Assignment 2: Word Prediction

Deadline: Sunday, December 11th, by 8pm.

Submission: Submit a PDF export of the completed notebook as well as the ipynb file.

In this assignment, we will make a neural network that can predict the next word in a sentence given the previous three.
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 you properly explain what you are doing and why.

In [ ]:

```
import pandas
import numpy as np
import matplotlib.pyplot as plt
import collections

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

Question 1. Data (18%)

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 the course page on Moodle and upload it to Google Drive. Then, mount Google Drive from your Google Colab notebook:

In [ ]:

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

```
Mounted at /content/gdrive
```

Find the path to `raw_sentences.txt`:

In [ ]:

```
file_path = '/content/gdrive/MyDrive/Colab Notebooks/raw_sentences.txt' # TODO - UPDATE M
E!
```

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 [ ]:

```
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 [ ]:

```
vocab = set([w for s in sentences for w in s]) #each distrinct word appears once only
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 [ ]:

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

Part (a) -- 3%

Display 10 sentences in the training set. Explain how punctuations are treated in our word representation, and how words with apostrophes are represented.

In [ ]:

```
train[20:30]
```

Out[ ]:

```
[['we', 'are', 'a', 'good', 'team', '.'],
 ['i', 'did', 'nt', 'know', 'about', 'it', ',', 'she', 'said', '.'],
 ['i', 'do', 'nt', 'think', 'we', 'are', 'going', 'to', 'make', 'that', '.'],
 ['we', 'did', 'nt', 'want', 'to', 'make', 'one', 'of', 'those', '.'],
 ['that', "'s", 'what', "'s", 'good', 'about', 'it', '.'],
 ['now', 'there', 'are', 'so', 'many', '.'],
 ['then', 'there', "'s", 'the', 'music', '.'],
 ['who', 'is', 'the', 'we', '?'],
 ['but', 'i', 'want', 'it', 'to', 'be', 'the', 'same', ',', 'i', 'said', '.'],
 ['we', 'never', 'left', 'the', 'house', '.']]
```

The punctuations are treated as individual words, as we can the, the comas and the points in every sentence are are seperated from the other words.

However, the apostrophs are not "stand-alone" words, actually they split each word into to parts. In most of the times, the first part of the word is a legal word, and the second word is an abbreviation, for example: "That's what's good about it." turns into " 'that', "'s", 'what', "'s", 'good', 'about', 'it', '.'"

Part (b) -- 4%

Print the 10 most common words in the vocabulary and how often does each of these words appear in the training sentences. Express the second quantity as a percentage (i.e. number of occurences of the word / total number of words in the training set).

These are useful 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 [ ]:

```
# Your code goes here
vocab = [w for s in train for w in s]
most_common = collections.Counter(vocab).most_common(10)
#print(most_common)

total_words = len(vocab)
for tup in most_common:
  percent = (tup[1]/total_words)
  print("The word   "+str(tup[0])+"   its percentage "+str(round(percent*100,2))+"%")
```

```
The word   .   its percentage 10.7%
The word   it   its percentage 3.85%
The word   ,   its percentage 3.25%
The word   i   its percentage 2.94%
The word   do   its percentage 2.69%
The word   to   its percentage 2.58%
The word   nt   its percentage 2.16%
The word   ?   its percentage 2.14%
The word   the   its percentage 2.09%
The word   's   its percentage 2.09%
```

Part (c) -- 11%

Our neural network will take as input three words and predict the next one. Therefore, we need our data set to be comprised of seuqnces of four consecutive words in a sentence, referred to as *4grams*.

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, where N is the number of 4grams (sequences of 4 words appearing one after the other) that can be found in the complete list of sentences. Examples of how these functions should operate are detailed in the code below.

You can use the defined `vocab`, `vocab_itos`, and `vocab_stoi` in your code.

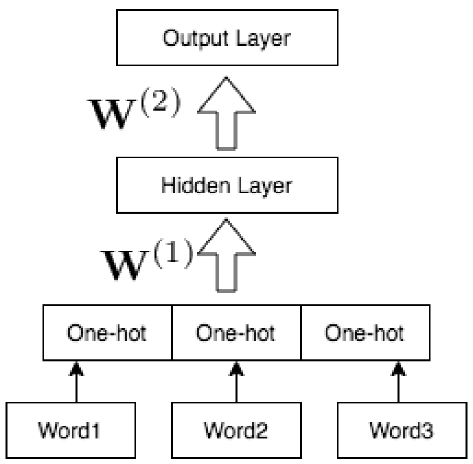
In [ ]:

```
# 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]]
    """

    lst = []
    for sent in sents:
      sublst = []
      for word in sent:
        sublst.append(vocab_stoi[word])
      lst.append(sublst)

    return lst

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 seq in seqs:
      l=len(seq);
      for i in range(l-3):
        grams.append(seq[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)

# We can now generate our data which will be used to train and test the network


train4grams = process_data(train)
valid4grams = process_data(valid)
test4grams = process_data(test)

# Check:

# 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]])
```

Question 2. A Multi-Layer Perceptron (44%)

In this section, we will build a two-layer multi-layer perceptron. Our model will look like this:



Since the sentences in the data are comprised of 250 distinct words, our task boils down to claissfication where the label space $\mathcal{S}$ is of cardinality $|\mathcal{S}| = 250$ while our input, which is comprised of a combination of three words, is treated as a vector of size 750×1 (i.e., the concatanation of three one-hot 250×1 vectors).

The following function `get_batch` will take as input the whole dataset and output a single batch for the training. The output size of the batch is explained below.

Implement yourself a function `make_onehot` which takes the data in index notation and output it in a onehot notation.

Start by reviewing the helper function, which is given to you:

In [ ]:

```
def make_onehot(data):
    """
    Convert one batch of data in the index notation into its corresponding onehot
    notation. Remember, the function should work for both xt and st.
```

```
    xt->one-hot vectors of size 3 each
    st ->true/false one-hot

    input - vector with shape D (1D or 2D)
    output - vector with shape (D,250)

    creating an indicator
    """

    data = np.array(data)
    D=np.shape(data)

    if (len(D)==2): #2 D data input
      out = np.zeros([D[0],D[1],250])
      for i in range(D[0]):
       for j in range(D[1]):
          out[i][j][data[i][j]] = 1

    else:  #1 D data input
      out = np.zeros([D[0],250])
      for i in range(D[0]):
        out[i][data[i]] = 1

    return out.astype(np.float32)

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 (xt, st) where:
     - `xt` is an numpy array of one-hot vectors of shape [batch_size, 3, 250]
     - `st` 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
    """
    xt = data[range_min:range_max, :3]
    xt = make_onehot(xt)
    st = data[range_min:range_max, 3]
    if onehot:
        st = make_onehot(st).reshape(-1, 250)
    return xt, st
```

Part (a) -- 8%

We build the 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 [ ]:

```
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]) #the -1 dimension is calculated by python
        # TODO: complete this function
        # Note that we will be using the nn.CrossEntropyLoss(), which computes the softma
x operation internally, as loss criterion
        # layer one
        out = self.layer1(inp) # fully-connected layer
```

```
        #  relu = self.relu(out) # activation function ReLU

         # layer two
         out = self.layer2(out)

         return out
```

Part (b) -- 10%

We next train the PyTorch model using the Adam optimizer and the cross entropy loss.

Complete the function `run_pytorch_gradient_descent`, and use it to train your PyTorch MLP model.

Obtain a training accuracy of at least 35% while changing only the hyperparameters of the train function.

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

In [ ]:

```
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
        xt, st = get_batch(data, i, i + batch_size, onehot=False)

        # forward pass prediction
        y = model(torch.Tensor(xt))
        y = y.detach().numpy() # convert the PyTorch tensor => numpy array
        pred = np.argmax(y, axis=1)
        correct += np.sum(pred == st)
        N += st.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`    (epochs)
    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/Intro_to_Deep_Learning/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 Intro_to_Deep_Learning, mlp, etc...). Your Google Drive will be populated
```

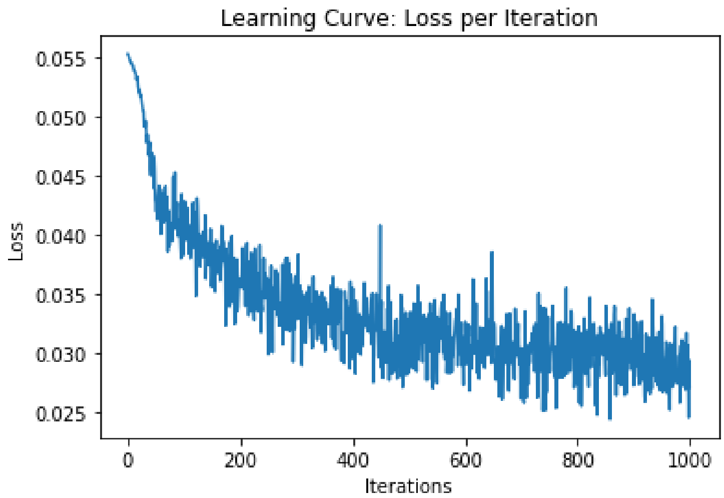
```
with files:

    - /content/gdrive/My Drive/Intro_to_Deep_Learning/mlp/ckpt-500.pk
    - /content/gdrive/My Drive/Intro_to_Deep_Learning/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/Intro_to_Deep_Learning
/mlp/ckpt-500.pk'))

    This function returns the training loss, and the training/validation accuracy,
    which we can use to plot the learning curve.
    """
    model.train() # switch the module mode to .train() so that new weights can be learned
after every epoch
    #Train PyTorch model on the dataset `train_data`
    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):  #batch_size is the stepValu
e
            if (i + batch_size) > train_data.shape[0]:
                break

            # get the input and targets of a minibatch
            xt, st = get_batch(train_data, i, i + batch_size, onehot=False)  # for oneho
t False the st is of shape [batch_size]
                                                                              # xt is an
array of one-hot vectors of shape [batch_size, 3, 250]

            # convert from numpy arrays to PyTorch tensors
            xt = torch.Tensor(xt)
            st = torch.Tensor(st).long()  #type int 64 bit

            optimizer.zero_grad()    # Clear gradients w.r.t. parameters

            zs = model(xt)           # Forward pass to get prediction
            loss = criterion(zs, st) # Calculate Loss: softmax --> cross entropy loss

            # Backward pass
            loss.backward()          # Getting gradients w.r.t. parameters
            optimizer.step()         # Updating parameters


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

            if n % 100 == 0:                       #changed from 500->makes plot more ac
curate
                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
```

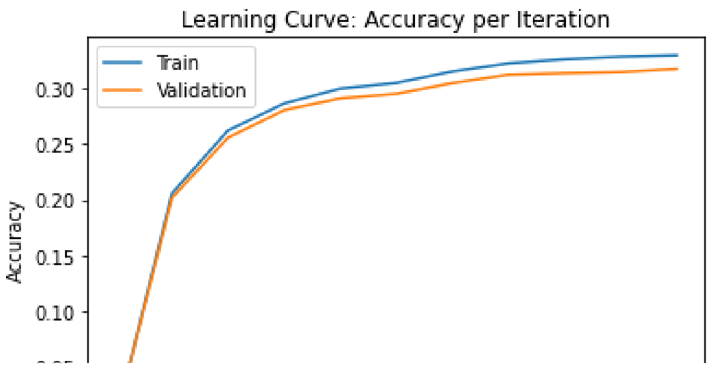
```
            if n > max_iters:
                return iters, losses, iters_sub, train_accs, val_accs


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()
```

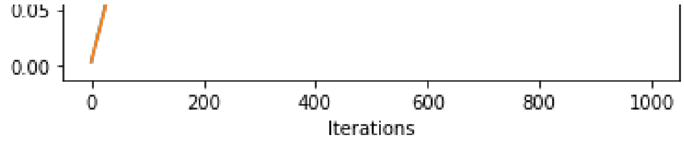
In [ ]:

```
pytorch_mlp = PyTorchMLP()
learning_curve_info = run_pytorch_gradient_descent(pytorch_mlp)


plot_learning_curve(*learning_curve_info)
```

```
Iter 0. [Val Acc 0%] [Train Acc 0%, Loss 5.527264]
Iter 100. [Val Acc 20%] [Train Acc 21%, Loss 4.052488]
Iter 200. [Val Acc 26%] [Train Acc 26%, Loss 3.618165]
Iter 300. [Val Acc 28%] [Train Acc 29%, Loss 3.513102]
Iter 400. [Val Acc 29%] [Train Acc 30%, Loss 3.547521]
Iter 500. [Val Acc 29%] [Train Acc 30%, Loss 2.889133]
Iter 600. [Val Acc 30%] [Train Acc 31%, Loss 3.265597]
Iter 700. [Val Acc 31%] [Train Acc 32%, Loss 3.040995]
Iter 800. [Val Acc 31%] [Train Acc 33%, Loss 3.010946]
Iter 900. [Val Acc 31%] [Train Acc 33%, Loss 2.723145]
Iter 1000. [Val Acc 32%] [Train Acc 33%, Loss 2.712780]
```







Part (c) -- 10%

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.

In [ ]:

```python
# 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()}

#print(convert_words_to_indices([['one', 'in', 'five', 'are', 'over', 'here'], ['other', 'one', 'since', 'yesterday'], ['you']]))
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
    model.eval()  # sets the PyTorch module to evaluation mode ->don’t want the model to learn new weights for this task

    input= []
    input = convert_words_to_indices([sentence])   # the given sentence in indexes of the vocab_itos
    input = input[0]
    input = make_onehot(input)
    input = torch.Tensor(input)
    out = model.forward(input)
    out= out.detach().numpy() # convert the PyTorch tensor => numpy array
    pred = np.argmax(out, axis=1)
   # print(pred[0])
   # print(vocab_itos[81])
    return vocab_itos[pred[0]]


make_prediction_torch(pytorch_mlp, ['you', 'are', 'a'])
```

Out[ ]:

```
'good'
```

Part (d) -- 10%

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"**

yesterday the federal

Do your predictions make sense?

In many cases where you overfit the model can either output the same results for all inputs or just memorize the dataset.

Print the output for all of these sentences and Write below if you encounter these effects or something else which indicates overfitting, if you do train again with better hyperparameters.

In [ ]:

```
def to_predict(model):
  data_to_predict = ["You are a","few companies show","There are no","yesterday i was","the game had","yesterday the federal"]
  sentences = []
  for sen in data_to_predict:
    words = sen.split()
    sentence = [word.lower() for word in words]
    sentences.append(sentence)
  #print(sentences)

  for sen in sentences:
    print("The sentences predicted:")
    print(' '.join(sen))
    print("Our model predicted: "+str(make_prediction_torch(model, sen))+"\n")
  return

to_predict(pytorch_mlp)

#Train again and look if overfits or not

retrained = run_pytorch_gradient_descent(pytorch_mlp, train_data=train4grams, validation_data=valid4grams, batch_size=1000,learning_rate=0.001, weight_decay=0, max_iters=500, checkpoint_path=None)
#Iter 500. [Val Acc 34%] [Train Acc 36%, Loss 2.545604] ->good predictions
print("\nPrediction after training again:\n")
to_predict(pytorch_mlp)

#retrained = run_pytorch_gradient_descent(pytorch_mlp, train_data=train4grams, validation_data=valid4grams, batch_size=1000,learning_rate=0.08, weight_decay=0, max_iters=700, checkpoint_path=None)
#Iter 700. [Val Acc 22%] [Train Acc 22%, Loss 4.359359] ->bad predictions
#print("\nPrediction after training again:\n")
#to_predict(pytorch_mlp)

#retrained = run_pytorch_gradient_descent(pytorch_mlp, train_data=train4grams, validation_data=valid4grams, batch_size=700,learning_rate=0.009, weight_decay=0, max_iters=700, checkpoint_path=None)
#Iter 700. [Val Acc 32%] [Train Acc 35%, Loss 2.856694] ->overfitting
#print("\nPrediction after training again:\n")
#to_predict(pytorch_mlp)
```

```
The sentences predicted:
you are a
Our model predicted: good

The sentences predicted:
few companies show
Our model predicted: .

The sentences predicted:
there are no
Our model predicted: other

The sentences predicted:
yesterday i was
Our model predicted: nt

The sentences predicted:
the game had
Our model predicted: to
```

```
The sentences predicted:
yesterday the federal
Our model predicted: way

Iter 0. [Val Acc 32%] [Train Acc 33%, Loss 2.760777]
Iter 100. [Val Acc 33%] [Train Acc 34%, Loss 2.884136]
Iter 200. [Val Acc 33%] [Train Acc 34%, Loss 2.789805]
Iter 300. [Val Acc 33%] [Train Acc 35%, Loss 2.796031]
Iter 400. [Val Acc 34%] [Train Acc 35%, Loss 2.699456]
Iter 500. [Val Acc 34%] [Train Acc 35%, Loss 2.609669]

Prediction after training again:

The sentences predicted:
you are a
Our model predicted: good

The sentences predicted:
few companies show
Our model predicted: .

The sentences predicted:
there are no
Our model predicted: other

The sentences predicted:
yesterday i was
Our model predicted: nt

The sentences predicted:
the game had
Our model predicted: the

The sentences predicted:
yesterday the federal
Our model predicted: states
```

Most of the predictions above do make sense. The model's predictions before retraining were for example to the sentence "yesterday the federal" was "way", which probably is not right word to follow as well as to the sentence "few companies show" which is followed by a fullstop. After retraining the model, some of these predictions were improved.

Even though 'period' is the most common "word" in our dataset, it was only predicted once in the prediction above. If we would gotten it several times that would indicate an underfit - this result is bringing our model to a local minimum.

In the following code frame, similar inputs for sentences were done in order to check if our model does overfit:

In [97]:

```
def to_predict(model):
  data_to_predict = ["now there are", "are so many"]
  sentences = []
  for sen in data_to_predict:
    words = sen.split()
    sentence = [word.lower() for word in words]
    sentences.append(sentence)
  #print(sentences)

  for sen in sentences:
    print("The sentences predicted:")
    print(' '.join(sen))
    print("Our model predicted: "+str(make_prediction_torch(model, sen))+"\n")
  return

to_predict(pytorch_mlp)
```

```
The sentences predicted:
now there are
```

```
now there are
Our model predicted: two

The sentences predicted:
are so many
Our model predicted: people
```

The data was tocken from the sentence (of the given txt file): ['now', 'there', 'are', 'so', 'many', '.'],

One can see different results for each sentence compared to the original sentence and predictions that still make sense. This indicates that our model is not memorizing the data but learning the meaning of the previous words in the sentence. Therefore, there is no overfitting.

Part (e) -- 6%

Report the test accuracy of your model

In [ ]:

```python
# Write your code here
#plot_learning_curve(*retrained)

print("The train accuracy of the model is: ")

estimate_accuracy_torch(pytorch_mlp,train4grams)

# A training accuracy of at least 35% while changing only the hyperparameters of the trai
n function is required.
```

```
The train accuracy of the model is:
```

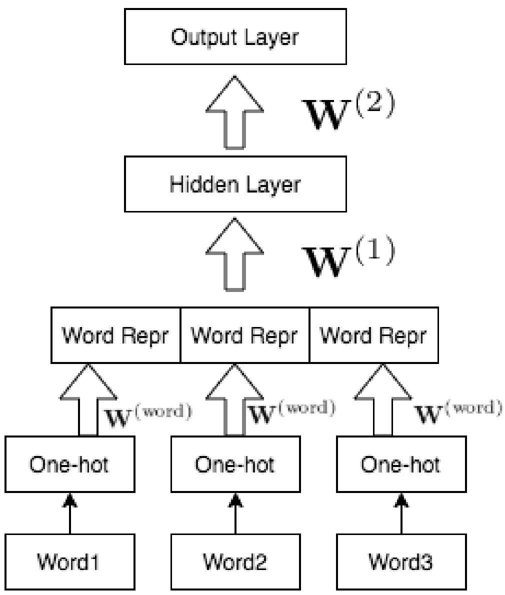
Out[ ]:

```
0.3525904761904762
```

Question 3. Learning Word Embeddings (24 %)

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%

Part (a) -- 10%

The PyTorch 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.

In [ ]:

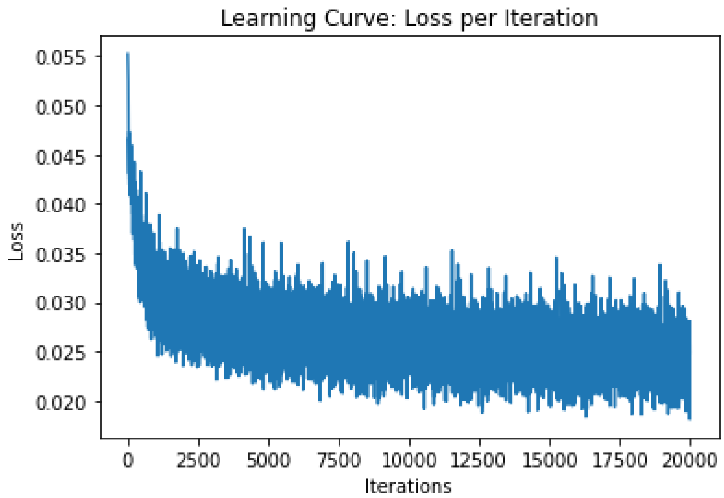
```
class PyTorchWordEmb(nn.Module):
    def __init__(self, emb_size=100, num_hidden=300, vocab_size=250):
        super(PyTorchWordEmb, self).__init__()
        self.word_emb_layer = nn.Linear(vocab_size, emb_size, bias=False)
        self.fc_layer1 = nn.Linear(emb_size * 3, num_hidden)
        self.fc_layer2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

plot_learning_curve(*result)



#A training accuracy of at least 38% is required.
```

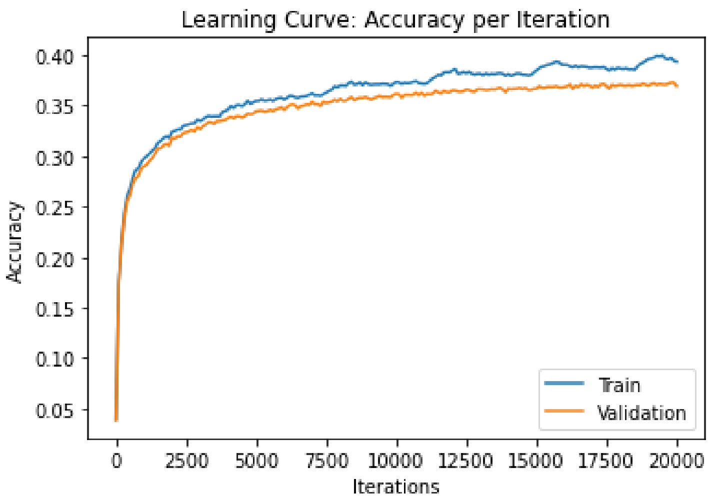
```
Iter 0. [Val Acc 4%] [Train Acc 4%, Loss 5.520952]
Iter 100. [Val Acc 17%] [Train Acc 17%, Loss 4.411149]
Iter 200. [Val Acc 21%] [Train Acc 21%, Loss 3.990214]
Iter 300. [Val Acc 24%] [Train Acc 24%, Loss 3.981677]
Iter 400. [Val Acc 26%] [Train Acc 26%, Loss 3.845290]
Iter 500. [Val Acc 26%] [Train Acc 27%, Loss 3.113897]
Iter 600. [Val Acc 27%] [Train Acc 28%, Loss 3.508665]
Iter 700. [Val Acc 28%] [Train Acc 29%, Loss 3.308282]
Iter 800. [Val Acc 28%] [Train Acc 29%, Loss 3.289096]
Iter 900. [Val Acc 29%] [Train Acc 29%, Loss 3.091829]
Iter 1000. [Val Acc 29%] [Train Acc 30%, Loss 2.889528]
Iter 1100. [Val Acc 29%] [Train Acc 30%, Loss 2.852162]
Iter 1200. [Val Acc 30%] [Train Acc 30%, Loss 2.930356]
Iter 1300. [Val Acc 30%] [Train Acc 31%, Loss 2.927501]
Iter 1400. [Val Acc 30%] [Train Acc 31%, Loss 2.936527]
Iter 1500. [Val Acc 31%] [Train Acc 31%, Loss 2.801054]
Iter 1600. [Val Acc 31%] [Train Acc 32%, Loss 2.569288]
Iter 1700. [Val Acc 31%] [Train Acc 32%, Loss 2.800225]
Iter 1800. [Val Acc 31%] [Train Acc 32%, Loss 2.743633]
Iter 1900. [Val Acc 31%] [Train Acc 32%, Loss 2.614609]
Iter 2000. [Val Acc 32%] [Train Acc 32%, Loss 2.747621]
Iter 2100. [Val Acc 32%] [Train Acc 32%, Loss 3.014339]
Iter 2200. [Val Acc 32%] [Train Acc 33%, Loss 2.807914]
Iter 2300. [Val Acc 32%] [Train Acc 33%, Loss 2.755364]
Iter 2400. [Val Acc 32%] [Train Acc 33%, Loss 2.858739]
Iter 2500. [Val Acc 32%] [Train Acc 33%, Loss 2.821852]
Iter 2600. [Val Acc 32%] [Train Acc 33%, Loss 3.254037]
Iter 2700. [Val Acc 33%] [Train Acc 33%, Loss 2.600566]
Iter 2800. [Val Acc 32%] [Train Acc 33%, Loss 2.987418]
Iter 2900. [Val Acc 33%] [Train Acc 34%, Loss 2.468778]
Iter 3000. [Val Acc 33%] [Train Acc 33%, Loss 2.579657]
Iter 3100. [Val Acc 33%] [Train Acc 34%, Loss 2.743947]
Iter 3200. [Val Acc 33%] [Train Acc 34%, Loss 2.572080]
Iter 3300. [Val Acc 33%] [Train Acc 34%, Loss 2.762425]
Iter 3400. [Val Acc 33%] [Train Acc 34%, Loss 3.028224]
Iter 3500. [Val Acc 33%] [Train Acc 34%, Loss 2.472183]
Iter 3600. [Val Acc 33%] [Train Acc 34%, Loss 2.614046]
Iter 3700. [Val Acc 33%] [Train Acc 34%, Loss 2.543582]
Iter 3800. [Val Acc 34%] [Train Acc 34%, Loss 2.699191]
Iter 3900. [Val Acc 34%] [Train Acc 34%, Loss 2.922577]
```

```
Iter 4000. [Val Acc 34%] [Train Acc 35%, Loss 2.708966]
Iter 4100. [Val Acc 34%] [Train Acc 35%, Loss 2.913502]
Iter 4200. [Val Acc 34%] [Train Acc 35%, Loss 2.658993]
Iter 4300. [Val Acc 34%] [Train Acc 35%, Loss 2.534877]
Iter 4400. [Val Acc 34%] [Train Acc 35%, Loss 2.311072]
Iter 4500. [Val Acc 34%] [Train Acc 35%, Loss 2.747543]
Iter 4600. [Val Acc 34%] [Train Acc 35%, Loss 2.777281]
Iter 4700. [Val Acc 34%] [Train Acc 35%, Loss 2.916097]
Iter 4800. [Val Acc 34%] [Train Acc 35%, Loss 2.659293]
Iter 4900. [Val Acc 34%] [Train Acc 35%, Loss 2.483544]
Iter 5000. [Val Acc 34%] [Train Acc 35%, Loss 2.785416]
Iter 5100. [Val Acc 34%] [Train Acc 35%, Loss 2.904554]
Iter 5200. [Val Acc 35%] [Train Acc 36%, Loss 2.597055]
Iter 5300. [Val Acc 34%] [Train Acc 35%, Loss 2.662821]
Iter 5400. [Val Acc 34%] [Train Acc 36%, Loss 2.579973]
Iter 5500. [Val Acc 35%] [Train Acc 36%, Loss 2.519674]
Iter 5600. [Val Acc 34%] [Train Acc 35%, Loss 2.553834]
Iter 5700. [Val Acc 35%] [Train Acc 36%, Loss 2.454363]
Iter 5800. [Val Acc 35%] [Train Acc 36%, Loss 2.634215]
Iter 5900. [Val Acc 35%] [Train Acc 36%, Loss 2.563962]
Iter 6000. [Val Acc 35%] [Train Acc 36%, Loss 2.754938]
Iter 6100. [Val Acc 35%] [Train Acc 36%, Loss 2.414279]
Iter 6200. [Val Acc 35%] [Train Acc 36%, Loss 2.745648]
Iter 6300. [Val Acc 35%] [Train Acc 36%, Loss 2.657359]
Iter 6400. [Val Acc 35%] [Train Acc 36%, Loss 2.724502]
Iter 6500. [Val Acc 35%] [Train Acc 36%, Loss 2.644387]
Iter 6600. [Val Acc 35%] [Train Acc 36%, Loss 2.845556]
Iter 6700. [Val Acc 35%] [Train Acc 36%, Loss 2.566372]
Iter 6800. [Val Acc 35%] [Train Acc 36%, Loss 2.918945]
Iter 6900. [Val Acc 35%] [Train Acc 36%, Loss 2.600557]
Iter 7000. [Val Acc 35%] [Train Acc 36%, Loss 2.933195]
Iter 7100. [Val Acc 35%] [Train Acc 36%, Loss 2.845326]
Iter 7200. [Val Acc 35%] [Train Acc 36%, Loss 2.553804]
Iter 7300. [Val Acc 35%] [Train Acc 36%, Loss 2.456339]
Iter 7400. [Val Acc 35%] [Train Acc 36%, Loss 2.560951]
Iter 7500. [Val Acc 35%] [Train Acc 36%, Loss 2.775098]
Iter 7600. [Val Acc 35%] [Train Acc 36%, Loss 2.593739]
Iter 7700. [Val Acc 35%] [Train Acc 37%, Loss 2.840216]
Iter 7800. [Val Acc 36%] [Train Acc 37%, Loss 2.446130]
Iter 7900. [Val Acc 35%] [Train Acc 37%, Loss 2.376009]
Iter 8000. [Val Acc 36%] [Train Acc 37%, Loss 2.209399]
Iter 8100. [Val Acc 35%] [Train Acc 37%, Loss 2.419271]
Iter 8200. [Val Acc 35%] [Train Acc 37%, Loss 2.566752]
Iter 8300. [Val Acc 36%] [Train Acc 37%, Loss 2.506143]
Iter 8400. [Val Acc 36%] [Train Acc 37%, Loss 2.659271]
Iter 8500. [Val Acc 36%] [Train Acc 37%, Loss 2.903915]
Iter 8600. [Val Acc 36%] [Train Acc 37%, Loss 2.655129]
Iter 8700. [Val Acc 36%] [Train Acc 37%, Loss 2.643939]
Iter 8800. [Val Acc 36%] [Train Acc 37%, Loss 2.676686]
Iter 8900. [Val Acc 36%] [Train Acc 37%, Loss 2.650839]
Iter 9000. [Val Acc 36%] [Train Acc 37%, Loss 2.399660]
Iter 9100. [Val Acc 36%] [Train Acc 37%, Loss 2.652247]
Iter 9200. [Val Acc 36%] [Train Acc 37%, Loss 2.521234]
Iter 9300. [Val Acc 36%] [Train Acc 37%, Loss 2.750009]
Iter 9400. [Val Acc 36%] [Train Acc 37%, Loss 2.668565]
Iter 9500. [Val Acc 36%] [Train Acc 37%, Loss 2.640807]
Iter 9600. [Val Acc 36%] [Train Acc 37%, Loss 2.611338]
Iter 9700. [Val Acc 36%] [Train Acc 37%, Loss 2.665118]
Iter 9800. [Val Acc 36%] [Train Acc 37%, Loss 2.524589]
Iter 9900. [Val Acc 36%] [Train Acc 37%, Loss 2.984416]
Iter 10000. [Val Acc 36%] [Train Acc 37%, Loss 2.353592]
Iter 10100. [Val Acc 36%] [Train Acc 37%, Loss 2.665622]
Iter 10200. [Val Acc 36%] [Train Acc 37%, Loss 2.676295]
Iter 10300. [Val Acc 36%] [Train Acc 37%, Loss 2.515035]
Iter 10400. [Val Acc 36%] [Train Acc 37%, Loss 2.754944]
Iter 10500. [Val Acc 36%] [Train Acc 37%, Loss 2.550500]
Iter 10600. [Val Acc 36%] [Train Acc 37%, Loss 2.566701]
Iter 10700. [Val Acc 36%] [Train Acc 37%, Loss 2.325579]
Iter 10800. [Val Acc 36%] [Train Acc 37%, Loss 2.609500]
Iter 10900. [Val Acc 36%] [Train Acc 37%, Loss 2.462485]
Iter 11000. [Val Acc 36%] [Train Acc 37%, Loss 2.666566]
Iter 11100. [Val Acc 36%] [Train Acc 37%, Loss 2.339998]
```

```
Iter 11200. [Val Acc 36%] [Train Acc 37%, Loss 2.452852]
Iter 11300. [Val Acc 36%] [Train Acc 38%, Loss 2.928117]
Iter 11400. [Val Acc 36%] [Train Acc 38%, Loss 2.708962]
Iter 11500. [Val Acc 36%] [Train Acc 38%, Loss 2.651714]
Iter 11600. [Val Acc 36%] [Train Acc 38%, Loss 2.655593]
Iter 11700. [Val Acc 36%] [Train Acc 38%, Loss 2.253503]
Iter 11800. [Val Acc 36%] [Train Acc 38%, Loss 2.250873]
Iter 11900. [Val Acc 36%] [Train Acc 38%, Loss 2.694898]
Iter 12000. [Val Acc 37%] [Train Acc 38%, Loss 2.448800]
Iter 12100. [Val Acc 37%] [Train Acc 39%, Loss 2.691409]
Iter 12200. [Val Acc 36%] [Train Acc 38%, Loss 2.416403]
Iter 12300. [Val Acc 36%] [Train Acc 38%, Loss 2.726260]
Iter 12400. [Val Acc 36%] [Train Acc 38%, Loss 2.285505]
Iter 12500. [Val Acc 37%] [Train Acc 38%, Loss 2.554138]
Iter 12600. [Val Acc 37%] [Train Acc 38%, Loss 2.441439]
Iter 12700. [Val Acc 36%] [Train Acc 38%, Loss 2.420176]
Iter 12800. [Val Acc 36%] [Train Acc 38%, Loss 2.866761]
Iter 12900. [Val Acc 37%] [Train Acc 38%, Loss 2.593208]
Iter 13000. [Val Acc 37%] [Train Acc 38%, Loss 2.237072]
Iter 13100. [Val Acc 37%] [Train Acc 38%, Loss 2.198202]
Iter 13200. [Val Acc 37%] [Train Acc 38%, Loss 2.748080]
Iter 13300. [Val Acc 37%] [Train Acc 38%, Loss 2.726922]
Iter 13400. [Val Acc 37%] [Train Acc 38%, Loss 2.654702]
Iter 13500. [Val Acc 37%] [Train Acc 38%, Loss 2.830051]
Iter 13600. [Val Acc 37%] [Train Acc 38%, Loss 2.504918]
Iter 13700. [Val Acc 37%] [Train Acc 38%, Loss 2.435002]
Iter 13800. [Val Acc 37%] [Train Acc 38%, Loss 2.446549]
Iter 13900. [Val Acc 36%] [Train Acc 38%, Loss 2.435920]
Iter 14000. [Val Acc 37%] [Train Acc 38%, Loss 2.578905]
Iter 14100. [Val Acc 37%] [Train Acc 38%, Loss 2.394343]
Iter 14200. [Val Acc 37%] [Train Acc 38%, Loss 2.527936]
Iter 14300. [Val Acc 37%] [Train Acc 38%, Loss 2.786232]
Iter 14400. [Val Acc 37%] [Train Acc 38%, Loss 2.370645]
Iter 14500. [Val Acc 37%] [Train Acc 38%, Loss 2.800895]
Iter 14600. [Val Acc 37%] [Train Acc 38%, Loss 2.373915]
Iter 14700. [Val Acc 37%] [Train Acc 38%, Loss 2.412007]
Iter 14800. [Val Acc 37%] [Train Acc 38%, Loss 2.639515]
Iter 14900. [Val Acc 37%] [Train Acc 38%, Loss 2.727235]
Iter 15000. [Val Acc 37%] [Train Acc 38%, Loss 2.264075]
Iter 15100. [Val Acc 37%] [Train Acc 39%, Loss 2.262968]
Iter 15200. [Val Acc 37%] [Train Acc 39%, Loss 2.422622]
Iter 15300. [Val Acc 37%] [Train Acc 39%, Loss 2.791663]
Iter 15400. [Val Acc 37%] [Train Acc 39%, Loss 2.537657]
Iter 15500. [Val Acc 37%] [Train Acc 39%, Loss 2.571178]
Iter 15600. [Val Acc 37%] [Train Acc 39%, Loss 2.587228]
Iter 15700. [Val Acc 37%] [Train Acc 39%, Loss 2.589490]
Iter 15800. [Val Acc 37%] [Train Acc 39%, Loss 2.186348]
Iter 15900. [Val Acc 37%] [Train Acc 39%, Loss 2.539443]
Iter 16000. [Val Acc 37%] [Train Acc 39%, Loss 2.148071]
Iter 16100. [Val Acc 37%] [Train Acc 39%, Loss 2.360549]
Iter 16200. [Val Acc 37%] [Train Acc 39%, Loss 2.446511]
Iter 16300. [Val Acc 37%] [Train Acc 39%, Loss 2.578602]
Iter 16400. [Val Acc 37%] [Train Acc 39%, Loss 2.621798]
Iter 16500. [Val Acc 37%] [Train Acc 39%, Loss 2.368655]
Iter 16600. [Val Acc 37%] [Train Acc 39%, Loss 2.727678]
Iter 16700. [Val Acc 37%] [Train Acc 39%, Loss 2.151709]
Iter 16800. [Val Acc 37%] [Train Acc 39%, Loss 2.583816]
Iter 16900. [Val Acc 37%] [Train Acc 39%, Loss 2.184199]
Iter 17000. [Val Acc 37%] [Train Acc 39%, Loss 2.456768]
Iter 17100. [Val Acc 37%] [Train Acc 39%, Loss 2.841880]
Iter 17200. [Val Acc 37%] [Train Acc 39%, Loss 2.435870]
Iter 17300. [Val Acc 37%] [Train Acc 39%, Loss 2.659151]
Iter 17400. [Val Acc 37%] [Train Acc 39%, Loss 2.411203]
Iter 17500. [Val Acc 37%] [Train Acc 39%, Loss 2.405800]
Iter 17600. [Val Acc 37%] [Train Acc 38%, Loss 2.772679]
Iter 17700. [Val Acc 37%] [Train Acc 39%, Loss 2.470072]
Iter 17800. [Val Acc 37%] [Train Acc 39%, Loss 2.474600]
Iter 17900. [Val Acc 37%] [Train Acc 39%, Loss 2.286120]
Iter 18000. [Val Acc 37%] [Train Acc 39%, Loss 2.776394]
Iter 18100. [Val Acc 37%] [Train Acc 39%, Loss 2.702604]
Iter 18200. [Val Acc 37%] [Train Acc 39%, Loss 2.387312]
Iter 18300. [Val Acc 37%] [Train Acc 39%, Loss 2.409623]
```

```
Iter 18400. [Val Acc 37%] [Train Acc 39%, Loss 2.194044]
Iter 18500. [Val Acc 37%] [Train Acc 39%, Loss 2.243478]
Iter 18600. [Val Acc 37%] [Train Acc 39%, Loss 2.501188]
Iter 18700. [Val Acc 37%] [Train Acc 39%, Loss 2.557046]
Iter 18800. [Val Acc 37%] [Train Acc 39%, Loss 2.664339]
Iter 18900. [Val Acc 37%] [Train Acc 39%, Loss 2.713795]
Iter 19000. [Val Acc 37%] [Train Acc 40%, Loss 2.378081]
Iter 19100. [Val Acc 37%] [Train Acc 40%, Loss 2.749915]
Iter 19200. [Val Acc 37%] [Train Acc 40%, Loss 2.600175]
Iter 19300. [Val Acc 37%] [Train Acc 40%, Loss 2.284850]
Iter 19400. [Val Acc 37%] [Train Acc 40%, Loss 2.170422]
Iter 19500. [Val Acc 37%] [Train Acc 40%, Loss 2.523248]
Iter 19600. [Val Acc 37%] [Train Acc 40%, Loss 2.593552]
Iter 19700. [Val Acc 37%] [Train Acc 40%, Loss 2.255308]
Iter 19800. [Val Acc 37%] [Train Acc 40%, Loss 2.731459]
Iter 19900. [Val Acc 37%] [Train Acc 39%, Loss 2.037187]
Iter 20000. [Val Acc 37%] [Train Acc 39%, Loss 2.491128]
```





Part (b) -- 10%

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?

Print the output for all of these sentences using the new network and Write below how the new results compare to the previous ones.

Just like before, if you encounter overfitting, 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 [ ]:

```
def to_predict(model):
  data_to_predict = ["You are a","few companies show","There are no","yesterday i was","the game had","yesterday the federal"]
  sentences = []
  for sen in data_to_predict:
    words = sen.split()
    sentence = [word.lower() for word in words]
    sentences.append(sentence)
  #print(sentences)

  for sen in sentences:
    print("The sentences predicted:")
    print(' '.join(sen))
    print("Our model predicted: "+str(make_prediction_torch(model, sen))+"\n")
  return

to_predict(pytorch_wordemb)
```

```
The sentences predicted:
you are a
Our model predicted: good

The sentences predicted:
few companies show
Our model predicted: .

The sentences predicted:
there are no
Our model predicted: other

The sentences predicted:
yesterday i was
Our model predicted: nt

The sentences predicted:
the game had
Our model predicted: to

The sentences predicted:
yesterday the federal
Our model predicted: government
```

The prediction of this architechture compared to the previous one is more sophisticated. Though, also in this model there are still predictions that don't make sense. Some predictions remained the same while others changed to clever options. We can see that the sentence "yesterday the federal..." was completed to the word "government" which makes more sense, and indicates a better understanding of the sentence.

Part (c) -- 4%

Report the test accuracy of your model

In [ ]:

```
print("The train accuracy of the model is: ")
estimate_accuracy_torch(pytorch_wordemb,train4grams)

#A training accuracy of at least 38% is required.
```

```
The train accuracy of the model is:
```

Out[ ]:

```
0.39361904761904765
```

Question 4. Visualizing Word Embeddings (14%)

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, which are a key concept in natural language processing.

Part (a) -- 4%

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 [ ]:

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

The word_emb*layer is a linear fully connected layer, meaning its output is defined by $\textbf{z} = \textbf{W}*{emb_layer}\bullet \textbf{a}_{one_hot}$.

The One hot vectors consist of vectors from the standard base. As a result, this multiplication extracts the indicated colomn of the matrix W_{emb_layer}

. After transpose we get the array of the weights, as required.

Therefore, for any word taken from the dictionary *stoi*, $\mathbf{a}_{one\_hot} = \mathbf{e}_i$

thus:

$$\mathbf{z} = \mathbf{W}_{emb\_layer}\mathbf{e}_i = [\mathbf{W}_{emb\_layer}]_{:,i}$$

When performing this multplication we transform our one-hot vector to a 100 elements vector, carrying the information about this specific word.

In fact, this matrix is another layer for our processing, that deals with processing each of the words independently. This way, we get more precision in our model.

Part (b) -- 5%

One 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 measure of similarity between vector v

and w

is defined as

$$d_{\cos}(\mathbf{v},\mathbf{w}) = \frac{\mathbf{v}^T\mathbf{w}}{||\mathbf{v}||\,||\mathbf{w}||}.$$

We also pre-scale the vectors to have a unit norm, using Numpy's `norm` method.

In [ ]:

```
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']])
```

```
0.1795332
-0.072956644
```

Compute the 5 closest words to the following words:

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

In [ ]:

```
# Write your code here

samples = ['four', 'go', 'what','should', 'school', 'your', 'yesterday', 'not']

print("The 5 closest words for each of the given words:")
for samp in samples:
  out = []
  print("The clostes word of the word "+str(samp)+":")
  a = similarities[:,vocab_stoi[samp]]
  closest_word = np.argsort(a)[-6:-1]
  for w in closest_word:
    print(vocab_itos[w], end=', ')
  print("\n")
```

```
The 5 closest words for each of the given words:
The clostes word of the word four:
think, take, university, these, few,

The clostes word of the word go:
new, law, up, play, $,

The clostes word of the word what:
where, as, when, who, how,

The clostes word of the word should:
will, may, can, could, would,

The clostes word of the word school:
him, here, house, it, same,

The clostes word of the word your:
the, united, my, their, our,

The clostes word of the word yesterday:
center, director, war, before, game,

The clostes word of the word not:
to, only, has, never, nt,
```

In the above list of closest words, for each word we got 5 similar words. One can see that there is'nt a distinct semantic similarity between some these words: for example 'four' is relatively close to 'few' but not to 'university'; the word '$' is not close to 'go' but the 'play' does make sense.

Part (c) -- 5%

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; we will 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 at least two clusters of related words.

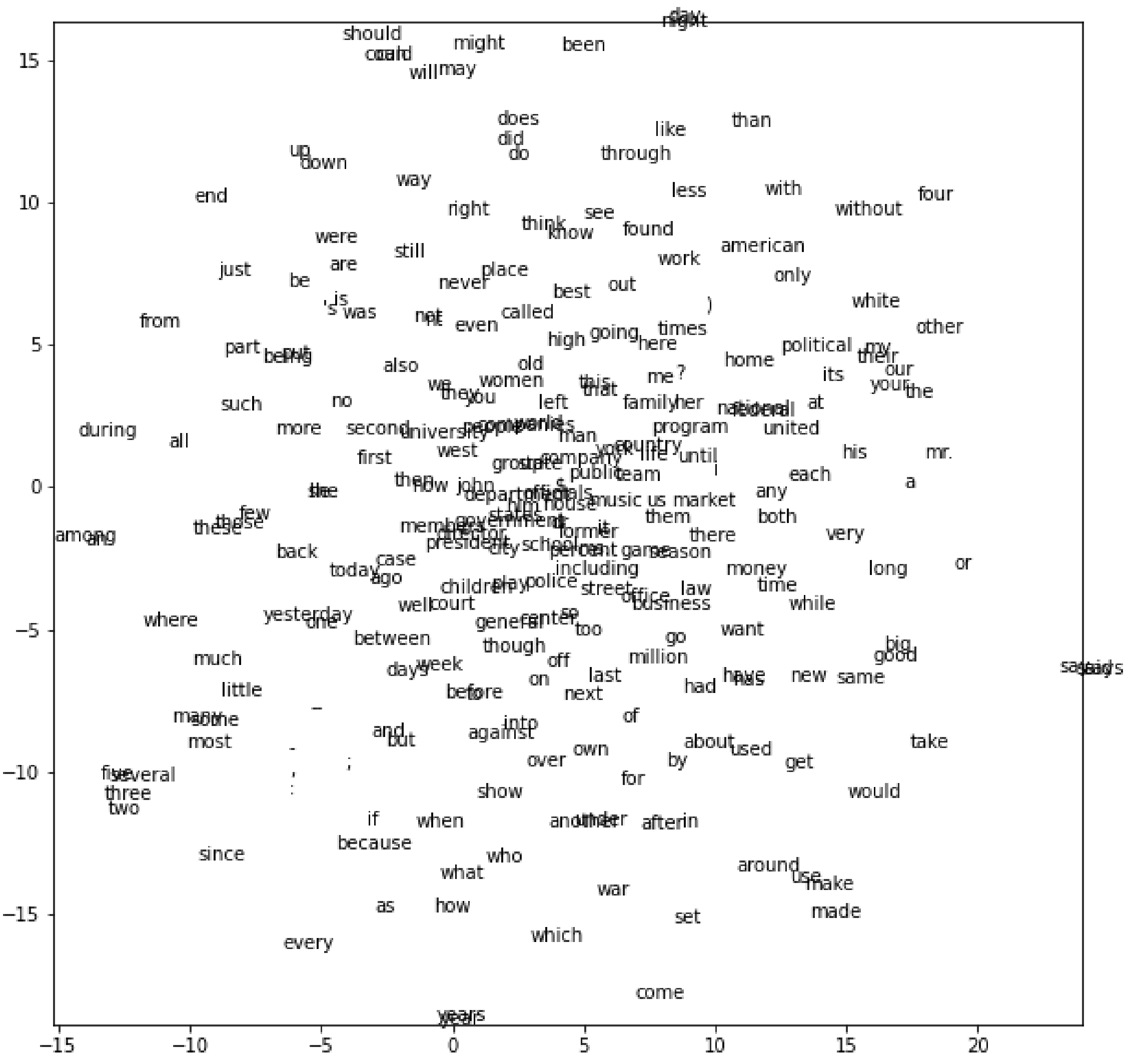
Write below for each cluster what is the commonality (if there is any) and if they make sense.

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.

In [ ]:

```
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()
```

```
/usr/local/lib/python3.8/dist-packages/sklearn/manifold/_t_sne.py:780: FutureWarning: The default initialization in TSNE will change from 'random' to 'pca' in 1.2.
  warnings.warn(
/usr/local/lib/python3.8/dist-packages/sklearn/manifold/_t_sne.py:790: FutureWarning: The default learning rate in TSNE will change from 200.0 to 'auto' in 1.2.
  warnings.warn(
```



After projecting the high dimensional vetors into a 2D map, we get a graph showing the similiraties between words. In a 2D map, it's easier to cluster the results. We identified a few of them:

1. **Belonging verbs - a cluster of 3 verbs varying between pronouns and tenses, coming from the same infinitive. e.g. Have/has/had.**
2. **Amounts - words describing the countable amounts. e.g. five/several/three/two.**
3. **Possessives - clusters of possesives of diffrent persons are close to each other. e.g. my/their/our/your**