Assignment2

April 18, 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.

[1]:
```
import pandas
import numpy as np
import matplotlib.pyplot as plt
import collections
import itertools
from itertools import repeat

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:

[2]:
```
# Skip this part if running this notebook locally
from google.colab import drive
drive.mount('/content/gdrive')
```

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

Find the path to `raw_sentences.txt`:

[3]:
```
# Local run:
base_path = 'C:/Users/Vova/Documents/BGU/Deep Learning/HW/2/'                    ␣
 ↪#TODO: insert the path to the directory

# Online run:
base_path = '/content/gdrive/My Drive/Intro_to_Deep_Learning/HW2/'

file_path = base_path + '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`.

[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.

[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.

[6]:
```
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.

[7]:
```
for i in range(10):
  print(train[i])
```

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

Answer:

The punctuations and apostrophes have individual indexes and are separated from the main word. The reason for this representation is that we want to analyzer the context, and punctuations can be in a sentence without relation to the context. Also, the most frequent words will be the punctuations. The apostrophes are separated, because they don't change the context of the word, and separated for the same reason.

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.

[8]:
```python
n = 10 # Number of words
print("The " + str(n) + " most common words in the vocabulary:")

# Concatenate into one vector
flat_list = list(itertools.chain(*train))

cont = dict(collections.Counter(flat_list))

# Convert the count into percentage:
for k in cont:
    cont[k] = cont[k]/len(flat_list)*100

# Sort the list
text_list_to_analyze = {k: v for k, v in sorted(cont.items(), key=lambda item:␣
 ↪item[1], reverse=True)}

# Gets n most common
most_common = {k: text_list_to_analyze[k] for k in list(text_list_to_analyze)[:
 ↪n]}

# Print
for count, value in enumerate(most_common):
  print(str(count+1) + ") {" + value + "} percentage: " + str(float("{:.2f}".
 ↪format(most_common[value]))) + "%")
```

```
The 10 most common words in the vocabulary:
1) {.} percentage: 10.7%
2) {it} percentage: 3.85%
3) {,} percentage: 3.25%
4) {i} percentage: 2.94%
5) {do} percentage: 2.69%
```

```
6) {to} percentage: 2.58%
7) {nt} percentage: 2.16%
8) {?} percentage: 2.14%
9) {the} percentage: 2.09%
10) {'s} percentage: 2.09%
```

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.

[9]:
```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()}

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
    new_list = [[] for x in repeat(None, len(sents))]
    for i, line in enumerate(sents):
      for j, word in enumerate(line):
        new_list[i].append(vocab_stoi.get(word))
    return new_list


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

    Example:

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

    "First step: find how many list of list will be in the 4gram output for␣
 ↪initialze the 4gram"
    N = 0;
    for i, line in enumerate(seqs):
      N = N + max(len(line) - 3,0)
    new_list = [[] for x in repeat(None, N)]

    "Secound step: generete 4gram"
    index = 0
    for i, line in enumerate(seqs):
      for j, word in enumerate(line):
        if j < len(line) - 3:
          new_list[index] = (line[j:j+4])
          index = index + 1
          #print("line: {} , word: {}, i: {} , j: {}, index: {}".format(line,␣
 ↪word, i, j, index))
    return new_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)
```

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:

[10]:
```
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 len(np.shape(data)) == 1:
      out_dim = np.concatenate((np.array(np.shape(data)), [250]))
      output = np.zeros(out_dim)
      for i in range(0,len(data)):
        #print(i)
        output[i,data[i]-1] = 1
      return output
    elif len(np.shape(data)) == 2:
      out_dim = np.concatenate((np.array(np.shape(data)), [250]))
      output = np.zeros(out_dim)
      for i in range(0,np.shape(data)[0]):
        for j in range(0,np.shape(data)[1]):
          output[i,j,data[i][j]-1] = 1
      return output


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:

[11]:
```
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])
        inp = self.layer1(inp)
        inp = self.layer2(inp)
        return inp
        # Note that we will be using the nn.CrossEntropyLoss(), which computes␣
 ↪the softmax operation internally, as loss criterion
```

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.

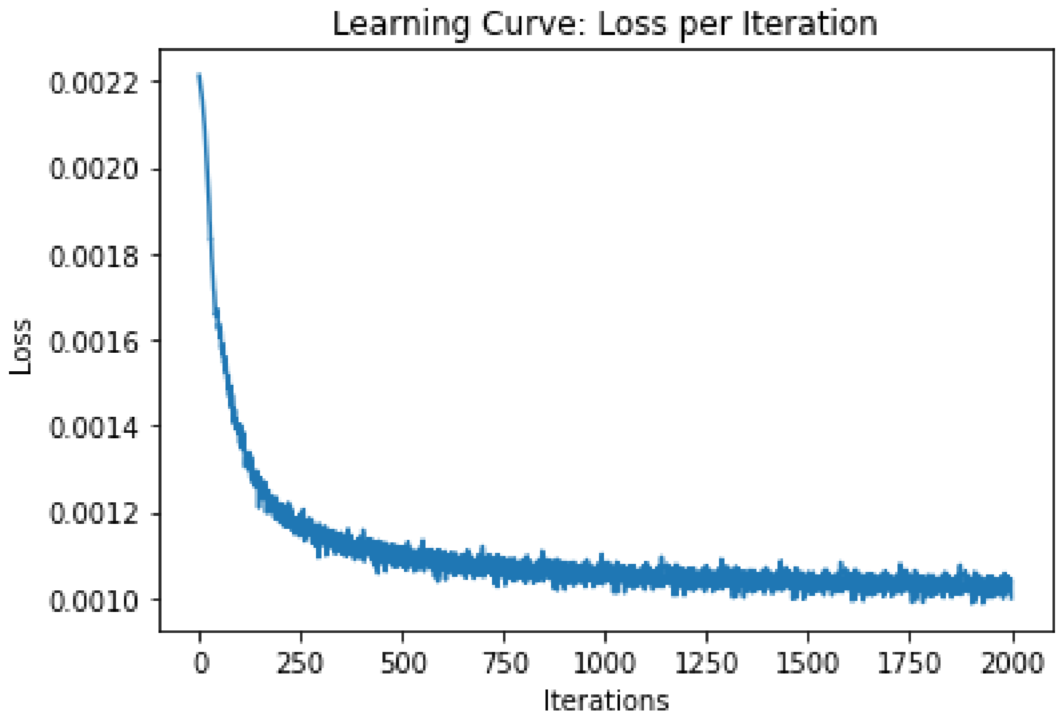
Obtain a training accuracy of at least 35% while changing only the hyperparameters of the train function.

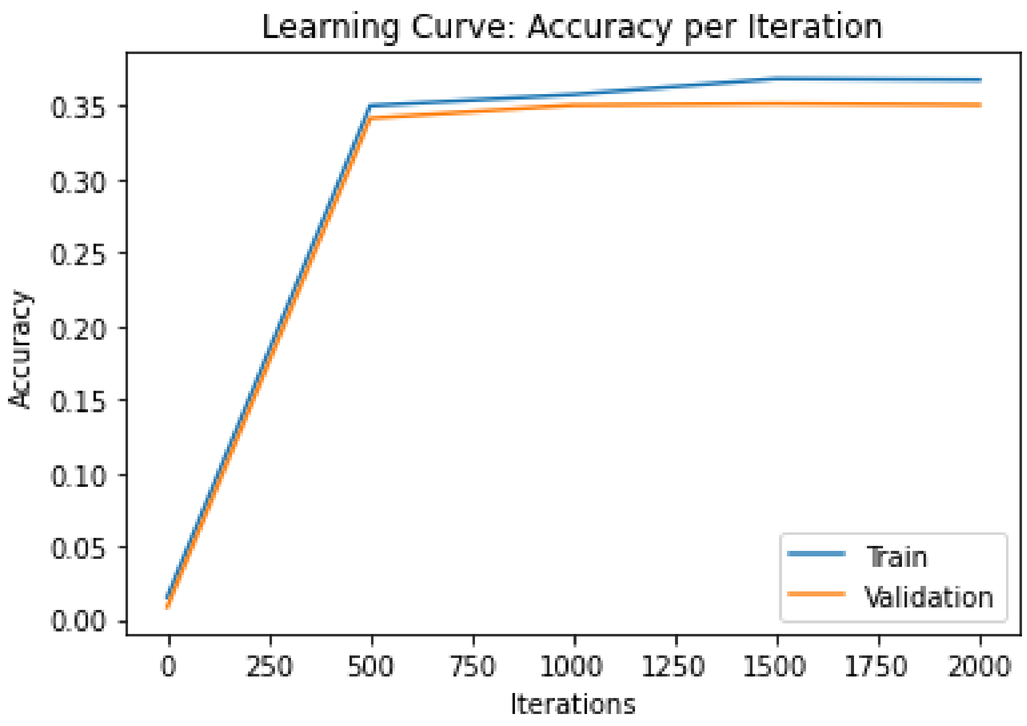
Plot the learning curve using the `plot_learning_curve` function provided to you, and include your plot in your PDF submission.

[12]:
```
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):
    for i in range(0, len(data[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/
↪HW2/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/HW2/mlp/ckpt-500.pk
    - /content/gdrive/My Drive/Intro_to_Deep_Learning/HW2/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/HW2/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()
```

[30]:
```
# Train Model
pytorch_mlp = PyTorchMLP()
learning_curve_info = run_pytorch_gradient_descent(pytorch_mlp,
                                                   train_data=train4grams,
                                                   validation_data=valid4grams,
                                                   batch_size=2500,
                                                   learning_rate=0.001,
                                                   weight_decay=1e-7,
                                                    max_iters=2000,
                                                    checkpoint_path = None)     ␣
↪# To save the model, change None to: base_path + 'mlp/ckpt-{}.pk'

plot_learning_curve(*learning_curve_info)
```

```
Iter 0. [Val Acc 1%] [Train Acc 2%, Loss 5.531074]
Iter 500. [Val Acc 34%] [Train Acc 35%, Loss 2.726658]
Iter 1000. [Val Acc 35%] [Train Acc 36%, Loss 2.620278]
Iter 1500. [Val Acc 35%] [Train Acc 37%, Loss 2.599512]
Iter 2000. [Val Acc 35%] [Train Acc 37%, Loss 2.496889]
```





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.

[15]:
```python
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
    wrd_ind = convert_words_to_indices([sentence])
    wrd_ind_oneshot = make_onehot(wrd_ind)
    pred_mat = model(torch.Tensor(wrd_ind_oneshot))
    return vocab_itos[int(pred_mat.argmax())]
```

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.

[ ]:
```
# Load model (uncomment this part to see the original model)
#pytorch_mlp = PyTorchMLP()
#pytorch_mlp.load_state_dict(torch.load(base_path + 'mlp/pytorch_mlp/ckpt-2000.
↪pk'))
```

[ ]:
```
<All keys matched successfully>
```

[16]:
```
# This part prints the completed sentences of the model
test_sentence_list = ["you are a", "few companies show", "there are no",␣
↪"yesterday i was", "the game had", "yesterday the federal"]
for idx in range(0,len(test_sentence_list)):
  sen = test_sentence_list[idx].split() # get one sentence from list
  pred = make_prediction_torch(pytorch_mlp, sen)
  sen.append(pred)
  print(str(idx+1) + ") " +' '.join(sen))
```

```
1) you are a good
2) few companies show .
3) there are no other
4) yesterday i was nt
5) the game had to
6) yesterday the federal government
```

Answer:

Predictions of model: 1. you are a **"good"** 2. few companies show **"."** 3. there are no **"other"** 4. yesterday i was **"nt"** 5. the game had **"to"** 6. yesterday the federal **"government"**

Explanation:

Most of the predictions make sense, as not fully sentence. The models with different hyperparameters have almost identical results, although the variance of the hyperparameters values (For example: smaller batch size vs larger batch size). For sentence 6, the prediction is always "government". In the dataset, the word "government" follows the word "federal" in many of the sentences. Therefore, the model "memorized" this data. Memorizing is overfitting. Sentence 2 shown in many model prediction ".", as the end of the sentence, although logically, the sentence is incomplete and it is missing words afterwards. This sentence is illogical and a bad prediction in form of linguistics.

1.2.5 Part (e) -- 4%

Report the test accuracy of your model

[32]:
```
# This section wouldn't work if the model wasn't trained (requires␣
 ↪learning_curve_info).
print("The actual train accuracy is " + str(learning_curve_info[-2][-1]*100)␣
 ↪+"%")

est_acc = estimate_accuracy_torch(pytorch_mlp, test4grams, batch_size=5000,␣
 ↪max_N=100000)
print("The estimated test accuracy is " + str(est_acc*100) +"%")
```

```
The actual train accuracy is 36.78%
The estimated test accuracy is 34.160000000000004%
```

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.

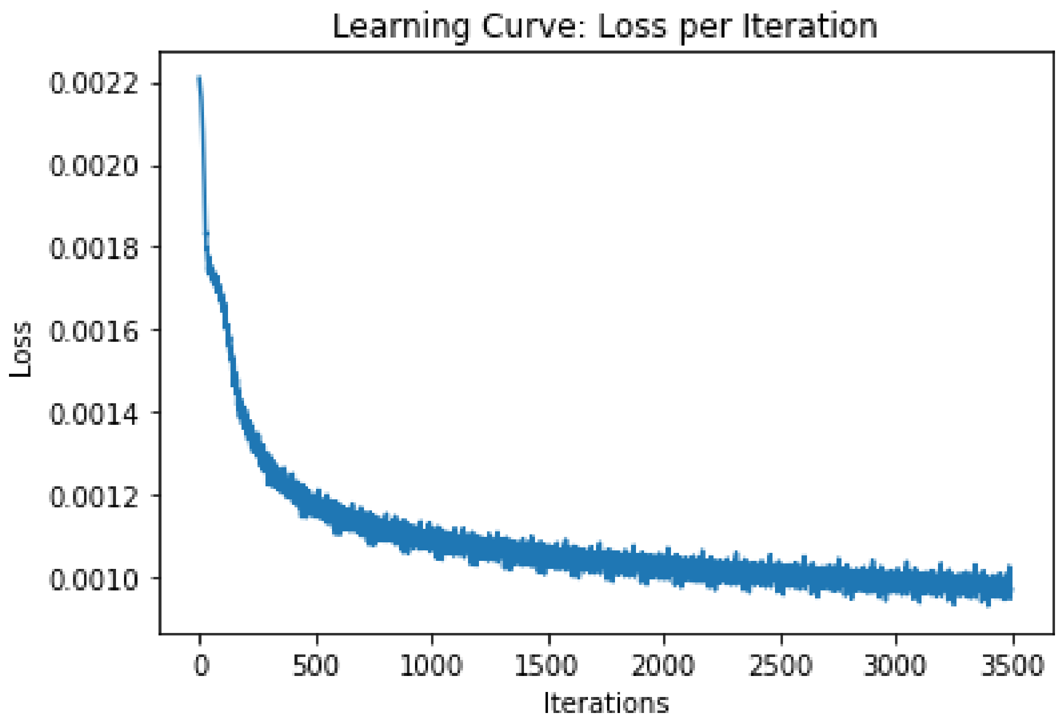
1.3.1 Part (a) -- 8%

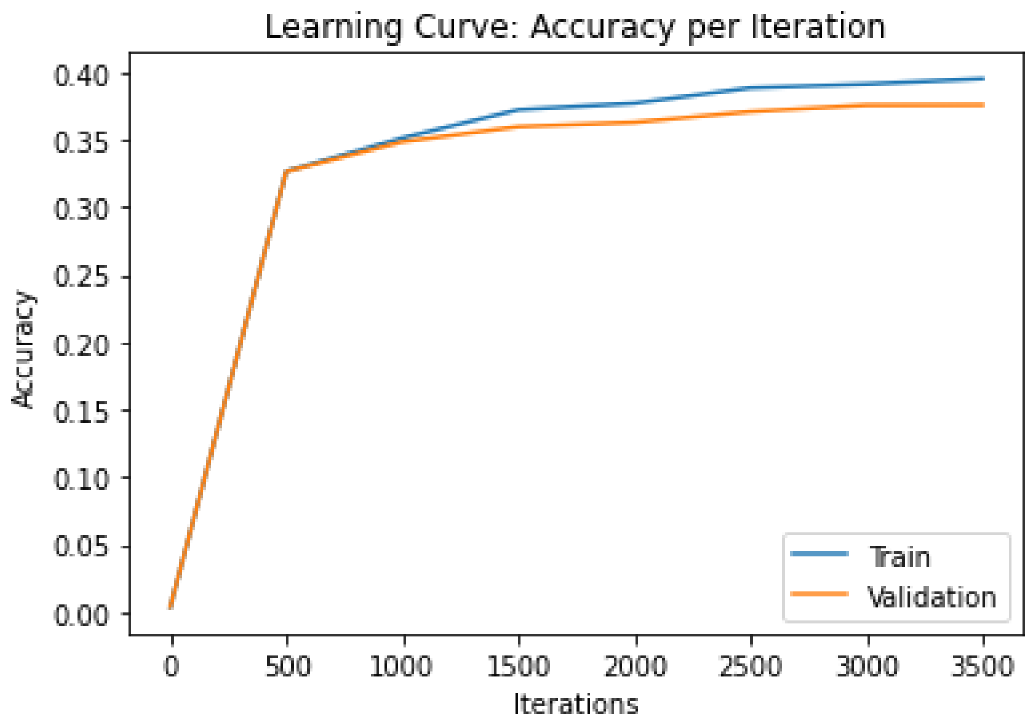
The PyTorch model is implemented for you. Use `run_pytorch_gradient_descent` to train your PyTorch MLP model to obtain a training accuracy of at least 38%. Plot the learning curve using the `plot_learning_curve` function provided to you, and include your plot in your PDF submission.

[25]:
```
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)
```

[26]:
```
# Train Model
pytorch_wordemb = PyTorchWordEmb()
learning_curve_info = run_pytorch_gradient_descent(pytorch_wordemb,
                                                    train_data=train4grams,
                                                    validation_data=valid4grams,
                                                    batch_size=2500,
                                                    learning_rate=0.001,
                                                    weight_decay=1e-7,
                                                    max_iters=3500,
                                                    checkpoint_path = None)     ␣
 ↪# To save the model, change None to: base_path + 'mlp/ckpt-{}.pk'

plot_learning_curve(*learning_curve_info)
```

```
Iter 0. [Val Acc 0%] [Train Acc 0%, Loss 5.522506]
Iter 500. [Val Acc 33%] [Train Acc 33%, Loss 2.920030]
Iter 1000. [Val Acc 35%] [Train Acc 35%, Loss 2.717283]
Iter 1500. [Val Acc 36%] [Train Acc 37%, Loss 2.636476]
Iter 2000. [Val Acc 36%] [Train Acc 38%, Loss 2.489208]
Iter 2500. [Val Acc 37%] [Train Acc 39%, Loss 2.577016]
Iter 3000. [Val Acc 38%] [Train Acc 39%, Loss 2.496780]
Iter 3500. [Val Acc 38%] [Train Acc 40%, Loss 2.424743]
```





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%.

[ ]:
```
# Load model
#pytorch_wordemb= PyTorchWordEmb()
#pytorch_wordemb.load_state_dict(torch.load(base_path + '/mlp/pytorch_wordemb/
↪ckpt-3500.pk'))
```

[ ]:
```
<All keys matched successfully>
```

[27]:
```
# This part prints the completed sentences of the model
test_sentence_list = ["You are a", "few companies show", "There are no",␣
↪"yesterday i was", "the game had", "yesterday the federal"] # Define␣
↪sentences
test_sentence_list = [x.lower() for x in test_sentence_list] # lower case all␣
↪the sentences
for idx in range(0,len(test_sentence_list)):
  sen = test_sentence_list[idx].split() # get one sentence from list
  pred = make_prediction_torch(pytorch_wordemb, sen)
  sen.append(pred)
  print(str(idx+1) + ") " +' '.join(sen))
print()
```

```
1) you are a good
2) few companies show up
3) there are no other
4) yesterday i was nt
5) the game had to
6) yesterday the federal government
```

Explanation:
the results are:

1. you are a **"good"**
2. few companies show **"up"**
3. there are no **"other"**
4. yesterday i was **"nt"**
5. the game had **"to"**
6. yesterday the federal **"government"**

Most of the sentences get similar results as the previous model. Unlike the previous model, the Word Embedd model predicts the word for sentence 2 differently and more precisely. Therefore, now the predicted word makes much more sense in the sentence. This is because the Embedd model is more accurate model than the previous model. Same as in the previous model, we encountered overfitting in sentences 1, 4 and 6.

1.3.3 Part (c) -- 4%

Report the test accuracy of your model

[29]:
```
# This section wouldn't work if the model wasn't trained (requires␣
↪learning_curve_info).
print("The actual train accuracy is " + str(learning_curve_info[-2][-1]*100)␣
↪+"%")


est_acc = estimate_accuracy_torch(pytorch_wordemb, test4grams, batch_size=5000,␣
↪max_N=100000)
```

```
print("The estimated test accuracy is "+ str(est_acc*100) +"%")
```

```
The actual train accuracy is 39.54%
The estimated test accuracy is 37.32%
```

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".

[33]:
```
word_emb_weights = list(pytorch_wordemb.word_emb_layer.parameters())[0]
word_emb = word_emb_weights.detach().numpy().T
```

Explanation:

The embedding layer generates the transformation between the one-hot presentation of the word to a new vector in 100-dimension space. The new vector is seemingly random, but it is actually representation of the input word on the new 100-dimension space. The embedding layer is just a transformation of the word, so the embeded vector is exactly the representation of the input word.

The input layer is an one-hot representation, also known as vector that all his element are zero besides one element with the value one. The feedforward through emb_layer can be described as matrix multiplication: **v** * **W** where **v** is the input vector in size of 1 x 250 and **W** is the matrix weights in size of 250 x 100. Because of the structure of **v** the result of the multiplication is a row from **W** - 1 X 100 size vector in the 100-dimension space.

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 **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.

[34]:
```
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']])
# This example should show a result close to 1:
print(similarities[vocab_stoi['any'], vocab_stoi['any']])
```

```
0.23553647
0.15589279
1.0
```

Compute the 5 closest words to the following words:

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

[35]:
```
# Write your code here
base_words = ["four", "go", "what", "should", "school", "your", "yesterday",␣
 ↪"not"]
for base_word in base_words:
  wrd_lst = np.zeros(0)
  for i in range(0,len(vocab_itos)):
    wrd_lst = np.append(wrd_lst,similarities[vocab_stoi[base_word], i])

  wrd_lst = np.nan_to_num(wrd_lst, nan=-9999)

  max_list = []
  wrd_lst[np.argmax(wrd_lst)]=0
  for i in range(0,5):
    max_list.append(np.argmax(wrd_lst))
    wrd_lst[max_list[i]]=0

  print('Five words that are similar to "' + base_word +'":')
  for i in range(0,len(max_list)):
      print(str(i+1) + ") " + str(vocab_itos[max_list[i]]))
  print()
```

```
Five words that are similar to "four":
1) are
2) about
3) political
4) two
5) our
```

```
Five words that are similar to "go":
1) get
2) was
3) by
4) both
5) few

Five words that are similar to "what":
1) i
2) house
3) through
4) when
5) west

Five words that are similar to "should":
1) second
2) long
3) former
4) each
5) us

Five words that are similar to "school":
1) years
2) want
3) companies
4) up
5) from

Five words that are similar to "your":
1) never
2) might
3) did
4) good
5) put

Five words that are similar to "yesterday":
1) good
2) to
3) some
4) both
5) off

Five words that are similar to "not":
1) them
2) government
3) me
4) her
5) make
```

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.

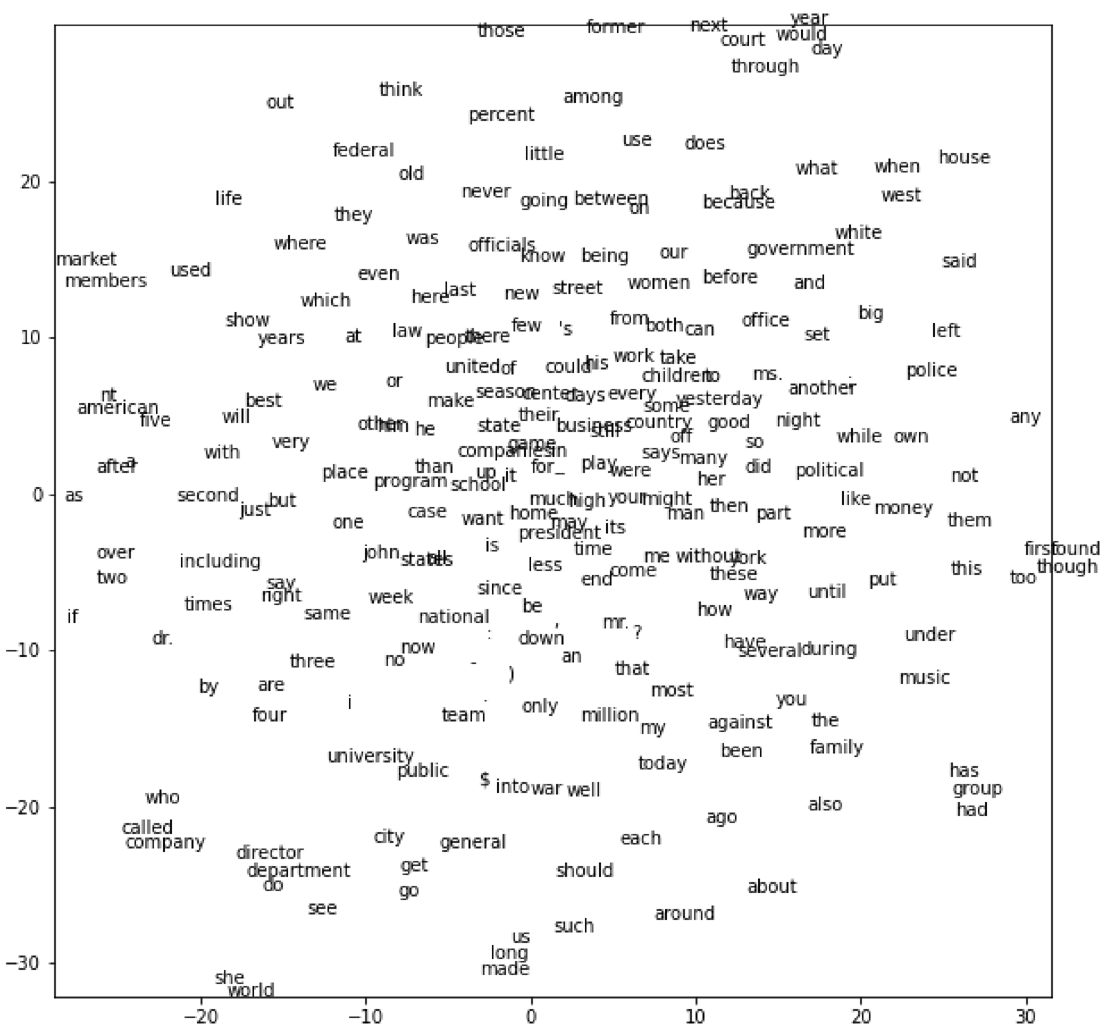
The following code runs the t-SNE algorithm and plots the result.

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.

[36]:
```python
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()
```



Explanation and discussion of the results:

The input vector (one-hot presentation) represents multi-dimensional space that each axis is a word from the dictionary of the network. Therefore, all the words in this space are uncorrelated.

The embedding layer generates new space that the words in it are correlated. The correlation created according to the training procedure (training set, loss function and initial hyperparameter). With the help of this correlation, the network can study similarity between different words. It is noticeable that there are similar words that are group together, like: 1. Punctuation: { : ,) , - , ' , . } 2. Question words: { when, what } & also the word {which} as we calculate in Part (B)

[ ]:
```
# This part used to save this notebook as PDF.
%cd /content/gdrive/MyDrive/HW2/
!sudo apt-get install texlive-xetex texlive-fonts-recommended␣
 ↪texlive-generic-recommended
!jupyter nbconvert --to pdf Assignment1.ipynb
```