CSC485/2501 A1 Tutorial 3

Samarendra Chandan Bindu Dash

UNIVERSITY OF TORONTO

Assignment 1

Part 1: Implement your own transition-based dependency parser

Part 2: Implement your own graph-based dependency parser

Assignment 1

Part 1: Implement your own transition-based dependency parser

Part 2: Implement your own graph-based dependency parser

Part 2: Graph-based Dependency Parser

Tutorial overview

What is projectivity?

Edge-factored Parsing Example

Biaffine-attention Neural Dependency Parser

- Arc scorer
- Label scorer

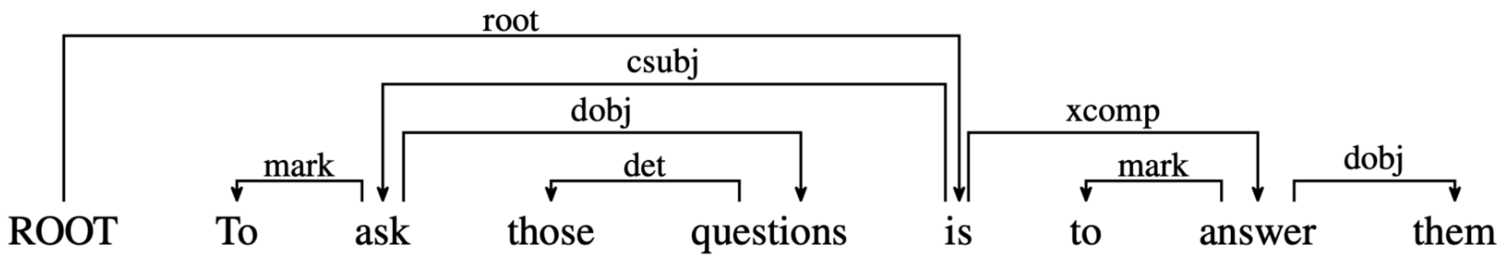
Tips for Batching

Projective Dependency Trees

Projectivity: If a dependency arc exist from i → j, then there exist a path from i → k for every min(i,j) < k < max(i,j).

A dependency tree is **projective** if all its arcs are projective.

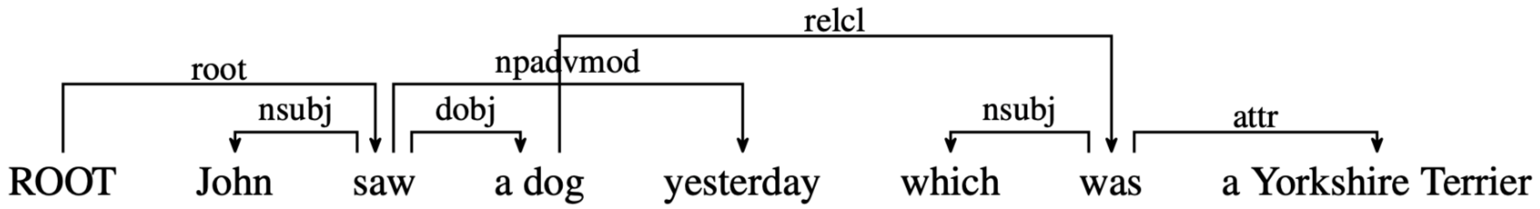
This corresponds to a planar graph:



Non-projective Dependency Trees

A **non-projective** dependency tree has one or more non-projective dependency arc.

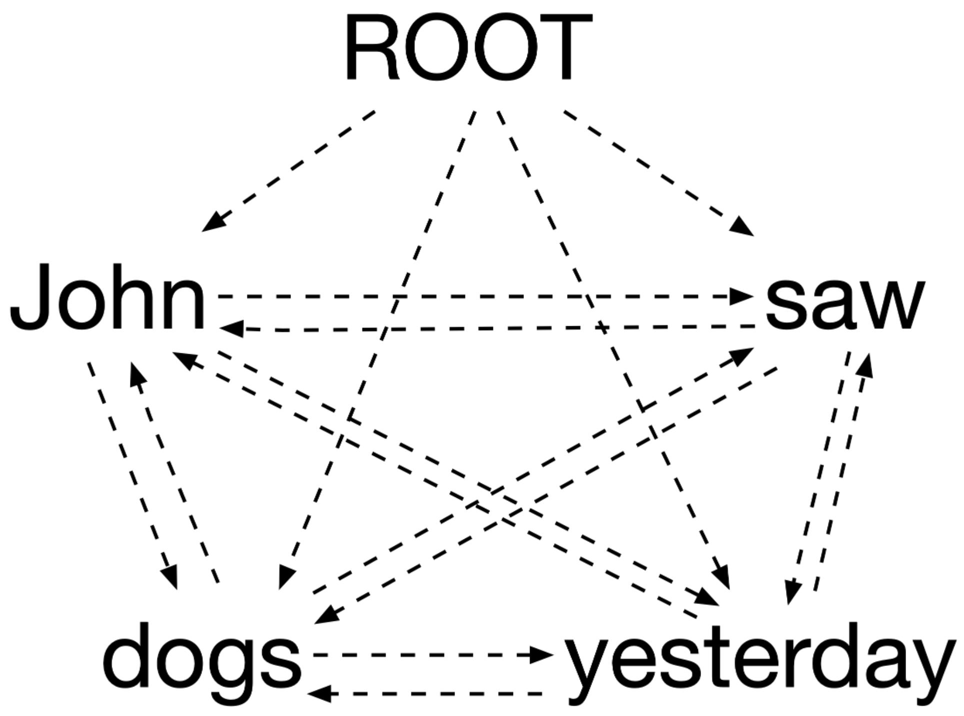
This corresponds to a graph with a cross-over arc:



Transition-based parsers cannot handle this.

Graph-based Dependency Parsing

Consider all possible arcs between pairs of words:



Edge-factored Parsing

An edge-factored parser involves the following steps:

1. Create a bidirectional, connected graph that corresponds to the sentence.
2. Score each arc in the graph.
3. Finding the maximum spanning tree in the graph and treat the constituent arcs as your dependency arcs.
4. Tag each arc with an appropriate dependency label.

Edge-factored Parsing

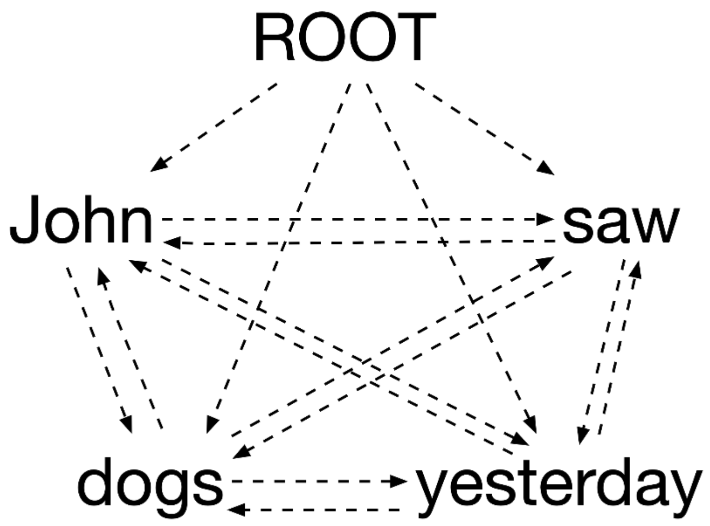
1. Create a bidirectional, connected graph that corresponds to the sentence.

Input

ROOT John saw dogs yesterday

Add ROOT if necessary

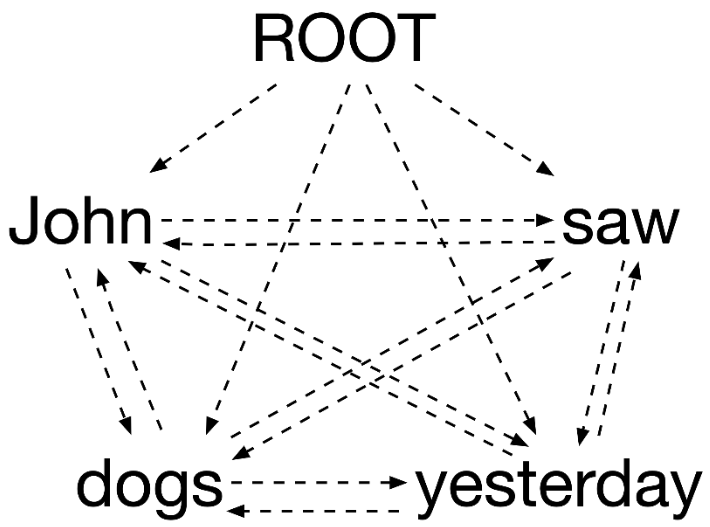
Output



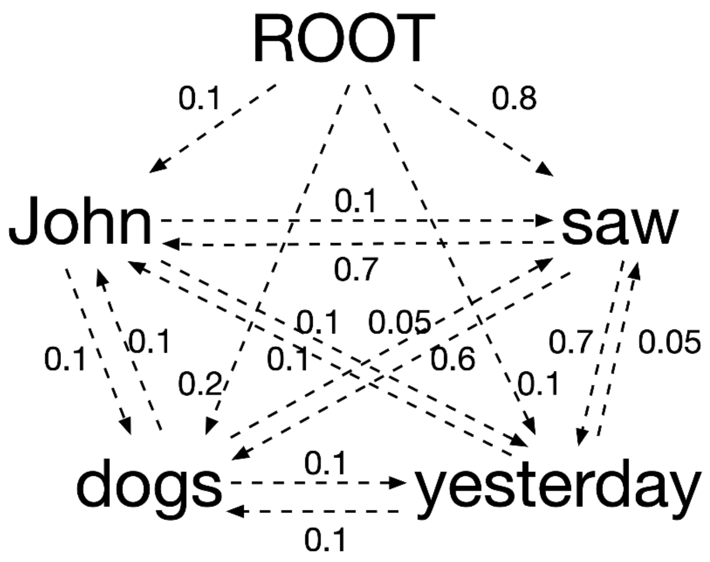
Edge-factored Parsing

2. Score each arc in the graph.

Input



Output

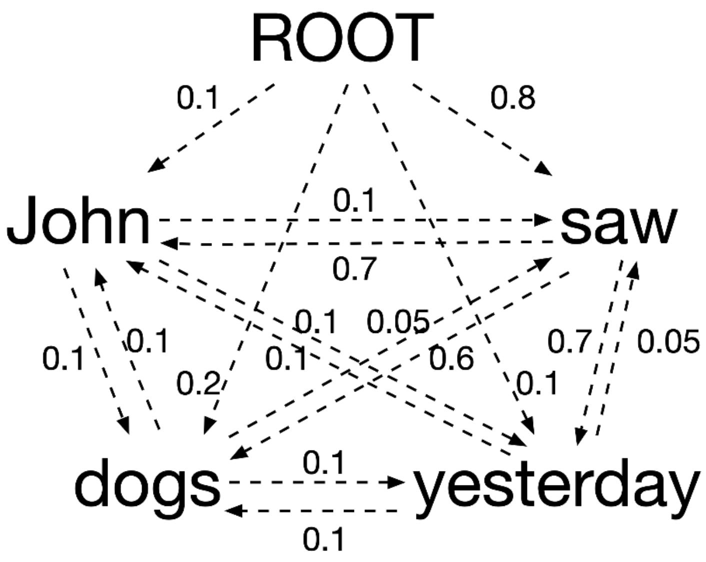


We will do this using a neural network (see the Arc Scorer)

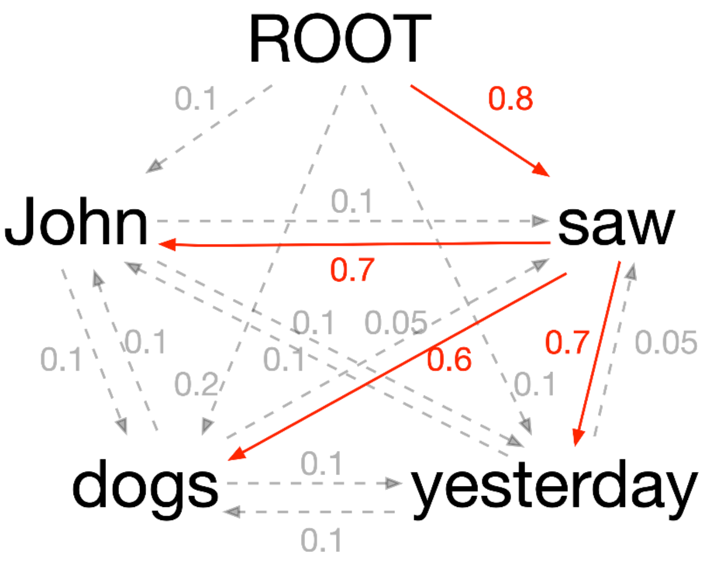
Edge-factored Parsing

3. Find the maximum spanning tree in the graph.

Input



Output



We will do this using a neural network (see the Arc Scorer)

Edge-factored Parsing

3. Find the maximum spanning tree in the graph.

- Find the MST using Chu-Liu Edmonds algorithm.
  - Pseudo-code from the textbook and example in the lecture
- The MST can then be viewed as an unlabeled dependency parse tree

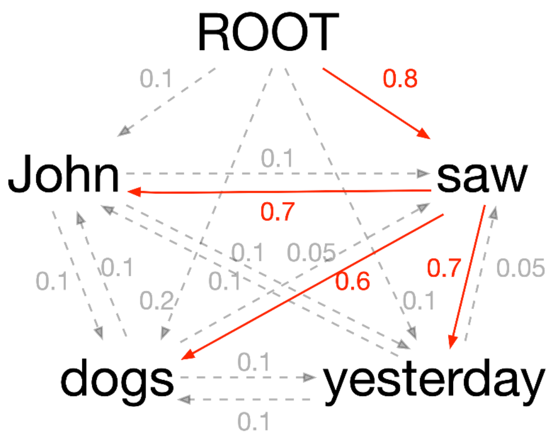
function MAXSPANNINGTREE(*G=(V,E), root, score*) **returns** *spanning tree*

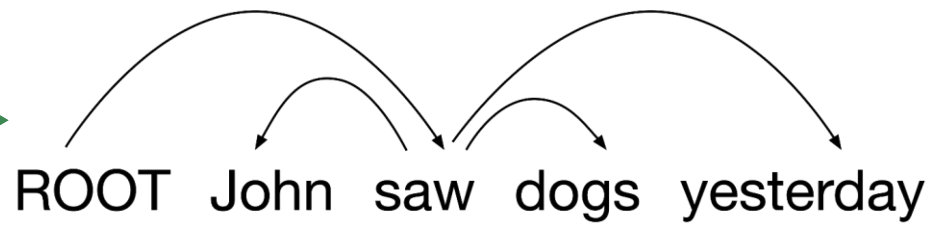
$F \leftarrow []$
$T' \leftarrow []$
$score' \leftarrow []$
for each $v \in V$ **do**
  $bestInEdge \leftarrow \mathrm{argmax}_{e=(u,v) \in E}\ score[e]$
  $F \leftarrow F \cup bestInEdge$
  for each $e=(u,v) \in E$ **do**
    $score'[e] \leftarrow score[e] - score[bestInEdge]$

if $T=(V,F)$ is a spanning tree **then return** it
else
  $C \leftarrow$ a cycle in F
  $G' \leftarrow$ CONTRACT(G, C)
  $T' \leftarrow$ MAXSPANNINGTREE$(G', root, score')$
  $T \leftarrow$ EXPAND(T', C)
  return T

function CONTRACT(*G, C*) **returns** *contracted graph*

function EXPAND(*T, C*) **returns** *expanded graph*

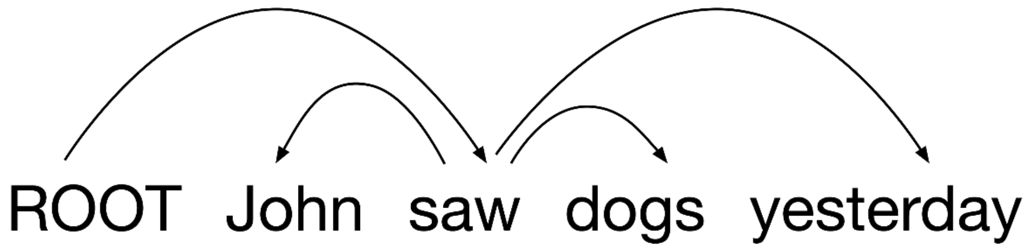




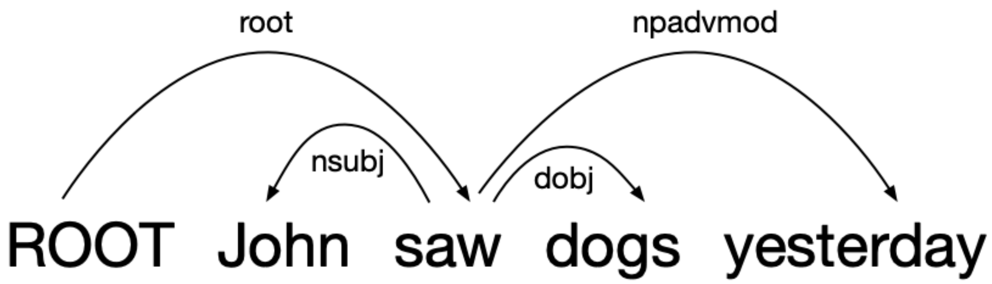
Edge-factored Parsing

4. Tag each arc with an appropriate dependency label.

Input



Output



We will also train a neural network to do this

(see the Label Scorer)

Biaffine-attention Neural Dependency Parser

We need to train two neural networks to complete our parser:

1. **Arc Scorer**: Scores the likelihood for each arc in the graph.
   - The Scores are then passed to an MST algorithm.
   - Why can't we just take the argmax here?

2. **Label Scorer**: Given an arc in the graph, output a score distribution over all possible tags.
   - In this case, we can simply take the argmax.

For the original architecture, see Dozat et al. (2017)

Quick note:

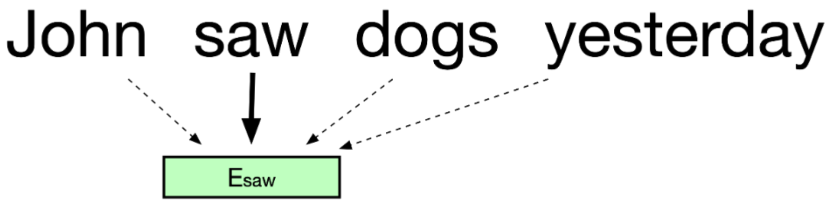
Contextualized Word Embedding

The graph structure does not encode word order information.

We can retain this information by using contextualized word embeddings, which is aware of its usage context:

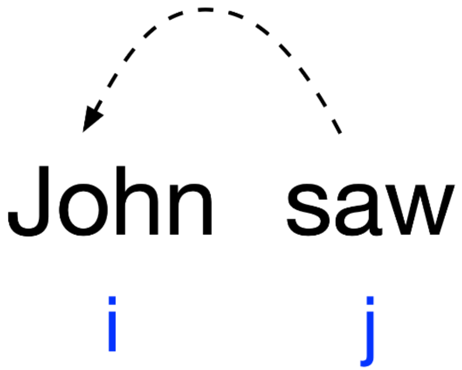
Word2Vec:

BERT:



Arc Scorer

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



Intuitions for the architecture:

- The MLP layers create a pair of distinct representations for each word in the sentence, one for when the word is viewed as a dependent in an arc, and another for when it is viewed as a head.
- The biaffine transformation between these representations then checks the appropriateness of having a directed edge between two words.
- The bias term acts as a prior checking how likely a word is to be used as a head of an arc.

Arc Scorer

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

S – Contextual word embeddings for the sentence.

D_A – Learned representations for all words in the sentence when considered as dependents.

H_A – Learned representations for all words in the sentence when considered as heads.

a_{ij} – Numeric score corresponding to the arc $j \longrightarrow i$

Arc Scorer

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

W_A and b_A are learnable parameters in the model.

Use `torch.nn.Parameter` to create these weights/biases.

- e.g. `b_A = torch.nn.Parameter(torch.ones((10)), requires_grad=True)`

This tells PyTorch to register the tensor with your neural network module, in which you can find the parameters using `self.parameters`.

Arc Scorer

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

For this assignment, create the multi-layer perceptron (a.k.a. fully connected layers) using `torch.nn.Sequential`:

- E.g. `nn.Sequential(nn.Linear(50, 100), nn.ReLU(), nn.Dropout(0.5))`

This creates an encapsulated layer that is equivalent to passing an input into the three layers sequentially.

Arc Scorer

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

The obtained score is then passed into a softmax layer:

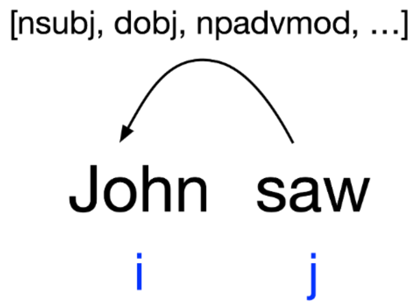
$$\hat{h}_i = \mathrm{softmax}(a_{ij})$$
$$J_{\mathrm{h}} = CE(h, \hat{h}) = -\sum_{i=1}^{n} h_i \log \hat{h}_i$$

Label Scorer

$$D_{\mathrm{L}} = \mathrm{MLP}_{\mathrm{Ld}}(S)$$

$$H_{\mathrm{L}} = \mathrm{MLP}_{\mathrm{Lh}}(S)$$

$$l_{ij} = [D_{\mathrm{L}}]_i W_{\mathrm{L}}([H_{\mathrm{L}}]_j)^{\mathrm{T}} + [H_{\mathrm{L}}]_j W_{\mathrm{Lh}} + [D_{\mathrm{L}}]_i W_{\mathrm{Ld}} + b_{\mathrm{L}}$$



Intuitions for the architecture:

- The MLP layers and the biaffine transformation are very similar, except now that the biaffine transformation checks the appropriateness of having each individual tag on a directed edge between two words.
- The two subsequent terms in the sum checks the appropriateness of having a certain word as the head/dependent of a given dependency relation.
- The bias term sets a prior over all possible dependency relations.

Label Scorer

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

S – Contextual word embeddings for the sentence.

D_L – Learned representations for all words in the sentence when considered as dependents.

H_L – Learned representations for all words in the sentence when considered as heads.

l_{ij} – Vector of numeric scores where the k'th value correspond to the score of having the k'th dependency relation labeled for the arc $j \rightarrow i$

Label Scorer

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

Since l_{ij} is a vector, W_L and b_L both have one extra dimension.

- Back in the arc scorer case, W is of shape [X, X], but now it is of shape [X, Y, X], where X and Y are both integers.
- The bias term is now a vector instead of a number.

Label Scorer

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

The score vector is then also passed into a softmax layer:

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

Taking an argmax over the softmax scores gives the predicted label.

Batching

We apply batched computation to take advantage of powerful parallelism in modern hardware (i.e. GPUs).

The tensors given to you in the assignment are already in batch format.

You'll need to maintain the given batch format in all neural network computations.

Batching

S1: Word_0, Word_2, Word_3
S2: Word_3, Word_1, Word_1, Word_0
S3: Word_3, Word_2, Word_1, Word_2, Word_0

Directly encoding this into an array gives us:

```
[ [0,2,3],
  [3,1,1,0],
  [3,2,1,2,0] ]
```

Cannot perform parallel computation because the sequence lengths are not matched.

Batching

S1: Word_0, Word_2, Word_3
S2: Word_3, Word_1, Word_1, Word_0
S3: Word_3, Word_2, Word_1, Word_2, Word_0

Trick: Pad the shorter sequences and keep track of sequence lengths.

```
[ [0,2,3,0,0],
  [3,1,1,0,0],          [3,4,5]
  [3,2,1,2,0] ]
```

This results in an extra batch dimension B in your tensor. In the example above, B=3.

Working with Batched Tensors

To perform efficient computation, you don’t want to unpack the batch dimension.

Let A, B be two batched tensors of shape (b, X, Y) and (b, Y, Z) respectively. Now, to compute a matrix multiplication:

```
for i in range(b):
    result[i] = A[i] * B[i]
```

```
torch.einsum('bij,bjk->bik', A, B)
```

Working with Batched Tensors

`torch.einsum(…)` allows efficient batched operations.

```
torch.einsum('bij,bjk->bik', A, B)
```

Separated by commas, each term to the left of the arrow defines the dimensions of the input tensors.

The term following the arrow specifies the output, where every missing dimension will be summed over.

Working with Batched Tensors

`torch.einsum(…)` allows efficient batched operations.

```
torch.einsum('bij,bjk->bik', A, B)
```

The matrix operation defined above is thus equivalent to computing:

$$\text{result}[b,i,k] = \Sigma_j \, A[b,i,j] * B[b,j,k] \quad \forall\, b, i, k$$

More examples in the PyTorch documentation:

- https://pytorch.org/docs/stable/generated/torch.einsum.html

Question

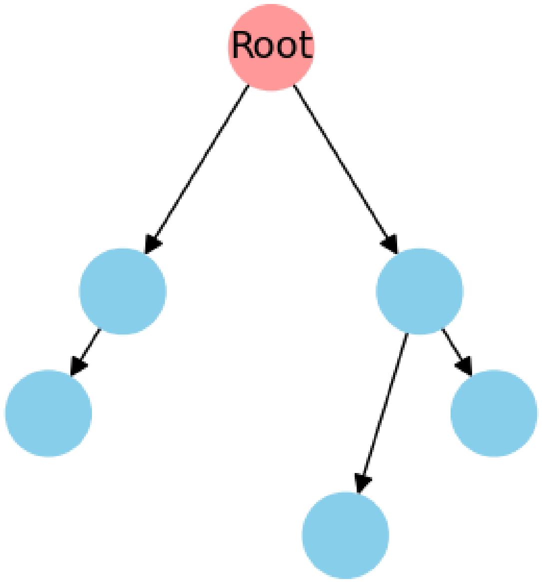
Why can’t we use Dijkstra's algorithm for minimum cost spanning tree?

Question

Why can’t we use Prim's algorithm for minimum cost spanning tree?

Chu-Liu-Edmond's Algorithm

- Algorithm for minimum cost arborescence.
  - What is an arborescence?



Chu-Liu-Edmond’s Algorithm

Advice for implementation

- Start with implementing the algorithm with for loop.
  - Verifying correctness is the most important requirement here
- Incrementally, try to convert each for loop to torch operations
  - You are not required to remove all the for loops.
  - Verify correctness at each incremental step
  - If the algorithm takes too long to execute, marks would be deducted.

That’s it!

Good luck on the assignment!

Please post any questions you have on Piazza.