MNIST Digits Classification using Neural Networks

In this part we will implement our first Neural Network! We will use fully connecter Neural Network in order to classify handwritten digits. We will use the well known MNIST dataset. The MNIST database (Modified National Institute of Standards and Technology database) is a large database of handwritten digits that is commonly used for training various image processing systems. The database is also widely used for training and testing in the field of machine

learning.

Imports

```
#importing modules that will be in use
%matplotlib inline
import os
import sys
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import random
import time
import copy
```

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision
```

```
from torchvision import datasets, transforms
from torch.utils.data import Dataset, DataLoader
from sklearn.model_selection import train_test_split
```

Mount your drive in order to run locally with colab

```
from google.colab import drive
drive.mount('/content/drive')
%cd /content/drive/MyDrive/Deep Learning/Assignment 2/FC
from utils import *
```

```
Mounted at /content/drive
/content/drive/MyDrive/Deep Learning/Assignment 2/FC
```

QUESTION 1: What are the problems with sigmoid?

ANSWER:

- In a large area, the function has zero derivatives, which means that there are many cases where we don't have gradients that we can improve
- The output range isn't centered around zero.

here we will implement the sigmoid activation function and it's gradient. You should not use any build-in function of sigmoid.

```
def sigmoid(x):
    # impement the sigmoid funciton
    # ====== YOUR CODE: ======
    sig = 1/(1 + torch.exp(-x))
    # ========================
    return sig
```

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


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

     """
    #for stability (do not change)
    max_per_row, inds = torch.max(x, dim=1)
    x = (x.T - max_per_row).T
```

```
    # ====== YOUR CODE: ======
    probabilities = torch.exp(x)/ sum(torch.exp(x))
    # ========================
    return probabilities
```

Implement a fully-vectorized loss function for the Softmax classifier.

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

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

    Returns:
    - loss as single float (do not forget to divide by the number of samples in the
    """
    # ====== YOUR CODE: ======
    # Compute loss
    y_size = y.shape[0]
    error = -torch.sum(t*torch.log(y))/y_size
    # ========================
    return error


def get_accuracy(y, t):
    """
    Computes the accuracy of the NN's predictions.
    Inputs:
    - t:  A torch 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 torch probabilities for the minibatch (at the end of the forward pass)
    Returns:
    - accuracy: a single float of the average accuracy.
    """
    # ====== YOUR CODE: ======
    pass
    tmax = torch.argmax(t, dim=1)
    ymax = torch.argmax(y, dim=1)
    correct = (tmax.cpu().numpy() == ymax.cpu().numpy()).sum()
    accuracy = float(correct / y.shape[0])
    # ========================
    return accuracy
```

Fully-connected Network

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.

We will use torch.nn for the linear functions

config

```
args={}
args['batch_size']=1000
args['test_batch_size']=1000
args['epochs']=35  #The number of Epochs
args['validation_ratio']=0.15 #The validation ratio from training set
args['eval_every']=1 #Will evaluate the model ever <eval_every> epochs
```

load the data

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print("using " + device)
```

```
    using cuda
```

```
def create_train_validation_loaders(dataset: Dataset,
                                    validation_ratio,
                                    batch_size=100):
    """
    Splits a dataset into a train and validation set, returning a
    DataLoader for each.
    :param dataset: The original dataset.
    :param validation_ratio: Ratio (in range 0,1) of the validation set size to
        total dataset size.
    :param batch_size: Batch size the loaders will return from each set.
    :return: A tuple of train, validation and test DataLoader instances.
    """
    if not(0.0 < validation_ratio < 1.0):
        raise ValueError(validation_ratio)


    # TODO: Create two DataLoader instances, dataloader_train and dataloader_valid
    # They should together represent a train/validation split of the given
    # dataset. Make sure that:
    # 1. Validation set size is validation_ratio * total number of samples.
    # 2. No sample is in both datasets. You can select samples at random
    #    from the dataset.
    # 3. you use shuffle=True in the train dataloader and shuffle=False in the val:


    # ====== YOUR CODE: ======
```

```
    train_dataset, validation_dataset = train_test_split(dataset, test_size = valic

    dl_train = DataLoader(train_dataset, batch_size, shuffle=True)
    dl_valid = DataLoader(validation_dataset, batch_size, shuffle=False)
    # =======================

    return dl_train, dl_valid

#load the data
dataset = datasets.MNIST('./data', train=True, download=True,
                         transform=transforms.Compose([transforms.ToTensor(),
                                                       transforms.Normalize((0.1307
                        )

train_loader, val_loader = create_train_validation_loaders(dataset,
                                                            validation_ratio = args[
                                                            batch_size= args['batch_


test_loader = torch.utils.data.DataLoader(
    datasets.MNIST('./data', train=False, download=True,
                   transform=transforms.Compose([
                       transforms.ToTensor(),
                       transforms.Normalize((0.1307,), (0.3081,))
                   ])),
    batch_size=args['test_batch_size'], shuffle=False)


dataloaders = {'training':train_loader,
               'val':val_loader,
               'test':test_loader
              }
```

Fully connected Neural Network

```
class FullyConnectedNeuralNetwork(nn.Module):
    #This defines the structure of the NN.
    def __init__(self,
                 hidden_layer_dim
                 ):
      super(FullyConnectedNeuralNetwork, self).__init__()
      # Define the model layers.
      # Use the torch.nn.Linear layers. Set the hidden layer dim to hidden_layer_di
      # Notice that the input dim is 784 and the output dim is 10 (number of classe
      # ====== YOUR CODE: ======
      self.fc1 = torch.nn.Linear(784, hidden_layer_dim)
      self.fc2 = nn.Linear(hidden_layer_dim, 10)
      # =======================

    def forward(self, x):
      x = torch.flatten(x, start_dim=1,end_dim=-1)
      # ====== YOUR CODE: ======
      x = sigmoid(self.fc1(x))
```

```
        y = softmax(self.fc2(x))
        # ======================
        return y
```

The following functions will train our model

```
def forward_one_epoch(loader,
                      optimizer,
                      net,
                      mode,
                      progress_bar_str,
                      num_of_epochs
                     ):


    losses, cur_accuracies = [], []
    all_preds,all_targets = [], []
    for batch_idx, (inputs, targets) in enumerate(loader):

        if mode == Mode.training:
            optimizer.zero_grad()

        inputs, targets =inputs.to(device), targets.to(device)
        outputs = net(inputs)
        targets = F.one_hot(targets, num_classes=10)
        loss = cross_entropy_error(outputs, targets)
        losses.append(loss.item())

        if mode == Mode.training:
            #do a step
            loss.backward()
            optimizer.step()

        if len(targets.shape) ==2:
            cur_accuracies.append(get_accuracy(outputs, targets))

        if batch_idx %20 ==0:
            progress_bar(batch_idx, len(loader), progress_bar_str
                   % (num_of_epochs, np.mean(losses), losses[-1], np.mean(cur_accu

        targets_cpu = targets.cpu().data.numpy()
        outputs_cpu = [i.cpu().data.numpy() for i in outputs]
        outputs_cpu = np.argmax(outputs_cpu, axis=1)

        all_targets.extend(targets_cpu)
        all_preds.extend(outputs_cpu)

        del inputs, targets, outputs
        torch.cuda.empty_cache()


    return losses, cur_accuracies, all_targets, all_preds
```

```
def train(args, dataloaders):
  seed = 0
  torch.manual_seed(seed)
  if torch.cuda.is_available():
      torch.cuda.manual_seed_all(seed)

  model = FullyConnectedNeuralNetwork(hidden_layer_dim = args['hidden_layer_dim'])
  model = model.to(device)

  optimizer = torch.optim.SGD(model.parameters(), args['lr'])

  training_accuracies, val_accuracies = [], []
  training_losses, val_losses = [], []


  training_loader = dataloaders['training']
  val_loader = dataloaders['val']
  test_loader = dataloaders['test']

  best_acc = -1

  #start training
  for epoch in range(1, args['epochs']+1):

      #training
      model = model.train()

      progress_bar_str = 'Train: repeat %d -- Mean Loss: %.3f | Last Loss: %.3f | T

      losses, cur_training_accuracies, _,_ =  forward_one_epoch(
          loader = training_loader,
          optimizer = optimizer,
          net = model,
          mode = Mode.training,
          progress_bar_str = progress_bar_str,
          num_of_epochs = epoch)


      train_epoch_acc = np.mean(cur_training_accuracies)
      train_epoch_loss= np.mean(losses)
      sys.stdout.flush()
      print()
      print(f'Train epoch {epoch}: accuracy {train_epoch_acc}, loss {train_epoch_lo
      training_accuracies.append(train_epoch_acc)
      training_losses.append(train_epoch_loss)



      # validation
      model.eval()
```

```
        if (epoch-1)%args['eval_every']==0:
            progress_bar_str = 'Validation: repeat %d -- Mean Loss: %.3f | Last Loss:

            losses, cur_val_accuracies,_,_ =  forward_one_epoch(val_loader,
                                                                optimizer,
                                                                model,
                                                                Mode.validation,
                                                                progress_bar_str,
                                                                epoch
                                                               )
            val_epoch_acc= np.mean(cur_val_accuracies)
            val_epoch_loss= np.mean(losses)
            sys.stdout.flush()
            val_accuracies.append(val_epoch_acc)

            val_epoch_acc = np.round(val_epoch_acc,3)
            print()
            print(f'Validation epoch {epoch//args["eval_every"]}: accuracy {val_epoch
            val_losses.append(val_epoch_loss)

            cur_acc_loss = {
              'training_accuracies':training_accuracies,
              'val_accuracies':val_accuracies,
              'training_losses':training_losses,
              'val_losses':val_losses
                          }

            if best_acc +0.001 < val_epoch_acc:

                best_acc = val_epoch_acc
                best_acc_epoch = epoch

                print(f'========== new best model! epoch {best_acc_epoch}, accuracy
                best_model = copy.deepcopy(model)


    progress_bar_str = 'Test: repeat %d -- Mean Loss: %.3f | Last Loss: %.3f | Acc: %
    test_losses, test_cur_test_accuracies, test_all_targets, test_all_preds = forward
                                                                None,
                                                                best_model,
                                                                Mode.test,
                                                                progress_bar_str,
                                                                0)

    test_epoch_acc= np.mean(test_cur_test_accuracies)
    test_epoch_loss= np.mean(test_losses)
    print("==================== Test Results ====================")
    print(f'Test Accuracy : {test_epoch_acc}')
    print(f'Test Loss : {test_epoch_loss}')
    return best_model, cur_acc_loss
```

Training Process

We will finetune two hyper parameters:

1. The hidden layer dimension.
2. The learning rate.

Finetuning hidden_layer_dim

hidden_layer_dim = 1, lr = 0.0001

Set the hidden_layer_dim to 1 and the lr to 0.0001 and train the model.

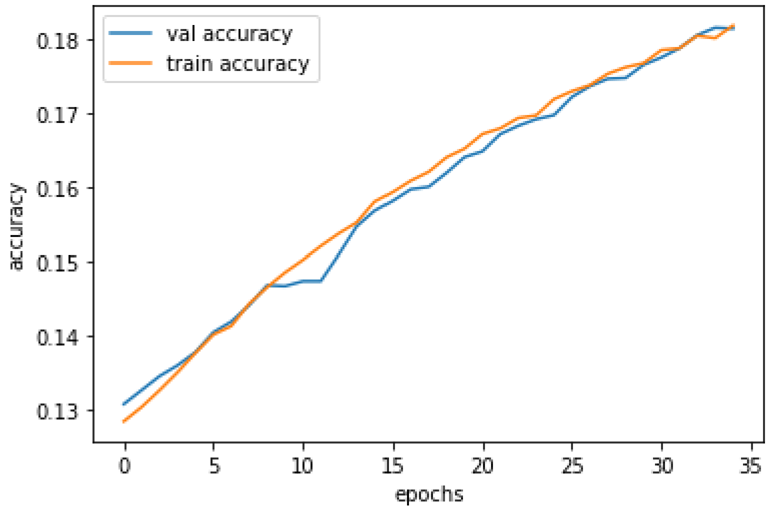
```
args['hidden_layer_dim'] = 1
args['lr']=0.0001
best_model, cur_acc_loss  = train(args, dataloaders)
```

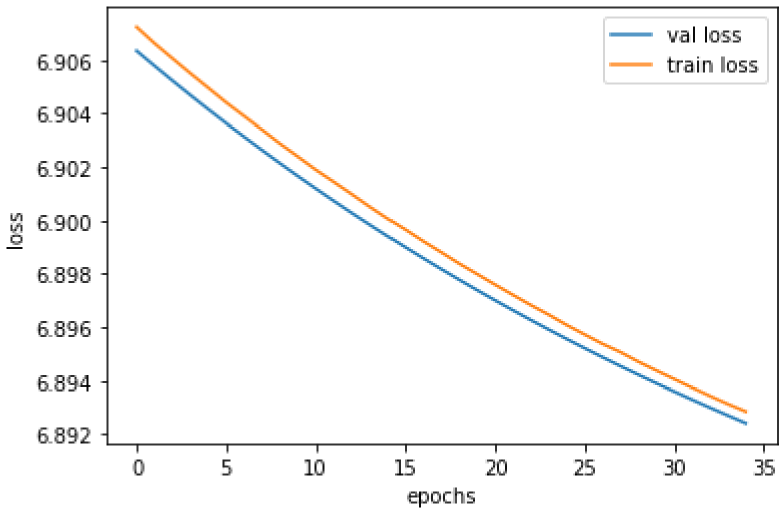
```
Train epoch 17: accuracy 0.16082352941176473, loss 6.899216754763734

Validation epoch 17: accuracy 0.16, loss 6.898579332563612
========== new best model! epoch 17, accuracy 0.16  ==========

Train epoch 18: accuracy 0.16201960784313726, loss 6.8988033930460615

Validation epoch 18: accuracy 0.16, loss 6.8981733322143555

Train epoch 19: accuracy 0.16398039215686275, loss 6.89838004579731

Validation epoch 19: accuracy 0.162, loss 6.8977753851148815
========== new best model! epoch 19, accuracy 0.162  ==========

Train epoch 20: accuracy 0.1651372549019608, loss 6.897987692963843

Validation epoch 20: accuracy 0.164, loss 6.897385491265191
========== new best model! epoch 20, accuracy 0.164  ==========

Train epoch 21: accuracy 0.1671176470588235, loss 6.897591777876312

Validation epoch 21: accuracy 0.165, loss 6.897004233466254

Train epoch 22: accuracy 0.16788235294117643, loss 6.897204932044534

Validation epoch 22: accuracy 0.167, loss 6.896630393134223
========== new best model! epoch 22, accuracy 0.167  ==========

Train epoch 23: accuracy 0.16927450980392159, loss 6.8968236025641945

Validation epoch 23: accuracy 0.168, loss 6.89626423517863

Train epoch 24: accuracy 0.16962745098039214, loss 6.896463066923852

Validation epoch 24: accuracy 0.169, loss 6.895905176798503
========== new best model! epoch 24, accuracy 0.169  ==========

Train epoch 25: accuracy 0.17182352941176474, loss 6.896077520707074
```

```
Validation epoch 25: accuracy 0.17, loss 6.89555385377672

Train epoch 26: accuracy 0.17288235294117643, loss 6.8957214261971265

Validation epoch 26: accuracy 0.172, loss 6.895209948221843
========== new best model! epoch 26, accuracy 0.172  ==========

Train epoch 27: accuracy 0.17370588235294118, loss 6.8953720822053794

Validation epoch 27: accuracy 0.174, loss 6.894872294531928
========== new best model! epoch 27, accuracy 0.174  ==========

Train epoch 28: accuracy 0.17523529411764707, loss 6.895062399845497

Validation epoch 28: accuracy 0.175, loss 6.894542005327013

Train epoch 29: accuracy 0.17611764705882352, loss 6.894705912646125

Validation epoch 29: accuracy 0.175, loss 6.894218338860406
```

```
training_accuracies = cur_acc_loss['training_accuracies']
val_accuracies = cur_acc_loss['val_accuracies']
training_losses = cur_acc_loss['training_losses']
val_losses = cur_acc_loss['val_losses']
plot_graphs(training_accuracies, val_accuracies, training_losses, val_losses)
```

```
Best val accuracy was 0.18144444444444446, at epoch 33
```






```
0.18144444444444446
```

QUESTION 2.1: What are the accuracy and loss values? Explain the loss and accuracy graphs.

ANSWER:

Test Accuracy : 0.1829

Test Loss : 6.892094945907592

On the graph we can see that as we make more epochs the model is getting better and the accuracy is increasing while the loss is decreasing.

Despite the improvements, accuracy is still low, and the loss is still high, which suggests that the hidden layer dim is too small, which weakens the model.

hidden_layer_dim = 5, lr = 0.0001

Set the hidden_layer_dim to 5 and the lr to 0.0001 and train the model.

```
args['hidden_layer_dim'] = 5
args['lr']=0.0001
best_model, cur_acc_loss  = train(args, dataloaders)
```

```
Validation epoch 24: accuracy 0.243, loss 6.864617347717285
========== new best model! epoch 24, accuracy 0.243  ==========

Train epoch 25: accuracy 0.24719607843137256, loss 6.862949969721775

Validation epoch 25: accuracy 0.247, loss 6.863710509406196
========== new best model! epoch 25, accuracy 0.247  ==========

Train epoch 26: accuracy 0.2500588235294118, loss 6.862043764076981

Validation epoch 26: accuracy 0.249, loss 6.862808174557156
========== new best model! epoch 26, accuracy 0.249  ==========

Train epoch 27: accuracy 0.25296078431372554, loss 6.8611343327690575

Validation epoch 27: accuracy 0.253, loss 6.861909972296821
========== new best model! epoch 27, accuracy 0.253  ==========

Train epoch 28: accuracy 0.2569019607843137, loss 6.8602188428243

Validation epoch 28: accuracy 0.256, loss 6.861015796661377
========== new best model! epoch 28, accuracy 0.256  ==========

Train epoch 29: accuracy 0.26050980392156864, loss 6.8592975466859105

Validation epoch 29: accuracy 0.259, loss 6.86012601852417
========== new best model! epoch 29, accuracy 0.259  ==========

Train epoch 30: accuracy 0.2634313725490196, loss 6.858407862046185

Validation epoch 30: accuracy 0.262, loss 6.859240108066135
========== new best model! epoch 30, accuracy 0.262  ==========

Train epoch 31: accuracy 0.26539215686274514, loss 6.857500805574305
```

```
Validation epoch 31: accuracy 0.264, loss 6.858358701070149
========== new best model! epoch 31, accuracy 0.264  ==========

Train epoch 32: accuracy 0.26843137254901955, loss 6.856607998118681

Validation epoch 32: accuracy 0.266, loss 6.8574814266628685
========== new best model! epoch 32, accuracy 0.266  ==========

Train epoch 33: accuracy 0.2715294117647059, loss 6.855698080623851

Validation epoch 33: accuracy 0.268, loss 6.856608443790012
========== new best model! epoch 33, accuracy 0.268  ==========

Train epoch 34: accuracy 0.2744901960784314, loss 6.85481388428632

Validation epoch 34: accuracy 0.271, loss 6.855739699469672
========== new best model! epoch 34, accuracy 0.271  ==========

Train epoch 35: accuracy 0.27723529411764714, loss 6.8539470597809435

Validation epoch 35: accuracy 0.272, loss 6.85487519370185
===================== Test Results =====================
Test Accuracy : 0.2785
```

```
training_accuracies = cur_acc_loss['training_accuracies']
val_accuracies = cur_acc_loss['val_accuracies']
training_losses = cur_acc_loss['training_losses']
val_losses = cur_acc_loss['val_losses']
plot_graphs(training_accuracies, val_accuracies, training_losses, val_losses)
```

```
Best val accuracy was 0.27244444444444443, at epoch 34
```

QUESTION 2.2: What are the accuracy and loss values? Explain the loss and accuracy graphs.

ANSWER:

Test Accuracy : 0.2785

Test Loss : 6.8530741214752195

The graphs are similar to the previous graphs, but one can see that after increasing the hidden layer dimension, the accuracy has improved, and the loss has decreased, although it's still not good enough, we can also see that the improvement between epochs is very small, which implies that the learning rate is too low, and the hidden layer dimension is still too small.

hidden_layer_dim = 100, lr = 0.0001

Set the hidden_layer_dim to 100 and the lr to 0.0001 and train the model.

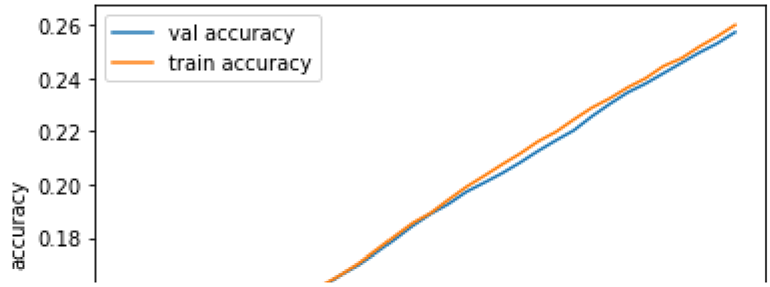
```
args['hidden_layer_dim'] = 100
args['lr']=0.0001
best_model, cur_acc_loss  = train(args, dataloaders)
```

```
========== new best model! epoch 24, accuracy 0.213  ==========

Train epoch 25: accuracy 0.21986274509803924, loss 6.880293743283141

Validation epoch 25: accuracy 0.217, loss 6.879737589094374
========== new best model! epoch 25, accuracy 0.217  ==========

Train epoch 26: accuracy 0.2245294117647059, loss 6.8791715771544215

Validation epoch 26: accuracy 0.22, loss 6.878629631466335
========== new best model! epoch 26, accuracy 0.22  ==========

Train epoch 27: accuracy 0.22892156862745097, loss 6.878061724644081

Validation epoch 27: accuracy 0.226, loss 6.877523793114556
========== new best model! epoch 27, accuracy 0.226  ==========

Train epoch 28: accuracy 0.23243137254901955, loss 6.876936977984858

Validation epoch 28: accuracy 0.23, loss 6.876418643527561
========== new best model! epoch 28, accuracy 0.23  ==========

Train epoch 29: accuracy 0.23654901960784314, loss 6.875819729823692

Validation epoch 29: accuracy 0.235, loss 6.875314553578694
========== new best model! epoch 29, accuracy 0.235  ==========

Train epoch 30: accuracy 0.24, loss 6.87472079781925

Validation epoch 30: accuracy 0.238, loss 6.874212053087023
```

```
========== new best model! epoch 30, accuracy 0.238  ==========

Train epoch 31: accuracy 0.24456862745098037, loss 6.873606728572471

Validation epoch 31: accuracy 0.242, loss 6.873110665215386
========== new best model! epoch 31, accuracy 0.242  ==========

Train epoch 32: accuracy 0.24750980392156863, loss 6.872502981447706

Validation epoch 32: accuracy 0.246, loss 6.872010601891412
========== new best model! epoch 32, accuracy 0.246  ==========

Train epoch 33: accuracy 0.2518823529411765, loss 6.871399561564128

Validation epoch 33: accuracy 0.25, loss 6.870911757151286
========== new best model! epoch 33, accuracy 0.25  ==========

Train epoch 34: accuracy 0.2557254901960785, loss 6.870271177852855

Validation epoch 34: accuracy 0.253, loss 6.869813919067383
========== new best model! epoch 34, accuracy 0.253  ==========

Train epoch 35: accuracy 0.2600588235294118, loss 6.86917391945334

Validation epoch 35: accuracy 0.257, loss 6.868717511494954
========== new best model! epoch 35, accuracy 0.257  ==========
==================== Test Results ====================
Test Accuracy : 0.2509
Test Loss : 6.870093584060669
```

```
training_accuracies = cur_acc_loss['training_accuracies']
val_accuracies = cur_acc_loss['val_accuracies']
training_losses = cur_acc_loss['training_losses']
val_losses = cur_acc_loss['val_losses']
plot_graphs(training_accuracies, val_accuracies, training_losses, val_losses)
```

```
Best val accuracy was 0.2573333333333333, at epoch 34
```



QUESTION 2.3: What are the accuracy and loss values? Explain the loss and accuracy graphs.

ANSWER:

Test Accuracy : 0.2509

Test Loss : 6.870093584060669

As we saw in the previous train, the model is improving between every epoch, but the improvement is too slow, probably because the learning rate is too low.

hidden_layer_dim = 200, lr = 0.0001

Set the hidden_layer_dim to 200 and the lr to 0.0001 and train the model.

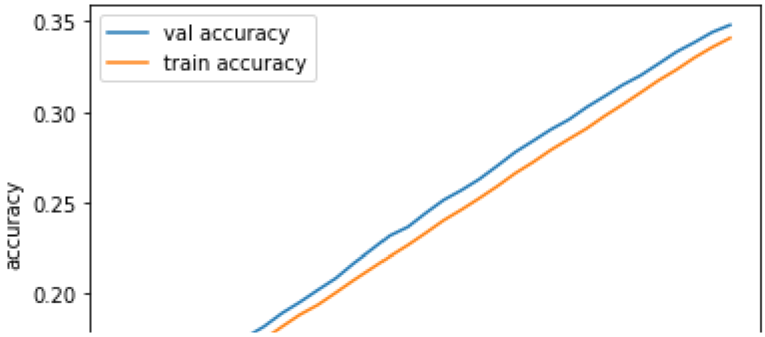
```
args['hidden_layer_dim'] = 200
args['lr']=0.0001
best_model, cur_acc_loss  = train(args, dataloaders)
```

```
========== new best model! epoch 24, accuracy 0.284  ==========

Train epoch 25: accuracy 0.2790392156862745, loss 6.86557507982441

Validation epoch 25: accuracy 0.29, loss 6.8647069401211205
========== new best model! epoch 25, accuracy 0.29  ==========

Train epoch 26: accuracy 0.2850196078431373, loss 6.864276652242623

Validation epoch 26: accuracy 0.296, loss 6.8634179963005915
========== new best model! epoch 26, accuracy 0.296  ==========

Train epoch 27: accuracy 0.2909019607843137, loss 6.862999074599323

Validation epoch 27: accuracy 0.303, loss 6.862130165100098
========== new best model! epoch 27, accuracy 0.303  ==========

Train epoch 28: accuracy 0.29776470588235293, loss 6.861695551404766

Validation epoch 28: accuracy 0.309, loss 6.860843393537733
========== new best model! epoch 28, accuracy 0.309  ==========

Train epoch 29: accuracy 0.3040392156862745, loss 6.860396955527511

Validation epoch 29: accuracy 0.315, loss 6.859557734595405
========== new best model! epoch 29, accuracy 0.315  ==========

Train epoch 30: accuracy 0.3106470588235294, loss 6.859106540679932
```

```
Validation epoch 30: accuracy 0.32, loss 6.858273771074083
========== new best model! epoch 30, accuracy 0.32  ==========

Train epoch 31: accuracy 0.31735294117647056, loss 6.857806822832893

Validation epoch 31: accuracy 0.327, loss 6.856990973154704
========== new best model! epoch 31, accuracy 0.327  ==========

Train epoch 32: accuracy 0.3232745098039216, loss 6.856512967277975

Validation epoch 32: accuracy 0.333, loss 6.855709340837267
========== new best model! epoch 32, accuracy 0.333  ==========

Train epoch 33: accuracy 0.3298823529411765, loss 6.855239241730933

Validation epoch 33: accuracy 0.338, loss 6.8544290860493975
========== new best model! epoch 33, accuracy 0.338  ==========

Train epoch 34: accuracy 0.33570588235294124, loss 6.853939514534146

Validation epoch 34: accuracy 0.344, loss 6.853150049845378
========== new best model! epoch 34, accuracy 0.344  ==========

Train epoch 35: accuracy 0.34080392156862743, loss 6.852659823847752

Validation epoch 35: accuracy 0.348, loss 6.851872232225206
========== new best model! epoch 35, accuracy 0.348  ==========
==================== Test Results ====================
Test Accuracy : 0.3556
```

```
training_accuracies = cur_acc_loss['training_accuracies']
val_accuracies = cur_acc_loss['val_accuracies']
training_losses = cur_acc_loss['training_losses']
val_losses = cur_acc_loss['val_losses']
plot_graphs(training_accuracies, val_accuracies, training_losses, val_losses)
```

```
Best val accuracy was 0.34800000000000003, at epoch 34
```



QUESTION 2.4: What are the accuracy and loss values? Explain the loss and accuracy graphs.

ANSWER:

Test Accuracy : 0.3556

Test Loss : 6.849764823913574

We can see from those graphs that train and test sets get almost the same results. That means that the model isn't learning only the test samples (overfitting), but that it exceeds to get the same accuracy for both graphs as well. As for the LR, its low value affects the model as we described above.

Finetuning learning rate

hidden_layer_dim = 100, lr = 0.000001

Set the hidden_layer_dim to 100 and the lr to 0.000001 and train the model.

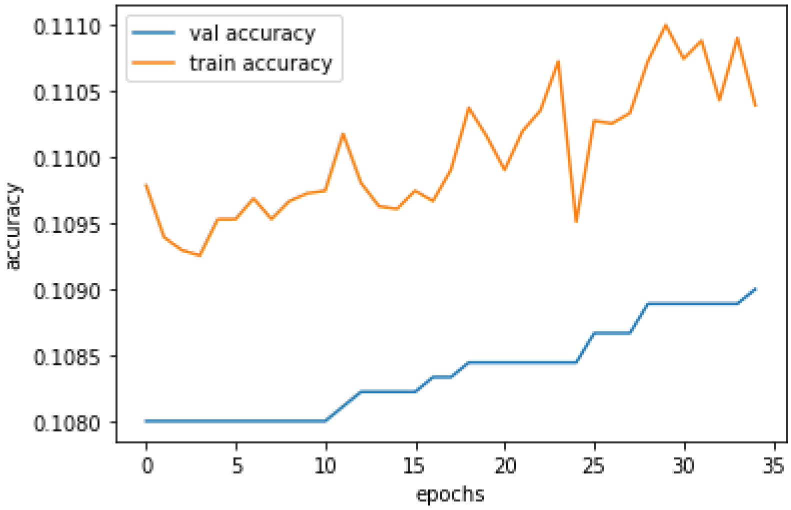
```
args['hidden_layer_dim'] = 100
args['lr']=0.000001
best_model, cur_acc_loss  = train(args, dataloaders)
```

```
Train epoch 22: accuracy 0.11019607843137255, loss 6.90787496753767 45

Validation epoch 22: accuracy 0.108, loss 6.907660802205403

Train epoch 23: accuracy 0.11035294117647058, loss 6.907874331754797

Validation epoch 23: accuracy 0.108, loss 6.907649411095513

Train epoch 24: accuracy 0.11072549019607845, loss 6.907846235761456

Validation epoch 24: accuracy 0.108, loss 6.907638125949436

Train epoch 25: accuracy 0.10950980392156863, loss 6.907859072965734

Validation epoch 25: accuracy 0.108, loss 6.907626787821452

Train epoch 26: accuracy 0.1102745098039216, loss 6.907839700287464

Validation epoch 26: accuracy 0.109, loss 6.907615396711561
```

```
Train epoch 27: accuracy 0.1102549019607843, loss 6.9078324822818535

Validation epoch 27: accuracy 0.109, loss 6.9076037936740455

Train epoch 28: accuracy 0.11033333333333331, loss 6.907813941731172

Validation epoch 28: accuracy 0.109, loss 6.9075925085279675

Train epoch 29: accuracy 0.11072549019607845, loss 6.907805442810059

Validation epoch 29: accuracy 0.109, loss 6.907581170399983

Train epoch 30: accuracy 0.11099999999999997, loss 6.907810753467036

Validation epoch 30: accuracy 0.109, loss 6.907569885253906

Train epoch 31: accuracy 0.11074509803921569, loss 6.907787977480421

Validation epoch 31: accuracy 0.109, loss 6.907558441162109

Train epoch 32: accuracy 0.11088235294117647, loss 6.907785967284558

Validation epoch 32: accuracy 0.109, loss 6.9075469970703125

Train epoch 33: accuracy 0.11043137254901962, loss 6.907776393142401

Validation epoch 33: accuracy 0.109, loss 6.907535817888048

Train epoch 34: accuracy 0.11090196078431372, loss 6.9077491573258945

Validation epoch 34: accuracy 0.109, loss 6.907524320814344

Train epoch 35: accuracy 0.1103921568627451, loss 6.907748680488736

Validation epoch 35: accuracy 0.109, loss 6.907512929704454
==================== Test Results ====================
Test Accuracy : 0.0982
```

```
training_accuracies = cur_acc_loss['training_accuracies']
val_accuracies = cur_acc_loss['val_accuracies']
training_losses = cur_acc_loss['training_losses']
val_losses = cur_acc_loss['val_losses']
plot_graphs(training_accuracies, val_accuracies, training_losses, val_losses)
```

```
Best val accuracy was 0.10899999999999999, at epoch 34
```



QUESTION 3.1: What are the accuracy and loss values? Explain the loss and accuracy graphs.

ANSWER:

Test Accuracy : 0.0982

Test Loss : 6.910411262512207

We can se that between the epochs the accuaracy and loss rate is barely change, the LR is too low.

hidden_layer_dim = 100, lr = 0.1

Set the hidden_layer_dim to 1 and the lr to 0.1 and train the model.

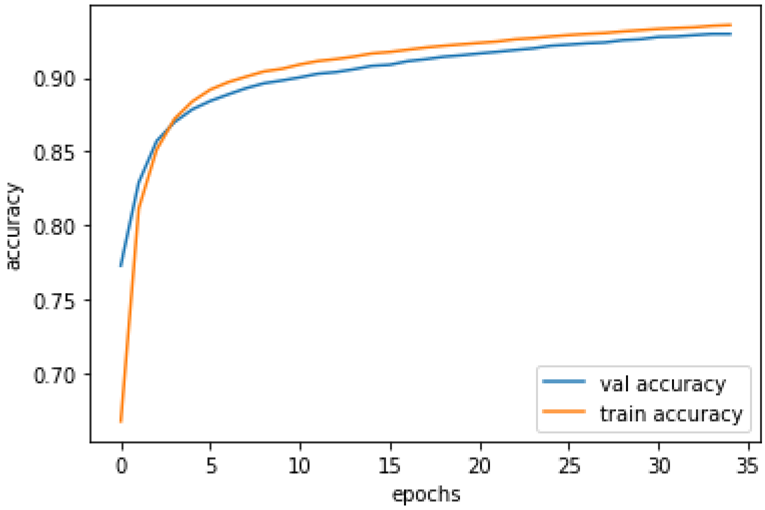
```
args['hidden_layer_dim'] = 100
args['lr']= 0.1
best_model, cur_acc_loss  = train(args, dataloaders)
```

```
Validation epoch 23: accuracy 0.919, loss 4.911419815487331

Train epoch 24: accuracy 0.9268627450980392, loss 4.885538652831433

Validation epoch 24: accuracy 0.92, loss 4.907391283247206
========== new best model! epoch 24, accuracy 0.92  ==========

Train epoch 25: accuracy 0.9279803921568626, loss 4.880699550404268

Validation epoch 25: accuracy 0.922, loss 4.903449323442247
========== new best model! epoch 25, accuracy 0.922  ==========

Train epoch 26: accuracy 0.928921568627451, loss 4.877273251028622

Validation epoch 26: accuracy 0.922, loss 4.89995977613661

Train epoch 27: accuracy 0.929705882352941, loss 4.873430261424944
```

```
Validation epoch 27: accuracy 0.923, loss 4.89656941095988

Train epoch 28: accuracy 0.9302941176470588, loss 4.869928537630567

Validation epoch 28: accuracy 0.924, loss 4.893135282728407
========== new best model! epoch 28, accuracy 0.924  ==========

Train epoch 29: accuracy 0.9313725490196079, loss 4.865903835670621

Validation epoch 29: accuracy 0.925, loss 4.889927175309923

Train epoch 30: accuracy 0.9322549019607842, loss 4.862701107473934

Validation epoch 30: accuracy 0.926, loss 4.8867687649197045
========== new best model! epoch 30, accuracy 0.926  ==========

Train epoch 31: accuracy 0.9330980392156861, loss 4.859874061509674

Validation epoch 31: accuracy 0.928, loss 4.883798016442193
========== new best model! epoch 31, accuracy 0.928  ==========

Train epoch 32: accuracy 0.9337058823529412, loss 4.856206547980215

Validation epoch 32: accuracy 0.928, loss 4.880805439419216

Train epoch 33: accuracy 0.9341960784313726, loss 4.853433562260048

Validation epoch 33: accuracy 0.929, loss 4.878067281511095

Train epoch 34: accuracy 0.9351960784313726, loss 4.851023963853424

Validation epoch 34: accuracy 0.93, loss 4.875068134731716
========== new best model! epoch 34, accuracy 0.93  ==========

Train epoch 35: accuracy 0.9357450980392157, loss 4.8476037137648635

Validation epoch 35: accuracy 0.93, loss 4.872203985850017
==================== Test Results ====================
Test Accuracy : 0.9355999999999998
Test Loss : 4.850043487548828
```

```
training_accuracies = cur_acc_loss['training_accuracies']
val_accuracies = cur_acc_loss['val_accuracies']
training_losses = cur_acc_loss['training_losses']
val_losses = cur_acc_loss['val_losses']
plot_graphs(training_accuracies, val_accuracies, training_losses, val_losses)
```

```
Best val accuracy was 0.9296666666666668, at epoch 33
```



QUESTION 3.2: What are the accuracy and loss values? Explain the loss and accuracy graphs.

ANSWER:

Test Accuracy : 0.9355999999999998

Test Loss : 4.850043487548828

Based on the learning rate and hidden layer dimension, we can see that both the accuracy and the loss are improving significantly between epochs, for both train and validation sets. As we said before because of the low learning rate, we didn't improve between epochs. Now we can see the effect of higher learning rate.

hidden_layer_dim = 100, lr = 0.001

Set the hidden_layer_dim to 100 and the lr to 0.001 and train the model.

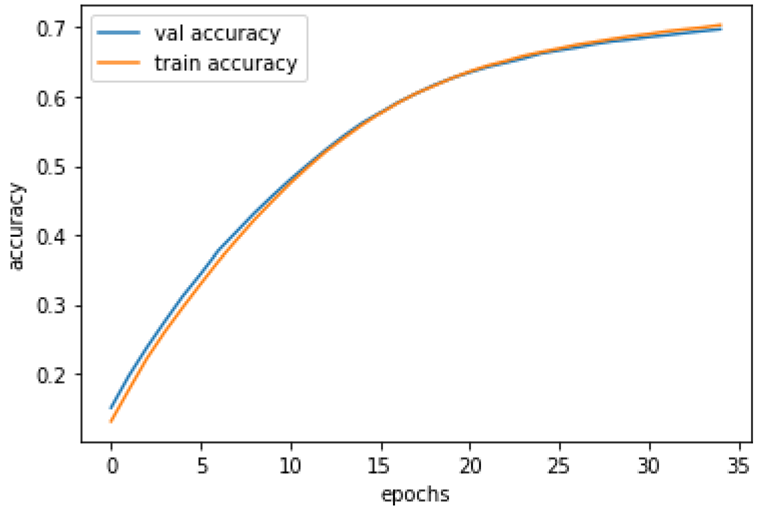
```
args['hidden_layer_dim'] = 100
args['lr']=0.001
best_model, cur_acc_loss  = train(args, dataloaders)
```

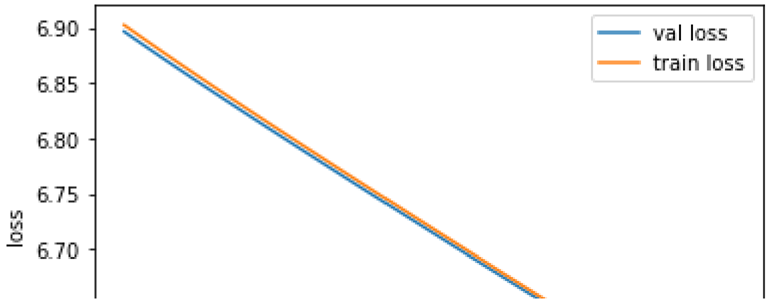
```
========== new best model! epoch 24, accuracy 0.655  ==========

Train epoch 25: accuracy 0.6644901960784313, loss 6.649177149230359

Validation epoch 25: accuracy 0.662, loss 6.645989312065972
========== new best model! epoch 25, accuracy 0.662  ==========

Train epoch 26: accuracy 0.6696470588235296, loss 6.639078944337134

Validation epoch 26: accuracy 0.667, loss 6.635922008090549
========== new best model! epoch 26, accuracy 0.667  ==========

Train epoch 27: accuracy 0.6751176470588235, loss 6.628969744140027
```

```
Validation epoch 27: accuracy 0.671, loss 6.625855763753255
========== new best model! epoch 27, accuracy 0.671  ==========

Train epoch 28: accuracy 0.6790980392156862, loss 6.618792272081562

Validation epoch 28: accuracy 0.676, loss 6.61578204896715
========== new best model! epoch 28, accuracy 0.676  ==========

Train epoch 29: accuracy 0.6834509803921568, loss 6.60857957017188

Validation epoch 29: accuracy 0.68, loss 6.6056952476501465
========== new best model! epoch 29, accuracy 0.68  ==========

Train epoch 30: accuracy 0.6873529411764706, loss 6.598399461484423

Validation epoch 30: accuracy 0.683, loss 6.595593876308865
========== new best model! epoch 30, accuracy 0.683  ==========

Train epoch 31: accuracy 0.6904901960784315, loss 6.588308811187744

Validation epoch 31: accuracy 0.686, loss 6.585485723283556
========== new best model! epoch 31, accuracy 0.686  ==========

Train epoch 32: accuracy 0.6943529411764707, loss 6.5780843659943224

Validation epoch 32: accuracy 0.689, loss 6.575360351138645
========== new best model! epoch 32, accuracy 0.689  ==========

Train epoch 33: accuracy 0.6972941176470586, loss 6.567938552183263

Validation epoch 33: accuracy 0.692, loss 6.565220726860894
========== new best model! epoch 33, accuracy 0.692  ==========

Train epoch 34: accuracy 0.6999411764705882, loss 6.557627827513452

Validation epoch 34: accuracy 0.695, loss 6.555067380269368
========== new best model! epoch 34, accuracy 0.695  ==========

Train epoch 35: accuracy 0.703529411764706, loss 6.547364421919281

Validation epoch 35: accuracy 0.697, loss 6.544892417060004
========== new best model! epoch 35, accuracy 0.697  ==========
==================== Test Results ====================
Test Accuracy : 0.7202999999999999
```

```
training_accuracies = cur_acc_loss['training_accuracies']
val_accuracies = cur_acc_loss['val_accuracies']
training_losses = cur_acc_loss['training_losses']
val_losses = cur_acc_loss['val_losses']
plot_graphs(training_accuracies, val_accuracies, training_losses, val_losses)
```

```
Best val accuracy was 0.6974444444444443, at epoch 34
```





QUESTION 3.3: What are the accuracy and loss values? Explain the loss and accuracy graphs.

ANSWER:

Test Accuracy : 0.7202999999999999

Test Loss : 6.5369964122772215

the accuracy and loss got worst from last attempt, the only thing we changed is to decrease the LR.

Again, the improvements between epochs are too slow - the LR is too small.

QUESTION 4: : Suggest a way to improve the results by changing the networks's architecture

ANSWER:

It is possible to create more hidden layers.

The more hidden layers we add to the model, the deeper the model can learn, meaning it will learn more distinctive and unique features through each hidden layer.

Explainability

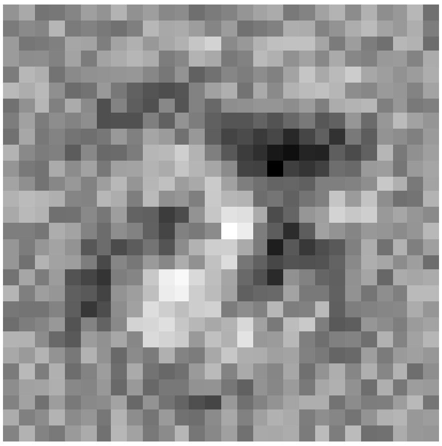
Here we will plot some of the network weights

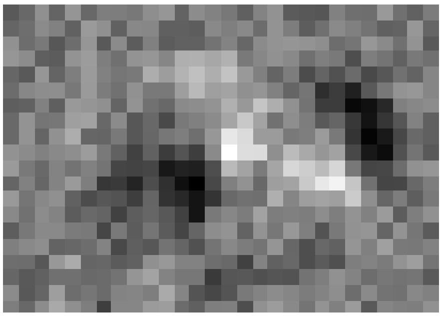
```
args['hidden_layer_dim'] = 100
args['lr'] = 0.1
```

```
best_model, cur_acc_loss = train(args, dataloaders)
    ========== new best model! epoch 11, accuracy 0.9  ==========

    Train epoch 12: accuracy 0.9112745098039214, loss 4.9618974479974485

    Validation epoch 12: accuracy 0.903, loss 4.977949937184651
    ========== new best model! epoch 12, accuracy 0.903  ==========

    Train epoch 13: accuracy 0.9126078431372548, loss 4.9514184091605395

    Validation epoch 13: accuracy 0.904, loss 4.968632062276204

    Train epoch 14: accuracy 0.9142941176470586, loss 4.942769031898648

    Validation epoch 14: accuracy 0.906, loss 4.960579130384657
    ========== new best model! epoch 14, accuracy 0.906  ==========

    Train epoch 15: accuracy 0.9164117647058824, loss 4.934740309621773

    Validation epoch 15: accuracy 0.908, loss 4.953455554114448
    ========== new best model! epoch 15, accuracy 0.908  ==========

    Train epoch 16: accuracy 0.9174509803921567, loss 4.927145836400051

    Validation epoch 16: accuracy 0.909, loss 4.946621470981174

    Train epoch 17: accuracy 0.9189803921568628, loss 4.920620562983494

    Validation epoch 17: accuracy 0.911, loss 4.940295908186171
    ========== new best model! epoch 17, accuracy 0.911  ==========

    Train epoch 18: accuracy 0.9204901960784312, loss 4.914412002937467

    Validation epoch 18: accuracy 0.913, loss 4.934867223103841
    ========== new best model! epoch 18, accuracy 0.913  ==========

    Train epoch 19: accuracy 0.9216470588235293, loss 4.909113753075693

    Validation epoch 19: accuracy 0.914, loss 4.929328441619873

    Train epoch 20: accuracy 0.9226274509803922, loss 4.904416757471421

    Validation epoch 20: accuracy 0.915, loss 4.924436622195774
    ========== new best model! epoch 20, accuracy 0.915  ==========

    Train epoch 21: accuracy 0.9236078431372549, loss 4.898767303018009

    Validation epoch 21: accuracy 0.916, loss 4.919769922892253

    Train epoch 22: accuracy 0.924549019607843, loss 4.893899337918151

    Validation epoch 22: accuracy 0.918, loss 4.915563901265462
    ========== new best model! epoch 22, accuracy 0.918  ==========

    Train epoch 23: accuracy 0.9260588235294117, loss 4.889362232357848

    Validation epoch 23: accuracy 0.919, loss 4.911419815487331

    Train epoch 24: accuracy 0.9268627450980392, loss 4.885538652831433
```

```
# Visualize some weights. features of digits should be somehow present.
def show_net_weights(net_params):
    W1 = net_params.fc1.weight.cpu().data.numpy().T
    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(best_model)
```

```
(784, 100)
```



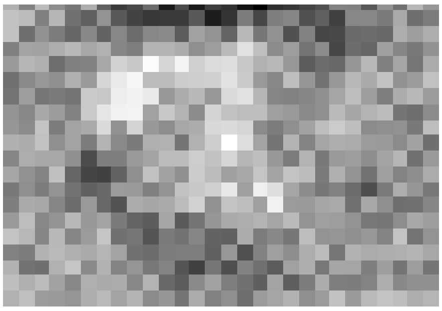


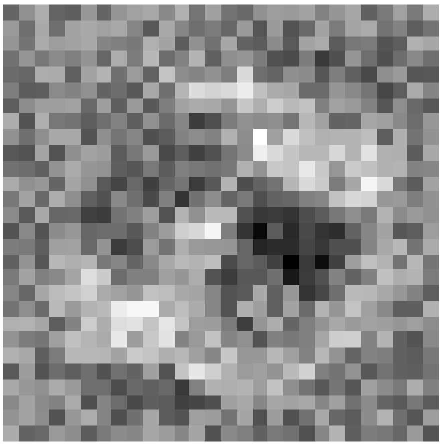
QUESTION 5: Where are the bright regions? why?

ANSWER:

the brightest region can be found in the middle.

The reason is that the most of the features can be found in the center area of any photo of digit.





✓ 0s completed at 5:49 PM