# -\*- coding: utf-8 -\*-

"""a1.ipynb

Automatically generated by Colaboratory.

Original file is located at

https://colab.research.google.com/drive/1Ez0ICGL6wlW22OxU\_UBJeJ\_8h427R\_Bo

# CSC413 Assignment 1: Word Embeddings

\*\*Deadline\*\*: February 5, 2021 by 10pm

\*\*Submission\*\*: Submit a PDF report containing your code, outputs,

and your written solutions.

You may export the completed notebook, but if you do so

\*\*it is your responsibly to make sure that your code and answers do not get cut off\*\*.

\*\*Late Submission\*\*: Please see the syllabus for the late submission criteria.

\*\*Working with a partner\*\*: You may work with a partner for this assignment.

If you decide to work with a partner, please create your group on Markus by

February 5, 10pm, even if you intend to use grace tokens. Markus does not allow

you to create groups past the deadline, even if you have grace tokens remaining.

Based on an assignment by George Dahl, Jing Yao Li, and Roger Grosse

In this assignment, we will make a neural network that can predict the next word

in a sentence given the previous three. We will apply an idea called \*weight sharing\*

to go beyond multi-layer perceptrons with only fully-connected layers.

We will also solve this problem problem twice: once in numpy, and once

using PyTorch. When using numpy, you'll implement the backpropagation

computation manually.

The prediction task is not very interesting on its own, but in learning to predict

subsequent words given the previous three, our neural networks will learn

about how to \*represent\* words. In the last part of the assignment, we'll explore

the \*vector representations\* of words that our model produces, and analyze these

representations.

You may modify the starter code, including changing the signatures of helper

functions and adding/removing helper functions. However, please make sure that your

TA can understand what you are doing and why.

"""

import pandas

import numpy as np

import matplotlib.pyplot as plt

import torch

import torch.nn as nn

import torch.optim as optim

"""## Question 1. Data

With any machine learning problem, the first thing that we would want to do

is to get an intuitive understanding of what our data looks like.

Download the file `raw\_sentences.txt` from Quercus.

If you're using Google Colab, upload the file to Google Drive.

Then, mount Google Drive from your Google Colab notebook:

"""

from google.colab import drive

drive.mount('/content/gdrive')

"""Find the path to `raw\_sentences.txt`:"""

file\_path = '/content/gdrive/My Drive/CSC413/A/A1/raw\_sentences.txt' # TODO - UPDATE ME!

"""You might find it helpful to know that you can run shell commands (like `ls`) by

using `!` in Google Colab, like this:

"""

# !ls /content/gdrive/My\ Drive/

# !mkdir /content/gdrive/My\ Drive/CSC413

"""The following code reads the sentences in our file, split each sentence into

its individual words, and stores the sentences (list of words) in the

variable `sentences`.

"""

sentences = []

for line in open(file\_path):

words = line.split()

sentence = [word.lower() for word in words]

sentences.append(sentence)

"""There are 97,162 sentences in total, and

these sentences are composed of 250 distinct words.

"""

vocab = set([w for s in sentences for w in s])

print(len(sentences)) # 97162

print(len(vocab)) # 250

"""We'll separate our data into training, validation, and test.

We'll use 10,000 sentences for test, 10,000 for validation, and

the rest for training.

"""

test, valid, train = sentences[:10000], sentences[10000:20000], sentences[20000:]

"""### Part (a) -- 2 pts

Display 10 sentences in the training set.

Explain how punctuations are treated in our word representation, and how words

with apostrophes are represented.

(Note that for questions like this, you'll need to supply both your code \*\*and\*\*

the output of your code to earn full credit.)

"""

# Your code goes here

for i in range(10):

print(train[i])

"""Punctuations are treated as a single words in our word representation exceopt for apostrophes, it will combine with the rest of the word/letters after the apostrophes and it will be inside a double quotation mark in the display.

### Part (b) -- 2 pts

What are the 10 most common words in the vocabulary? How often does each of these

words appear in the training sentences? Express the second quantity a percentage

(i.e. number of occurrences of the word / total number of words in the training set).

These are good quantities to compute, because one of the first things that most

machine learning model will learn is to predict the \*\*most common\*\* class.

Getting a sense of the distribution of our data will help you understand our

model's behaviour.

You might find Python's `collections.Counter` class helpful.

"""

# Your code goes here

from collections import Counter

count = Counter([item for sublist in train for item in sublist])

common\_word = sorted(count, key=count.get,reverse=True)

times = sorted(count.values(),reverse=True)

total = sum(count.values())

print("10 most common words in the vocabulary : ", common\_word[:10])

print("total time each of these words appear in the training sentences : ",times[:10])

print("Precentage of these words appear in the training sentences : ", [x / total for x in times[:10]])

"""### Part (c) -- 4 pts

Complete the helper functions `convert\_words\_to\_indices` and

`generate\_4grams`, so that the function `process\_data` will take a

list of sentences (i.e. list of list of words), and generate an

$N \times 4$ numpy matrix containing indices of 4 words that appear

next to each other. You can use the constants `vocab`, `vocab\_itos`,

and `vocab\_stoi` in your code.

"""

# A list of all the words in the data set. We will assign a unique

# identifier for each of these words.

vocab = sorted(list(set([w for s in train for w in s])))

# A mapping of j => word (string)

vocab\_itos = dict(enumerate(vocab))

# A mapping of word => its j

vocab\_stoi = {word:j for j, word in vocab\_itos.items()}

def convert\_words\_to\_indices(sents):

"""

This function takes a list of sentences (list of list of words)

and returns a new list with the same structure, but where each word

is replaced by its j in `vocab\_stoi`.

Example:

>>> convert\_words\_to\_indices([['one', 'in', 'five', 'are', 'over', 'here'],

['other', 'one', 'since', 'yesterday'],

['you']])

[[148, 98, 70, 23, 154, 89], [151, 148, 181, 246], [248]]

"""

# Write your code here

ans = []

curr = 0

for i in sents:

ans.append([])

for j in range(len(i)):

ans[curr].append(vocab\_stoi[i[j]])

curr += 1

return ans

def generate\_4grams(seqs):

"""

This function takes a list of sentences (list of lists) and returns

a new list containing the 4-grams (four consequentively occuring words)

that appear in the sentences. Note that a unique 4-gram can appear multiple

times, one per each time that the 4-gram appears in the data parameter `seqs`.

Example:

>>> generate\_4grams([[148, 98, 70, 23, 154, 89], [151, 148, 181, 246], [248]])

[[148, 98, 70, 23], [98, 70, 23, 154], [70, 23, 154, 89], [151, 148, 181, 246]]

>>> generate\_4grams([[1, 1, 1, 1, 1]])

[[1, 1, 1, 1], [1, 1, 1, 1]]

"""

# Write your code here

ans = []

for i in seqs:

for j in range(len(i)):

if j <= len(i) - 4 :

ans.append([i[j], i[j+1], i[j+2], i[j+3]])

return ans

def process\_data(sents):

"""

This function takes a list of sentences (list of lists), and generates an

numpy matrix with shape [N, 4] containing indices of words in 4-grams.

"""

indices = convert\_words\_to\_indices(sents)

fourgrams = generate\_4grams(indices)

return np.array(fourgrams)

train4grams = process\_data(train)

valid4grams = process\_data(valid)

test4grams = process\_data(test)

"""## Question 2. MLP Math

Suppose we were to use a 2-layer multilayer perceptron to solve this prediction

problem. Our model will look like this:

<img src="https://www.cs.toronto.edu/~lczhang/321/hw/p2\_model1.png" />

\begin{align\*}

\bf{x} &= \text{concatenation of the one-hot vector for words 1, 2 and 3} \\

\bf{m} &= \bf{W^{(1)}} \bf{x} + \bf{b^{(1)}} \\

\bf{h} &= \textrm{ReLU}(\bf{m}) \\

\bf{z} &= \bf{W^{(2)}} \bf{h} + \bf{b^{(2)}} \\

\bf{y} &= \textrm{softmax}(\bf{z}) \\

\end{align\*}

### Part (a) -- 2 pts

What is the shape of the input vector $\bf{x}$?

What is the shape of the output vector $\bf{y}$?

Let $k$ represent the size of the hidden layer. What are the

dimension of $W^{(1)}$ and $W^{(2)}$?

### Part (b) -- 2 pts

Draw a computation graph for $\bf{y}$. Your graph should include

the quantities $\bf{W^{(1)}}$, $\bf{W^{(2)}}$, $\bf{b^{(1)}}$, $\bf{b^{(2)}}$,

$\bf{x}$, $\bf{m}$, $\bf{h}$, $\bf{z}$ and $\bf{y}$.

### Part (c) -- 3 pts

Derive the gradient descent update rule for ${\bf W}^{(2)}$.

You should begin by deriving the update rule for $W^{(2)}\_{ij}$,

and then vectorize your answer. Assume that we will use the softmax

activation and cross-entropy loss.

Note: if you use the derivative of the softamx activation and

the cross-entropy loss, you \*\*must\*\* derive them.

### Part (d) -- 4 pts

What would be the update rule for $W^{(2)}\_{ij}$, if we use the square loss

$\mathcal{L}\_{SE}(\bf{y}, \bf{t}) = \frac{1}{2}(\bf{y} - \bf{t})^2$ ?

Show that we will not get good gradient signal

to update $W^{(2)}\_{ij}$ if we use this square loss.

### Part (e) -- 4 pts

In this question, we'll show a similar issue with using

the sigmoid activation. Let's assume we have a deep neural network

as follows:

\begin{align\*}

h\_1 &= \sigma(w\_1 x + b\_1) \\

h\_2 &= \sigma(w\_2 h\_1 + b\_2) \\

\dots

\end{align\*}

where, for simplicity, $x$, $w\_1$, $b\_1$, $h\_1$, $w\_2$, $b\_2$, $h\_2$, etc., are all scalars. Show that

\begin{align\*}

|\frac{\partial h\_1}{\partial x}| \le \frac{1}{4} |w\_1|

\end{align\*}

In order to do so, you will need to first

show that $\sigma'(z) = \sigma(z) (1 - \sigma(z))$ (worth 1 point).

Include a plot (or sketch) of the function $\sigma'(z)$ (worth 1 point).

### Part (f) -- 2 pts

Continue from the previous question, show that for a deeper neural network.

\begin{align\*}

|\frac{\partial h\_N}{\partial x}| \le \frac{1}{4^N} |w\_1| |w\_2| \cdots |w\_N|

\end{align\*}

What would be a problem with this result?

### Part (g) -- 1 pts

Would we have the same issue as in part(f) we we replaced the sigmoid activation

with ReLU activations? Why or why not?

## Question 3. Weight Sharing - Math

From this point onward, we will change our architecture to introduce weight sharing.

In particular, the input $\bf{x}$ consists of three one-hot vectors concatenated

together. We can think of $\bf{h}$ as a representation of those three words

(all together). However, $\bf{W^{(1)}}$ needs to learn about the first word

separately from the second and third word, when some of the information could

be shared. Consider the following architecture:

<img src="https://www.cs.toronto.edu/~lczhang/321/hw/p2\_model2.png" />

Here, we add an extra \*embedding\* layer to the neural network, where we compute

the representation of \*\*each\*\* word before concatenating them together! We use

the same weight $\bf{W}^{(word)}$ for each of the three words:

\begin{align\*}

\bf{x\_a} &= \textrm{the one-hot vector for word 1} \\

\bf{x\_b} &= \textrm{the one-hot vector for word 2} \\

\bf{x\_c} &= \textrm{the one-hot vector for word 3} \\

\bf{v\_a} &= \bf{W}^{(word)} \bf{x\_a} \\

\bf{v\_b} &= \bf{W}^{(word)} \bf{x\_b} \\

\bf{v\_c} &= \bf{W}^{(word)} \bf{x\_c} \\

\bf{v} &= \textrm{concatenation of } \bf{v\_a}, \bf{v\_b}, \bf{v\_c} \\

\bf{m} &= \bf{W^{(1)}} \bf{v} + \bf{b^{(1)}} \\

\bf{h} &= \textrm{ReLU}(\bf{m}) \\

\bf{z} &= \bf{W^{(2)}} \bf{h} + \bf{b^{(2)}} \\

\bf{y} &= \textrm{softmax}(\bf{z}) \\

\end{align\*}

Note that there are no biases in the embedding layer.

### Part (a) -- 4 pts

Draw a computation graph for $\bf{y}$. Your graph should include

the quantities $\bf{W}^{(word)}$, $\bf{W^{(1)}}$, $\bf{W^{(2)}}$,

$\bf{b^{(1)}}$, $\bf{b^{(2)}}$,

$\bf{x\_a}$,$\bf{x\_b}$, $\bf{x\_c}$,

$\bf{v\_a}$,$\bf{v\_b}$, $\bf{v\_c}$, $\bf{v}$,

$\bf{m}$, $\bf{h}$, $\bf{z}$ and $\bf{y}$.

### Part (b) -- 2 pts

Using the computation graph from part (e), use the chain rule to

write the quantity $\frac{\partial{\bf y}}{\partial{\bf W}^{(word)}}$

in terms of derivatives along the edges in the computation graph

(e.g. $\frac{\partial \bf{y}}{\partial {\bf z}} \frac{\partial \bf{z}}{\partial {\bf \cdot}} \dots$)

You don't need to compute the actual derivatives along the edges

for this question. However, you will need to in Q4(a).

## Question 4. NumPy

In this question, we will implement the model from Question 3

using NumPy. Start by reviewing these helper functions,

which are given to you:

"""

def make\_onehot(indicies, total=250):

"""

Convert indicies into one-hot vectors by

1. Creating an identity matrix of shape [total, total]

2. Indexing the appropriate columns of that identity matrix

"""

I = np.eye(total)

return I[indicies]

def softmax(x):

"""

Compute the softmax of vector x, or row-wise for a matrix x.

We subtract x.max(axis=0) from each row for numerical stability.

"""

x = x.T

exps = np.exp(x - x.max(axis=0))

probs = exps / np.sum(exps, axis=0)

return probs.T

def get\_batch(data, range\_min, range\_max, onehot=True):

"""

Convert one batch of data in the form of 4-grams into input and output

data and return the training data (xs, ts) where:

- `xs` is an numpy array of one-hot vectors of shape [batch\_size, 3, 250]

- `ts` is either

- a numpy array of shape [batch\_size, 250] if onehot is True,

- a numpy array of shape [batch\_size] containing indicies otherwise

Preconditions:

- `data` is a numpy array of shape [N, 4] produced by a call

to `process\_data`

- range\_max > range\_min

"""

xs = data[range\_min:range\_max, :3]

xs = make\_onehot(xs)

ts = data[range\_min:range\_max, 3]

if onehot:

ts = make\_onehot(ts).reshape(-1, 250)

return xs, ts

def estimate\_accuracy(model, data, batch\_size=5000, max\_N=100000):

"""

Estimate the accuracy of the model on the data. To reduce

computation time, use at most `max\_N` elements of `data` to

produce the estimate.

"""

correct = 0

N = 0

for i in range(0, data.shape[0], batch\_size):

xs, ts = get\_batch(data, i, i + batch\_size, onehot=False)

z = model(xs)

pred = np.argmax(z, axis=1)

correct += np.sum(ts == pred)

N += ts.shape[0]

if N > max\_N:

break

return correct / N

"""### Part (a) -- 8 point

Your first task is to implement the model from Question 3 in NumPy.

We will represent the model as a Python class. We set up the

class methods and APIs to be similar to that of PyTorch, so that you

have some intuition about what PyTorch is doing under the hood.

Here's what you need to do:

1. in the `\_\_init\_\_` method, initialize the weights and biases to have the correct shapes. You may want to look back at your answers in the previous question. (0 points)

2. complete the `forward` method to compute the predictions given a \*\*batch\*\* of inputs. This function will also store the intermediate values obtained in the computation; we will need these values for gradient descent. (3 points)

3. complete the `backward` method to compute the gradients of the loss with respect to the weights and biases. (4 points)

4. complete the `update` method that uses the stored gradients to update the weights and biases. (1 point)

"""

class NumpyWordEmbModel(object):

def \_\_init\_\_(self, vocab\_size=250, emb\_size=100, num\_hidden=100):

"""

Initialize the weights and biases to zero. Update this method

so that weights and baises have the correct shape.

"""

self.vocab\_size = vocab\_size

self.emb\_size = emb\_size

self.num\_hidden = num\_hidden

self.emb\_weights = np.zeros((self.emb\_size, self.vocab\_size)) # W^{(word)}

self.weights1 = np.zeros((self.num\_hidden, 3 \* self.emb\_size)) # W^{(1)}

self.bias1 = np.zeros(self.num\_hidden) # b^{(1)}

self.weights2 = np.zeros((self.vocab\_size, self.num\_hidden)) # W^{(2)}

self.bias2 = np.zeros(self.vocab\_size) # b^{(2)}

self.cleanup()

def initializeParams(self):

"""

Randomly initialize the weights and biases of this two-layer MLP.

The randomization is necessary so that each weight is updated to

a different value.

You do not need to change this method.

"""

self.emb\_weights = np.random.normal(0, 2/self.emb\_size, self.emb\_weights.shape)

self.weights1 = np.random.normal(0, 2/self.emb\_size, self.weights1.shape)

self.bias1 = np.random.normal(0, 2/self.emb\_size, self.bias1.shape)

self.weights2 = np.random.normal(0, 2/self.num\_hidden, self.weights2.shape)

self.bias2 = np.random.normal(0, 2/self.num\_hidden, self.bias2.shape)

def forward(self, inputs):

"""

Compute the forward pass prediction for inputs.

Note that for vectorization, `inputs` will be a rank-3 numpy array

with shape [N, 3, vocab\_size], where N is the batch size.

The returned value will contain the predictions for the N

data points in the batch, so the return value shape should be

[N, something].

You should refer to the mathematical expressions we provided in Q3

when completing this method. However, because we are computing

forward pass for a batch of data at a time, you may need to rearrange

some computation (e.g. some matrix-vector multiplication will become

matrix-matrix multiplications, and you'll need to be careful about

arranging the dimensions of your matrices.)

For numerical stability reasons, we will return the \*\*logit z\*\*

instead of the \*\*probability y\*\*. The loss function assumes that

we return the logits from this function.

After writing this function, you might want to check that your code

runs before continuing, e.g. try

xs, ts = get\_batch(train4grams, 0, 8, onehot=True)

m = NumpyWordEmbModel()

m.forward(xs)

"""

self.N = inputs.shape[0]

ipt = np.split(inputs,3,axis=1)

self.xa = ipt[0].reshape((self.N,inputs.shape[2])) # todo

self.xb = ipt[1].reshape((self.N,inputs.shape[2])) # todo

self.xc = ipt[2].reshape((self.N,inputs.shape[2])) # todo

self.va = self.xa @ (self.emb\_weights).T # todo

self.vb = self.xb @ (self.emb\_weights).T # todo

self.vc = self.xc @ (self.emb\_weights).T # todo

self.v = np.concatenate([self.va,self.vb,self.vc],axis=1) # todo

self.m = self.v @ (self.weights1).T + self.bias1 # todo

self.h = np.maximum(self.m, 0) # todo

self.z = self.h @ (self.weights2).T + self.bias2 # todo

self.y = softmax(self.z)

return self.z

def \_\_call\_\_(self, inputs):

"""

This function is here so that if you call the object like a function,

the `backward` method will get called. For example, if we have

m = NumpyWordEmbModel()

Calling `m(foo)` is equivalent to calling `m.forward(foo)`.

You do not need to change this method.

"""

return self.forward(inputs)

def backward(self, ts):

"""

Compute the backward pass, given the ground-truth, one-hot targets.

Note that `ts` needs to be a numpy array with shape [N, vocab\_size].

Complete this method. You might want to refer to your answers to Q2

and Q3. But be careful: we are computing the backward pass for an

entire batch of data at a time! Carefully track the dimensions of your

quantities!

You may assume that the forward() method has already been called, so

you can access values like self.N, self.y, etc..

This function needs to be called before calling the update() method.

"""

z\_bar = (self.y - ts) / self.N

self.w2\_bar = z\_bar.T @ self.h # todo, compute gradient for W^{(2)}

self.b2\_bar = z\_bar.T @ np.ones(self.N) # todo, compute gradient for b^{(2)}

h\_bar = z\_bar @ self.weights2 # todo

m\_bar = (self.m > 0) \* h\_bar # todo

self.w1\_bar = m\_bar.T @ self.v # todo

self.b1\_bar = m\_bar.T @ np.ones(self.N) # todo

# ...

v\_bar = m\_bar @ self.weights1

v\_bar0 = (v\_bar[:,:self.num\_hidden]).T @ self.xa

v\_bar1 = (v\_bar[:,self.num\_hidden:self.num\_hidden\*2]).T @ self.xb

v\_bar2 = (v\_bar[:,self.num\_hidden\*2:]).T @ self.xc

self.emb\_bar = v\_bar0 + v\_bar1 + v\_bar2# todo, compute gradient for W^{(word)}

def update(self, alpha):

"""

Compute the gradient descent update for the parameters.

Complete this method. Use `alpha` as the learning rate.

You can assume that the forward() and backward() methods have already

been called, so you can access values like self.w1\_bar.

"""

self.weights1 = self.weights1 - alpha \* self.w1\_bar

# todo... update the other weights/biases

self.bias1 = self.bias1 - alpha \* self.b1\_bar

self.weights2 = self.weights2 - alpha \* self.w2\_bar

self.bias2 = self.bias2 - alpha \* self.b2\_bar

self.emb\_weights = self.emb\_weights - alpha \* self.emb\_bar

def cleanup(self):

"""

Erase the values of the variables that we use in our computation.

You do not need to change this method.

"""

self.N = None

self.xa = None

self.xb = None

self.xc = None

self.va = None

self.vb = None

self.vc = None

self.v = None

self.m = None

self.h = None

self.z = None

self.y = None

self.z\_bar = None

self.w2\_bar = None

self.b2\_bar = None

self.w1\_bar = None

self.b1\_bar = None

self.emb\_bar = None

"""### Part (b) -- 2 points

Complete the `run\_gradient\_descent` function. Train your numpy model

to obtain a training accuracy of at least 25%. You do not need to train

this model to convergence, but you do need to clearly show

that your model reached at least 25% training accuracy.

"""

def run\_gradient\_descent(model,

train\_data=train4grams,

validation\_data=valid4grams,

batch\_size=100,

learning\_rate=0.1,

max\_iters=5000):

"""

Use gradient descent to train the numpy model on the dataset train4grams.

"""

n = 0

while n < max\_iters:

# shuffle the training data, and break early if we don't have

# enough data to remaining in the batch

np.random.shuffle(train\_data)

for i in range(0, train\_data.shape[0], batch\_size):

if (i + batch\_size) > train\_data.shape[0]:

break

# get the input and targets of a minibatch

xs, ts = get\_batch(train\_data, i, i + batch\_size, onehot=True)

# erase any accumulated gradients

model.cleanup()

# forward pass: compute prediction

# TODO: add your code here

y = softmax(model.forward(xs))

# backward pass: compute error

model.backward(ts)

model.update(learning\_rate)

# increment the iteration count

n += 1

# compute and plot the \*validation\* loss and accuracy

if (n % 100 == 0):

train\_cost = -np.sum(ts \* np.log(y)) / batch\_size

train\_acc = estimate\_accuracy(model, train\_data)

val\_acc = estimate\_accuracy(model, validation\_data)

model.cleanup()

print("Iter %d. [Val Acc %.0f%%] [Train Acc %.0f%%, Loss %f]" % (

n, val\_acc \* 100, train\_acc \* 100, train\_cost))

if n >= max\_iters:

return

numpy\_model= NumpyWordEmbModel()

numpy\_model.initializeParams()

run\_gradient\_descent(numpy\_model)

"""### Part (c) -- 2 pts

If we do not call `numpy\_model.initializeParams()`, your model weights will

not change. Clearly explain (mathematically) why this is the case.

### Part (d) -- 2 pts

The `estimate\_accuracy` function takes the continuous predictions `z`

and turns it into a discrete prediction `pred`. Show that for a given

data point, `pred` is equal to 1 only if the predictive probability `y`

is at least 0.5.

## Question 5. PyTorch

Now, we will build the same model in PyTorch.

### Part (a) -- 2 pts

Since PyTorch uses

automatic differentiation, we only need to write the \*forward pass\* of our

model. Complete the `\_\_init\_\_` and `forward` methods below.

Hint: You might want to look up the `reshape` method in PyTorch.

"""

class PyTorchWordEmb(nn.Module):

def \_\_init\_\_(self, emb\_size=100, num\_hidden=300, vocab\_size=250):

super(PyTorchWordEmb, self).\_\_init\_\_()

self.word\_emb\_layer = nn.Linear(vocab\_size, # num input W^(word)

emb\_size, # num output W^(word)

bias=False)

self.fc\_layer1 = nn.Linear(3 \* emb\_size, # num input W^(1)

num\_hidden) # num output W^(1)

self.fc\_layer2 = nn.Linear(num\_hidden, # num input W^(2)

vocab\_size) # num output W^(2)

self.num\_hidden = num\_hidden

self.emb\_size = emb\_size

def forward(self, inp):

vs = self.word\_emb\_layer(inp)

v = vs.reshape((-1,3\*self.emb\_size)) # TODO: what do you need to do here?

m = self.fc\_layer1(v)

h = torch.relu(m)

z = self.fc\_layer2(h) # TODO: what do you need to do here?

return z

"""### Part (b) -- 2 pts

The function `run\_pytorch\_gradient\_descent` is given to you. It is similar

to the code that you wrote fro the PyTorch model, with a few differences:

1. We will use a slightly fancier optimizer called \*\*Adam\*\*. For this optimizer,

a smaller learning rate usually works better, so the default learning

rate is set to 0.001.

2. Since we get weight decay for free, you are welcome to use weight decay.

Use this function and train your PyTorch model to obtain a training

accuracy of at least 37%. Plot the learning curve using the `plot\_learning\_curve`

function provided to you, and include your plot in your PDF submission.

"""

def estimate\_accuracy\_torch(model, data, batch\_size=5000, max\_N=100000):

"""

Estimate the accuracy of the model on the data. To reduce

computation time, use at most `max\_N` elements of `data` to

produce the estimate.

"""

correct = 0

N = 0

for i in range(0, data.shape[0], batch\_size):

# get a batch of data

xs, ts = get\_batch(data, i, i + batch\_size, onehot=False)

# forward pass prediction

z = model(torch.Tensor(xs))

z = z.detach().numpy() # convert the PyTorch tensor => numpy array

pred = np.argmax(z, axis=1)

correct += np.sum(pred == ts)

N += ts.shape[0]

if N > max\_N:

break

return correct / N

def run\_pytorch\_gradient\_descent(model,

train\_data=train4grams,

validation\_data=valid4grams,

batch\_size=100,

learning\_rate=0.001,

weight\_decay=0,

max\_iters=1000,

checkpoint\_path=None):

"""

Train the PyTorch model on the dataset `train\_data`, reporting

the validation accuracy on `validation\_data`, for `max\_iters`

iteration.

If you want to \*\*checkpoint\*\* your model weights (i.e. save the

model weights to Google Drive), then the parameter

`checkpoint\_path` should be a string path with `{}` to be replaced

by the iteration count:

For example, calling

>>> run\_pytorch\_gradient\_descent(model, ...,

checkpoint\_path = '/content/gdrive/My Drive/CSC413/mlp/ckpt-{}.pk')

will save the model parameters in Google Drive every 500 iterations.

You will have to make sure that the path exists (i.e. you'll need to create

the folder CSC413, mlp, etc...). Your Google Drive will be populated with files:

- /content/gdrive/My Drive/CSC413/mlp/ckpt-500.pk

- /content/gdrive/My Drive/CSC413/mlp/ckpt-1000.pk

- ...

To load the weights at a later time, you can run:

>>> model.load\_state\_dict(torch.load('/content/gdrive/My Drive/CSC413/mlp/ckpt-500.pk'))

This function returns the training loss, and the training/validation accuracy,

which we can use to plot the learning curve.

"""

criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(model.parameters(),

lr=learning\_rate,

weight\_decay=weight\_decay)

iters, losses = [], []

iters\_sub, train\_accs, val\_accs = [], [] ,[]

n = 0 # the number of iterations

while True:

for i in range(0, train\_data.shape[0], batch\_size):

if (i + batch\_size) > train\_data.shape[0]:

break

# get the input and targets of a minibatch

xs, ts = get\_batch(train\_data, i, i + batch\_size, onehot=False)

# convert from numpy arrays to PyTorch tensors

xs = torch.Tensor(xs)

ts = torch.Tensor(ts).long()

zs = model(xs)

loss = criterion(zs, ts) # compute the total loss

loss.backward() # compute updates for each parameter

optimizer.step() # make the updates for each parameter

optimizer.zero\_grad() # a clean up step for PyTorch

# save the current training information

iters.append(n)

losses.append(float(loss)/batch\_size) # compute \*average\* loss

if n % 500 == 0:

iters\_sub.append(n)

train\_cost = float(loss.detach().numpy())

train\_acc = estimate\_accuracy\_torch(model, train\_data)

train\_accs.append(train\_acc)

val\_acc = estimate\_accuracy\_torch(model, validation\_data)

val\_accs.append(val\_acc)

print("Iter %d. [Val Acc %.0f%%] [Train Acc %.0f%%, Loss %f]" % (

n, val\_acc \* 100, train\_acc \* 100, train\_cost))

if (checkpoint\_path is not None) and n > 0:

torch.save(model.state\_dict(), checkpoint\_path.format(n))

# increment the iteration number

n += 1

if n > max\_iters:

return iters, losses, iters\_sub, train\_accs, val\_accs

def plot\_learning\_curve(iters, losses, iters\_sub, train\_accs, val\_accs):

"""

Plot the learning curve.

"""

plt.title("Learning Curve: Loss per Iteration")

plt.plot(iters, losses, label="Train")

plt.xlabel("Iterations")

plt.ylabel("Loss")

plt.show()

plt.title("Learning Curve: Accuracy per Iteration")

plt.plot(iters\_sub, train\_accs, label="Train")

plt.plot(iters\_sub, val\_accs, label="Validation")

plt.xlabel("Iterations")

plt.ylabel("Accuracy")

plt.legend(loc='best')

plt.show()

pytorch\_model = PyTorchWordEmb()

learning\_curve\_info = run\_pytorch\_gradient\_descent(pytorch\_model,max\_iters=8000,checkpoint\_path= '/content/gdrive/My Drive/CSC413/A/A1/Q5/ckpt-{}.pk')

# you might want to save the `learning\_curve\_info` somewhere, so that you can plot

# the learning curve prior to exporting your PDF file

plot\_learning\_curve(\*learning\_curve\_info)

"""### Part (c) -- 3 points

Write a function `make\_prediction` that takes as parameters

a PyTorchWordEmb model and sentence (a list of words), and produces

a prediction for the next word in the sentence.

Start by thinking about what you need to do, step by step, taking

care of the difference between a numpy array and a PyTorch Tensor.

"""

def make\_prediction\_torch(model, sentence):

"""

Use the model to make a prediction for the next word in the

sentence using the last 3 words (sentence[:-3]). You may assume

that len(sentence) >= 3 and that `model` is an instance of

PyTorchWordEmb. You might find the function torch.argmax helpful.

This function should return the next word, represented as a string.

Example call:

>>> make\_prediction\_torch(pytorch\_model, ['you', 'are', 'a'])

"""

global vocab\_stoi, vocab\_itos

# Write your code here

sentence = [word.lower() for word in sentence]

last = [sentence[-3:]]

index = convert\_words\_to\_indices(last)

one\_hot = make\_onehot(index)

pred = torch.Tensor(one\_hot)

y = model.forward(pred)

word\_idx = torch.argmax(y)

return vocab\_itos[word\_idx.item()]

"""### Part (d) -- 4 points

Use your code to predict what the next word should be in each

of the following sentences:

- "You are a"

- "few companies show"

- "There are no"

- "yesterday i was"

- "the game had"

- "yesterday the federal"

Do your predictions make sense? (If all of your predictions are the same,

train your model for more iterations, or change the hyper parameters in your

model. You may need to do this even if your training accuracy is >=37%)

One concern you might have is that our model may be "memorizing" information

from the training set. Check if each of 3-grams (the 3 words appearing next

to each other) appear in the training set. If so, what word occurs immediately

following those three words?

"""

# Write your code and answers here

print(make\_prediction\_torch(pytorch\_model, ['You', 'are', 'a']))

print(make\_prediction\_torch(pytorch\_model, ['few', 'companies', 'show']))

print(make\_prediction\_torch(pytorch\_model, ['There', 'are', 'no']))

print(make\_prediction\_torch(pytorch\_model, ['yesterday', 'i', 'was']))

print(make\_prediction\_torch(pytorch\_model, ['the', 'game', 'had']))

print(make\_prediction\_torch(pytorch\_model, ['yesterday', 'the', 'federal']))

"""### Part (3) -- 1 points

Report the test accuracy of your model. The test accuracy is the percentage

of correct predictions across your test set.

"""

# Write your code here

print("The test accuracy is", estimate\_accuracy\_torch(pytorch\_model,test4grams))

"""## Question 6. Visualizing Word Embeddings

While training the `PyTorchWordEmb`, we trained the `word\_emb\_layer`, which takes a one-hot

representation of a word in our vocabulary, and returns a low-dimensional vector

representation of that word. In this question, we will explore these word embeddings.

### Part (a) -- 1 pts

The code below extracts the \*\*weights\*\* of the word embedding layer,

and converts the PyTorch tensor into an numpy array.

Explain why each \*row\* of `word\_emb` contains the vector representing

of a word. For example `word\_emb[vocab\_stoi["any"],:]` contains the

vector representation of the word "any".

"""

word\_emb\_weights = list(pytorch\_model.word\_emb\_layer.parameters())[0]

word\_emb = word\_emb\_weights.detach().numpy().T

# Write your explanation here

"""### Part (b) -- 1 pts

Once interesting thing about these word embeddings is that distances

in these vector representations of words make some sense! To show this,

we have provided code below that computes the cosine similarity of

every pair of words in our vocabulary.

"""

norms = np.linalg.norm(word\_emb, axis=1)

word\_emb\_norm = (word\_emb.T / norms).T

similarities = np.matmul(word\_emb\_norm, word\_emb\_norm.T)

# Some example distances. The first one should be larger than the second

print(similarities[vocab\_stoi['any'], vocab\_stoi['many']])

print(similarities[vocab\_stoi['any'], vocab\_stoi['government']])

"""Compute the 5 closest words to the following words:

- "four"

- "go"

- "what"

- "should"

- "school"

- "your"

- "yesterday"

- "not"

"""

# Write your code here

def closest\_words(w):

similar = similarities[vocab\_stoi[w]]

index = (np.argsort(similar))[::-1]

print("5 closest words for", w,":")

for i in range(5):

print(vocab\_itos[index[i]])

return

following\_words = ["four", "go","what","should","school","your","yesterday","not"]

for i in following\_words:

closest\_words(i)

"""### Part (c) -- 2 pts

We can visualize the word embeddings by reducing the dimensionality of

the word vectors to 2D. There are many dimensionality reduction techniques

that we could use, and we will use an algorithm called t-SNE.

(You don’t need to know what this is for the assignment,

but we may cover it later in the course.)

Nearby points in this 2-D space are meant to correspond to nearby points

in the original, high-dimensional space.

The following code runs the t-SNE algorithm and plots the result.

Look at the plot and find two clusters of related words.

What do the words in each cluster have in common?

Note that there is randomness in the initialization of the t-SNE

algorithm. If you re-run this code, you may get a different image.

Please make sure to submit your image in the PDF file for your TA to see.

"""

import sklearn.manifold

tsne = sklearn.manifold.TSNE()

Y = tsne.fit\_transform(word\_emb)

plt.figure(figsize=(10, 10))

plt.xlim(Y[:,0].min(), Y[:, 0].max())

plt.ylim(Y[:,1].min(), Y[:, 1].max())

for i, w in enumerate(vocab):

plt.text(Y[i, 0], Y[i, 1], w)

plt.show()

"""## Question 7. Work Allocation -- 2 pts

This question is to make sure that if you are working with a partner, that

you and your partner contributed equally to the assignment.

Please have each team member write down the times that you worked on the

assignment, and your contribution to the assignment.

"""

# Example answer:

# I worked on the assignment on Jan 20 afternoon, Jan 26th 12pm-2pm,

# and then Feb 4th in the evening. My partner and I had a meeting on

# Jan 20th to read the entire assignment, and we did Question 1 together

# while screensharing. I worked out the math for Q2, and checked my

# partner's implementation in Q3. I also wrote the Q3 helper functions,

# and Q4(b).

"""This assignment is finished individually by Hongyu Chen"""