1. Machine Learning & Neural Networks (8 points)

(a) (4 points) Adam Optimizer

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

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

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.

m is accumulating the effect of previous gradient values, giving higher importance to the last values and lower importance to current value.

If we have accumulated gradients with the same direction, the updates will be bigger ⇒ learning will be accelerated. If there are few gradient with the opposite direction the effect will be dampered because of the $(1-\beta)$ factor.

Reduces the SGD noise / variance

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:

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

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?

Frequently occurring features have a greater effect on the gradient, then its value is large. It's still large when squared / sqr rooted causing the learning rate to be small in these cases. As these features have a great effect on the loss it is good to use a small learning rate to avoid big jumps that might make us shift direction to the wrong direction (far from the minimum)

On the contrary infrequent occurring features don't affect the loss too much, the gradient is small causing a larger learning rate accelerating learning which is good.

(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 $i \in \{1, \ldots, D_h\}$.

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.

$\gamma = \frac{1}{p_{drop}}$ why? Because we don't need to affect the input values coming from previous layers. Then γ has a counter effect with p_{drop}.

$$E_{p_{drop}}[\underbrace{\gamma\, p_{drop}}_{1}\, y^2] = y^2$$ the 2nd input

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.)

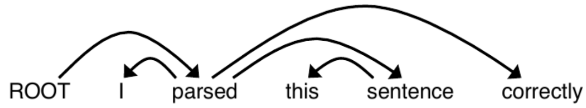
To avoid overfitting, specially when the NN architecture is very complex. Dropping out simplifies the layers reducing specialization.

We can't apply Dropout during test/evaluation because we don't want to "split-out" the NN in 2^n thinned networks why? Because we would need to average the predictions and that will affect the output. It is better to re-scale the weights with p_{drop} in the dropped out layers.

Drop out is similar to split the NN in 2^n thinner NNs.

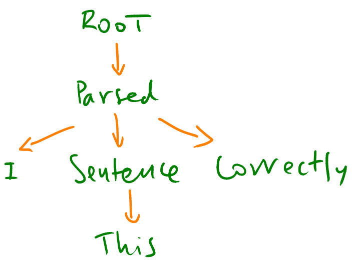
2. Neural Transition-Based Dependency Parsing (44 points)

(a) (4 points) Go through the sequence of transitions needed for parsing the sentence *"I parsed this sentence correctly"*. 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, parsed, this, sentence, correctly]		Initial Configuration
[ROOT, I]	[parsed, this, sentence, correctly]		SHIFT
[ROOT, I, parsed]	[this, sentence, correctly]		SHIFT
[ROOT, parsed]	[this, sentence, correctly]	parsed→I	LEFT-ARC
[ROOT, Parsed, this]	[sentence, correctly]		SHIFT
[ROOT, Parsed, this, sentence]	[correctly]		SHIFT
[ROOT, Parsed, sentence]	[correctly]	Sentence → This	LEFT-ARC
[ROOT, Parsed]	[correctly]	Parsed → Sentence	RIGHT-ARC
[ROOT, Parsed, correctly]	[ ]		SHIFT
[ROOT, Parsed]	[ ]	Parsed → correctly	RIGHT-ARC
[ROOT]	[ ]	ROOT → Parsed	RIGHT-ARC

11 steps
5 words



(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.

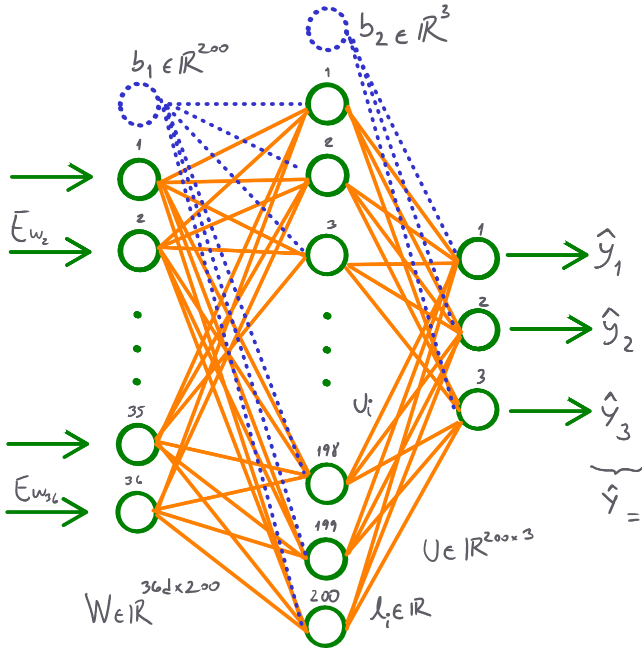
2n. Because we'll need to add the sentences in "n" SHIFT movements and we'll need to take them out of the stack with "n" LEFT/RIGHT-ARC movements.

Network Architecture

$E \in \mathbb{R}^{|V| \times d}$ is an embedding matrix

↳ input vector
$X = [E_{w_1}, \dots, E_{w_m}] \in \mathbb{R}^{dm}$

$m = \#$ of features { words / dependents, Pos Tags ?, Arcs ? }
↳ Each feature is embed in "d" dimensions



U_{do}
↳ orig in
↓ destination

$\hat{y} = \text{softmax}(l)$ ↳ $hU + b_2$
↓ Prob. distribution of next transition given current state of buffer, stack, dep.

Training Process

1) Extract a features vector representing current state (buffer, stack, dependencies, etc.) ↳ list of tokens

$w = [w_1, w_2, \dots, w_m]$ $\quad m = \#$ of features
↳ index of token associated with 1st feature

2) Look up the embeddings for each word in the vector w ↳ features vector

$E = \begin{bmatrix} [&] \\ [&] \\ \vdots & \\ [&] \end{bmatrix}_{|V| \times d}$ ↗ word embeddings

$X = [E_{w_1} \; E_{w_2} \; \dots \; E_{w_m}] \in \mathbb{R}^{dm}$
↳ Embedding of the word associated with the feature 1 (w_1)

We perform the look up process in batches: (e.g. if $d = 2$, $m = 2$ $\Rightarrow X_{batch}[0] = [\underbrace{1.76 \;\; -1.01}_{E_{w_1}} \;\; \underbrace{0.17 \;\; -1.29}_{E_{w_2}}]$)

$\text{w\_batch} = \begin{bmatrix} [&] \\ [&] \\ \vdots & \\ [&] \end{bmatrix}_{\underset{4}{batch\_size} \times \underset{3}{m}} \Rightarrow X_{batch} = \begin{bmatrix} [&] \\ [&] \\ \vdots & \\ [&] \end{bmatrix}_{batch\_size \times dm}$

Note: The lookup operation must be as efficient as possible since will be heavily used during training. NO FOR LOOPS. Use native PyTorch functions:

1) Flatten w_batch $\Rightarrow$ w_flatten = $[\overbrace{- \; - \; - \; - \; - \; -}^{12}]$
2) Index select the embeddings with w_flatten
3) Reshape 2) output to batch_size x dm using view

Alternatively we can use nn.embedding but the whole point of this exercise is to build one!

3) Compute the forward pass with:

$$h = \text{ReLU}(xW + b_1)$$

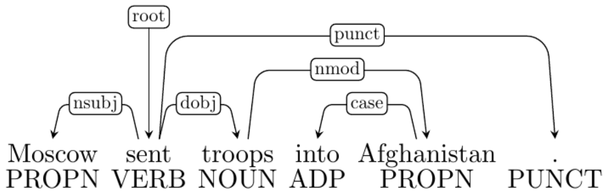
$$l = hU + b_2$$

$$\hat{y} = \text{softmax}(l)$$

4) Update the weights with:

$$J(\theta) = CE(y, \hat{y}) = -\sum_{i=1}^{3} y_i \log \hat{y}_i$$

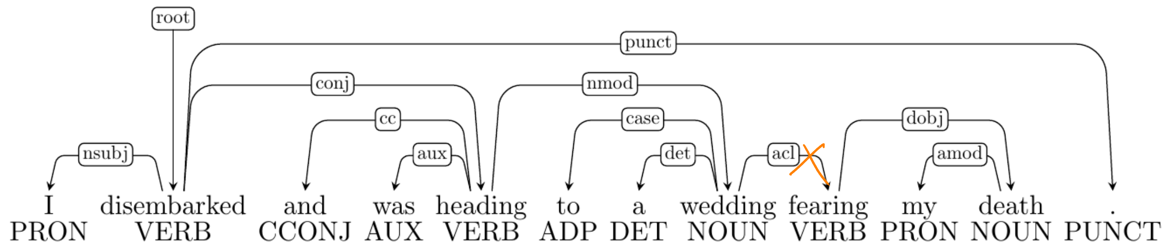
(f) (12 points) We'd like to look at example dependency parses and understand where parsers like ours might be wrong. For example, in this sentence:



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

i.



I disembarked and was heading to a wedding fearing my death.

ROOT

verb phrase

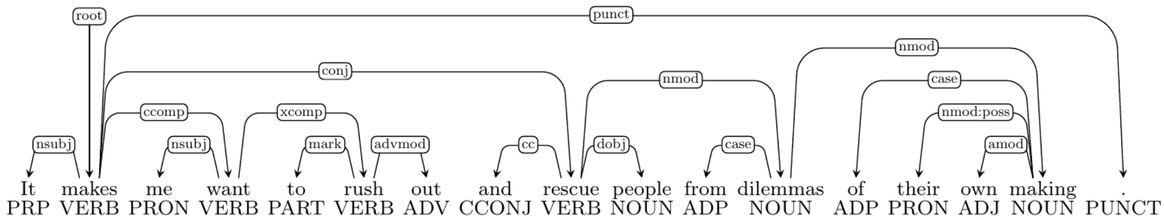
verb phrase

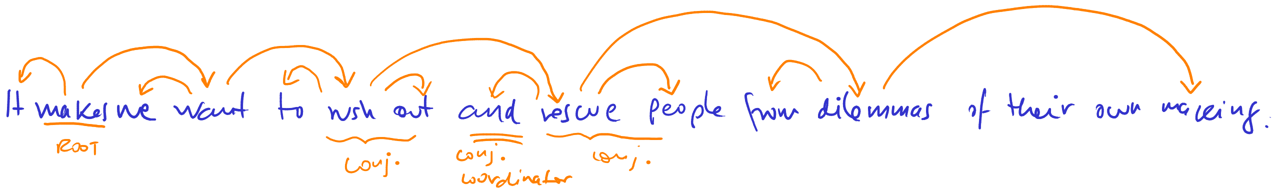
Error type: Verb Phrase Attachment Error

Incorrect dependency: wedding → fearing

Correct dependency: heading → fearing

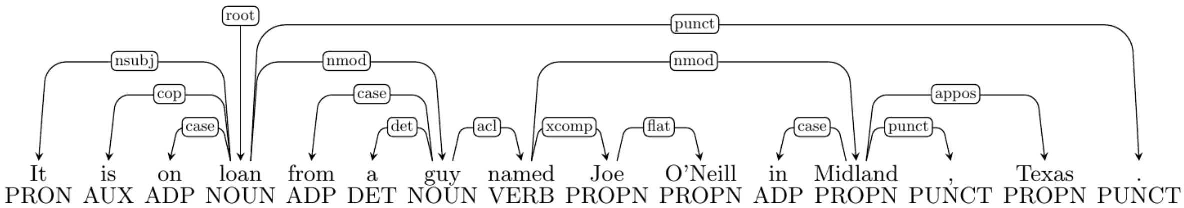
ii.

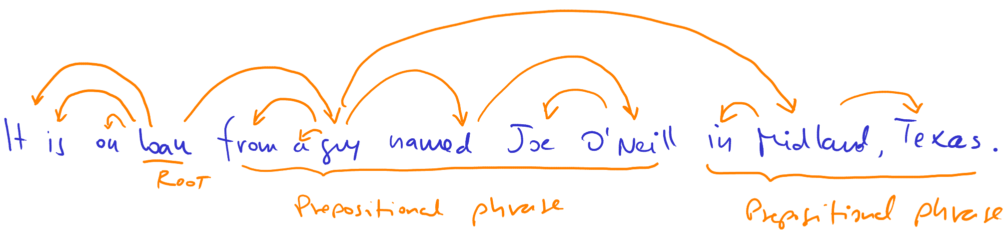




Error type: Coordination attachment error
Incorrect dependency: makes → rescue
Correct dependency: rush → rescue

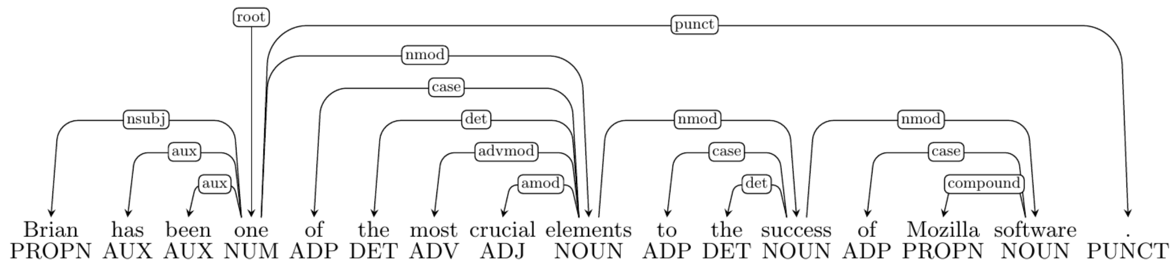
iii.

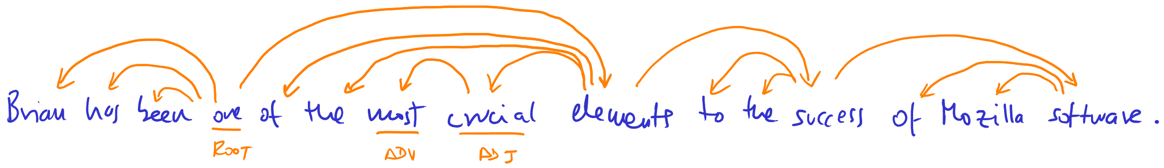




Error type: Prepositional Phrase Attachment Error
Incorrect dependency: named → Midland
Correct dependency: guy → midland

iv.





Error type : Modifier Attachment Error
Incorrect dependency: elements → most
Correct dependency: crucial → most