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.

In [1]:

```
import pandas
import numpy as np
import matplotlib.pyplot as plt
import random
from collections import Counter
import time

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 `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:

In [4]:

```
from google.colab import drive
drive.mount('/content/gdrive')
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-4-4996ee3d8d09> in <module>
----> 1 from google.colab import drive
      2 drive.mount('/content/gdrive')

ModuleNotFoundError: No module named 'google'
```

Find the path to `raw_sentences.txt`:

In [5]:

```
file_path = '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:

In [6]:

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

```
!mkdir /content/gdrive/My\ Drive/CSC321
```

```
'ls' is not recognized as an internal or external command,
operable program or batch file.
The syntax of the command is incorrect.
```

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`.

In [7]:

```
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.

In [8]:

```
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.

In [9]:

```
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.

In [10]:

```
# Your code goes here
for s in train[:10]:
    print(s)

"""
Punctiations are treated like there own words, as can be seen with
the example of the commas in the first sentence. Words with apostrophes
are treated as two words; the first half before the apostrophe, and the
second half that includes the apostrophe.
"""
```

```
['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', '.']
```

Out[10]:

```
'\nPunctiations are treated like there own words, as can be seen with\nthe example of the commas i
```

```
 \nContractions are treated like their own words, as can be seen with\nthe example of the commas i
n the first sentence. Words with apostrophes\nare treated as two words; the first half before the
apostrophe, and the\nsecond half that includes the apostrophe.\n'
```

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.

In [56]:

```
words=[word for sentence in train for word in sentence]
vocab_counter = Counter(words)
print(vocab_counter.most_common(10))
for tup in vocab_counter.most_common(10):
    print(str(tup[0])+": "+str(round((100*tup[1])/len(words), 5))+"%")

# 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()}
```

```
[('.', 64297), ('it', 23118), (',', 19537), ('i', 17684), ('do', 16181), ('to', 15490), ('nt',
13009), ('?', 12881), ('the', 12583), ("'s", 12552)]
.: 10.69572%
it: 3.84565%
,: 3.24995%
i: 2.94171%
do: 2.69169%
to: 2.57674%
nt: 2.16403%
?: 2.14274%
the: 2.09317%
's: 2.08801%
```

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.

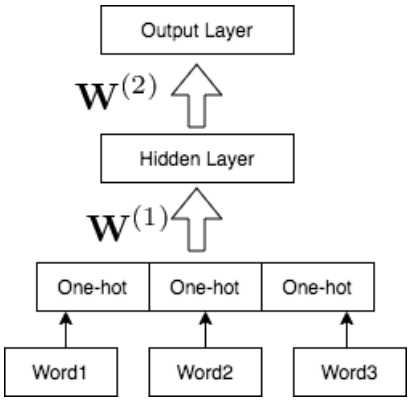
In [57]:

```
# 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]]
```

```
    """
    inds = []
    for sent in sents:
        temp = []
        for word in sent:
            temp.append(vocab_stoi[word])
        inds.append(temp)
    return inds

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]]
    """
    grams = []
    for sent in seqs:
        for i in range(len(sent)-3):
            grams.append(sent[i:i+4])
    return grams

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. 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:

In [14]:

```
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]
```

```
    return x[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
```

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**.

In [15]:

```
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])

        self.N = X.shape[0]
        self.X = X
        self.z1 =  np.matmul(self.X, (self.weights1).T) + self.bias1
        self.h = np.maximum(self.z1, np.zeros_like(self.z1))
        self.z2 = np.matmul(self.h, (self.weights2).T) + self.bias2
        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
```

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.

In [37]:

```
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
    start = time.time()
    last = start
    print("")
    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

            y = 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))
                now = time.time()
                print("last iteration: "+str(round(now-last, 5))+" seconds")
                print("elapsed time: "+str(round(now-start, 5))+" seconds\n")
                last = now

            if n >= max_iters:
                return

def load_model():
    model = NumpyMLPModel()
    data = np.load("model1.npy", allow_pickle=True)
```

```
    model.weights1 = data[0]
    model.bias1 = data[1]
    model.weights2 = data[2]
    model.bias2 = data[3]
    return model

numpy_mlp = NumpyMLPModel()
numpy_mlp.initializeParams()
run_gradient_descent(numpy_mlp, learning_rate=0.08, max_iters=15000)
```

```
Iter 100. [Val Acc 17%] [Train Acc 17%, Loss 5.157012]
last iteration: 6.0359 seconds
elapsed time: 6.0359 seconds

Iter 200. [Val Acc 17%] [Train Acc 17%, Loss 4.917038]
last iteration: 5.41053 seconds
elapsed time: 11.44643 seconds

Iter 300. [Val Acc 17%] [Train Acc 17%, Loss 4.647211]
last iteration: 4.95276 seconds
elapsed time: 16.39919 seconds

Iter 400. [Val Acc 17%] [Train Acc 17%, Loss 4.758050]
last iteration: 5.0116 seconds
elapsed time: 21.41079 seconds

Iter 500. [Val Acc 17%] [Train Acc 17%, Loss 4.402967]
last iteration: 5.04754 seconds
elapsed time: 26.45833 seconds

Iter 600. [Val Acc 17%] [Train Acc 17%, Loss 4.714210]
last iteration: 4.96967 seconds
elapsed time: 31.428 seconds

Iter 700. [Val Acc 17%] [Train Acc 17%, Loss 4.275777]
last iteration: 5.23101 seconds
elapsed time: 36.65901 seconds

Iter 800. [Val Acc 17%] [Train Acc 17%, Loss 4.821175]
last iteration: 4.88993 seconds
elapsed time: 41.54894 seconds

Iter 900. [Val Acc 17%] [Train Acc 17%, Loss 4.581405]
last iteration: 4.95874 seconds
elapsed time: 46.50768 seconds

Iter 1000. [Val Acc 17%] [Train Acc 17%, Loss 4.424040]
last iteration: 5.0106 seconds
elapsed time: 51.51828 seconds

Iter 1100. [Val Acc 17%] [Train Acc 17%, Loss 4.553234]
last iteration: 4.76326 seconds
elapsed time: 56.28154 seconds

Iter 1200. [Val Acc 17%] [Train Acc 17%, Loss 4.693345]
last iteration: 5.00661 seconds
elapsed time: 61.28816 seconds

Iter 1300. [Val Acc 17%] [Train Acc 17%, Loss 4.453204]
last iteration: 4.79219 seconds
elapsed time: 66.08034 seconds

Iter 1400. [Val Acc 17%] [Train Acc 17%, Loss 4.200523]
last iteration: 5.18015 seconds
elapsed time: 71.26049 seconds

Iter 1500. [Val Acc 17%] [Train Acc 17%, Loss 4.276233]
last iteration: 4.81114 seconds
elapsed time: 76.07163 seconds

Iter 1600. [Val Acc 17%] [Train Acc 17%, Loss 4.108299]
last iteration: 5.5711 seconds
elapsed time: 81.64273 seconds

Iter 1700. [Val Acc 17%] [Train Acc 17%, Loss 4.529019]
last iteration: 5.66286 seconds
```

```
elapsed time: 87.30559 seconds

Iter 1800. [Val Acc 18%] [Train Acc 18%, Loss 4.269886]
last iteration: 5.22802 seconds
elapsed time: 92.53361 seconds

Iter 1900. [Val Acc 18%] [Train Acc 18%, Loss 4.203082]
last iteration: 4.89691 seconds
elapsed time: 97.43051 seconds

Iter 2000. [Val Acc 19%] [Train Acc 19%, Loss 4.475025]
last iteration: 5.63693 seconds
elapsed time: 103.06744 seconds

Iter 2100. [Val Acc 19%] [Train Acc 19%, Loss 4.097056]
last iteration: 5.00262 seconds
elapsed time: 108.07006 seconds

Iter 2200. [Val Acc 19%] [Train Acc 19%, Loss 4.332635]
last iteration: 4.96971 seconds
elapsed time: 113.03978 seconds

Iter 2300. [Val Acc 19%] [Train Acc 19%, Loss 4.286814]
last iteration: 4.80914 seconds
elapsed time: 117.84892 seconds

Iter 2400. [Val Acc 19%] [Train Acc 19%, Loss 3.950877]
last iteration: 4.75628 seconds
elapsed time: 122.6052 seconds

Iter 2500. [Val Acc 20%] [Train Acc 20%, Loss 4.531344]
last iteration: 5.13028 seconds
elapsed time: 127.73548 seconds

Iter 2600. [Val Acc 20%] [Train Acc 21%, Loss 4.225405]
last iteration: 4.78221 seconds
elapsed time: 132.51769 seconds

Iter 2700. [Val Acc 20%] [Train Acc 20%, Loss 4.026461]
last iteration: 4.74432 seconds
elapsed time: 137.26201 seconds

Iter 2800. [Val Acc 21%] [Train Acc 21%, Loss 4.133651]
last iteration: 5.32277 seconds
elapsed time: 142.58477 seconds

Iter 2900. [Val Acc 21%] [Train Acc 21%, Loss 4.494134]
last iteration: 5.84138 seconds
elapsed time: 148.42615 seconds

Iter 3000. [Val Acc 21%] [Train Acc 21%, Loss 4.010564]
last iteration: 5.49331 seconds
elapsed time: 153.91947 seconds

Iter 3100. [Val Acc 22%] [Train Acc 22%, Loss 3.871241]
last iteration: 5.23301 seconds
elapsed time: 159.15248 seconds

Iter 3200. [Val Acc 22%] [Train Acc 22%, Loss 4.168862]
last iteration: 5.21904 seconds
elapsed time: 164.37152 seconds

Iter 3300. [Val Acc 22%] [Train Acc 22%, Loss 3.971784]
last iteration: 5.28088 seconds
elapsed time: 169.6524 seconds

Iter 3400. [Val Acc 22%] [Train Acc 22%, Loss 4.029842]
last iteration: 5.39557 seconds
elapsed time: 175.04797 seconds

Iter 3500. [Val Acc 23%] [Train Acc 23%, Loss 3.980051]
last iteration: 4.77423 seconds
elapsed time: 179.8222 seconds

Iter 3600. [Val Acc 23%] [Train Acc 23%, Loss 3.677514]
last iteration: 4.7134 seconds
elapsed time: 184.5356 seconds
```

```
Iter 3700. [Val Acc 23%] [Train Acc 23%, Loss 3.989220]
last iteration: 5.77655 seconds
elapsed time: 190.31215 seconds

Iter 3800. [Val Acc 23%] [Train Acc 23%, Loss 4.042273]
last iteration: 5.32476 seconds
elapsed time: 195.63692 seconds

Iter 3900. [Val Acc 23%] [Train Acc 24%, Loss 3.877979]
last iteration: 4.81513 seconds
elapsed time: 200.45204 seconds

Iter 4000. [Val Acc 23%] [Train Acc 24%, Loss 4.027058]
last iteration: 4.81412 seconds
elapsed time: 205.26617 seconds

Iter 4100. [Val Acc 24%] [Train Acc 24%, Loss 3.809827]
last iteration: 4.91138 seconds
elapsed time: 210.17754 seconds

Iter 4200. [Val Acc 24%] [Train Acc 24%, Loss 4.035900]
last iteration: 4.9348 seconds
elapsed time: 215.11235 seconds

Iter 4300. [Val Acc 24%] [Train Acc 25%, Loss 3.997222]
last iteration: 4.73434 seconds
elapsed time: 219.84669 seconds

Iter 4400. [Val Acc 24%] [Train Acc 24%, Loss 3.604698]
last iteration: 4.78321 seconds
elapsed time: 224.6299 seconds

Iter 4500. [Val Acc 24%] [Train Acc 24%, Loss 3.625452]
last iteration: 4.92184 seconds
elapsed time: 229.55174 seconds

Iter 4600. [Val Acc 24%] [Train Acc 25%, Loss 3.812889]
last iteration: 4.74531 seconds
elapsed time: 234.29705 seconds

Iter 4700. [Val Acc 24%] [Train Acc 25%, Loss 3.945082]
last iteration: 4.81213 seconds
elapsed time: 239.10918 seconds

Iter 4800. [Val Acc 24%] [Train Acc 25%, Loss 3.959831]
last iteration: 4.77024 seconds
elapsed time: 243.87943 seconds

Iter 4900. [Val Acc 24%] [Train Acc 25%, Loss 3.767312]
last iteration: 5.22802 seconds
elapsed time: 249.10745 seconds

Iter 5000. [Val Acc 24%] [Train Acc 25%, Loss 3.719713]
last iteration: 5.09338 seconds
elapsed time: 254.20083 seconds

Iter 5100. [Val Acc 25%] [Train Acc 25%, Loss 3.789946]
last iteration: 4.91186 seconds
elapsed time: 259.11269 seconds

Iter 5200. [Val Acc 25%] [Train Acc 25%, Loss 3.712813]
last iteration: 4.98168 seconds
elapsed time: 264.09437 seconds

Iter 5300. [Val Acc 25%] [Train Acc 25%, Loss 3.900471]
last iteration: 5.33972 seconds
elapsed time: 269.4341 seconds

Iter 5400. [Val Acc 25%] [Train Acc 25%, Loss 4.065673]
last iteration: 5.05648 seconds
elapsed time: 274.49057 seconds

Iter 5500. [Val Acc 25%] [Train Acc 25%, Loss 3.631937]
last iteration: 4.80716 seconds
elapsed time: 279.29774 seconds
```

```
Iter 5600. [Val Acc 25%] [Train Acc 26%, Loss 3.762074]
last iteration: 5.0056 seconds
elapsed time: 284.30334 seconds

Iter 5700. [Val Acc 25%] [Train Acc 26%, Loss 3.787734]
last iteration: 4.99265 seconds
elapsed time: 289.29599 seconds

Iter 5800. [Val Acc 25%] [Train Acc 26%, Loss 4.068585]
last iteration: 4.98966 seconds
elapsed time: 294.28565 seconds

Iter 5900. [Val Acc 25%] [Train Acc 26%, Loss 3.457975]
last iteration: 4.862 seconds
elapsed time: 299.14765 seconds

Iter 6000. [Val Acc 26%] [Train Acc 26%, Loss 3.361753]
last iteration: 4.80415 seconds
elapsed time: 303.9518 seconds

Iter 6100. [Val Acc 25%] [Train Acc 26%, Loss 3.679491]
last iteration: 5.59005 seconds
elapsed time: 309.54186 seconds

Iter 6200. [Val Acc 26%] [Train Acc 26%, Loss 3.709185]
last iteration: 4.9388 seconds
elapsed time: 314.48066 seconds

Iter 6300. [Val Acc 26%] [Train Acc 26%, Loss 3.597729]
last iteration: 4.64357 seconds
elapsed time: 319.12423 seconds

Iter 6400. [Val Acc 26%] [Train Acc 26%, Loss 3.461937]
last iteration: 4.88893 seconds
elapsed time: 324.01316 seconds

Iter 6500. [Val Acc 26%] [Train Acc 26%, Loss 3.370166]
last iteration: 4.92284 seconds
elapsed time: 328.936 seconds

Iter 6600. [Val Acc 26%] [Train Acc 27%, Loss 3.546579]
last iteration: 5.34172 seconds
elapsed time: 334.27772 seconds

Iter 6700. [Val Acc 26%] [Train Acc 27%, Loss 3.786743]
last iteration: 4.96972 seconds
elapsed time: 339.24744 seconds

Iter 6800. [Val Acc 26%] [Train Acc 27%, Loss 3.599861]
last iteration: 5.1961 seconds
elapsed time: 344.44354 seconds

Iter 6900. [Val Acc 26%] [Train Acc 27%, Loss 3.313291]
last iteration: 5.58307 seconds
elapsed time: 350.02661 seconds

Iter 7000. [Val Acc 26%] [Train Acc 27%, Loss 3.932094]
last iteration: 4.79518 seconds
elapsed time: 354.82179 seconds

Iter 7100. [Val Acc 26%] [Train Acc 27%, Loss 3.528189]
last iteration: 4.83706 seconds
elapsed time: 359.65885 seconds

Iter 7200. [Val Acc 26%] [Train Acc 27%, Loss 3.361747]
last iteration: 4.83009 seconds
elapsed time: 364.48894 seconds

Iter 7300. [Val Acc 26%] [Train Acc 27%, Loss 3.472092]
last iteration: 4.96373 seconds
elapsed time: 369.45266 seconds

Iter 7400. [Val Acc 26%] [Train Acc 27%, Loss 3.487114]
last iteration: 5.21206 seconds
elapsed time: 374.66473 seconds

Iter 7500. [Val Acc 27%] [Train Acc 27%, Loss 3.631711]
```

```
last iteration: 4.7513 seconds
elapsed time: 379.41602 seconds

Iter 7600. [Val Acc 26%] [Train Acc 27%, Loss 3.602961]
last iteration: 4.87995 seconds
elapsed time: 384.29597 seconds

Iter 7700. [Val Acc 27%] [Train Acc 27%, Loss 3.392856]
last iteration: 4.73035 seconds
elapsed time: 389.02633 seconds

Iter 7800. [Val Acc 27%] [Train Acc 27%, Loss 3.442197]
last iteration: 4.75229 seconds
elapsed time: 393.77862 seconds

Iter 7900. [Val Acc 26%] [Train Acc 27%, Loss 3.575725]
last iteration: 4.84504 seconds
elapsed time: 398.62366 seconds

Iter 8000. [Val Acc 27%] [Train Acc 27%, Loss 3.300799]
last iteration: 5.80049 seconds
elapsed time: 404.42415 seconds

Iter 8100. [Val Acc 27%] [Train Acc 27%, Loss 3.634378]
last iteration: 5.30182 seconds
elapsed time: 409.72598 seconds

Iter 8200. [Val Acc 27%] [Train Acc 27%, Loss 3.216212]
last iteration: 5.35668 seconds
elapsed time: 415.08265 seconds

Iter 8300. [Val Acc 27%] [Train Acc 27%, Loss 3.515132]
last iteration: 4.65056 seconds
elapsed time: 419.73322 seconds

Iter 8400. [Val Acc 27%] [Train Acc 27%, Loss 3.533975]
last iteration: 4.94478 seconds
elapsed time: 424.67799 seconds

Iter 8500. [Val Acc 27%] [Train Acc 27%, Loss 3.330265]
last iteration: 4.95675 seconds
elapsed time: 429.63474 seconds

Iter 8600. [Val Acc 27%] [Train Acc 27%, Loss 3.607432]
last iteration: 5.07642 seconds
elapsed time: 434.71117 seconds

Iter 8700. [Val Acc 27%] [Train Acc 27%, Loss 3.329062]
last iteration: 4.75828 seconds
elapsed time: 439.46944 seconds

Iter 8800. [Val Acc 27%] [Train Acc 28%, Loss 3.654950]
last iteration: 4.85203 seconds
elapsed time: 444.32147 seconds

Iter 8900. [Val Acc 27%] [Train Acc 27%, Loss 3.415573]
last iteration: 5.70375 seconds
elapsed time: 450.02522 seconds

Iter 9000. [Val Acc 27%] [Train Acc 28%, Loss 3.608850]
last iteration: 5.14724 seconds
elapsed time: 455.17246 seconds

Iter 9100. [Val Acc 27%] [Train Acc 28%, Loss 3.619944]
last iteration: 5.58905 seconds
elapsed time: 460.76151 seconds

Iter 9200. [Val Acc 27%] [Train Acc 28%, Loss 3.170861]
last iteration: 5.6828 seconds
elapsed time: 466.44431 seconds

Iter 9300. [Val Acc 27%] [Train Acc 28%, Loss 3.573332]
last iteration: 5.74763 seconds
elapsed time: 472.19195 seconds

Iter 9400. [Val Acc 27%] [Train Acc 28%, Loss 3.428316]
last iteration: 5.58108 seconds
```

```
elapsed time: 477.77302 seconds

Iter 9500. [Val Acc 27%] [Train Acc 28%, Loss 3.488060]
last iteration: 5.67981 seconds
elapsed time: 483.45284 seconds

Iter 9600. [Val Acc 28%] [Train Acc 28%, Loss 3.519185]
last iteration: 5.57609 seconds
elapsed time: 489.02892 seconds

Iter 9700. [Val Acc 27%] [Train Acc 28%, Loss 3.508342]
last iteration: 5.54218 seconds
elapsed time: 494.57111 seconds

Iter 9800. [Val Acc 27%] [Train Acc 28%, Loss 3.230712]
last iteration: 5.57808 seconds
elapsed time: 500.14919 seconds

Iter 9900. [Val Acc 27%] [Train Acc 28%, Loss 3.538315]
last iteration: 5.3547 seconds
elapsed time: 505.50389 seconds

Iter 10000. [Val Acc 28%] [Train Acc 28%, Loss 3.098868]
last iteration: 5.70274 seconds
elapsed time: 511.20662 seconds

Iter 10100. [Val Acc 28%] [Train Acc 28%, Loss 3.399303]
last iteration: 5.16419 seconds
elapsed time: 516.37081 seconds

Iter 10200. [Val Acc 28%] [Train Acc 28%, Loss 3.366771]
last iteration: 5.67582 seconds
elapsed time: 522.04664 seconds

Iter 10300. [Val Acc 28%] [Train Acc 28%, Loss 3.303192]
last iteration: 5.93912 seconds
elapsed time: 527.98576 seconds

Iter 10400. [Val Acc 28%] [Train Acc 28%, Loss 3.445169]
last iteration: 5.2729 seconds
elapsed time: 533.25866 seconds

Iter 10500. [Val Acc 28%] [Train Acc 28%, Loss 3.149304]
last iteration: 5.00262 seconds
elapsed time: 538.26128 seconds

Iter 10600. [Val Acc 28%] [Train Acc 28%, Loss 3.710706]
last iteration: 5.09987 seconds
elapsed time: 543.36115 seconds

Iter 10700. [Val Acc 28%] [Train Acc 28%, Loss 3.095655]
last iteration: 5.53719 seconds
elapsed time: 548.89835 seconds

Iter 10800. [Val Acc 28%] [Train Acc 28%, Loss 3.445182]
last iteration: 5.04451 seconds
elapsed time: 553.94286 seconds

Iter 10900. [Val Acc 28%] [Train Acc 28%, Loss 3.433156]
last iteration: 4.77723 seconds
elapsed time: 558.72008 seconds

Iter 11000. [Val Acc 28%] [Train Acc 28%, Loss 3.341609]
last iteration: 4.71539 seconds
elapsed time: 563.43547 seconds

Iter 11100. [Val Acc 28%] [Train Acc 28%, Loss 3.352846]
last iteration: 5.08839 seconds
elapsed time: 568.52387 seconds

Iter 11200. [Val Acc 28%] [Train Acc 28%, Loss 2.841020]
last iteration: 4.6755 seconds
elapsed time: 573.19937 seconds

Iter 11300. [Val Acc 28%] [Train Acc 28%, Loss 3.082048]
last iteration: 7.04417 seconds
elapsed time: 580.24353 seconds
```

```
Iter 11400. [Val Acc 28%] [Train Acc 28%, Loss 3.381079]
last iteration: 6.61682 seconds
elapsed time: 586.86035 seconds

Iter 11500. [Val Acc 28%] [Train Acc 29%, Loss 3.567849]
last iteration: 5.72668 seconds
elapsed time: 592.58704 seconds

Iter 11600. [Val Acc 28%] [Train Acc 29%, Loss 3.442981]
last iteration: 5.25495 seconds
elapsed time: 597.84199 seconds

Iter 11700. [Val Acc 28%] [Train Acc 28%, Loss 3.164452]
last iteration: 5.1981 seconds
elapsed time: 603.04009 seconds

Iter 11800. [Val Acc 28%] [Train Acc 29%, Loss 3.383758]
last iteration: 4.92583 seconds
elapsed time: 607.96592 seconds

Iter 11900. [Val Acc 28%] [Train Acc 29%, Loss 3.365125]
last iteration: 4.50795 seconds
elapsed time: 612.47386 seconds

Iter 12000. [Val Acc 28%] [Train Acc 29%, Loss 3.439294]
last iteration: 4.66752 seconds
elapsed time: 617.14138 seconds

Iter 12100. [Val Acc 28%] [Train Acc 29%, Loss 3.244426]
last iteration: 4.61167 seconds
elapsed time: 621.75305 seconds

Iter 12200. [Val Acc 28%] [Train Acc 29%, Loss 3.216751]
last iteration: 5.35867 seconds
elapsed time: 627.11172 seconds

Iter 12300. [Val Acc 28%] [Train Acc 29%, Loss 3.180321]
last iteration: 4.99664 seconds
elapsed time: 632.10836 seconds

Iter 12400. [Val Acc 28%] [Train Acc 29%, Loss 3.264039]
last iteration: 5.88726 seconds
elapsed time: 637.99562 seconds

Iter 12500. [Val Acc 28%] [Train Acc 29%, Loss 3.083204]
last iteration: 5.61798 seconds
elapsed time: 643.61359 seconds

Iter 12600. [Val Acc 29%] [Train Acc 29%, Loss 3.323577]
last iteration: 4.64458 seconds
elapsed time: 648.25818 seconds

Iter 12700. [Val Acc 28%] [Train Acc 29%, Loss 3.168982]
last iteration: 4.63261 seconds
elapsed time: 652.89079 seconds

Iter 12800. [Val Acc 28%] [Train Acc 29%, Loss 3.002505]
last iteration: 7.72634 seconds
elapsed time: 660.61713 seconds

Iter 12900. [Val Acc 29%] [Train Acc 29%, Loss 3.125154]
last iteration: 6.26325 seconds
elapsed time: 666.88038 seconds

Iter 13000. [Val Acc 29%] [Train Acc 29%, Loss 3.599047]
last iteration: 5.57709 seconds
elapsed time: 672.45747 seconds

Iter 13100. [Val Acc 29%] [Train Acc 29%, Loss 3.214170]
last iteration: 5.09139 seconds
elapsed time: 677.54885 seconds

Iter 13200. [Val Acc 29%] [Train Acc 29%, Loss 3.868356]
last iteration: 4.86798 seconds
elapsed time: 682.41684 seconds
```

```
Iter 13300. [Val Acc 29%] [Train Acc 29%, Loss 3.216472]
last iteration: 4.73634 seconds
elapsed time: 687.15318 seconds

Iter 13400. [Val Acc 29%] [Train Acc 29%, Loss 3.107086]
last iteration: 4.49697 seconds
elapsed time: 691.65015 seconds

Iter 13500. [Val Acc 29%] [Train Acc 29%, Loss 3.427946]
last iteration: 4.46606 seconds
elapsed time: 696.11621 seconds

Iter 13600. [Val Acc 29%] [Train Acc 29%, Loss 3.112631]
last iteration: 4.62961 seconds
elapsed time: 700.74583 seconds

Iter 13700. [Val Acc 29%] [Train Acc 29%, Loss 3.429849]
last iteration: 4.59571 seconds
elapsed time: 705.34154 seconds

Iter 13800. [Val Acc 29%] [Train Acc 29%, Loss 3.355166]
last iteration: 5.06645 seconds
elapsed time: 710.40799 seconds

Iter 13900. [Val Acc 29%] [Train Acc 29%, Loss 2.939028]
last iteration: 5.05149 seconds
elapsed time: 715.45948 seconds

Iter 14000. [Val Acc 29%] [Train Acc 29%, Loss 2.922391]
last iteration: 4.87297 seconds
elapsed time: 720.33246 seconds

Iter 14100. [Val Acc 29%] [Train Acc 29%, Loss 3.117308]
last iteration: 4.69445 seconds
elapsed time: 725.0269 seconds

Iter 14200. [Val Acc 29%] [Train Acc 29%, Loss 3.069769]
last iteration: 4.61266 seconds
elapsed time: 729.63957 seconds

Iter 14300. [Val Acc 29%] [Train Acc 29%, Loss 3.121438]
last iteration: 7.98964 seconds
elapsed time: 737.6292 seconds

Iter 14400. [Val Acc 29%] [Train Acc 29%, Loss 3.377222]
last iteration: 6.30115 seconds
elapsed time: 743.93036 seconds

Iter 14500. [Val Acc 29%] [Train Acc 29%, Loss 3.341363]
last iteration: 5.55315 seconds
elapsed time: 749.48351 seconds

Iter 14600. [Val Acc 29%] [Train Acc 30%, Loss 3.453232]
last iteration: 5.15522 seconds
elapsed time: 754.63872 seconds

Iter 14700. [Val Acc 29%] [Train Acc 29%, Loss 3.125353]
last iteration: 4.76226 seconds
elapsed time: 759.40099 seconds

Iter 14800. [Val Acc 29%] [Train Acc 29%, Loss 2.924799]
last iteration: 5.06446 seconds
elapsed time: 764.46545 seconds

Iter 14900. [Val Acc 29%] [Train Acc 30%, Loss 2.952323]
last iteration: 4.54086 seconds
elapsed time: 769.0063 seconds

Iter 15000. [Val Acc 29%] [Train Acc 30%, Loss 3.160202]
last iteration: 4.65156 seconds
elapsed time: 773.65787 seconds
```

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:

In [18]:

```
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):
        X = inp.reshape([-1, 750])
        N = X.shape[0]
        z1 = self.layer1(X)
        h = torch.max(z1, torch.zeros_like(z1))
        z2 = self.layer2(h)
        return z2
```

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.

In [19]:

```
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
    start = time.time()
    last = start
    print("")
    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.forward(xs)
            loss = criterion(zs, ts)
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()

            # 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))
                now = time.time()
                print("last iteration: "+str(round(now-last, 5))+" seconds")
                print("elapsed time: "+str(round(now-start, 5))+" seconds\n")
                last = now
                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
```

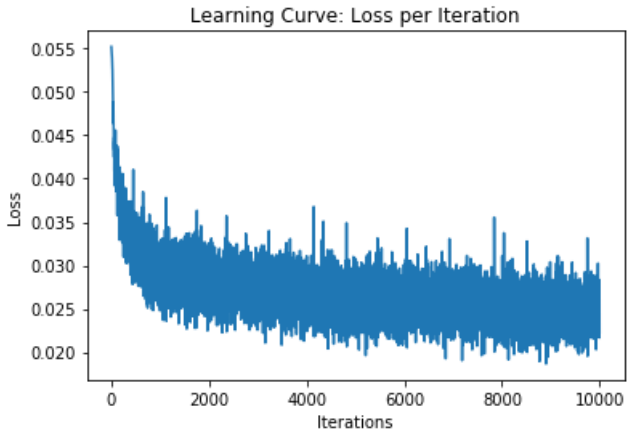
```
def plot_learning_curve(iters, losses, iters_sub, train_accs, val_accs):
    """
    Plot the learning curve.
    """
    plt.title("Learning Curve: Loss per Iteration")
    plt.plot(iters, losses, label="Train")
    plt.xlabel("Iterations")
    plt.ylabel("Loss")
    plt.show()

    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()
```

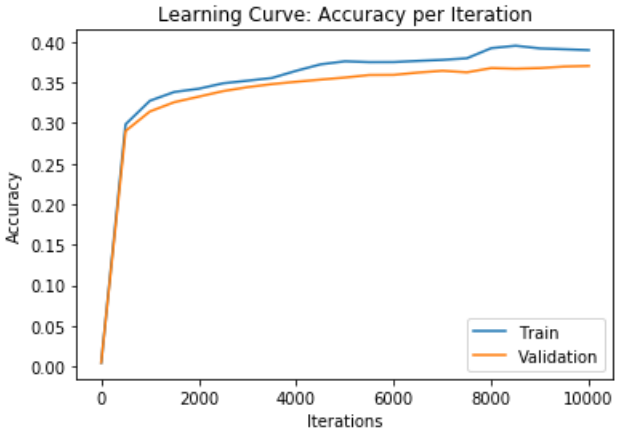
In [23]:

```
pytorch_mlp = PyTorchMLP()
learning_curve_info = run_pytorch_gradient_descent(pytorch_mlp, learning_rate=0.001,
max_iters=10000, checkpoint_path="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)

pytorch_mlp.load_state_dict(torch.load('ckpt-10000.pk'))
```

```
Iter 0. [Val Acc 0%] [Train Acc 0%, Loss 5.520795]
last iteration: 2.59724 seconds
elapsed time: 2.59724 seconds

Iter 500. [Val Acc 29%] [Train Acc 30%, Loss 2.928955]
last iteration: 6.04085 seconds
elapsed time: 8.63808 seconds

Iter 1000. [Val Acc 31%] [Train Acc 33%, Loss 2.732797]
last iteration: 5.52223 seconds
elapsed time: 14.16032 seconds

Iter 1500. [Val Acc 33%] [Train Acc 34%, Loss 2.678330]
last iteration: 5.68879 seconds
elapsed time: 19.8491 seconds

Iter 2000. [Val Acc 33%] [Train Acc 34%, Loss 2.645863]
last iteration: 5.30482 seconds
elapsed time: 25.15392 seconds

Iter 2500. [Val Acc 34%] [Train Acc 35%, Loss 2.621405]
last iteration: 5.31678 seconds
elapsed time: 30.4707 seconds

Iter 3000. [Val Acc 34%] [Train Acc 35%, Loss 2.448958]
last iteration: 5.40056 seconds
elapsed time: 35.87126 seconds

Iter 3500. [Val Acc 35%] [Train Acc 36%, Loss 2.410585]
last iteration: 5.48833 seconds
elapsed time: 41.35959 seconds

Iter 4000. [Val Acc 35%] [Train Acc 36%, Loss 2.612296]
last iteration: 5.85235 seconds
elapsed time: 47.21194 seconds

Iter 4500. [Val Acc 35%] [Train Acc 37%, Loss 2.602068]
last iteration: 5.58008 seconds
elapsed time: 52.79202 seconds

Iter 5000. [Val Acc 36%] [Train Acc 38%, Loss 2.631595]
last iteration: 5.97004 seconds
elapsed time: 58.76206 seconds
```

```
Iter 5500. [Val Acc 36%] [Train Acc 38%, Loss 2.463904]
last iteration: 5.53919 seconds
elapsed time: 64.30125 seconds

Iter 6000. [Val Acc 36%] [Train Acc 38%, Loss 2.610732]
last iteration: 6.27921 seconds
elapsed time: 70.58046 seconds

Iter 6500. [Val Acc 36%] [Train Acc 38%, Loss 2.540700]
last iteration: 6.67515 seconds
elapsed time: 77.25561 seconds

Iter 7000. [Val Acc 36%] [Train Acc 38%, Loss 2.927498]
last iteration: 9.95438 seconds
elapsed time: 87.20999 seconds

Iter 7500. [Val Acc 36%] [Train Acc 38%, Loss 2.573669]
last iteration: 8.61506 seconds
elapsed time: 95.82505 seconds

Iter 8000. [Val Acc 37%] [Train Acc 39%, Loss 2.087473]
last iteration: 7.03118 seconds
elapsed time: 102.85623 seconds

Iter 8500. [Val Acc 37%] [Train Acc 40%, Loss 2.787875]
last iteration: 6.24729 seconds
elapsed time: 109.10353 seconds

Iter 9000. [Val Acc 37%] [Train Acc 39%, Loss 2.399692]
last iteration: 5.85834 seconds
elapsed time: 114.96186 seconds

Iter 9500. [Val Acc 37%] [Train Acc 39%, Loss 2.543618]
last iteration: 5.64191 seconds
elapsed time: 120.60378 seconds

Iter 10000. [Val Acc 37%] [Train Acc 39%, Loss 2.230226]
last iteration: 5.51226 seconds
elapsed time: 126.11604 seconds
```





Out[23]:

```
<All keys matched successfully>
```

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.

In [45]:

```
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

    try:
        index1 = vocab_stoi[sentence[-3]]
        index2 = vocab_stoi[sentence[-2]]
        index3 = vocab_stoi[sentence[-1]]
    except Exception:
        return None # if any given word is not in the training set, model can't predict
    x = np.zeros((250, 3))
    x[:,0] = make_onehot(index1)
    x[:,1] = make_onehot(index2)
    x[:,2] = make_onehot(index3)
    x = torch.Tensor(x.T)
    y = model.forward(x)
    return vocab_itos[torch.argmax(y).item()]
```

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?

In [46]:

```
print("You are a "+make_prediction_torch(pytorch_mlp, ["you", "are", "a"]))
print("few companies show "+make_prediction_torch(pytorch_mlp, ["few", "companies", "show"]))
print("There are no "+make_prediction_torch(pytorch_mlp, ["there", "are", "no"]))
print("yesterday i was "+make_prediction_torch(pytorch_mlp, ["yesterday", "i", "was"]))
print("the game had "+make_prediction_torch(pytorch_mlp, ["the", "game", "had"]))
print("yesterday the federal "+make_prediction_torch(pytorch_mlp, ["yesterday", "the",
"federal"]))
```

```
You are a man
```

```
few companies show .
There are no other
yesterday i was nt
the game had to
yesterday the federal government
```

Part (g) -- 1 points

Report the test accuracy of your model

In [29]:

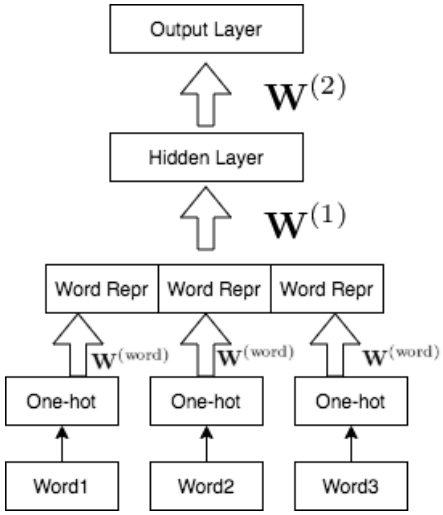
```
print("[Test acc: "+str(round(estimate_accuracy_torch(pytorch_mlp, test4grams)*100, 3))+"%]")
```

```
[Test acc: 37.393%]
```

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.

Part (a) - 10 pts

Complete the methods in `NumpyWordEmbModel`.

In [38]:

```
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()

    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.
        """
        self.X = inputs.reshape([-1, 750])
        self.N = self.X.shape[0]
        self.pre_emb = np.zeros((self.X.shape[0], self.emb_size*3)) # I'm going to slice this in a
weird way
        for i in range(3):
            self.pre_emb[:,self.emb_size*i:self.emb_size*(i+1)] =
np.matmul(self.X[:,self.vocab_size*i:self.vocab_size*(i+1)], (self.emb_weights).T)
        self.emb = np.maximum(self.pre_emb, np.zeros_like(self.pre_emb)) # ReLU
        self.z1 = np.matmul(self.emb, (self.weights1).T) + self.bias1
        self.h = np.maximum(self.z1, np.zeros_like(self.z1))
        self.z2 = np.matmul(self.h, (self.weights2).T) + self.bias2
        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.
        """
        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.emb)
        self.b1_bar = np.dot(self.z1_bar.T, np.ones(self.N))
        #
        self.h0_bar = np.matmul(self.z1_bar, self.weights1)
        self.z0_bar = self.h0_bar * (self.pre_emb > 0)
        self.emb_weights_bar = np.matmul(self.w1_bar, np.matmul(self.z0_bar.T, self.X[:,:250]))
        + np.matmul(self.w1_bar, np.matmul(self.z0_bar.T, self.X[:,250:500]))
        + np.matmul(self.w1_bar, np.matmul(self.z0_bar.T, self.X[:,500:]))

    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
        self.emb_weights = self.emb_weights    - alpha * self.emb_weights_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.emb = None # I just realized my naming convention is very confusing
        self.z2_bar = 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
        self.h0_bar = None
        self.z0_bar = None
        self.emb_weights_bar = None
```

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).

In [39]:

```
numpy_wordemb = NumpyWordEmbModel()
run_gradient_descent(numpy_wordemb, train4grams[:64], batch_size=64, max_iters=500)
'''
we didn't get total convergence because we got stuck calculating the gradient for
the embedded weight, but we know that this should converge if we had the correct
gradient (we were very close to getting it, we just couldn't get the dimensions
to align)
'''
```

```
Iter 100. [Val Acc 17%] [Train Acc 12%, Loss 5.118052]
last iteration: 1.35139 seconds
elapsed time: 1.35139 seconds

Iter 200. [Val Acc 17%] [Train Acc 12%, Loss 4.775714]
last iteration: 1.36834 seconds
elapsed time: 2.71974 seconds

Iter 300. [Val Acc 17%] [Train Acc 12%, Loss 4.526086]
last iteration: 1.28855 seconds
elapsed time: 4.00829 seconds

Iter 400. [Val Acc 17%] [Train Acc 12%, Loss 4.357959]
last iteration: 1.25663 seconds
elapsed time: 5.26492 seconds

Iter 500. [Val Acc 17%] [Train Acc 12%, Loss 4.238633]
last iteration: 1.21774 seconds
elapsed time: 6.48267 seconds
```

Out[39]:

```
"\nwe didn't get total convergence because we got stuck calculating the gradient for\nthe embedded
weight, but we know that this should converge if we had the correct\ngradient (we were very close
to getting it, we just couldn't get the dimensions\nto align)\n"
```

Part (c) -- 2 pts

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

In [40]:

```
run_gradient_descent(numpy_wordemb, max_iters=5000)
'''
once again, no convergence for the same reason as above
'''
```

```
Iter 100. [Val Acc 17%] [Train Acc 17%, Loss 4.542798]
last iteration: 4.08806 seconds
```

last iteration: 1.00000 seconds
elapsed time: 4.08806 seconds

Iter 200. [Val Acc 17%] [Train Acc 17%, Loss 4.652813]
last iteration: 3.63229 seconds
elapsed time: 7.72035 seconds

Iter 300. [Val Acc 17%] [Train Acc 17%, Loss 4.639475]
last iteration: 3.97637 seconds
elapsed time: 11.69672 seconds

Iter 400. [Val Acc 17%] [Train Acc 17%, Loss 4.726529]
last iteration: 4.05815 seconds
elapsed time: 15.75487 seconds

Iter 500. [Val Acc 17%] [Train Acc 17%, Loss 4.639154]
last iteration: 3.96341 seconds
elapsed time: 19.71828 seconds

Iter 600. [Val Acc 17%] [Train Acc 17%, Loss 4.387364]
last iteration: 4.04418 seconds
elapsed time: 23.76245 seconds

Iter 700. [Val Acc 17%] [Train Acc 17%, Loss 4.663090]
last iteration: 3.80981 seconds
elapsed time: 27.57227 seconds

Iter 800. [Val Acc 17%] [Train Acc 17%, Loss 4.493691]
last iteration: 3.98335 seconds
elapsed time: 31.55562 seconds

Iter 900. [Val Acc 17%] [Train Acc 17%, Loss 4.502291]
last iteration: 3.62132 seconds
elapsed time: 35.17694 seconds

Iter 1000. [Val Acc 17%] [Train Acc 17%, Loss 4.398840]
last iteration: 4.05615 seconds
elapsed time: 39.23309 seconds

Iter 1100. [Val Acc 17%] [Train Acc 17%, Loss 4.543645]
last iteration: 4.22869 seconds
elapsed time: 43.46178 seconds

Iter 1200. [Val Acc 17%] [Train Acc 17%, Loss 4.439559]
last iteration: 4.40522 seconds
elapsed time: 47.867 seconds

Iter 1300. [Val Acc 17%] [Train Acc 17%, Loss 4.513348]
last iteration: 4.18082 seconds
elapsed time: 52.04782 seconds

Iter 1400. [Val Acc 17%] [Train Acc 17%, Loss 4.684847]
last iteration: 3.69911 seconds
elapsed time: 55.74693 seconds

Iter 1500. [Val Acc 17%] [Train Acc 17%, Loss 4.575275]
last iteration: 3.77191 seconds
elapsed time: 59.51885 seconds

Iter 1600. [Val Acc 17%] [Train Acc 17%, Loss 4.544442]
last iteration: 3.63628 seconds
elapsed time: 63.15512 seconds

Iter 1700. [Val Acc 17%] [Train Acc 17%, Loss 4.550700]
last iteration: 4.08607 seconds
elapsed time: 67.2412 seconds

Iter 1800. [Val Acc 17%] [Train Acc 17%, Loss 4.511775]
last iteration: 3.57943 seconds
elapsed time: 70.82062 seconds

Iter 1900. [Val Acc 17%] [Train Acc 17%, Loss 4.647605]
last iteration: 3.5914 seconds
elapsed time: 74.41202 seconds

Iter 2000. [Val Acc 17%] [Train Acc 17%, Loss 4.537618]
last iteration: 3.57344 seconds
elapsed time: 77.98547 seconds

```
elapsed time: 77.90547 seconds

Iter 2100. [Val Acc 17%] [Train Acc 17%, Loss 4.451121]
last iteration: 3.6293 seconds
elapsed time: 81.61476 seconds

Iter 2200. [Val Acc 17%] [Train Acc 17%, Loss 4.405941]
last iteration: 3.60636 seconds
elapsed time: 85.22112 seconds

Iter 2300. [Val Acc 17%] [Train Acc 17%, Loss 4.477833]
last iteration: 3.51261 seconds
elapsed time: 88.73373 seconds

Iter 2400. [Val Acc 17%] [Train Acc 17%, Loss 4.314979]
last iteration: 3.61334 seconds
elapsed time: 92.34706 seconds

Iter 2500. [Val Acc 17%] [Train Acc 17%, Loss 4.661035]
last iteration: 3.61334 seconds
elapsed time: 95.9604 seconds

Iter 2600. [Val Acc 17%] [Train Acc 17%, Loss 4.670698]
last iteration: 3.58441 seconds
elapsed time: 99.54482 seconds

Iter 2700. [Val Acc 17%] [Train Acc 17%, Loss 4.461138]
last iteration: 3.71806 seconds
elapsed time: 103.26288 seconds

Iter 2800. [Val Acc 17%] [Train Acc 17%, Loss 4.227867]
last iteration: 3.51561 seconds
elapsed time: 106.77849 seconds

Iter 2900. [Val Acc 17%] [Train Acc 17%, Loss 4.190715]
last iteration: 3.52955 seconds
elapsed time: 110.30804 seconds

Iter 3000. [Val Acc 17%] [Train Acc 17%, Loss 4.316841]
last iteration: 3.61334 seconds
elapsed time: 113.92138 seconds

Iter 3100. [Val Acc 17%] [Train Acc 17%, Loss 4.325480]
last iteration: 3.55948 seconds
elapsed time: 117.48086 seconds

Iter 3200. [Val Acc 17%] [Train Acc 17%, Loss 4.546013]
last iteration: 3.95742 seconds
elapsed time: 121.43828 seconds

Iter 3300. [Val Acc 17%] [Train Acc 17%, Loss 4.508635]
last iteration: 4.12796 seconds
elapsed time: 125.56624 seconds

Iter 3400. [Val Acc 17%] [Train Acc 17%, Loss 4.306953]
last iteration: 3.8507 seconds
elapsed time: 129.41694 seconds

Iter 3500. [Val Acc 17%] [Train Acc 17%, Loss 4.471777]
last iteration: 3.9654 seconds
elapsed time: 133.38234 seconds

Iter 3600. [Val Acc 17%] [Train Acc 17%, Loss 4.237211]
last iteration: 3.70011 seconds
elapsed time: 137.08245 seconds

Iter 3700. [Val Acc 17%] [Train Acc 17%, Loss 4.392957]
last iteration: 4.0392 seconds
elapsed time: 141.12165 seconds

Iter 3800. [Val Acc 17%] [Train Acc 17%, Loss 4.390327]
last iteration: 3.4049 seconds
elapsed time: 144.52654 seconds

Iter 3900. [Val Acc 17%] [Train Acc 17%, Loss 4.605230]
last iteration: 3.41387 seconds
elapsed time: 147.94041 seconds
```

```
Iter 4000. [Val Acc 17%] [Train Acc 17%, Loss 4.524452]
last iteration: 3.47272 seconds
elapsed time: 151.41313 seconds

Iter 4100. [Val Acc 17%] [Train Acc 17%, Loss 4.447223]
last iteration: 3.64825 seconds
elapsed time: 155.06137 seconds

Iter 4200. [Val Acc 17%] [Train Acc 17%, Loss 4.523162]
last iteration: 3.51859 seconds
elapsed time: 158.57996 seconds

Iter 4300. [Val Acc 17%] [Train Acc 17%, Loss 4.749586]
last iteration: 3.45177 seconds
elapsed time: 162.03173 seconds

Iter 4400. [Val Acc 17%] [Train Acc 17%, Loss 4.568340]
last iteration: 4.01925 seconds
elapsed time: 166.05099 seconds

Iter 4500. [Val Acc 17%] [Train Acc 17%, Loss 4.400958]
last iteration: 4.04917 seconds
elapsed time: 170.10016 seconds

Iter 4600. [Val Acc 17%] [Train Acc 17%, Loss 4.418864]
last iteration: 4.05467 seconds
elapsed time: 174.15483 seconds

Iter 4700. [Val Acc 17%] [Train Acc 17%, Loss 4.327309]
last iteration: 4.01825 seconds
elapsed time: 178.17308 seconds

Iter 4800. [Val Acc 17%] [Train Acc 17%, Loss 4.146078]
last iteration: 3.69811 seconds
elapsed time: 181.87119 seconds

Iter 4900. [Val Acc 17%] [Train Acc 17%, Loss 4.433993]
last iteration: 3.50263 seconds
elapsed time: 185.37383 seconds

Iter 5000. [Val Acc 17%] [Train Acc 17%, Loss 4.685889]
last iteration: 3.68216 seconds
elapsed time: 189.05599 seconds
```

Out[40]:

```
'\nonce again, no convergence for the same reason as above\n'
```

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 ...

In [42]:

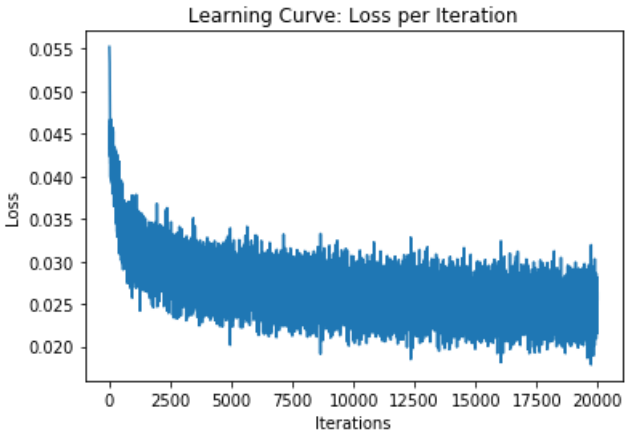
```
class PyTorchWordEmb(nn.Module):
    def __init__(self, emb_size=100, num_hidden=300, vocab_size=250):
        super(PyTorchWordEmb, self).__init__()
        self.word_emb_layer = nn.Linear(vocab_size, emb_size, bias=False)
        self.fc_layer1 = nn.Linear(emb_size * 3, num_hidden)
        self.fc_layer2 = 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)
```

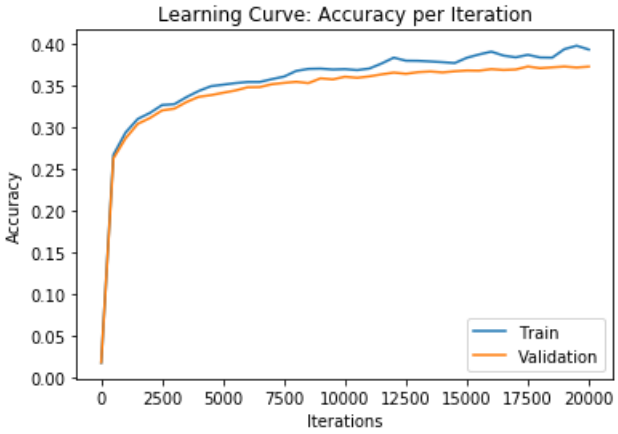
```
pytorch_wordemb= PyTorchWordEmb()

result = run_pytorch_gradient_descent(pytorch_wordemb, max_iters=20000, checkpoint_path='ckpt2-{}.pk')

pytorch_wordemb.load_state_dict(torch.load('ckpt2-20000.pk'))

plot_learning_curve(*result)
```

```
Iter 0. [Val Acc 2%] [Train Acc 2%, Loss 5.522576]
last iteration: 2.09938 seconds
elapsed time: 2.09938 seconds

Iter 500. [Val Acc 26%] [Train Acc 27%, Loss 3.360476]
last iteration: 4.92583 seconds
elapsed time: 7.02521 seconds

Iter 1000. [Val Acc 29%] [Train Acc 29%, Loss 3.107778]
last iteration: 5.02955 seconds
elapsed time: 12.05476 seconds

Iter 1500. [Val Acc 30%] [Train Acc 31%, Loss 3.029926]
last iteration: 5.235 seconds
elapsed time: 17.28977 seconds

Iter 2000. [Val Acc 31%] [Train Acc 32%, Loss 2.781154]
last iteration: 5.5741 seconds
elapsed time: 22.86386 seconds

Iter 2500. [Val Acc 32%] [Train Acc 33%, Loss 2.989281]
last iteration: 6.34603 seconds
elapsed time: 29.20989 seconds

Iter 3000. [Val Acc 32%] [Train Acc 33%, Loss 2.462952]
last iteration: 5.89324 seconds
elapsed time: 35.10313 seconds

Iter 3500. [Val Acc 33%] [Train Acc 34%, Loss 2.707418]
last iteration: 5.82244 seconds
elapsed time: 40.92558 seconds

Iter 4000. [Val Acc 34%] [Train Acc 34%, Loss 2.844786]
last iteration: 5.83937 seconds
elapsed time: 46.76495 seconds

Iter 4500. [Val Acc 34%] [Train Acc 35%, Loss 2.623051]
last iteration: 6.33207 seconds
elapsed time: 53.09702 seconds

Iter 5000. [Val Acc 34%] [Train Acc 35%, Loss 2.571400]
last iteration: 5.87529 seconds
elapsed time: 58.97231 seconds

Iter 5500. [Val Acc 34%] [Train Acc 35%, Loss 2.280872]
last iteration: 5.90421 seconds
elapsed time: 64.87652 seconds

Iter 6000. [Val Acc 35%] [Train Acc 35%, Loss 2.976673]
last iteration: 5.50029 seconds
elapsed time: 70.37681 seconds

Iter 6500. [Val Acc 35%] [Train Acc 35%, Loss 2.758784]
last iteration: 5.84066 seconds
elapsed time: 76.21747 seconds

Iter 7000. [Val Acc 35%] [Train Acc 36%, Loss 2.890918]
last iteration: 5.48533 seconds
elapsed time: 81.7028 seconds

Iter 7500. [Val Acc 35%] [Train Acc 36%, Loss 2.665479]
last iteration: 5.53521 seconds
elapsed time: 87.23801 seconds

Iter 8000. [Val Acc 36%] [Train Acc 37%, Loss 2.481605]
last iteration: 5.44144 seconds
elapsed time: 92.67945 seconds
```

```
elapsed time: 92.07945 seconds

Iter 8500. [Val Acc 35%] [Train Acc 37%, Loss 2.350145]
last iteration: 5.55516 seconds
elapsed time: 98.23461 seconds

Iter 9000. [Val Acc 36%] [Train Acc 37%, Loss 2.752564]
last iteration: 5.4973 seconds
elapsed time: 103.73191 seconds

Iter 9500. [Val Acc 36%] [Train Acc 37%, Loss 2.653728]
last iteration: 5.43347 seconds
elapsed time: 109.16538 seconds

Iter 10000. [Val Acc 36%] [Train Acc 37%, Loss 2.672191]
last iteration: 5.42749 seconds
elapsed time: 114.59287 seconds

Iter 10500. [Val Acc 36%] [Train Acc 37%, Loss 2.536632]
last iteration: 5.38361 seconds
elapsed time: 119.97648 seconds

Iter 11000. [Val Acc 36%] [Train Acc 37%, Loss 2.717220]
last iteration: 5.44943 seconds
elapsed time: 125.4259 seconds

Iter 11500. [Val Acc 36%] [Train Acc 38%, Loss 2.894140]
last iteration: 5.39956 seconds
elapsed time: 130.82546 seconds

Iter 12000. [Val Acc 37%] [Train Acc 38%, Loss 2.476303]
last iteration: 5.43646 seconds
elapsed time: 136.26193 seconds

Iter 12500. [Val Acc 36%] [Train Acc 38%, Loss 2.709149]
last iteration: 5.53919 seconds
elapsed time: 141.80112 seconds

Iter 13000. [Val Acc 37%] [Train Acc 38%, Loss 2.529503]
last iteration: 5.35468 seconds
elapsed time: 147.1558 seconds

Iter 13500. [Val Acc 37%] [Train Acc 38%, Loss 2.355676]
last iteration: 5.34072 seconds
elapsed time: 152.49652 seconds

Iter 14000. [Val Acc 37%] [Train Acc 38%, Loss 2.402967]
last iteration: 5.42848 seconds
elapsed time: 157.925 seconds

Iter 14500. [Val Acc 37%] [Train Acc 38%, Loss 2.453164]
last iteration: 5.88427 seconds
elapsed time: 163.80927 seconds

Iter 15000. [Val Acc 37%] [Train Acc 38%, Loss 2.704140]
last iteration: 5.7975 seconds
elapsed time: 169.60677 seconds

Iter 15500. [Val Acc 37%] [Train Acc 39%, Loss 2.442965]
last iteration: 5.38859 seconds
elapsed time: 174.99536 seconds

Iter 16000. [Val Acc 37%] [Train Acc 39%, Loss 2.548620]
last iteration: 5.33374 seconds
elapsed time: 180.32909 seconds

Iter 16500. [Val Acc 37%] [Train Acc 39%, Loss 2.112705]
last iteration: 5.40754 seconds
elapsed time: 185.73664 seconds

Iter 17000. [Val Acc 37%] [Train Acc 38%, Loss 2.801236]
last iteration: 5.43846 seconds
elapsed time: 191.17509 seconds

Iter 17500. [Val Acc 37%] [Train Acc 39%, Loss 2.853754]
last iteration: 5.99098 seconds
elapsed time: 197.16607 seconds
```

```
Iter 18000. [Val Acc 37%] [Train Acc 38%, Loss 2.150152]
last iteration: 5.61399 seconds
elapsed time: 202.78006 seconds

Iter 18500. [Val Acc 37%] [Train Acc 38%, Loss 2.375998]
last iteration: 5.57908 seconds
elapsed time: 208.35914 seconds

Iter 19000. [Val Acc 37%] [Train Acc 39%, Loss 2.333347]
last iteration: 5.3487 seconds
elapsed time: 213.70784 seconds

Iter 19500. [Val Acc 37%] [Train Acc 40%, Loss 2.594208]
last iteration: 5.44843 seconds
elapsed time: 219.15627 seconds

Iter 20000. [Val Acc 37%] [Train Acc 39%, Loss 2.562430]
last iteration: 5.44743 seconds
elapsed time: 224.60371 seconds
```





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%.

In [47]:

```
print("You are a "+make_prediction_torch(pytorch_wordemb, ["you", "are", "a"]))
print("few companies show "+make_prediction_torch(pytorch_wordemb, ["few", "companies", "show"]))
print("There are no "+make_prediction_torch(pytorch_wordemb, ["there", "are", "no"]))
print("yesterday i was "+make_prediction_torch(pytorch_wordemb, ["yesterday", "i", "was"]))
print("the game had "+make_prediction_torch(pytorch_wordemb, ["the", "game", "had"]))
print("yesterday the federal "+make_prediction_torch(pytorch_wordemb, ["yesterday", "the",
"federal"]))
```

```
You are a good
few companies show .
There are no place
yesterday i was nt
the game had to
yesterday the federal government
```

Part (f) -- 1 pts

Report the test accuracy of your model

In [48]:

```
print("[Test acc: "+str(round(estimate_accuracy_torch(pytorch_wordemb, test4grams)*100, 3))+"%]")
```

```
[Test acc: 37.483%]
```

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.

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".

In [49]:

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

'''
the purpose of the word_emb_weigths matrix is to reduce the dimensionality of the
vocabulary of the input. In this particular project, we converted a 250 word vocabulary
to a 100 word vocabulary, making the word_emb_weights matrix 100x250. The matrix has
the function of transforming a higher dimensionality vocabulary into a lower dimensionality
one, and it achieves this by "grouping" words together; taking a word and transforming it
into an equivalent word in the lower dimensional space, hence each row corresponds to a
word in higher dimensional space, and each column corresponds to a word in lower dimensional
space.
'''
```

Out[49]:

```
'\nthe purpose of the word_emb_weigths matrix is to reduce the dimensionality of the\nvocabulary o
f the input. In this particular project, we converted a 250 word vocabulary\nto a 100 word vocabul
ary, making the word_emb_weights matrix 100x250. The matrix has\nthe function of transforming a hi
gher dimensionality vocabulary into a lower dimensionality\none, and it achieves this by
"grouping" words together; taking a word and transforming it\ninto an equivalent word in the lower
dimensional space, hence each row corresponds to a\nword in higher dimensional space, and each col
umn corresponds to a word in lower dimensional\nspace.\n'
```

Part (b) -- 2 pts

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.

In [50]:

```
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']])
```

```
0.3310949
0.08890724
```

Compute the 5 closest words to the following words:

- "four"
- "go"
- "what"
- "should"
- "school"
- "your"
- "yesterday"
- "not"

In [51]:

```
maxes = []
for word in ["four", "go", "what", "should", "school", "your", "yesterday", "not"]:
    l = similarities[vocab_stoi[word], :]
    maxes.append([])
    for i in range(5):
        big = None
        for proximal in range(len(l)):
            if big == None or (l[proximal] > l[big] and proximal not in maxes[-1]):
                big = proximal
        maxes[-1].append(big)
close_words = []
for i in maxes:
    close_words.append([])
    for word in i:
        close_words[-1].append(vocab_itos[word])

examples = ["four", "go", "what", "should", "school", "your", "yesterday", "not"]

for word in range(len(examples)):
    print("'"+examples[word]+"' has the five closest words: "+str(close_words[word]))
```

```
'four' has the five closest words: ['four', 'two', 'million', 'three', 'few']
'go' has the five closest words: ['go', 'percent', 'come', 'going', 'back']
'what' has the five closest words: ['what', 'where', 'when', 'how', 'ms.']
'should' has the five closest words: ['should', 'could', 'may', 'can', 'would']
'school' has the five closest words: ['school', 'day', 'department', 'president', 'ago']
'your' has the five closest words: ['your', 'my', 'mr.', 'united', 'american']
'yesterday' has the five closest words: ['yesterday', 'today', 'director', 'said', 'family']
'not' has the five closest words: ['not', 'nt', 'never', 'been', 'end']
```

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.

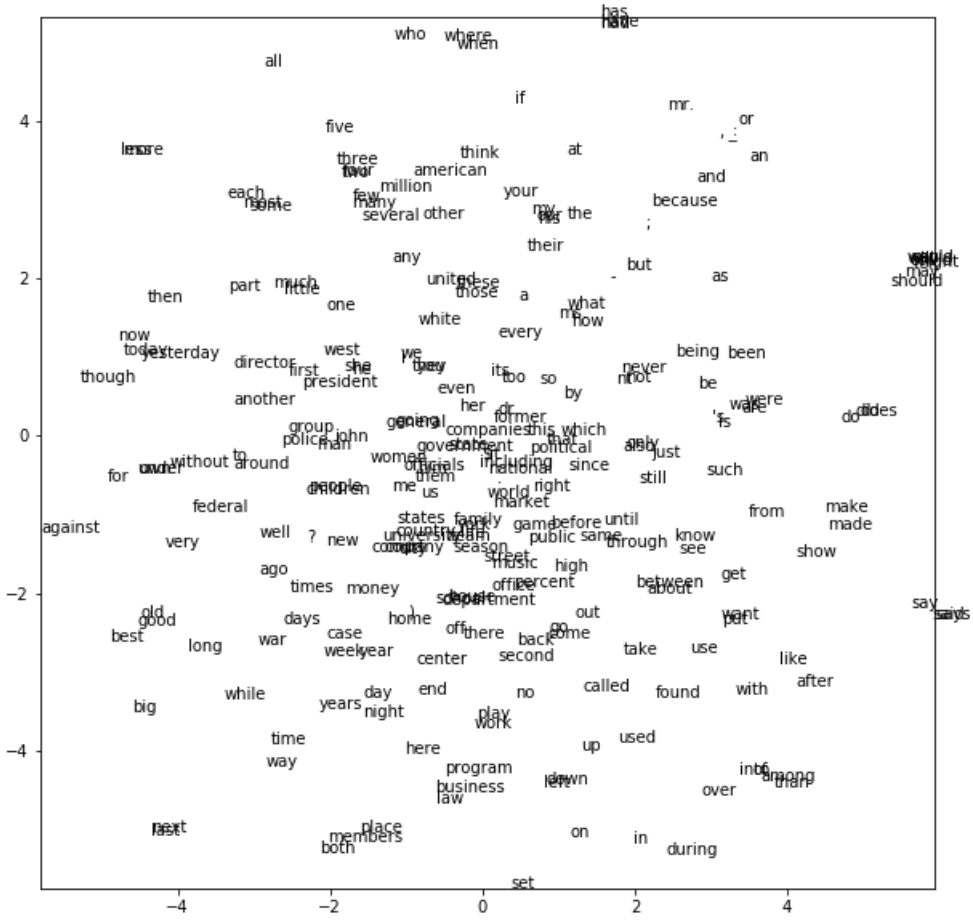
The following code runs the t-SNE algorithm and plots the result. Look at the plot and find two clusters of related words. What do the

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.

In [55]:

```
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()

'''
What we noticed is that not only are words that have a similar letter composition placed close
together, but also words that are used in similar contexts are close together. We figure that
this could be because the similarity of sentences that each word is used in. The two clusters
we observed are the "government" cluster in the lower-center, and the "time" cluster farther to th
e bottom left.
The "government" cluster has lots of words that have an "ent" in the word such as government, perc
ent,
department, and center, and the "time" cluster has words that relate to the concept of time such a
s day,
night,week, year, days, yesterday, and time.
'''
```



Out[55]:

```
'\nWhat we noticed is that not only are words that have a similar letter composition placed close\
ntogether, but also words that are used in similar contexts are close together. We figure
that\nthis could be because the similarity of sentences that each word is used in. The two
clusters\nwe observed are the "government" cluster in the center, and the "time" cluster farther t
o the left.\nThe "government" cluster has lots of words that have an "ent" in the word such as
government, percent,\ndepartment, and center, and the "time" cluster has words that relate to the
concept of time such as day,\nnight,week, year, days, yesterday, and time.\n'
```