p2

February 15, 2020

1 CSC321H5 Project 2.

Deadline: Thursday, Feb. 13, by 9pm

Submission: Submit a PDF export of the completed notebook.

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

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'll explore a couple of different models to perform this prediction task. We will also do this problem twice: once in PyTorch, and once using numpy. When using numpy, you'll implement the backpropagation computation.

In doing this prediction task, our neural networks will learn about *words* and about how to represent words. We'll explore the *vector representations* of words that our model produces, and analyze these representations.

You may modify the starter code as you see fit, including changing the signatures of functions and adding/removing helper functions. However, please make sure that your TA can understand what you are doing and why.

[0]:
```
import pandas
import numpy as np
import matplotlib.pyplot as plt

import torch
import torch.nn as nn
import torch.optim as optim

import collections
```

1.1 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 `https://www.cs.toronto.edu/~lczhang/321/hw/raw_sentences.txt` and upload it to Google Drive. Then, mount Google Drive from your Google Colab notebook:

[33]:
```
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Drive already mounted at /content/gdrive; to attempt to forcibly remount, call
drive.mount("/content/gdrive", force_remount=True).
```

Find the path to `raw_sentences.txt`:

[0]:
```
file_path = '/content/gdrive/My Drive/CSC321/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:

[35]:
```
!ls /content/gdrive/My\ Drive/
#!mkdir /content/gdrive/My\ Drive/CSC321
```

```
 A1.pdf                                Notability
'Clearbanc Data Science Intern.mp4'    Nox
'Colab Notebooks'                     'Past CourseWork'
'Contact Information.gform'            STA304.gdoc
 CSC321                               'STA304 Group Project.gdoc'
'CSC343 Summary.gdoc'                 'Untitled form.gform'
```

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`.

[0]:
```
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.

[37]:
```
vocab = set([w for s in sentences for w in s])
print(len(sentences)) # 97162
print(len(vocab)) # 250
```

```
97162
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.

[0]:
```
# Training set -> Train the model
# Validation set -> Adjust hyperparameters accordingly to not overfit our data
# Test set -> unseen data by the model which we run our model on
test, valid, train = sentences[:10000], sentences[10000:20000], sentences[20000:
↪]
```

1.1.1 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.

[39]:
```
# Your code goes here
# print(train[:10]) # This gets cut off in pDF

# Or pretty print
for row in train[0:10]:
  print(row)

# We can see that the punctuation marks are treated as a seperate word.
# And Words with apostrophes are broken down into two seperate strings:
# 1) Characters before the apostrophe & 2) Apostrophe, and the added letter(s)␣
 ↪in double quotes
```

```
['last', 'night', ',', 'he', 'said', ',', 'did', 'it', 'for', 'me', '.']
['on', 'what', 'can', 'i', 'do', '?']
['now', 'where', 'does', 'it', 'go', '?']
['what', 'did', 'the', 'court', 'do', '?']
['but', 'at', 'the', 'same', 'time', ',', 'we', 'have', 'a', 'long', 'way',
'to', 'go', '.']
['that', 'was', 'the', 'only', 'way', '.']
['this', 'team', 'will', 'be', 'back', '.']
['so', 'that', 'is', 'what', 'i', 'do', '.']
['we', 'have', 'a', 'right', 'to', 'know', '.']
['now', 'they', 'are', 'three', '.']
```

1.1.2 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 occurences of the word / total number of words in the training set).

These are good quantities to compute, because one of the first things a 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 can use Python's `collections.Counter` class if you would like to.

[40]:
```
# Your code goes here
""" Make list of words to feed counter """
wl=[]
for sl in sentences:
    for word in sl:
        wl.append(word)
```

```
# Do we want to include punctuation here?
cnt = collections.Counter()
# Populate counter object
for word in wl:
  cnt[word] += 1
# Return top 10 words
k=cnt.most_common(10)
total=sum(cnt.values())
for i in k:
  print("Word: "+i[0]+", Frequency: "+str(i[1])+", Percentage␣
 ↪"+str(round(i[1]*100/total,3))+"%")


# Print the number of occureces for each one
```

```
Word: ., Frequency: 80974, Percentage 10.694%
Word: it, Frequency: 29200, Percentage 3.856%
Word: ,, Frequency: 24583, Percentage 3.247%
Word: i, Frequency: 22267, Percentage 2.941%
Word: do, Frequency: 20245, Percentage 2.674%
Word: to, Frequency: 19537, Percentage 2.58%
Word: nt, Frequency: 16460, Percentage 2.174%
Word: ?, Frequency: 16210, Percentage 2.141%
Word: the, Frequency: 15939, Percentage 2.105%
Word: that, Frequency: 15795, Percentage 2.086%
```

1.1.3 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 constances `vocab`, `vocab_itos`, and `vocab_stoi` in your code.

[0]:
```
# 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 index => word (string)
vocab_itos = dict(enumerate(vocab))
# A mapping of word => its index
vocab_stoi = {word:index for index, 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 index 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
    # Create container list
    list_list = []
    # Iter through list of sublists
    for word_list in sents:

      index_list = []
      # Iter through words in sublist
      for word in word_list:
          # Append word indeces to index_list
          index_list.append(vocab_stoi[word])
        # Append index_list to list_list
        # Works, checked against example case
      list_list.append(index_list)
    return list_list


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
    # Tested
    return_list = []
    # Iter through seqs - list of the list
```

```
    for seq in seqs:
      if len(seq)==4:
        return_list.append(seq)
      elif len(seq)>4:
        k=len(seq)
        i=0
        while i<(k-3):
          return_list.append(seq[i:i+4])
          i+=1
    return return_list


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)
#convert_words_to_indices([['one', 'in', 'five', 'are', 'over', 'here'],␣
 ↪['other', 'one', 'since', 'yesterday'], ['you']])
#generate_4grams([[148, 98, 70, 23, 154, 89], [151, 148, 181, 246], [248]])
#generate_4grams([[1, 1, 1, 1, 1]])
```

1.2 Question 2. A Multi-Layer Perceptron

In this section, we will build a two-layer multi-layer perceptron. We will first do this in numpy, and then once more in PyTorch. Our model will look like this:

Start by reviewing these helper functions, which are given to you:

[0]:
```
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)
        y = model(xs)
        pred = np.argmax(y, axis=1)
        correct += np.sum(ts == pred)
        N += ts.shape[0]

        if N > max_N:
            break
    return correct / N
```

1.2.1 Part (a) – 2 point

Your first task is to implement MLP model in Numpy. This model is very similar to the one we built in Tutorial 5. However, we will write our code differently from Tutorial 5, so that the class methods and APIs are similar to that of PyTorch. This is to give you some intuition about what PyTorch is doing under the hood.

We already wrote code for the backward pass for this model in Tutorial 5, so the code is given to you. To make sure you understand how the model works, **write the code to compute the forward pass**.

[0]:
```
class NumpyMLPModel(object):
    def __init__(self, num_features=250*3, num_hidden=400, num_classes=250):
        """
        Initialize the weights and biases of this two-layer MLP.
        """
        self.num_features = num_features
        self.num_hidden = num_hidden
        self.num_classes = num_classes
        self.weights1 = np.zeros([num_hidden, num_features])
        self.bias1 = np.zeros([num_hidden])
        self.weights2 = np.zeros([num_classes, num_hidden])
        self.bias2 = np.zeros([num_classes])
        self.cleanup()

    def initializeParams(self):
        """
        Initialize the weights and biases of this two-layer MLP to be random.
        This random initialization is necessary to break the symmetry in the
        gradient descent update for our hidden weights and biases. If all our
        weights were initialized to the same value, then their gradients will
        all be the same!
        """
        self.weights1 = np.random.normal(0, 2/self.num_features, self.weights1.
↪shape)
        self.bias1 = np.random.normal(0, 2/self.num_features, 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 `inputs` will be a rank-3 numpy array with shape [N, 3, 250],
        so we will need to flatten the tensor to [N, 750] first.

        For the ReLU activation, you may find the function `np.maximum` helpful
        """
```

```
        X = inputs.reshape([-1, 750])

        # TODO:

        self.N = X.shape[0]
        self.X = X
        #self.z1 =  None
        #self.z1=make_onehot(self.X)
        self.z1=np.dot(self.X,self.weights1.T) + self.bias1
        #self.h = None
        self.h=np.maximum(self.z1,0)
        #self.z2 = None
        self.z2=np.dot(self.h,self.weights2.T) + self.bias2
        #self.y = None
        self.y=softmax(self.z2)
        return self.y

    def __call__(self, inputs):
        """
        To be compatible with PyTorch API. With this code, the following two
        calls are identical:

        >>> m = TwoLayerMLP()
        >>> m.forward(inputs)

        and

        >>> m = TwoLayerMLP()
        >>> m(inputs)
        """
        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 rank 2 numpy array with shape [N, 250].
        """
        self.z2_bar = (self.y - ts) / self.N
        self.w2_bar = np.dot(self.z2_bar.T, self.h)
        self.b2_bar = np.dot(self.z2_bar.T, np.ones(self.N))
        self.h_bar = np.matmul(self.z2_bar, self.weights2)
        self.z1_bar = self.h_bar * (self.z1 > 0)
        self.w1_bar = np.dot(self.z1_bar.T, self.X)
        self.b1_bar = np.dot(self.z1_bar.T, np.ones(self.N))

    def update(self, alpha):
        """
```

```
        Compute the gradient descent update for the parameters.
        """
        self.weights1 = self.weights1 - alpha * self.w1_bar
        self.bias1    = self.bias1    - alpha * self.b1_bar
        self.weights2 = self.weights2 - alpha * self.w2_bar
        self.bias2    = self.bias2    - alpha * self.b2_bar

    def cleanup(self):
        """
        Erase the values of the variables that we use in our computation.
        """
        self.N = None
        self.X = None
        self.z1 = None
        self.h = None
        self.z2 = None
        self.y = None
        self.z2_bar = None
        self.w2_bar = None
        self.b2_bar = None
        self.h_bar = None
        self.z1_bar = None
        self.w1_bar = None
        self.b1_bar = None
```

1.2.2 Part (b) – 2 points

Complete the `run_gradient_descent` function. Train your numpy MLP model to obtain a training accuracy of at least 25%. You do not need to train this model to convergence.

[44]:
```
def run_gradient_descent(model,
                         train_data=train4grams,
                         validation_data=valid4grams,
                         batch_size=100,
                         learning_rate=0.3,
                         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)

            # forward pass: compute prediction

            # TODO: add your code here
            y=model.forward(xs)

            # backward pass: compute error

            # TODO: add your code here
            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_mlp = NumpyMLPModel()
numpy_mlp.initializeParams()
# run_gradient_descent(...)
# To reduce wait time, we change the learning rate and max iterations to␣
 ↪illustrate we have reached 25% + accuracy quickly.
run_gradient_descent(numpy_mlp,learning_rate=1,max_iters=700)
```

```
Iter 100. [Val Acc 17%] [Train Acc 17%, Loss 4.304126]
Iter 200. [Val Acc 20%] [Train Acc 20%, Loss 4.468402]
Iter 300. [Val Acc 22%] [Train Acc 23%, Loss 3.930970]
Iter 400. [Val Acc 24%] [Train Acc 25%, Loss 3.806297]
Iter 500. [Val Acc 25%] [Train Acc 26%, Loss 3.385852]
Iter 600. [Val Acc 26%] [Train Acc 26%, Loss 3.515004]
Iter 700. [Val Acc 27%] [Train Acc 27%, Loss 3.435304]
```

1.2.3 Part (c) – 2 pts

We will do build the same model in PyTorch. Since PyTorch uses automatic differentiation, we only need to write the *forward pass* of our model. Complete the `forward` function below:

[0]:
```
class PyTorchMLP(nn.Module):
    def __init__(self, num_hidden=400):
        super(PyTorchMLP, self).__init__()
        self.layer1 = nn.Linear(750, num_hidden)
        self.layer2 = nn.Linear(num_hidden, 250)
        self.num_hidden = num_hidden
    def forward(self, inp):
        inp = inp.reshape([-1, 750])
        # TODO: complete this function
        ##############
        self.N = inp.shape[0]
        self.X = inp
        #self.z1 =  None
        #self.z1=make_onehot(self.X)
        self.z1=self.layer1(self.X)
        #self.h = None
        self.h=torch.relu(self.z1)
        #self.z2 = None
        self.z2=self.layer2(self.h)
        #self.y = None
        self.y=self.z2
        return self.y
```

1.2.4 Part (d) – 4 pts

We'll write similar code to train 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.

Complete the function `run_pytorch_gradient_descent`, and use it to train your PyTorch MLP model to obtain a training accuracy of at least 38%. Plot the learning curve using the `plot_learning_curve` function provided to you, and include your plot in your PDF submission.

[0]:
```
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
        y = model(torch.Tensor(xs))
        y = y.detach().numpy() # convert the PyTorch tensor => numpy array
        pred = np.argmax(y, 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/CSC321/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 CSC321, mlp, etc...). Your Google Drive will be populated with␣
 ↪files:

    - /content/gdrive/My Drive/CSC321/mlp/ckpt-500.pk
    - /content/gdrive/My Drive/CSC321/mlp/ckpt-1000.pk
    - ...
```

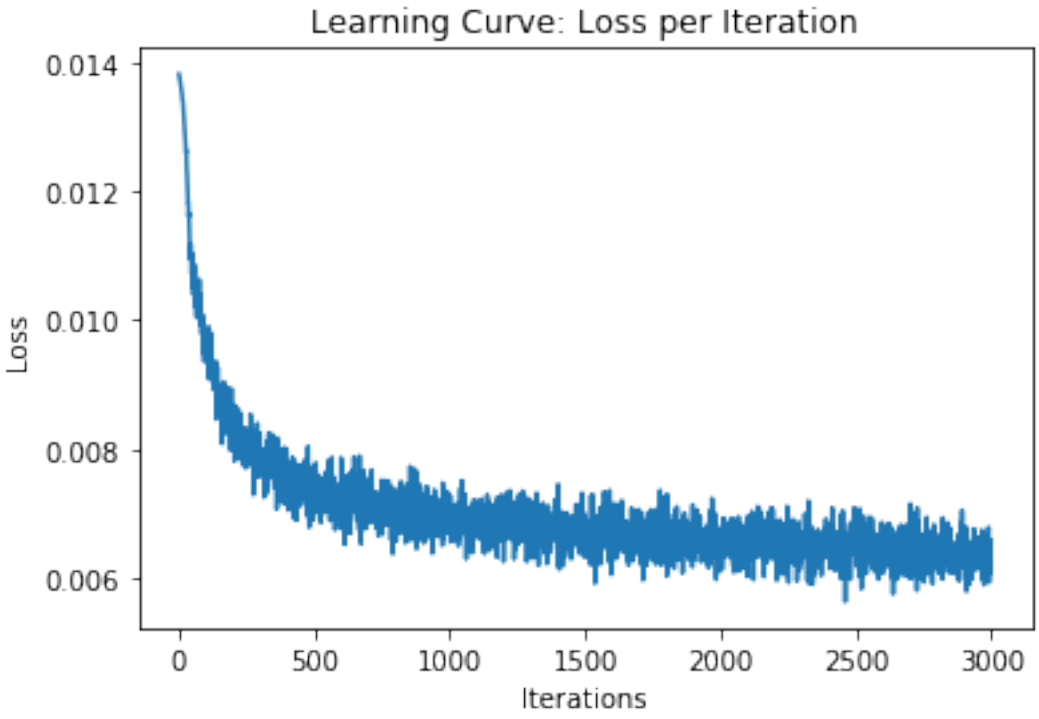
```
    To load the weights at a later time, you can run:

    >>> model.load_state_dict(torch.load('/content/gdrive/My Drive/CSC321/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 =                  # compute prediction logit
            # loss =                    # compute the total loss
            # ...                       # compute updates for each parameter
            # ...                       # make the updates for each parameter
            # ...                       # a clean up step for PyTorch
            ####################
            zs=model.forward(xs)

            loss=criterion(zs,ts)
            loss.backward() # compute updates for each parameter
            optimizer.step() # make the updates for each parameter
            optimizer.zero_grad() # clean up step for PyTorch

            # save the current training information
            iters.append(n)
            losses.append(float(loss)/batch_size)  # compute *average* loss
```

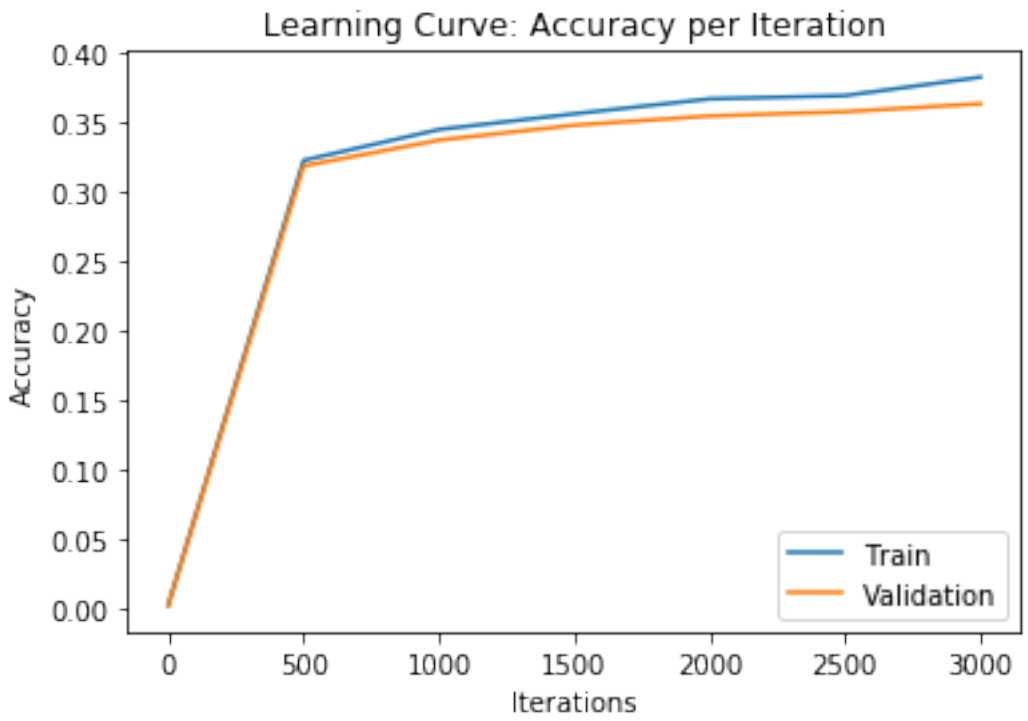
```
            if n % 500 == 0:
                iters_sub.append(n)
                train_cost = float(loss.detach().numpy())
                train_acc = estimate_accuracy_torch(model, train_data)
                train_accs.append(train_acc)
                val_acc = estimate_accuracy_torch(model, validation_data)
                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()
```

[49]:
```
pytorch_mlp = PyTorchMLP()
learning_curve_info = run_pytorch_gradient_descent(pytorch_mlp,max_iters=3000,␣
 ↪batch_size=400)

# you might want to save the `learning_curve_info` somewhere, so that you can␣
 ↪plot
# the learning curve prior to exporting your PDF file
iters=learning_curve_info[0]
losses=learning_curve_info[1]
```

```
iters_sub=learning_curve_info[2]
train_accs=learning_curve_info[3]
val_accs=learning_curve_info[4]
plot_learning_curve(iters, losses, iters_sub, train_accs, val_accs)
```

```
Iter 0. [Val Acc 0%] [Train Acc 0%, Loss 5.529212]
Iter 500. [Val Acc 32%] [Train Acc 32%, Loss 2.824338]
Iter 1000. [Val Acc 34%] [Train Acc 34%, Loss 2.717762]
Iter 1500. [Val Acc 35%] [Train Acc 36%, Loss 2.655774]
Iter 2000. [Val Acc 35%] [Train Acc 37%, Loss 2.648370]
Iter 2500. [Val Acc 36%] [Train Acc 37%, Loss 2.596951]
Iter 3000. [Val Acc 36%] [Train Acc 38%, Loss 2.490390]
```





1.2.5 Part (e) – 3 points

Write a function `make_prediction` that takes as parameters a PyTorchMLP 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.

[0]:
```
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
    PYTorchMLP.

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

    Example call:
    >>> make_prediction_torch(pytorch_mlp, ['you', 'are', 'a'])
    """
    global vocab_stoi, vocab_itos
    sentence=sentence[-3:]
    indeces=[]
```

```
    for i in sentence:
      indeces.append([i])
    indeces=convert_words_to_indices(indeces)
    npmat=make_onehot(indeces[0])
    npmat=np.append(npmat,make_onehot(indeces[1]),axis=0)
    npmat=np.append(npmat,make_onehot(indeces[2]),axis=0)

    x=torch.tensor(npmat)
    y=pytorch_mlp(x.float())
    y=y.data.numpy()

    k=np.argmax(y)

    return vocab_itos[k]
#make_prediction_torch(pytorch_mlp, ['you', 'are', 'a','good'])
    #  Write your code here
```

1.2.6 Part (f) – 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 hyperparameters in your model. You may need to do this even if your training accuracy is >=38%)

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?

[57]:
```
# Write your code and answers here
print("you are a " +make_prediction_torch(pytorch_mlp,['you','are','a'])+'\n')
print("few companies show" +make_prediction_torch(pytorch_mlp,["few",␣
 ↪"companies", "show"])+'\n')
print("there are no "␣
 ↪+make_prediction_torch(pytorch_mlp,['there','are','no'])+'\n')
print("yesterday i was "␣
 ↪+make_prediction_torch(pytorch_mlp,['yesterday','i','was'])+'\n')
print("the game had "␣
 ↪+make_prediction_torch(pytorch_mlp,['the','game','had'])+'\n')
```

```
print("yesterday the federal "␣
 ↪+make_prediction_torch(pytorch_mlp,['yesterday','the','federal'])+'\n')

# These predictions do make sense, they aren't perfect but they are quite good␣
 ↪especialy for such low validation accuraccy
# Check if each of 3-grams (the 3 words
```

```
you are a good

few companies show.

there are no other

yesterday i was nt

the game had to

yesterday the federal government
```

[57]:
```
"These predictions do make sense, they aren't perfect but they are quite good
especialy for such low validation accuraccy"
```

1.2.7 Part (g) – 1 points

Report the test accuracy of your model

[58]:
```
# Write your code here
estimate_accuracy_torch(pytorch_mlp,test4grams)
```

[58]:
```
0.36541070869429076
```

1.3 Question 3. Learning Word Embeddings

In this section, we will build a slightly different model with a different architecture. In particular, we will first compute a lower-dimensional *representation* of the three words, before using a multi-layer perceptron.

Our model will look like this:

This model has 3 layers instead of 2, but the first layer of the network is **not** fully-connected. Instead, we compute the representations of each of the three words **separately**. In addition, the first layer of the network will not use any biases. The reason for this will be clear in question 4.

1.3.1 Part (a) - 10 pts

Complete the methods in `NumpyWordEmbModel`.

[0]:
```
class NumpyWordEmbModel(object):
    def __init__(self, vocab_size=250, emb_size=100, num_hidden=100):
        self.vocab_size = vocab_size
        self.emb_size = emb_size
        self.num_hidden = num_hidden
        self.emb_weights = np.zeros([emb_size, vocab_size]) # no biases in this␣
↪layer
        self.weights1 = np.zeros([num_hidden, emb_size * 3])
        self.bias1 = np.zeros([num_hidden])
        self.weights2 = np.zeros([vocab_size, num_hidden])
        self.bias2 = np.zeros([vocab_size])
        self.cleanup()
        self.num_features=3*emb_size

    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.
        """
        self.emb_weights = np.random.normal(0, 2/self.num_hidden, self.
↪emb_weights.shape)
        self.weights1 = np.random.normal(0, 2/self.num_features, self.weights1.
↪shape)
        self.bias1 = np.random.normal(0, 2/self.num_features, 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 `inputs` will be a rank-3 numpy array with shape [N, 3, 250].

        For numerical stability reasons, we **do not** apply the softmax
        activation in the forward function. The loss function assumes that
        we return the logits from this function.
        """
        # TODO
        self.word1=inputs[: ,0, :]
        self.word2=inputs[:,1,:]
        self.word3=inputs[:,2,:]
        self.N=inputs.shape[0]
```

```
        k=np.matmul(inputs,self.emb_weights.T)

        self.X=k.reshape([-1,300])
        self.X = np.maximum(0, self.X)
        self.z1=np.matmul(self.X,self.weights1.T)+self.bias1.T
        self.h=np.maximum(self.z1,0)
        self.z2=np.add(np.matmul(self.h,self.weights2.T),self.bias2.T)

        self.y=softmax(self.z2)
        return self.y




    def __call__(self, inputs):
        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 rank 2 numpy array with shape [N, 250].

        Remember the multivariate chain rule: if a weight affects the loss
        through different paths, then the error signal from all the paths
        must be added together.
        """
        # TODO

        self.z2b = (self.y - ts) / self.N
        self.w2_bar = np.dot(self.z2b.T, self.h)
        #self.b2_bar = np.dot(self.z2b.T, np.ones(self.N))
        self.b2_bar = np.sum(self.z2b.T)
        self.h_bar = np.matmul(self.z2b, self.weights2)
        self.z1_bar = self.h_bar * (self.z1 > 0)
        self.w1_bar = np.dot(self.z1_bar.T, self.X)

        self.b1_bar=np.sum(self.z1_bar,axis=0)

        self.emb_weights_bar=np.matmul(self.z1_bar.T, self.word1)+np.
↪matmul(self.z1_bar.T, self.word2)+np.matmul(self.z1_bar.T, self.word3)


    def update(self, alpha):
        """
        Compute the gradient descent update for the parameters.
        """
```

```
        # TODO
        self.emb_weights = self.emb_weights-alpha*self.emb_weights_bar
        self.weights1 = self.weights1-alpha*self.w1_bar
        self.bias1 = self.bias1-alpha*self.b1_bar
        self.weights2 = self.weights2-alpha*self.w2_bar
        self.bias2=self.bias2-alpha*self.b2_bar


    def cleanup(self):
        """
        Erase the values of the variables that we use in our computation.
        """
        # TODO
        self.N = None
        self.X = None
        self.z1 = None
        self.h = None
        self.z2 = None
        self.y = None
        self.z2b = None
        self.w2_bar = None
        self.b2_bar = None
        self.h_bar = None
        self.z1_bar = None
        self.w1_bar = None
        self.b1_bar = None
        self.emb_weights_bar = None
        self.word1 = None
        self.word2 = None
        self.word3 = None
```

1.3.2 Part (b) – 1 pts

One strategy that machine learning practitioners use to debug their code is to *first try to overfit their model to a small training set.* If the gradient computation is correct and the data is encoded properly, then your model should easily achieve 100% training accuracy on a small training set.

Show that your model is implemented correctly by showing that your model can achieve an 100% training accuracy within a few hundred iterations, when using a small training set (e.g. one batch).

[122]:
```
"""numpy_wordemb = NumpyWordEmbModel()
#print(str(train4grams[:64].shape))
xs = get_batch(train4grams[:100], 0,  100, onehot=False)
ts=xs[1]
xs=xs[0]
 # convert from numpy arrays to PyTorch tensors
```

```
xs = torch.Tensor(xs)
#ts = torch.Tensor(ts).long()
numpy_wordemb(xs)
numpy_wordemb.backward(ts)
# run_pytorch_gradient_descent(numpy_wordemb, train4grams[:64], batch_size=64, .
 ↪..)"""
numpy_wordemb = NumpyWordEmbModel()
numpy_wordemb.initializeParams()
run_gradient_descent(numpy_wordemb, train4grams[:40], valid4grams, 30,␣
 ↪max_iters=1000)
```

```
Iter 100. [Val Acc 17%] [Train Acc 18%, Loss 3.276141]
Iter 200. [Val Acc 2%] [Train Acc 22%, Loss 3.247838]
Iter 300. [Val Acc 5%] [Train Acc 32%, Loss 2.528780]
Iter 400. [Val Acc 4%] [Train Acc 88%, Loss 0.717060]
Iter 500. [Val Acc 3%] [Train Acc 98%, Loss 0.166000]
Iter 600. [Val Acc 3%] [Train Acc 100%, Loss 0.071170]
Iter 700. [Val Acc 3%] [Train Acc 100%, Loss 0.030284]
Iter 800. [Val Acc 3%] [Train Acc 100%, Loss 0.016734]
Iter 900. [Val Acc 3%] [Train Acc 100%, Loss 0.013969]
Iter 1000. [Val Acc 3%] [Train Acc 100%, Loss 0.010550]
```

1.3.3 Part (c) – 2 pts

Train your model from part (a) to obtain a training accuracy of at least 25%.

[61]:
```
# Your code goes here
run_gradient_descent(numpy_wordemb, train4grams, valid4grams, batch_size=300,␣
 ↪max_iters=1000)
```

```
Iter 100. [Val Acc 20%] [Train Acc 20%, Loss 4.160442]
Iter 200. [Val Acc 21%] [Train Acc 22%, Loss 4.264535]
Iter 300. [Val Acc 23%] [Train Acc 23%, Loss 4.060728]
Iter 400. [Val Acc 24%] [Train Acc 24%, Loss 3.813635]
Iter 500. [Val Acc 24%] [Train Acc 24%, Loss 3.783613]
Iter 600. [Val Acc 25%] [Train Acc 25%, Loss 3.736000]
Iter 700. [Val Acc 25%] [Train Acc 25%, Loss 3.747912]
Iter 800. [Val Acc 26%] [Train Acc 26%, Loss 3.647005]
Iter 900. [Val Acc 26%] [Train Acc 26%, Loss 3.557349]
Iter 1000. [Val Acc 26%] [Train Acc 26%, Loss 3.597314]
```

1.3.4 Part (d) – 2 pts

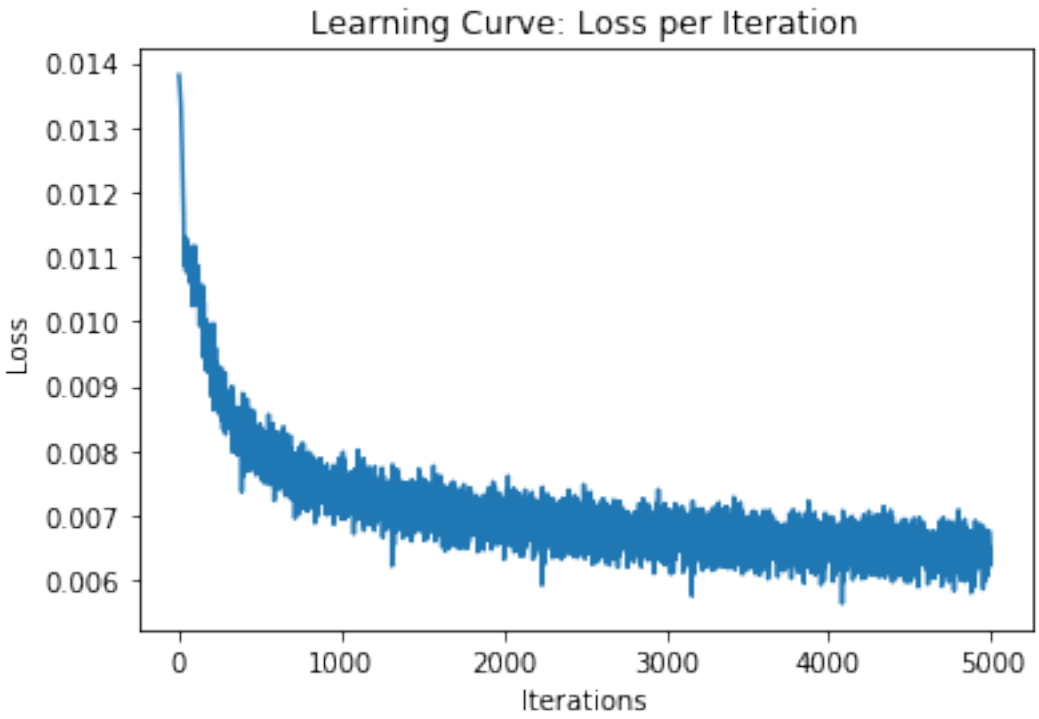
The PyTorch version of the model is implemented for you. Use `run_pytorch_gradient_descent` to train your PyTorch MLP model to obtain a training accuracy of at least 38%. Plot the learning

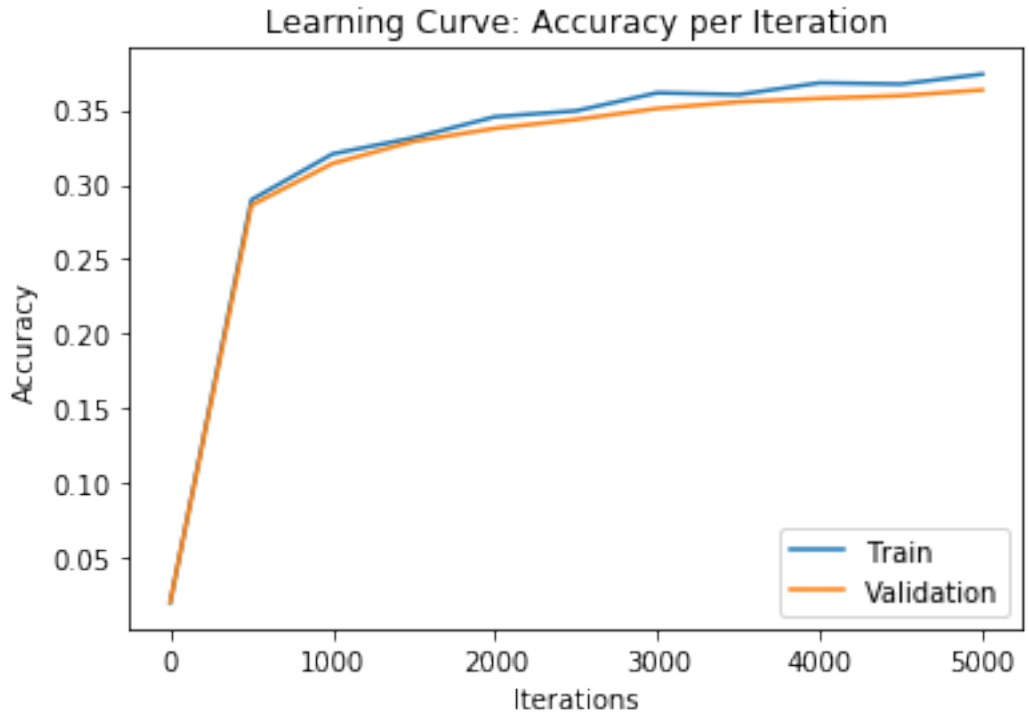
curve using the `plot_learning_curve` function provided to you, and include your plot in your PDF submission.

Make sure that you checkpoint frequently. We will be using …

[62]:
```
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, emb_size, bias=False)
        self.fc_layer1 = nn.Linear(emb_size * 3, num_hidden)
        self.fc_layer2 = nn.Linear(num_hidden, 250)
        self.num_hidden = num_hidden
        self.emb_size = emb_size
    def forward(self, inp):
        embeddings = torch.relu(self.word_emb_layer(inp))
        embeddings = embeddings.reshape([-1, self.emb_size * 3])
        hidden = torch.relu(self.fc_layer1(embeddings))
        return self.fc_layer2(hidden)

pytorch_wordemb= PyTorchWordEmb()

a,b,c,d,e =␣
 ↪run_pytorch_gradient_descent(pytorch_wordemb,train4grams,valid4grams,batch_size=400,max_ite


plot_learning_curve(a,b,c,d,e)
```

```
Iter 0. [Val Acc 2%] [Train Acc 2%, Loss 5.523728]
Iter 500. [Val Acc 29%] [Train Acc 29%, Loss 3.087826]
Iter 1000. [Val Acc 31%] [Train Acc 32%, Loss 2.909438]
Iter 1500. [Val Acc 33%] [Train Acc 33%, Loss 2.996434]
Iter 2000. [Val Acc 34%] [Train Acc 35%, Loss 2.665084]
Iter 2500. [Val Acc 34%] [Train Acc 35%, Loss 2.529903]
Iter 3000. [Val Acc 35%] [Train Acc 36%, Loss 2.732430]
Iter 3500. [Val Acc 36%] [Train Acc 36%, Loss 2.648010]
Iter 4000. [Val Acc 36%] [Train Acc 37%, Loss 2.612783]
Iter 4500. [Val Acc 36%] [Train Acc 37%, Loss 2.384377]
Iter 5000. [Val Acc 36%] [Train Acc 37%, Loss 2.541126]
```


Learning Curve: Loss per Iteration
Loss
0.014
0.013
0.012
0.011
0.010
0.009
0.008
0.007
0.006
0
1000
2000
3000
4000
5000
Iterations


Learning Curve: Accuracy per Iteration
Accuracy
0.35
0.30
0.25
0.20
0.15
0.10
0.05
Train
Validation
0
1000
2000
3000
4000
5000
Iterations

1.3.5 Part (e) – 2 pts

Use the function `make_prediction` that you wrote earlier 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"

How do these predictions compared to the previous model?

Just like before, if all of your predictions are the same, train your model for more iterations, or change the hyperparameters in your model. You may need to do this even if your training accuracy is >=38%.

[63]:
```
# Your code goes here
print("you are a "␣
 ↪+make_prediction_torch(pytorch_wordemb,['you','are','a'])+'\n')
print("few companies show" +make_prediction_torch(pytorch_wordemb,["few",␣
 ↪"companies", "show"])+'\n')
print("there are no "␣
 ↪+make_prediction_torch(pytorch_wordemb,['there','are','no'])+'\n')
print("yesterday i was "␣
 ↪+make_prediction_torch(pytorch_wordemb,['yesterday','i','was'])+'\n')
print("the game had "␣
 ↪+make_prediction_torch(pytorch_wordemb,['the','game','had'])+'\n')
print("yesterday the federal "␣
 ↪+make_prediction_torch(pytorch_wordemb,['yesterday','the','federal'])+'\n')
# The predictions are almost identical with the exception of the last one,␣
 ↪which is seemingly better
```

```
you are a good

few companies show.

there are no other

yesterday i was nt

the game had to

yesterday the federal government
```

[63]:
```
' The predictions are almost identical with the exception of the last one, which
is seemingly better'
```

1.3.6 Part (f) – 1 pts

Report the test accuracy of your model

[64]:
```
# Write your code here
estimate_accuracy_torch(pytorch_wordemb,test4grams)
```

[64]:
```
0.3648679678530425
```

1.4 Question 4. Visualizing Word Embeddings

While training the `PyTorchMLP`, 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.

1.4.1 Part (a) – 2 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".

[0]:
```
word_emb_weights = list(pytorch_wordemb.word_emb_layer.parameters())[0]
word_emb = word_emb_weights.detach().numpy().T

# This avoids the most common word always being predicted, and allows␣
↪prediction to be logical given the current situation
```

1.4.2 Part (b) – 2 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. This code should look familiar, since we have seen it in project 1.

[67]:
```
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']])
```

```
0.17933208
0.022569353
```

Compute the 5 closest words to the following words:

- “four”
- “go”
- “what”
- “should”
- “school”
- “your”
- “yesterday”
- “not”

[117]:
```
# Write your code here

def ComputeClosestWords(myword):
  distances = []
  indexes = []
  for word,index in vocab_stoi.items():
    distances.append(similarities[vocab_stoi[myword], vocab_stoi[word]])
    indexes.append(index)

  # Mapping the list with its corresponding index
  Mapped = list(map(lambda x,y: tuple((x,y)), distances, indexes))
  # Sort it according to the smallest one.
  Mapped.sort(key=lambda x: x[0])

  FiveWords = []
  for i in range(5):
    FiveWords.append(vocab_itos[Mapped[i][1]])

  return(FiveWords)

print("your:", ComputeClosestWords("your"))
print("not:", ComputeClosestWords("not"))
print("four:", ComputeClosestWords("four"))
print("yesterday:", ComputeClosestWords("yesterday"))
print("go:", ComputeClosestWords("go"))
print("what:", ComputeClosestWords("what"))
print("should:", ComputeClosestWords("should"))
print("school:", ComputeClosestWords("school"))
```

```
your: ['former', 'children', 'season', 'day', 'between']
not: ['those', 'percent', 'several', 'former', 'the']
four: ['$', 'he', ':', 'dr.', 'nt']
yesterday: ['former', 'federal', 'ms.', 'get', 'come']
go: ['former', ',', 'dr.', 'when', 'if']
what: ['federal', '?', 'any', 'former', 'four']
should: ['former', '$', 'many', 'national', 'which']
school: ['the', 'former', 'federal', 'such', 'dr.']
```

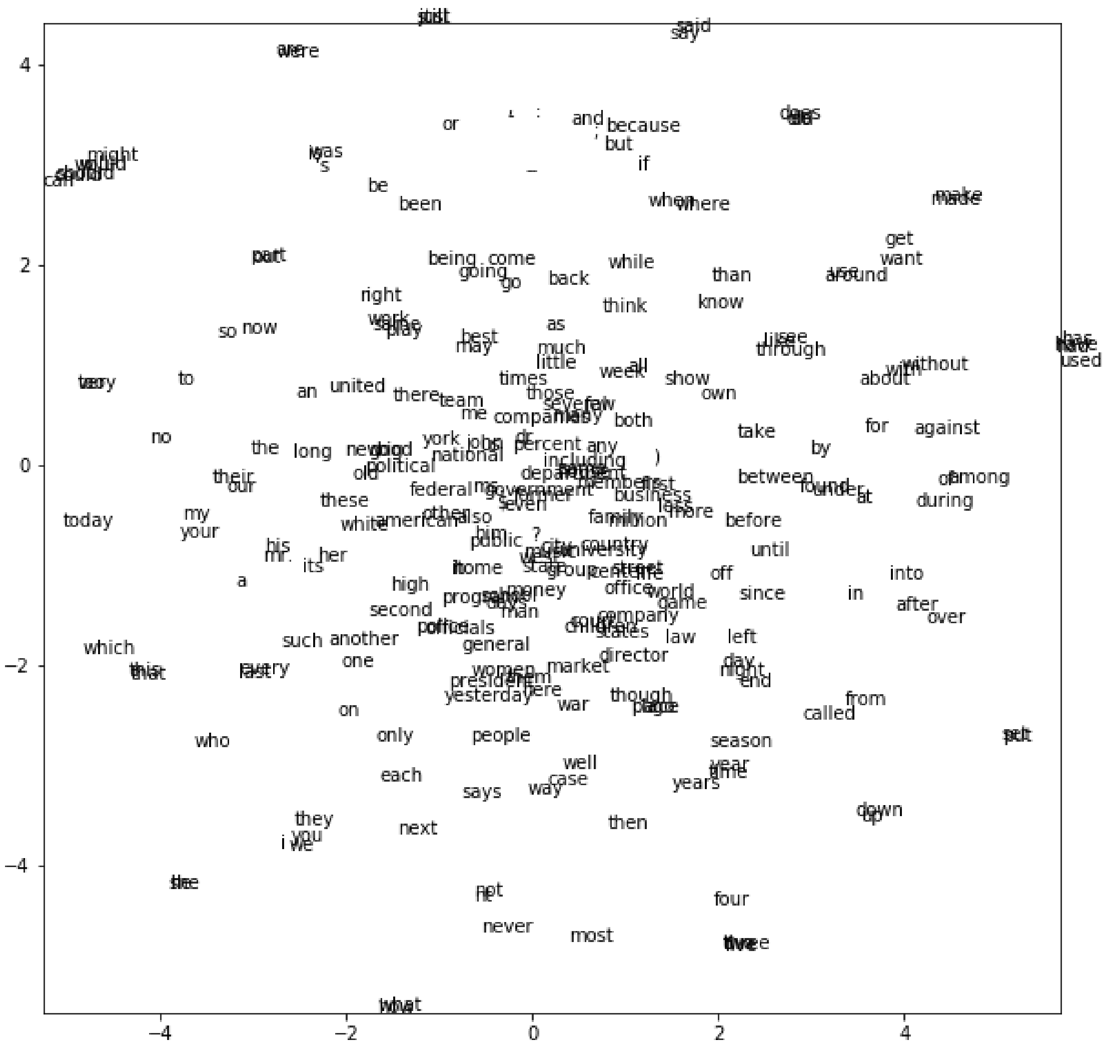
1.4.3 Part (c) – 2 pts

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.

[110]:
```
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()

# We can see have a few clusters. We mention 2 of them below which indeed␣
 ↪contain related words:
# 1) far top Left: might, would, could, can
# 2) Middle Bottom Left: they, you, i, we
```


4
2
0
−2
−4
−4
−2
0
2
4
said
say
were
or
and
because
but
if
might
was
be
been
when
where
make
made
get
want
being
come
going
go
back
while
than
know
around
think
right
work
play
so
now
as
best
may
much
little
through
has
used
without
about
to
an
united
there
team
times
those
week
all
show
own
few
me
both
take
for
against
no
the
long
york
john
percent
any
by
political
national
old
including
between
among
their
our
these
federal
government
business
members
more
less
during
today
my
your
white
american
also
him
public
family
million
before
until
his
mr.
her
its
a
city
country
university
home
state
group
street
off
into
high
second
money
office
world
game
since
in
after
over
officials
company
law
left
which
such
another
general
states
this
that
last
one
director
day
night
end
women
market
president
yesterday
though
war
from
called
on
who
only
people
season
well
case
way
says
years
year
time
each
they
you
we
next
then
down
up
four
never
most
what