Assignment 2: n-Gram Part of Speech Tagger

LING 380/780, Fall 2021
Neural Network Models of Linguistic Structure

Assigned: Thursday, October 14
Due: Friday, October 29, 11:59 PM
Total Points: 100

In this assignment, you will build a *part of speech tagger*—a model that assigns parts of speech (i.e., grammatical categories like "noun" or "verb") to words in a text. You will do this by training a neural network that takes as input a word (with or without its surrounding context) and then predicts the part of speech of that word. For many words, the assignment of part of speech is straightforward: the word *bird* is always a noun, while the word *annoy* is always a verb. However, there are also cases of part of speech ambiguity: *break* can be either a noun or verb, depending on context.

For the programming portion, you are provided with an incomplete NumPy implementation of a multi-layer perceptron. You will need to complete the codebase, including the forward and backward passes for different types of network objects as well as the core of the stochastic gradient descent algorithm. The amount of code you will have to write is quite modest, but each line will require you to think very carefully. In some cases, you will need to derive array-based mathematical expressions before implementing them in NumPy.

After you have completed your implementation of the neural network, you will run some experiments to analyze the representations learned by your neural network.

1 Model Implementation

In this first part of the assignment, you will implement the part of speech tagger as a multi-layer perceptron.

1.1 Installation

For this assignment, you will continue to use NumPy, which you learned in the previous assignment. In addition, this assignment requires you to install the PyConll package, which provides an interface for reading CoNLL-X files. CoNLL-X (Buchholz and Marsi, 2006) is a common file format used in NLP. It represents natural language sentences where each token is annotated with grammatical properties, including its part of speech tag. Please install PyConll using `pip` by running the following command in Terminal (Mac OS and Linux) or Command Prompt (Windows).

```
pip install pyconll
```

1.2 Task Specification and Network Architecture

In the POS tagging task, the neural network tagger will receive an n-gram

$$w = w_{-k}w_{-k+1}\dots w_{-1}w_0w_1w_2\dots w_k,$$

where n is odd and $k = (n-1)/2$. The goal is to classify the middle token w_0 into one of following 17 possible parts of speech, defined by the Universal Dependencies project.

ADJ: adjective	ADP: adposition
ADV: adverb	AUX: auxiliary
CCONJ: coordinating conjunction	DET: determiner
INTJ: interjection	NOUN: noun
NUM: numeral	PART: particle
PRON: pronoun	PROPN: proper noun
PUNCT: punctuation	SCONJ: subordinating conjunction
SYM: symbol	VERB: verb
X: other	

Figure 1 shows the architecture of the neural network POS tagger you will implement for this task. This network consists of an embedding layer, an output layer, and a variable number of hidden layers determined by the hyperparameter h. Below we explain the various parts of the model.

- **Input Representation.** The input to the model is mini-batch of n-grams, represented as a 2D array of shape (batch size, n). Each row (i.e., set of entries along dimension 0) of the array represents an n-gram, while each column (i.e., set of entries along dimension 1) represents all the w_is in the batch for some position i where $-k \leq i \leq k$. The entries of the matrix are *indices*: integers that represent one of the possible tokens in the *vocabulary*, including the `[UNK]`, `[BOS]`, and `[EOS]` tokens.
- **Embedding Layer.** The embedding layer takes a batch of indices as input and replaces each index with its corresponding word embedding vector. The output of the embedding layer is a 3D array of shape (batch size, n, embedding size). The embedding layer is parameterized by a word embedding matrix of shape (vocabulary size, embedding size).

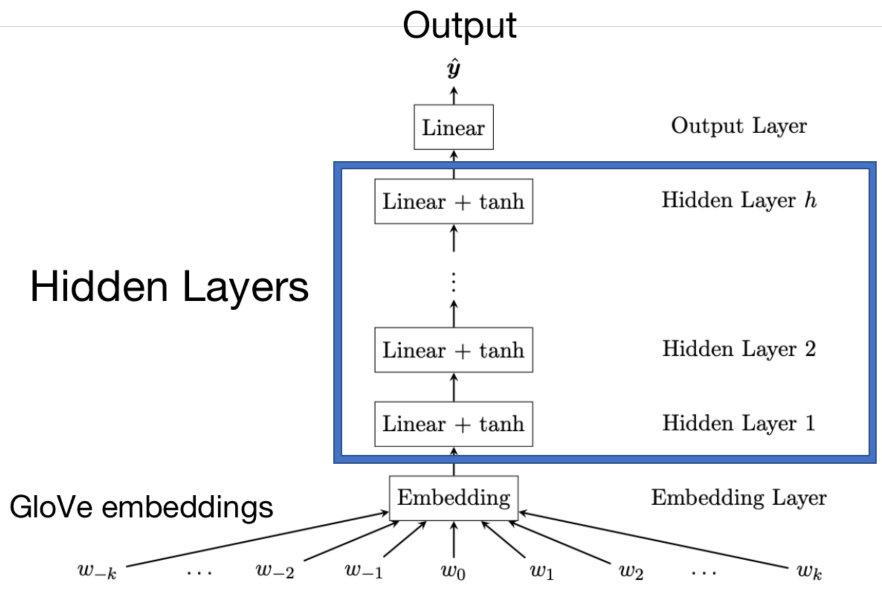

Figure 1: The multi-layer perceptron architecture for the n-gram POS tagger, where $k = (n-1)/2$.

Input

$\hat{\mathbf{Y}} = \mathrm{softmax}(\mathbf{O})$ $\quad \hat{\mathbf{Y}} \in \mathbb{R}^{b\times 17}$

$\mathbf{O} = \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}$ $\quad \mathbf{O} \in \mathbb{R}^{b\times 17}$

$\mathbf{H} = \tanh\left(\mathbf{E}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}\right)$ $\quad \mathbf{H} \in \mathbb{R}^{b\times h}$

$\mathbf{E} = \mathrm{GloVe}(\mathbf{X})$ $\quad \mathbf{E} \in \mathbb{R}^{b\times n\times e} \rightarrow \mathbb{R}^{b\times(ne)}$

$\mathbf{X} \in \mathbb{R}^{b\times n}$

Hyperparameters:
vocabulary size
b: batch size
n: length of n-gram
h: hidden size
e: embedding size
17: class size

$\mathbf{W}^{(1)} \in \mathbb{R}^{(ne)\times h}$
$\mathbf{W}^{(2)} \in \mathbb{R}^{h\times 17}$
$\mathbf{b}^{(1)} \in \mathbb{R}^{1\times h}$
$\mathbf{b}^{(2)} \in \mathbb{R}^{1\times 17}$

- Hyperbolic tangent:

$$\tanh x = \frac{\sinh x}{\cosh x} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1}$$

- softmax activation function:

$$\text{softmax}(\mathbf{x}) = \frac{e^{\mathbf{x}}}{\mathbf{1}^T e^{\mathbf{x}}}$$

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

- cross-entropy loss for a n-gram sample

$$L_{CE}(\text{softmax}(\hat{\mathbf{y}}), y) = -\ln(\text{softmax}(\hat{\mathbf{y}})_y) = -\ln\left(\frac{e^{\hat{\mathbf{y}}_y}}{\mathbf{1}^T e^{\hat{\mathbf{y}}}}\right) = -\ln\left(\frac{e^{\hat{\mathbf{y}}_y}}{\sum_{i=1}^{17} e^{\hat{\mathbf{y}}_i}}\right)$$

- Objective function

$$\theta^* = \arg\min_\theta \mathcal{L}$$

$$\mathcal{L} = \sum_{i=1}^{b} L_{CE}\left[\text{softmax}(\hat{\mathbf{Y}}_i), Y_i\right]$$

$$\theta = \begin{bmatrix} \mathbf{W}^{(1)} \\ \mathbf{W}^{(2)} \\ \mathbf{b}^{(1)} \\ \mathbf{b}^{(2)} \end{bmatrix}$$

- **Hidden Layers.** Each hidden layer consists of a linear layer followed by a tanh activation function. The output of the hidden layer, an array of shape (batch size, hidden size), is a batch of vectors known as *hidden representations*, where the *hidden size* is a hyperparameter of the model. Hidden representations are understood to be latent features computed by the network—we don't know what they represent or how they are interpreted. The first hidden layer takes an input of shape (batch size, $n \times$ embedding size) containing all the embeddings in the mini-batch concatenated together; the output of the Embedding layer must be reshaped to this format. The other hidden layers take an input of shape (batch size, hidden size) containing the outputs of the previous layers.
- **Output Layer.** The output layer, the final layer of the network, consists solely of a linear layer. For reasons of computational efficiency, we exclude the softmax activation from the output layer, and instead implement softmax as part of the cross-entropy loss function. The output of the output layer (which is the final output of the network) is an array of shape (batch size, 17) containing confidence scores assigned by the model to each of the 17 possible POS tags. These scores are on the *logit scale*, meaning that the softmax of the scores is interpreted as a probability distribution over the possible POS tags. For each row of the output, we interpret column containing the highest confidence score to represent the network's predicted POS tag for the corresponding item in the mini-batch.

Problem 1. Suppose our model uses the following hyperparameters.

- Vocabulary size: 100
- n (length of n-grams): 3 (i.e., $k = 1$)
- Embedding size: 50
- Hidden size: 16
- h (number of hidden layers): 1

How many trainable parameters does the neural network have? (In other words, how many array entries are there among the embedding matrix as well as the weight matrix and bias vector for all the hidden layers and the output layer?) Which part of the network has the most parameters?

1.3 Starter Code

The file archive for this assignment should have the following directory structure.

- `data`
  - `en_ewt-ud-train.conllu`

    - en_ewt-ud-dev.conllu
    - en_ewt-ud-test.conllu
    - glove_embeddings.txt
    - unambiguous_pos_tags.csv
- data_loader.py
- layers.py
- loss.py
- metrics.py
- model.py
- train.py

The three `.conllu` files in the `data` folder contain the sentences that we will be using for training, validating, and testing our POS tagger. The dataset files are in CoNLL-U format, a variant of CoNLL-X. The file `glove_embeddings.txt` is similar to the file `word2vec_embeddings.txt` from Assignment 1, except that the word embeddings are trained using the GloVe model (Pennington et al., 2014) instead of the word2vec model. The file `unambiguous_pos_tags.csv` will be used in Part 2 (Analysis) of the assignment.

The `.py` files in the root directory are Python *modules*—Python files containing code that can be used in other files and scripts using the `import` keyword. For example, the function `load_embeddings` in the file `data_loader.py` can be used as follows:

```python
# Import load_embeddings from the data_loader module
from data_loader import load_embeddings

# Call the load_embeddings function
all_tokens, all_embeddings = load_embeddings()
```

As you can see, the name of each module is simply its filename, but without the `.py` extension. A collection of modules within a folder, such as this one, is known as a *package*.

Now, examine the contents of each module in the package. Even though you only have to modify a relatively small number of locations in the code, it is important to understand the overall structure of the package and the role each module plays in the code. Broadly speaking, the modules are organized into the following three functional categories.

Data Interface. In the `data_loader` module, we have provided functions that read CoNLL-X files and convert them into Python data structures that you will use in other parts of the code. We have also provided (a slightly modified version of) the `load_embeddings` function from Assignment 1. When run the first time, this function loads the GloVe embeddings from the `glove_embeddings.txt` file and then saves it in binary format as a Pickle file. The second time it is run, it loads the binary file (much more quickly).

The `data_loader` module contains two important classes.

- `data_loader.Vocabulary`: A `Vocabulary` represents a collection of possible tokens and a mapping of each token form to a unique numerical identifier known as its *index*. You will need to create two `Vocabulary`s in this assignment: one that represents possible tokens in the text and one that represents possible POS tags.
- `data_loader.Dataset`: A `Dataset` represents a collection of n-grams, where n is odd, labeled with the POS tag of the token in the middle. You will use the `from_conll` function to load data from a CoNLL-X file and convert it into NumPy arrays, and you will use the `get_batches` function to divide the data into mini-batches. Like `load_embeddings`, the `from_conll` function will store a binary version of the data which will be loaded (more quickly) on subsequent calls.

Example:

```python
# Import everything from data_loader
from data_loader import *

# Create vocabularies
all_tokens, _ = load_embeddings()
token_vocab = Vocabulary(all_tokens +
                         ["[UNK]", "[BOS]", "[EOS]"])
pos_tag_vocab = Vocabulary(all_pos_tags)

# Load a CoNLL-U file and convert it to trigrams (n=3)
train_data = Dataset.from_conll("data/en_ewt-ud-train.conllu",
                                token_vocab, pos_tag_vocab,
                                ngram_size=3)

# Loop over mini-batches of size 16
for ngrams, pos_tags in train_data.get_batches(16):
    ...
```

Neural Network Components. The modules `layers`, `model`, and `loss` contain code used to implement the multi-layer perceptron and cross-entropy loss function. The functionality of each module is as follows.

- `layers` contains Python classes for three types of network layers: `Embedding`, `Linear`, and `Tanh`. You will need to complete the forward and backward computations in the implementations of these classes.
- `model` contains the `MultiLayerPerceptron` class, which represents the full neural network that you will train. An object of this class contains within it objects of the various layer types, which are assembled into the architecture depicted above. The `forward` and `backward` methods perform the forward and backward computations by calling the forward and backward methods for each layer in the appropriate order (forward for the forward computation, in reverse for the backward computation).
- `loss` contains code that combines the softmax activation function of the neural network with the cross-entropy loss function. As you will soon discover, implementing these two components together is more efficient than implementing them separately. You will need to implement the backward computation of this combined softmax–loss function unit.

At the heart of these three modules is the `layers.Layer` class, an abstract class[1] that represents neural network layers. Please study the API for our neural networks by looking at the functions declared by `Layer`. The abstract class comes with code for resetting gradients to 0 (the `clear_grad` function) and updating parameters during SGD (the `update` function). Each subclass of `Layer` must implement a forward pass (`forward` function) and a backward pass (`backward` function) for backpropagation. The constructor (`__init__` method) of each `Layer` subclass must declare its parameters and place them within the `params` dict. `Layer` objects themselves can be called as functions; this has the effect of calling the `forward` function while saving the input to forward. For example:

```python
import numpy as np
from layers import Linear

linear = Linear(2, 3)
x = np.random.rand(5, 2)

# Calls linear.forward
print(linear(x))

# Stored layer input
print(linear.x)
```

When working with `Layers`, you should always run the forward pass by calling the `Layer` object, and never call the `forward` function directly.

[1]Please read this blog post if you don't know what an abstract class is: https://www.geeksforgeeks.org/abstract-classes-in-python/

Notice that `model.MultiLayerPerceptron` and `loss.CrossEntropySoftmaxLoss` are also subclasses of `layers.Layer`, even though they represent objects that are not typically thought of as “neural network layers.” This is because both classes can be included in a computation graph, and therefore require a forward and backward pass implementation, making them compatible with the `Layer` interface.

Training and Evaluation Code. The remaining modules contain code for training, validating, and testing a POS tagger.

- `train` contains functions to train and test your network. Below `if __name__ == "__main__"`, the `train.py` file also contains a Python script that can be called from the command line.[2] Doing so will train a network using SGD with one particular configuration of hyperparameters and test the performance with the best validation accuracy.
- `metrics` provides code for the assessment of average loss and accuracy, which will be used during training.

1.4 Forward Computations

For this part of the assignment, you will implement the forward pass for the `layers.Tanh` activation function and `layers.Linear` layer. The forward pass of the `layers.Embedding` layer and `model.MultiLayerPerceptron` model has already been implemented for you.

Problem 2. Implement the `forward` function for `layers.Tanh`. Please use the `np.tanh` function for the forward pass.

Problem 3. Implement the `forward` function for `layers.Linear`. The `forward` function should return the value of the linear map

$$\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$

for input $\mathbf{x}$, where $\boldsymbol{W}$ is the weight matrix of the layer and $\boldsymbol{b}$ is the bias vector. To do this, you will need to have access to the parameters of the linear layer, which are stored in the `params` dict. These can be retrieved as `self.params["w"]` for the weight matrix and `self.params["b"]` for the bias vector.

Hints:

1. Be sure to do your computations via matrix multiplication and ***do not use loops*** to calculate the separate values—otherwise your implementation will be too slow!

[2]The line `if __name__ == "__main__"` allows a Python file to be used either as a module or a script. The code under the `if` statement is not executed when the file is used as a module.

2. Your `Linear` must be able to apply to matrices of shape (batch size, hidden size) or (batch size, $n \times$ embedding size), but the weight matrix has shape (output size, input size) and the bias vector has shape (output size,). This means that you will not be able to perform the matrix multiplication exactly as in the equation given above. Instead, you may need to transpose one of the arrays in the forward pass in order to make the matrix multiplication work.

1.5 Backward Computations

Next, you will implement the backward pass for the `layers.Tanh` activation function and `model.MultiLayerPerceptron` model. The backward pass of `layers.Embedding` and `layers.Linear` has already been implemented for you.

Problem 4. Implement the `backward` functions for `layers.Tanh`. Use the following formula for the gradient of tanh:

$$\frac{d}{dx}\tanh(x) = 1 - \tanh(x)^2.$$

Recall that the input to `backward` is the Jacobian

$$\boldsymbol{\delta} = \frac{\partial \mathcal{L}}{\partial \tanh(\boldsymbol{x})}.$$

The shape of $\boldsymbol{\delta}$ is the same as that of the output $\tanh(\boldsymbol{x})$. Each entry of $\boldsymbol{\delta}$ contains the partial derivative of $\mathcal{L}$ with respect to the corresponding entry of $\tanh(\boldsymbol{x})$. The return value of `backward` needs to be the Jacobian

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{x}},$$

represented as an array of the same shape as $\boldsymbol{x}$ where each entry contains the partial derivative of $\mathcal{L}$ with respect to the corresponding entry of $\boldsymbol{x}$.

Problem 5. Implement the `backward` functions for `layers.Linear`. In addition to returning the Jacobian $\partial\mathcal{L}/\partial\boldsymbol{x}$, you will also need to update the values of `self.grad["w"]` and `self.grad["b"]`, which contain the gradients $\nabla_{\boldsymbol{W}}\mathcal{L} = (\partial\mathcal{L}/\partial\boldsymbol{W})^\top$ and $\nabla_{\boldsymbol{b}}\mathcal{L} = (\partial\mathcal{L}/\partial\boldsymbol{b})^\top$ respectively.[3] The gradients are represented as arrays of the same shape as their respective parameters. Your code will need to compute the gradients $\nabla_{\boldsymbol{W}}\mathcal{L}$ and $\nabla_{\boldsymbol{b}}\mathcal{L}$ using the chain rule and add them to the existing values stored in `self.grad`. (As we learned in class, this is called "gradient accumulation.")

Problem 6. Implement the `backward` function of `model.MultiLayerPerceptron`. The `forward` function has already been implemented for you: it first applies the `forward` of the

[3]If $\boldsymbol{W} \in \mathbb{R}^{p\times q}$, then technically $\nabla_{\boldsymbol{W}}\mathcal{L}$ is a $p \times q$ matrix while $\partial\mathcal{L}/\partial\boldsymbol{W}$ is a $1 \times p \times q$ 3D array (a "tensor"). The "transpose operation" is really just getting rid of that first dimension with only one row.

embedding layer, then reshapes its output to concatenate the word embeddings together, then applies the `forward` of each of the layers. Remember that `model.MultiLayerPerceptron` *does not* include the softmax function.

As with `layers.Tanh`, the input to `backward` will be the Jacobian $\boldsymbol{\delta}$:

$$\boldsymbol{\delta} = \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{y}}},$$

where $\hat{\boldsymbol{y}}$ is the output shown in Figure 1 (which does not include softmax). Your code should iterate through the list of layers in `self.layers` (in reverse order), applying each layer's `backward` method to compute a new value of $\boldsymbol{\delta}$ representing the partial derivative of the loss with respect to the input to that layer, in accordance with the chain rule. For example, if $\boldsymbol{z}$ is the input to the output layer, then we can push $\boldsymbol{\delta}$ one step backward through the output layer as follows:

$$\boldsymbol{\delta} \leftarrow \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}} = \boldsymbol{\delta}\frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{z}}.$$

(Notice that the provided implmentations of `layers.Embedding.backward` and `layers.Linear.backward` store the gradients with respect to their parameters in the `grad` dict in addition to returning the gradients with respect to their inputs.)

Hints:

1. As with the previous problem, you need to express the computations in terms of matrix operations, rather than `for` loops. Once again, work through the derivatives on pencil and paper. First, apply the chain rule to compute the derivatives with respect to individual units, weights, and biases. Next, take the formulas you've derived, and express them in matrix form. You should be able to express all of the required computations using only matrix addition, matrix multiplication, matrix transpose, and elementwise operations—no `for` loops!

2. Notice that the output and $\boldsymbol{\delta}$ of the `Embedding` layer has shape (batch size, n, embedding size), while the output of the `backward` function for the first `Linear` layer (i.e., the first layer in the `layers` list) has shape (batch size, $n \times$ embedding size). When propagating the gradient to the `Embedding`, you will need to reshape $\boldsymbol{\delta}$ to the correct shape by un-concatenating the word embeddings in the n-gram.

1.6 Stochastic Gradient Descent

SGD is implemented by the `layers.Layer.update` function, which updates the parameters (in the `params` dict) of a layer based on gradients (in the `grad` dict) previously computed by `backward`.

Problem 7. Implement the `update` function for the `layers.Layer` base class. This will implement SGD for all the layers in `layers` in one fell swoop, since those layers do not override the `update` function.

Problem 8. Implement the `update` function for `models.MultiLayerPerceptron`. This function will need to call `update` on the embedding layer as well as all the hidden layers and output layer of the multi-layer perceptron.

1.7 Implementing Softmax and Cross-Entropy Loss

Now, you will implement the softmax activation function and the cross-entropy loss function. Recall that these are defined as follows:

$$\text{softmax}(\boldsymbol{x}) = \frac{e^{\boldsymbol{x}}}{\mathbf{1}^\top \boldsymbol{x}}$$

$$L_{\text{CE}}(\text{softmax}(\hat{\boldsymbol{y}}), y) = -\ln(\text{softmax}(\hat{\boldsymbol{y}})_y) = -\ln\left(\frac{e^{\hat{y}_y}}{\sum_{i=1}^{17} e^{\hat{y}_i}}\right).$$

Unfortunately, the softmax function suffers from issues of *numerical stability*: because e^x grows extremely quickly relative to x, implementing softmax directly from its definition may lead to overflow errors. To try to avoid this, we shift $\boldsymbol{x}$ by a constant factor so that $e^{\boldsymbol{x}}$ will not be too large.

Problem 9. Implement the `softmax` function in the `loss` module. On input $\hat{\boldsymbol{y}}$, this function should return the vector

$$\text{softmax}(\hat{\boldsymbol{y}} - c),$$

where

$$c = \max_i(\hat{y}_i)$$

is the largest (i.e., most positive) entry of $\hat{\boldsymbol{y}}$. The input and output to `softmax` should both be arrays of shape (batch size, 17). The constant c should be computed separately for each logit vector in the batch (i.e., the max should be taken along the last dimension).

Problem 10. Next, verify that your code for Problem 9 is a valid implementation of softmax. Prove that for any input vector $\boldsymbol{x}$ and scalar c,

$$\text{softmax}(\boldsymbol{x}) = \text{softmax}(\boldsymbol{x} + c).$$

That is, show that the softmax function is invariant to constant offsets in the input. This means that the stabilization trick of subtracting the maximum entry of $\hat{\boldsymbol{y}}$ does not affect the output of softmax.

To complete the code for your neural network POS tagger, you will simultaneously implement the backward pass for both the cross-entropy loss function and the softmax activation function. It turns out that there is a simple mathematical expression for the gradient of these two units put together, which makes it more efficient to implement them as a single layer than as two separate layers.

Problem 11. Implement the `forward` function of `CrossEntropySoftmaxLoss` using your own implementation of softmax from Problem 10.

Problem 12. Implement the `backward` function of `CrossEntropySoftmaxLoss`. Since the δ of the loss function is always 1[4], this `backward` function does not take a `delta` parameter. Its output should be the $\boldsymbol{\delta}$ for the `model.MultiLayerPerceptron` model:

$$\boldsymbol{\delta} = \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{y}}} = \frac{\partial}{\partial \hat{\boldsymbol{y}}} L_{\text{CE}}(\text{softmax}(\hat{\boldsymbol{y}}), y).$$

As before, your output should be a matrix of shape (batch size, 17) where each row represents an example in the input batch, and each column contains the derivative of the loss with respect to the corresponding column of $\hat{\boldsymbol{y}}$.

Hints:

1. Using pencil and paper, write out the definition of $L_{\text{CE}}(\text{softmax}(\hat{\boldsymbol{y}}), y)$ and expand this expression. Then, compute the partial derivative of this expression with respect to each entry $\hat{y}_i$ of the output layer output.

2. After you have computed $\partial \mathcal{L} / \partial \hat{y}_i$ by hand, try to find a clean, array-based expression for $\boldsymbol{\delta}$.

1.8 Training Code

The final step is to train the model. The functions `train_epoch` and `run_trial` from `train` implement the main training procedure. However, `train_epoch`, which trains a given model for a single epoch, is incomplete—you are responsible for filling in the gaps.

Problem 13. Fill in the marked lines of code in the main loop of `train.train_epoch` (starting from line 31 in the file). This `for`-loop is iterating over mini-batches of data. You will need to implement one step of the SGD algorithm: first clear the stored gradients, then perform the forward pass, then perform the backward pass, then update the parameters of the model.

Once you have implemented the training loop, there are two ways you can train the model. The first is to call the `train.py` file as a script. You can do this from Terminal (Mac OS and Linux) or Command Prompt (Windows) by calling:

[4]Noting that $\mathcal{L}$ is the sum of losses, convince yourself that this is the case.

```
python train.py
```

or by executing a system call in Jupyter Notebook:

```
!python train.py
```

Another way to train the model is to recreate the training script at the bottom of the `train.py` file in your own script or in a Jupyter Notebook. You will need to import the `run_trial` function as follows:

```
from train import run_trial
```

We will not grade any scripts you choose to write for training the model, including the script at the bottom of `train.py`. Please do not include any extra Python scripts or Jupyter Notebooks with your submission.

2 Analysis

In the second part of the assignment, you will try to understand what your neural network has learned during training.

2.1 Part of Speech Ambiguity

Most English words only have one part of speech; for these words, the network simply needs to memorize their POS tag. Words with multiple parts of speech are more challenging for the model. Can your trained network handle these more difficult cases?

Problem 14. Find three words that can take on more than one part of speech. For each of your three words, give two 3-grams with your word in the middle: one that forces your word to have one part of speech, and one that forces your word to have a different part of speech.

Problem 15. Train a 3-gram POS tagger and feed your six 3-grams from Problem 14 into your POS tagger. Report the testing accuracy of your model; make sure that it is at least 75%. What POS tags does your tagger assign to the six 3-grams? For which of the 3-grams does your model make a correct prediction?

Hints:

1. Try using the following hyperparameters for training your model: learning rate 0.1, batch size 5, step size 4, $\gamma = .25$, 1 layer.
2. Use the following code template in a Python script or Jupyter Notebook to run your model on a 3-gram.

```python
# Create Vocabularies
token_vocab = \
    Vocabulary(all_tokens + ["[UNK]", "[BOS]", "[EOS]"])
pos_tag_vocab = Vocabulary(all_pos_tags)

# Prepare model input
tokens = ["Hello", "world", "!"]
ngrams = np.array(token_vocab.get_ngrams(tokens, 3))[1:2]

# Get model output
predictions = model(ngrams).argmax(axis=-1)
pos_tags = [pos_tag_vocab.get_form(p) for p in predictions]
print(pos_tags)
```

Problem 16. Find three symmetrical skip-grams with window size 1 (i.e., three 3-grams with the middle word missing) such that only words belonging to one particular part of speech can be inserted into the middle of the skip-gram. Then, form three 3-grams by inserting `[UNK]` into the middle of your three skip-grams. What POS tag is assigned to your three 3-grams by the POS tagger you trained in Problem 15?

2.2 POS and Word Embeddings

The POS tagging model learns embeddings that are optimized to carry out the tagging task. In this final part, you will consider the structure of this embedding space and how it facilitates the task of POS tagging. To do this, we will focus our attention on the embeddings of POS-unambiguous words, i.e., words that appear tagged with only a single part of speech in the training data. POS-ambiguous words will be represented as some combination of the multiple parts of speech with which they are associated (sensitive to the frequency with which they occur in each), and therefore will not be easily interpretable.

In the `data` folder we have provided with this assignment, there is a CSV file `unambiguous_pos_tags.csv` which contains all of the POS-unambiguous words from the training set. You can load these words, together with their corresponding POS tags, using Python's `csv` module:

```python
import csv

# Open a CSV file for reading
with open("data/unambiguous_pos_tags.csv", "r") as f:
    # Convert the CSV file to a list, remove header row
    unambig_pos_tags = list(csv.reader(f))[1:]
```

In order to explore the high-dimensional embeddings of these words, we must perform some sort of dimensionality reduction. In this case, we will use a technique called *t-distributed stochastic neighbor embedding*, or *t*-SNE. (You don't need to know what this is for the assignment, but if you are curious, you can read about it at the end of Chapter 1 of the course notes.) The important point is that nearby points in the low-dimensional space that *t*-SNE produces will correspond to nearby points in the high-dimensional space of word embeddings—i.e., the embedding will be (roughly) isometric. To create *t*-SNE embeddings, you need to look up the embedding corespond to the words in the `unambig_pos_tags` list you just created. Given an embedding matrix `embeddings`, you can do this for just the first 1000 embeddings—performing *t*-SNE for larger sets gets computationally expensive—and produce a 2-dimensional plot of the result as follows:

```python
from sklearn.manifold import TSNE  # Creates t-SNE embeddings
from matplotlib.pyplot as plt  # Creates plots


def plot_embeddings_by_pos(tsne_embeddings, unambig_pos_tags,
                           token_vocab, pos_tag_vocab):
    """
    Plots a set of t-SNE embeddings and organizes the
    points by POS tag.
    """
    for pos in pos_tag_vocab:
        # Get indices for all words whose POS tag is pos
        indices = [token_vocab.get_index(w)
                   for w, p in unambig_pos_tags if p == pos]

        # Remove indices outside of the first 1,000 words
        indices = [i for i in indices if i < 1000]

        # Add the points for these indices to the plot,
        # assigning a unique color to this POS tag
        plt.plot(tsne_embeddings[indices, 0],
                 tsne_embeddings[indices, 1],
                 marker=".",
                 linestyle="",
                 markersize=12,
                 label=pos)

    plt.legend(loc="best", bbox_to_anchor=(-0.1, 1.1))

```

```python
# You can omit if __name__ == "__main__": if running in a
# Jupyter Notebook
if __name__ == "__main__":
    # Extract embeddings from a model
    embeddings = model.embedding_layer.params["embeddings"]
    embeddings = embeddings[:1000]

    # Fit t-SNE embeddings
    tsne_model = TSNE(n_components=2)
    tsne_embeddings = tsne_model.fit_transform(embeddings)

    # Plot the embeddings
    plot_embeddings_by_pos(tsne_embeddings, unambig_pos_tags,
                           token_vocab, pos_tag_vocab)
```

Problem 17. First, initialize a `MultiLayerPerceptron` model *without loading pre-trained GloVe embeddings* and create t-SNE plots for the model's randomly initialized embeddings. Then, train the model until it reaches at least 75% accuracy, and create t-SNE plots for the model's (trained) word embeddings. How do these plots compare? What does this tell us about the information encoded in the network's word embeddings pre- and post-training?

Now create a t-SNE plot for pre-trained GloVe or word2vec embeddings, loaded from `glove_embeddings.txt` or the `word2vec_embeddings.txt` file from Assignment 1, that have not undergone any POS-tag training. What can you conclude about the degee to which the pre-trained embeddings encode information about part of speech? Given your knowledge of word2vec (assume that GloVe is also based on distributional semantics), is this what you would expect? Why?

3 Submission Instructions

To submit your completed assignment, please upload the following files to CodePost. Please ensure that your files have the same filenames as indicated below. **Failure to submit your assignment correctly will result in a deduction of 5 points.**

- All the modules in the provided Python code package, with your code filled in. Do not include the `data` directory, and do not change any of the filenames in the package.
- A Markdown document called `assignment2.md`, containing your responses to Problems 1, 10, 14, 15, 16, and 17.
- Any images of t-SNE plots that are embedded in your `assignment2.md` file for Problem 17.

References

Sabine Buchholz and Erwin Marsi. 2006. CoNLL-X Shared Task on Multilingual Dependency Parsing. In *Proceedings of the Tenth Conference on Computational Natural Language Learning (CoNLL-X)*, pages 149–164, New York, NY, USA. Association for Computational Linguistics.

Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. Glove: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar. Association for Computational Linguistics.