ (exclude the initial configuration step) steps to parse a sentence containing n words. This is because each word requires two steps: a SHIFT operation to add it to the stack, and a REDUCE (LEFT-ARC or RIGHT-ARC) operation to combine it with other words in the stack.

(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.

(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.*

(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 $\mathbf{w} = [w_1, w_2, \ldots, w_m]$ 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). 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:

$$\mathbf{h} = \text{ReLU}(\mathbf{x}\mathbf{W} + \mathbf{b}_1)$$
$$\mathbf{l} = \mathbf{h}\mathbf{U} + \mathbf{b}_2$$
$$\hat{\mathbf{y}} = \text{softmax}(l)$$

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.

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).

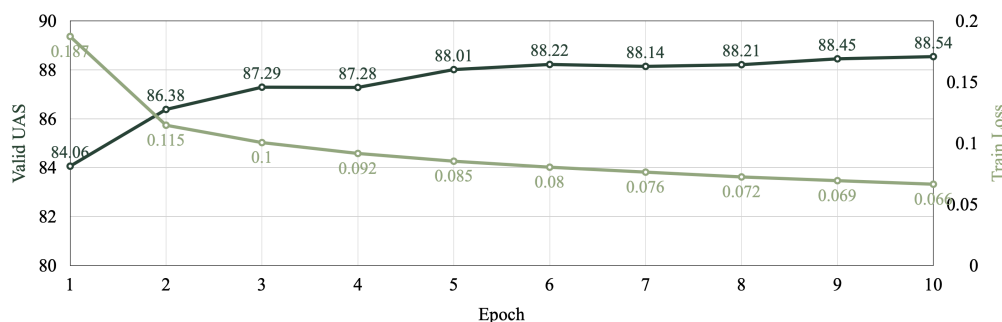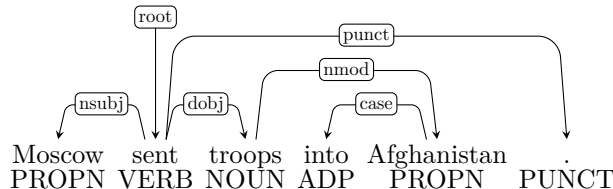- **Report the best UAS your model achieves on the dev set and the UAS it achieves on the test set in your writeup**.

Figure 1: Cost Curve

The best UAS on the dev set is **88.54**, and the best UAS on the test set is **88.69**.

(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:
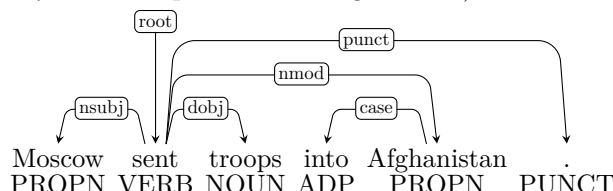
More generally, here are four types of parsing error:

- **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:
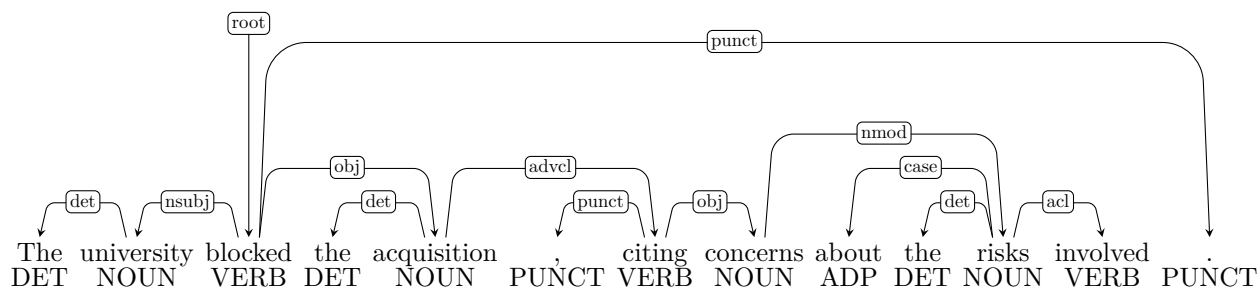
- **Error type**: Prepositional Phrase Attachment Error
- **Incorrect dependency**: troops → Afghanistan
- **Correct dependency**: sent → Afghanistan
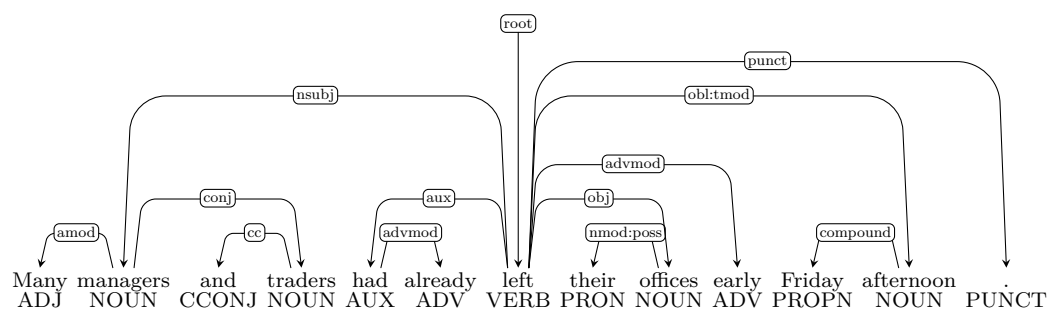
---

[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.*
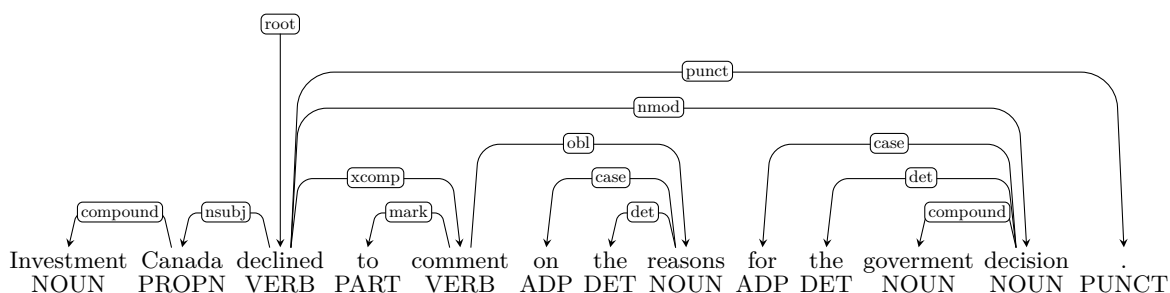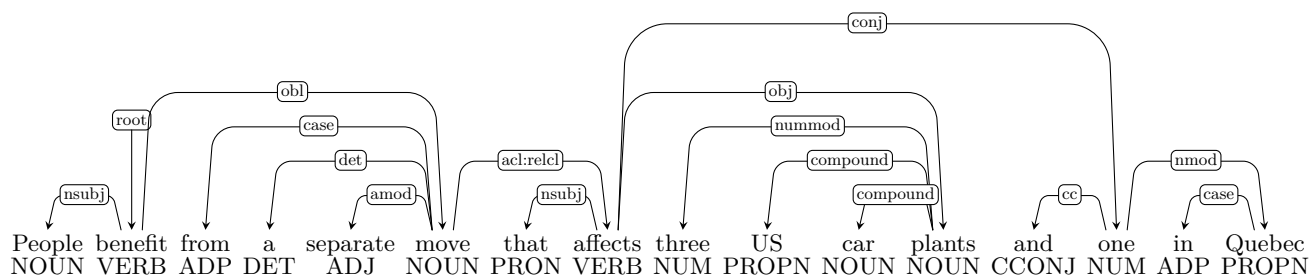
i.

The university blocked the acquisition , citing concerns about the risks involved .
DET NOUN VERB DET NOUN PUNCT VERB NOUN ADP DET NOUN VERB PUNCT

ii.

Many managers and traders had already left their offices early Friday afternoon .
ADJ NOUN CCONJ NOUN AUX ADV VERB PRON NOUN ADV PROPN NOUN PUNCT

iii.

Investment Canada declined to comment on the reasons for the goverment decision .
NOUN PROPN VERB PART VERB ADP DET NOUN ADP DET NOUN NOUN PUNCT

iv.

___

1. • **Error type**: Verb Phrase Attachment Error
    • **Incorrect dependency**: acquisition → citing
    • **Correct dependency**: blocked → citing

2. • **Error type**: Modifier Attachment Error
    • **Incorrect dependency**: left → early
    • **Correct dependency**: afternoon → early

3. • **Error type**: Prepositional Phrase Attachment Error
    • **Incorrect dependency**: declined → decisions
    • **Correct dependency**: reasons → decision

4. • **Error type**: Coordination Attachment Error:
    • **Incorrect dependency**: affects → one
    • **Correct dependency**: plants → 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?

1. **Improved accuracy**: POS tags provide more information about the input, which can help the parser to make better predictions. For example, if we know that a word is a noun, we can predict that it is more likely to be the head of a verb than the head of a preposition.

2. **Reduced ambiguity**: POS tags reduce the ambiguity of words in context. For example, bank can be a noun or a verb, but if we know that it is a noun, we can reduce the number of possible meanings of the word.

3. **Efficient computation**: POS tags reduce the number of features that need to be considered. For example, if we know that a word is a noun, we can ignore features that are only relevant to verbs.

4. **Improved generalization**: POS tags provide a more abstract representation of the input than raw words, which can help the parser to generalize better to new, unseen data. For example, if we know that a word is a noun, we can generalize to other nouns that we have not seen before.

Overall, using POS tags as features gives the parser more information about the input, which can help the parser to achieve higher accuracy and efficiency, and to better capture the syntactic structure of the input.

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]".