University of Toronto, Department of Computer Science
CSC 485H/2501H: Computational linguistics, Fall 2025

Assignment 1

Due date: 17:00 on Monday, October 6, 2025.
Late assignments will not be accepted without a valid medical certificate or other documentation of an emergency.
This assignment is worth 30% (CSC 485) or 25% (CSC 2501) of your final grade.

- **Read the whole assignment carefully.**
- Type the written parts of your submission in no less than 12pt font.
- What you turn in must be your own work. You may not work with anyone else on any of the problems in this assignment. If you need assistance, contact the instructor or TA for the assignment.
- Any clarifications to the problems will be posted on the Piazza forum for the class. You will be responsible for taking into account in your solutions any information that is posted there, or discussed in class, so you should check the page regularly between now and the due date.
- The starter code directory for this assignment is distributed via MarkUs. In this handout, code files we refer to are located in that directory.
- When implementing code, make sure to **read the docstrings carefully** as they provide important instructions and implementation details.
- Fill in your name, student number, and UTORid on the relevant lines at the top of each Python file that you submit. (Do not add new lines; just replace the `NAME`, `NUMBER`, and `UTORid` placeholders.)

1. Transition-based dependency parsing (42 marks)

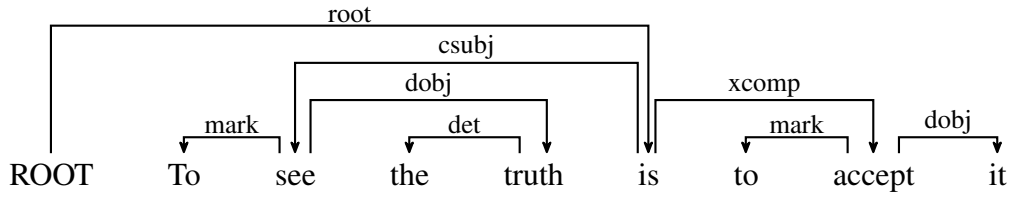
Dependency grammars posit relationships between "head" words and their modifiers. These relationships constitute trees where each word depends on exactly one parent: either another word or, for the head of the sentence, a dummy symbol, "ROOT". The first part of this assignment concerns a parser that builds a parse incrementally. At each step, the state of the parse is represented by:

- 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 advances by applying a *transition* to the parse until its buffer is empty and the stack is of size 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-from-top item (i.e., second-most recently added word) on the stack as a dependant of the first item and removes the second item from the stack.
- RIGHT-ARC: marks the top item (i.e., most recently added word) on the stack as a dependant of the second item and removes the first item from the stack.

(a) (4 marks) Complete the sequence of transitions needed for parsing the sentence "To see the truth is to accept it" with the dependency tree shown below. At each step, indicate the state of the stack and the buffer, as well as which transition to apply at this step, including what, if any, new dependency to add. The first four steps are provided to get you started.



Step	Stack	Buffer	New dep	Transition
0	[ROOT]	[To, see, the, truth, is, to, accept, it]		
1	[ROOT, To]	[see, the, truth, is, to, accept, it]		SHIFT
2	[ROOT, To, see]	[the, truth, is, to, accept, it]		SHIFT
3	[ROOT, see]	[the, truth, is, to, accept, it]	see $\overset{mark}{\rightarrow}$ To	LEFT-ARC
4	[ROOT, see, the]	[truth, is, to, accept, it]		SHIFT

Algorithm 1: Minibatch dependency parsing

input : a list of `sentences` to be parsed and a `model`, which makes parser decisions.

Initialize a list of `partial_parses`, one for each sentence in `sentences`;
Initialize a shallow copy of `partial_parses` called `unfinished_parses`;
while `unfinished_parses` is not empty **do**
    Use 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 those parses that are completed from `unfinished_parses`;
end
return The `arcs` for each (now completed) parse in `partial_parses`.

(b) (1 mark) A sentence containing n words will be parsed in how many steps, in terms of n? (Exact, not asymptotic.) Briefly explain why.

Next, you will implement a transition-based dependency parser that uses a neural network as a classifier to decide which transition to apply at a given *partial parse* state. A partial parse state is collectively defined by a sentence buffer as above, a stack as above with any number of items on it, and a set of correct dependency arcs for the sentence.

(c) (6 marks) Implement the `complete` and `parseStep` methods in the `PartialParse` class in `q1_parse.py`. These implement the transition mechanism of the parser. Also implement `getNRightMost` and `getNLeftMost`. You can run basic (non-exhaustive) tests by running `python3.12 run_test.py q1-c`.

(d) (6 marks) Our network will predict which transition should be applied next to a partial parse. In principle, we could use the network to parse a single sentence simply by applying predicted transitions until the parse is complete. However, neural networks run much more efficiently when making predictions about "minibatches" of data at a time; in this case, that means predicting the next transition for many different partial parses simultaneously. We can parse sentences in minibatches according to algorithm 1.

Implement this algorithm in the `minibatchParse` function in `q1_parse.py`. You can run basic (non-exhaustive) tests by running `python3.12 run_test.py q1-d`.

Training your model to predict the right transitions will require you to have a notion of how well it performs on each training example. The parser's ability to produce a good dependency tree for a sentence is measured using an **attachment score**. This is the percentage of words in the sentence that are assigned as a dependant of the correct head. The unlabelled attachment score (**UAS**) considers only this, while the labelled attachment score (**LAS**) considers the label on the dependency relation as well. While this is ultimately the score we want to maximize, it is difficult to use this score to improve our model on a continuing basis.

Instead, we will use the model's per-transition accuracy (per partial parse) as a proxy for the parser's attachment score, so we will need the correct transitions. But the data set just provides sentences and corresponding dependency arcs. You must therefore implement an *oracle* that, given

a partial parse and a set of correct, final target dependency arcs, provides the next transition to take to advance the partial parse towards the final solution set of dependencies. The classifier will later be trained to try to predict the correct transition by minimizing the error in its predictions versus the transitions provided by the oracle.

(e) (12 marks) Implement your oracle in the `getOracle` method of `q1_parse.py`. You can run basic tests by running `python3.12 run_test.py q1-e`.

The last step is to construct and train a neural network to predict which transition should be applied next, given the state of the stack, buffer, and dependencies. First, the model extracts a feature vector representing the current state. The function that extracts the features that we will use has been implemented for you in `data.py`. This feature vector consists of a list of tokens (e.g., the last word in the stack, first word in the buffer, dependant of the second-to-last word in the stack if there is one, etc.). They can be represented as a list of integers:

$$[w_1, w_2, \ldots, w_m]$$

where m is the number of features and each $0 \leq w_i < |V|$ is the index of a token in the vocabulary ($|V|$ is the vocabulary size). Using pre-trained *word embeddings* (i.e., word vectors), our network will first look up the embedding for each word and concatenate them into a single input vector:

$$x_w = [L_{w_0}\,;\, L_{w_1}\,;\, \ldots\,;\, L_{w_m}] \in \mathbb{R}^{dm}$$

where $L \in \mathbb{R}^{|V| \times d}$ is an embedding matrix where each row L_i is the vector for the i-th word and the semicolon represents concatenation. The embeddings for tags (x_t) and arc-labels (x_l) are generated in a similar fashion from their own embedding matrices. The three vectors are then concatenated together:

$$x = [x_w\,;\, x_t\,;\, x_l]$$

From the combined input, we then compute our prediction probabilities as:

$$h = \max\left(xW_h + b_h, 0\right)$$
$$\hat{y} = \text{softmax}\left(hW_o + b_o\right).$$

The function for h is called the rectified linear unit (ReLU) function: $\text{ReLU}(z) = \max(z, 0)$.

The objective function for this network (i.e., the value we will aim to minimize) with parameters θ is the cross-entropy loss:

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

To compute the loss for the training set, we average this $J(\theta)$ across all training examples.

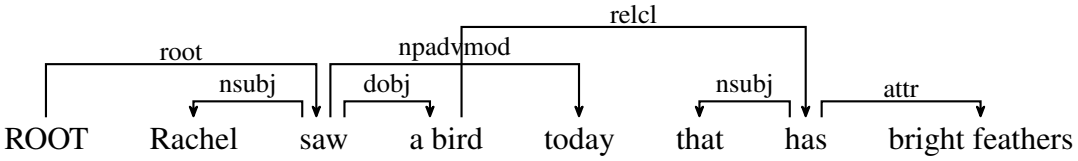
(f) (13 marks) In `q1_model.py`, implement the neural network classifier governing the dependency parser by filling in the appropriate sections; they are marked by `ENTER YOUR CODE BELOW`.

You will train and evaluate the model on a corpus of English Web text that has been annotated with Universal Dependencies. Run `python3.12 train.py q1` to train your model and compute predictions on the test data. With everything correctly implemented, you should be able to get an LAS of *around* 80% on both the validation and test sets (with the best-performing model out of all of the epochs at the end of training).

(g) **Bonus** (3 mark). In this question, you will explore whether recent large language models can do dependency parsing in a zero-shot or few-shot manner. Choose any 3 LLMs of your choice. Compare the dependency parse of your model and the LLMs using the sentence "To raise those doubts is to resolve them." What prompt did you use for your experiment? Based on your observations, evaluate the performance of using an LLM and your own model qualitatively. Are there certain advantages and disadvantages you can attribute to each system?

2. Graph-based dependency parsing (58 marks)

The transition-based mechanism above is limited to only being able to parse *projective* dependency trees. A *projective* dependency tree is one in which the edges can be drawn above the words without crossing other edges when the words, preceded by ROOT, are arranged in linear order. Equivalently, every word forms a contiguous substring of the sentence when taken together with its descendants. The tree in the above figure was projective. The tree below is not projective.



(a) (3 marks) Why is the parsing mechanism described above insufficient to generate non-projective dependency trees?

(b) (8 marks) Implement the `isProjective` function in `q2_algorithm.py`. You can run some basic tests based on the trees in the handout by running `python3.12 q2_algorithm.py`.

Run the `python3.12 run_test.py q2` script and report the result.

(c) (4 marks) A related concept to projectivity is *gap degree*. The *gap degree* of a word in a dependency tree is the least k for which the subsequence consisting of the word and its descendants (both direct and indirect) is entirely comprised of $k+1$ maximally contiguous substrings. Equivalently, the gap degree of a word is the *number* of gaps in the subsequence formed by the word and all of its descendants, regardless of the *size* of the gaps. The gap degree of a dependency tree is the greatest gap degree of any word in the tree.

What is the gap degree of each of the two trees above? Show your work.

Unlike the transition-based parser above, in this part of the assignment you will be implementing a *graph-based* parser that doesn't have the same limitation for non-projective trees. In particular, you will be implementing an *edge-factored* parser, which learns to score *possible* edges such that the correct edges should receive higher scores than incorrect edges. A sentence containing n words has a dependency graph with $n+1$ vertices (with the one extra vertex being for ROOT); there are therefore $(n+1)^2$ possible edges.

In this assignment, the vertices for the sentence words (including ROOT) will each be represented as vectors of size p. The vertices for the entire sentence can then be represented as a matrix $S \in \mathbb{R}^{(n+1)\times p}$.[1] Assume that ROOT is placed before the first word in the sentence.

The model you are to implement has two components: an *arc* scorer and a *label* scorer. The arc scorer produces a single score for each vertex pair; the label scorer produces $|R|$ scores for each vertex pair, where R is the set of dependency relations (det, dobj, etc.). The arc scores allow selecting an edge, after which the label score can be used to select a dependency relation for that edge.

Let a_{ij} denote the arc score for dependency edge *from vertex j to vertex i*. **Note the ordering!** It may be different from what you're used to: a_{ij} is a score corresponding to j being the ***head*** of i; in other words, a_{ij} corresponds to an edge ***from*** vertex j ***to*** vertex i. Then the matrix A with elements $a_{ij} = [A]_{ij}$ is defined as:

$$
\begin{aligned}
D_{\mathrm{A}} &= \mathrm{MLP}_{\mathrm{Ad}}(S) \\
H_{\mathrm{A}} &= \mathrm{MLP}_{\mathrm{Ah}}(S) \\
a_{ij} &= [D_{\mathrm{A}}]_i W_{\mathrm{A}}([H_{\mathrm{A}}]_j)^{\mathrm{T}} + [H_{\mathrm{A}}]_j \cdot b_{\mathrm{A}}
\end{aligned}
$$

$\mathrm{MLP}_{\mathrm{a,d}}$ and $\mathrm{MLP}_{\mathrm{a,h}}$ are (separate) Multi-Layer Perceptron layers that transform the original input sentence matrix S to a smaller dimensionality $d_{\mathrm{A}} < p$. For this assignment, we will be using MLPs with one hidden layer and one output layer, both of width d_{A}, along with Rectified Linear Unit (ReLU) activation and dropout for each layer. W_{A} and b_{A} are trainable parameters.

Lastly, for each word i in the sentence, we can compute the probability of word j being its head with softmax and then use cross-entropy as the loss function J_{h} for training:

$$
\begin{aligned}
\hat{h}_i &= \mathrm{softmax}(a_{ij}) \\
J_{\mathrm{h}} &= CE(h, \hat{h}) = -\sum_{i=1}^{n} h_i \log \hat{h}_i
\end{aligned}
$$

(d) (6 marks) Implement `createArcLayers` and `scoreArcs` in `q2_model.py`. You will need to determine the appropriate dimensions for W_{A} and b_{A}, as well as how to implement the arc score calculations for the batched sentence tensors. You should be able to do this entirely with PyTorch tensor operations; using a loop for this question will result in a penalty.

(e) (1 mark) For effective, stable training, especially for deeper networks, it is important to initialize weight matrices carefully. A standard initialization strategies for ReLU layers is

[1] Beware that the vectors are computed differently compared to Part 1. There, we used word2vec; here we use BERT, a *contextual* representation model that provides a vector representation for a word in context of a sentence; in other words, the same word will have different vector representations depending on the sentence it appears in.

called Kaiming initialization[2]. For a given layer's weight matrix W of dimension $m \times n$, where m is the number of input units to the layer and n is the number of units in the layer, Kaiming initialization samples values W_{ij} from a Gaussian distribution with mean $\mu = 0$ and variance $\sigma^2 = 2/m$.

However, it is common to instead sample from a *uniform* distribution $U(a,b)$ with lower bound a and upper bound b; in fact, the `create*` methods in `q2_model.py` direct you to initialize some parameters according to a uniform distribution. Derive the values of a and b that yield a uniform distribution $U(a,b)$ with the mean and variance given above.

The label scorer proceeds in a similar manner, but is a bit more complex. Let $l_{ij} \in \mathbb{R}^{|R|}$ denote a *vector* of scores, one for each possible dependency relation $r \in R$. We use similar MLPs as for the arc scorer (but with different dimensionality), and then compute the score vector as:

$$\begin{aligned} D_{\mathrm{L}} &= \mathrm{MLP}_{\mathrm{Ld}}(S) \\ H_{\mathrm{L}} &= \mathrm{MLP}_{\mathrm{Lh}}(S) \\ l_{ij} &= [D_{\mathrm{L}}]_i W_{\mathrm{L}}([H_{\mathrm{L}}]_j)^{\mathrm{T}} + [H_{\mathrm{L}}]_j W_{\mathrm{Lh}} + [D_{\mathrm{L}}]_i W_{\mathrm{Ld}} + b_{\mathrm{L}} \end{aligned}$$

W_{L} is a third-order tensor, W_{Lh} and W_{Ld} are matrices, and b_{L} is a vector so that l_{ij} is a vector as defined above. The MLPs for the label scorer have the same structure as for the arc scorer (two hidden layers, ReLU activations and dropout for each), but we use a smaller dimensionality $d_{\mathrm{L}} < d_{\mathrm{A}} < p$.

To compute prediction probabilities, we can again use softmax and then use cross-entropy loss for training:

$$\begin{aligned} \hat{r}_{ij} &= \mathrm{softmax}(l_{ij}) \\ J_{\mathrm{r}} = CE(r, \hat{r}) &= -\sum_{i=1}^{n} \sum_{j=0}^{n} r_{ij} \log \hat{r}_{ij} \end{aligned}$$

The two parts are trained together, such that the whole model has loss function $J = J_{\mathrm{h}} + J_{\mathrm{r}}$.

(f) (8 marks) Implement `createLabelLayers` and `scoreLabels` in `q2_model.py`. As above, you will need to determine the appropriate dimensions for the trainable parameters and how to implement the label score calculations for the batched sentence tensors. You should be able to do this entirely with PyTorch tensor operations; using a loop for this question will result in a penalty.

(g) (2 marks) In the arc scorer, the score function includes a term that has H_{A} but not D_{A}, but there isn't a term that has the opposite inclusions (D_{A} but not H_{A}). For the label scorer, both are included. Why does it not make sense to include a term just for D_{A}, but it does for D_{L}?

(h) (2 marks) In standard classification problems, we typically multiply the input by a weight matrix and add a per-class bias to produce a class score for the given input. Why do we have to multiply W_{A} and W_{L} by (transformed versions of) the input twice in this case?

[2] Also referred to as He initialization.

(i) (2 marks) There are some constraints on which arcs and arc-label combinations are possible. Implement these constraints in `maskPossible` in `q2_model.py`.

Finally, how do we make our final predictions given an input sentence? Since we know that we our desired answer has a tree structure, and since the parsing model is trained to increase the scores assigned to the correct edges, we can use a maximum spanning tree algorithm to select the set of arcs. Then, for each arc from i to j, we can use argmax over l_{ij} to predict the dependency relation.

(j) (4 marks) Why do we need to use a maximum spanning tree algorithm to make the final arc prediction? Why can't we just use argmax, like we would in a typical classification scenario, and like we do for the dependency relations? What happens if we use argmax instead?

(k) (12 marks) Finish the rest of the implementation: `isSingleRoot` and `mstSingleRoot` in `q2_algorithm.py`. You can refer to the pseudocode in the third edition of the Jurafsky & Martin textbook to guide your implementation.[3]

As in part 1, you will train and evaluate the model on a corpus of English Web text that has been annotated with Universal Dependencies. Run `python3.12 train.py q2` to train your model and compute predictions on the test data. With everything correctly implemented, you should be able to get an LAS of *around* 82% on both the validation and test sets (with the best-performing model out of all of the peochs at the end of training).[4]

(l) (6 marks) Find at least three example sentences where the transition-based parser gives a different parse than the graph-based parser, but both parses are wrong as judged by the annotation in the corpus. For each example, argue for which of the two parses you prefer.

Notes

- While debugging, you may find it useful to run your training code with the `--debug` flag, like so:

  ```
  $ python3.12 train.py --debug q1
  $ python3.12 train.py --debug q2
  ```

  This will cause the code to run over a small subset of the data so that each training epoch takes less time. Remember to change the setting back prior to submission, and before doing your final run.

- Training should run within 1–5 hours on a CPU, but may be a bit faster or slower depending on the specific CPU. A GPU is much faster, taking around 5–10 minutes or less—again, with some variation depending on the specific GPU.

[3] The name of the relevant algorithm is Chu-Liu-Edmonds; the pseudocode is given in Fig. 19.12 in the dependency parsing chapter, which you can download here.

[4] Keep in mind that the two parsers use different word vectors, so the LAS values that the two parsers get aren't directly comparable.

- Via ssh to `teach.cs.toronto.edu`, you can run your model on a GPU-equipped machine. The starter code directory includes a `gpu-train.sh` script that will run the relevant command to run your `train.py` file, assuming the latter is in your current working directory (i.e., assuming that you run your the command while your shell was in the same directory as `train.py`). Note that the GPU machines are shared resources, so there may be limits on how long your code is allowed to run.

  The `gpu-train.sh` script should be sufficient; it takes care of submitting and running the job to the cluster-management software. If you like, you are free to manage this yourself; see the documentation online at the relevant page on the Teaching Labs site. The partition you can use for this class is `csc485`.

  If you are accessing the labs in person, the Teaching Labs FAQ page lists the labs that have GPU-equipped machines.

- Keep in mind that your mark is based on your implementation, *not* (directly) on the LAS your implementation achieves. Getting the same LAS as mentioned above doesn't guarantee full marks; nor does getting a LAS lower than that mentioned guarantee that marks will be docked. **The end result can vary based on differences in environment** (hardware or software) so this number is approximate! (But the further away your LAS, the more likely it is that you have an error in your implementation.)

- Except for `is_single_root` and `mst_single_root`, make sure that all of your changes occur under `ENTER YOUR CODE BELOW` and do not change the rest of the starter code. **Do not add any extra package or module imports and do not modify any other parts of the files** other than to fill your details in at the top of the file as instructed on the front page of this assignment handout.

  You may alter other parts of the code as needed *for debugging*, but make sure that you revert any such changes for your final run and submission.

- You will be better off with partially complete implementations that work but are sometimes incorrect than you will with an almost-complete implementation that Python can't run. Keep this in mind for your final submission. Also, make sure your submitted files are importable in Python (i.e., make sure that `'import graphalg'` and `'import graphdep'` don't yield errors). If you prefer to work locally on your own computer, you should still check and make sure your code work on teach.cs.

- Please keep an eye on the Assignment 1 page on the course website for essential information, important dates, relevant links, and access to the errata and FAQ page with all clarifications.

- Corpora and model weights are available on Teaching Labs machines and the teach.cs servers at the path `/u/csc485h/fall/pub/a1`. If you prefer to work locally on your own computer, you can download the data and model weights.[5] Once downloaded, unzip the data in your local directory and specify the path using the `--dir-path` option.

  ```
  $ python3.12 train.py --dir-path <your-local-path> q1
  $ python3.12 train.py --dir-path <your-local-path> q2
  ```

[5] https://www.cs.toronto.edu/~niu/teaching/csc485/a1_data.zip

Submission Instructions

This assignment is submitted electronically via MarkUs. You should submit a total of five ("7") required files as follows:

- `a1written.pdf`: a PDF document containing your written answers as applicable for each question.
- `q1_model.py`: the (entire) `q1_model.py` file with your implementations filled in.
- `q1_parse.py`: the (entire) `q1_parse.py` file with your implementations filled in.
- `q2_algorithm.py`: the (entire) `q2_algorithm.py` file with your implementations filled in.
- `q2_model.py`: the (entire) `q2_model.py` file with your implementations filled in.
- `weights-q1.pt` and `weights-q2.pt`: the weights file produced by your models' final runs.

Again, except for `is_single_root` and `mst_single_root`, ensure that there are no changes to the existing code beside adding in your code beneath `ENTER YOUR CODE BELOW` and filling in your details at the top of the file; **you will lose marks if there are**. Do not change or remove the boundary comments either.

Appendix A PyTorch

You will be using PyTorch to implement components of your neural dependency parser. PyTorch is an open-source library for numerical computation that provides automatic differentiation, making the back-propagation aspect of neural networks easier. Computation is done in units of *tensors*; the storage of and operations on tensors is provided in both CPU and GPU implementations, making it simple to switch between the two.

The first tutorial covers the essential components of PyTorch needed to complete this assignment. Additionally, we recommend exploring some tutorials on the official PyTorch website to familiarize yourself with PyTorch's mechanics. In particular, the *Tensors* and *nn module* sections of the Learning PyTorch with Examples tutorial will be most relevant for this assignment.

If you want to run this code on your own machine, you can install the `torch` package via `pip3`, or any of the other installation options available on the PyTorch site. Make sure to use PyTorch version 2.3.0. You will also need to install the `conllu` and `transformers` packages.

Make sure to consult the PyTorch documentation as needed. The API documentation for `torch`, `torch.nn`, `torch.nn.functional`, and `torch.Tensor` will be useful while implementing the code in this assignment.