Assignment2

April 19, 2021

1 Assignment 2: Word Prediction

Deadline: Sunday, April 18th, by 9pm.

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.

Submited By: Yuval Haitman & Amit Efraim

[ ]:
```
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
```

1.1 Question 1. Data (15%)

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:

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

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

Find the path to `raw_sentences.txt`:

[ ]:
```
file_path = '/content/gdrive/MyDrive/Colab Notebooks/Into To Deep Learning/
↪Assignment2/raw_sentences.txt'
```

The following code reads the sentences in our file, split each sentence into its individual words, and stores the sentences (list of words) in the variable `sentences`.

[ ]:
```
sentences = []
for line in open(file_path):
    words = line.split()
    sentence = [word.lower() for word in words]
    sentences.append(sentence)
```

There are 97,162 sentences in total, and these sentences are composed of 250 distinct words.

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

```
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.

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

1.1.1 Part (a) – 2%

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

[ ]:
```
import random
[s for s in random.sample(train,10)]
```

[ ]:
```
[['i', 'say', 'use', 'it', '.'],
 ['make', 'that', 'five', 'million', '.'],
 ['but', 'it', 'is', 'no', 'game', '.'],
 ['i', 'know', 'we', 'can', 'do', 'it', '.'],
 ['i',
  'did',
  'nt',
  'think',
  'of',
  'it',
  'that',
  'way',
  'at',
  'the',
  'time',
  '.'],
```

```
['where', 'did', 'he', 'or', 'she', 'get', 'it', '?'],
['it', 'might', 'not', 'play', 'any', 'part', 'at', 'all', '.'],
['then', 'they', 'said', 'what', 'about', 'next', 'year', '?'],
['she', 'had', 'to', 'work', 'at', 'it', '.'],
['there', 'is', 'still', 'much', 'to', 'work', 'out', '.']]
```

Write your answers here:

punctuations are considered as seperate word while apostrophes are considered as part of the next word (i.e. it's -> [it, 's]).

1.1.2 Part (b) – 3%

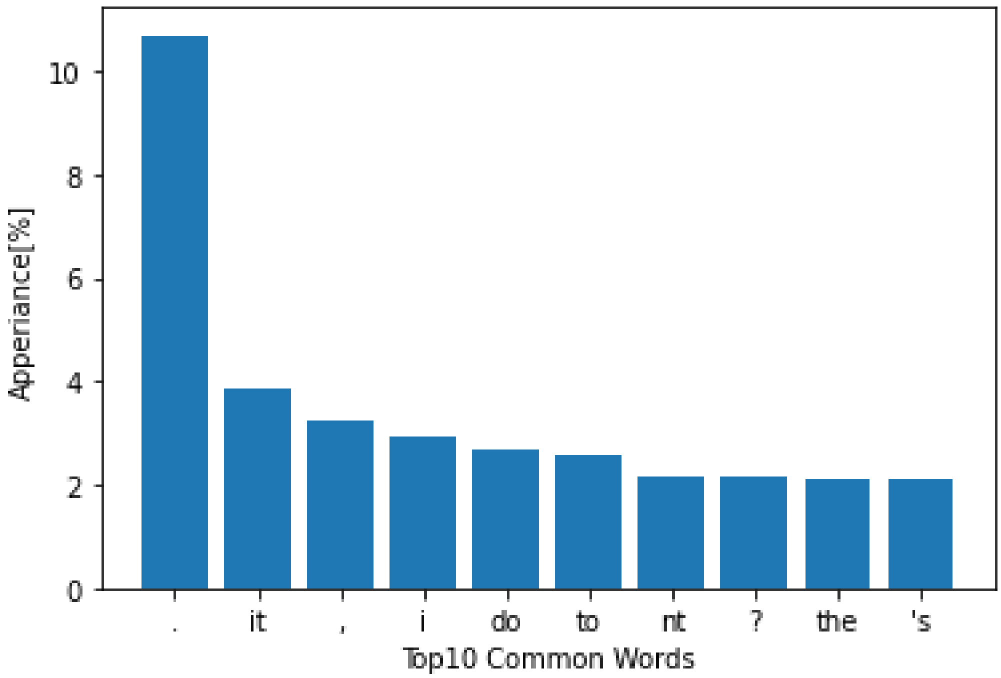
Print the 10 most common words in the vocabulary and how often does each of these words appear in the training sentences. Express the second quantity as a percentage (i.e. number of occurences of the word / total number of words in the training set).

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.

[ ]:
```
all_train_words =[w for s in train for w in s ]
word_counter_dict = collections.Counter(all_train_words)
#Sort the counter dict by value keeping keys from large to small
top10_common_words = sorted(word_counter_dict, key=word_counter_dict.
 ↪get,reverse=True)[:10]
top10_common_words_appearince = [100 *word_counter_dict[w]/len(all_train_words)␣
 ↪for w in top10_common_words]
plt.bar(np.arange(10),␣
 ↪top10_common_words_appearince,tick_label=top10_common_words)

plt.ylabel('Apperiance[%]')
plt.xlabel('Top10 Common Words')
```

[ ]:
```
Text(0.5, 0, 'Top10 Common Words')
```



1.1.3 Part (c) – 10%

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.

[ ]:
```
# 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
```

```python
    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
    return [[vocab_stoi[w] for w in s] for s in sents]

def generate_4grams(seqs):
    """
    This function takes a list of sentences (list of lists) and returns
    a new list containing the 4-grams (four consequentively occuring words)
    that appear in the sentences. Note that a unique 4-gram can appear multiple
    times, one per each time that the 4-gram appears in the data parameter␣
 ↪`seqs`.

    Example:

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

    # Write your code here
    res = []
    for s in seqs:
      if len(s) < 4:
        continue
      _4grams_list = [s[i:i+4] for i in range(len(s)-3)]
      res.extend(_4grams_list)

    return res


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)
```

1.2 Question 2. A Multi-Layer Perceptron (40%)

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:

[ ]:
```
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)
    """

    #Consist dimentions
    D = data.shape[0]
    data = np.reshape(data, (D,-1))
    numOfWords = data.shape[1]

    return (np.arange(250) == np.reshape(data,(D,numOfWords,1))).astype(int)

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
```

1.2.1 Part (a) – 7%

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:

```
[ ]: class PyTorchMLP(nn.Module):
         def __init__(self, num_hidden=400):
             super(PyTorchMLP, self).__init__()
             self.layer1 = nn.Linear(750, num_hidden)
             self.layer2 = nn.Linear(num_hidden, 250)
             self.num_hidden = num_hidden
         def forward(self, inp):
             inp = inp.reshape([-1, 750])

             # TODO: complete this function
             # Note that we will be using the nn.CrossEntropyLoss(), which computes␣
      ↪the softmax operation internally, as loss criterion
             net = self.layer1(inp)
             #net = torch.relu(net) #Check if needed
             net = self.layer2(net)
             return net
```

1.2.2 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.

```python
[ ]: 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')
```

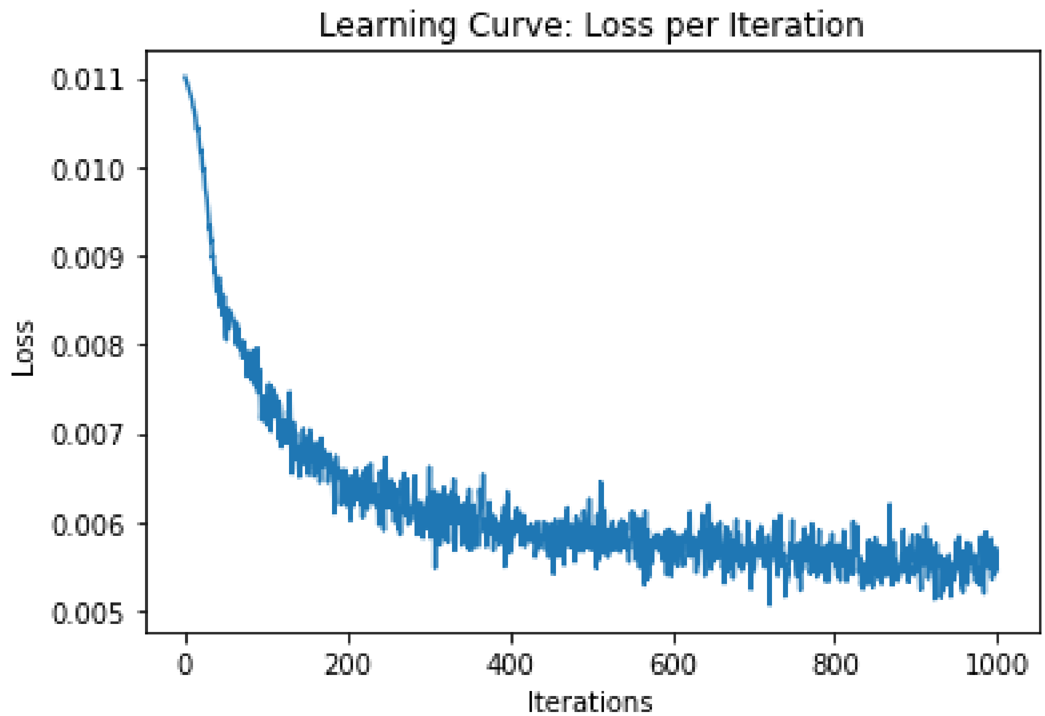
```
    will save the model parameters in Google Drive every 500 iterations.
    You will have to make sure that the path exists (i.e. you'll need to create
    the folder Intro_to_Deep_Learning, mlp, etc...). Your Google Drive will be␣
↪populated with files:

    - /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
```

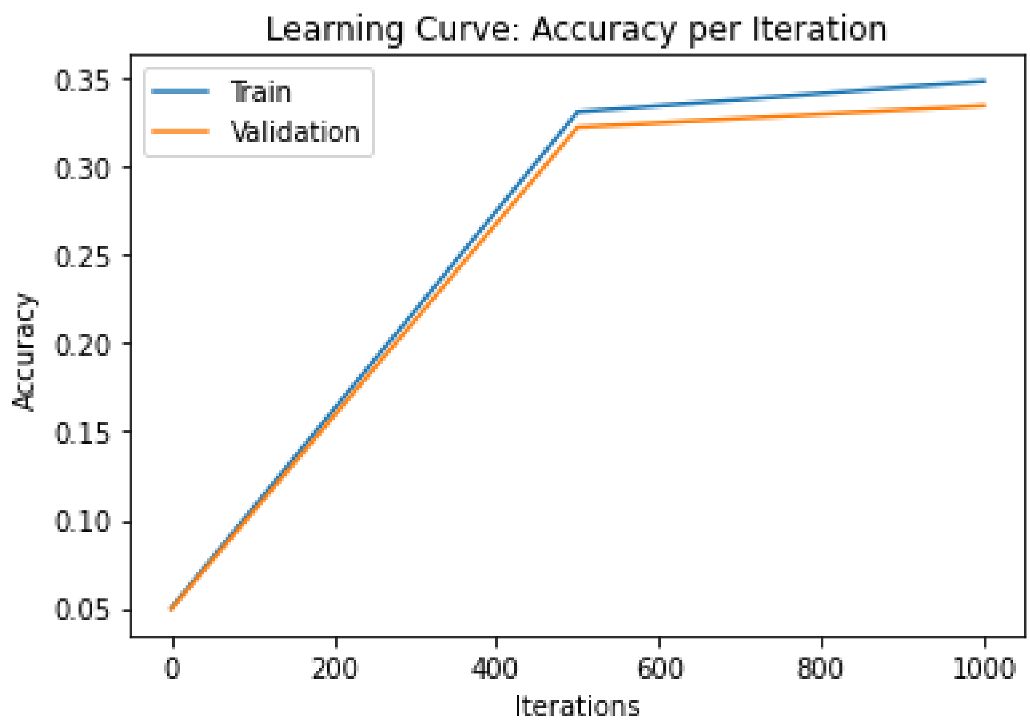
```
            # save the current training information
            iters.append(n)
            losses.append(float(loss)/batch_size)  # compute *average* loss

            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()
```

[ ]:
```
random.seed(42)
pytorch_mlp = PyTorchMLP()
# learning_curve_info = run_pytorch_gradient_descent(pytorch_mlp, ...)
learning_curve_info = run_pytorch_gradient_descent(pytorch_mlp,learning_rate=0.
 ↪001,batch_size=500)
```

```
plot_learning_curve(*learning_curve_info)
```

```
Iter 0. [Val Acc 5%] [Train Acc 5%, Loss 5.505772]
Iter 500. [Val Acc 32%] [Train Acc 33%, Loss 2.841699]
Iter 1000. [Val Acc 33%] [Train Acc 35%, Loss 2.730380]
```





1.2.3 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.

```
[ ]: 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

    #Preprocess Sentece
    word_idxs = np.array(convert_words_to_indices([sentence[-3:]]))
```

```
    input_data = torch.Tensor(np.reshape(make_onehot(word_idxs), (-1,250)))


    #Model Eval
    model.eval()
    y = model(input_data)
    y = y.detach().numpy()
    pred = np.argmax(y, axis=1)[0]

    #Convert to Word
    return vocab_itos[pred]
```

1.2.4 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"

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.

[ ]:
```
# Write your code here
def pred_next_word(sentences, model):
  """ Print the next predicted word in each given sentence using the given model␣
 ↪"""

  sentences_splited = [sentences[i].split() for i in range(len(sentences))]
  # len(sentences_splited)
  for i in range(len(sentences_splited)):
    print("{} [{}]".format(sentences[i],␣
 ↪make_prediction_torch(model,sentences_splited[i])))

sentences = ["you are a", "few companies show", "there are no", "yesterday i␣
 ↪was", "the game had", "yesterday the federal"]
pred_next_word(sentences, model=pytorch_mlp)
```

```
you are a [good]
few companies show [.]
there are no [other]
yesterday i was [nt]
```

```
the game had [a]
yesterday the federal [government]
```

Write your answers here:

It seems that there is no overfitting from the above reported results. All the completions seems reasonable when we considered them as a part from a greater sentence.

1.2.5 Part (e) – 4%

Report the test accuracy of your model

```
[ ]: # Write your code here
print("pytorch_mlp Model Test ACC: {}".
 ↪format(estimate_accuracy_torch(pytorch_mlp, test4grams)))
```

```
pytorch_mlp Model Test ACC: 0.3348710990502035
```

1.3 Question 3. Learning Word Embeddings (20 %)

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

Our model will look like this:

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

1.3.1 Part (a) – 8%

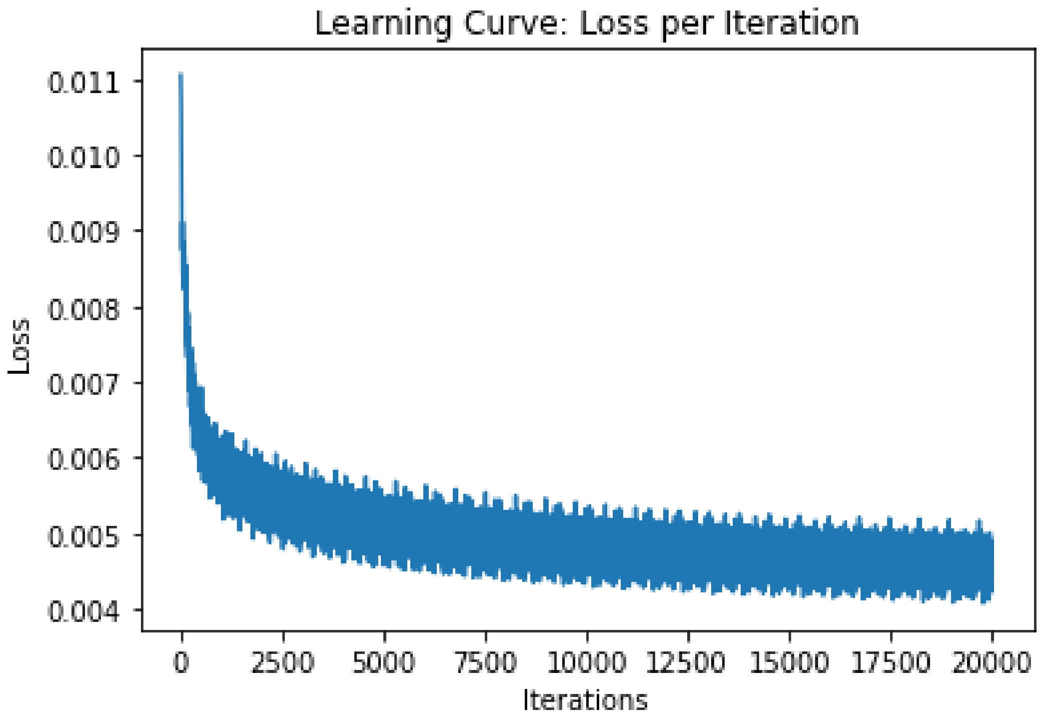
The PyTorch model is implemented for you. Use `run_pytorch_gradient_descent` to train your PyTorch MLP model to obtain a training accuracy of at least 38%. Plot the learning curve using the `plot_learning_curve` function provided to you, and include your plot in your PDF submission.

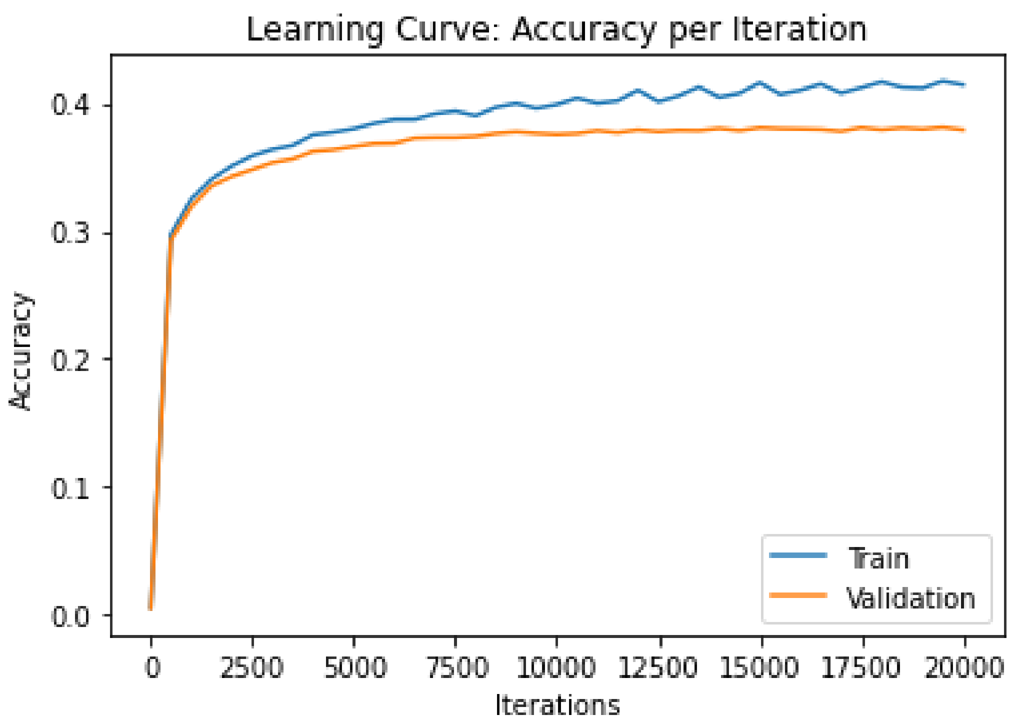
```
[ ]: class PyTorchWordEmb(nn.Module):
         def __init__(self, emb_size=100, num_hidden=300, vocab_size=250):
             super(PyTorchWordEmb, self).__init__()
             self.word_emb_layer = nn.Linear(vocab_size, emb_size, bias=False)
             self.fc_layer1 = nn.Linear(emb_size * 3,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

```python
result = run_pytorch_gradient_descent(pytorch_wordemb,
                                      max_iters=20000,
                                      learning_rate=0.001,
                                      batch_size=500)

plot_learning_curve(*result)
```

```
Iter 0. [Val Acc 0%] [Train Acc 1%, Loss 5.527793]
Iter 500. [Val Acc 29%] [Train Acc 30%, Loss 3.094113]
Iter 1000. [Val Acc 32%] [Train Acc 33%, Loss 2.883622]
Iter 1500. [Val Acc 34%] [Train Acc 34%, Loss 2.712956]
Iter 2000. [Val Acc 34%] [Train Acc 35%, Loss 2.747206]
Iter 2500. [Val Acc 35%] [Train Acc 36%, Loss 2.867279]
Iter 3000. [Val Acc 35%] [Train Acc 36%, Loss 2.589725]
Iter 3500. [Val Acc 36%] [Train Acc 37%, Loss 2.521402]
Iter 4000. [Val Acc 36%] [Train Acc 38%, Loss 2.502347]
Iter 4500. [Val Acc 36%] [Train Acc 38%, Loss 2.613141]
Iter 5000. [Val Acc 37%] [Train Acc 38%, Loss 2.303342]
Iter 5500. [Val Acc 37%] [Train Acc 38%, Loss 2.389072]
Iter 6000. [Val Acc 37%] [Train Acc 39%, Loss 2.623964]
Iter 6500. [Val Acc 37%] [Train Acc 39%, Loss 2.412852]
Iter 7000. [Val Acc 37%] [Train Acc 39%, Loss 2.270297]
Iter 7500. [Val Acc 37%] [Train Acc 39%, Loss 2.395451]
Iter 8000. [Val Acc 37%] [Train Acc 39%, Loss 2.429410]
Iter 8500. [Val Acc 38%] [Train Acc 40%, Loss 2.320392]
Iter 9000. [Val Acc 38%] [Train Acc 40%, Loss 2.318739]
Iter 9500. [Val Acc 38%] [Train Acc 40%, Loss 2.348451]
Iter 10000. [Val Acc 38%] [Train Acc 40%, Loss 2.537990]
Iter 10500. [Val Acc 38%] [Train Acc 40%, Loss 2.509419]
Iter 11000. [Val Acc 38%] [Train Acc 40%, Loss 2.483846]
Iter 11500. [Val Acc 38%] [Train Acc 40%, Loss 2.398628]
Iter 12000. [Val Acc 38%] [Train Acc 41%, Loss 2.324288]
Iter 12500. [Val Acc 38%] [Train Acc 40%, Loss 2.354191]
Iter 13000. [Val Acc 38%] [Train Acc 41%, Loss 2.271341]
Iter 13500. [Val Acc 38%] [Train Acc 41%, Loss 2.353883]
Iter 14000. [Val Acc 38%] [Train Acc 40%, Loss 2.206335]
Iter 14500. [Val Acc 38%] [Train Acc 41%, Loss 2.321839]
Iter 15000. [Val Acc 38%] [Train Acc 42%, Loss 2.359423]
Iter 15500. [Val Acc 38%] [Train Acc 41%, Loss 2.102255]
Iter 16000. [Val Acc 38%] [Train Acc 41%, Loss 2.296530]
Iter 16500. [Val Acc 38%] [Train Acc 42%, Loss 2.286941]
Iter 17000. [Val Acc 38%] [Train Acc 41%, Loss 2.372958]
Iter 17500. [Val Acc 38%] [Train Acc 41%, Loss 2.356478]
Iter 18000. [Val Acc 38%] [Train Acc 42%, Loss 2.310150]
Iter 18500. [Val Acc 38%] [Train Acc 41%, Loss 2.351719]
Iter 19000. [Val Acc 38%] [Train Acc 41%, Loss 2.370573]
Iter 19500. [Val Acc 38%] [Train Acc 42%, Loss 2.167024]
```

```
Iter 20000. [Val Acc 38%] [Train Acc 41%, Loss 2.197388]
```





1.3.2 Part (b) – 8%

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%.

[ ]:
```
# Your code goes here
sentences = ["you are a", "few companies show", "there are no", "yesterday i␣
↪was", "the game had", "yesterday the federal"]
pred_next_word(sentences, model=pytorch_wordemb)
```

```
you are a [good]
few companies show [.]
there are no [more]
yesterday i was [nt]
the game had [to]
yesterday the federal [government]
```

Write your explanation here:

From the above results we can see that there is no overfiting. We got very simmilar results to those shown in the Pytorch_MLP, most of the sentences got same comlletion except for 2 whice are also reasonable completions.

1.3.3 Part (c) – 4%

Report the test accuracy of your model

[ ]:
```
# Write your code here
print("pytorch_wordemb Model Test ACC: {}".
↪format(estimate_accuracy_torch(pytorch_wordemb, test4grams)))
```

```
pytorch_wordemb Model Test ACC: 0.3847197578540862
```

1.4 Question 4. Visualizing Word Embeddings (15%)

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.

1.4.1 Part (a) – 5%

The code below extracts the **weights** of the word embedding layer, and converts the PyTorch tensor into an numpy array. Explain why each *row* of word_emb contains the vector representing of a word. For example word_emb[vocab_stoi["any"],:] contains the vector representation of the word "any".

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

[ ]:
```
array([-2.7030141e-03, -6.2076524e-03, -3.2315094e-02,  6.6087127e-01,
        5.5215102e-01, -9.7870808e-03, -6.1160695e-02, -6.7887963e-03,
        1.1452578e-01, -1.3035231e-02, -5.1333848e-03,  6.1236298e-01,
       -3.8276315e-03, -2.9205784e-02,  2.7207384e-01, -1.3526751e-02,
       -3.3846961e-03, -3.3751661e-03, -1.0508759e-02, -4.5689803e-02,
       -1.5520877e-02, -3.2779528e-03, -6.8700211e-03, -2.5644431e-02,
       -4.8682358e-02, -4.5190370e-03, -5.2925744e-03, -5.0898958e-02,
        4.8416382e-01, -1.3071888e-02, -2.4699664e-03, -3.9506280e-03,
       -3.6847364e-02, -2.3257807e-02,  1.7926799e-01,  7.1129382e-01,
       -1.5419571e-03,  1.2432112e-01, -5.5673182e-02, -2.6860235e-03,
       -1.5128863e-02, -4.4186715e-02, -5.3444123e-03, -3.6458366e-02,
       -2.7136356e-03, -5.5039041e-02, -4.0189424e-03, -9.8583531e-03,
       -4.6590837e-03, -4.5702313e-03, -1.4516124e-02, -2.6459008e-02,
       -5.8739088e-02, -2.2418018e-02, -2.4926029e-03, -4.9879947e-03,
       -8.0318572e-03, -1.0191725e-02, -2.2655269e-02, -5.8187425e-02,
       -4.0297797e-03,  5.4775500e-01, -1.8820362e-02,  3.0253893e-01,
       -4.2693331e-03, -5.7425546e-03, -7.6309300e-04, -3.9311972e-02,
       -3.2196473e-02,  3.9444143e-01,  1.5565370e-01, -3.3636268e-02,
       -6.9739777e-03, -2.9330146e-03, -3.5383923e-03, -5.9446447e-02,
       -2.6466749e-03, -2.9696979e-02,  3.4494880e-01, -3.7471761e-03,
        6.6360229e-01, -3.3496775e-02, -3.7218891e-02, -5.6944592e-03,
       -3.8848929e-02, -9.5453430e-03, -3.5610897e-03, -1.1496383e-02,
       -2.1843098e-03, -3.7440215e-03,  1.6262302e-01, -5.6854948e-02,
       -1.8041927e-02, -4.7443308e-02, -2.6146590e-04,  1.5751551e-01,
       -6.2864788e-02, -2.0975524e-04, -5.2859791e-02, -5.8686612e-03],
      dtype=float32)
```

Write your explanation here:

word_emb is a matrix in $\mathbb{R}^{250\times 100}$ which can be treated as a map (embedding) from a 250 dimensional space into a 100 dimensional space, i.e. we "encode" each word in our vocabulary (which presented as a one-hot vector of size 250) as a vector in $\mathbb{R}^{100}$.

1.4.2 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.

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

```
0.24525286
0.1009573
```

Compute the 5 closest words to the following words:

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

[ ]:
```
# Write your code here
word_list = ["four", "go", "what", "should", "school", "your", "yesterday",␣
 ↪"not"]
for word in word_list:
  close_words_idxs = np.argsort(similarities[vocab_stoi[word],:])[-6:-1]
  close_words = [vocab_itos[word_idx] for word_idx in np.flip(close_words_idxs)]
  print("{}: {}".format(word, close_words))
```

```
four: ['five', 'three', 'two', 'many', 'several']
go: ['going', 'come', 'get', 'out', 'same']
what: ['who', 'where', 'how', 'when', 'department']
should: ['could', 'would', 'can', 'might', 'will']
school: ['office', 'first', 'new', 'good', 'every']
your: ['our', 'my', 'other', 'mr.', 'its']
yesterday: ['department', 'today', 'general', 'end', 'ago']
not: ['nt', 'never', 'even', 'yesterday', 'would']
```

1.4.3 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.

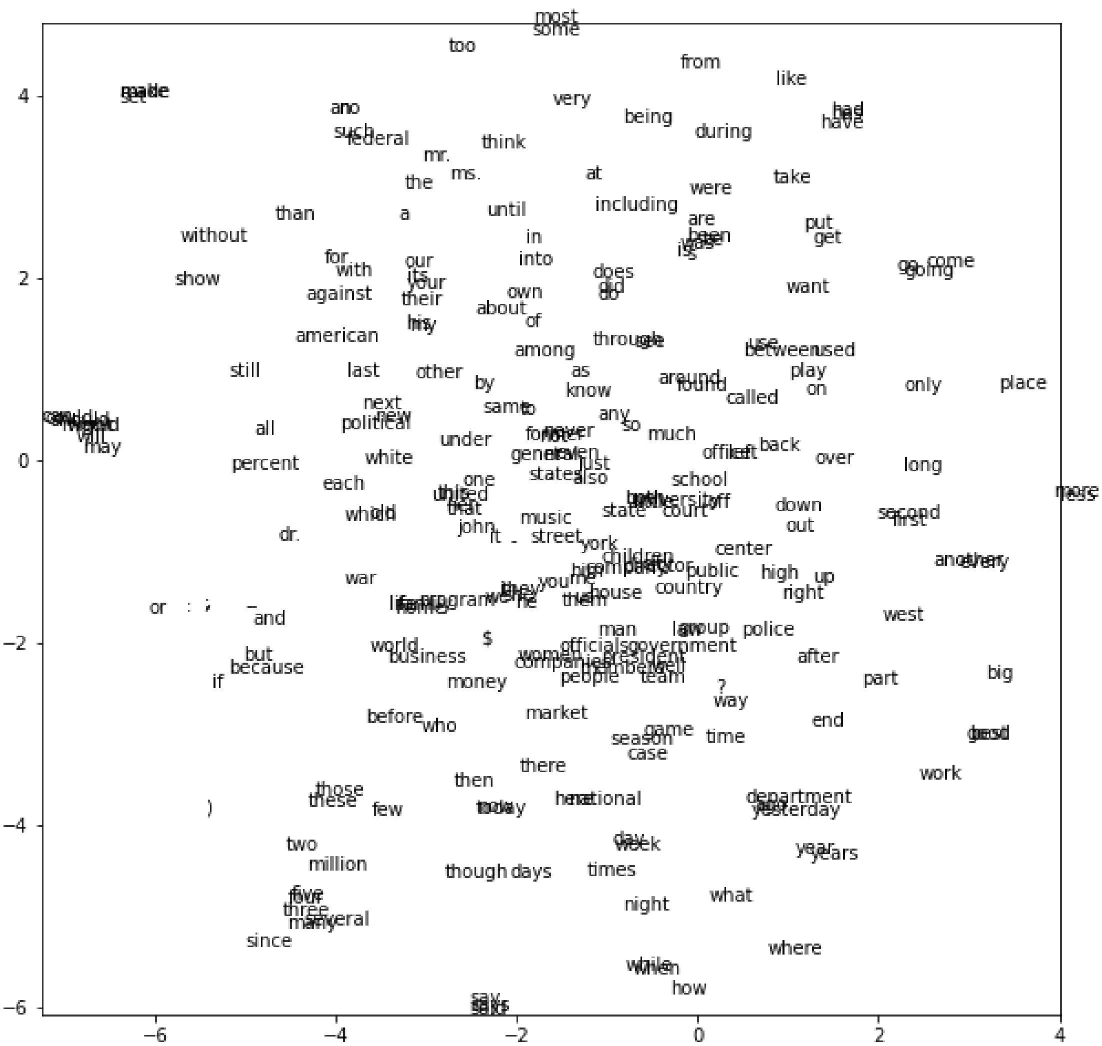
The following code runs the t-SNE algorithm and plots the result.

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.

[ ]:
```
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()
```



Explain and discuss your results here:

By using the t-SNE we can see a visulaizaion of the embedding that was used. We can clearly see that there are some clusters in this represention, for example there are cluster for numbers and relative words (two, three, four, five, million, many, few), a cluster for words related to time (day, week, night, while, when, times) a cluster for possessives adjectives (our, its, your, their, his), etc. To conclude we can see that words with simmilar meaning or a relation in the lower dimention (and also in the higher dimension as explained) are close to each other.