

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 [1]:

```
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 [2]:

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

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

Find the path to `raw_sentences.txt` :

In [3]:

```
file_path = '/content/gdrive/MyDrive/Deep Learning/Assignment 2/raw_sentences.txt' # TODO
- UPDATE ME!
```

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 [4]:

```
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 [5]:

```

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 [6]:

```

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 [7]:

```

# Your code goes here
for i in range(10):
    print(train[i])

```

```

# train[:10] # Another option

```

```

['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', '.']

```

Write your answers here:

Punctuations are represented as a one word str, the same as all other words, for example:

- 'last'
- ','
- '?'

Words with apostrophe are divided to 2 words the first is the word until the apostrophe and the second is the apostrophe with the remainder of the word, for example:

- 'she', "'s", 'still'
- 'that', "'s", 'what', "'s", 'good', 'about', 'it', '.'

Usually in NLP tasks we remove punctuation and “stopwords” from the corpus. This is valid when we are dealing with large corpora and we want to do some specific tasks like document similarity, classification, clustering etc. In some projects there is a need to keep everything.

When dealing with apostrophes (which is a very common thing in NLP), there are different approaches. One of the most common approach is to use a contraction dictionary, which will look something like this:

```

contractions = {

```

```

    "ain't": "am not / are not",

```

```

    "aren't": "are not / am not",

```

```

    "can't": "cannot"

```

```
can't : cannot ,
```

```
...}
```

Then, we will write some code to modify your text according to the dictionary, something like this (an example appears below).

In [8]:

```
# An example!
contractions = {
    "ain't": "am not / are not",
    "aren't": "are not / am not",
    "can't": "cannot",
    "what's": "what is",
}

text="What's the best way to ensure this?"
for word in text.split():
    if word.lower() in contractions:
        text = text.replace(word, contractions[word.lower()])
print(text)
```

what is the best way to ensure this?

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 occurrences 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 [9]:

```
# Your code goes here
lis=[]

cnt=collections.Counter

for sen in train:
    lis+=sen
dicte=cnt(lis).most_common(10)
ListDict=[]

for i in range(10):
    TupToList=[0,0]
    TupToList[0]=dicte[i][0]
    TupToList[1]=dicte[i][1]
    ListDict.append(TupToList)
    ListDict[i][1]=100*ListDict[i][1]/len(lis)

print(ListDict)
```

```
[['.', 10.695720015237537], ['it', 3.845648402137913], [',', 3.2499538382458866], ['i', 2.9417097648328943], ['do', 2.6916877236349843], ['to', 2.576740797176065], ['nt', 2.164029763102868], ['?', 2.1427371341784953], ['the', 2.0931652324639396], [''s', 2.0880084238963184]]
```

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 sequences 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

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 [10]:

```
# 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(train))
def convert_words_to_indices(sents):
    """
    This function takes a list of sentences (list of list of words)
    and returns a new list with the same structure, but where each word
    is replaced by its index in `vocab_stoi`.

    Example:
    >>> convert_words_to_indices(['one', 'in', 'five', 'are', 'over', 'here'], ['other',
    'one', 'since', 'yesterday'], ['you'])
    [[148, 98, 70, 23, 154, 89], [151, 148, 181, 246], [248]]
    """

    # Write your code here
    temp1=[]
    global vocab_stoi
    if isinstance(sents[0], str):
        for word in range(len(sents)):
            wo = sents[word]
            temp1.append(vocab_stoi[str(wo)])
    else:
        for sent in range(len(sents)):
            temp2=[]
            for word in range(len(sents[sent])):
                wo = sents[sent][word]
                temp2.append(vocab_stoi[str(wo)])
            temp1.append(temp2)
    return temp1
def generate_4grams(seqs):
    """
    This function takes a list of sentences (list of lists) and returns
    a new list containing the 4-grams (four consecutively occurring words)
    that appear in the sentences. Note that a unique 4-gram can appear multiple
    times, one per each time that the 4-gram appears in the data parameter `seqs`.

    Example:
    >>> generate_4grams([[148, 98, 70, 23, 154, 89], [151, 148, 181, 246], [248]])
    [[148, 98, 70, 23], [98, 70, 23, 154], [70, 23, 154, 89], [151, 148, 181, 246]]
    >>> generate_4grams([[1, 1, 1, 1, 1]])
    [[1, 1, 1, 1], [1, 1, 1, 1]]
    """

    # Write your code here
    grams_list = []
    for sent in seqs:
        if len(sent)>=4:
            for i in range(len(sent)-3):
                grams_list.append(sent[i:i+4])
    return grams_list
def process_data(sents):
    """
    This function takes a list of sentences (list of lists), and generates an
    numpy matrix with shape [N, 4] containing indices of words in 4-grams.
```

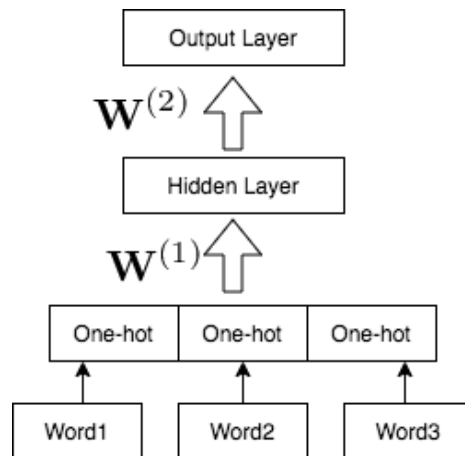
```

"""
indices = convert_words_to_indices(sents)
fourgrams = generate_4grams(indices)
return np.array(fourgrams)
#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)

```

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 classification 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 concatenation 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 [11]:

```

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.

    input - vector with shape D (1D or 2D)
    output - vector with shape (D,250)
    """
    if np.size(data[0])==1:
        temp=np.zeros((len(data),250))
        for i in range(len(data)):
            temp[i][data[i]]=1
    else:
        temp=np.zeros((len(data),len(data[0]),250))
        for i in range(len(data)):
            for j in range(len(data[i])):
                temp[i][j][data[i][j]]=1
    return temp

# Write your code here

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 a 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 indices 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 [12]:

```
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])
        hid = self.layer1(inp)
        out = self.layer2(hid)
        return out
    # Note that we will be using the nn.CrossEntropyLoss(), which computes the softmax
    # operation internally, as loss criterion
```

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 [13]:

```
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`
    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.
    """
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(),
                             lr=learning_rate,
                             weight_decay=weight_decay)

    iters, losses = [], []
    iters_sub, train_accs, val_accs = [], [], []

    n = 0 # the number of iterations
    while True:
        for i in range(0, train_data.shape[0], batch_size):
            if (i + batch_size) > train_data.shape[0]:
                break

            # get the input and targets of a minibatch
            xt, st = get_batch(train_data, i, i + batch_size, onehot=False)

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

            zs = model(xt) # compute prediction logit
            loss = criterion(zs, st) # compute the total loss
            loss.backward() # compute updates for each parameter

```

```

optimizer.step() # make the updates for each parameter
optimizer.zero_grad() # a clean up step for PyTorch

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

if n % 500 == 0:
    iters_sub.append(n)
    train_cost = float(loss.detach().numpy())
    train_acc = estimate_accuracy_torch(model, train_data)
    train_accs.append(train_acc)
    val_acc = estimate_accuracy_torch(model, validation_data)
    val_accs.append(val_acc)
    print("Iter %d. [Val Acc %.0f%%] [Train Acc %.0f%%, Loss %f]" % (
        n, val_acc * 100, train_acc * 100, train_cost))

    if (checkpoint_path is not None) and n > 0:
        torch.save(model.state_dict(), checkpoint_path.format(n))

# increment the iteration number
n += 1

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

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

    plt.title("Learning Curve: Accuracy per Iteration")
    plt.plot(iters_sub, train_accs, label="Train")
    plt.plot(iters_sub, val_accs, label="Validation")
    plt.xlabel("Iterations")
    plt.ylabel("Accuracy")
    plt.legend(loc='best')
    plt.show()

```

In [14]:

```

pytorch_mlp = PyTorchMLP()
learning_curve_info = run_pytorch_gradient_descent(pytorch_mlp, train_data=train4grams,
                                                    validation_data=valid4grams,
                                                    batch_size=900,
                                                    learning_rate=0.008,
                                                    weight_decay=0,
                                                    max_iters=1000,
                                                    checkpoint_path=None)

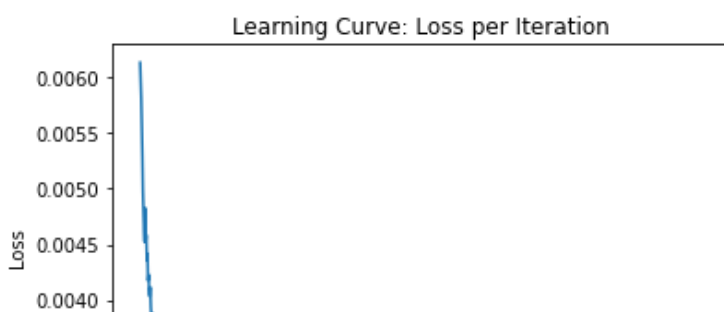
plot_learning_curve(*learning_curve_info)

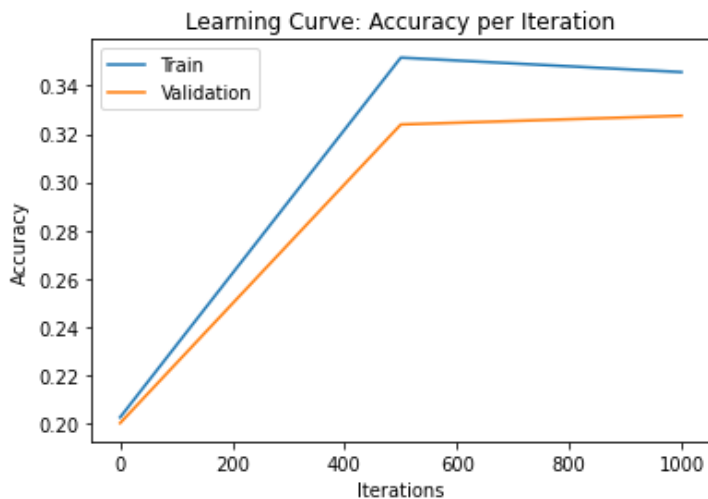
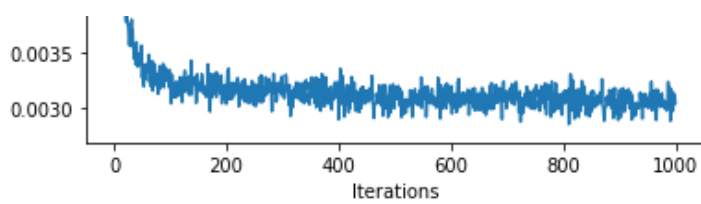
```

```

Iter 0. [Val Acc 20%] [Train Acc 20%, Loss 5.519297]
Iter 500. [Val Acc 32%] [Train Acc 35%, Loss 2.787149]
Iter 1000. [Val Acc 33%] [Train Acc 35%, Loss 2.735418]

```





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 [15]:

```
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

    # Write your code here

    data=sentence[-3:]

    for i in range(3):
        data[i]=vocab_stoi[str(data[i])]

    data=make_onehot(np.array(data))
    y=model(torch.Tensor(data))
    y = y.detach().numpy() # convert the PyTorch tensor => numpy array
    prediction = np.argmax(y, axis=1)

    return vocab_itos[int(prediction)]
```

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 name had"

- "the game had"
- "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 [16]:

```
# Write your code here

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

good
.
other
nt
to
government

Write your answers here:

We don't see any indication that we overfit, each sentence had a different output. Some of the answers also make sense. the words: "other" , "goverment", "nt" ,seems like a good complition to the sentence, not necessarily as the original, but do make sense. Rest of the words doesn't

Part (e) -- 6%

Report the test accuracy of your model

In [17]:

```
# Write your code here
counter=0
for sentence in test4grams:
    data=sentence
    temp1=[]

    for i in range(4):
        temp1.append(vocab_itos[int(data[i])])
    temp = make_prediction_torch(pytorch_mlp,temp1[:3])

    # We chose to use the function previously defined to calculate the predicted word
    if (str(temp)==str(temp1[3])):
        counter+=1

print("Test accuracy = " + str(round(counter/len(test4grams)*100)) + "%")
```

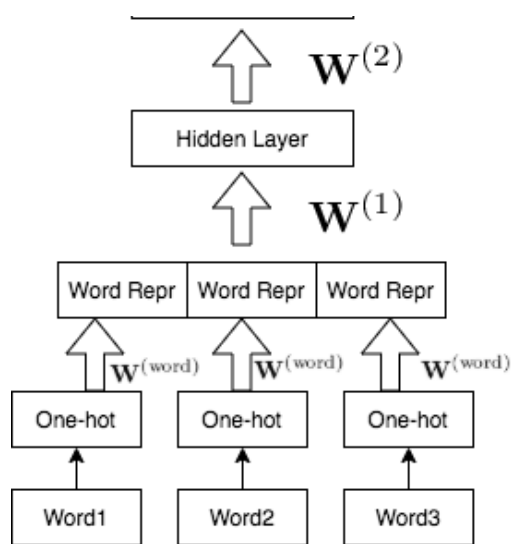
Test accuracy = 33%

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:

Output Layer



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%

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 [18]:

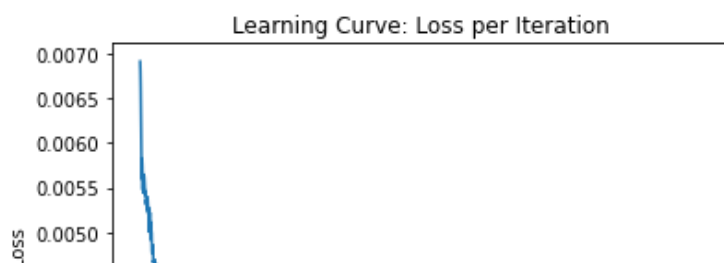
```
class PyTorchWordEmb(nn.Module):
    def __init__(self, emb_size=100, num_hidden=300, vocab_size=250):
        super(PyTorchWordEmb, self).__init__()
        self.word_emb_layer = nn.Linear(vocab_size, emb_size, bias=False)
        self.fc_layer1 = nn.Linear(emb_size * 3, num_hidden)
        self.fc_layer2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

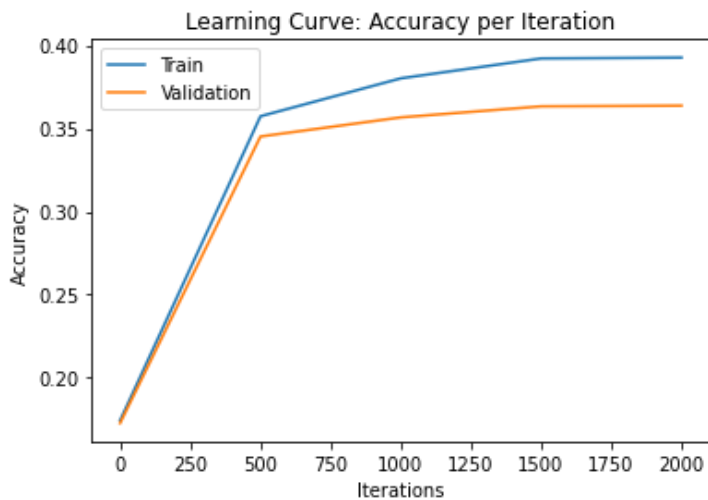
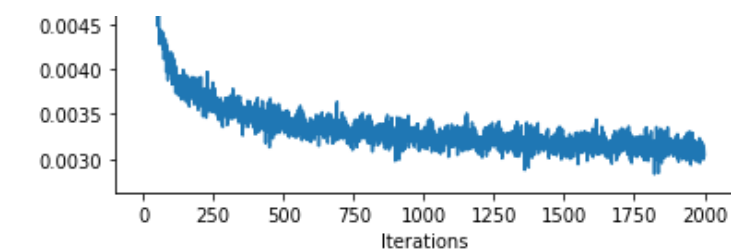
pytorch_wordemb= PyTorchWordEmb()

result = run_pytorch_gradient_descent(pytorch_wordemb, train_data=train4grams,
                                     validation_data=valid4grams,
                                     batch_size=800,
                                     learning_rate=0.008,
                                     weight_decay=0,
                                     max_iters=2000)

plot_learning_curve(*result)
```

```
Iter 0. [Val Acc 17%] [Train Acc 17%, Loss 5.528659]
Iter 500. [Val Acc 35%] [Train Acc 36%, Loss 2.690449]
Iter 1000. [Val Acc 36%] [Train Acc 38%, Loss 2.589056]
Iter 1500. [Val Acc 36%] [Train Acc 39%, Loss 2.474044]
Iter 2000. [Val Acc 36%] [Train Acc 39%, Loss 2.434615]
```





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 $\geq 38\%$.

In [19]:

```
# Your code goes here

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

```
man
.
children
going
been
government
```

Write your explanation here:

As before we aren't seeing any indications for overfitting, and some of the words are the same as before: " .", "government", the other words: "man", "going", children, seems like a good completion to the sentences, however

the other word "been", seems like a bad completion, and the word before: government seems like a much better prediction.

Part (c) -- 4%

Report the test accuracy of your model

In [20]:

```
# Write your code here
counter=0
for sentence in test4grams:
    data=sentence
    temp1=[]

    for i in range(4):
        temp1.append(vocab_itos[int(data[i])])
    temp = make_prediction_torch(pytorch_wordemb, temp1[:3])

    # We chose to use the function previously defined to calculate the predicted word
    if (str(temp)==str(temp1[3])):
        counter+=1

print("Test accuracy = " + str(round(counter/len(test4grams)*100)) + "%")
```

Test accuracy = 37%

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 [21]:

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

Write your explanation here:

We get the word representation of a word by multiplying its one-hot vector and the embeded weights for that word.

From the initializer of the `pytorch_wordemb` model, the shape of `word_emb_weights` is `[vocab_size, emb_size] = [250, 100]` This means that each row contain the weights for a word.

Dimensions wise: the shapes of `word_emb_weights` and `word_emb`, W , are both `[250, 100]` the shape of the one hot vector of 1 word, X , is `[1, 250]` $X(W.T)$ give a vector of `[1, 100]` which is the vector representation of the word.

Code wise: We get the index of a given word, i , using `vocab_stoi["any"]` Since the one hot vector only has 1 at index i , The matrix multiplication of $X(W.T)$ will only return row i of `word_emb`, ie `word_emb[i, :]`. Its shape is `[1, 100]` which confirms our intuition above. Therefore, `word_emb[i,:]` gives us the vector word representation of the word.

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 \mathbf{v} and \mathbf{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 [22]:

```
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']])

0.033606134
-0.12215116
```

Compute the 5 closest words to the following words:

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

In [23]:

```
# Write your code here

def compute_closest(word, k=5):
    # sort the element of the training set by distance to the i-th
    # element of val_xs
    neighbours = sorted(enumerate(similarities[:, vocab_stoi[word]]),
                        key=lambda r: r[1],
                        reverse=True)

    # obtain the top k closest index and return it
    neighbour_indices = [index for (index, dist) in neighbours[:k]]
    neighbour_words = [vocab_itos[i] for i in neighbour_indices]
    return neighbour_words

print(compute_closest('four'))
print(compute_closest('go'))
print(compute_closest('what'))
print(compute_closest('should'))
print(compute_closest('school'))
print(compute_closest('your'))
print(compute_closest('yesterday'))
print(compute_closest('not'))
```

output: nearest word should be the word itself

```
['four', 'group', 'two', 'yesterday', 'until']
['go', 'come', 'being', 'going', 'program']
['what', 'where', 'how', 'who', 'think']
['should', 'would', 'could', 'can', 'will']
['school', '?', 'office', 'court', 'west']
['your', 'my', 'his', 'its', 'their']
['yesterday', 'times', 'group', 'four', 'man']
['not', 'nt', 'never', 'us', 'well']
```

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.

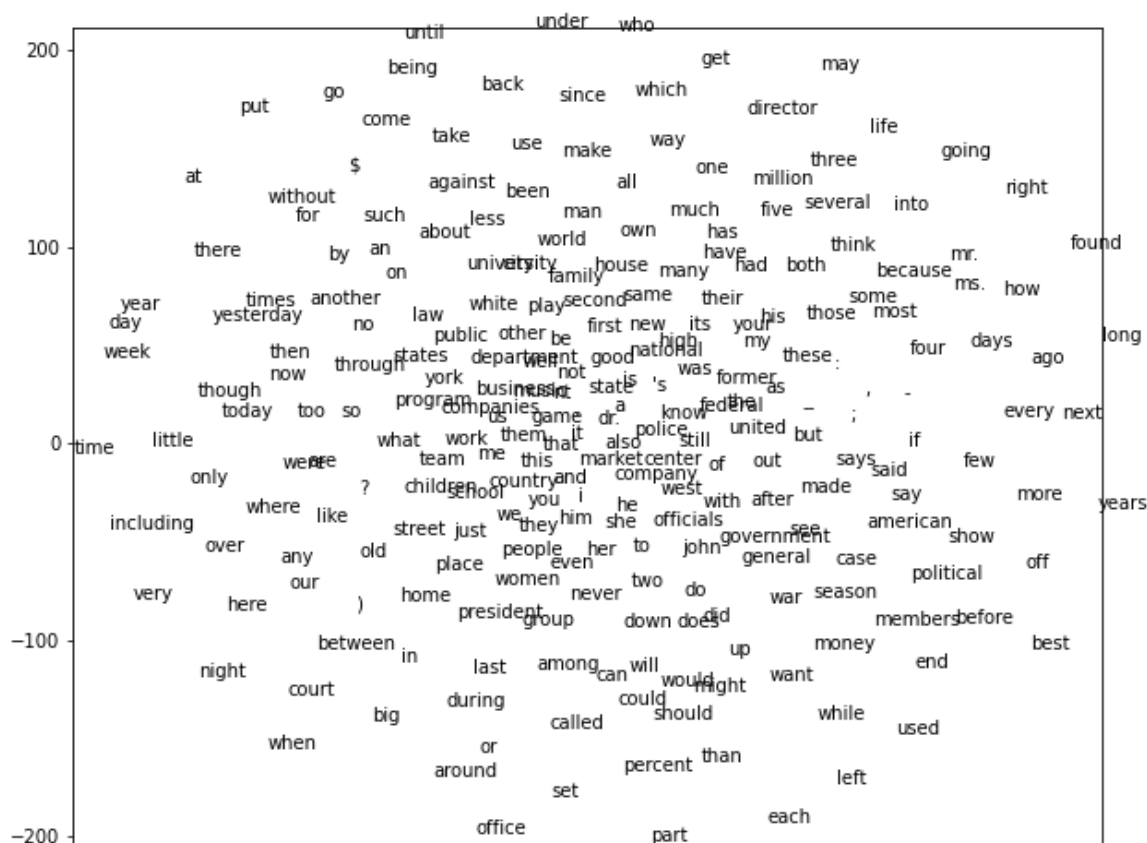
In [24]:

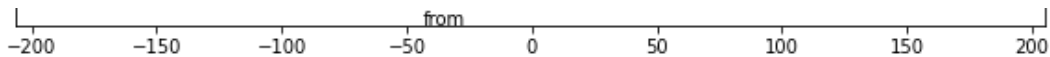
```
import sklearn.manifold
tsne = sklearn.manifold.TSNE()
Y = tsne.fit_transform(word_emb)

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/python3.7/dist-packages/sklearn/manifold/_t_sne.py:783: FutureWarning: The
default initialization in TSNE will change from 'random' to 'pca' in 1.2.
```

```
FutureWarning,  
/usr/local/lib/python3.7/dist-packages/sklearn/manifold/_t_sne.py:793: FutureWarning: The  
default learning rate in TSNE will change from 200.0 to 'auto' in 1.2.  
FutureWarning,
```





Explain and discuss your results here:

Clusters for example:

- cluster1 = [year, day, week], which are entities related to descriptions of time, and reveals a bias in the data that it relates roles of governance to males
- cluster2 = [we, you, they, him, i, he, she, her] are entities that of preposition words.

We can see that there are some cluster of words that are close which unreadable. We will try to talk about clusters that are readable: the first cluster is the cluster at $\sim[50,-100]$ do, does, did. All this words are represent almost the same meaning but for different times and subject. It makes alot of sense, because these words indeed could be replaced by each other at the same sentences with depandence on time and subject. Another word cluster is at $\sim[-180,60]$: year,day week, and more time related words a bit further: years, days, ago and more. speaking on the first 3 words. All of them could be placed at the same sentence.

In [24]: