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 [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import collections

import torch
import torch.nn as nn
import torch.nn.functional as F

import torch.optim as optim

from collections import Counter
```

In [2]:
```
# helper class to print nice outputs
class txt:
   BOLD = '\033[1m'
   UNDERLINE = '\033[4m'
   END = '\033[0m'

def make_bold(text):
    return txt.BOLD+ text + txt.END

def make_underline(text):
    return txt.UNDERLINE+ text + txt.END
```

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:

In [3]:
```
from google.colab import drive
drive.mount('/content/gdrive')
```

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

Find the path to `raw_sentences.txt`:

In [4]:
```
file_path = '/content/gdrive/My Drive/Intro_to_Deep_Learning/assignment2/raw_sentences
```

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 [5]:
```
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 [6]:
```
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 [7]:
```
test, valid, train = sentences[:10000], sentences[10000:20000], sentences[20000:]
```

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.

In [8]:
```
for i,sent in enumerate(train[10:20]):
  print(str(i+1)+".", " ".join(sent))
```

```
1. but for me , now , this is it .
2. she 's still there for us .
3. it 's part of this game , man .
4. it was : how do we get there ?
5. but they do nt last too long .
6. more are like me , she said .
7. who do you think they want to be like ?
8. no , he could not .
9. so i left it up to them .
10. we were nt right .
```

We can tell that puncuations such as dot or a comma is treated as a word by itself . When looking at sentence number 2 for example we observe that words with apostrophes are represented as 2 different words: the prefix of the ord and the apostrophes .

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.

In [9]:
```
train_words=[]
for sentence in train:
  for word in sentence:
    train_words.append(word)

print(make_bold("10 most common words in the vocabulary:"))
c=Counter(train_words)
freq=c.most_common(10)
df=pd.DataFrame(freq,columns=['word','num of occ'])
df['precentege']=(df['num of occ']/len(train_words))*100
df
```

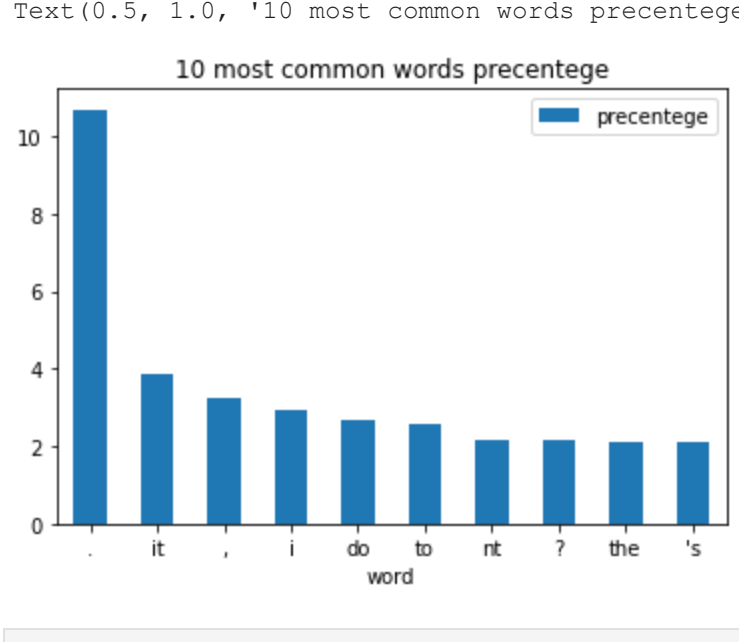
10 most common words in the vocabulary:

Out[9]:

	word	num of occ	precentege
0	.	64297	10.695720
1	it	23118	3.845648
2	,	19537	3.249954
3	i	17684	2.941710
4	do	16181	2.691688
5	to	15490	2.576741
6	nt	13009	2.164030
7	?	12881	2.142737
8	the	12583	2.093165
9	's	12552	2.088008

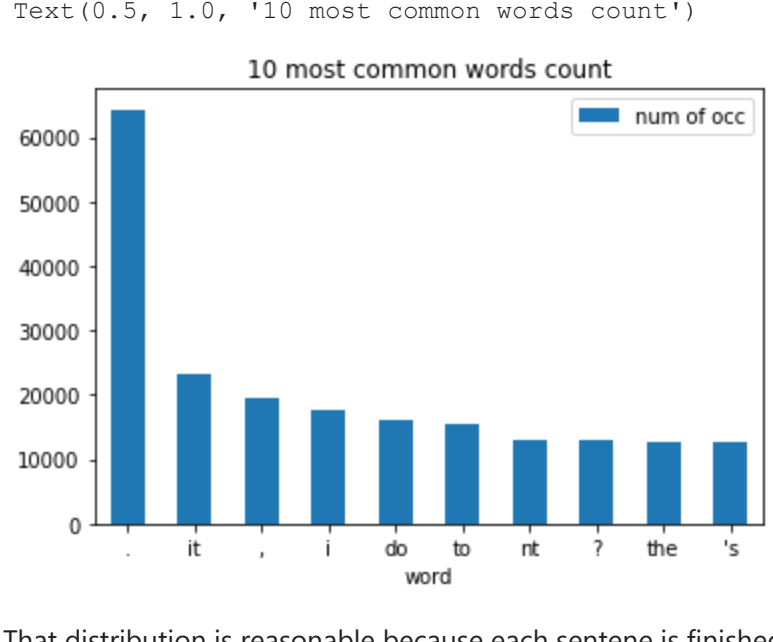
In [10]:
```
ax=df.plot.bar(x='word', y='precentege', rot=0)
ax.set_title("10 most common words precentege")
```

Out[10]:
```
Text(0.5, 1.0, '10 most common words precentege')
```



In [11]:
```
ax=df.plot.bar(x='word', y='num of occ', rot=0)
ax.set_title("10 most common words count")
```

Out[11]:
```
Text(0.5, 1.0, '10 most common words count')
```



That distribution is reasonable because each sentene is finished with dot or question mark for example.

Let's check words distribution without puncuation

In [12]:
```
import string
print(make_bold("10 most common words in  vocabulary without puncuation:"))
c=Counter([w for w in train_words if w not in string.punctuation])
freq=c.most_common(10)
df=pd.DataFrame(freq,columns=['word','num of occ'])
df['precentege']=(df['num of occ']/len(train_words))*100
df
```

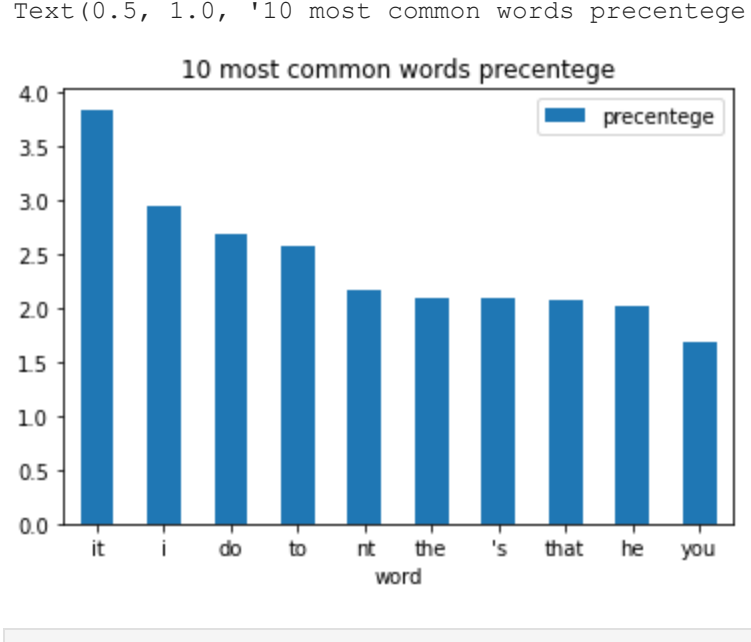
10 most common words in vocabulary without puncuation:

Out[12]:

	word	num of occ	precentege
0	it	23118	3.845648
1	i	17684	2.941710
2	do	16181	2.691688
3	to	15490	2.576741
4	nt	13009	2.164030
5	the	12583	2.093165
6	's	12552	2.088008
7	that	12535	2.085180
8	he	12192	2.028123
9	you	10166	1.691101

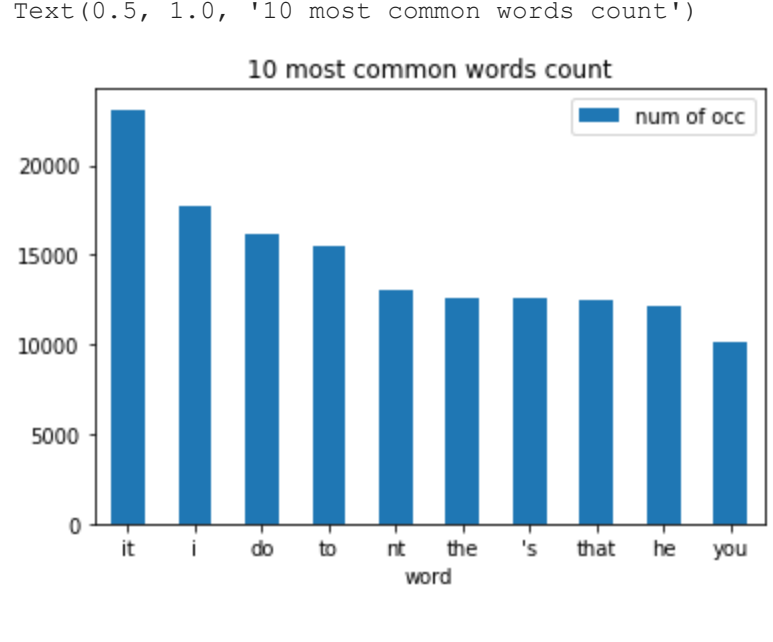
In [13]:
```
ax=df.plot.bar(x='word', y='precentege', rot=0)
ax.set_title("10 most common words precentege")
```

Out[13]:
```
Text(0.5, 1.0, '10 most common words precentege')
```



In [14]:
```
ax=df.plot.bar(x='word', y='num of occ', rot=0)
ax.set_title("10 most common words count")
```

Out[14]:
```
Text(0.5, 1.0, '10 most common words count')
```



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.

In [15]:
```
# 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'], ['othe
    [[148, 98, 70, 23, 154, 89], [151, 148, 181, 246], [248]]
    """
    new_sents=[]
    for s in sents:
      new_sents.append([vocab_stoi[w] for w in s])
    return new_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]]
    """
    four_grams=[]
    for seq in seqs:
      if (len(seq)<4):
        continue

      start=0
      end=len(seq)

      while(start+4<=end):
        four_grams.append(seq[start:start+4])
        start+=1
    return four_grams

def process_data(sents):
    """
    This function takes a list of sentences (list of lists), and generates an
    numpy matrix with shape [N, 4] containing indices of words in 4-grams.
    """
    indices = convert_words_to_indices(sents)
    fourgrams = generate_4grams(indices)
    return np.array(fourgrams)

# We can now generate our data which will be used to train and test the network
train4grams = process_data(train)
valid4grams = process_data(valid)
test4grams = process_data(test)
```

In [16]:
```
convert_words_to_indices([['one', 'in', 'five', 'are', 'over', 'here'], ['other', 'one
# assert : [[148, 98, 70, 23, 154, 89], [151, 148, 181, 246], [248]]
```

Out[16]:
```
[[148, 98, 70, 23, 154, 89], [151, 148, 181, 246], [248]]
```

In [17]:
```
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]]
```

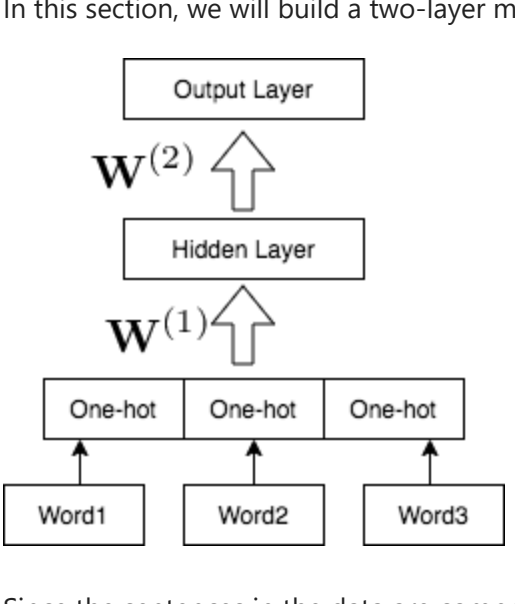
Out[17]:
```
[[148, 98, 70, 23], [98, 70, 23, 154], [70, 23, 154, 89], [151, 148, 181, 246]]
```

In [18]:
```
 generate_4grams([[1, 1, 1, 1, 1]])
  #  [[1, 1, 1, 1], [1, 1, 1, 1]]
```

Out[18]:
```
[[1, 1, 1, 1], [1, 1, 1, 1]]
```

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:

In [19]:
```
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)
    """

    output=[]
    # we go through all the batch
    for line in data:
      # deal with 1D dimention to use same code as 2D
      if data.ndim==1:
        line=[line]

      # First we want to each line make a 250 dimension vector of zeros
      one_hot_line=np.zeros((len(line),250))
      for i,d in enumerate(line):
        # to make one_hot we only put 1 in the word index
        one_hot_line[i][d]=1

      # add one_hot_line to the entire one hot encoding
      output.append(one_hot_line)

    return np.array(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
```

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:

In [20]:
```
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

  # We didn't use activation function between the layers because,as mention in the ta
    def forward(self, inp):
        inp = inp.reshape([-1, 750])
        return self.layer2(self.layer1(inp))
```

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 [21]:
```
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/ckp

    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 popula

    - /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_Learn

    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.forward(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 parame
            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

# change name of original signature
def plot_single_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()

def plot_losses(learning_curve_infos,rows,cols,parameters,parameter_name):
  fig, axes = plt.subplots(rows,cols, figsize=(20, 10))
  fig.suptitle("Loss per iteration",fontsize= 30)
  fig.subplots_adjust(top=0.9, wspace=0.3)

  i=0
  for learning_curve_info,ax in zip(learning_curve_infos, axes.flatten()):
    iters, losses, iters_sub, train_accs, val_accs=learning_curve_info
    ax.plot(iters, losses, label="Train")
    ax.title.set_text(parameter_name+str(parameters[i]))
    ax.set_xlabel("Iterations")
    ax.set_ylabel("Loss")
    ax.legend(loc='best')
    i+=1


def plot_acc(learning_curve_infos,rows,cols,parameters,parameter_name):
  fig, axes = plt.subplots(rows,cols, figsize=(20, 10))
  fig.suptitle("Accuracy per iteration",fontsize= 30)
  fig.subplots_adjust(top=0.9, wspace=0.3)

  i=0
  for learning_curve_info,ax in zip(learning_curve_infos, axes.flatten()):
    iters, losses, iters_sub, train_accs, val_accs=learning_curve_info
    ax.title.set_text(parameter_name+' ' +str(parameters[i]))
    ax.plot(iters_sub, train_accs, label="Train")
    ax.plot(iters_sub, val_accs, label="Validation")
    ax.set_xlabel("Iterations")
    ax.set_ylabel("Accuracy")
    ax.legend(loc='best')
    i+=1


# change the method to implement multiple plotting
def plot_learning_curve(learning_curve_infos,rows=1,cols=1,parameters=[],parameter_nar
  if rows==1 and cols==1:
    plot_single_learning_curve(*learning_curve_infos[0])
    return
  plot_losses(learning_curve_infos,rows,cols,parameters,parameter_name)
  plot_acc(learning_curve_infos,rows,cols,parameters,parameter_name)
```

We Will observe our model accuracy and loss with different Hyperparameters

Choosing Learning rate

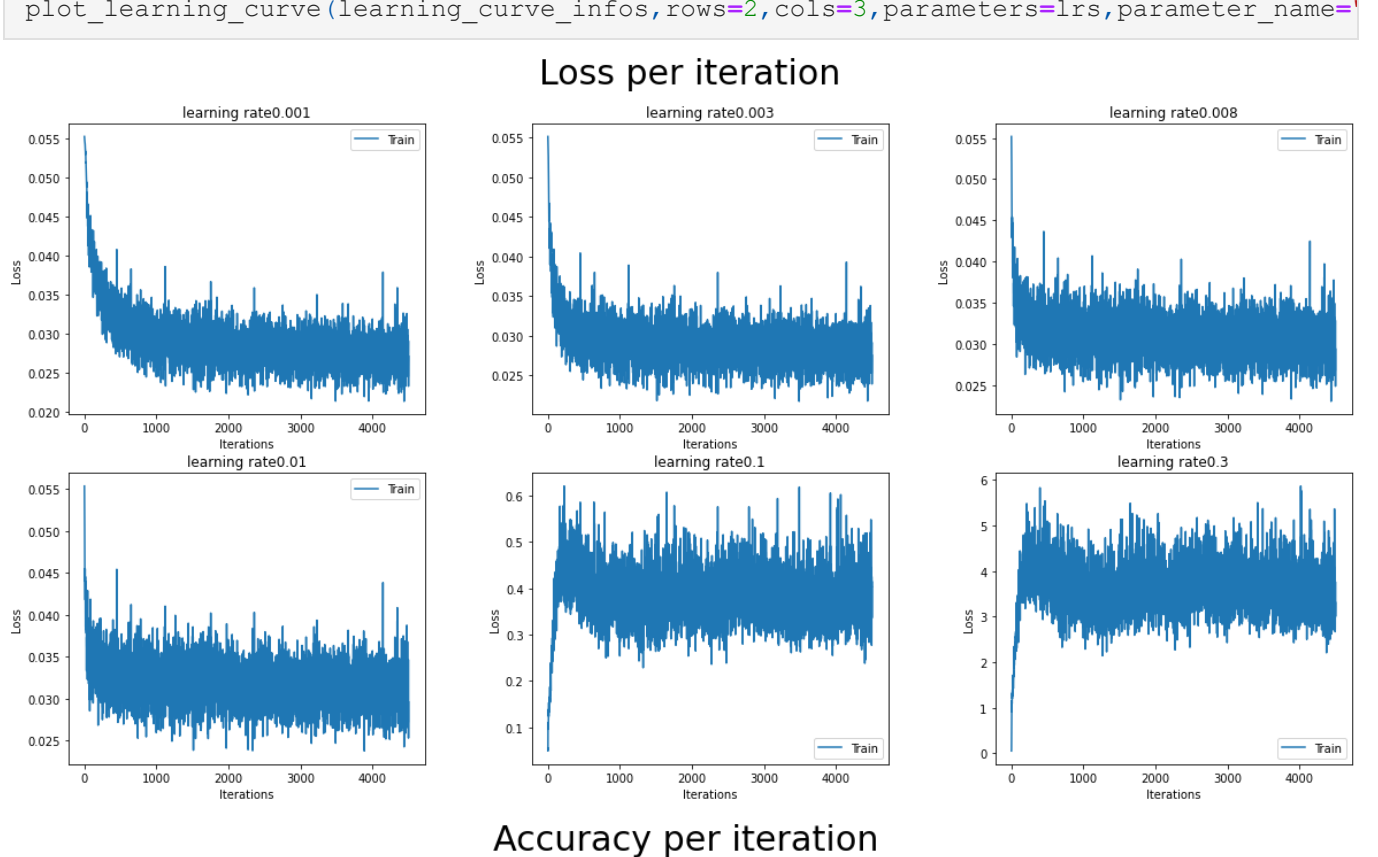
The learning rate is a hyperparameter that controls how much to change the model in response to the estimated error each time the model weights are updated.

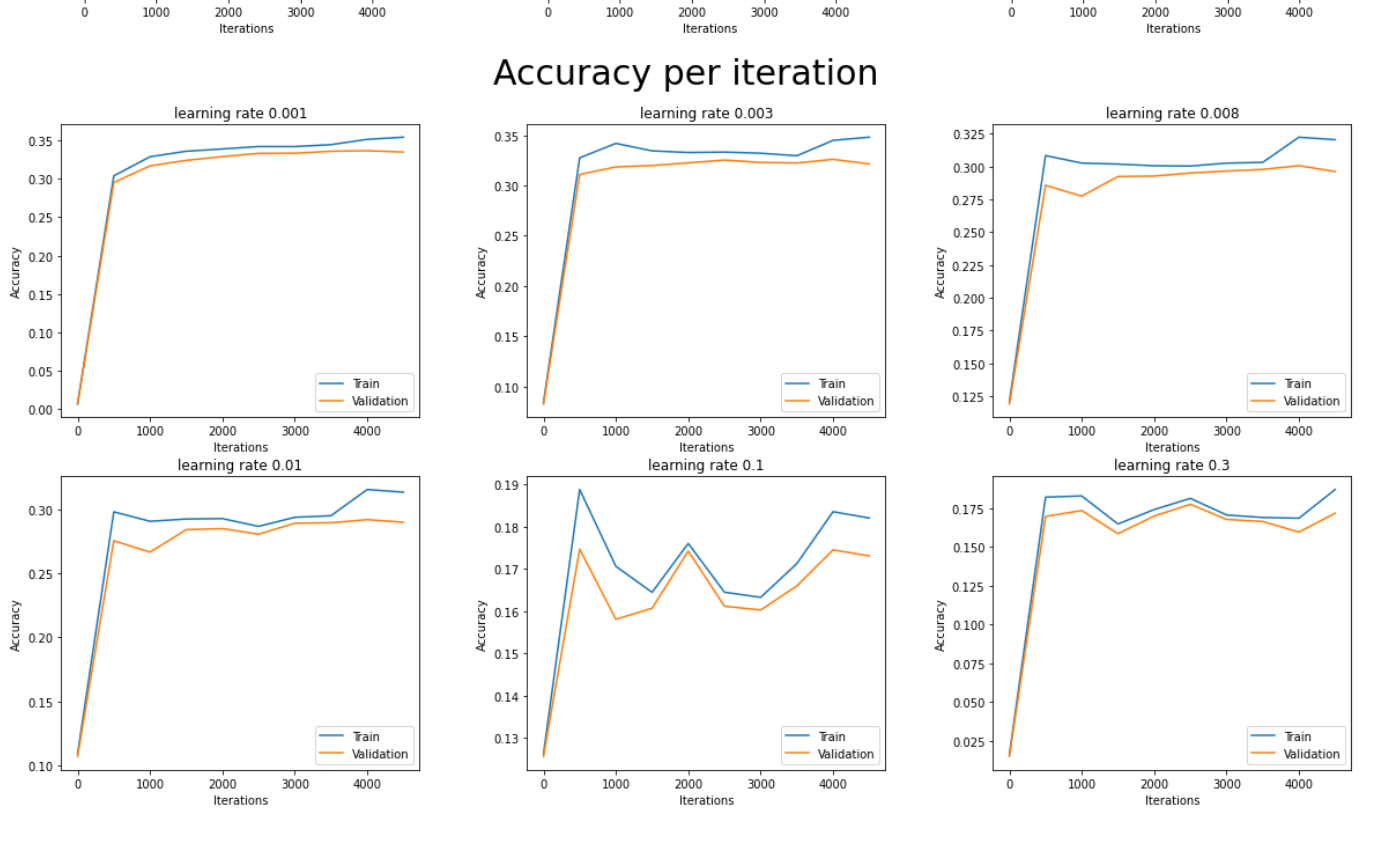
Learning rate too small may result in a long training process that could get stuck, whereas a value too large may result in learning a sub-optimal set of weights too fast or an unstable training process.

In [22]:
```
learning_curve_infos=[]
lrs=[0.001,0.003,0.008,0.01,0.1,0.3]
for lr in lrs:
  print(make_bold("learning rate"),lr)
  pytorch_mlp = PyTorchMLP()
  learning_curve_infos.append(run_pytorch_gradient_descent(pytorch_mlp,learning_rate=
  print("====================")
```

```
learning rate 0.001
Iter 0. [Val Acc 1%] [Train Acc 1%, Loss 5.522418]
Iter 500. [Val Acc 30%] [Train Acc 30%, Loss 2.887436]
Iter 1000. [Val Acc 32%] [Train Acc 33%, Loss 2.701034]
Iter 1500. [Val Acc 32%] [Train Acc 34%, Loss 2.674323]
Iter 2000. [Val Acc 33%] [Train Acc 34%, Loss 2.627582]
Iter 2500. [Val Acc 33%] [Train Acc 34%, Loss 2.698905]
Iter 3000. [Val Acc 33%] [Train Acc 34%, Loss 2.488635]
Iter 3500. [Val Acc 34%] [Train Acc 34%, Loss 2.494673]
Iter 4000. [Val Acc 34%] [Train Acc 35%, Loss 2.695042]
Iter 4500. [Val Acc 33%] [Train Acc 35%, Loss 2.712915]
====================
learning rate 0.003
Iter 0. [Val Acc 8%] [Train Acc 8%, Loss 5.515428]
Iter 500. [Val Acc 31%] [Train Acc 33%, Loss 2.841200]
Iter 1000. [Val Acc 32%] [Train Acc 34%, Loss 2.709029]
Iter 1500. [Val Acc 32%] [Train Acc 33%, Loss 2.784605]
Iter 2000. [Val Acc 32%] [Train Acc 33%, Loss 2.627650]
Iter 2500. [Val Acc 33%] [Train Acc 33%, Loss 2.771595]
Iter 3000. [Val Acc 32%] [Train Acc 33%, Loss 2.540770]
Iter 3500. [Val Acc 32%] [Train Acc 33%, Loss 2.572183]
Iter 4000. [Val Acc 33%] [Train Acc 34%, Loss 2.811066]
Iter 4500. [Val Acc 32%] [Train Acc 35%, Loss 2.748220]
====================
learning rate 0.008
Iter 0. [Val Acc 12%] [Train Acc 12%, Loss 5.516511]
Iter 500. [Val Acc 29%] [Train Acc 31%, Loss 3.017821]
Iter 1000. [Val Acc 28%] [Train Acc 30%, Loss 2.948218]
Iter 1500. [Val Acc 29%] [Train Acc 30%, Loss 3.025840]
Iter 2000. [Val Acc 29%] [Train Acc 30%, Loss 2.803314]
Iter 2500. [Val Acc 29%] [Train Acc 30%, Loss 2.943727]
Iter 3000. [Val Acc 30%] [Train Acc 30%, Loss 2.757127]
Iter 3500. [Val Acc 30%] [Train Acc 30%, Loss 2.733748]
Iter 4000. [Val Acc 30%] [Train Acc 32%, Loss 3.099771]
Iter 4500. [Val Acc 30%] [Train Acc 32%, Loss 2.938830]
====================
learning rate 0.01
Iter 0. [Val Acc 11%] [Train Acc 11%, Loss 5.531075]
Iter 500. [Val Acc 28%] [Train Acc 30%, Loss 3.061078]
Iter 1000. [Val Acc 27%] [Train Acc 29%, Loss 3.025113]
Iter 1500. [Val Acc 28%] [Train Acc 29%, Loss 3.148421]
Iter 2000. [Val Acc 28%] [Train Acc 29%, Loss 2.874151]
Iter 2500. [Val Acc 28%] [Train Acc 29%, Loss 2.964483]
Iter 3000. [Val Acc 29%] [Train Acc 29%, Loss 2.840285]
Iter 3500. [Val Acc 29%] [Train Acc 29%, Loss 2.776643]
Iter 4000. [Val Acc 29%] [Train Acc 32%, Loss 3.151916]
Iter 4500. [Val Acc 29%] [Train Acc 31%, Loss 2.958945]
====================
learning rate 0.1
Iter 0. [Val Acc 13%] [Train Acc 13%, Loss 5.525649]
Iter 500. [Val Acc 17%] [Train Acc 19%, Loss 34.750870]
Iter 1000. [Val Acc 16%] [Train Acc 17%, Loss 38.650402]
Iter 1500. [Val Acc 16%] [Train Acc 16%, Loss 33.163296]
Iter 2000. [Val Acc 17%] [Train Acc 18%, Loss 35.801147]
Iter 2500. [Val Acc 16%] [Train Acc 16%, Loss 38.608898]
Iter 3000. [Val Acc 16%] [Train Acc 16%, Loss 33.370453]
Iter 3500. [Val Acc 17%] [Train Acc 17%, Loss 29.342501]
Iter 4000. [Val Acc 17%] [Train Acc 18%, Loss 32.356991]
Iter 4500. [Val Acc 17%] [Train Acc 18%, Loss 40.221344]
====================
learning rate 0.3
Iter 0. [Val Acc 1%] [Train Acc 2%, Loss 5.519961]
Iter 500. [Val Acc 17%] [Train Acc 18%, Loss 356.180298]
Iter 1000. [Val Acc 17%] [Train Acc 18%, Loss 320.097992]
Iter 1500. [Val Acc 16%] [Train Acc 16%, Loss 337.695496]
Iter 2000. [Val Acc 17%] [Train Acc 17%, Loss 331.631531]
Iter 2500. [Val Acc 18%] [Train Acc 18%, Loss 415.418945]
Iter 3000. [Val Acc 17%] [Train Acc 17%, Loss 317.707214]
Iter 3500. [Val Acc 17%] [Train Acc 17%, Loss 303.282318]
Iter 4000. [Val Acc 16%] [Train Acc 17%, Loss 308.270508]
Iter 4500. [Val Acc 17%] [Train Acc 19%, Loss 330.497681]
====================
```

In [23]:
```
plot_learning_curve(learning_curve_infos,rows=2,cols=3,parameters=lrs,parameter_name='
```





- We get best results when using lr=0.01 .
- We can tell that for great lr greater or equals to 0.1 we get unstable training proccess.
- For lr of 0.01 , 0.08 we get learning proccess much slower that with lr of 0.001 . That might be because we do oscillation so the learning proccess is much slower.

Choosing Batch size

Batch size is the number of examples from the training dataset used in the estimate of the error gradient .

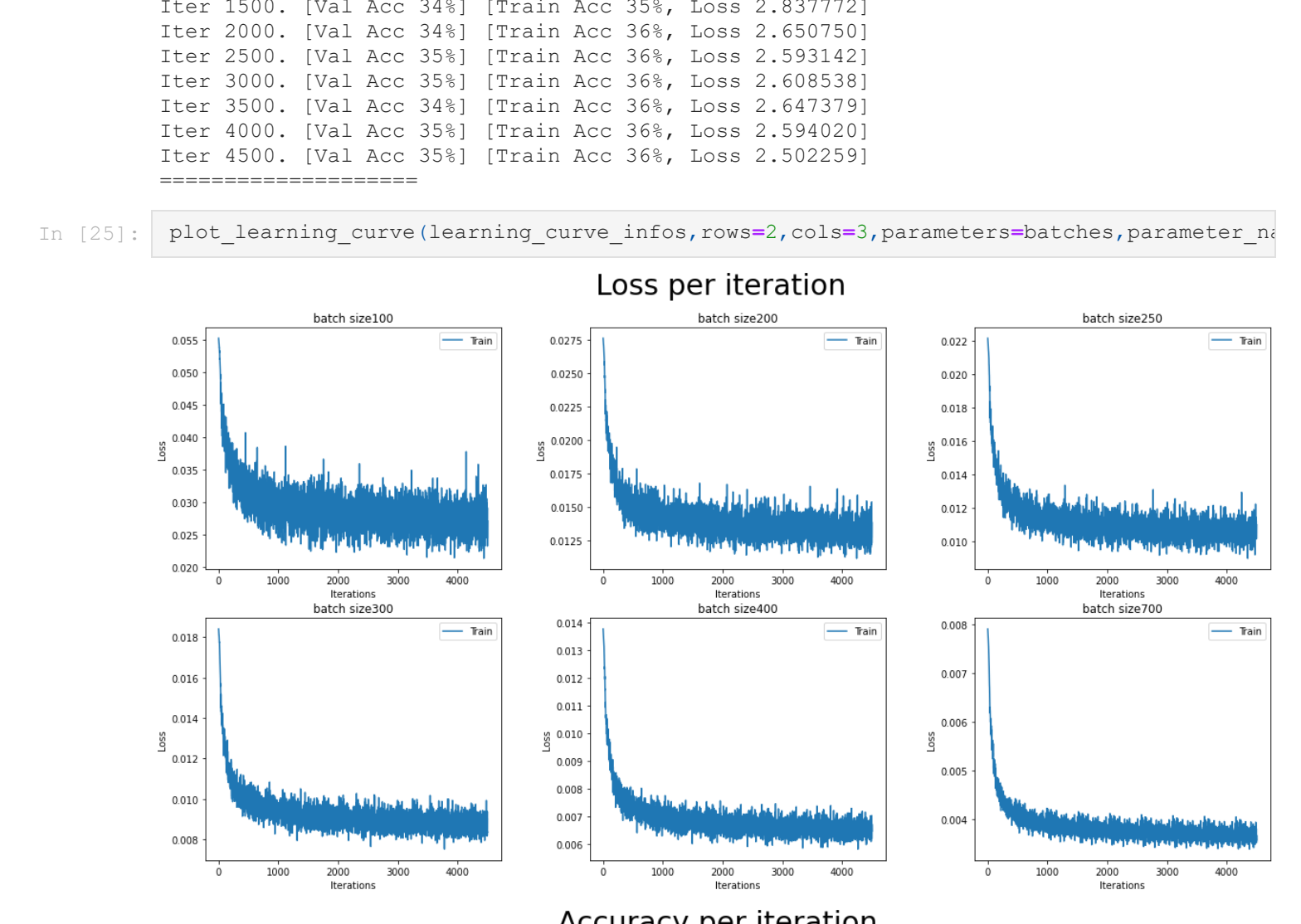
We may want to use a larger batch size to train out model as it allows computational speedups. However, it is well known that too large of a batch size will lead to poor generalization.

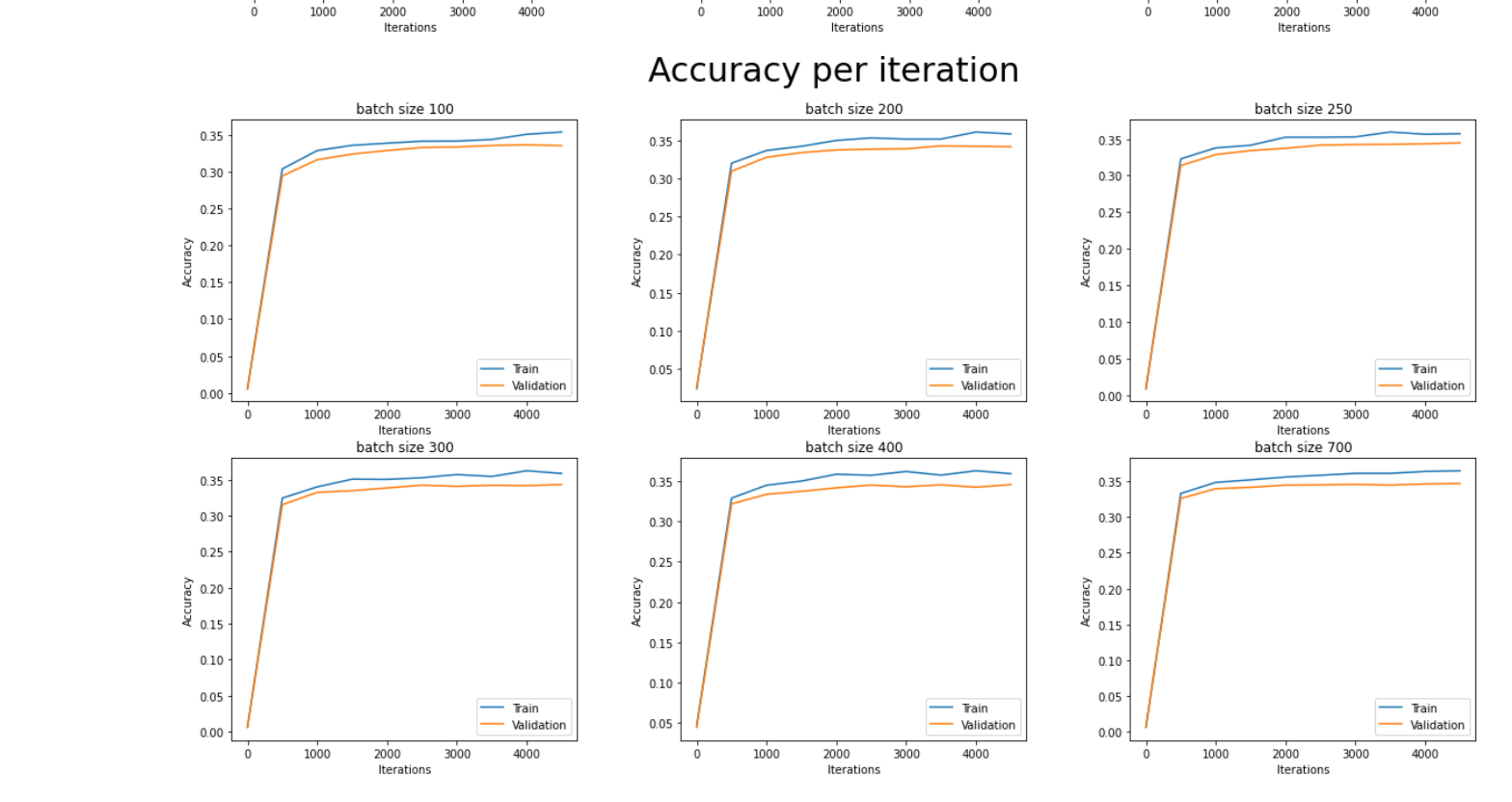
In [24]:
```
learning_curve_infos=[]
batches=[100,200,250,300,400,700]
for bs in batches:
  print(make_bold("batch size"),bs)
  pytorch_mlp = PyTorchMLP()
  learning_curve_infos.append(run_pytorch_gradient_descent(pytorch_mlp,batch_size=bs,r
  print("====================")
```

```
batch size 100
Iter 0. [Val Acc 1%] [Train Acc 1%, Loss 5.527025]
Iter 500. [Val Acc 29%] [Train Acc 30%, Loss 2.888649]
Iter 1000. [Val Acc 32%] [Train Acc 33%, Loss 2.705420]
Iter 1500. [Val Acc 32%] [Train Acc 34%, Loss 2.670044]
Iter 2000. [Val Acc 33%] [Train Acc 34%, Loss 2.624721]
Iter 2500. [Val Acc 33%] [Train Acc 34%, Loss 2.682849]
Iter 3000. [Val Acc 33%] [Train Acc 34%, Loss 2.492404]
Iter 3500. [Val Acc 34%] [Train Acc 34%, Loss 2.491659]
Iter 4000. [Val Acc 34%] [Train Acc 35%, Loss 2.701662]
Iter 4500. [Val Acc 34%] [Train Acc 35%, Loss 2.712550]
====================
batch size 200
Iter 0. [Val Acc 2%] [Train Acc 2%, Loss 5.522978]
Iter 500. [Val Acc 31%] [Train Acc 32%, Loss 2.823993]
Iter 1000. [Val Acc 33%] [Train Acc 34%, Loss 2.831392]
Iter 1500. [Val Acc 33%] [Train Acc 34%, Loss 2.735790]
Iter 2000. [Val Acc 34%] [Train Acc 35%, Loss 2.811525]
Iter 2500. [Val Acc 34%] [Train Acc 35%, Loss 2.830536]
Iter 3000. [Val Acc 34%] [Train Acc 35%, Loss 2.814664]
Iter 3500. [Val Acc 34%] [Train Acc 35%, Loss 2.987745]
Iter 4000. [Val Acc 34%] [Train Acc 36%, Loss 2.431539]
Iter 4500. [Val Acc 34%] [Train Acc 36%, Loss 2.770582]
====================
batch size 250
Iter 0. [Val Acc 1%] [Train Acc 1%, Loss 5.534167]
Iter 500. [Val Acc 31%] [Train Acc 32%, Loss 3.022698]
Iter 1000. [Val Acc 33%] [Train Acc 34%, Loss 2.612625]
Iter 1500. [Val Acc 33%] [Train Acc 34%, Loss 2.822279]
Iter 2000. [Val Acc 34%] [Train Acc 35%, Loss 2.798891]
Iter 2500. [Val Acc 34%] [Train Acc 35%, Loss 2.693784]
Iter 3000. [Val Acc 34%] [Train Acc 35%, Loss 2.434810]
Iter 3500. [Val Acc 34%] [Train Acc 36%, Loss 2.671269]
Iter 4000. [Val Acc 34%] [Train Acc 36%, Loss 2.639935]
Iter 4500. [Val Acc 34%] [Train Acc 36%, Loss 2.743180]
====================
batch size 300
Iter 0. [Val Acc 1%] [Train Acc 1%, Loss 5.523864]
Iter 500. [Val Acc 32%] [Train Acc 32%, Loss 2.975564]
Iter 1000. [Val Acc 33%] [Train Acc 34%, Loss 2.773191]
Iter 1500. [Val Acc 33%] [Train Acc 35%, Loss 2.825233]
Iter 2000. [Val Acc 34%] [Train Acc 35%, Loss 2.668298]
Iter 2500. [Val Acc 34%] [Train Acc 35%, Loss 2.594198]
Iter 3000. [Val Acc 34%] [Train Acc 36%, Loss 2.657933]
Iter 3500. [Val Acc 34%] [Train Acc 35%, Loss 2.685802]
Iter 4000. [Val Acc 34%] [Train Acc 36%, Loss 2.686276]
Iter 4500. [Val Acc 34%] [Train Acc 36%, Loss 2.801028]
====================
batch size 400
Iter 0. [Val Acc 4%] [Train Acc 5%, Loss 5.508717]
Iter 500. [Val Acc 32%] [Train Acc 33%, Loss 2.803985]
Iter 1000. [Val Acc 33%] [Train Acc 34%, Loss 2.858241]
Iter 1500. [Val Acc 34%] [Train Acc 35%, Loss 2.726085]
Iter 2000. [Val Acc 34%] [Train Acc 36%, Loss 2.578766]
Iter 2500. [Val Acc 34%] [Train Acc 36%, Loss 2.538229]
Iter 3000. [Val Acc 34%] [Train Acc 36%, Loss 2.592945]
Iter 3500. [Val Acc 34%] [Train Acc 36%, Loss 2.710787]
Iter 4000. [Val Acc 34%] [Train Acc 36%, Loss 2.613741]
Iter 4500. [Val Acc 35%] [Train Acc 36%, Loss 2.584879]
====================
batch size 700
Iter 0. [Val Acc 1%] [Train Acc 1%, Loss 5.530070]
```

```
Iter 500. [Val Acc 33%] [Train Acc 33%, Loss 2.845834]
Iter 1000. [Val Acc 34%] [Train Acc 35%, Loss 2.775978]
Iter 1500. [Val Acc 34%] [Train Acc 35%, Loss 2.837772]
Iter 2000. [Val Acc 34%] [Train Acc 36%, Loss 2.650750]
Iter 2500. [Val Acc 35%] [Train Acc 36%, Loss 2.593142]
Iter 3000. [Val Acc 35%] [Train Acc 36%, Loss 2.608538]
Iter 3500. [Val Acc 35%] [Train Acc 36%, Loss 2.647379]
Iter 4000. [Val Acc 35%] [Train Acc 36%, Loss 2.594020]
Iter 4500. [Val Acc 35%] [Train Acc 36%, Loss 2.502259]
====================
```

In [25]:
```python
plot_learning_curve(learning_curve_infos,rows=2,cols=3,parameters=batches,parameter_na
```





We indeed see much faster convergence when raise the batch size . We will choose batch size of 400 because we get great accuracy but not too big .

Choosing weight decay One way to penalize complexity, would be to add powers of all the weight to our loss function.However, it might result in our loss getting so huge that the best model would be to set all the parameters to 0.

To prevent that from happening, we multiply the sum of squares with another smaller number. This number is called weight decay or wd.

Loss = MSE(y_hat, y) + wd * sum(w^2)

In [26]:
```python
learning_curve_infos=[]
wds=[0,0.001,0.01]
for wd in wds:
  print(make_bold("weight decay"),wd)
  pytorch_mlp = PyTorchMLP()
  learning_curve_infos.append(run_pytorch_gradient_descent(pytorch_mlp,batch_size=450,
  print("====================")
```

```
weight decay 0
Iter 0. [Val Acc 0%] [Train Acc 0%, Loss 5.532355]
Iter 500. [Val Acc 32%] [Train Acc 33%, Loss 2.925249]
Iter 1000. [Val Acc 34%] [Train Acc 35%, Loss 2.716105]
Iter 1500. [Val Acc 34%] [Train Acc 35%, Loss 2.708787]
Iter 2000. [Val Acc 34%] [Train Acc 36%, Loss 2.672164]
Iter 2500. [Val Acc 34%] [Train Acc 36%, Loss 2.883249]
Iter 3000. [Val Acc 34%] [Train Acc 36%, Loss 2.666606]
Iter 3500. [Val Acc 34%] [Train Acc 36%, Loss 2.658825]
Iter 4000. [Val Acc 35%] [Train Acc 36%, Loss 2.637183]
Iter 4500. [Val Acc 34%] [Train Acc 36%, Loss 2.765458]
====================
weight decay 0.001
Iter 0. [Val Acc 0%] [Train Acc 0%, Loss 5.530281]
Iter 500. [Val Acc 29%] [Train Acc 30%, Loss 3.253338]
Iter 1000. [Val Acc 30%] [Train Acc 30%, Loss 3.064569]
Iter 1500. [Val Acc 30%] [Train Acc 30%, Loss 3.036934]
Iter 2000. [Val Acc 30%] [Train Acc 31%, Loss 3.348096]
Iter 2500. [Val Acc 30%] [Train Acc 31%, Loss 3.229254]
Iter 3000. [Val Acc 30%] [Train Acc 31%, Loss 3.122923]
Iter 3500. [Val Acc 30%] [Train Acc 31%, Loss 3.131729]
Iter 4000. [Val Acc 30%] [Train Acc 31%, Loss 3.116805]
Iter 4500. [Val Acc 30%] [Train Acc 31%, Loss 3.213165]
====================
weight decay 0.01
Iter 0. [Val Acc 1%] [Train Acc 1%, Loss 5.519382]
Iter 500. [Val Acc 21%] [Train Acc 22%, Loss 4.180748]
Iter 1000. [Val Acc 21%] [Train Acc 22%, Loss 3.939726]
Iter 1500. [Val Acc 21%] [Train Acc 22%, Loss 4.032385]
Iter 2000. [Val Acc 21%] [Train Acc 22%, Loss 4.072530]
Iter 2500. [Val Acc 21%] [Train Acc 22%, Loss 4.183617]
Iter 3000. [Val Acc 21%] [Train Acc 22%, Loss 4.092411]
Iter 3500. [Val Acc 21%] [Train Acc 22%, Loss 4.097340]
Iter 4000. [Val Acc 21%] [Train Acc 22%, Loss 4.089321]
Iter 4500. [Val Acc 21%] [Train Acc 22%, Loss 4.134225]
====================
```

We can tell that the weight decay it is in fact use as a regulizer. For now we will choost wd=0 because the model results are not greate and we do not need the regulizer right now.

If in the next sections we will think we get overfitting we will change this parameter.

In [27]:
```python
# Choosing best model
pytorch_mlp = PyTorchMLP()
learning_curve_info=run_pytorch_gradient_descent(pytorch_mlp,batch_size=450,max_iters
```

```
Iter 0. [Val Acc 1%] [Train Acc 1%, Loss 5.530056]
Iter 500. [Val Acc 32%] [Train Acc 33%, Loss 2.917321]
Iter 1000. [Val Acc 34%] [Train Acc 35%, Loss 2.711804]
Iter 1500. [Val Acc 34%] [Train Acc 35%, Loss 2.707022]
Iter 2000. [Val Acc 34%] [Train Acc 36%, Loss 2.666207]
Iter 2500. [Val Acc 34%] [Train Acc 36%, Loss 2.883750]
Iter 3000. [Val Acc 34%] [Train Acc 36%, Loss 2.668899]
Iter 3500. [Val Acc 35%] [Train Acc 36%, Loss 2.658391]
Iter 4000. [Val Acc 34%] [Train Acc 36%, Loss 2.639144]
Iter 4500. [Val Acc 34%] [Train Acc 36%, Loss 2.767608]
```

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 [28]:
```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
    # get last 3 words in the sentence
    xt=np.array([sentence[-3:]])
    # convert words to indices
    xt = np.array([vocab_stoi[w] for w in xt])
    xt = make_onehot(xt)
    # y is the output of the model
    y = pytorch_mlp(torch.Tensor(xt))
    y = y.detach().numpy() # convert the PyTorch tensor => numpy array
    # the prediction is the word with highest probability
    pred = np.argmax(y, axis=1)
    # index-> word so we will return the word itself
    return vocab_itos[pred[0]]
```

In [28]:
```python
make_prediction_torch(pytorch_mlp, ['you', 'are', 'a'])
```

Out[28]: 'good'

"you are a good" it is a good result in our opinion because it's a reasonable sentence , or more accurate a reasonable prefix for a sentence for example : " you are a good boy\girl\teacher\worker\dog\cat and etc.

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 [31]:
```python
print("you are a",make_bold(make_underline(make_prediction_torch(pytorch_mlp, ['you',
print("few companies shows",make_bold(make_underline(make_prediction_torch(pytorch_
print("there are no",make_bold(make_underline(make_prediction_torch(pytorch_mlp, ['th
print("yesterday i was",make_bold(make_underline(make_prediction_torch(pytorch_mlp, [
print("the game had",make_bold(make_underline(make_prediction_torch(pytorch_mlp, ['th
print("yesterday the federal",make_bold(make_underline(make_prediction_torch(pytorch_
```

you are a **good**
few companies shows **.**
there are no **other**
yesterday i was **nt**
the game had **to**
yesterday the federal **government**

The first sentence is reasonable as we have seen in the preview section .

- ***few companies shows .*** - This is not a reasonable sentence - First, grammarly I belived it more accurate to write is "few comapny show" and not "shows". In addition, we miss the object or some p reposition for example : "few companies show how that ..." , "few companies shows us ..." and etc.
- ***there are no other*** - This sentence makes a reasonable prefix of a sentence beacuse we can say : "there are no other options" ,"I finished my homework , there are no other assignment to do" , and etc.
- ***yesterday i was nt*** - this is not a reasonable result .
- ***the game had to*** - this is a reasonable result beacuse we can say "the game had to end\continue\be outside and etc. The pair of word "had to" is really common in the English language , hence we would like to observe if we got this result mainly beacuse it came after the word "had" or it was really depend on the whole history.
- ***yesterday the federal government***-this is a reasonable result beacuse we can say "yesterday the federal government showed\announced\told us\ told\old us" and etc. Here as well , The pair of word "federal government" is extremely common in the English language , hence we would like to check if we got this result mainly beacuse it came after the word "federal".

In [ ]:
```python
[w for w in train if]
```

Part (e) -- 4%

Report the test accuracy of your model

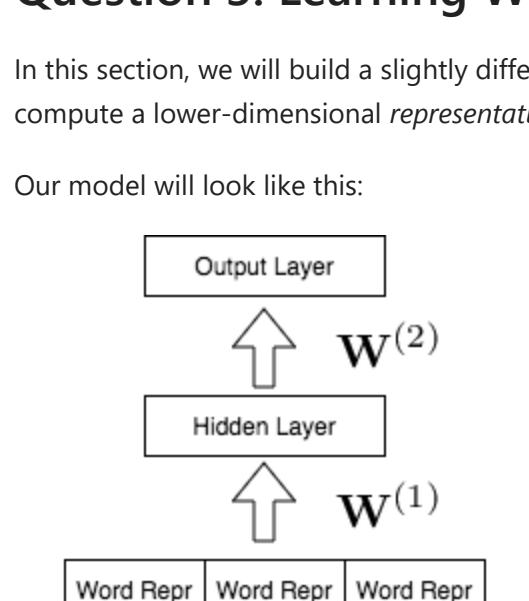
In [33]:
```python
estimate_accuracy_torch(pytorch_mlp, test4grams)
```

Out[33]: 0.3453292975680354

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.

Part (a) -- 8%

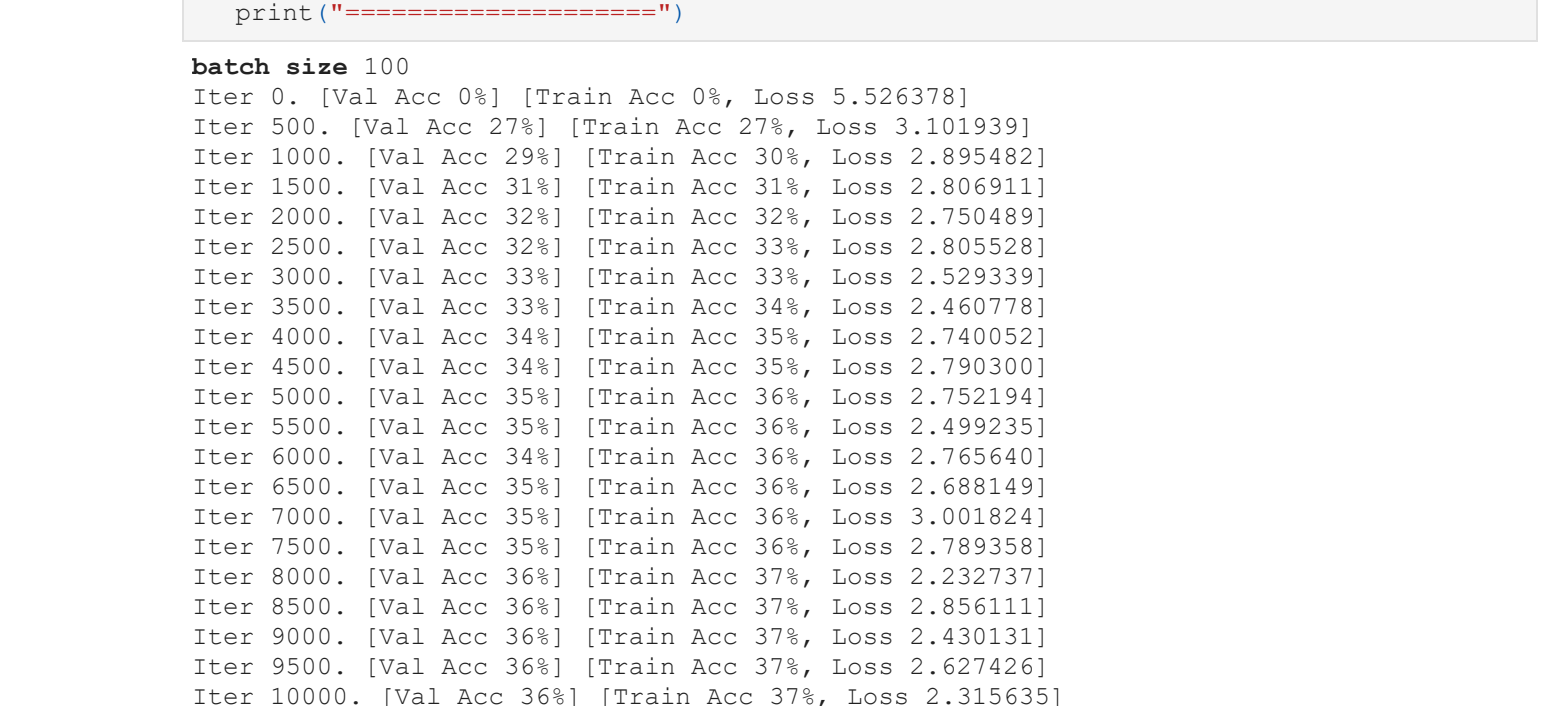
The PyTorch model is implemented for you. Use `run_pytorch_gradient_descent` to train your PyTorch MLP model to obtain a training accuracy of at least 38%. Plot the learning curve using the `plot_learning_curve` function provided to you, and include your plot in your PDF submission.

In [32]:
```python
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 [33]:
```python
results=[]
lrs=[0.001,0.003,0.008,0.01,0.1,0.3]
for lr in lrs:
  print(make_bold("learning rate"),lr)
  pytorch_wordemb= PyTorchWordEmb()
  results.append(run_pytorch_gradient_descent(pytorch_wordemb,
                                              max_iters=20000,learning_rate=lr))
  print("====================")
```

```
learning rate 0.001
Iter 0. [Val Acc 1%] [Train Acc 1%, Loss 5.519268]
Iter 500. [Val Acc 26%] [Train Acc 27%, Loss 3.100244]
Iter 1000. [Val Acc 29%] [Train Acc 30%, Loss 2.902720]
Iter 1500. [Val Acc 31%] [Train Acc 31%, Loss 2.828872]
Iter 2000. [Val Acc 31%] [Train Acc 32%, Loss 2.793866]
Iter 2500. [Val Acc 32%] [Train Acc 32%, Loss 2.787998]
Iter 3000. [Val Acc 33%] [Train Acc 33%, Loss 2.547953]
Iter 3500. [Val Acc 33%] [Train Acc 34%, Loss 2.465931]
Iter 4000. [Val Acc 33%] [Train Acc 34%, Loss 2.747259]
Iter 4500. [Val Acc 34%] [Train Acc 35%, Loss 2.780846]
Iter 5000. [Val Acc 34%] [Train Acc 35%, Loss 2.758509]
Iter 5500. [Val Acc 34%] [Train Acc 35%, Loss 2.516714]
Iter 6000. [Val Acc 34%] [Train Acc 36%, Loss 2.751294]
Iter 6500. [Val Acc 35%] [Train Acc 36%, Loss 2.677497]
Iter 7000. [Val Acc 35%] [Train Acc 36%, Loss 2.865299]
Iter 7500. [Val Acc 35%] [Train Acc 36%, Loss 2.757982]
Iter 8000. [Val Acc 35%] [Train Acc 37%, Loss 2.163836]
Iter 8500. [Val Acc 36%] [Train Acc 37%, Loss 2.845458]
Iter 9000. [Val Acc 36%] [Train Acc 37%, Loss 2.403343]
Iter 9500. [Val Acc 36%] [Train Acc 37%, Loss 2.657409]
Iter 10000. [Val Acc 36%] [Train Acc 37%, Loss 2.269865]
Iter 10500. [Val Acc 36%] [Train Acc 37%, Loss 2.585980]
Iter 11000. [Val Acc 36%] [Train Acc 38%, Loss 2.415839]
Iter 11500. [Val Acc 36%] [Train Acc 38%, Loss 2.499356]
Iter 12000. [Val Acc 36%] [Train Acc 38%, Loss 2.400043]
Iter 12500. [Val Acc 37%] [Train Acc 38%, Loss 2.509468]
Iter 13000. [Val Acc 37%] [Train Acc 38%, Loss 2.224254]
Iter 13500. [Val Acc 37%] [Train Acc 38%, Loss 2.788789]
Iter 14000. [Val Acc 37%] [Train Acc 38%, Loss 2.566184]
Iter 14500. [Val Acc 37%] [Train Acc 38%, Loss 2.703955]
Iter 15000. [Val Acc 37%] [Train Acc 39%, Loss 2.240930]
Iter 15500. [Val Acc 37%] [Train Acc 39%, Loss 2.543189]
Iter 16000. [Val Acc 37%] [Train Acc 39%, Loss 2.232550]
Iter 16500. [Val Acc 37%] [Train Acc 39%, Loss 2.385546]
Iter 17000. [Val Acc 37%] [Train Acc 39%, Loss 2.727089]
Iter 17500. [Val Acc 37%] [Train Acc 39%, Loss 2.300090]
Iter 18000. [Val Acc 37%] [Train Acc 39%, Loss 2.437120]
Iter 18500. [Val Acc 37%] [Train Acc 40%, Loss [illegible]]
Iter 19000. [Val Acc 37%] [Train Acc 40%, Loss [illegible]]
Iter 19500. [Val Acc 37%] [Train Acc 40%, Loss 2.444881]
Iter 20000. [Val Acc 37%] [Train Acc 40%, Loss 2.436245]
====================
learning rate 0.003
Iter 0. [Val Acc 0%] [Train Acc 0%, Loss 5.526578]
Iter 500. [Val Acc 29%] [Train Acc 30%, Loss 2.684386]
Iter 1000. [Val Acc 32%] [Train Acc 33%, Loss 2.705063]
Iter 1500. [Val Acc 33%] [Train Acc 34%, Loss 2.735406]
Iter 2000. [Val Acc 34%] [Train Acc 35%, Loss 2.636649]
Iter 2500. [Val Acc 34%] [Train Acc 35%, Loss 2.632671]
Iter 3000. [Val Acc 35%] [Train Acc 35%, Loss 2.384767]
Iter 3500. [Val Acc 35%] [Train Acc 36%, Loss 2.647838]
Iter 4000. [Val Acc 35%] [Train Acc 36%, Loss 2.450407]
Iter 4500. [Val Acc 35%] [Train Acc 36%, Loss 2.425554]
Iter 5000. [Val Acc 35%] [Train Acc 36%, Loss 2.518296]
Iter 5500. [Val Acc 35%] [Train Acc 37%, Loss 2.448766]
Iter 6000. [Val Acc 36%] [Train Acc 37%, Loss 2.565966]
Iter 6500. [Val Acc 36%] [Train Acc 37%, Loss 2.953698]
Iter 7000. [Val Acc 36%] [Train Acc 37%, Loss 2.702233]
Iter 7500. [Val Acc 36%] [Train Acc 37%, Loss 2.098846]
Iter 8000. [Val Acc 36%] [Train Acc 38%, Loss 2.457522]
Iter 8500. [Val Acc 36%] [Train Acc 38%, Loss 2.890601]
Iter 9000. [Val Acc 36%] [Train Acc 38%, Loss 2.652076]
Iter 9500. [Val Acc 36%] [Train Acc 38%, Loss 2.361770]
Iter 10000. [Val Acc 36%] [Train Acc 38%, Loss 2.481927]
Iter 10500. [Val Acc 37%] [Train Acc 38%, Loss 2.490603]
Iter 11000. [Val Acc 37%] [Train Acc 39%, Loss 2.589154]
Iter 11500. [Val Acc 37%] [Train Acc 39%, Loss 2.364857]
Iter 12000. [Val Acc 37%] [Train Acc 39%, Loss 2.473296]
Iter 12500. [Val Acc 37%] [Train Acc 39%, Loss 2.330908]
Iter 13000. [Val Acc 37%] [Train Acc 39%, Loss 2.780708]
Iter 13500. [Val Acc 37%] [Train Acc 39%, Loss 2.484586]
Iter 14000. [Val Acc 37%] [Train Acc 39%, Loss 2.731416]
Iter 14500. [Val Acc 37%] [Train Acc 39%, Loss 2.264750]
Iter 15000. [Val Acc 37%] [Train Acc 40%, Loss 2.470318]
Iter 15500. [Val Acc 37%] [Train Acc 40%, Loss 2.234471]
Iter 16000. [Val Acc 37%] [Train Acc 40%, Loss 2.349515]
Iter 16500. [Val Acc 37%] [Train Acc 40%, Loss 2.406139]
Iter 17000. [Val Acc 37%] [Train Acc 40%, Loss 2.361968]
Iter 17500. [Val Acc 37%] [Train Acc 40%, Loss 2.477860]
Iter 18000. [Val Acc 37%] [Train Acc 40%, Loss 2.239039]
Iter 18500. [Val Acc 37%] [Train Acc 41%, Loss 2.412004]
Iter 19000. [Val Acc 37%] [Train Acc 40%, Loss 2.475663]
Iter 19500. [Val Acc 37%] [Train Acc 40%, Loss 2.479686]
Iter 20000. [Val Acc 37%] [Train Acc 40%, Loss [illegible]]
====================
learning rate 0.008
Iter 0. [Val Acc 1%] [Train Acc 1%, Loss 5.512646]
Iter 500. [Val Acc 30%] [Train Acc 31%, Loss 2.807651]
Iter 1000. [Val Acc 33%] [Train Acc 34%, Loss 2.713894]
Iter 1500. [Val Acc 33%] [Train Acc 34%, Loss 2.758679]
Iter 2000. [Val Acc 34%] [Train Acc 35%, Loss 2.601867]
Iter 2500. [Val Acc 34%] [Train Acc 35%, Loss 2.682910]
Iter 3000. [Val Acc 35%] [Train Acc 35%, Loss 2.456670]
Iter 3500. [Val Acc 35%] [Train Acc 36%, Loss 2.617520]
Iter 4000. [Val Acc 35%] [Train Acc 36%, Loss 2.727284]
Iter 4500. [Val Acc 35%] [Train Acc 36%, Loss 2.575799]
Iter 5000. [Val Acc 35%] [Train Acc 36%, Loss 2.730227]
Iter 5500. [Val Acc 35%] [Train Acc 37%, Loss 2.504980]
Iter 6000. [Val Acc 36%] [Train Acc 37%, Loss 2.758368]
Iter 6500. [Val Acc 36%] [Train Acc 37%, Loss 2.534030]
Iter 7000. [Val Acc 36%] [Train Acc 37%, Loss 2.905443]
Iter 7500. [Val Acc 36%] [Train Acc 37%, Loss 2.448021]
Iter 8000. [Val Acc 36%] [Train Acc 37%, Loss 2.144898]
Iter 8500. [Val Acc 36%] [Train Acc 38%, Loss 2.939853]
Iter 9000. [Val Acc 36%] [Train Acc 38%, Loss 2.452962]
Iter 9500. [Val Acc 36%] [Train Acc 38%, Loss 2.780459]
Iter 10000. [Val Acc 36%] [Train Acc 38%, Loss 2.235870]
Iter 10500. [Val Acc 36%] [Train Acc 38%, Loss 2.651011]
Iter 11000. [Val Acc 36%] [Train Acc 38%, Loss 2.430880]
Iter 11500. [Val Acc 36%] [Train Acc 38%, Loss 2.640204]
Iter 12000. [Val Acc 36%] [Train Acc 38%, Loss 2.407128]
Iter 12500. [Val Acc 36%] [Train Acc 38%, Loss 2.543581]
Iter 13000. [Val Acc 36%] [Train Acc 38%, Loss 2.229008]
Iter 13500. [Val Acc 36%] [Train Acc 38%, Loss 2.426442]
Iter 14000. [Val Acc 36%] [Train Acc 38%, Loss 2.564492]
Iter 14500. [Val Acc 36%] [Train Acc 38%, Loss 2.856139]
Iter 15000. [Val Acc 36%] [Train Acc 38%, Loss 2.364792]
Iter 15500. [Val Acc 36%] [Train Acc 38%, Loss 2.631128]
Iter 16000. [Val Acc 36%] [Train Acc 38%, Loss 2.265149]
Iter 16500. [Val Acc 36%] [Train Acc 38%, Loss 2.465870]
Iter 17000. [Val Acc 36%] [Train Acc 38%, Loss 2.500021]
Iter 17500. [Val Acc 36%] [Train Acc 38%, Loss 2.740179]
Iter 18000. [Val Acc 36%] [Train Acc 38%, Loss 2.395359]
Iter 18500. [Val Acc 36%] [Train Acc 38%, Loss 2.496350]
Iter 19000. [Val Acc 36%] [Train Acc 38%, Loss 2.668398]
Iter 19500. [Val Acc 36%] [Train Acc 38%, Loss 2.501441]
Iter 20000. [Val Acc 36%] [Train Acc 38%, Loss [illegible]]
====================
learning rate 0.01
Iter 0. [Val Acc 1%] [Train Acc 1%, Loss 5.517945]
Iter 500. [Val Acc 30%] [Train Acc 30%, Loss 2.819769]
Iter 1000. [Val Acc 32%] [Train Acc 33%, Loss 2.715975]
Iter 1500. [Val Acc 33%] [Train Acc 34%, Loss 2.740938]
Iter 2000. [Val Acc 34%] [Train Acc 35%, Loss 2.716451]
Iter 2500. [Val Acc 34%] [Train Acc 35%, Loss 2.470433]
Iter 3000. [Val Acc 35%] [Train Acc 36%, Loss 2.585647]
Iter 3500. [Val Acc 35%] [Train Acc 36%, Loss 2.731110]
Iter 4000. [Val Acc 35%] [Train Acc 36%, Loss 2.618104]
Iter 4500. [Val Acc 35%] [Train Acc 36%, Loss 2.708660]
Iter 5000. [Val Acc 35%] [Train Acc 36%, Loss 2.773429]
Iter 5500. [Val Acc 35%] [Train Acc 36%, Loss 2.619218]
Iter 6000. [Val Acc 35%] [Train Acc 36%, Loss [illegible]]
Iter 6500. [Val Acc 35%] [Train Acc 37%, Loss 2.894778]
Iter 7000. [Val Acc 35%] [Train Acc 37%, Loss 2.779774]
Iter 7500. [Val Acc 35%] [Train Acc 37%, Loss 2.172861]
Iter 8000. [Val Acc 35%] [Train Acc 37%, Loss 2.689774]
Iter 8500. [Val Acc 35%] [Train Acc 37%, Loss 2.355607]
Iter 9000. [Val Acc 35%] [Train Acc 37%, Loss 2.839906]
Iter 9500. [Val Acc 36%] [Train Acc 37%, Loss 2.333467]
Iter 10000. [Val Acc 36%] [Train Acc 37%, Loss 2.522873]
Iter 10500. [Val Acc 36%] [Train Acc 37%, Loss 2.778725]
Iter 11000. [Val Acc 36%] [Train Acc 37%, Loss 2.729805]
Iter 11500. [Val Acc 36%] [Train Acc 37%, Loss [illegible]]
Iter 12000. [Val Acc 36%] [Train Acc 38%, Loss 2.540784]
Iter 12500. [Val Acc 35%] [Train Acc 37%, Loss 2.469973]
Iter 13000. [Val Acc 35%] [Train Acc 36%, Loss 2.298291]
Iter 13500. [Val Acc 36%] [Train Acc 37%, Loss 2.969486]
Iter 14000. [Val Acc 35%] [Train Acc 36%, Loss 2.644630]
Iter 14500. [Val Acc 35%] [Train Acc 36%, Loss 2.883139]
Iter 15000. [Val Acc 35%] [Train Acc 37%, Loss 2.351715]
Iter 15500. [Val Acc 36%] [Train Acc 38%, Loss 2.679552]
Iter 16000. [Val Acc 35%] [Train Acc 37%, Loss 2.253974]
Iter 16500. [Val Acc 36%] [Train Acc 37%, Loss 2.671885]
Iter 17000. [Val Acc 35%] [Train Acc 37%, Loss 2.591083]
Iter 17500. [Val Acc 35%] [Train Acc 36%, Loss 2.735941]
Iter 18000. [Val Acc 36%] [Train Acc 36%, Loss 2.938481]
Iter 18500. [Val Acc 35%] [Train Acc 36%, Loss 2.419359]
Iter 19000. [Val Acc 35%] [Train Acc 38%, Loss 2.505806]
Iter 19500. [Val Acc 35%] [Train Acc 38%, Loss 2.801003]
Iter 20000. [Val Acc 35%] [Train Acc 37%, Loss 2.598671]
====================
learning rate 0.1
Iter 0. [Val Acc 17%] [Train Acc 17%, Loss 5.503949]
Iter 500. [Val Acc 18%] [Train Acc 18%, Loss 3.974689]
Iter 1000. [Val Acc 18%] [Train Acc 18%, Loss 3.971218]
Iter 1500. [Val Acc 18%] [Train Acc 18%, Loss 4.310612]
Iter 2000. [Val Acc 17%] [Train Acc 17%, Loss 4.348298]
Iter 2500. [Val Acc 18%] [Train Acc 18%, Loss 4.408175]
Iter 3000. [Val Acc 18%] [Train Acc 18%, Loss 4.223944]
Iter 3500. [Val Acc 17%] [Train Acc 17%, Loss 5.291592]
Iter 4000. [Val Acc 17%] [Train Acc 17%, Loss 4.374187]
Iter 4500. [Val Acc 17%] [Train Acc 17%, Loss 4.510012]
Iter 5000. [Val Acc 17%] [Train Acc 17%, Loss 4.303131]
Iter 5500. [Val Acc 17%] [Train Acc 17%, Loss 4.325607]
Iter 6000. [Val Acc 17%] [Train Acc 17%, Loss 4.390027]
Iter 6500. [Val Acc 17%] [Train Acc 17%, Loss 4.337491]
Iter 7000. [Val Acc 17%] [Train Acc 17%, Loss 4.532055]
Iter 7500. [Val Acc 17%] [Train Acc 17%, Loss 4.489871]
Iter 8000. [Val Acc 17%] [Train Acc 17%, Loss 4.227412]
Iter 8500. [Val Acc 17%] [Train Acc 17%, Loss 4.461556]
Iter 9000. [Val Acc 17%] [Train Acc 17%, Loss 4.366011]
Iter 9500. [Val Acc 17%] [Train Acc 17%, Loss 4.466624]
Iter 10000. [Val Acc 17%] [Train Acc 17%, Loss 4.333961]
Iter 10500. [Val Acc 17%] [Train Acc 17%, Loss 4.322211]
Iter 11000. [Val Acc 17%] [Train Acc 17%, Loss 4.703192]
Iter 11500. [Val Acc 17%] [Train Acc 17%, Loss 4.499238]
Iter 12000. [Val Acc 17%] [Train Acc 17%, Loss 4.502230]
Iter 12500. [Val Acc 17%] [Train Acc 17%, Loss 4.248088]
Iter 13000. [Val Acc 17%] [Train Acc 17%, Loss 4.319182]
Iter 13500. [Val Acc 17%] [Train Acc 17%, Loss 4.382495]
Iter 14000. [Val Acc 17%] [Train Acc 17%, Loss 4.536654]
Iter 14500. [Val Acc 17%] [Train Acc 17%, Loss 4.540475]
Iter 15000. [Val Acc 17%] [Train Acc 17%, Loss 4.437110]
Iter 15500. [Val Acc 17%] [Train Acc 17%, Loss 4.388434]
Iter 16000. [Val Acc 17%] [Train Acc 17%, Loss 4.116671]
Iter 16500. [Val Acc 17%] [Train Acc 17%, Loss 4.435702]
Iter 17000. [Val Acc 17%] [Train Acc 17%, Loss 4.462017]
Iter 17500. [Val Acc 17%] [Train Acc 17%, Loss 4.579310]
Iter 18000. [Val Acc 17%] [Train Acc 17%, Loss 4.660410]
Iter 18500. [Val Acc 17%] [Train Acc 17%, Loss 4.287421]
Iter 19000. [Val Acc 17%] [Train Acc 17%, Loss 4.412161]
Iter 19500. [Val Acc 17%] [Train Acc 17%, Loss 4.476344]
Iter 20000. [Val Acc 17%] [Train Acc 17%, Loss 4.592700]
====================
learning rate 0.3
Iter 0. [Val Acc 17%] [Train Acc 17%, Loss 5.523943]
Iter 500. [Val Acc 17%] [Train Acc 17%, Loss 4.247097]
Iter 1000. [Val Acc 17%] [Train Acc 17%, Loss 4.188886]
Iter 1500. [Val Acc 17%] [Train Acc 17%, Loss 4.473576]
Iter 2000. [Val Acc 17%] [Train Acc 17%, Loss 4.425730]
Iter 2500. [Val Acc 17%] [Train Acc 17%, Loss 4.462398]
Iter 3000. [Val Acc 17%] [Train Acc 17%, Loss 4.347627]
Iter 3500. [Val Acc 17%] [Train Acc 17%, Loss 4.158957]
Iter 4000. [Val Acc 17%] [Train Acc 17%, Loss 4.427967]
Iter 4500. [Val Acc 17%] [Train Acc 17%, Loss 4.633073]
Iter 5000. [Val Acc 17%] [Train Acc 17%, Loss 4.382990]
Iter 5500. [Val Acc 17%] [Train Acc 17%, Loss 4.384750]
Iter 6000. [Val Acc 17%] [Train Acc 17%, Loss 4.440473]
Iter 6500. [Val Acc 17%] [Train Acc 17%, Loss 4.386702]
Iter 7000. [Val Acc 17%] [Train Acc 17%, Loss 4.552884]
Iter 7500. [Val Acc 17%] [Train Acc 17%, Loss 4.539216]
Iter 8000. [Val Acc 17%] [Train Acc 17%, Loss 4.269054]
Iter 8500. [Val Acc 17%] [Train Acc 17%, Loss 4.531420]
Iter 9000. [Val Acc 17%] [Train Acc 17%, Loss 4.386399]
Iter 9500. [Val Acc 17%] [Train Acc 17%, Loss 4.526526]
Iter 10000. [Val Acc 17%] [Train Acc 17%, Loss 4.371215]
Iter 10500. [Val Acc 17%] [Train Acc 17%, Loss 4.350615]
Iter 11000. [Val Acc 17%] [Train Acc 17%, Loss 4.786835]
Iter 11500. [Val Acc 17%] [Train Acc 17%, Loss 4.536767]
Iter 12000. [Val Acc 17%] [Train Acc 17%, Loss 4.543468]
Iter 12500. [Val Acc 17%] [Train Acc 17%, Loss 4.321394]
Iter 13000. [Val Acc 17%] [Train Acc 17%, Loss 4.356948]
Iter 13500. [Val Acc 17%] [Train Acc 17%, Loss 4.464230]
Iter 14000. [Val Acc 17%] [Train Acc 17%, Loss 4.640809]
Iter 14500. [Val Acc 17%] [Train Acc 17%, Loss 4.616512]
Iter 15000. [Val Acc 17%] [Train Acc 17%, Loss 4.529929]
Iter 15500. [Val Acc 17%] [Train Acc 17%, Loss 4.426794]
Iter 16000. [Val Acc 17%] [Train Acc 17%, Loss 4.153410]
Iter 16500. [Val Acc 17%] [Train Acc 17%, Loss 4.516141]
Iter 17000. [Val Acc 17%] [Train Acc 17%, Loss 4.524895]
Iter 17500. [Val Acc 17%] [Train Acc 17%, Loss 4.619377]
Iter 18000. [Val Acc 17%] [Train Acc 17%, Loss 4.743680]
Iter 18500. [Val Acc 17%] [Train Acc 17%, Loss 4.313883]
Iter 19000. [Val Acc 17%] [Train Acc 17%, Loss 4.468691]
Iter 19500. [Val Acc 17%] [Train Acc 17%, Loss 4.493729]
Iter 20000. [Val Acc 17%] [Train Acc 17%, Loss 4.620727]
====================
```

In [34]:
```python
plot_learning_curve(results,rows=2,cols=3,parameters=lrs,parameter_name="learning rate
```



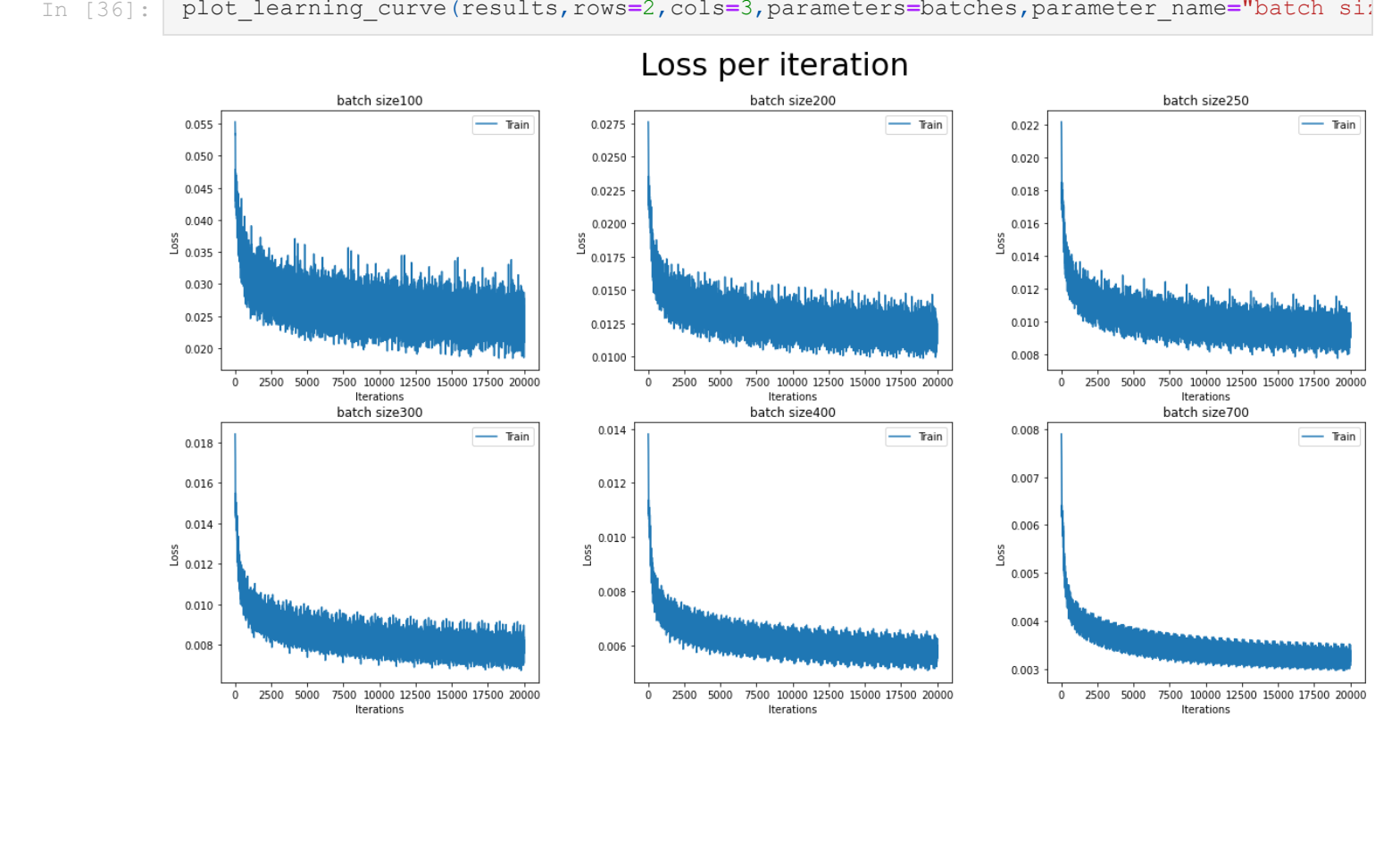
Here again we can tell that learning rate too high such as 0.1 or 0.3 gives us really bad training - it look like it convergece to another local optima.

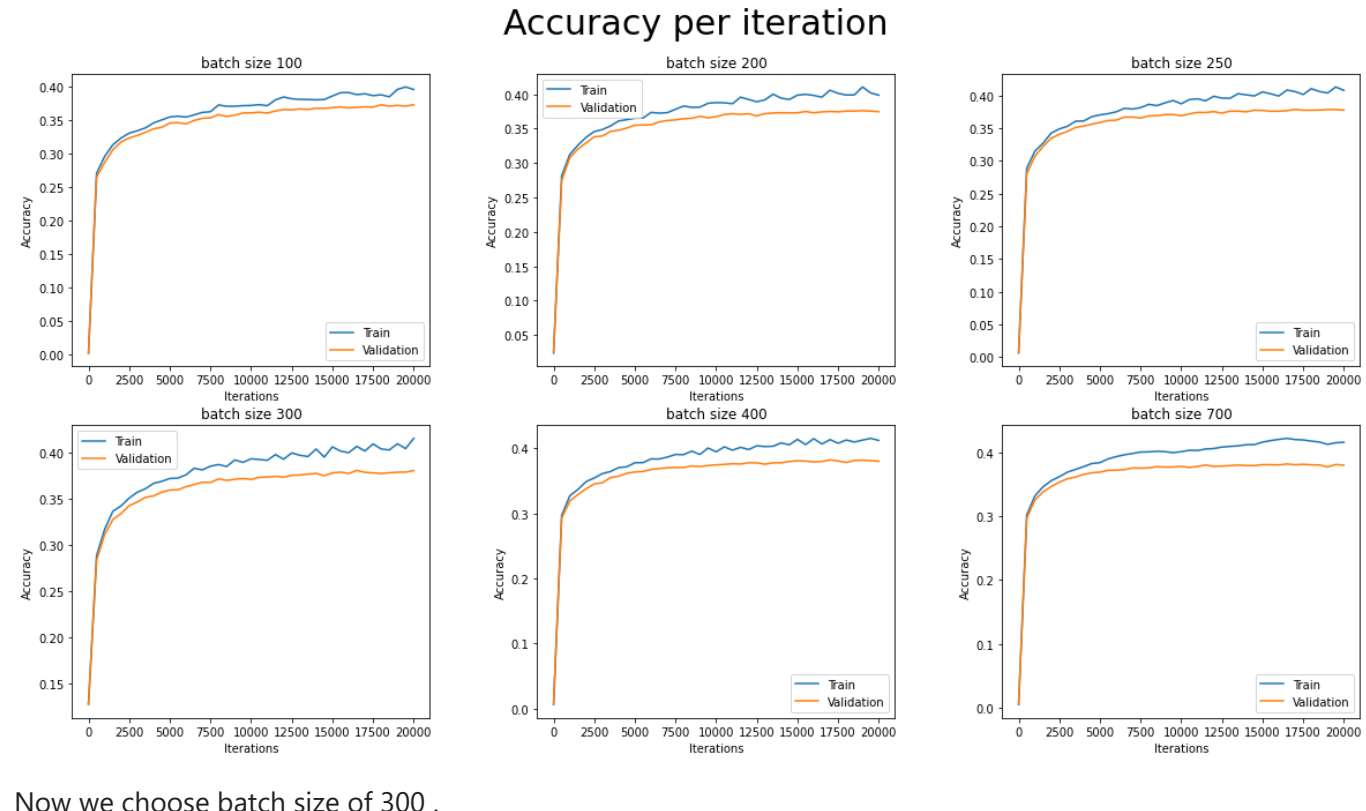
We again choose lr of 0.01 .

In [35]:
```python
results=[]
batches=[100,200,250,300,400,700]
for bs in batches:
  print(make_bold("batch size"),bs)
  pytorch_wordemb= PyTorchWordEmb()
  results.append(run_pytorch_gradient_descent(pytorch_wordemb,
                                              max_iters=20000,batch_size=bs))
  print("====================")
```

```
batch size 100
Iter 0. [Val Acc 0%] [Train Acc 0%, Loss 5.526378]
Iter 500. [Val Acc 27%] [Train Acc 27%, Loss 3.101939]
Iter 1000. [Val Acc 29%] [Train Acc 30%, Loss 2.895482]
Iter 1500. [Val Acc 31%] [Train Acc 31%, Loss 2.806911]
Iter 2000. [Val Acc 32%] [Train Acc 32%, Loss 2.750489]
Iter 2500. [Val Acc 32%] [Train Acc 33%, Loss 2.805528]
Iter 3000. [Val Acc 33%] [Train Acc 33%, Loss 2.529339]
Iter 3500. [Val Acc 33%] [Train Acc 34%, Loss 2.460778]
Iter 4000. [Val Acc 34%] [Train Acc 35%, Loss 2.740052]
Iter 4500. [Val Acc 34%] [Train Acc 35%, Loss 2.790300]
Iter 5000. [Val Acc 35%] [Train Acc 36%, Loss 2.752194]
Iter 5500. [Val Acc 35%] [Train Acc 36%, Loss 2.499235]
Iter 6000. [Val Acc 34%] [Train Acc 36%, Loss 2.765640]
Iter 6500. [Val Acc 35%] [Train Acc 36%, Loss 2.688149]
Iter 7000. [Val Acc 35%] [Train Acc 36%, Loss 3.001824]
Iter 7500. [Val Acc 35%] [Train Acc 36%, Loss 2.789358]
Iter 8000. [Val Acc 36%] [Train Acc 37%, Loss 2.232737]
Iter 8500. [Val Acc 36%] [Train Acc 37%, Loss 2.856111]
Iter 9000. [Val Acc 36%] [Train Acc 37%, Loss 2.430131]
Iter 9500. [Val Acc 36%] [Train Acc 37%, Loss 2.627426]
Iter 10000. [Val Acc 36%] [Train Acc 37%, Loss 2.315635]
Iter 10500. [Val Acc 36%] [Train Acc 37%, Loss 2.620976]
Iter 11000. [Val Acc 36%] [Train Acc 37%, Loss 2.629445]
Iter 11500. [Val Acc 36%] [Train Acc 38%, Loss 2.623104]
Iter 12000. [Val Acc 37%] [Train Acc 39%, Loss 2.407727]
Iter 12500. [Val Acc 37%] [Train Acc 38%, Loss 2.530764]
Iter 13000. [Val Acc 37%] [Train Acc 38%, Loss 2.219435]
Iter 13500. [Val Acc 37%] [Train Acc 38%, Loss 2.812496]
Iter 14000. [Val Acc 37%] [Train Acc 38%, Loss 2.539797]
Iter 14500. [Val Acc 37%] [Train Acc 38%, Loss 2.731133]
Iter 15000. [Val Acc 37%] [Train Acc 39%, Loss 2.333888]
Iter 15500. [Val Acc 37%] [Train Acc 39%, Loss 2.527103]
Iter 16000. [Val Acc 37%] [Train Acc 39%, Loss 2.202881]
Iter 16500. [Val Acc 37%] [Train Acc 39%, Loss 2.409341]
Iter 17000. [Val Acc 37%] [Train Acc 39%, Loss 2.480404]
Iter 17500. [Val Acc 37%] [Train Acc 39%, Loss 2.460396]
Iter 18000. [Val Acc 37%] [Train Acc 39%, Loss 2.764069]
Iter 18500. [Val Acc 37%] [Train Acc 39%, Loss 2.317552]
Iter 19000. [Val Acc 37%] [Train Acc 40%, Loss 2.422202]
Iter 19500. [Val Acc 37%] [Train Acc 40%, Loss 2.597000]
Iter 20000. [Val Acc 37%] [Train Acc 40%, Loss 2.438059]
====================
batch size 200
Iter 0. [Val Acc 2%] [Train Acc 2%, Loss 5.523004]
Iter 500. [Val Acc 27%] [Train Acc 28%, Loss 3.074404]
Iter 1000. [Val Acc 31%] [Train Acc 31%, Loss 3.025978]
Iter 1500. [Val Acc 32%] [Train Acc 33%, Loss 2.896214]
Iter 2000. [Val Acc 33%] [Train Acc 34%, Loss 2.890149]
Iter 2500. [Val Acc 34%] [Train Acc 35%, Loss 2.871179]
Iter 3000. [Val Acc 34%] [Train Acc 35%, Loss 2.869666]
Iter 3500. [Val Acc 35%] [Train Acc 35%, Loss 2.950333]
Iter 4000. [Val Acc 35%] [Train Acc 36%, Loss 2.527406]
Iter 4500. [Val Acc 35%] [Train Acc 36%, Loss 2.782269]
Iter 5000. [Val Acc 35%] [Train Acc 37%, Loss 2.379203]
Iter 5500. [Val Acc 36%] [Train Acc 37%, Loss 2.820544]
Iter 6000. [Val Acc 36%] [Train Acc 37%, Loss 2.504524]
Iter 6500. [Val Acc 36%] [Train Acc 37%, Loss 2.418793]
Iter 7000. [Val Acc 36%] [Train Acc 37%, Loss 2.644671]
Iter 7500. [Val Acc 36%] [Train Acc 38%, Loss 2.466000]
Iter 8000. [Val Acc 36%] [Train Acc 38%, Loss 2.495760]
Iter 8500. [Val Acc 36%] [Train Acc 38%, Loss 2.510692]
Iter 9000. [Val Acc 37%] [Train Acc 38%, Loss 2.704087]
Iter 9500. [Val Acc 37%] [Train Acc 39%, Loss 2.472752]
Iter 10000. [Val Acc 37%] [Train Acc 39%, Loss 2.480747]
Iter 10500. [Val Acc 37%] [Train Acc 39%, Loss 2.482722]
Iter 11000. [Val Acc 37%] [Train Acc 39%, Loss 2.436819]
Iter 11500. [Val Acc 37%] [Train Acc 40%, Loss 2.505828]
Iter 12000. [Val Acc 37%] [Train Acc 39%, Loss 2.430474]
Iter 12500. [Val Acc 37%] [Train Acc 39%, Loss 2.641376]
Iter 13000. [Val Acc 37%] [Train Acc 39%, Loss 2.454337]
Iter 13500. [Val Acc 37%] [Train Acc 40%, Loss 2.408736]
Iter 14000. [Val Acc 37%] [Train Acc 39%, Loss 2.407519]
Iter 14500. [Val Acc 37%] [Train Acc 39%, Loss 2.386906]
Iter 15000. [Val Acc 37%] [Train Acc 40%, Loss 2.259748]
Iter 15500. [Val Acc 37%] [Train Acc 40%, Loss 2.258995]
Iter 16000. [Val Acc 37%] [Train Acc 40%, Loss 2.432252]
Iter 16500. [Val Acc 37%] [Train Acc 40%, Loss 2.271112]
Iter 17000. [Val Acc 37%] [Train Acc 41%, Loss 2.632599]
Iter 17500. [Val Acc 37%] [Train Acc 40%, Loss 2.405840]
Iter 18000. [Val Acc 38%] [Train Acc 40%, Loss 2.202290]
Iter 18500. [Val Acc 38%] [Train Acc 40%, Loss 2.235475]
Iter 19000. [Val Acc 38%] [Train Acc 41%, Loss 2.346335]
Iter 19500. [Val Acc 38%] [Train Acc 40%, Loss 2.322599]
Iter 20000. [Val Acc 37%] [Train Acc 40%, Loss 2.491361]
====================
batch size 250
Iter 0. [Val Acc 1%] [Train Acc 1%, Loss 5.534055]
Iter 500. [Val Acc 28%] [Train Acc 29%, Loss 3.262550]
Iter 1000. [Val Acc 31%] [Train Acc 31%, Loss 2.810121]
Iter 1500. [Val Acc 32%] [Train Acc 33%, Loss 3.011421]
Iter 2000. [Val Acc 33%] [Train Acc 34%, Loss 2.886730]
Iter 2500. [Val Acc 34%] [Train Acc 35%, Loss 2.778221]
Iter 3000. [Val Acc 34%] [Train Acc 35%, Loss 2.492815]
Iter 3500. [Val Acc 35%] [Train Acc 36%, Loss 2.688294]
Iter 4000. [Val Acc 35%] [Train Acc 36%, Loss 2.625030]
Iter 4500. [Val Acc 36%] [Train Acc 37%, Loss 2.718032]
Iter 5000. [Val Acc 36%] [Train Acc 37%, Loss 2.729337]
Iter 5500. [Val Acc 36%] [Train Acc 37%, Loss 2.586135]
Iter 6000. [Val Acc 36%] [Train Acc 37%, Loss 2.560919]
Iter 6500. [Val Acc 37%] [Train Acc 38%, Loss 2.548860]
Iter 7000. [Val Acc 37%] [Train Acc 38%, Loss 2.266838]
Iter 7500. [Val Acc 37%] [Train Acc 38%, Loss 2.485837]
Iter 8000. [Val Acc 37%] [Train Acc 39%, Loss 2.404397]
Iter 8500. [Val Acc 37%] [Train Acc 38%, Loss 2.440297]
Iter 9000. [Val Acc 37%] [Train Acc 39%, Loss 2.528423]
Iter 9500. [Val Acc 37%] [Train Acc 39%, Loss 2.635228]
Iter 10000. [Val Acc 37%] [Train Acc 39%, Loss 2.248409]
Iter 10500. [Val Acc 37%] [Train Acc 39%, Loss 2.587329]
Iter 11000. [Val Acc 37%] [Train Acc 39%, Loss 2.487069]
Iter 11500. [Val Acc 37%] [Train Acc 39%, Loss 2.373604]
Iter 12000. [Val Acc 38%] [Train Acc 40%, Loss 2.454542]
Iter 12500. [Val Acc 37%] [Train Acc 40%, Loss 2.459876]
Iter 13000. [Val Acc 38%] [Train Acc 40%, Loss 2.271512]
Iter 13500. [Val Acc 38%] [Train Acc 40%, Loss 2.365951]
Iter 14000. [Val Acc 37%] [Train Acc 40%, Loss 2.163965]
Iter 14500. [Val Acc 38%] [Train Acc 40%, Loss 2.075785]
Iter 15000. [Val Acc 38%] [Train Acc 41%, Loss 2.313109]
Iter 15500. [Val Acc 38%] [Train Acc 40%, Loss 2.230323]
Iter 16000. [Val Acc 38%] [Train Acc 40%, Loss 2.463951]
Iter 16500. [Val Acc 38%] [Train Acc 41%, Loss 2.338200]
Iter 17000. [Val Acc 38%] [Train Acc 41%, Loss 2.229771]
Iter 17500. [Val Acc 38%] [Train Acc 40%, Loss 2.374499]
Iter 18000. [Val Acc 38%] [Train Acc 41%, Loss 2.052632]
Iter 18500. [Val Acc 38%] [Train Acc 41%, Loss 2.341962]
Iter 19000. [Val Acc 38%] [Train Acc 40%, Loss 2.367618]
Iter 19500. [Val Acc 38%] [Train Acc 41%, Loss 2.279229]
Iter 20000. [Val Acc 38%] [Train Acc 41%, Loss 2.472797]
====================
batch size 300
Iter 0. [Val Acc 13%] [Train Acc 13%, Loss 5.521667]
Iter 500. [Val Acc 28%] [Train Acc 29%, Loss 3.209822]
Iter 1000. [Val Acc 31%] [Train Acc 32%, Loss 2.953104]
Iter 1500. [Val Acc 33%] [Train Acc 34%, Loss 2.975886]
Iter 2000. [Val Acc 33%] [Train Acc 34%, Loss 2.792212]
Iter 2500. [Val Acc 34%] [Train Acc 35%, Loss 2.676473]
Iter 3000. [Val Acc 35%] [Train Acc 36%, Loss 2.661474]
Iter 3500. [Val Acc 35%] [Train Acc 36%, Loss 2.658430]
Iter 4000. [Val Acc 35%] [Train Acc 37%, Loss 2.644662]
Iter 4500. [Val Acc 36%] [Train Acc 37%, Loss 2.757029]
Iter 5000. [Val Acc 36%] [Train Acc 37%, Loss 2.617356]
Iter 5500. [Val Acc 36%] [Train Acc 37%, Loss 2.531268]
Iter 6000. [Val Acc 36%] [Train Acc 38%, Loss 2.572738]
Iter 6500. [Val Acc 37%] [Train Acc 38%, Loss 2.564902]
Iter 7000. [Val Acc 37%] [Train Acc 38%, Loss 2.478485]
Iter 7500. [Val Acc 37%] [Train Acc 39%, Loss 2.542446]
Iter 8000. [Val Acc 37%] [Train Acc 39%, Loss 2.561606]
Iter 8500. [Val Acc 37%] [Train Acc 39%, Loss 2.623568]
Iter 9000. [Val Acc 37%] [Train Acc 39%, Loss 2.460914]
Iter 9500. [Val Acc 37%] [Train Acc 39%, Loss 2.480378]
Iter 10000. [Val Acc 37%] [Train Acc 39%, Loss 2.293827]
Iter 10500. [Val Acc 37%] [Train Acc 39%, Loss 2.426989]
Iter 11000. [Val Acc 37%] [Train Acc 39%, Loss 2.324562]
Iter 11500. [Val Acc 37%] [Train Acc 40%, Loss 2.598696]
Iter 12000. [Val Acc 37%] [Train Acc 39%, Loss 2.280764]
Iter 12500. [Val Acc 38%] [Train Acc 40%, Loss 2.354424]
Iter 13000. [Val Acc 38%] [Train Acc 40%, Loss 2.391370]
Iter 13500. [Val Acc 38%] [Train Acc 40%, Loss 2.250719]
Iter 14000. [Val Acc 38%] [Train Acc 40%, Loss 2.369291]
Iter 14500. [Val Acc 38%] [Train Acc 40%, Loss 2.267428]
Iter 15000. [Val Acc 38%] [Train Acc 41%, Loss 2.655765]
Iter 15500. [Val Acc 38%] [Train Acc 40%, Loss 2.398968]
Iter 16000. [Val Acc 38%] [Train Acc 40%, Loss 2.224263]
Iter 16500. [Val Acc 38%] [Train Acc 41%, Loss 2.199419]
Iter 17000. [Val Acc 38%] [Train Acc 40%, Loss 2.485811]
Iter 17500. [Val Acc 38%] [Train Acc 41%, Loss 2.405241]
Iter 18000. [Val Acc 38%] [Train Acc 40%, Loss 2.192171]
Iter 18500. [Val Acc 38%] [Train Acc 40%, Loss 2.403778]
Iter 19000. [Val Acc 38%] [Train Acc 41%, Loss 2.469780]
Iter 19500. [Val Acc 38%] [Train Acc 40%, Loss 2.426896]
Iter 20000. [Val Acc 38%] [Train Acc 42%, Loss 2.462701]
====================
batch size 400
Iter 0. [Val Acc 1%] [Train Acc 1%, Loss 5.521790]
Iter 500. [Val Acc 29%] [Train Acc 30%, Loss 3.044958]
Iter 1000. [Val Acc 32%] [Train Acc 33%, Loss 2.971767]
Iter 1500. [Val Acc 33%] [Train Acc 34%, Loss 2.874527]
Iter 2000. [Val Acc 34%] [Train Acc 35%, Loss 2.657554]
Iter 2500. [Val Acc 35%] [Train Acc 35%, Loss 2.532186]
Iter 3000. [Val Acc 35%] [Train Acc 36%, Loss 2.593323]
Iter 3500. [Val Acc 35%] [Train Acc 36%, Loss 2.723529]
Iter 4000. [Val Acc 36%] [Train Acc 37%, Loss 2.596775]
Iter 4500. [Val Acc 36%] [Train Acc 37%, Loss 2.565476]
Iter 5000. [Val Acc 36%] [Train Acc 38%, Loss 2.549410]
Iter 5500. [Val Acc 36%] [Train Acc 38%, Loss 2.597364]
Iter 6000. [Val Acc 37%] [Train Acc 38%, Loss 2.657845]
Iter 6500. [Val Acc 37%] [Train Acc 38%, Loss 2.562326]
Iter 7000. [Val Acc 37%] [Train Acc 39%, Loss 2.516499]
Iter 7500. [Val Acc 37%] [Train Acc 39%, Loss 2.483115]
Iter 8000. [Val Acc 37%] [Train Acc 39%, Loss 2.647861]
Iter 8500. [Val Acc 37%] [Train Acc 40%, Loss 2.387849]
Iter 9000. [Val Acc 37%] [Train Acc 39%, Loss 2.377347]
Iter 9500. [Val Acc 37%] [Train Acc 40%, Loss 2.408330]
Iter 10000. [Val Acc 37%] [Train Acc 39%, Loss 2.469422]
Iter 10500. [Val Acc 38%] [Train Acc 40%, Loss 2.434742]
Iter 11000. [Val Acc 38%] [Train Acc 40%, Loss 2.379071]
Iter 11500. [Val Acc 38%] [Train Acc 40%, Loss 2.492099]
Iter 12000. [Val Acc 38%] [Train Acc 40%, Loss 2.182925]
Iter 12500. [Val Acc 38%] [Train Acc 40%, Loss 2.339549]
Iter 13000. [Val Acc 38%] [Train Acc 40%, Loss 2.234231]
Iter 13500. [Val Acc 38%] [Train Acc 40%, Loss 2.274555]
Iter 14000. [Val Acc 38%] [Train Acc 41%, Loss 2.425606]
Iter 14500. [Val Acc 38%] [Train Acc 41%, Loss 2.574355]
Iter 15000. [Val Acc 38%] [Train Acc 41%, Loss 2.532904]
Iter 15500. [Val Acc 38%] [Train Acc 41%, Loss 2.176127]
Iter 16000. [Val Acc 38%] [Train Acc 41%, Loss 2.226541]
Iter 16500. [Val Acc 38%] [Train Acc 41%, Loss 2.332900]
Iter 17000. [Val Acc 38%] [Train Acc 41%, Loss 2.433203]
Iter 17500. [Val Acc 38%] [Train Acc 41%, Loss 2.340644]
Iter 18000. [Val Acc 38%] [Train Acc 41%, Loss 2.352070]
Iter 18500. [Val Acc 38%] [Train Acc 41%, Loss 2.363289]
Iter 19000. [Val Acc 38%] [Train Acc 41%, Loss 2.287627]
Iter 19500. [Val Acc 38%] [Train Acc 42%, Loss 2.386053]
Iter 20000. [Val Acc 38%] [Train Acc 41%, Loss 2.406852]
====================
batch size 700
Iter 0. [Val Acc 0%] [Train Acc 0%, Loss 5.530474]
Iter 500. [Val Acc 30%] [Train Acc 30%, Loss 3.113223]
Iter 1000. [Val Acc 33%] [Train Acc 33%, Loss 2.881227]
Iter 1500. [Val Acc 34%] [Train Acc 35%, Loss 2.914876]
Iter 2000. [Val Acc 35%] [Train Acc 36%, Loss 2.752861]
Iter 2500. [Val Acc 35%] [Train Acc 36%, Loss 2.578657]
Iter 3000. [Val Acc 36%] [Train Acc 37%, Loss 2.606543]
Iter 3500. [Val Acc 36%] [Train Acc 37%, Loss 2.660316]
Iter 4000. [Val Acc 37%] [Train Acc 38%, Loss 2.560211]
Iter 4500. [Val Acc 37%] [Train Acc 38%, Loss 2.448159]
Iter 5000. [Val Acc 37%] [Train Acc 38%, Loss 2.459700]
Iter 5500. [Val Acc 37%] [Train Acc 39%, Loss 2.311467]
Iter 6000. [Val Acc 37%] [Train Acc 39%, Loss 2.433672]
Iter 6500. [Val Acc 37%] [Train Acc 40%, Loss 2.457984]
Iter 7000. [Val Acc 38%] [Train Acc 40%, Loss 2.368914]
Iter 7500. [Val Acc 38%] [Train Acc 40%, Loss 2.373318]
Iter 8000. [Val Acc 38%] [Train Acc 40%, Loss 2.452297]
Iter 8500. [Val Acc 38%] [Train Acc 40%, Loss 2.364030]
Iter 9000. [Val Acc 38%] [Train Acc 40%, Loss 2.506859]
Iter 9500. [Val Acc 38%] [Train Acc 40%, Loss 2.512262]
Iter 10000. [Val Acc 38%] [Train Acc 40%, Loss 2.399554]
Iter 10500. [Val Acc 38%] [Train Acc 40%, Loss 2.375337]
Iter 11000. [Val Acc 38%] [Train Acc 40%, Loss 2.384466]
Iter 11500. [Val Acc 38%] [Train Acc 41%, Loss 2.316691]
Iter 12000. [Val Acc 38%] [Train Acc 41%, Loss 2.464194]
Iter 12500. [Val Acc 38%] [Train Acc 41%, Loss 2.418828]
Iter 13000. [Val Acc 38%] [Train Acc 41%, Loss 2.272688]
Iter 13500. [Val Acc 38%] [Train Acc 41%, Loss 2.289494]
Iter 14000. [Val Acc 38%] [Train Acc 41%, Loss 2.334672]
Iter 14500. [Val Acc 38%] [Train Acc 41%, Loss 2.326525]
Iter 15000. [Val Acc 38%] [Train Acc 42%, Loss 2.203389]
Iter 15500. [Val Acc 38%] [Train Acc 42%, Loss 2.197224]
Iter 16000. [Val Acc 38%] [Train Acc 42%, Loss 2.314939]
Iter 16500. [Val Acc 38%] [Train Acc 42%, Loss 2.238414]
Iter 17000. [Val Acc 38%] [Train Acc 42%, Loss 2.305051]
Iter 17500. [Val Acc 38%] [Train Acc 42%, Loss 2.309361]
Iter 18000. [Val Acc 38%] [Train Acc 42%, Loss 2.180372]
Iter 18500. [Val Acc 38%] [Train Acc 42%, Loss 2.211914]
Iter 19000. [Val Acc 38%] [Train Acc 41%, Loss 2.237951]
Iter 19500. [Val Acc 38%] [Train Acc 42%, Loss 2.284401]
Iter 20000. [Val Acc 38%] [Train Acc 42%, Loss 2.258657]
====================
```

In [36]:
```python
plot_learning_curve(results,rows=2,cols=3,parameters=batches,parameter_name="batch si
```





Now we choose batch size of 300 .

In [38]:
```
# best model
pytorch_wordemb= PyTorchWordEmb()
result=(run_pytorch_gradient_descent(pytorch_wordemb,
                                     max_iters=20000,batch_size=300))
```

```
Iter 0. [Val Acc 2%] [Train Acc 2%, Loss 5.518041]
Iter 500. [Val Acc 28%] [Train Acc 29%, Loss 3.215420]
Iter 1000. [Val Acc 31%] [Train Acc 32%, Loss 2.944386]
Iter 1500. [Val Acc 33%] [Train Acc 34%, Loss 2.961300]
Iter 2000. [Val Acc 33%] [Train Acc 34%, Loss 2.780610]
Iter 2500. [Val Acc 34%] [Train Acc 35%, Loss 2.710120]
Iter 3000. [Val Acc 35%] [Train Acc 36%, Loss 2.656940]
Iter 3500. [Val Acc 35%] [Train Acc 36%, Loss 2.661702]
Iter 4000. [Val Acc 35%] [Train Acc 37%, Loss 2.624449]
Iter 4500. [Val Acc 36%] [Train Acc 37%, Loss 2.745435]
Iter 5000. [Val Acc 36%] [Train Acc 37%, Loss 2.633929]
Iter 5500. [Val Acc 36%] [Train Acc 38%, Loss 2.522348]
Iter 6000. [Val Acc 36%] [Train Acc 38%, Loss 2.546383]
Iter 6500. [Val Acc 37%] [Train Acc 38%, Loss 2.577958]
Iter 7000. [Val Acc 37%] [Train Acc 38%, Loss 2.508410]
Iter 7500. [Val Acc 37%] [Train Acc 39%, Loss 2.533877]
Iter 8000. [Val Acc 37%] [Train Acc 39%, Loss 2.613716]
Iter 8500. [Val Acc 37%] [Train Acc 39%, Loss 2.581933]
Iter 9000. [Val Acc 37%] [Train Acc 39%, Loss 2.459998]
Iter 9500. [Val Acc 37%] [Train Acc 39%, Loss 2.522073]
Iter 10000. [Val Acc 37%] [Train Acc 39%, Loss 2.320807]
Iter 10500. [Val Acc 37%] [Train Acc 39%, Loss 2.424054]
Iter 11000. [Val Acc 38%] [Train Acc 39%, Loss 2.315505]
Iter 11500. [Val Acc 37%] [Train Acc 40%, Loss 2.657997]
Iter 12000. [Val Acc 38%] [Train Acc 39%, Loss 2.254560]
Iter 12500. [Val Acc 38%] [Train Acc 40%, Loss 2.381103]
Iter 13000. [Val Acc 38%] [Train Acc 40%, Loss 2.432642]
Iter 13500. [Val Acc 38%] [Train Acc 40%, Loss 2.206470]
Iter 14000. [Val Acc 38%] [Train Acc 40%, Loss 2.385060]
Iter 14500. [Val Acc 38%] [Train Acc 40%, Loss 2.265549]
Iter 15000. [Val Acc 38%] [Train Acc 41%, Loss 2.649423]
Iter 15500. [Val Acc 38%] [Train Acc 40%, Loss 2.392514]
Iter 16000. [Val Acc 38%] [Train Acc 40%, Loss 2.252542]
Iter 16500. [Val Acc 38%] [Train Acc 41%, Loss 2.212328]
Iter 17000. [Val Acc 38%] [Train Acc 40%, Loss 2.485659]
Iter 17500. [Val Acc 38%] [Train Acc 41%, Loss 2.416045]
Iter 18000. [Val Acc 38%] [Train Acc 40%, Loss 2.225931]
Iter 18500. [Val Acc 38%] [Train Acc 40%, Loss 2.386328]
Iter 19000. [Val Acc 38%] [Train Acc 41%, Loss 2.456148]
Iter 19500. [Val Acc 38%] [Train Acc 40%, Loss 2.434269]
Iter 20000. [Val Acc 38%] [Train Acc 42%, Loss 2.480032]
```

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%.

In [39]:
```
print("you are a",make_bold(make_underline(make_prediction_torch(pytorch_wordemb, ['yo
print("few companies shows" , make_bold(make_underline(make_prediction_torch(pytorch_w
print("there are no",make_bold(make_underline(make_prediction_torch(pytorch_wordemb,
print("yesterday i was",make_bold(make_underline(make_prediction_torch(pytorch_wordemb
print("the game had",make_bold(make_underline(make_prediction_torch(pytorch_wordemb,
print("yesterday the federal",make_bold(make_underline(make_prediction_torch(pytorch_w
```

```
you are a good
few companies shows .
there are no other
yesterday i was nt
the game had to
yesterday the federal government
```

We got the same result as in the preview model!

Part (c) -- 4%

Report the test accuracy of your model

In [40]:
```
estimate_accuracy_torch(pytorch_wordemb, test4grams)
```

Out[40]: `0.3844901367289427`

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.

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".

In [45]:
```
word_emb_weights = list(pytorch_wordemb.word_emb_layer.parameters())[0]
word_emb = word_emb_weights.detach().numpy().T
```

Explaination: each row of word_emb is the result of multiply and sums all weights with the i'th input for the layer, which is a one hot represntation of the i'th word. But since the one-hot representation will gives us vectors of zeros and only 1 in the i'th index we get that each row is the vector represening of the word itself .

For example, `word_emb[vocab_stoi["any"],:]` contains the vector representation of the word "any" .

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 [43]:
```
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(make_bold("similarity between 'any' and 'many'"),similarities[vocab_stoi['any'],
print(make_bold("similarity between 'any' and 'government'"),similarities[vocab_stoi[
```

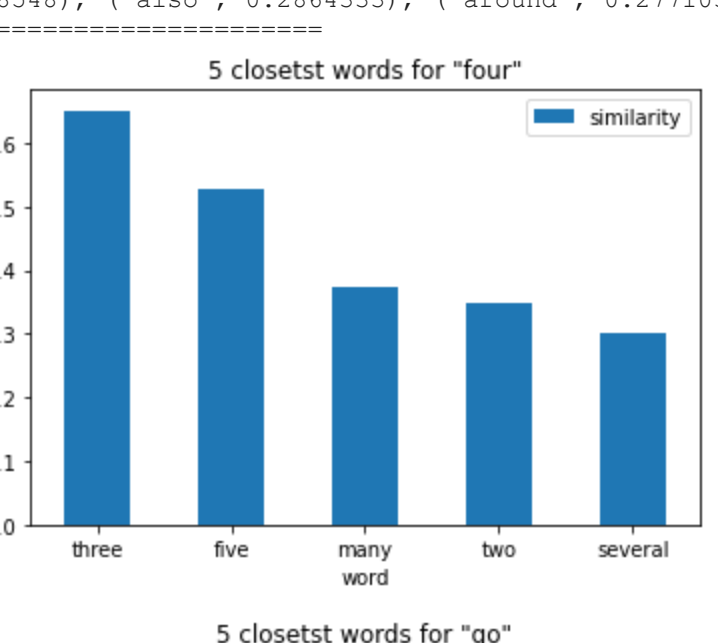
```
similarity between 'any' and 'many' 0.18362503
similarity between 'any' and 'government' 0.1561606
```

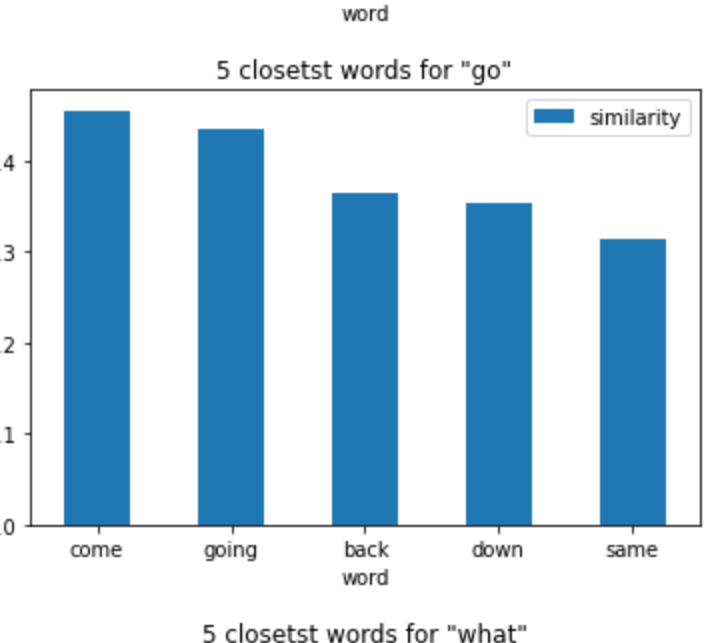
Compute the 5 closest words to the following words:

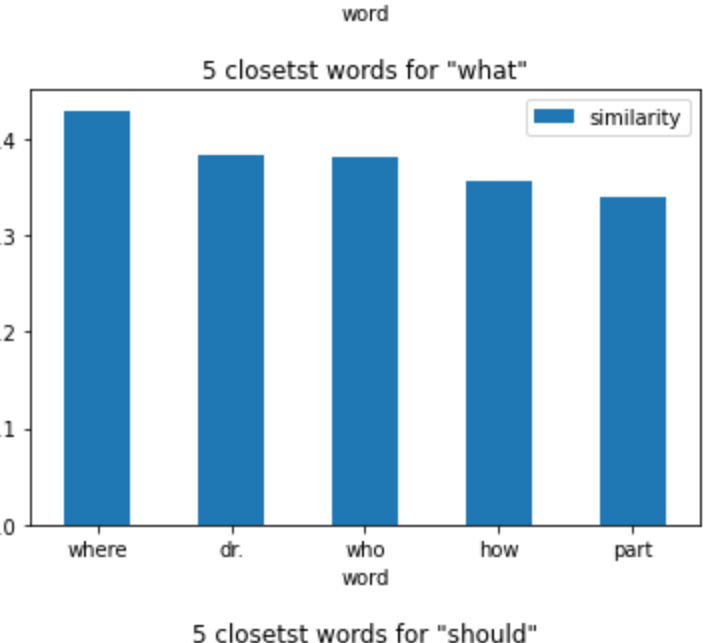
- "four"
- "go"
- "what"
- "should"
- "school"
- "your"
- "yesterday"
- "not"

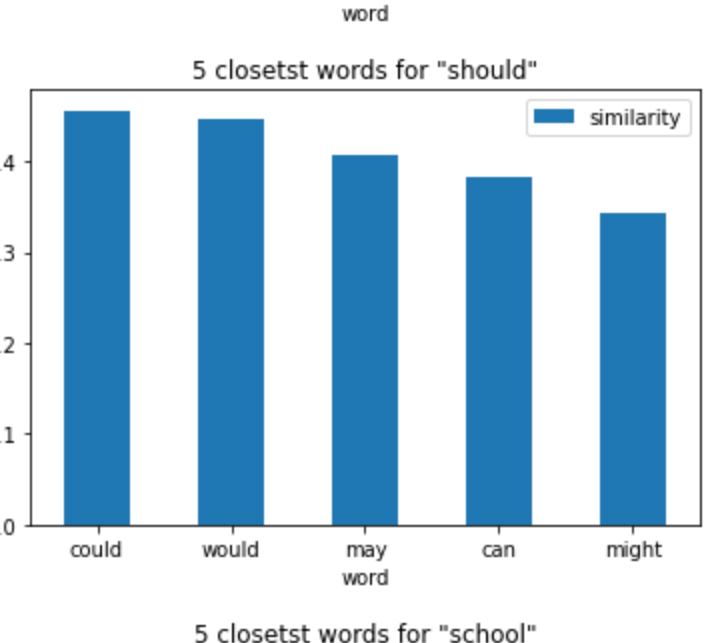
In [46]:
```
for word in ["four","go","what","should","school","your","yesterday","not"]:
    words_similarities=[(w,similarities[vocab_stoi[word], vocab_stoi[w]]) for w in vocab
    similarities_dict = {}
    [similarities_dict.update({k:v}) for k,v in words_similarities]
    c=Counter(similarities_dict)
    five_most_common=c.most_common(5)
    df=pd.DataFrame(five_most_common,columns=['word','similarity'])
    ax=df.plot.bar(x='word', y='similarity', rot=0)
    ax.set_title("5 closetst words for \""+word+"\"")
    print(make_bold("5 closetst words for \""+word+"\":"),five_most_common)
    print("=====================")
```

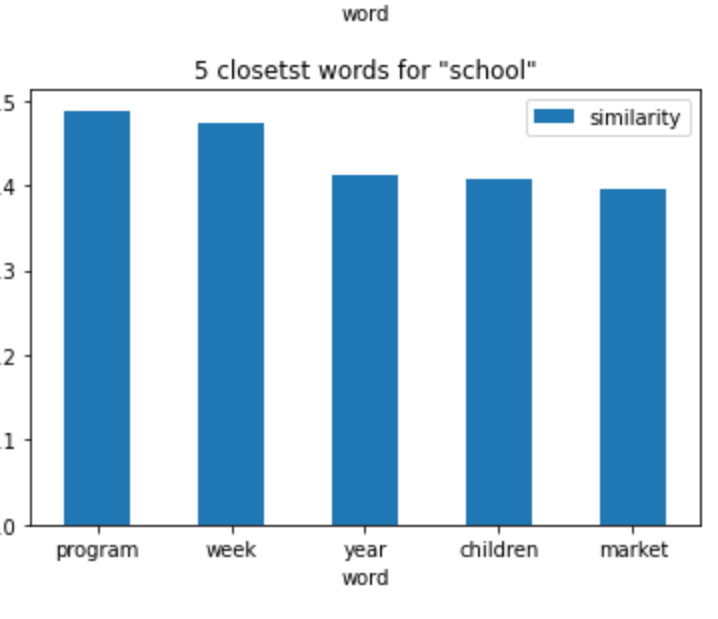
```
5 closetst words for "four": [('three', 0.6514785), ('five', 0.5263721), ('many', 0.37
343597), ('two', 0.34956172), ('several', 0.30247074)]
=====================
5 closetst words for "go": [('come', 0.45551512), ('going', 0.4357491), ('back', 0.365
01148), ('down', 0.3545441), ('same', 0.31410465)]
=====================
5 closetst words for "what": [('where', 0.42930898), ('dr.', 0.3823012), ('who', 0.380
6837), ('how', 0.35584918), ('part', 0.34056073)]
=====================
5 closetst words for "should": [('could', 0.45598066), ('would', 0.4472415), ('may',
 0.40604687), ('can', 0.38214138), ('might', 0.34271118)]
=====================
5 closetst words for "school": [('program', 0.4882231), ('week', 0.47374997), ('year',
 0.4128692), ('children', 0.4074157), ('market', 0.3952649)]
=====================
5 closetst words for "your": [('my', 0.5222146), ('each', 0.49735525), ('federal', 0.3
0004522), ('other', 0.2969315), ('right', 0.28340703)]
=====================
5 closetst words for "yesterday": [('week', 0.44687232), (')', 0.4189693), ('ago', 0.4
180559), ('said', 0.38785744), ('well', 0.3846288)]
=====================
5 closetst words for "not": [('nt', 0.47753257), ('never', 0.3488891), ('might', 0.342
98548), ('also', 0.2864333), ('around', 0.27710342)]
=====================
```

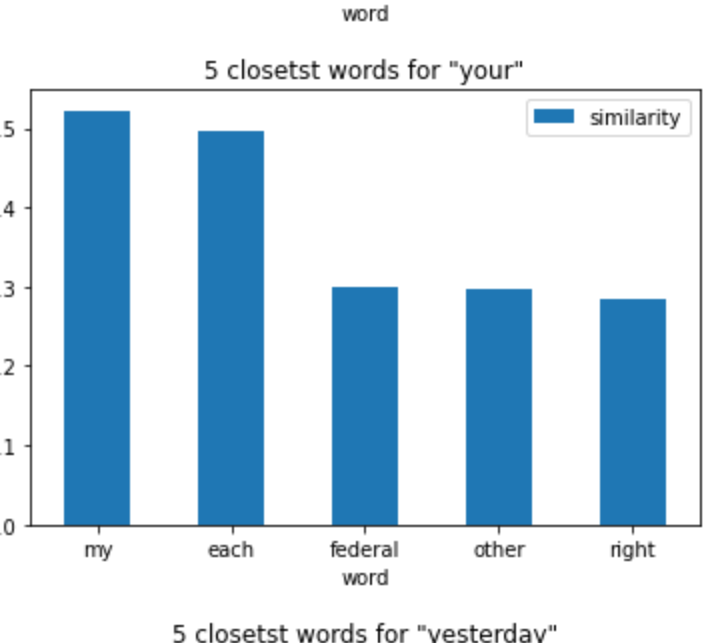


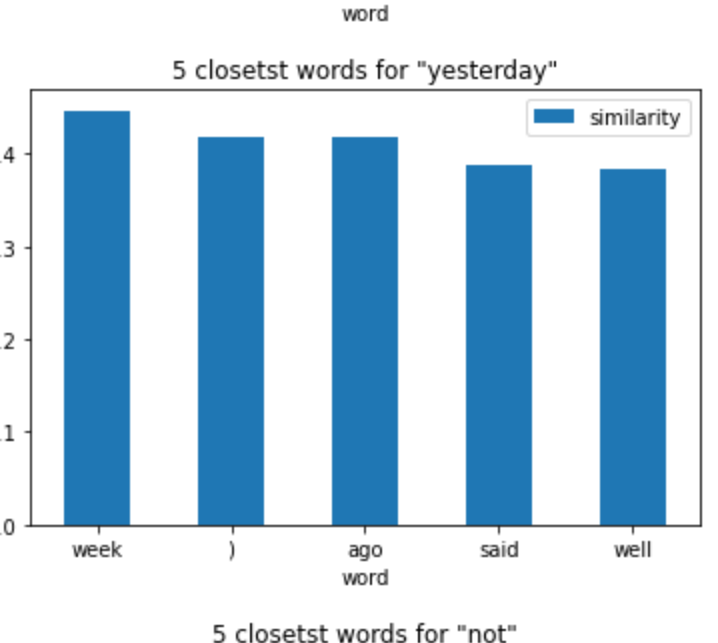


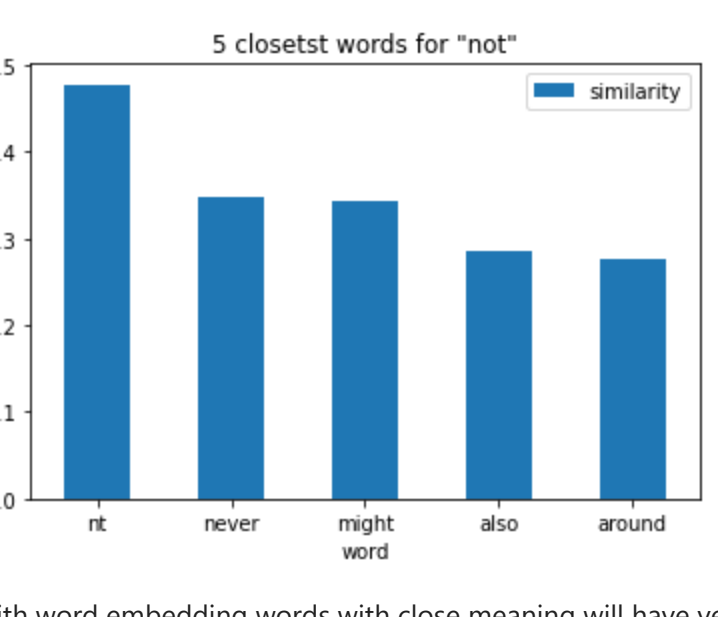












With word embedding words with close meaning will have vectors with close distance between them . Hence for example the 5 closest word to "four" are "three","five","many","several" and two . All of this options can replace the word four in a senetnce and the sentnce will still be reasonable and with similiar meaning.

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.

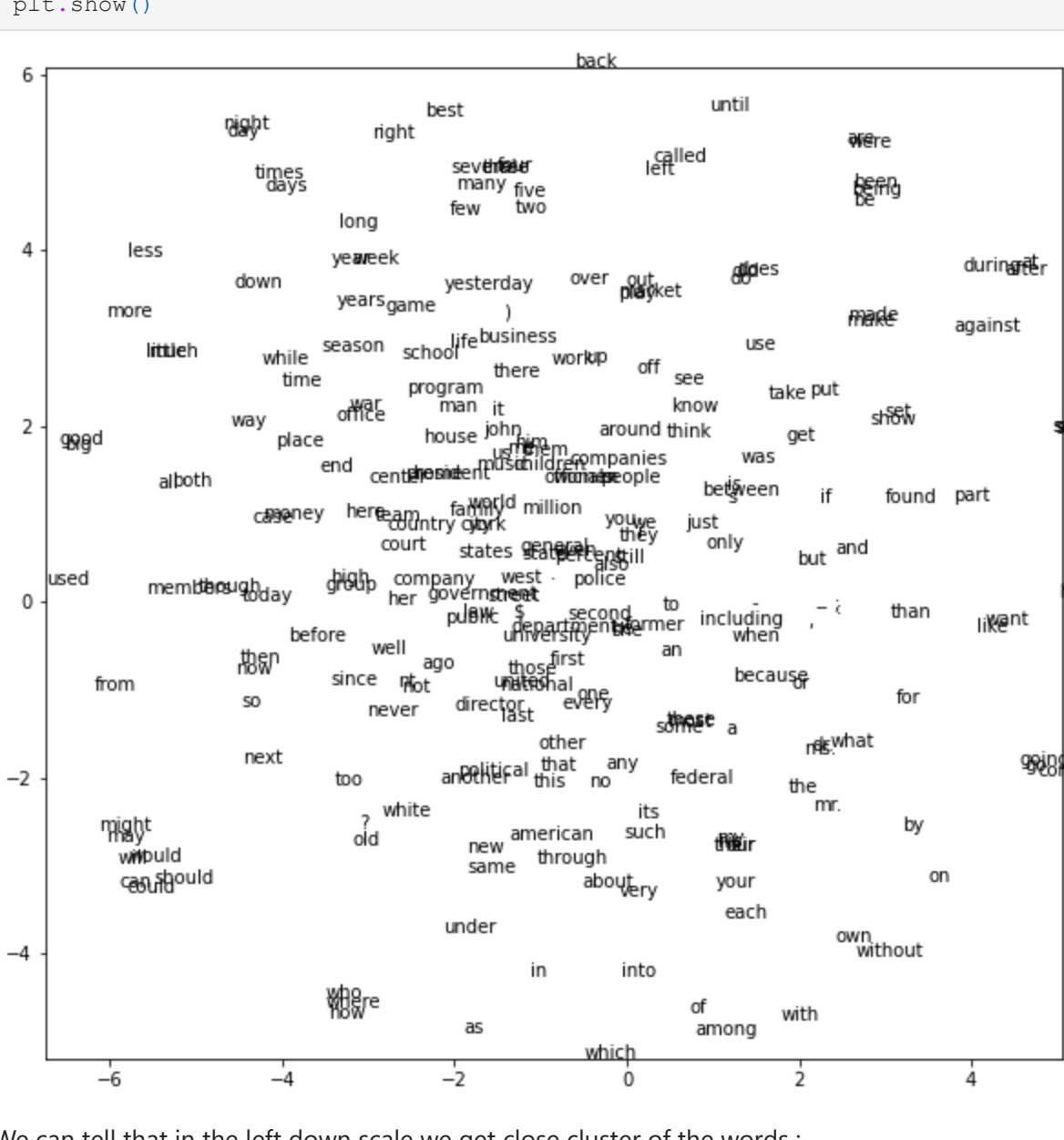
Look at the plot and find at least two clusters of related words.

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 [47]:
```
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()
```



We can tell that in the left down scale we get close cluster of the words : "might","may","would","should","can" and "could" .
All those words represent a meaning somethin we are not sure about in the present or future: "I might eat this cake" means I can eat the cake, but I'm not sure I will do it. Same as "I could have won the competition" - that mean I think I have the ability to win but it does not mean i won.

In adition above we have the cluster "four","several","many","few","five","two" - which all have the same meaning of describe something that came in plural (more than one object).