Natural Language Processing
A4: Dependency Parsing

Due: Tuesday, May 23 (by midnight)

Submission: Each group should upload a .zip file to Canvas with:

1. All code and output files (named as instructed below);
2. A PDF with all written responses and a brief description of how each group member contributed to the assignment.

Note on groups: You will be working in your same group as for A2. If your group members were absent, or if you have any other issues with finding a full group of 3 people, please let us know ASAP. Last minute requests to find group members cannot be easily accommodated.

Starter code: A4.zip

In this assignment, you will build a neural dependency parser using PyTorch. If you are new to PyTorch, please allocate the first few days to understanding its functionality, as learning how to use PyTorch is part of the objective of this assignment. Here is a notebook to help you: **PyTorch tutorial**[1]. The official PyTorch website is also a great resource that includes tutorials for understanding PyTorch's Tensor library and neural networks.

Grading: The assignment is worth 100 points, distributed as follows (more detailed point distribution in text below). Note: Parts 3 and Bonus are not dependent on Parts 1-2 and may be completed prior/in parallel:

- **Part 1:** Learn about general neural network techniques and explain them in your own words. (15 points)
- **Part 2:** Implement and train a dependency parser. This part will focus on applying what you learn in Part 1 and is heavily focused on coding. (50 points)
- **Part 3:** Analyze erroneous dependency parses. This part will focus more on linguistic analysis and a high-level understanding of dependency parsing. (35 points)
- **Bonus:** Cross-lingual dependency parsing. (up to 10 points)

Allocating your time: You have two weeks to complete this assignment. Start early! We recommend you take the first few days to become familiar with PyTorch and work on Part 1. Part 2 will likely take a full week of daily work to run successfully. The last few days can then be spent on Part 3, analysis, and the bonus, if you choose. Though the assignment may look like a lot, it is all very doable if you spend enough time with it. We also understand that different group members have different strengths (i.e. coding, linguistics, math). This assignment uses all skills and is best completed working together.

1. Machine Learning & Neural Networks (15 points)

(a) (5 points) **Stochastic Gradient Descent.**
As we saw briefly in class, stochastic gradient descent is an optimization algorithm for minimizing the loss of a predictive model with regard to a training dataset. To review how neural networks learn using gradient descent, please watch this video from 3Blue1Brown. In 3-4 sentences, please explain *stochastic* gradient descent and how it could be useful for an NLP task.

(b) (5 points) **Adam Optimizer**
The standard Stochastic Gradient Descent update rule states:

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

[1] `https://colab.research.google.com/drive/13HGy3-uIIy1KDWFhG4nVrxJC-3nUUkP?usp=sharing`

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[2] uses a more sophisticated update rule with two additional steps.[3]

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

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

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.

ii. (3 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?

(c) (5 points) Dropout[4] 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 $i \in \{1, \ldots, D_h\}$.

i. (2 points) Why should dropout be applied during training?

ii. (3 points) Why should dropout **NOT** be applied during evaluation? (Hint: it may help to look at the paper linked above in the write-up.)

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

[3] 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`

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

2. Neural Transition-Based Dependency Parsing (50 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.

As we saw in class, 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:

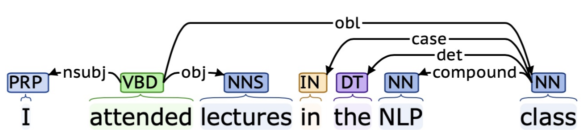
- A *stack* of words that are currently being processed.
- A *buffer* of words yet to be processed.
- A list of *dependencies* predicted by the parser.

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 Configuration
[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$\rightarrow$I	`LEFT-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.

(c) (2 points) Inspect the data you will be using manually. What format is it in? How is it separated? Explain how the data is labeled already for your use.

(d) (4 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`.

(e) (6 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.

(f) (28 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*.[5] 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

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

then compute our prediction as:

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

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 $\mathrm{ReLU}(z) = \max(z, 0)$). 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.

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).

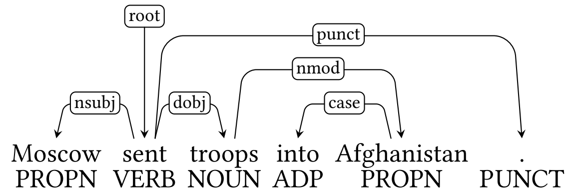
(g) (4 points) Report the best UAS your model achieves on the dev set and the UAS it achieves on the test set in your write-up. Why is UAS a useful metric for evaluating dependency parsing? Cite specific examples where the UAS metric does and does not provide meaningful insight into the performance of your parser.

Deliverables:

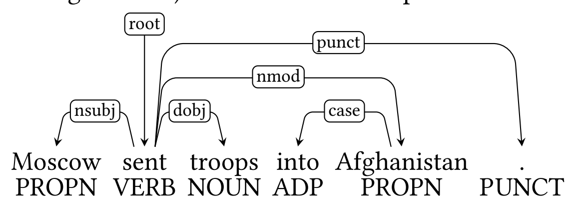
- Working implementation of the transition mechanics that the neural dependency parser uses in `parser_transitions.py`.
- Working implementation of minibatch dependency parsing in `parser_transitions.py`.
- Working implementation of the neural dependency parser in `parser_model.py`. (We'll look at and 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).
- Write-up with all answers to above questions.

3. Error Analysis (30 points)

(a) (16 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[6]. 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[7]. 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*.

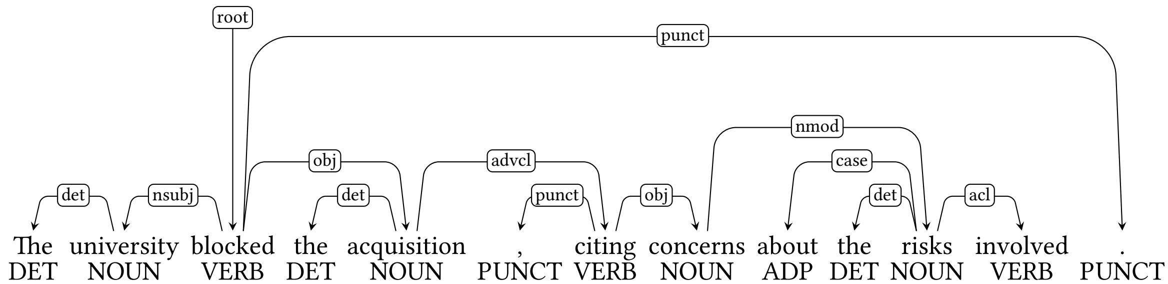
Below are four sentences with dependency parses obtained from a parser. Each sentence has one error type, and there is one example of each of the four types above. For each sentence, state the type of error, the incorrect dependency, and the correct dependency. While each sentence should have a unique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

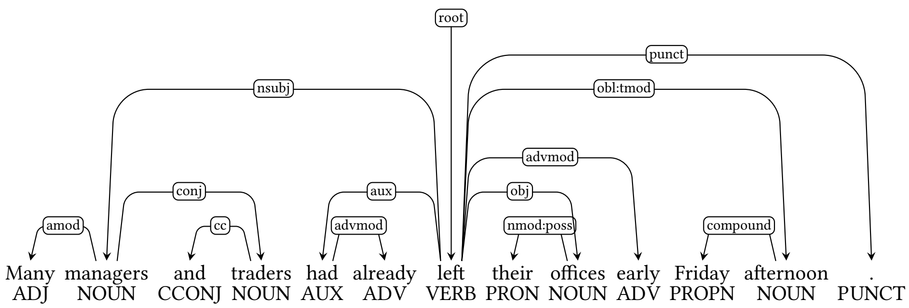
[6] For examples of prepositional phrases, see: https://www.grammarly.com/blog/prepositional-phrase/

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

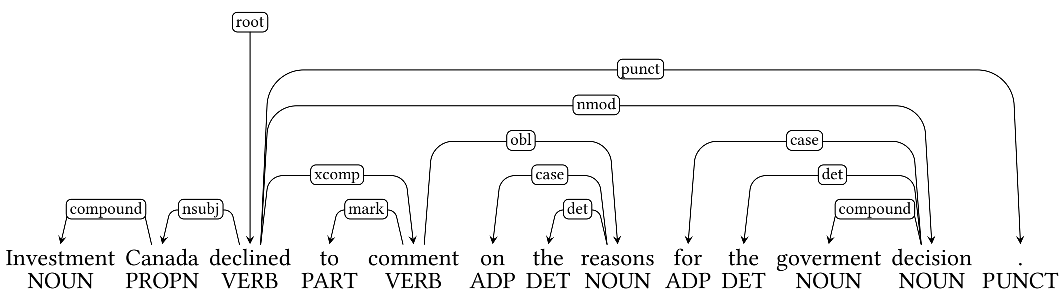
i.



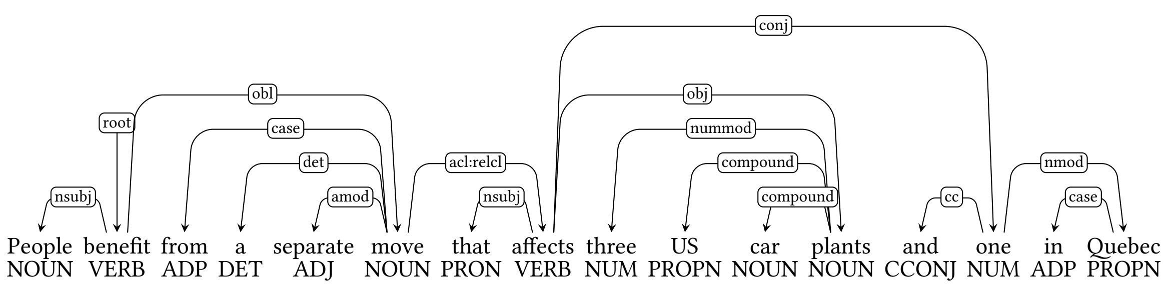
ii.



iii.



iv.



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

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

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. You* ***do not*** *need to look at the labels on the the dependency edges (though you may do so) – it suffices to just look at the edges themselves.*

(b) (10 points) In class, we saw an example of a *garden-path sentence*, or a grammatically correct sentence that starts in such a way that a listener's or reader's likely parses the structure incorrectly. The sentence we discussed was "*The horse raced past the barn fell*". Using an online dependency parser (e.g. CoreNLP), show (i) the initial "garden path" parse structure computed before the parse is corrected, and (ii) the final, correct parser of the sentence. Having now worked with a transition-based dependency parser, how do you think it would handle such sentences?

(c) (4 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. You may cite examples from your parser and garden-path sentences as support.

4. Bonus: Cross-lingual dependency parsing (10 points)

From the UD website: "Universal Dependencies (UD) is a framework for consistent annotation of grammar (parts of speech, morphological features, and syntactic dependencies) across different human languages. UD is an open community effort with over 300 contributors producing nearly 200 treebanks in over 100 languages."

Choose a language you are familiar with other than English, and explore how the four types of parsing errors introduced in (3a) may be challenges in cross-lingual dependency parsing. To do this, follow these steps:

- Choose an example English sentence that demonstrates each error type. You may use sentences from this assignment or come up with your own.
- Translate these sentences into the language you are working with. You may use an automatic translator or translate the sentence yourself. Explain your translation process.
- Run a dependency parser for your language on the translated sentences. For common NLP languages, online demos like CoreNLP may support your language. For other languages, you may need to install a new parser: see the Universal Dependencies Website for supported UD languages.
- Assess how the challenges of English dependency parsing using a transition-based parser translate into the language you have chosen. Are the challenges the same? Are there new challenges that English may not face? Feel free to add additional examples that may not be directly translatable from English.