CS 224n Assignment #2: Word2Vec and Dependency Parsing

Due Date: April 18th, Thursday, 4:30 PM PST.

In this assignment, you will review the mathematics behind Word2Vec and 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 explore the partial derivatives involved in training a Word2vec model using the naive softmax loss. In Part 2, you will learn about two general neural network techniques (Adam Optimization and Dropout). In Part 3, you will implement and train a dependency parser using the techniques from Part 2, before analyzing a few erroneous dependency parses.

If you are using LaTeX, you can use `\ifans{}` to type your solutions.

Please tag the questions correctly on Gradescope, otherwise the TAs will take points off if you don't tag questions.

1. Understanding word2vec (15 points)

Recall that the key insight behind `word2vec` is that *'a word is known by the company it keeps'*. Concretely, consider a 'center' word c surrounded before and after by a context of a certain length. We term words in this contextual window 'outside words' (O). For example, in Figure 1, the context window length is 2, the center word c is 'banking', and the outside words are 'turning', 'into', 'crises', and 'as':

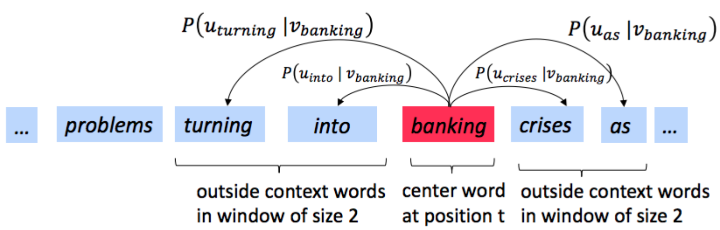


Figure 1: The word2vec skip-gram prediction model with window size 2

Skip-gram `word2vec` aims to learn the probability distribution $P(O|C)$. Specifically, given a specific word o and a specific word c, we want to predict $P(O = o|C = c)$: the probability that word o is an 'outside' word for c (i.e., that it falls within the contextual window of c). We model this probability by taking the softmax function over a series of vector dot-products:

$$P(O = o \mid C = c) = \frac{\exp(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{w \in \text{Vocab}} \exp(\mathbf{u}_w^\top \mathbf{v}_c)} \tag{1}$$

For each word, we learn vectors u and v, where $\mathbf{u}_o$ is the 'outside' vector representing outside word o, and $\mathbf{v}_c$ is the 'center' vector representing center word c. We store these parameters in two matrices, $\mathbf{U}$ and $\mathbf{V}$. The columns of $\mathbf{U}$ are all the 'outside' vectors $\mathbf{u}_w$; the columns of $\mathbf{V}$ are all of the 'center' vectors $\mathbf{v}_w$. Both $\mathbf{U}$ and $\mathbf{V}$ contain a vector for every $w \in$ Vocabulary.[1]

Recall from lectures that, for a single pair of words c and o, the loss is given by:

$$\mathbf{J}_{\text{naive-softmax}}(\mathbf{v}_c, o, \mathbf{U}) = -\log P(O = o|C = c). \tag{2}$$

[1]Assume that every word in our vocabulary is matched to an integer number k. Bolded lowercase letters represent vectors. $\mathbf{u}_k$ is both the k^{th} column of $\mathbf{U}$ and the 'outside' word vector for the word indexed by k. $\mathbf{v}_k$ is both the k^{th} column of $\mathbf{V}$ and the 'center' word vector for the word indexed by k. **In order to simplify notation we shall interchangeably use k to refer to word k and the index of word k.**

We can view this loss as the cross-entropy[2] between the true distribution $\mathbf{y}$ and the predicted distribution $\hat{\mathbf{y}}$, for a particular center word c and a particular outside word o. Here, both $\mathbf{y}$ and $\hat{\mathbf{y}}$ are vectors with length equal to the number of words in the vocabulary. Furthermore, the k^{th} entry in these vectors indicates the conditional probability of the k^{th} word being an 'outside word' for the given c. The true empirical distribution $\mathbf{y}$ is a one-hot vector with a 1 for the true outside word o, and 0 everywhere else, for this particular example of center word c and outside word o.[3] The predicted distribution $\hat{\mathbf{y}}$ is the probability distribution $P(O|C = c)$ given by our model in equation (1).

Note: Throughout this homework, when computing derivatives, please use the method reviewed during the lecture (i.e. no Taylor Series Approximations).

[2]The **cross-entropy loss** between the true (discrete) probability distribution p and another distribution q is $-\sum_i p_i \log(q_i)$.
[3]Note that the true conditional probability distribution of context words for the entire training dataset would not be one-hot.

(a) (2 points) Prove that the naive-softmax loss (Equation 2) is the same as the cross-entropy loss between $\mathbf{y}$ and $\hat{\mathbf{y}}$, i.e. (note that $\mathbf{y}$ (true distribution), $\hat{\mathbf{y}}$ (predicted distribution) are vectors and $\hat{\mathbf{y}}_o$ is a scalar):

$$-\sum_{w \in \text{Vocab}} \mathbf{y}_w \log(\hat{\mathbf{y}}_w) = -\log(\hat{\mathbf{y}}_o). \tag{3}$$

Your answer should be one line. You may describe your answer in words.

Solution:

The true empirical distribution of $\mathbf{y}$ is a **one-hot** vector with a 1 for the true outside word o, and 0 everywhere else. Hence, the naive-softmax loss can be represented as follows:

$$-\sum_{w \in \text{Vocab}} \mathbf{y}_w \log(\hat{\mathbf{y}}_w) = \sum_{w \neq o} -0 \cdot \log(\hat{\mathbf{y}}_w) + \sum_{w = o} -1 \cdot \log(\hat{\mathbf{y}}_w) = -log(\hat{\mathbf{y}}_o)$$

(b) (6 points) i. Compute the partial derivative of $\mathbf{J}_{\text{naive-softmax}}(\mathbf{v}_c, o, \mathbf{U})$ with respect to $\mathbf{v}_c$. *Please write your answer in terms of* $\mathbf{y}$, $\hat{\mathbf{y}}$, $\mathbf{U}$, *and show your work to receive full credit.*

- **Note**: Your final answers for the partial derivative should follow the shape convention: the partial derivative of any function $f(x)$ with respect to x should have the **same shape** as x.[4]
- Please provide your answers for the partial derivative in vectorized form. For example, when we ask you to write your answers in terms of $\mathbf{y}$, $\hat{\mathbf{y}}$, and $\mathbf{U}$, you may not refer to specific elements of these terms in your final answer (such as $\mathbf{y}_1$, $\mathbf{y}_2$, $\ldots$).

ii. When is the gradient you computed equal to zero?
Hint: You may wish to review and use some introductory linear algebra concepts.

iii. The gradient you found is the difference between the two terms. Provide an interpretation of how each of these terms improves the word vector when this gradient is subtracted from the word vector v_c.

Solution:

i. for a sample (c, o), we have gradient:

$$\begin{aligned}
\frac{\partial}{\partial \mathbf{v}_c} \mathbf{J}_{\text{naive-softmax}}(\mathbf{v}_c, o, \mathbf{U}) &= \frac{\partial}{\partial \mathbf{v}_c}[-\log \frac{\exp(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{w \in \text{Vocab}} \exp(\mathbf{u}_w^\top \mathbf{v}_c)}] && (4)\\
&= \frac{\partial}{\partial \mathbf{v}_c}[-\log(\exp(\mathbf{u}_o^\top \mathbf{v}_c)) + \log(\sum_{w \in \text{Vocab}} \exp(\mathbf{u}_w^\top \mathbf{v}_c))] && (5)\\
&= -\mathbf{u}_o + \frac{\partial}{\partial \mathbf{v}_c} \log(\sum_{w \in \text{Vocab}} \exp(\mathbf{u}_w^\top \mathbf{v}_c)) && (6)\\
&= -\mathbf{U}\mathbf{y} + \frac{1}{\sum_{w \in \text{Vocab}} \exp(\mathbf{u}_w^\top \mathbf{v}_c)} \frac{\partial}{\partial \mathbf{v}_c} \sum_{w \in \text{Vocab}} \exp(\mathbf{u}_w^\top \mathbf{v}_c) && (7)\\
&= -\mathbf{U}\mathbf{y} + \frac{1}{\sum_{w \in \text{Vocab}} \exp(\mathbf{u}_w^\top \mathbf{v}_c)} \sum_{w' \in \text{Vocab}} \exp(\mathbf{u}_{w'}^\top \mathbf{v}_c) \mathbf{u}_{w'} && (8)\\
&= -\mathbf{U}\mathbf{y} + \sum_{w' \in \text{Vocab}} \frac{\exp(\mathbf{u}_{w'}^\top \mathbf{v}_c)}{\sum_{w \in \text{Vocab}} \exp(\mathbf{u}_w^\top \mathbf{v}_c)} \mathbf{u}_{w'} && (9)
\end{aligned}$$

[4]This allows us to efficiently minimize a function using gradient descent without worrying about reshaping or dimension mismatching. While following the shape convention, we're guaranteed that $\theta := \theta - \alpha \frac{\partial J(\theta)}{\partial \theta}$ is a well-defined update rule.

for a fixed w', the $\frac{\exp(\mathbf{u}_{w'}^\top \mathbf{v}_c)}{\sum_{w \in \text{Vocab}} \exp(\mathbf{u}_w^\top \mathbf{v}_c)}$ is just $P(O = w' \mid C = c)$, which is $\hat{y}_{w'}$, here $\hat{y}_{w'}$ is a scalar
hence:

$$\frac{\partial}{\partial \mathbf{v}_c} \mathbf{J}_{\text{naive-softmax}}(\mathbf{v}_c, o, \mathbf{U}) = -\mathbf{U}\mathbf{y} + \sum_{w' \in \text{Vocab}} \hat{\mathbf{y}}_{w'} \mathbf{u}_{w'} \tag{10}$$

$$= -\mathbf{U}\mathbf{y} + \mathbf{U}\hat{\mathbf{y}} \tag{11}$$

$$= \mathbf{U}(\hat{\mathbf{y}} - \mathbf{y}) \tag{12}$$

ii. The gradient is equal to zero when $(\hat{\mathbf{y}} - \mathbf{y})$ lies in the null space of the matrix $\mathbf{U}$ which stores the outside word vectors as columns. For a matrix $\mathbf{U} \in \mathbb{R}^{d \times |V|}$, where the embedding dimension d is much smaller than the vocabulary size $|V|$, the columns are linearly dependent, so there exist solutions with $\hat{\mathbf{y}} \neq \mathbf{y}$.

iii. The gradient descent step can be defined as follows:

$$\mathbf{v}_c^{\text{new}} \leftarrow \mathbf{v}_c^{\text{old}} - \eta \mathbf{U}\hat{\mathbf{y}} + \eta \mathbf{U}\mathbf{y}$$

, where η is the step size. Intuitively speaking, adding the vector $\eta\mathbf{U}\mathbf{y}$ to $\mathbf{v}_c^{\text{old}}$ will make $\mathbf{v}_c$ move towards the direction of $\mathbf{u}_o$ of the outside word o which lies in the context window of the center word c. On the other hand, subtracting the vector $\eta\mathbf{U}\hat{\mathbf{y}}$ will make $\mathbf{v}_c$ move away from all the other vectors in $\mathbf{U}$. Note that all probabilities follow $0 < \hat{\mathbf{y}}_i < 1$.

(c) (1 point) In many downstream applications using word embeddings, L2 normalized vectors (e.g. $\mathbf{u}/||\mathbf{u}||_2$ where $||\mathbf{u}||_2 = \sqrt{\sum_i u_i^2})$ are used instead of their raw forms (e.g. $\mathbf{u}$). Let's consider a hypothetical downstream task of binary classification of phrases as being positive or negative, where you decide the sign based on the sum of individual embeddings of the words. When would L2 normalization take away useful information for the downstream task? When would it not?

Hint: Consider the case where $\mathbf{u}_x = \alpha \mathbf{u}_y$ for some words $x \neq y$ and some scalar α. When α is positive, what will be the value of normalized $\mathbf{u}_x$ and normalized $\mathbf{u}_y$? How might $\mathbf{u}_x$ and $\mathbf{u}_y$ be related for such a normalization to affect or not affect the resulting classification?

Solution:

First of all, word embeddings are vectors fully described by magnitude and direction. Suppose there exists a word embedding $\mathbf{v} = \alpha\mathbf{u}$, where α is a scaler. The normalized form of $\mathbf{u}$ is $\frac{\mathbf{u}}{\|\mathbf{u}\|}$ and $\frac{\alpha\mathbf{u}}{|\alpha|\|\mathbf{u}\|}$ for $\mathbf{v}$. As can be observed, the magnitude information of $\mathbf{v}$ is thrown away after L2 normalization is done.

So, the answer is: For a task that takes magnitude (e.g., binary classification based on the sum of embeddings) into consideration, L2 normalization will affect the result (e.g., the sum leading to change of sign).

Otherwise, for a task that only consider the direction of embeddings, L2 normalization will not affext the result.

(d) (5 points) Compute the partial derivatives of $\mathbf{J}_{\text{naive-softmax}}(\mathbf{v}_c, o, \mathbf{U})$ with respect to each of the 'outside' word vectors, $\mathbf{u}_w$'s. There will be two cases: when $w = o$, the true 'outside' word vector, and $w \neq o$, for all other words. Please write your answer in terms of $\mathbf{y}$, $\hat{\mathbf{y}}$, and $\mathbf{v}_c$. In this subpart, you may use specific elements within these terms as well (such as $\mathbf{y}_1$, $\mathbf{y}_2$, $\ldots$). Note that $\mathbf{u}_w$ is a vector while $\mathbf{y}_1, \mathbf{y}_2, \ldots$ are scalars. Show your work to receive full credit.

Solution:

$$\frac{\partial}{\partial \mathbf{u}_w}\mathbf{J}_{\text{naive-softmax}}(\mathbf{v}_c, o, \mathbf{U}) = \frac{\partial}{\partial \mathbf{u}_w}[-\log(\exp(\mathbf{u}_o^\top \mathbf{v}_c)) + \log(\sum_{w \in \text{Vocab}} \exp(\mathbf{u}_w^\top \mathbf{v}_c))] \tag{13}$$

$$= \frac{\partial}{\partial \mathbf{u}_w}(-\mathbf{u}_o^\top \mathbf{v}_c) + \frac{\partial}{\partial \mathbf{u}_w} \log \sum_{w \in \text{Vocab}} \exp(\mathbf{u}_w^\top \mathbf{v}_c) \tag{14}$$

$$= \frac{\partial}{\partial \mathbf{u}_w}(-\mathbf{u}_o^\top \mathbf{v}_c) + \frac{1}{\sum_{w' \in \text{Vocab}} \exp(\mathbf{u}_{w'}^\top \mathbf{v}_c)} \frac{\partial}{\partial \mathbf{u}_w} \exp(\mathbf{u}_w^\top \mathbf{v}_c) \tag{15}$$

$$= \frac{\partial}{\partial \mathbf{u}_w}(-\mathbf{u}_o^\top \mathbf{v}_c) + \frac{\exp(\mathbf{u}_w^\top \mathbf{v}_c)}{\sum_{w' \in \text{Vocab}} \exp(\mathbf{u}_{w'}^\top \mathbf{v}_c)} \mathbf{v}_c \tag{16}$$

$$= \frac{\partial}{\partial \mathbf{u}_w}(-\mathbf{u}_o^\top \mathbf{v}_c) + \hat{\mathbf{y}}_w \mathbf{v}_c \tag{17}$$

$$= \begin{cases} (\hat{\mathbf{y}}_w - 1)\mathbf{v}_c & \text{, if } w = o \\ \hat{\mathbf{y}}_w \mathbf{v}_c & \text{, otherwise} \end{cases} \tag{18}$$

(e) (1 point) Write down the partial derivative of $\mathbf{J}_{\text{naive-softmax}}(\mathbf{v}_c, o, \mathbf{U})$ with respect to $\mathbf{U}$. Please break down your answer in terms of the column vectors $\frac{\partial \mathbf{J}(\mathbf{v}_c,o,\mathbf{U})}{\partial \mathbf{u}_1}$, $\frac{\partial \mathbf{J}(\mathbf{v}_c,o,\mathbf{U})}{\partial \mathbf{u}_2}$, $\cdots$, $\frac{\partial \mathbf{J}(\mathbf{v}_c,o,\mathbf{U})}{\partial \mathbf{u}_{|\text{Vocab}|}}$. No derivations are necessary, just an answer in the form of a matrix.

Solution:

Following the answer in (d), we can define $\frac{\partial \mathbf{J}_{\text{naive-softmax}}(\mathbf{v}_c,o,\mathbf{U})}{\partial \mathbf{U}} \in \mathbb{R}^{d \times |\text{Vocab}|}$ as follows:

$$\frac{\partial \mathbf{J}(\mathbf{v}_c, o, \mathbf{U})}{\partial \mathbf{U}} = \begin{bmatrix} \hat{\mathbf{y}}_1 \mathbf{v}_c & \hat{\mathbf{y}}_2 \mathbf{v}_c & \ldots & (\hat{\mathbf{y}}_o - 1)\mathbf{v}_c & \ldots & \hat{\mathbf{y}}_{|\text{Vocab}|} \mathbf{v}_c \end{bmatrix} \tag{19}$$

$$= \mathbf{v}_c (\hat{\mathbf{y}} - \mathbf{y})^\top \tag{20}$$

2. 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[5] uses a more sophisticated update rule with two additional steps.[6]

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. (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:

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

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?

Solution:

i. Instead of updating model parameters based solely on the current gradient, *momentum* smooths out the updates by accumulating an exponentially decaying average of past gradients. This approach prevents sudden shifts in the parameters due to noisy gradients and reduces oscillations when gradients fluctuate in direction. Additionally, *momentum* helps the parameter update maintain speed, which prevents learning from getting stuck in plateaus or local minima.

ii. Model parameters with smaller second moment estimates (i.e., lower uncentered variance in the gradients) will receive larger updates. Instead of applying the same learning rate to all parameters, the scaling operation $/\sqrt{\mathbf{v}}$ adapts the learning rate for each parameter. This allows parameters with infrequent or small gradients to be updated with larger step sizes, leading to faster convergence. Conversely, parameters with larger gradient magnitudes receive smaller updates, preventing overshooting and stabilizing the learning process. Overall, *adaptive learning rates* help control parameter-wise updates, enhancing the model's generalizability.

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

[6] 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`

(b) (4 points) Dropout[7] 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.

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 dropout paper linked.

Solution:

i.

$$\gamma = \frac{1}{1 - p_{\text{drop}}}$$

It's quite intuitive to write down the value of γ. Following provides the complete derivation:

$$\begin{aligned}
\mathbb{E}_{p_{\text{drop}}}[\mathbf{h}_{\text{drop}}] &= \mathbb{E}_{p_{\text{drop}}}[\gamma \mathbf{d} \odot \mathbf{h}] && (21)\\
&= \gamma \mathbf{h} \odot \mathbb{E}_{p_{\text{drop}}}[\mathbf{h}] && (22)\\
&= \gamma \mathbf{h} \odot \begin{bmatrix} 0 \times p_{\text{drop}} + 1 \times (1 - p_{\text{drop}}) \\ \vdots \\ \times D_h \end{bmatrix} && (23)\\
&= \gamma (1 - p_{\text{drop}}) \mathbf{h} && (24)
\end{aligned}$$

$$\gamma (1 - p_{\text{drop}}) \mathbf{h} = \mathbf{h} \Rightarrow \gamma = \frac{1}{1 - p_{\text{drop}}} \tag{25}$$

ii. During training, randomly masking a portion of neurons helps alleviate a phenomenon, called **co-adaptation**, introduced in the paper. Concretely speaking, dropout prevents each neuron from heavily depending on a specific set of other neurons to correct the mistake, which can lead to better generalizability.
For evaluation (or inference), dropout is disabled so all neurons contribute to the prediction, ensuring stable and consistent outputs. To align expected outputs at training and test time, one approach (called **inverted dropout**) scales activations during training by dividing the keep probability, while keeping test activations unchanged. Alternatively, training remains unchanged, and activations are scaled down during testing.

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

3. Neural Transition-Based Dependency Parsing (54 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 2.1.2 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:

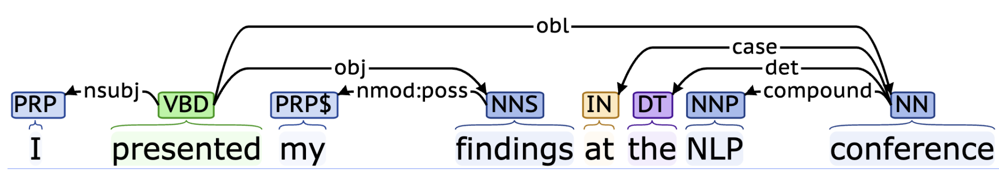
- A *stack* of words that are currently being processed.
- A *buffer* of words yet to be processed.
- A list of *dependencies* predicted by the parser.

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 presented my findings at the NLP conference"*. 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, presented, my, findings, at, the, NLP, conference]		Initial Configuration
[ROOT, I]	[presented, my, findings, at, the, NLP, conference]		`SHIFT`
[ROOT, I, presented]	[my, findings, at, the, NLP, conference]		`SHIFT`
[ROOT, presented]	[my, findings, at, the, NLP, conference]	presented→I	`LEFT-ARC`

Solution:

Stack	Buffer	New dependency	Transition
[ROOT]	[I, presented, my, findings, at, the, NLP, conference]		Initial Configuration
[ROOT, I]	[presented, my, findings, at, the, NLP, conference]		`SHIFT`
[ROOT, I, presented]	[my, findings, at, the, NLP, conference]		`SHIFT`
[ROOT, presented]	[my, findings, at, the, NLP, conference]	presented→I	`LEFT-ARC`
[ROOT, presented, my]	[findings, at, the, NLP, conference]		`SHIFT`
[ROOT, presented, my, findings]	[at, the, NLP, conference]		`SHIFT`
[ROOT, presented, findings]	[at, the, NLP, conference]	findings→my	`LEFT-ARC`
[ROOT, presented]	[at, the, NLP, conference]	presented→findings	`RIGHT-ARC`
[ROOT, presented, at]	[the, NLP, conference]		`SHIFT`
[ROOT, presented, at, the]	[NLP, conference]		`SHIFT`
[ROOT, presented, at, the, NLP]	[conference]		`SHIFT`
[ROOT, presented, at, the, NLP, conference]	[]		`SHIFT`
[ROOT, presented, at, the, conference]	[]	conference→NLP	`LEFT-ARC`
[ROOT, presented, at, conference]	[]	conference→the	`LEFT-ARC`
[ROOT, presented, conference]	[]	conference→at	`LEFT-ARC`
[ROOT, presented]	[]	presented→conference	`RIGHT-ARC`
[ROOT]	[]	ROOT→presented	`RIGHT-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.

Solution:

Each word is added to the stack once and removed once (like popping from a stack), resulting in $2n$ operations for a sentence with n words. Therefore, the overall time complexity remains $O(n)$. Note that `LEFT-ARC` removes the word second to the top of 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.

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`.

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`.

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) (20 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*.[8] 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 with dimension d. We then compute our prediction as:

$$\begin{aligned}
\mathbf{h} &= \text{ReLU}(\mathbf{xW} + \mathbf{b}_1) \\
\mathbf{l} &= \mathbf{hU} + \mathbf{b}_2 \\
\hat{\mathbf{y}} &= \text{softmax}(l)
\end{aligned}$$

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:

$$J(\theta) = CE(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{j=1}^{3} \mathbf{y}_j \log \hat{\mathbf{y}}_j$$

where $\mathbf{y}_j$ denotes the jth element of $\mathbf{y}$. To compute the loss for the training set, we average this $J(\theta)$ across all training examples.

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

i. Compute the derivative of $\mathbf{h} = \text{ReLU}(\mathbf{xW} + \mathbf{b}_1)$ with respect to $\mathbf{x}$. For simplicity, you only need to show the derivative $\frac{\partial h_i}{\partial x_j}$ for some index i and j. You may ignore the case where the derivative is not defined at 0.

Solution:

we have to notice that $\mathbf{x}$ is a $\mathbb{R}^{1\times dm}$ matrix, since:

$$\frac{\partial\,\text{ReLU}(z)}{\partial z} = \begin{cases} 1 & \text{if } z > 0, \\ 0 & \text{otherwise.} \end{cases}$$

Linear component expansion:

$$z = xW + b_1 \Rightarrow z_i = \sum_k x_k W_{k,i} + b_{1,i}.$$

where $W_{k,i}$ is the element at row k, column i of the weight matrix W.

Using the Chain rule, we get:

$$\frac{\partial h_i}{\partial x_j} = \begin{cases} W_{j,i} & \text{if } (xW + b_1)_i > 0, \\ 0 & \text{otherwise.} \end{cases}$$

ii. Recall in part 1b, we computed the partial derivative of $\mathbf{J}_{\text{naive-softmax}}(\mathbf{v}_c, o, \mathbf{U})$. Likewise, please compute the partial derivative of $J(\theta)$ with respect to the ith entry of $\mathbf{l}$, which is denoted as $\mathbf{l}_i$. Specifically, compute $\frac{\partial CE(\mathbf{y},\hat{\mathbf{y}})}{\partial \mathbf{l}_i}$, assuming that $\mathbf{l} \in \mathbb{R}^3$, $\hat{\mathbf{y}} \in \mathbb{R}^3$, $\mathbf{y} \in \mathbb{R}^3$, and the true label is c.

Hints: You may recall from part 1a, $\frac{\partial CE(\mathbf{y},\hat{\mathbf{y}})}{\partial \mathbf{l}_i} = \sum_j \frac{\partial CE(\mathbf{y},\hat{\mathbf{y}})}{\partial \hat{\mathbf{y}}_j} \frac{\partial \hat{\mathbf{y}}_j}{\partial \mathbf{l}_i}$, and $\frac{\partial CE(\mathbf{y},\hat{\mathbf{y}})}{\partial \hat{\mathbf{y}}_j} = 0$ if $j \neq c$.

Solution: For $\hat{y}_j = \frac{e^{l_j}}{\sum_{k=1}^3 e^{l_k}}$, the derivative $\frac{\partial \hat{y}_j}{\partial l_i}$ is:

$$\frac{\partial \hat{y}_j}{\partial l_i} = \begin{cases} \hat{y}_i(1-\hat{y}_i) & j = i \\ -\hat{y}_i\hat{y}_j & j \neq i \end{cases}$$

Using Chain rule,

$$\frac{\partial J}{\partial l_i} = \sum_{j=1}^3 \frac{\partial J}{\partial \hat{y}_j} \cdot \frac{\partial \hat{y}_j}{\partial l_i}$$

Since $\frac{\partial J}{\partial \hat{y}_j} = -\frac{y_j}{\hat{y}_j}$, and y_j is non-zero only when $j = c$, this simplifies to:

$$\frac{\partial J}{\partial l_i} = -\frac{1}{\hat{y}_c} \cdot \frac{\partial \hat{y}_c}{\partial l_i}$$

Combining these results, the derivative is:

$$\frac{\partial J}{\partial l_i} = \hat{y}_i - y_i$$

where y_i is the one-hot encoded true label (1 if $i = c$, 0 otherwise).

iii. 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.

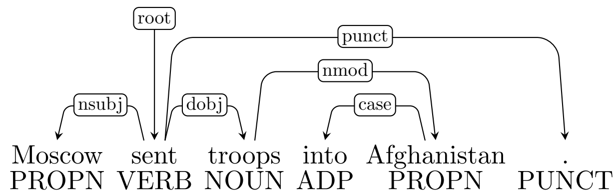
Hints:

- 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 up to **15 minutes** to train the model on the entire 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).

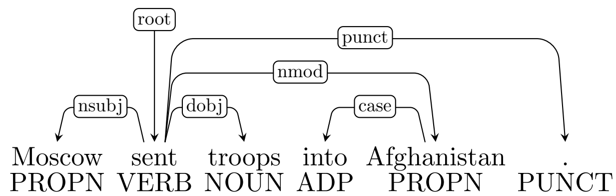
Deliverables:

- 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 written submission**.

(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[9]. 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[10]. 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:

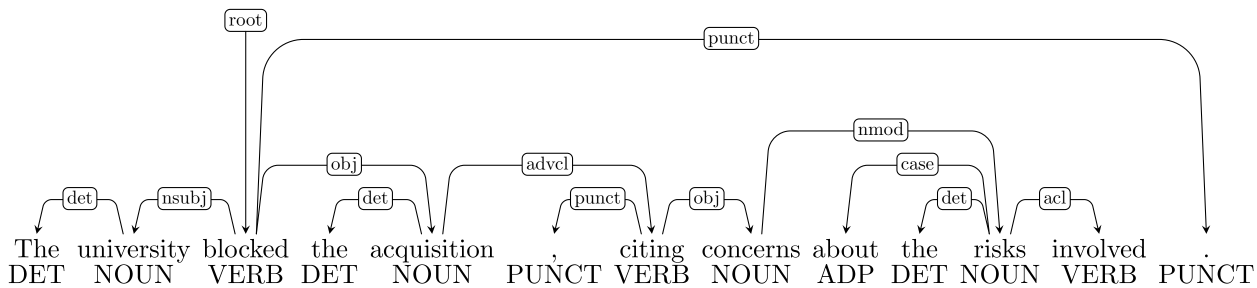
- **Error type**: Prepositional Phrase Attachment Error
- **Incorrect dependency**: troops → Afghanistan
- **Correct dependency**: sent → Afghanistan

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.`

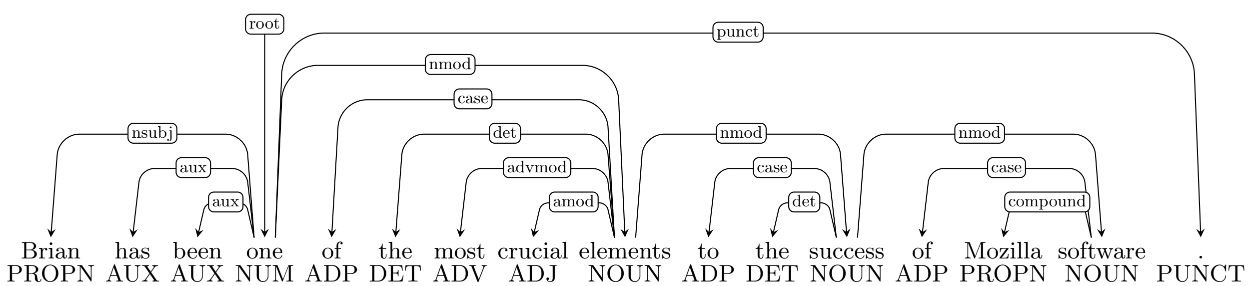
[9]For examples of prepositional phrases, see: https://www.grammarly.com/blog/prepositional-phrase/
[10]For examples of verb phrases, see: https://examples.yourdictionary.com/verb-phrase-examples.html

org[11] *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.*

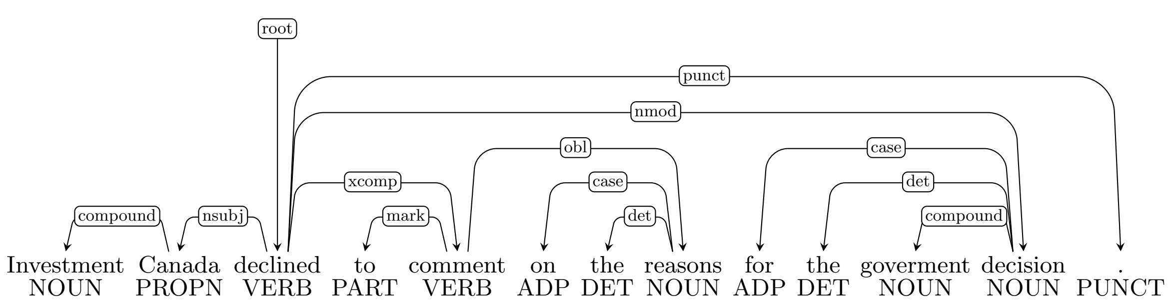
i.



ii.

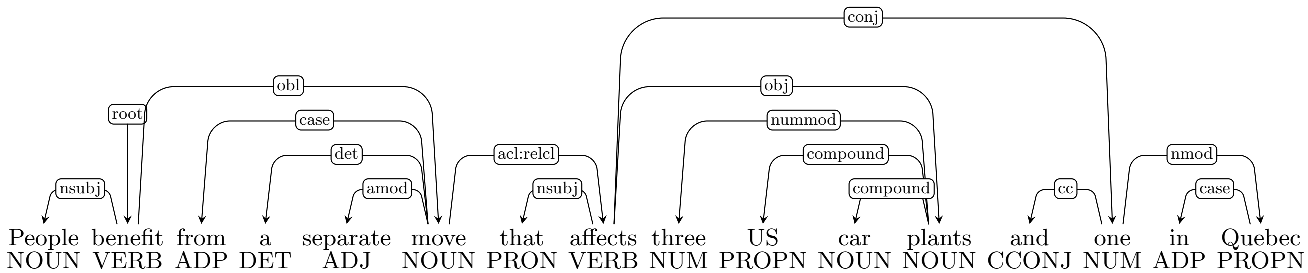


iii.



iv.

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



(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?

Submission Instructions

You shall submit this assignment on GradeScope as two submissions – one for "Assignment 2 [coding]" and another for 'Assignment 2 [written]":

1. Run the `collect_submission.sh` script to produce your `assignment2.zip` file.
2. Upload your `assignment2.zip` file to GradeScope to "Assignment 2 [coding]".
3. Upload your written solutions to GradeScope to "Assignment 2 [written]".