Open in Colab

MNIST Digits Classification using Neural Networks

Mount your drive in order to run locally with colab

In [52]:
```
from google.colab import drive
drive.mount('/content/gdrive')
```

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

download & load the MNIST dataset.

*just run the next two cells and observe the outputs (shift&enter)

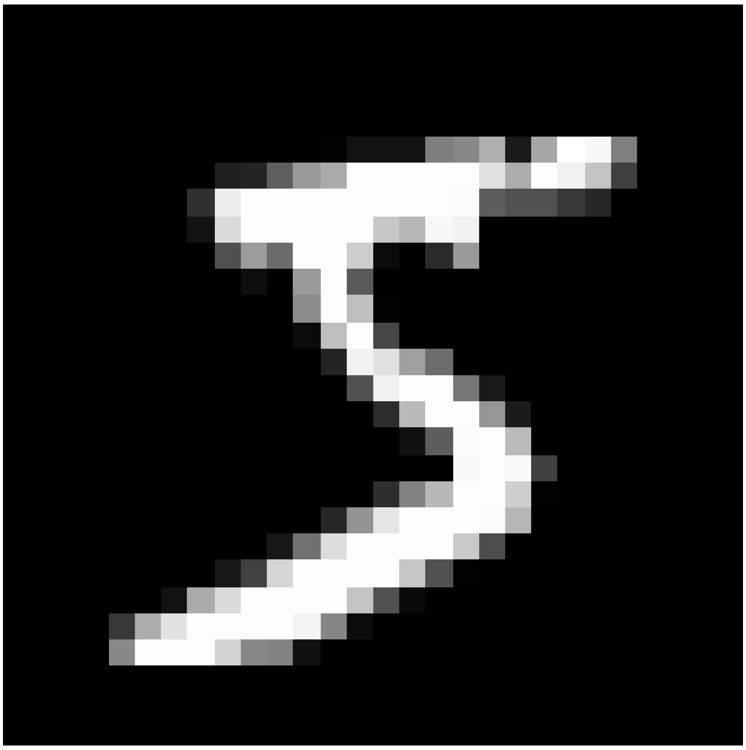
In [53]:
```
#importing modules that will be in use
%matplotlib inline
import os
import numpy as np
import matplotlib.pyplot as plt
import urllib.request
import gzip
import pickle
from PIL import Image
import random
import numpy as np

def _download(file_name):
    file_path = os.path.join(dataset_dir,file_name)

    if os.path.exists(file_path):
        return

    print("Downloading " + file_name + " ... ")
    urllib.request.urlretrieve(url_base + file_name, file_name)
    print("Done")

def download_mnist():
    for v in key_file.values():
       _download(v)

def _load_label(file_name):
    file_path =  os.path.join(dataset_dir, file_name)

    print("Converting " + file_name + " to NumPy Array ...")
    with gzip.open(file_path, 'rb') as f:
            labels = np.frombuffer(f.read(), np.uint8, offset=8)
    print("Done")

    return labels

def _load_img(file_name):
    file_path = os.path.join(dataset_dir,file_name)
```

```
    print("Converting " + file_name + " to NumPy Array ...")
    with gzip.open(file_path, 'rb') as f:
            data = np.frombuffer(f.read(), np.uint8, offset=16)
    data = data.reshape(-1, img_size)
    print("Done")

    return data

def _convert_numpy():
    dataset = {}
    dataset['train_img'] =  _load_img(key_file['train_img'])
    dataset['train_label'] = _load_label(key_file['train_label'])
    dataset['test_img'] = _load_img(key_file['test_img'])
    dataset['test_label'] = _load_label(key_file['test_label'])

    return dataset

def init_mnist():
    download_mnist()
    dataset = _convert_numpy()
    print("Creating pickle file ...")
    with open(save_file, 'wb') as f:
        pickle.dump(dataset, f, -1)
    print("Done")

def _change_one_hot_label(X):
    T = np.zeros((X.size, 10))
    for idx, row in enumerate(T):
        row[X[idx]] = 1

    return T

def load_mnist(normalize=True, flatten=True, one_hot_label=False):
    """
    Parameters
    ----------
    normalize : Normalize the pixel values
    flatten : Flatten the images as one array
    one_hot_label : Encode the labels as a one-hot array

    Returns
    -------
    (Trainig Image, Training Label), (Test Image, Test Label)
    """
    if not os.path.exists(save_file):
        init_mnist()

    with open(save_file, 'rb') as f:
        dataset = pickle.load(f)

    if normalize:
        for key in ('train_img', 'test_img'):
            dataset[key] = dataset[key].astype(np.float32)
            dataset[key] /= 255.0

    if not flatten:
         for key in ('train_img', 'test_img'):
            dataset[key] = dataset[key].reshape(-1, 1, 28, 28)

    if one_hot_label:
        dataset['train_label'] = _change_one_hot_label(dataset['train_label'])
        dataset['test_label'] = _change_one_hot_label(dataset['test_label'])

    return (dataset['train_img'], dataset['train_label']), (dataset['test_img'], d
```

```
# Load the MNIST dataset
url_base = 'http://yann.lecun.com/exdb/mnist/'
key_file = {
    'train_img':'train-images-idx3-ubyte.gz',
    'train_label':'train-labels-idx1-ubyte.gz',
    'test_img':'t10k-images-idx3-ubyte.gz',
    'test_label':'t10k-labels-idx1-ubyte.gz'
}

dataset_dir = '/content'
save_file = dataset_dir + "/mnist.pkl"

train_num = 60000
test_num = 10000
img_dim = (1, 28, 28)
img_size = 784

(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True, flatten=True)


# printing data shape

print('the training data set contains '+ str(x_train.shape[0]) + ' samples')

img = x_train[0]
label = t_train[0]

img = img.reshape(28, 28)
print('each sample image from the training data set is a column-stacked grayscale
      + '\n this vectorized arrangement of the data is suitable for a Fully-Connec
print('these column-stacked images can be reshaped to an image of ' +str(img.shape

# printing a sample from the dataset

plt.imshow(img, cmap='gray')
plt.axis('off')
plt.title('The ground truth label of this image is '+str(label))
plt.show()
```

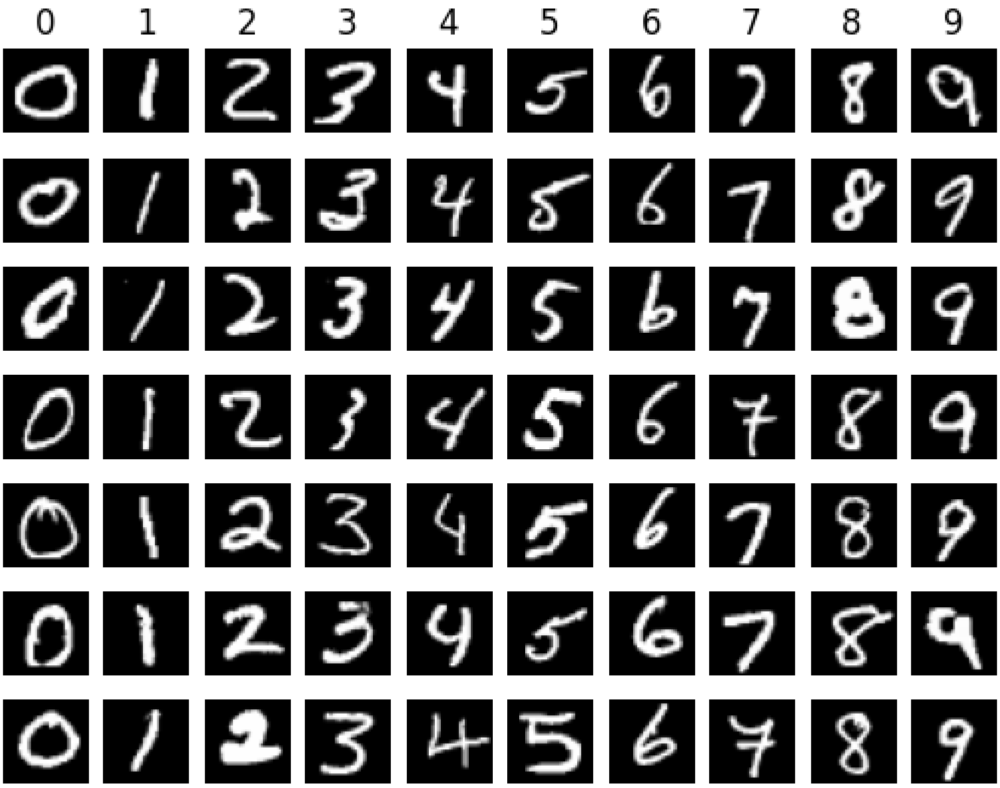
```
the training data set contains 60000 samples
each sample image from the training data set is a column-stacked grayscale image of
784 pixels
 this vectorized arrangement of the data is suitable for a Fully-Connected NN (as ap
posed to a Convolutional NN)
these column-stacked images can be reshaped to an image of (28, 28) pixels
```

The ground truth label of this image is 5



In [54]:

```
# Visualize some examples from the dataset.
# We'll show a few examples of training images from each class.
num_classes = 10
samples_per_class = 7
for cls in range(num_classes):
    idxs = np.argwhere(t_train==cls)
    sample = np.random.choice(idxs.shape[0], samples_per_class, replace=False) # r
    idxs=idxs[sample]

    for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + cls + 1
        plt.subplot(samples_per_class, num_classes, plt_idx)
        img = x_train[idx].reshape(28, 28)

        plt.imshow(img, cmap='gray')
        plt.axis('off')
        if i == 0:
            plt.title(cls)
plt.show()
```



QUESTION 1:What are vanishing gradients? Name one known activation function that has this problem and one that does not.

ANSWER: Vanishing gradients occur when gradients become very small during backpropagation, cuasing to an ineffectively weight updates during training. The sigmoid activation function is prone to this issue due to saturation, where gradients approach zero for large inputs. Conversely, the Rectified Linear Unit (ReLU) activation function avoids saturation for positive inputs, alleviating the vanishing gradient problem.

here we will implement the sigmoid activation function and it's gradient

In [55]:
```
def sigmoid(x):
    #############################################################################
    #                              YOUR CODE                                    #
    #############################################################################
    sig = 1 / (1 + np.exp(-x))
    ############################################################################
    #                              END OF YOUR CODE                            #
    ############################################################################
    return sig

def sigmoid_grad(x):
    #############################################################################
    #                              YOUR CODE                                    #
    #############################################################################
    sig = sigmoid(x)
    sig_grad = sig*(1-sig)
    ############################################################################
    #                              END OF YOUR CODE                            #
    ############################################################################
    return sig_grad
```

Implement a fully-vectorized loss function for the Softmax classifier Make sure the softmax is stable. To make our softmax function numerically stable,we simply normalize the values in the vector, by multiplying the numerator and denominator with a constant C. We can choose an arbitrary value for log(C) term, but generally log(C)=−max(a) is chosen, as it shifts all of elements in the vector to negative to zero, and negatives with large exponents saturate to zero rather than the infinity.

In [56]:

```
def softmax(x):
    """
  Softmax loss function, should be implemented in a vectorized fashion (without lo


  Inputs:
  - X: A numpy array of shape (N, C) containing a minibatch of data.
  Returns:
  - probabilities: A numpy array of shape (N, C) containing the softmax probabilit

  if you are not careful here, it is easy to run into numeric instability
     """
    ###########################################################################
    #                              YOUR CODE                                  #
    ###########################################################################
    # Shift logits by subtracting the maximum value to prevent overflow
    exp_logits = np.exp(x - np.max(x, axis=1, keepdims=True))
    probabilities = exp_logits / np.sum(exp_logits, axis=1, keepdims=True)

    ##########################################################################
    #                              END OF YOUR CODE                          #
    ##########################################################################
    return probabilities


def cross_entropy_error(y, t):
    """
    Inputs:

    - t:  A numpy array of shape (N,C) containing  a minibatch of training labels,
      with t[GT]=1 and t=0 elsewhere, where GT is the ground truth label ;
    - y: A numpy array of shape (N, C) containing the softmax probabilities (the N

    Returns a tuple of:
    - loss as single float (do not forget to divide by the number of samples in th
    """
    ###########################################################################
    #                              YOUR CODE                                  #
    ###########################################################################
    error = -np.sum(t * np.log(y)) / y.shape[0]
    ##########################################################################
    #                              END OF YOUR CODE                          #
    ##########################################################################
    return error
```

We will design and train a two-layer fully-connected neural network with sigmoid nonlinearity and softmax cross entropy loss. We assume an input dimension of D=784, a hidden dimension of H, and perform classification over C classes.

The architecture should be fullyconnected -> sigmoid -> fullyconnected -> softmax.

The learnable parameters of the model are stored in the dictionary, 'params', that maps parameter names to numpy arrays.

In the next cell we will initialize the weights and biases, design the fully connected(fc) forward and backward functions that will be in use for the training (using SGD).

In [57]:

```
def TwoLayerNet( input_size, hidden_size, output_size, weight_init_std=0.01):
    ############################################################################
    # TODO: Initialize the weights and biases of the two-layer net. Weights    #
    # should be initialized from a Gaussian with standard deviation equal to   #
    # weight_init_std, and biases should be initialized to zero. All weights and
    # biases should be stored in the dictionary 'params', with first layer  #
    # weights and biases using the keys 'W1' and 'b1' and second layer weights #
    # and biases using the keys 'W2' and 'b2'.                                 #
    ############################################################################
    params = {}

    # Initialize weights with a Gaussian distribution and biases to zeros for the
    params['W1'] = np.random.normal(0, weight_init_std, size=(input_size, hidden_s
    params['b1'] = np.zeros(hidden_size)
    params['W2'] = np.random.normal(0, weight_init_std, size=(hidden_size, output_
    params['b2'] = np.zeros(output_size)

    ############################################################################
    #                              END OF YOUR CODE                            #
    ############################################################################
    return params


def FC_forward(x, w, b):
    """
    Computes the forward pass for a fully-connected layer.
    The input x has shape (N, D) and contains a minibatch of N
    examples, where each example x[i] has shape D and will be transformed to an ou
    Inputs:
    - x: A numpy array containing input data, of shape (N, D)
    - w: A numpy array of weights, of shape (D, M)
    - b: A numpy array of biases, of shape (M,)

    Returns a tuple of:
    - out: output result of the forward pass, of shape (N, M)
    - cache: (x, w, b)
    """
    out = None
    #############################################################################
    #                              YOUR CODE                                    #
    #############################################################################
    out = x.dot(w) + b
    #############################################################################
    #                              END OF YOUR CODE                             #
    #############################################################################
    cache = (x, w, b)
    return out, cache



def FC_backward(dout, cache):
    """
    Computes the backward pass for a fully-connected layer.
    Inputs:
    - dout: Upstream derivative, of shape (N, M)
    - cache: Tuple of:
```

```
      - x: A numpy array containing input data, of shape (N, D)
      - w: A numpy array of weights, of shape (D, M)
      - b: A numpy array of biases, of shape (M,)
    Returns a tuple of:
    - dx: Gradient with respect to x, of shape (N, D)
    - dw: Gradient with respect to w, of shape (D, M)
    - db: Gradient with respect to b, of shape (M,)
    """
    x, w, b = cache
    dx, dw, db = None, None, None

    ###########################################################################
    #                              YOUR CODE                                  #
    ###########################################################################
    # Compute the gradient of the loss with respect to x
    dx = dout.dot(w.T)

    # Compute the gradient of the loss with respect to w
    dw = x.T.dot(dout)

    # Compute the gradient of the loss with respect to b
    db = np.sum(dout, axis=0)

    ###########################################################################
    #                              END OF YOUR CODE                           #
    ###########################################################################
    return dx, dw, db
```

Here we will design the entire model, which outputs the NN's probabilities and gradients.

In [58]:

```
def Model(params, x, t):
    """
    Computes the backward pass for a fully-connected layer.
    Inputs:
    - params:  dictionary with first layer weights and biases using the keys 'W1'
    and biases using the keys 'W2' and 'b2'. each with dimensions corresponding it
    - x: Input data, of shape (N,D)
    - t:  A numpy array of shape (N,C) containing training labels, it is a one-hot
      with t[GT]=1 and t=0 elsewhere, where GT is the ground truth label ;
    Returns:
    - y: the output probabilities for the minibatch (at the end of the forward pas
    - grads: dictionary containing gradients of the loss with respect to W1, W2, b

    note: use the FC_forward ,FC_backward functions.

    """
    W1, W2 = params['W1'], params['W2']
    b1, b2 = params['b1'], params['b2']
    grads = {'W1': None, 'W2': None, 'b1': None, 'b2': None}

    # Forward pass
    a1, cache1 = FC_forward(x, W1, b1)  # FC layer
    z1 = sigmoid(a1)  # Sigmoid activation
    a2, cache2 = FC_forward(z1, W2, b2)  # FC layer
    y = softmax(a2)  # Softmax layer

    # Compute loss (assuming cross_entropy_error is defined elsewhere)
    #loss = cross_entropy_error(y, t)

    # Backward pass
    # Derivative of cross entropy error with softmax
    # dy = (y - t) / x.shape[0]  # dL/dy
```

```
    dcost_dy=y-t
    dy_dz=np.multiply(y,np.ones(np.shape(y))-y)
    dcost_dz=np.multiply(dcost_dy,dy_dz)

    # Second layer gradients
    dz1, dW2, db2 = FC_backward(dcost_dz, cache2)
    grads['W2'], grads['b2'] = dW2, db2

    # Derivative through sigmoid
    da1 = sigmoid_grad(a1) * dz1

    # First layer gradients
    dx, dW1, db1 = FC_backward(da1, cache1)
    grads['W1'], grads['b1'] = dW1, db1


    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################

    return grads, y
```

Compute the accuracy of the NNs predictions.

In [59]:

```
def accuracy(y,t):
    """
    Computes the accuracy of the NN's predictions.
    Inputs:
    - t:  A numpy array of shape (N,C) containing training labels, it is a one-hot
      with t[GT]=1 and t=0 elsewhere, where GT is the ground truth label ;
    - y: the output probabilities for the minibatch (at the end of the forward pas
    Returns:
    - accuracy: a single float of the average accuracy.
    """
    ###########################################################################
    #                             YOUR CODE                                   #
    ###########################################################################
    # Find the index of the maximum score in the predictions
    y_pred = np.argmax(y, axis=1)
    # Similarly, for the true labels
    t_actual = np.argmax(t, axis=1)

    # Calculate the number of correctly predicted examples
    correct_predictions = np.sum(y_pred == t_actual)

    # Calculate the accuracy
    accuracy = correct_predictions / y.shape[0]

    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################
    return accuracy
```

Trianing the model: To train our network we will use minibatch SGD.

*Note that the test dataset is actually used as the validation dataset in the training

In [60]:

```
# You should be able to receive at least 97% accuracy, choose hyperparameters acco
epochs = 300
mini_batch_size = 800
learning_rate = 0.002
num_hidden_cells = 300
```

```
def Train(epochs_num, batch_size, lr, H):
    #  Dividing a dataset into training data and test data

    (x_train, t_train), (x_test, t_test) = load_mnist(normalize=True, one_hot_labe
    C=10
    D=x_train.shape[1]
    network_params = TwoLayerNet(input_size=D, hidden_size=H, output_size=C) #hida

    train_size = x_train.shape[0]
    train_loss_list = []
    train_acc_list = []
    test_acc_list = []
    iter_per_epoch = round(train_size / batch_size)

    print('training of ' + str(epochs_num) +' epochs, each epoch will have '+ str(
    for i in range(epochs_num):

        train_loss_iter= []
        train_acc_iter= []

        for k in range(iter_per_epoch):


            ######################################################################
            #                              YOUR CODE
            ######################################################################
            # 1. Select part of training data (mini-batch) randomly
            indices = np.random.choice(x_train.shape[0], mini_batch_size, replace=
            x_batch = x_train[indices]  # Use the selected indices to get the mini
            t_batch = t_train[indices]
            # 2.1 Calculate the predictions and the gradients to reduce the value
            grads, y_batch = Model(network_params,x_batch,t_batch)

            # 3. Update weights and biases with the gradients
            network_params['W1'] -= lr * grads['W1']
            network_params['b1'] -= lr * grads['b1']
            network_params['W2'] -= lr * grads['W2']
            network_params['b2'] -= lr * grads['b2']

            ######################################################################
            #                              END OF YOUR CODE
            ######################################################################

            # Calculate the loss and accuracy for visalizaton
            error=cross_entropy_error(y_batch, t_batch)
            train_loss_iter.append(error)
            acc_iter=accuracy(y_batch, t_batch)
            train_acc_iter.append(acc_iter)
            if k == iter_per_epoch-1:
                train_acc = np.mean(train_acc_iter)
                train_acc_list.append(train_acc)
                train_loss_list.append(np.mean(train_loss_iter))

                _, y_test = Model(network_params, x_test, t_test)
                test_acc = accuracy(y_test, t_test)
                test_acc_list.append(test_acc)
                print("train acc: " + str(train_acc)[:5] + "% |  test acc: "   + s
    return train_acc_list, test_acc_list, train_loss_list, network_params

train_acc, test_acc, train_loss, net_params = Train(epochs, mini_batch_size, learn

markers = {'train': 'o', 'test': 's'}
x = np.arange(len(train_acc))
```

```
plt.plot(x, train_acc, label='train acc')
plt.plot(x, test_acc, label='test acc', linestyle='--')
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.legend(loc='lower right')
plt.show()


markers = {'train': 'o'}
x = np.arange(len(train_loss))
plt.plot(x, train_loss, label='train loss')
plt.xlabel("epochs")
plt.ylabel("Loss")
plt.legend(loc='lower right')
plt.show()
```

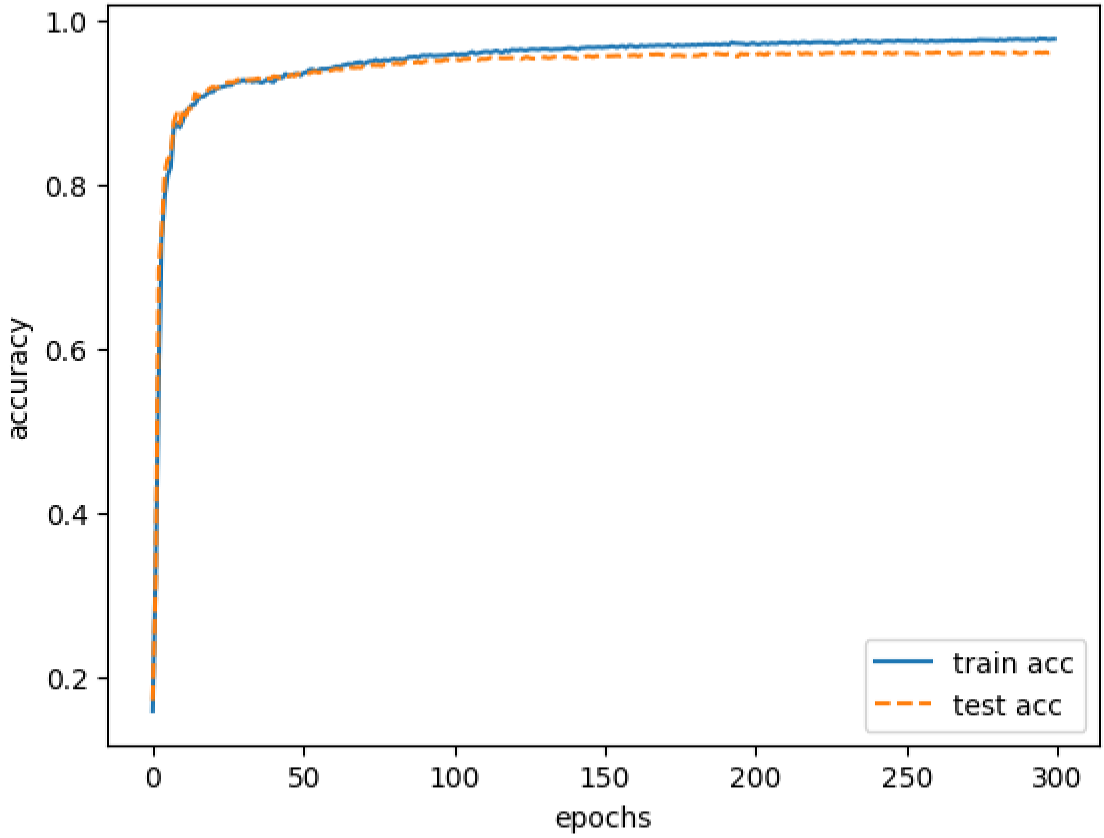
```
training of 300 epochs, each epoch will have 75 iterations
train acc: 0.158% | test acc: 0.1709% | loss for epoch 0: 2.2705668585847936
train acc: 0.328% | test acc: 0.3153% | loss for epoch 1: 1.9891961749203135
train acc: 0.567% | test acc: 0.6963% | loss for epoch 2: 1.7713238883406044
train acc: 0.732% | test acc: 0.7425% | loss for epoch 3: 1.4772616338842721
train acc: 0.786% | test acc: 0.8142% | loss for epoch 4: 1.2494359851500536
train acc: 0.811% | test acc: 0.8303% | loss for epoch 5: 1.0846450941600851
train acc: 0.820% | test acc: 0.8337% | loss for epoch 6: 0.8936970308031413
train acc: 0.867% | test acc: 0.8758% | loss for epoch 7: 0.5204705554171241
train acc: 0.873% | test acc: 0.8858% | loss for epoch 8: 0.4413892377097871
train acc: 0.868% | test acc: 0.8736% | loss for epoch 9: 0.44674516140105985
train acc: 0.877% | test acc: 0.8917% | loss for epoch 10: 0.42933617973639676
train acc: 0.888% | test acc: 0.8831% | loss for epoch 11: 0.39895076647555816
train acc: 0.891% | test acc: 0.8922% | loss for epoch 12: 0.3921306608480927
train acc: 0.896% | test acc: 0.8928% | loss for epoch 13: 0.37945729020543467
train acc: 0.896% | test acc: 0.9099% | loss for epoch 14: 0.376604707842955
train acc: 0.903% | test acc: 0.9061% | loss for epoch 15: 0.35749055789388784
train acc: 0.906% | test acc: 0.9104% | loss for epoch 16: 0.3564237217785557
train acc: 0.906% | test acc: 0.9127% | loss for epoch 17: 0.34996405218152926
train acc: 0.909% | test acc: 0.9134% | loss for epoch 18: 0.3466395091191414
train acc: 0.912% | test acc: 0.9164% | loss for epoch 19: 0.33140989006325905
train acc: 0.912% | test acc: 0.9184% | loss for epoch 20: 0.3371026842068618
train acc: 0.915% | test acc: 0.9168% | loss for epoch 21: 0.3228886555261457
train acc: 0.917% | test acc: 0.9181% | loss for epoch 22: 0.3151529115019276
train acc: 0.918% | test acc: 0.9202% | loss for epoch 23: 0.3179036725217456
train acc: 0.919% | test acc: 0.9223% | loss for epoch 24: 0.3094844388179486
train acc: 0.920% | test acc: 0.9239% | loss for epoch 25: 0.31134817091324635
train acc: 0.920% | test acc: 0.9239% | loss for epoch 26: 0.3100720087320746
train acc: 0.923% | test acc: 0.9239% | loss for epoch 27: 0.3063162498307051
train acc: 0.924% | test acc: 0.9266% | loss for epoch 28: 0.29610692001851385
train acc: 0.924% | test acc: 0.9271% | loss for epoch 29: 0.29755322275553947
train acc: 0.926% | test acc: 0.9248% | loss for epoch 30: 0.2911205938741047
train acc: 0.924% | test acc: 0.9266% | loss for epoch 31: 0.3012258382833913
train acc: 0.926% | test acc: 0.9266% | loss for epoch 32: 0.2961631277021612
train acc: 0.924% | test acc: 0.9272% | loss for epoch 33: 0.3049994842090867
train acc: 0.925% | test acc: 0.9275% | loss for epoch 34: 0.3042777284906642
train acc: 0.925% | test acc: 0.9286% | loss for epoch 35: 0.309385010747687
train acc: 0.923% | test acc: 0.9263% | loss for epoch 36: 0.3089223397189576
train acc: 0.924% | test acc: 0.9281% | loss for epoch 37: 0.3055811094801001
train acc: 0.927% | test acc: 0.9285% | loss for epoch 38: 0.2977109639122226
train acc: 0.925% | test acc: 0.9282% | loss for epoch 39: 0.3059867836709508
train acc: 0.923% | test acc: 0.931% | loss for epoch 40: 0.3068462016957478
train acc: 0.927% | test acc: 0.9306% | loss for epoch 41: 0.30719862135524684
train acc: 0.930% | test acc: 0.9294% | loss for epoch 42: 0.2945348320116454
train acc: 0.929% | test acc: 0.9294% | loss for epoch 43: 0.29764479382465747
train acc: 0.933% | test acc: 0.9321% | loss for epoch 44: 0.28697654826496477
train acc: 0.931% | test acc: 0.9307% | loss for epoch 45: 0.29361921261238044
train acc: 0.932% | test acc: 0.933% | loss for epoch 46: 0.2889196553096936
train acc: 0.933% | test acc: 0.9311% | loss for epoch 47: 0.28563950433750007
train acc: 0.934% | test acc: 0.9337% | loss for epoch 48: 0.28516366937042226
train acc: 0.931% | test acc: 0.9341% | loss for epoch 49: 0.2856549418027272
train acc: 0.935% | test acc: 0.9341% | loss for epoch 50: 0.2830973332913748
train acc: 0.936% | test acc: 0.9351% | loss for epoch 51: 0.2754965506219703
train acc: 0.939% | test acc: 0.9347% | loss for epoch 52: 0.26988604906855235
train acc: 0.939% | test acc: 0.9364% | loss for epoch 53: 0.26471459712896367
train acc: 0.937% | test acc: 0.937% | loss for epoch 54: 0.27599257101643604
train acc: 0.939% | test acc: 0.9354% | loss for epoch 55: 0.26497206697626524
train acc: 0.940% | test acc: 0.9364% | loss for epoch 56: 0.26562499354857444
train acc: 0.940% | test acc: 0.9364% | loss for epoch 57: 0.26017643016086206
train acc: 0.940% | test acc: 0.9374% | loss for epoch 58: 0.2596325919877017
train acc: 0.940% | test acc: 0.9381% | loss for epoch 59: 0.25196230009964193
train acc: 0.942% | test acc: 0.9392% | loss for epoch 60: 0.255131146536072
train acc: 0.943% | test acc: 0.9401% | loss for epoch 61: 0.2489403261855439
train acc: 0.942% | test acc: 0.9391% | loss for epoch 62: 0.2514358447672347
```

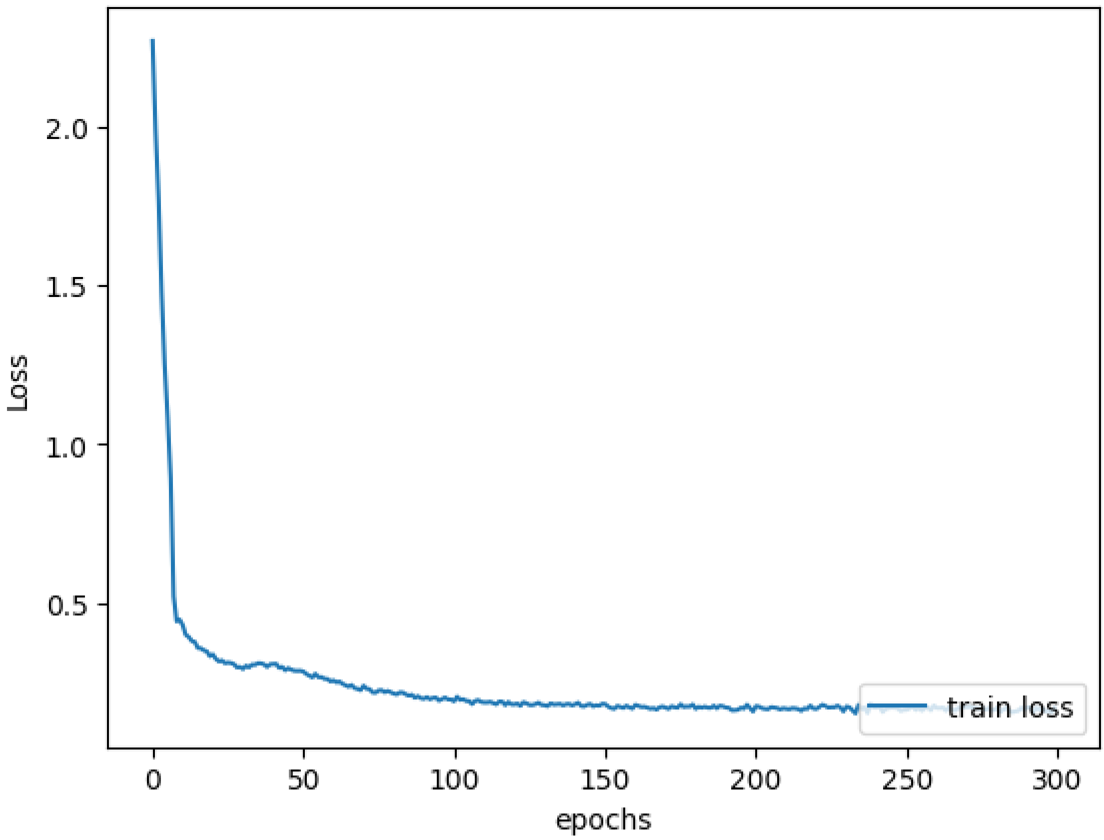
```
train acc: 0.944% | test acc: 0.9396% | loss for epoch 63: 0.2444135991258248
train acc: 0.945% | test acc: 0.9413% | loss for epoch 64: 0.2402487014172228
train acc: 0.946% | test acc: 0.9402% | loss for epoch 65: 0.23543908343857023
train acc: 0.945% | test acc: 0.942% | loss for epoch 66: 0.24062602806963318
train acc: 0.947% | test acc: 0.9414% | loss for epoch 67: 0.23078372469610978
train acc: 0.947% | test acc: 0.9421% | loss for epoch 68: 0.22846677604859622
train acc: 0.948% | test acc: 0.9415% | loss for epoch 69: 0.2243067288096672
train acc: 0.945% | test acc: 0.9438% | loss for epoch 70: 0.23820834217101813
train acc: 0.948% | test acc: 0.9435% | loss for epoch 71: 0.22878435321599777
train acc: 0.949% | test acc: 0.9434% | loss for epoch 72: 0.2275005020850598
train acc: 0.949% | test acc: 0.9449% | loss for epoch 73: 0.21708790062978736
train acc: 0.951% | test acc: 0.9444% | loss for epoch 74: 0.21660780096332266
train acc: 0.949% | test acc: 0.9447% | loss for epoch 75: 0.22346673466187453
train acc: 0.950% | test acc: 0.9438% | loss for epoch 76: 0.2241758510856223
train acc: 0.951% | test acc: 0.9458% | loss for epoch 77: 0.21783300396060815
train acc: 0.949% | test acc: 0.9462% | loss for epoch 78: 0.22205096005529312
train acc: 0.951% | test acc: 0.9452% | loss for epoch 79: 0.2174535200283466
train acc: 0.952% | test acc: 0.9464% | loss for epoch 80: 0.21358018932977363
train acc: 0.951% | test acc: 0.9471% | loss for epoch 81: 0.2109450271412412
train acc: 0.952% | test acc: 0.946% | loss for epoch 82: 0.21711721697384467
train acc: 0.951% | test acc: 0.9468% | loss for epoch 83: 0.21669652449619614
train acc: 0.952% | test acc: 0.9485% | loss for epoch 84: 0.21118925816043022
train acc: 0.954% | test acc: 0.9475% | loss for epoch 85: 0.20690761880834968
train acc: 0.954% | test acc: 0.9481% | loss for epoch 86: 0.2097015116187357
train acc: 0.955% | test acc: 0.9477% | loss for epoch 87: 0.20025386803137513
train acc: 0.953% | test acc: 0.9487% | loss for epoch 88: 0.20529472405528668
train acc: 0.956% | test acc: 0.9473% | loss for epoch 89: 0.19909207334776619
train acc: 0.956% | test acc: 0.9491% | loss for epoch 90: 0.19711150715073777
train acc: 0.955% | test acc: 0.9497% | loss for epoch 91: 0.20243824443924696
train acc: 0.957% | test acc: 0.95% | loss for epoch 92: 0.19580579626417
train acc: 0.955% | test acc: 0.9494% | loss for epoch 93: 0.19988175098391855
train acc: 0.956% | test acc: 0.9497% | loss for epoch 94: 0.20092806657704018
train acc: 0.956% | test acc: 0.9496% | loss for epoch 95: 0.19140062018456372
train acc: 0.956% | test acc: 0.9499% | loss for epoch 96: 0.19622585350054314
train acc: 0.956% | test acc: 0.9516% | loss for epoch 97: 0.20094670065011552
train acc: 0.957% | test acc: 0.9509% | loss for epoch 98: 0.19514575026985923
train acc: 0.957% | test acc: 0.9499% | loss for epoch 99: 0.19493642258927263
train acc: 0.958% | test acc: 0.951% | loss for epoch 100: 0.18930695721282023
train acc: 0.956% | test acc: 0.9513% | loss for epoch 101: 0.2038737014620677
train acc: 0.957% | test acc: 0.9505% | loss for epoch 102: 0.19364823403333217
train acc: 0.957% | test acc: 0.9522% | loss for epoch 103: 0.19615686567929405
train acc: 0.957% | test acc: 0.952% | loss for epoch 104: 0.1904725170711718
train acc: 0.959% | test acc: 0.9525% | loss for epoch 105: 0.18995234293594815
train acc: 0.960% | test acc: 0.9525% | loss for epoch 106: 0.1803816086779587
train acc: 0.958% | test acc: 0.9527% | loss for epoch 107: 0.19025850456917845
train acc: 0.959% | test acc: 0.9508% | loss for epoch 108: 0.19299055979933166
train acc: 0.960% | test acc: 0.9523% | loss for epoch 109: 0.1872228161628974
train acc: 0.961% | test acc: 0.952% | loss for epoch 110: 0.18562256594856025
train acc: 0.961% | test acc: 0.9523% | loss for epoch 111: 0.18568295087977524
train acc: 0.960% | test acc: 0.9539% | loss for epoch 112: 0.18751291772249257
train acc: 0.961% | test acc: 0.9529% | loss for epoch 113: 0.18374983886636256
train acc: 0.962% | test acc: 0.9538% | loss for epoch 114: 0.17976795542483723
train acc: 0.960% | test acc: 0.9534% | loss for epoch 115: 0.1884834813622054
train acc: 0.959% | test acc: 0.9531% | loss for epoch 116: 0.187880700010588
train acc: 0.962% | test acc: 0.9531% | loss for epoch 117: 0.17714902848286188
train acc: 0.960% | test acc: 0.9533% | loss for epoch 118: 0.18697123995443476
train acc: 0.963% | test acc: 0.9535% | loss for epoch 119: 0.17980120751922898
train acc: 0.963% | test acc: 0.9534% | loss for epoch 120: 0.1793710468670053
train acc: 0.962% | test acc: 0.9531% | loss for epoch 121: 0.1821036143273581
train acc: 0.963% | test acc: 0.952% | loss for epoch 122: 0.17481510111338597
train acc: 0.962% | test acc: 0.9532% | loss for epoch 123: 0.18603911204624146
train acc: 0.962% | test acc: 0.9548% | loss for epoch 124: 0.1830296048852687
train acc: 0.963% | test acc: 0.9533% | loss for epoch 125: 0.1776428544752252
train acc: 0.964% | test acc: 0.9527% | loss for epoch 126: 0.1773896756489613
```

```
train acc: 0.963% |  test acc: 0.9527% |  loss for epoch 127: 0.18023044825885357
train acc: 0.963% |  test acc: 0.9534% |  loss for epoch 128: 0.18469176400616785
train acc: 0.964% |  test acc: 0.9532% |  loss for epoch 129: 0.1778403720766143
train acc: 0.964% |  test acc: 0.9547% |  loss for epoch 130: 0.17706872925309233
train acc: 0.964% |  test acc: 0.9533% |  loss for epoch 131: 0.17395730988280156
train acc: 0.963% |  test acc: 0.9536% |  loss for epoch 132: 0.1823971822754196
train acc: 0.964% |  test acc: 0.9547% |  loss for epoch 133: 0.17884471936503785
train acc: 0.964% |  test acc: 0.9545% |  loss for epoch 134: 0.17988045412157685
train acc: 0.964% |  test acc: 0.9544% |  loss for epoch 135: 0.18103621313236867
train acc: 0.964% |  test acc: 0.9556% |  loss for epoch 136: 0.17625450801115514
train acc: 0.964% |  test acc: 0.9545% |  loss for epoch 137: 0.18005486482263847
train acc: 0.965% |  test acc: 0.9551% |  loss for epoch 138: 0.18079746427026314
train acc: 0.965% |  test acc: 0.9536% |  loss for epoch 139: 0.17607815170971905
train acc: 0.965% |  test acc: 0.9532% |  loss for epoch 140: 0.17983743579147599
train acc: 0.963% |  test acc: 0.9536% |  loss for epoch 141: 0.18465897313387095
train acc: 0.965% |  test acc: 0.9553% |  loss for epoch 142: 0.1743157410164906
train acc: 0.965% |  test acc: 0.9546% |  loss for epoch 143: 0.1733371392546666
train acc: 0.966% |  test acc: 0.9557% |  loss for epoch 144: 0.17822219123331598
train acc: 0.966% |  test acc: 0.9556% |  loss for epoch 145: 0.18074124962344007
train acc: 0.966% |  test acc: 0.9544% |  loss for epoch 146: 0.17330879674553168
train acc: 0.966% |  test acc: 0.9558% |  loss for epoch 147: 0.17687058867353744
train acc: 0.966% |  test acc: 0.9556% |  loss for epoch 148: 0.17490397170472852
train acc: 0.966% |  test acc: 0.9562% |  loss for epoch 149: 0.18124992424920153
train acc: 0.966% |  test acc: 0.9566% |  loss for epoch 150: 0.1802891790652622
train acc: 0.966% |  test acc: 0.9555% |  loss for epoch 151: 0.17227184405276005
train acc: 0.967% |  test acc: 0.9558% |  loss for epoch 152: 0.1676818819216811
train acc: 0.967% |  test acc: 0.9558% |  loss for epoch 153: 0.16571054400058574
train acc: 0.966% |  test acc: 0.9562% |  loss for epoch 154: 0.17447558869865104
train acc: 0.967% |  test acc: 0.9558% |  loss for epoch 155: 0.17168971791341375
train acc: 0.968% |  test acc: 0.9566% |  loss for epoch 156: 0.16829170704821983
train acc: 0.966% |  test acc: 0.9559% |  loss for epoch 157: 0.17511348153082223
train acc: 0.968% |  test acc: 0.9562% |  loss for epoch 158: 0.17000033539103282
train acc: 0.968% |  test acc: 0.9566% |  loss for epoch 159: 0.16637529403259949
train acc: 0.967% |  test acc: 0.9568% |  loss for epoch 160: 0.17799836882025652
train acc: 0.966% |  test acc: 0.9554% |  loss for epoch 161: 0.17434554499198238
train acc: 0.968% |  test acc: 0.9567% |  loss for epoch 162: 0.17109807180628583
train acc: 0.967% |  test acc: 0.9575% |  loss for epoch 163: 0.17198540384630218
train acc: 0.967% |  test acc: 0.9572% |  loss for epoch 164: 0.16880631393404685
train acc: 0.968% |  test acc: 0.9577% |  loss for epoch 165: 0.16567821494564242
train acc: 0.968% |  test acc: 0.9574% |  loss for epoch 166: 0.16769788609905636
train acc: 0.968% |  test acc: 0.9579% |  loss for epoch 167: 0.16309995258779933
train acc: 0.968% |  test acc: 0.9578% |  loss for epoch 168: 0.1702029817600145
train acc: 0.968% |  test acc: 0.9572% |  loss for epoch 169: 0.17271692673072384
train acc: 0.968% |  test acc: 0.9575% |  loss for epoch 170: 0.16940950077561798
train acc: 0.969% |  test acc: 0.9572% |  loss for epoch 171: 0.1656242807169581
train acc: 0.968% |  test acc: 0.9559% |  loss for epoch 172: 0.17338542626002154
train acc: 0.967% |  test acc: 0.9565% |  loss for epoch 173: 0.17038230860839923
train acc: 0.969% |  test acc: 0.9565% |  loss for epoch 174: 0.16892860429713785
train acc: 0.968% |  test acc: 0.9566% |  loss for epoch 175: 0.17986715368279407
train acc: 0.968% |  test acc: 0.9557% |  loss for epoch 176: 0.17120093572440515
train acc: 0.969% |  test acc: 0.9562% |  loss for epoch 177: 0.17294757239449401
train acc: 0.968% |  test acc: 0.9576% |  loss for epoch 178: 0.1723137770177915
train acc: 0.969% |  test acc: 0.9562% |  loss for epoch 179: 0.1791568797585164
train acc: 0.968% |  test acc: 0.956% |  loss for epoch 180: 0.16572204116143466
train acc: 0.969% |  test acc: 0.9561% |  loss for epoch 181: 0.17187159512255376
train acc: 0.969% |  test acc: 0.9576% |  loss for epoch 182: 0.16885207923468293
train acc: 0.970% |  test acc: 0.9576% |  loss for epoch 183: 0.17231524953188287
train acc: 0.969% |  test acc: 0.9565% |  loss for epoch 184: 0.1689124239460437
train acc: 0.970% |  test acc: 0.9568% |  loss for epoch 185: 0.16863321593798156
train acc: 0.969% |  test acc: 0.9573% |  loss for epoch 186: 0.17513146762568432
train acc: 0.970% |  test acc: 0.9588% |  loss for epoch 187: 0.16705574778014284
train acc: 0.969% |  test acc: 0.9581% |  loss for epoch 188: 0.1752138397759968
train acc: 0.970% |  test acc: 0.9577% |  loss for epoch 189: 0.17333315779375888
train acc: 0.970% |  test acc: 0.9573% |  loss for epoch 190: 0.16950239242563844
```

```
train acc: 0.969% |  test acc: 0.9585% |  loss for epoch 191: 0.16862722145542064
train acc: 0.972% |  test acc: 0.9584% |  loss for epoch 192: 0.16078958126354592
train acc: 0.971% |  test acc: 0.9562% |  loss for epoch 193: 0.16122061992016123
train acc: 0.971% |  test acc: 0.9552% |  loss for epoch 194: 0.16244771756001763
train acc: 0.970% |  test acc: 0.9586% |  loss for epoch 195: 0.1691850225502211
train acc: 0.970% |  test acc: 0.9577% |  loss for epoch 196: 0.16622615574433725
train acc: 0.969% |  test acc: 0.9581% |  loss for epoch 197: 0.17855275308421162
train acc: 0.970% |  test acc: 0.9572% |  loss for epoch 198: 0.16755031380745075
train acc: 0.971% |  test acc: 0.9574% |  loss for epoch 199: 0.15724423139722232
train acc: 0.970% |  test acc: 0.9569% |  loss for epoch 200: 0.17396770469877107
train acc: 0.969% |  test acc: 0.9586% |  loss for epoch 201: 0.17233428437065956
train acc: 0.971% |  test acc: 0.9581% |  loss for epoch 202: 0.16738029159785717
train acc: 0.971% |  test acc: 0.957% |  loss for epoch 203: 0.16530617949553092
train acc: 0.970% |  test acc: 0.9592% |  loss for epoch 204: 0.1643796662501388
train acc: 0.971% |  test acc: 0.9563% |  loss for epoch 205: 0.1708880716282895
train acc: 0.971% |  test acc: 0.958% |  loss for epoch 206: 0.1696329059745453
train acc: 0.970% |  test acc: 0.9584% |  loss for epoch 207: 0.16782867704265755
train acc: 0.972% |  test acc: 0.9585% |  loss for epoch 208: 0.1635222848736463
train acc: 0.972% |  test acc: 0.9589% |  loss for epoch 209: 0.16730724349766396
train acc: 0.972% |  test acc: 0.9578% |  loss for epoch 210: 0.16574717559387087
train acc: 0.971% |  test acc: 0.9581% |  loss for epoch 211: 0.1641127934983506
train acc: 0.972% |  test acc: 0.9585% |  loss for epoch 212: 0.1674650516991181
train acc: 0.972% |  test acc: 0.959% |  loss for epoch 213: 0.16676944823560597
train acc: 0.971% |  test acc: 0.9597% |  loss for epoch 214: 0.16464452918901532
train acc: 0.972% |  test acc: 0.958% |  loss for epoch 215: 0.1593247163176499
train acc: 0.972% |  test acc: 0.9586% |  loss for epoch 216: 0.1669599464128394
train acc: 0.972% |  test acc: 0.9592% |  loss for epoch 217: 0.16532054381824918
train acc: 0.971% |  test acc: 0.9582% |  loss for epoch 218: 0.17547071168084755
train acc: 0.972% |  test acc: 0.9588% |  loss for epoch 219: 0.1667823170052421
train acc: 0.972% |  test acc: 0.9577% |  loss for epoch 220: 0.1652874795282412
train acc: 0.972% |  test acc: 0.9591% |  loss for epoch 221: 0.1712174660883927
train acc: 0.972% |  test acc: 0.958% |  loss for epoch 222: 0.17829519653957018
train acc: 0.972% |  test acc: 0.9592% |  loss for epoch 223: 0.17452037402661713
train acc: 0.972% |  test acc: 0.9592% |  loss for epoch 224: 0.16922296181745078
train acc: 0.972% |  test acc: 0.9593% |  loss for epoch 225: 0.17068384205685344
train acc: 0.971% |  test acc: 0.9598% |  loss for epoch 226: 0.17023071676335558
train acc: 0.971% |  test acc: 0.9586% |  loss for epoch 227: 0.17522020556189247
train acc: 0.972% |  test acc: 0.9599% |  loss for epoch 228: 0.16976822176569853
train acc: 0.973% |  test acc: 0.9592% |  loss for epoch 229: 0.15758238811036124
train acc: 0.972% |  test acc: 0.9592% |  loss for epoch 230: 0.17280114353503348
train acc: 0.972% |  test acc: 0.9601% |  loss for epoch 231: 0.17022665888988173
train acc: 0.973% |  test acc: 0.9591% |  loss for epoch 232: 0.1607720572383218
train acc: 0.974% |  test acc: 0.9598% |  loss for epoch 233: 0.1512273986275168
train acc: 0.972% |  test acc: 0.9598% |  loss for epoch 234: 0.1778108050273728
train acc: 0.973% |  test acc: 0.9595% |  loss for epoch 235: 0.16456510548502815
train acc: 0.972% |  test acc: 0.9595% |  loss for epoch 236: 0.1694859217601174
train acc: 0.974% |  test acc: 0.9598% |  loss for epoch 237: 0.15205834945125324
train acc: 0.973% |  test acc: 0.9597% |  loss for epoch 238: 0.18021264726572123
train acc: 0.972% |  test acc: 0.9589% |  loss for epoch 239: 0.1695408845853687
train acc: 0.973% |  test acc: 0.9596% |  loss for epoch 240: 0.16932920559823936
train acc: 0.972% |  test acc: 0.9593% |  loss for epoch 241: 0.1733934490100041
train acc: 0.973% |  test acc: 0.9609% |  loss for epoch 242: 0.15533379516193266
train acc: 0.974% |  test acc: 0.9587% |  loss for epoch 243: 0.16497891196405323
train acc: 0.974% |  test acc: 0.9599% |  loss for epoch 244: 0.16790023702892704
train acc: 0.974% |  test acc: 0.9583% |  loss for epoch 245: 0.1696722888618837
train acc: 0.973% |  test acc: 0.9588% |  loss for epoch 246: 0.16890433854448764
train acc: 0.974% |  test acc: 0.9597% |  loss for epoch 247: 0.16739712382321242
train acc: 0.974% |  test acc: 0.96% |  loss for epoch 248: 0.1585738675107031
train acc: 0.974% |  test acc: 0.9585% |  loss for epoch 249: 0.16216016890859355
train acc: 0.974% |  test acc: 0.9594% |  loss for epoch 250: 0.16227001581103417
train acc: 0.974% |  test acc: 0.9593% |  loss for epoch 251: 0.1673314823450963
train acc: 0.973% |  test acc: 0.9591% |  loss for epoch 252: 0.1738645083674373
train acc: 0.975% |  test acc: 0.9583% |  loss for epoch 253: 0.1616618230315772
train acc: 0.973% |  test acc: 0.9585% |  loss for epoch 254: 0.1699962178567163
```

```
train acc: 0.974% |  test acc: 0.9608% |  loss for epoch 255: 0.15959142426900869
train acc: 0.973% |  test acc: 0.9593% |  loss for epoch 256: 0.16794405273220508
train acc: 0.973% |  test acc: 0.9598% |  loss for epoch 257: 0.1700216097037573
train acc: 0.974% |  test acc: 0.9594% |  loss for epoch 258: 0.16058839605719213
train acc: 0.974% |  test acc: 0.9595% |  loss for epoch 259: 0.1760140038040019
train acc: 0.973% |  test acc: 0.9603% |  loss for epoch 260: 0.16784454231068677
train acc: 0.974% |  test acc: 0.9597% |  loss for epoch 261: 0.16776284535322092
train acc: 0.973% |  test acc: 0.9588% |  loss for epoch 262: 0.16932562310415555
train acc: 0.974% |  test acc: 0.9591% |  loss for epoch 263: 0.1594309562083818
train acc: 0.974% |  test acc: 0.96% |  loss for epoch 264: 0.1708106266422423
train acc: 0.975% |  test acc: 0.9582% |  loss for epoch 265: 0.1636758940356823
train acc: 0.974% |  test acc: 0.9586% |  loss for epoch 266: 0.1596420002181337
train acc: 0.974% |  test acc: 0.9593% |  loss for epoch 267: 0.1679314478175654
train acc: 0.974% |  test acc: 0.9593% |  loss for epoch 268: 0.16697464201284268
train acc: 0.974% |  test acc: 0.9601% |  loss for epoch 269: 0.17299822225657582
train acc: 0.974% |  test acc: 0.9594% |  loss for epoch 270: 0.17058278170255906
train acc: 0.974% |  test acc: 0.9589% |  loss for epoch 271: 0.17322973418991
train acc: 0.974% |  test acc: 0.9594% |  loss for epoch 272: 0.16762141766228775
train acc: 0.974% |  test acc: 0.959% |  loss for epoch 273: 0.1683164484605891
train acc: 0.975% |  test acc: 0.959% |  loss for epoch 274: 0.16559628130288537
train acc: 0.975% |  test acc: 0.96% |  loss for epoch 275: 0.16919909685020126
train acc: 0.975% |  test acc: 0.9595% |  loss for epoch 276: 0.16283355492790605
train acc: 0.975% |  test acc: 0.9589% |  loss for epoch 277: 0.1674136129281224
train acc: 0.975% |  test acc: 0.9598% |  loss for epoch 278: 0.16042322430045505
train acc: 0.975% |  test acc: 0.9594% |  loss for epoch 279: 0.16200362583862232
train acc: 0.974% |  test acc: 0.96% |  loss for epoch 280: 0.16927701020188377
train acc: 0.975% |  test acc: 0.9592% |  loss for epoch 281: 0.1551252749510957
train acc: 0.975% |  test acc: 0.9589% |  loss for epoch 282: 0.1635745119765746
train acc: 0.975% |  test acc: 0.9589% |  loss for epoch 283: 0.1682189485179382
train acc: 0.976% |  test acc: 0.9598% |  loss for epoch 284: 0.15981501368755094
train acc: 0.975% |  test acc: 0.9603% |  loss for epoch 285: 0.15416080319992506
train acc: 0.976% |  test acc: 0.9603% |  loss for epoch 286: 0.15789359638387
train acc: 0.976% |  test acc: 0.9592% |  loss for epoch 287: 0.15908690566300976
train acc: 0.975% |  test acc: 0.9586% |  loss for epoch 288: 0.16852313199738084
train acc: 0.975% |  test acc: 0.9596% |  loss for epoch 289: 0.1631860615056949
train acc: 0.975% |  test acc: 0.9599% |  loss for epoch 290: 0.17044381861757898
train acc: 0.975% |  test acc: 0.9595% |  loss for epoch 291: 0.16918534215227102
train acc: 0.976% |  test acc: 0.9591% |  loss for epoch 292: 0.1553984180075724
train acc: 0.974% |  test acc: 0.9598% |  loss for epoch 293: 0.16864940942755613
train acc: 0.977% |  test acc: 0.9602% |  loss for epoch 294: 0.1630652089056074
train acc: 0.976% |  test acc: 0.9596% |  loss for epoch 295: 0.16757331552543273
train acc: 0.976% |  test acc: 0.9603% |  loss for epoch 296: 0.15727696207871514
train acc: 0.975% |  test acc: 0.9596% |  loss for epoch 297: 0.16010325530148845
train acc: 0.976% |  test acc: 0.9601% |  loss for epoch 298: 0.1538038088459029
train acc: 0.976% |  test acc: 0.9594% |  loss for epoch 299: 0.16798405377357772
```





You should be able to receive at least 97% accuracy, choose hyperparameters accordingly.

QUESTION 2: Explain the results looking at the visualizations above, base your answer on the hyperparameters.

ANSWER:

The accuracy and loss curves suggest that the model with the given hyperparameters is learning effectively and generalizing well to unseen data. The accuracy for both training and test sets quickly reaches a high level and plateaus, indicating no overfitting and that the model has likely reached its performance capacity with the current settings. The learning rate of 0.002 is appropriate, as evidenced by the smooth convergence without oscillations. The mini-batch size of 800 provides a good balance between gradient estimation and computational efficiency. With 300 hidden cells, the model has enough capacity to learn the task to a high accuracy, as no divergence between training and test accuracy is observed. Overall, the chosen hyperparameters appear suitable for this classification task.

QUESTION 3: Suggest a way to improve the results by changing the networks's architecture

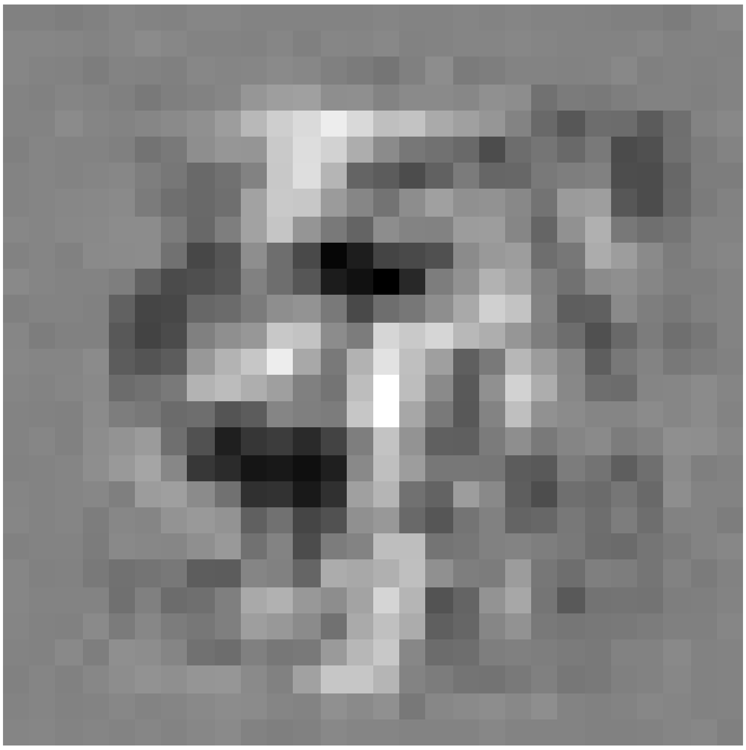
ANSWER: There are several ways for improving the results and the training process, the main ways are:

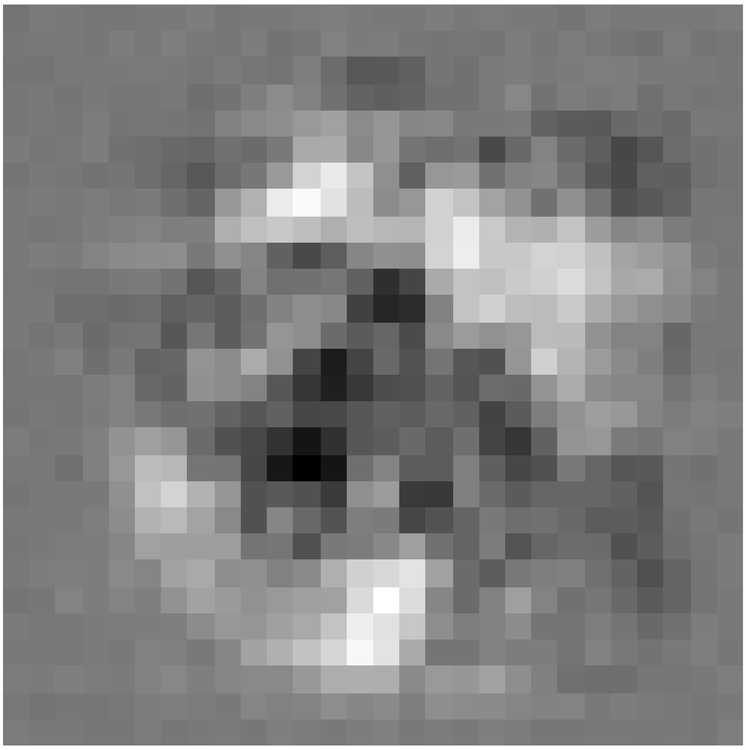
1. Using regularizations such as batch normalizations and dropout
2. Adding more layers making the net capable to handle with more complex patterns
3. Using Adam optimization for the gradients updates
4. Using Relu activation function

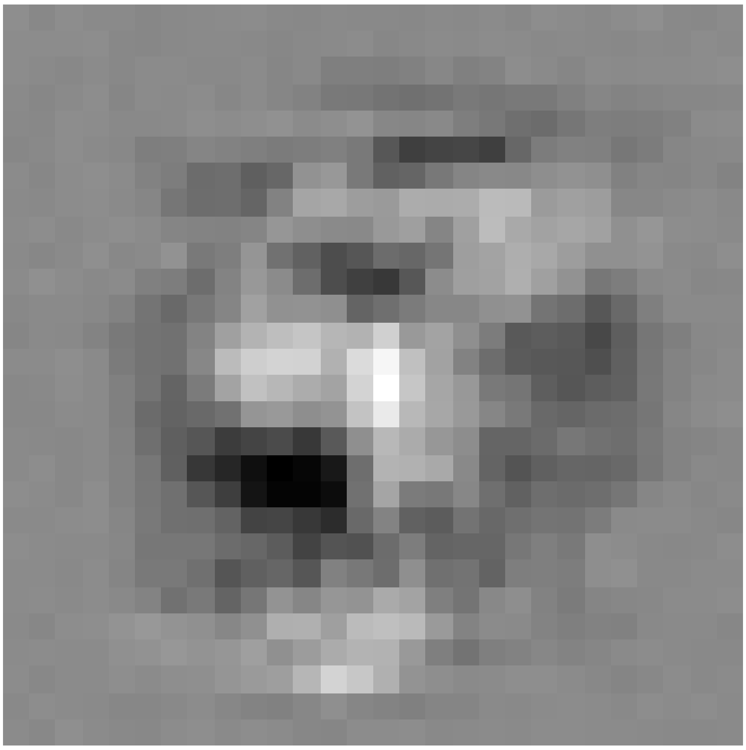
In [61]:

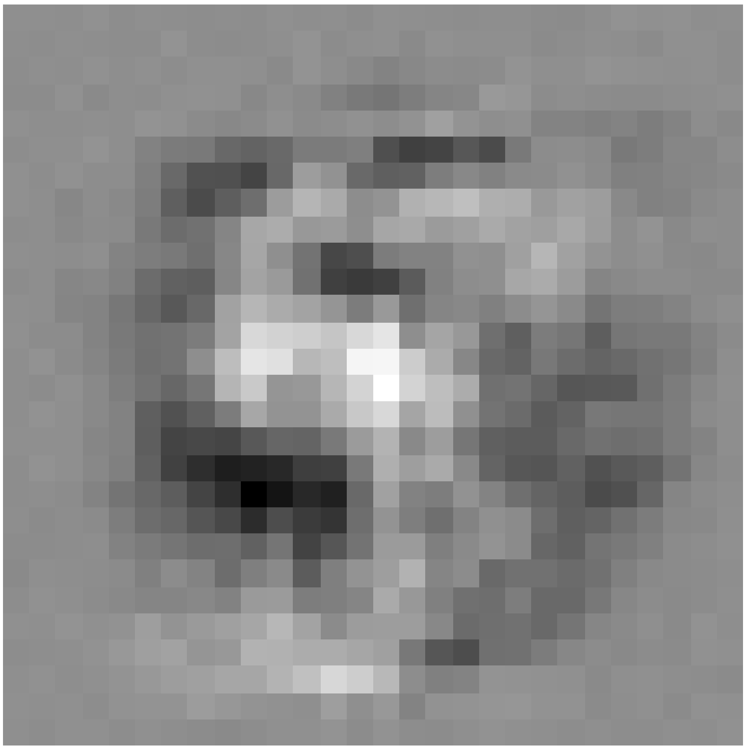
```
# Visualize some weights. features of digits should be somehow present.
def show_net_weights(params):
    W1 = params['W1']
    print(W1.shape)
    for i in range(5):
        W = W1[:,i*5].reshape(28, 28)
        plt.imshow(W,cmap='gray')
        plt.axis('off')
        plt.show()

show_net_weights(net_params)
```

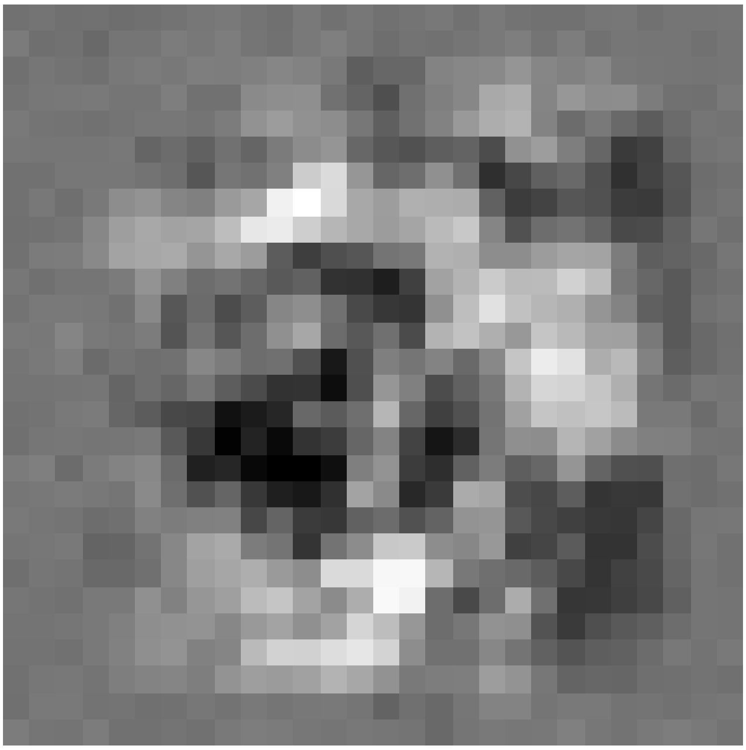
```
(784, 300)
```











Implement, train and test the same two-layer network, using a **deep learning library** (pytorch/tensorflow/keras).

As before, you should be able to receive at least 97% accuracy.

Please note, that in this section you will need to implement the model, the training and the testing by yourself (you may use the code in earlier sections) Don't forget to print the accuracy during training (in the same format as before).

For installing a deep learning library, you should use "!pip3 install..." (lookup the compatible syntex for your library)

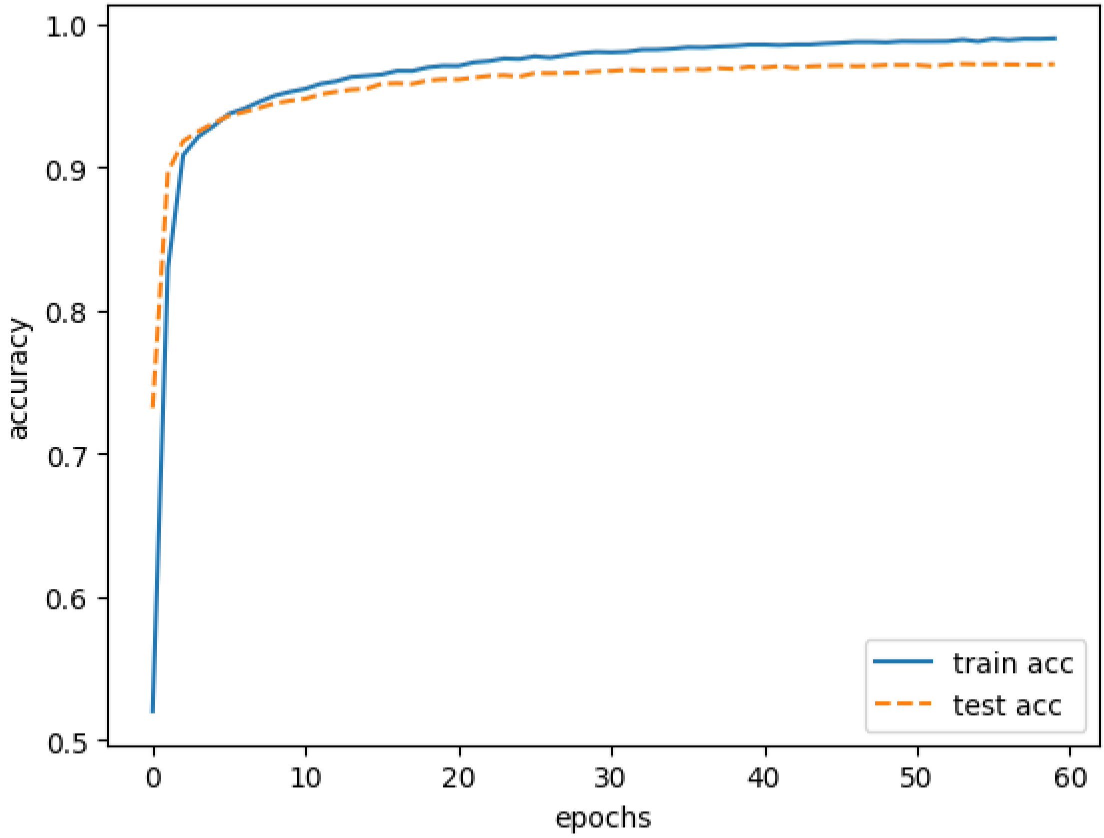
In [62]:

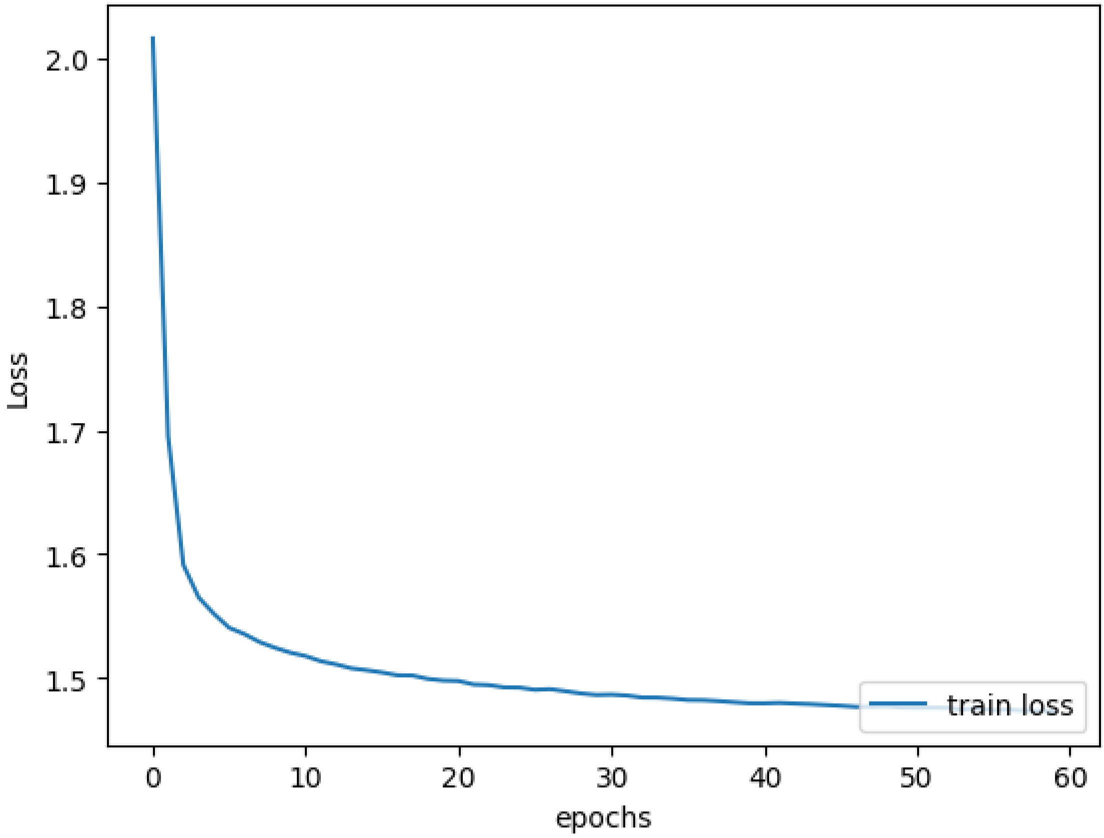
```
 ################################################################################
 #                                YOUR CODE                                     #
 ################################################################################
import torch
from torch import nn, optim

# model architecture using pytorch
class Net(nn.Module):

  def __init__(self,d,h,c):
    super(Net, self).__init__()
    self.fc1 = nn.Linear(d, h)
    self.sigmoid = nn.Sigmoid()
    self.fc2 = nn.Linear(h, c)
    self.softmax = nn.Softmax(dim=1)

  def forward(self, x): # We don't really need so 3 layers, but for the example.
    x = self.fc1(x)
    x = self.sigmoid(x)
    x = self.fc2(x)
    x = self.softmax(x)
    return x
```

```
epochs = 60
mini_batch_size = 800
learning_rate = 0.002
num_hidden_cells = 100

### Train ###
def Train(epochs_num, batch_size, lr, H):
  (x_train, t_train), (x_test, t_test) = load_mnist(normalize=True, one_hot_label=
  x_train = torch.tensor(x_train)
  t_train = torch.tensor(t_train)
  x_test = torch.tensor(x_test)
  t_test = torch.tensor(t_test)

  D = x_train.shape[1]
  model = Net(D, num_hidden_cells,c=10)
  loss_fn = nn.CrossEntropyLoss()
  optimizer = optim.Adam(model.parameters(), lr=learning_rate)


  train_size = x_train.shape[0]
  iter_per_epoch = round(train_size / batch_size)

  train_loss_list = []
  train_acc_list = []
  test_acc_list = []

  for i in range(epochs_num):
    train_loss_iter= []
    train_acc_iter= []

    # Make sure gradient tracking is on, and do a pass over the data
    model.train(True)
    for k in range(iter_per_epoch):
        indices = np.random.choice(x_train.shape[0], mini_batch_size, replace=Fals
        x_batch = x_train[indices]  # Use the selected indices to get the mini-bat
        t_batch = t_train[indices]
        y_batch = model(x_batch) # Feed-forward

        loss = loss_fn(y_batch, t_batch) # Evaluate loss
        optimizer.zero_grad() # Zero the gradients before running the backward pas
        loss.backward() # Compute gradient of the loss with respect to all the lea
        optimizer.step() # Update weights


        # Calculate the loss and accuracy for visalizaton
        train_loss_iter.append(loss.detach().numpy())
        acc_iter=accuracy(y_batch.detach().numpy(), t_batch.detach().numpy())
        train_acc_iter.append(acc_iter)
        if k == iter_per_epoch-1:
            train_acc = np.mean(train_acc_iter)
            train_acc_list.append(train_acc)
            train_loss_list.append(np.mean(train_loss_iter))

            # Set the model to evaluation mode, disabling dropout and using popula
            # statistics for batch normalization.
            model.eval()
            # Disable gradient computation and reduce memory consumption.
            with torch.no_grad():
              y_test = model(x_test)
              test_acc = accuracy(y_test.detach().numpy(), t_test.detach().numpy()
              test_acc_list.append(test_acc)

            print("train acc: " + str(train_acc)[:5] + "% |  test acc: "   + str(t
```

```
  return train_acc_list, test_acc_list, train_loss_list

train_acc, test_acc, train_loss = Train(epochs, mini_batch_size, learning_rate, nu

markers = {'train': 'o', 'test': 's'}
x = np.arange(len(train_acc))
plt.plot(x, train_acc, label='train acc')
plt.plot(x, test_acc, label='test acc', linestyle='--')
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.legend(loc='lower right')
plt.show()

markers = {'train': 'o'}
x = np.arange(len(train_loss))
plt.plot(x, train_loss, label='train loss')
plt.xlabel("epochs")
plt.ylabel("Loss")
plt.legend(loc='lower right')
plt.show()
```

```
train acc: 0.520% | test acc: 0.732% | loss for epoch 0: 2.016349718598525
train acc: 0.829% | test acc: 0.8969% | loss for epoch 1: 1.6951955275754131
train acc: 0.908% | test acc: 0.9184% | loss for epoch 2: 1.5913616967618465
train acc: 0.921% | test acc: 0.9252% | loss for epoch 3: 1.5650449964960416
train acc: 0.928% | test acc: 0.9311% | loss for epoch 4: 1.5517863562901817
train acc: 0.937% | test acc: 0.9362% | loss for epoch 5: 1.5405179875095685
train acc: 0.941% | test acc: 0.9388% | loss for epoch 6: 1.5353892270584901
train acc: 0.945% | test acc: 0.9417% | loss for epoch 7: 1.5289276842931907
train acc: 0.950% | test acc: 0.9448% | loss for epoch 8: 1.524341185839971
train acc: 0.952% | test acc: 0.9466% | loss for epoch 9: 1.5204329003552595
train acc: 0.954% | test acc: 0.948% | loss for epoch 10: 1.5176524895270664
train acc: 0.958% | test acc: 0.9511% | loss for epoch 11: 1.5134926960547768
train acc: 0.960% | test acc: 0.9529% | loss for epoch 12: 1.5109228506584962
train acc: 0.963% | test acc: 0.9543% | loss for epoch 13: 1.507769939806064
train acc: 0.963% | test acc: 0.955% | loss for epoch 14: 1.5062395562529565
train acc: 0.964% | test acc: 0.9585% | loss for epoch 15: 1.5044368065973122
train acc: 0.967% | test acc: 0.9588% | loss for epoch 16: 1.5021684700449307
train acc: 0.967% | test acc: 0.9584% | loss for epoch 17: 1.5019247476716835
train acc: 0.969% | test acc: 0.9607% | loss for epoch 18: 1.4992444056590395
train acc: 0.970% | test acc: 0.9616% | loss for epoch 19: 1.4979133800049624
train acc: 0.970% | test acc: 0.9614% | loss for epoch 20: 1.4975512198607126
train acc: 0.973% | test acc: 0.9627% | loss for epoch 21: 1.49471493066748
train acc: 0.974% | test acc: 0.9639% | loss for epoch 22: 1.4942074022491771
train acc: 0.976% | test acc: 0.9644% | loss for epoch 23: 1.4923927547295888
train acc: 0.975% | test acc: 0.9634% | loss for epoch 24: 1.492200052922964
train acc: 0.977% | test acc: 0.9659% | loss for epoch 25: 1.4904465621570746
train acc: 0.976% | test acc: 0.9658% | loss for epoch 26: 1.4909415003319582
train acc: 0.978% | test acc: 0.966% | loss for epoch 27: 1.489226430215438
train acc: 0.979% | test acc: 0.9662% | loss for epoch 28: 1.4874568323512871
train acc: 0.980% | test acc: 0.9671% | loss for epoch 29: 1.4862785847703615
train acc: 0.980% | test acc: 0.9674% | loss for epoch 30: 1.4865213697711626
train acc: 0.980% | test acc: 0.9682% | loss for epoch 31: 1.4857885480920472
train acc: 0.982% | test acc: 0.9677% | loss for epoch 32: 1.4841684090912344
train acc: 0.982% | test acc: 0.9681% | loss for epoch 33: 1.4840994959553084
train acc: 0.982% | test acc: 0.9682% | loss for epoch 34: 1.4833053302983443
train acc: 0.983% | test acc: 0.9687% | loss for epoch 35: 1.4821642176826795
train acc: 0.983% | test acc: 0.9683% | loss for epoch 36: 1.4820411660770576
train acc: 0.984% | test acc: 0.9693% | loss for epoch 37: 1.4812977693835896
train acc: 0.984% | test acc: 0.9688% | loss for epoch 38: 1.480405053647359
train acc: 0.985% | test acc: 0.9704% | loss for epoch 39: 1.479542271943887
train acc: 0.985% | test acc: 0.9698% | loss for epoch 40: 1.4795046944955985
train acc: 0.985% | test acc: 0.9708% | loss for epoch 41: 1.4798519312918186
train acc: 0.985% | test acc: 0.9695% | loss for epoch 42: 1.4792826894481974
train acc: 0.985% | test acc: 0.9706% | loss for epoch 43: 1.478849167609215
train acc: 0.986% | test acc: 0.971% | loss for epoch 44: 1.4781509316424528
train acc: 0.987% | test acc: 0.9712% | loss for epoch 45: 1.4775067014912762
train acc: 0.987% | test acc: 0.9707% | loss for epoch 46: 1.4766339571237563
train acc: 0.987% | test acc: 0.971% | loss for epoch 47: 1.4767116667449476
train acc: 0.987% | test acc: 0.9716% | loss for epoch 48: 1.4769429445266726
train acc: 0.988% | test acc: 0.9715% | loss for epoch 49: 1.475986590298017
train acc: 0.988% | test acc: 0.9716% | loss for epoch 50: 1.475967425831159
train acc: 0.988% | test acc: 0.9707% | loss for epoch 51: 1.4759527928948402
train acc: 0.988% | test acc: 0.9719% | loss for epoch 52: 1.4758059307297071
train acc: 0.989% | test acc: 0.9723% | loss for epoch 53: 1.474506852742036
train acc: 0.988% | test acc: 0.972% | loss for epoch 54: 1.4755352566818394
train acc: 0.989% | test acc: 0.9721% | loss for epoch 55: 1.4737331668953102
train acc: 0.988% | test acc: 0.9719% | loss for epoch 56: 1.474411089424292
train acc: 0.989% | test acc: 0.9718% | loss for epoch 57: 1.4735004662116369
train acc: 0.989% | test acc: 0.9717% | loss for epoch 58: 1.4734897760490575
train acc: 0.990% | test acc: 0.9722% | loss for epoch 59: 1.4730522946854432
```


accuracy
epochs
train acc
test acc
1.0
0.9
0.8
0.7
0.6
0.5
0
10
20
30
40
50
60


Loss
epochs
train loss
2.0
1.9
1.8
1.7
1.6
1.5
0
10
20
30
40
50
60