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.

In [33]:

```python
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 [34]:

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

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

Find the path to `raw_sentences.txt` :

In [35]:

```python
file_path = '/content/gdrive/My Drive/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 [36]:

```
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 [37]:

```
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 [38]:

```
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 [ ]:

```
# Your code goes here
for i in range(10):
  print(' '.join(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 .
```

Write your answers here:

The punctuations are treated in our word representation as words. The punctuations in the sentences can be found in the vocabulary 'vocab' that we defined above, and they are: [')', ',', '-', '--', '.', ':', ';', '?'].

words with apostrophes are slipted into 2 words: the word and its ending with the apostrophe. For example, we present two sentences from the dataset in which how words with apostrophes are represented:

['So', 'it', "'s", 'not', 'going', 'to', 'get', 'in', 'my', 'way', '.'] => "it" and "'s" are reprsenting "it's".

['There', "'s", "still" ,'time', 'for', 'them', 'to', 'do', 'it', '.'] => "There" and "'s" are reprsenting "There's".

We saw that the vocabulary 'vocab' contains only at least one apostrophe and its ending is :["'s"] (second position in our vocab)

To sum up: punctuations they are treated as words, and words with apostrophes are split them into two words where one is the word and the other is the ending of the word.

Part (b) -- 4%

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

These are useful quantities to compute, because one of the first things a machine learning model will learn is to predict the **most common** class. Getting a sense of the distribution of our data will help you understand our model's behaviour.

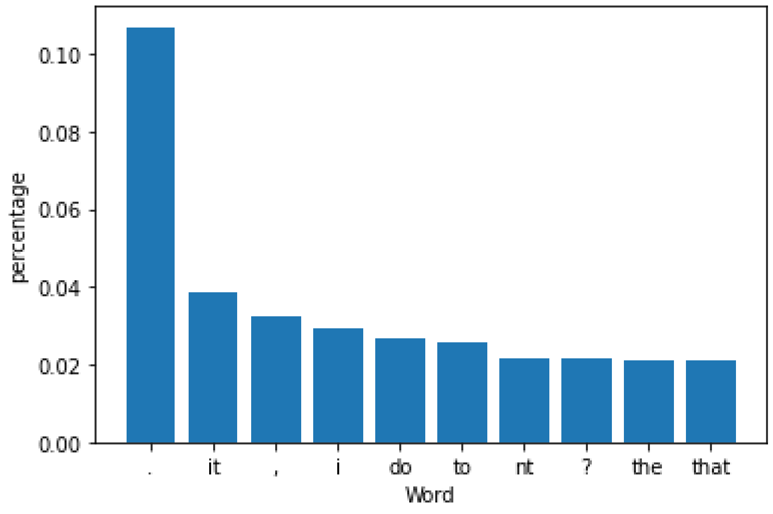
You can use Python's `collections.Counter` class if you would like to.

In [ ]:

```
# Your code goes

# Vocabulary
import collections
flat_list = [item for sublist in sentences for item in sublist]
common_words_voc = collections.Counter(flat_list).most_common(10)
print('10 most common words in the vocabulary')
print('--------------------------------------')
for idx, (word, num) in enumerate(common_words_voc):
    print(f'{word} with {num} repetitions')
print('\n')

# Train
flat_train = [item for sublist in train for item in sublist]
Total_mum_of_words_train = len(flat_train)
train_words_rep = collections.Counter(flat_train)
print('measure frequency of 10 common words in the vocabulary on the training set')
print('----------------------------------------------------------------------------')
words_list = [] # will conatain 10 most common words in vocab
meas_rep = [] # will conatain the frequency of 10 common words in the vocabulary on the tra
for i in range(10):
  word = common_words_voc[i][0]
  words_list.append(word)
  rep = train_words_rep[word]/Total_mum_of_words_train
  meas_rep.append(rep)
  print(f'{word} with {train_words_rep[word]} repetitions that is {100*rep}%')

# Visualization
print('\n')
print('Visual presentation of the results')
x = np.arange(10)
plt.bar(x, height=meas_rep)
plt.xticks(x, words_list)
plt.xlabel('Word')
plt.ylabel('percentage')
plt.show()
```

```
10 most common words in the vocabulary
--------------------------------------
. with 80974 repetitions
it with 29200 repetitions
, with 24583 repetitions
i with 22267 repetitions
do with 20245 repetitions
to with 19537 repetitions
nt with 16460 repetitions
? with 16210 repetitions
the with 15939 repetitions
that with 15795 repetitions



measure frequency of 10 common words in the vocabulary on the training set
---------------------------------------------------------------------------
-
. with 64297 repetitions that is 10.695720015237537%
it with 23118 repetitions that is 3.8456484021379134%
, with 19537 repetitions that is 3.2499538382458866%
i with 17684 repetitions that is 2.9417097648328947%
```

```
do with 16181 repetitions that is 2.6916877236349843%
to with 15490 repetitions that is 2.576740797176065%
nt with 13009 repetitions that is 2.164029763102868%
? with 12881 repetitions that is 2.1427371341784953%
the with 12583 repetitions that is 2.0931652324639396%
that with 12535 repetitions that is 2.0851804966173%


Visual presentation of the results
```



Part (c) -- 11%

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

Complete the helper functions `convert_words_to_indices` and `generate_4grams` , so that the function `process_data` will take a list of sentences (i.e. list of list of words), and generate an $N \times 4$ numpy matrix containing indices of 4 words that appear next to each other, where N is the number of 4grams (sequences of 4 words appearing one after the other) that can be found in the complete list of sentences. Examples of how these functions should operate are detailed in the code below.

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

In [ ]:

```
# A list of all the words in the data set. We will assign a unique
# identifier for each of these words.
vocab = sorted(list(set([w for s in train for w in s])))
# A mapping of index => word (string)
vocab_itos = dict(enumerate(vocab))
# A mapping of word => its index
vocab_stoi = {word:index for index, word in vocab_itos.items()}

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

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

    Indices_list = []
    for i in range(len(sents)):
      indices_list = []
      sen = sents[i]
      for word in sen:
        indices_list.append(vocab_stoi[word])
      Indices_list.append(indices_list)
    return Indices_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]]
    """

    # Write your code here
    List_4grams = []
    for _, sub_list in enumerate(seqs):
      if len(sub_list) > 3:
        for i in range(len(sub_list) % 4 + 1):
          List_4grams.append(sub_list[i:i+4])
    return List_4grams



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)
```

In [ ]:

```
# Checking our function convert_words_to_indices
convert_words_to_indices([['one', 'in', 'five', 'are', 'over', 'here'],
                          ['other', 'one', 'since', 'yesterday'], ['you']])
```

Out[28]:

```
[[148, 98, 70, 23, 154, 89], [151, 148, 181, 246], [248]]
```

In [ ]:

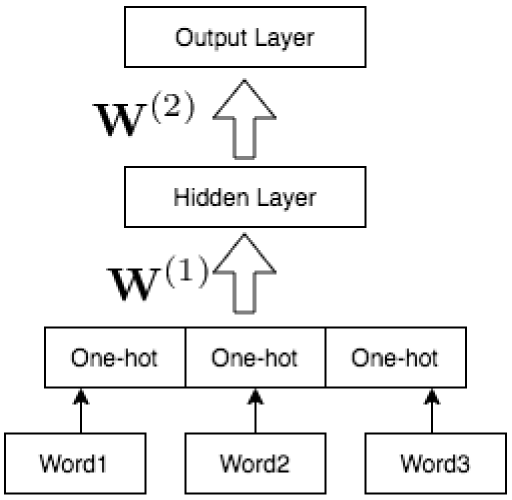
```

# Checking our function generate_4grams
generate_4grams([[148, 98, 70, 23, 154, 89], [151, 148, 181, 246], [248]])
```

Out[29]:

```
[[148, 98, 70, 23], [98, 70, 23, 154], [70, 23, 154, 89], [151, 148, 181, 24
6]]
```

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

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



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

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

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

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

In [ ]:

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

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

    # Write your code here
    batch_size = len(data)
    Num_of_classes = 250
    output = []
    for idx, indicies_for_specific_row in enumerate(data):
      output.append(np.eye(Num_of_classes)[indicies_for_specific_row])
    return np.reshape(output, (batch_size, -1, Num_of_classes))



def get_batch(data, range_min, range_max, onehot=True):
    """
    Convert one batch of data in the form of 4-grams into input and output
    data and return the training data (xt, st) where:
     - `xt` is an numpy array of one-hot vectors of shape [batch_size, 3, 250]
     - `st` is either
            - a numpy array of shape [batch_size, 250] if onehot is True,
            - a numpy array of shape [batch_size] containing indicies otherwise

    Preconditions:
     - `data` is a numpy array of shape [N, 4] produced by a call
        to `process_data`
     - range_max > range_min
    """
    xt = data[range_min:range_max, :3]
    xt = make_onehot(xt)
    st = data[range_min:range_max, 3]
    if onehot:
        st = make_onehot(st).reshape(-1, 250)
    return xt, st
```

Part (a) -- 8%

We build the model in PyTorch. Since PyTorch uses automatic differentiation, we only need to write the *forward pass* of our model.

Complete the `forward` function below:

In [ ]:

```
class PyTorchMLP(nn.Module):
    def __init__(self, num_hidden=400):
        super(PyTorchMLP, self).__init__()
        self.layer1 = nn.Linear(750, num_hidden)
        self.layer2 = nn.Linear(num_hidden, 250)
        self.relu = nn.ReLU()
        self.num_hidden = num_hidden
    def forward(self, inp):
        inp = inp.reshape([-1, 750])
        z1 = self.layer1(inp)
        a1 = self.relu(z1)
        z2 = self.layer2(a1)
        a2 = self.relu(z2)
        return a2
        # 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 [42]:

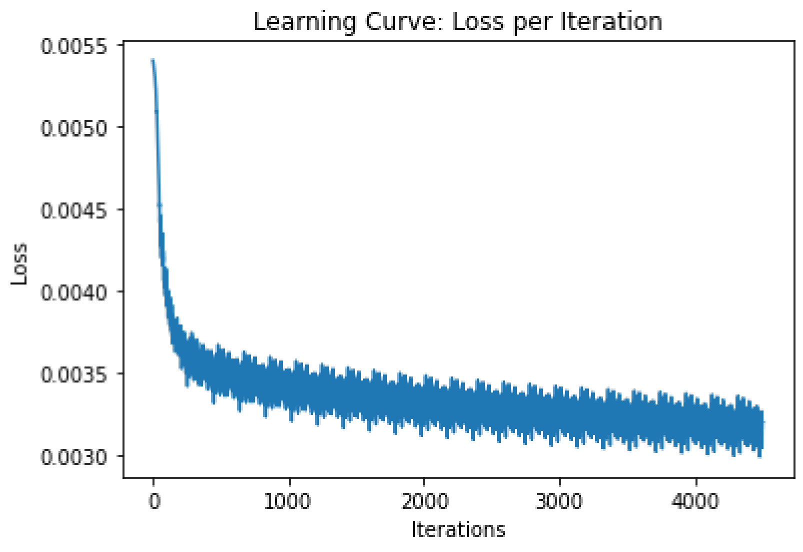
```
def estimate_accuracy_torch(model, data, batch_size=5000, max_N=100000):
    """
    Estimate the accuracy of the model on the data. To reduce
    computation time, use at most `max_N` elements of `data` to
    produce the estimate.
    """
    correct = 0
    N = 0
    for i in range(0, data.shape[0], 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-{}.

    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 wi

    - /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/m

    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()

            # compute prediction logit
            zs = model(xt)
            # compute the total loss
            loss = criterion(zs, st)
            # zero the gradients before we calc our gradients is a clean up step
            # for PyTorch
            optimizer.zero_grad()
            # backward pass to compute the gradient of loss with respect to our
            # learnable params
            loss.backward()
            # make the updates for each parameter with our optimizer that we
            # define earlier (Adam)
            optimizer.step()

            # 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
```

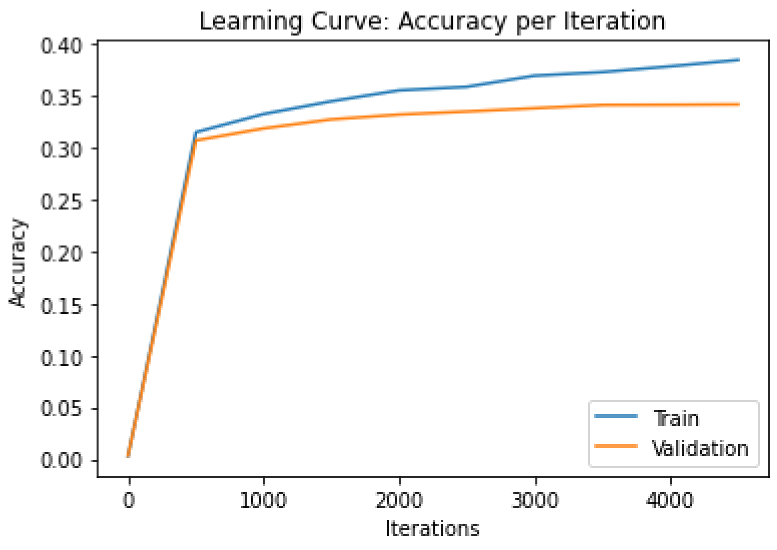
```
def plot_learning_curve(iters, losses, iters_sub, train_accs, val_accs):
    """
    Plot the learning curve.
    """
    plt.title("Learning Curve: Loss per Iteration")
    plt.plot(iters, losses, label="Train")
    plt.xlabel("Iterations")
    plt.ylabel("Loss")
    plt.show()
    plt.title("Learning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

```
pytorch_mlp = PyTorchMLP()
learning_curve_info = run_pytorch_gradient_descent(pytorch_mlp,batch_size=1024,
                                                   max_iters=4500)
plot_learning_curve(*learning_curve_info)
```

```
Iter 0. [Val Acc 0%] [Train Acc 0%, Loss 5.525034]
Iter 500. [Val Acc 31%] [Train Acc 31%, Loss 3.549820]
Iter 1000. [Val Acc 32%] [Train Acc 33%, Loss 3.514182]
Iter 1500. [Val Acc 33%] [Train Acc 34%, Loss 3.396918]
Iter 2000. [Val Acc 33%] [Train Acc 36%, Loss 3.258571]
Iter 2500. [Val Acc 33%] [Train Acc 36%, Loss 3.452786]
Iter 3000. [Val Acc 34%] [Train Acc 37%, Loss 3.389115]
Iter 3500. [Val Acc 34%] [Train Acc 37%, Loss 3.398278]
Iter 4000. [Val Acc 34%] [Train Acc 38%, Loss 3.212755]
Iter 4500. [Val Acc 34%] [Train Acc 38%, Loss 3.280239]
```





We changed the hyperparameters of the function run_pytorch_gradient_descent in order to achieve better results. The hyperparameters that we change is max_iters because 1000 was not enough, another thing that we change is batch_size to 1024 (for batch_size equal to 100 (default) we get a pretty strong noise during the training and we wanted to smooth it so we increased its size to 1024).

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 [ ]:

```
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
    inp = []
    inp.append(sentence)

    indices = convert_words_to_indices(inp)
    # convert to tensor before operate our model
    xt = torch.Tensor(make_onehot(indices))
    # operate our model
    output = model(xt)
    # taking argmax of output will give the index of our predicted word in vocab
    # because the index will contain the highest value before activating softmax
    # it will also have the highest value after activating softmax
    # (exp is monotonically rising)

    argmax_index = torch.argmax(output).item()
    predicted_word = vocab_itos[argmax_index]
    return predicted_word
```

In [ ]:

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

Out[25]:

```
'good'
```

Part (d) -- 10%

Use your code to predict what the next word should be in each of the following sentences:

- "You are a"
- "few companies show"
- "There are no"
- "yesterday i was"
- "the game had"
- "yesterday the federal"

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 [ ]:

```python
# Write your code here

input = [['you', 'are', 'a'], ['few', 'companies', 'show'],
         ['there', 'are', 'no'], ['yesterday', 'i', 'was'],
         ['the', 'game', 'had'], ['yesterday', 'the', 'federal']]
for idx, sentence in enumerate(input):
  pred_next_word = make_prediction_torch(pytorch_mlp, sentence)
  print(f'The next word of the sentence {sentence} is: {pred_next_word}')
```

```
The next word of the sentence ['you', 'are', 'a'] is: good
The next word of the sentence ['few', 'companies', 'show'] is: the
The next word of the sentence ['there', 'are', 'no'] is: other
The next word of the sentence ['yesterday', 'i', 'was'] is: nt
The next word of the sentence ['the', 'game', 'had'] is: to
The next word of the sentence ['yesterday', 'the', 'federal'] is: day
```

Write your answers here: We did not encounter the effect that the predicted words for all the sentences are the same. Therefore, combined with the fact the accuracy plot of the validation did not show a decrease in performance, we can conclude that the model was not over-fitted. To check if our model memorizes our data we checked if the combination of the four words above for each sentence appears in our dataset, what came out is that none of them appears except the first sentence (it can be seen that the sentence 'you are a good man' is in the database) so we conclude that our model does not memorize the sequence from the dataset.

Another interesting thing to note is that our dataset is unbalanced meaning not every word appears in the same amount (this can be seen in the first question in the first part). Therefore there is a higher probability that the more common words are those that will be at the origin of the model in most cases. That is, there is a chance that the model will pred a word that is high frequency even though it has no logical connection to the sentence we want to predict. One way to handle it might be using undersampling technique, or using different weights for classes in the loss function.

Part (e) -- 6%

Report the test accuracy of your model

In [ ]:

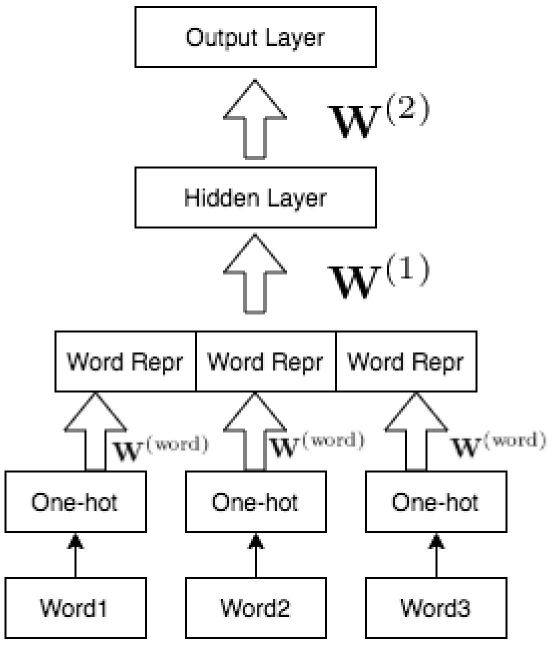
```
# Write your code here
test_acc = estimate_accuracy_torch(pytorch_mlp, test4grams)
print("Test accuracy is {:.4f}%".format(test_acc * 100))
```

```
Test accuracy is 32.4335%
```

Question 3. Learning Word Embeddings (24 %)

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

Our model will look like this:



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

Part (a) -- 10%

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.

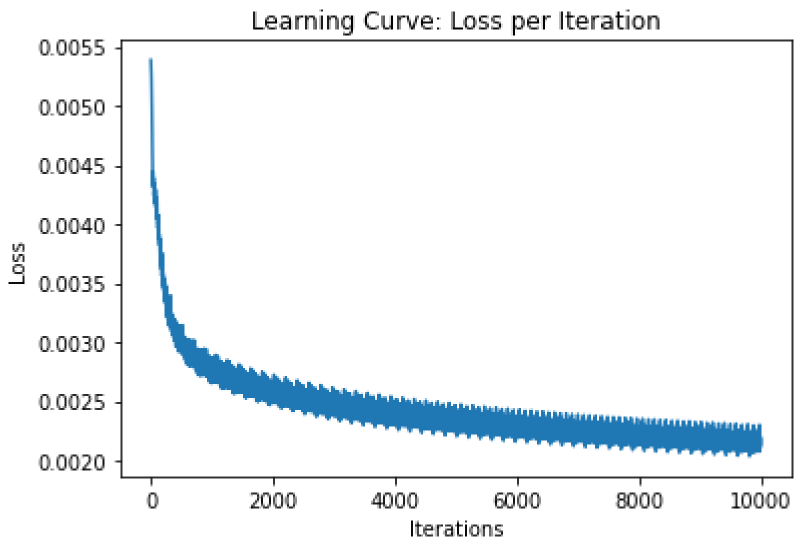
In [40]:

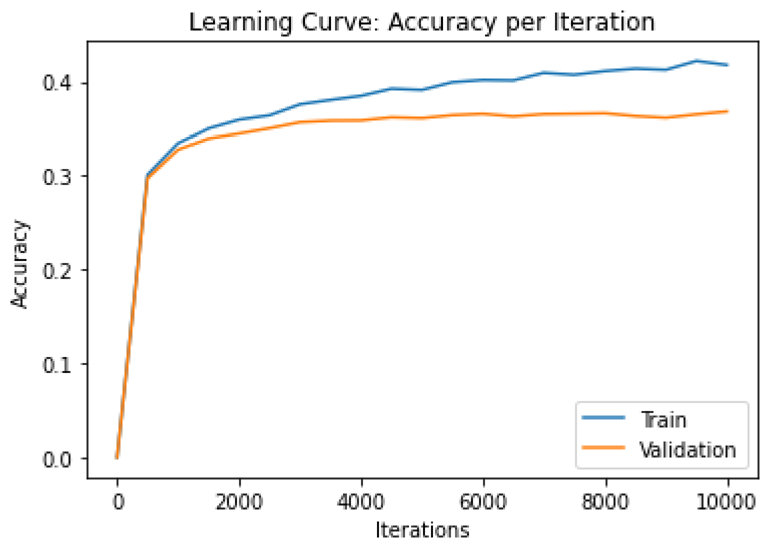
```
class PyTorchWordEmb(nn.Module):
    def __init__(self, emb_size=100, num_hidden=300, vocab_size=250):
        super(PyTorchWordEmb, self).__init__()
        self.word_emb_layer = nn.Linear(vocab_size, emb_size, bias=False)
        self.fc_layer1 = nn.Linear(emb_size * 3, num_hidden)
        self.fc_layer2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

In [45]:

```
pytorch_wordemb= PyTorchWordEmb()
result = run_pytorch_gradient_descent(pytorch_wordemb,max_iters=10000,batch_size=1024)
plot_learning_curve(*result)
```

```
Iter 0. [Val Acc 0%] [Train Acc 0%, Loss 5.517323]
Iter 500. [Val Acc 30%] [Train Acc 30%, Loss 2.993967]
Iter 1000. [Val Acc 33%] [Train Acc 33%, Loss 2.855276]
Iter 1500. [Val Acc 34%] [Train Acc 35%, Loss 2.671486]
Iter 2000. [Val Acc 34%] [Train Acc 36%, Loss 2.573308]
Iter 2500. [Val Acc 35%] [Train Acc 36%, Loss 2.528945]
Iter 3000. [Val Acc 36%] [Train Acc 38%, Loss 2.594248]
Iter 3500. [Val Acc 36%] [Train Acc 38%, Loss 2.562563]
Iter 4000. [Val Acc 36%] [Train Acc 38%, Loss 2.407380]
Iter 4500. [Val Acc 36%] [Train Acc 39%, Loss 2.499310]
Iter 5000. [Val Acc 36%] [Train Acc 39%, Loss 2.389215]
Iter 5500. [Val Acc 36%] [Train Acc 40%, Loss 2.331805]
Iter 6000. [Val Acc 37%] [Train Acc 40%, Loss 2.357832]
Iter 6500. [Val Acc 36%] [Train Acc 40%, Loss 2.365462]
Iter 7000. [Val Acc 37%] [Train Acc 41%, Loss 2.272843]
Iter 7500. [Val Acc 37%] [Train Acc 41%, Loss 2.289229]
Iter 8000. [Val Acc 37%] [Train Acc 41%, Loss 2.271743]
Iter 8500. [Val Acc 36%] [Train Acc 41%, Loss 2.189012]
Iter 9000. [Val Acc 36%] [Train Acc 41%, Loss 2.185272]
Iter 9500. [Val Acc 37%] [Train Acc 42%, Loss 2.238309]
Iter 10000. [Val Acc 37%] [Train Acc 42%, Loss 2.168796]
```





Part (b) -- 10%

Use the function `make_prediction` that you wrote earlier to predict what the next word should be in each of the following sentences:

- "You are a"
- "few companies show"
- "There are no"
- "yesterday i was"
- "the game had"
- "yesterday the federal"

How do these predictions compared to the previous model?

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

Just like before, if you encounter overfitting, train your model for more iterations, or change the hyperparameters in your model. You may need to do this even if your training accuracy is >=38%.

In [ ]:

```
# Write your code here

input = [['you', 'are', 'a'], ['few', 'companies', 'show'],
         ['there', 'are', 'no'], ['yesterday', 'i', 'was'],
         ['the', 'game', 'had'], ['yesterday', 'the', 'federal']]
for idx, sentence in enumerate(input):
  pred_next_word = make_prediction_torch(pytorch_wordemb, sentence)
  print(f'The next word of the sentence {sentence} is: {pred_next_word}')
```

```
The next word of the sentence ['you', 'are', 'a'] is: good
The next word of the sentence ['few', 'companies', 'show'] is: about
The next word of the sentence ['there', 'are', 'no'] is: people
The next word of the sentence ['yesterday', 'i', 'was'] is: nt
The next word of the sentence ['the', 'game', 'had'] is: to
The next word of the sentence ['yesterday', 'the', 'federal'] is: government
```

Write your explanation here: As we explained earlier since all of the predictions are reasonable, and the accuracy on the validation set was decreased during the training, we think that the model was not over-fitted. It can be seen that all the predictions of the model for each of the above sentences gave a logical result. It can be seen that in the previous model we built the prediction of the last sentence came out illogical unlike the prediction of this model. yesterday the federal => **day** , yesterday the federal => **government**

Part (c) -- 4%

Report the test accuracy of your model

In [ ]:

```
# Write your code here
test_acc = estimate_accuracy_torch(pytorch_wordemb, test4grams)
print("Test accuracy is {:.4f}%".format(test_acc * 100))
```

```
Test accuracy is 36.6223%
```

Question 4. Visualizing Word Embeddings (14%)

While training the `PyTorchMLP` , we trained the `word_emb_layer` , which takes a one-hot representation of a word in our vocabulary, and returns a low-dimensional vector representation of that word. In this question, we will explore these word embeddings, which are a key concept in natural language processing.

Part (a) -- 4%

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

In [ ]:

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

Write your explanation here:

The reason why each row of weights contains the vector representation is because in one -hot encoding only one index is lifted and the other neurons get zero, therefore the only neuron that affects the output is the neuron in the coordination of the word representation and we can treat this output as the vector representation of the input word.

We know that for FC it holds that Z = WX + b in our case there is no bias and therefore it holds that Z = WX. When we put in vector X the value 1 only in position i then it will extract from the matrix W the corresponding column i, but we notice that the definition of the word_embd matrix is: **word_emb = word_emb_weights.T** where T represents transpose and therefore the i column of word_emb_weights is exactly the i-row of word_emb which is exactly what we wanted to see. This result shows us why it was important to set the bias to zero in this case.

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 [ ]:

```python
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']])
```

```
0.35270673
0.20609482
```

Compute the 5 closest words to the following words:

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

In [ ]:

```
# Write your code here
from termcolor import colored

words = ["four", "go", "what", "should", "school", "your", "yesterday", "not"]
for word in words:
  word_ind = vocab_stoi[word]
  word_similarities = similarities[word_ind, :] # the similarites of the word to other word
  word_similarities[word_ind] = -np.inf # disabling similarities between the word with hims
  closest_word_indx = word_similarities.argsort()[-5:][::-1] # the indices of the 5 closest
  print(f"The 5 closest words to {colored(word, 'red')} are: ")
  for idx, closest_word_ind in enumerate(closest_word_indx):
    print(f"position {idx + 1}.  the word {colored(vocab_itos[closest_word_ind], 'red')} wi
  print('\n')
```

```
The 5 closest words to four are:
position 1.  the word three with similarity 0.72010756
position 2.  the word two with similarity 0.6303034
position 3.  the word five with similarity 0.5871363
position 4.  the word million with similarity 0.54022795
position 5.  the word many with similarity 0.45306003


The 5 closest words to go are:
position 1.  the word get with similarity 0.43599644
position 2.  the word going with similarity 0.41052175
position 3.  the word next with similarity 0.39849392
position 4.  the word over with similarity 0.37997785
position 5.  the word see with similarity 0.34486917


The 5 closest words to what are:
position 1.  the word how with similarity 0.4551689
position 2.  the word when with similarity 0.37049952
position 3.  the word where with similarity 0.36389217
position 4.  the word who with similarity 0.35103813
position 5.  the word if with similarity 0.3467375


The 5 closest words to should are:
position 1.  the word could with similarity 0.5597263
position 2.  the word would with similarity 0.4876801
position 3.  the word can with similarity 0.46828604
position 4.  the word has with similarity 0.40997794
position 5.  the word will with similarity 0.36046764


The 5 closest words to school are:
position 1.  the word members with similarity 0.47050154
position 2.  the word world with similarity 0.4612146
position 3.  the word market with similarity 0.40233698
position 4.  the word few with similarity 0.4006654
position 5.  the word going with similarity 0.3980383
```

```
The 5 closest words to your are:
position 1.  the word our with similarity 0.63156444
position 2.  the word white with similarity 0.41939205
position 3.  the word the with similarity 0.40318593
position 4.  the word united with similarity 0.3708244
position 5.  the word an with similarity 0.36293775


The 5 closest words to yesterday are:
position 1.  the word me with similarity 0.4613043
position 2.  the word season with similarity 0.42209673
position 3.  the word year with similarity 0.409731
position 4.  the word week with similarity 0.40964732
position 5.  the word ms. with similarity 0.40124047


The 5 closest words to not are:
position 1.  the word nt with similarity 0.5380067
position 2.  the word director with similarity 0.37449327
position 3.  the word group with similarity 0.33563855
position 4.  the word also with similarity 0.32283944
position 5.  the word $ with similarity 0.31084713
```

Part (c) -- 5%

We can visualize the word embeddings by reducing the dimensionality of the word vectors to 2D. There are many dimensionality reduction techniques that we could use, and we will use an algorithm called t-SNE. (You don’t need to know what this is for the assignment; we will cover it later in the course.) Nearby points in this 2-D space are meant to correspond to nearby points in the original, high-dimensional space.

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

Look at the plot and find at least two clusters of related words.

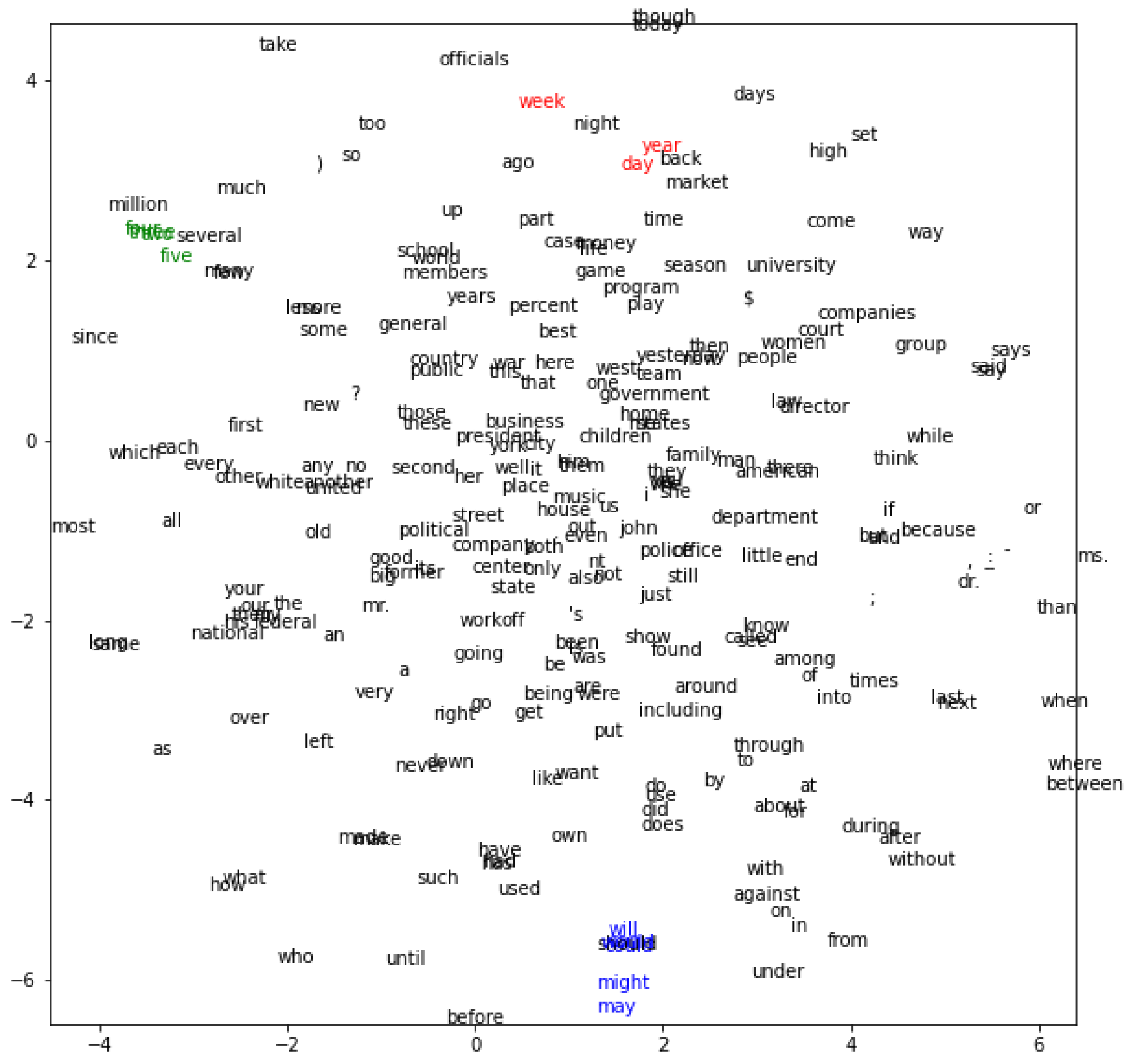
Write below for each cluster what is the commonality (if there is any) and if they make sense.

Note that there is randomness in the initialization of the t-SNE algorithm. If you re-run this code, you may get a different image. Please make sure to submit your image in the PDF file.

In [ ]:

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

# Define our clusters
time_set = ['day', 'week', 'year']
set_of_numbers = ['two', 'three', 'four', 'five']
set_of_future_words = ['will', 'could', 'might', 'may', 'can', 'would']

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):
    if w in time_set:
      plt.text(Y[i, 0], Y[i, 1], w, c='r')
    elif w in set_of_numbers:
      plt.text(Y[i, 0], Y[i, 1], w, c='g')
    elif w in set_of_future_words:
      plt.text(Y[i, 0], Y[i, 1], w, c='b')
    else:
      plt.text(Y[i, 0], Y[i, 1], w)
plt.show()
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/manifold/_t_sne.py:783: Futur
eWarning: The default initialization in TSNE will change from 'random' to 'p
ca' in 1.2.
  FutureWarning,
/usr/local/lib/python3.7/dist-packages/sklearn/manifold/_t_sne.py:793: Futur
eWarning: The default learning rate in TSNE will change from 200.0 to 'auto'
in 1.2.
  FutureWarning,
```



Explain and discuss your results here:

We can see from the image that there are several clusters that have close meaning:

1. time: we can see a cluster that include the words: 'year', 'week', 'day' they have close meaning that is time.
2. numbers : there is a cluster that include numbers like: 'four', 'two', 'three', 'five'.
3. The cluster that contains: will, could, might, may, can, would, all of them have close meaning in the manner that doing a some action/thing in the future.

We can conclude that there are clusters that reasonable/make sense, while there are other words that doesn't belong to each cluster although their meaning is indeed close to a particular group like word 'times' that doesn't belong to cluster time set (as we would expect).