In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
!pip install shutup
```

In [ ]:
```
import torch
import torchvision
import torch.nn as nn
import torchvision.transforms as transforms
import torchvision.datasets as dsets
from torchvision.datasets import MNIST
import matplotlib.pylab as plt
import numpy as np
from torch.optim import Adam, SGD
import random
import math
import random
from random import choice
torch.manual_seed(17)
import os
from statistics import mode
from scipy import stats
import pickle
import cv2
import sklearn
from sklearn.model_selection import train_test_split
import time
import pandas as pd
import matplotlib
import shutup; shutup.please()
from torchsummary import summary
```

In [ ]:
```
# def show_data(data_sample):
#   plt.imshow(data_sample[0].reshape(32,32), cmap='gray')
#   diff = np.max(np.absolute(np.subtract(data_sample[0],data_sample[3]))
#   plt.title('y_true = '+ str(data_sample[1]) + ' y_pred = '+str(data_sa

def show_data_2(data_sample, pred_label):
  plt.figure()
  plt.imshow(data_sample.reshape(28,28), cmap='gray')
  plt.title('y_pred = '+str(pred_label))
  plt.show()
```

In [ ]:
```
x_train = pd.read_csv('/content/drive/MyDrive/Colab Notebooks/ECE657A/A4/
x_test = pd.read_csv('/content/drive/MyDrive/Colab Notebooks/ECE657A/A4/x
y_train = pd.read_csv('/content/drive/MyDrive/Colab Notebooks/ECE657A/A4/
y_test = pd.read_csv('/content/drive/MyDrive/Colab Notebooks/ECE657A/A4/y
```

In [ ]:
```
x_train_data = x_train.values
x_test_data = x_test.values
y_train_data = y_train.values
y_test_data = y_test.values
```

In [ ]:
```
x_train_data, x_val_data, y_train_data, y_val_data = train_test_split(x_t
```

In [ ]:
```
def build_dataloader(data=x_train_data,label = y_train_data, batch_size =
    ip = []
    lab = []
    tmp_ip = []
    tmp_lab = []
    count = 0
    for t,l in zip(data,label):
        tmp_ip.append(cv2.resize(t,(32,32)).reshape(1,32,32))
        tmp_lab.append(int(l))
        count+=1
        if count%batch_size == 0:
            ip.append(torch.tensor(tmp_ip))
            lab.append(torch.tensor(np.array(tmp_lab)))
            tmp_ip = []
            tmp_lab = []
    return (ip,lab)

def preprocess_ip(x):
  return torch.tensor(cv2.resize(x,(32,32)).reshape(1,1,32,32))
```

In [ ]:
```
loader= build_dataloader()
val_loader = build_dataloader(data = x_val_data, label = y_val_data )
test_loader = build_dataloader(data = x_test_data, label = y_test_data)
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:12: UserWarn
ing: Creating a tensor from a list of numpy.ndarrays is extremely slow. P
lease consider converting the list to a single numpy.ndarray with numpy.a
rray() before converting to a tensor. (Triggered internally at  ../torch/
csrc/utils/tensor_new.cpp:201.)
  if sys.path[0] == '':
```

In [ ]:
```
class CNN(nn.Module):
  def __init__(self):
    super(CNN, self).__init__()
    self.cnn1 = nn.Conv2d(in_channels = 1, out_channels = 32, kernel_size
    self.relu1 = nn.ReLU()
    self.maxpool1 = nn.MaxPool2d(kernel_size=2)
    self.cnn2 = nn.Conv2d(in_channels = 32, out_channels = 32, kernel_siz
    self.fc1 = nn.Linear(2048,5)
    self.softmax = nn.Softmax()

  def forward(self,x):
    out = self.cnn1(x)
    out = self.relu1(out)
    out = self.maxpool1(out)
    out = self.cnn2(out)
    out = self.relu1(out)
    out = self.maxpool1(out)
    out = out.view(out.size(0), -1)
    out = self.fc1(out)
    out = self.softmax(out)
    return out

  def train(self, train_loader,opt, loss_fn, BATCH_SIZE, eval = True):
    loss_tot = 0
    acc = 0
```

```
    cnt = 0
    for x,y in zip(train_loader[0], train_loader[1]):
      cnt+=1
      x = x.float()
      x = x.cuda()
      y = y.cuda()
      opt.zero_grad()
      z = self.forward(x)
      y = torch.nn.functional.one_hot(y, num_classes=5)
      loss = loss_fn(z.double(),y.double())
      loss_tot+= loss.item()
      loss.backward()
      opt.step()
      if cnt%1000==0:
        print("Loss is ",loss.item())
    loss_tr = loss_tot/(len(train_loader[1]))
    if eval:
      correct = 0
      for x,y in zip(train_loader[0],train_loader[1]):
        x = x.float()
        x = x.cuda()
        y = y.cuda()
        z = self.forward(x)
        _,yhat = torch.max(z.data,1)
        correct += (yhat== y).sum().item()
      acc = correct/(len(train_loader[1])*BATCH_SIZE)
    return loss_tr,acc

  def eval(self,loader,BATCH_SIZE,loss_fn=nn.CrossEntropyLoss()):
    correct = 0
    acc = 0
    loss_tot = 0
    for x,y in zip(loader[0],loader[1]):
      x = x.float()
      x = x.cuda()
      y = y.cuda()
      z = self.forward(x)
      loss = loss_fn(z.double(),y.long())
      loss_tot += loss.item()
      _ , yhat = torch.max(z.data,1)
      correct += (yhat == y).sum().item()
    acc = correct/(len(loader[1]))
    loss_val = loss_tot/(len(loader[1])*BATCH_SIZE)
    return loss_val, acc
```

[CM1] Classification with CNN

Default Network

Model 1 - summary

In [ ]:
```
model = CNN()
model = model.cuda()
summary(model, (1,32,32))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 32, 32]             320
              ReLU-2           [-1, 32, 32, 32]               0
         MaxPool2d-3           [-1, 32, 16, 16]               0
            Conv2d-4           [-1, 32, 16, 16]           9,248
              ReLU-5           [-1, 32, 16, 16]               0
         MaxPool2d-6             [-1, 32, 8, 8]               0
            Linear-7                    [-1, 5]          10,245
           Softmax-8                    [-1, 5]               0
================================================================
Total params: 19,813
Trainable params: 19,813
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.70
Params size (MB): 0.08
Estimated Total Size (MB): 0.78
----------------------------------------------------------------
```

Network Architecture:
We add convolutional layers and flatten the final result to feed into the densely connected layers.Finally we add the densely connected layers. We configure the network as per the given network specification.

Activation Functions:

1. **Relu**
   effectively means "If X>0 return X, else return 0" -- so what it does is it only passes values0 or greater to the next layer in the network. ReLU stands for Rectifi ed Linear Unit. The mainadvantage of using the ReLU function over other activation functions is that it does notactivate all the neurons at the same time. This means that the neurons will only be deactivatedif the output of the linear transformation is less than 0. For the negative input values, the resultis zero, that means the neuron does not get activated. Since only a certain number of neuronsare activated, the ReLU function is far more computationally efficient when compared to the sigmoid and tanh function. Due to this reason, during the backpropogation process, the weights and biases for some neurons are not updated.
2. **Softmax**
   Since our problem is a classification problem, we employ softmax as the activation function of last layer.

Optimizer: We have used "adam" optimizer while compiling the model because Adam combines thebest properties of the AdaGrad and RMSProp algorithms to provide an optimization algorithm thatcan handle sparse gradients on noisy problems. Adam is relatively easy to confi gure where thedefault confi guration parameters do well on most problems.

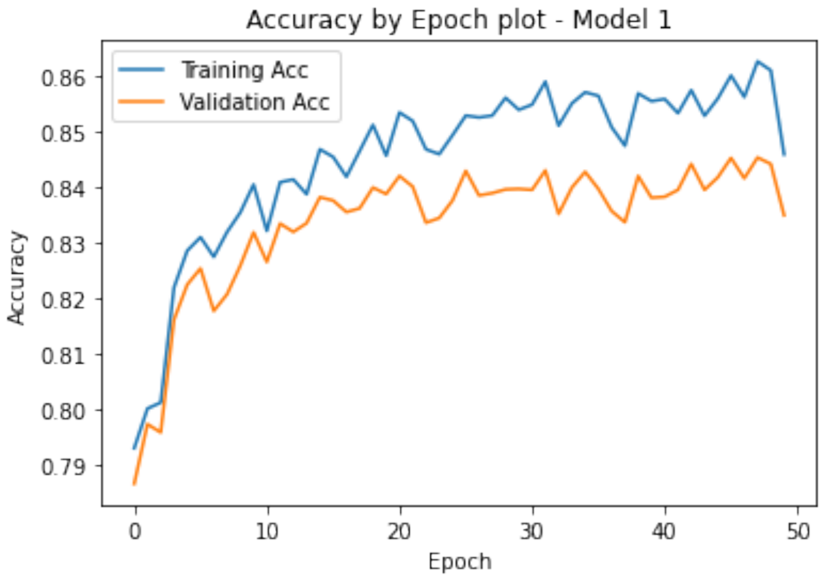
Loss Function: Since our problem is a classification problem, we use CrossEntropyLoss.

In [ ]:
```
opt = Adam(model.parameters(), lr = 0.001)
crit = nn.CrossEntropyLoss()
model = model.cuda()

tr_loss = []
tr_acc = []
val_acc = []
val_loss = []
training_time_tot = 0
for epoch in range(50):
  start_time = time.time()
  l, a = model.train(train_loader=loader,opt = opt, loss_fn = crit, BATCH_
  end_time = time.time()
  training_time_tot += end_time - start_time
  print("Epoch ",epoch+1,"--> ",a)
  tr_loss.append(l)
  tr_acc.append(a)
  l,a = model.eval(loader=val_loader,BATCH_SIZE=100)
  val_acc.append(a)
  val_loss.append(l)
```

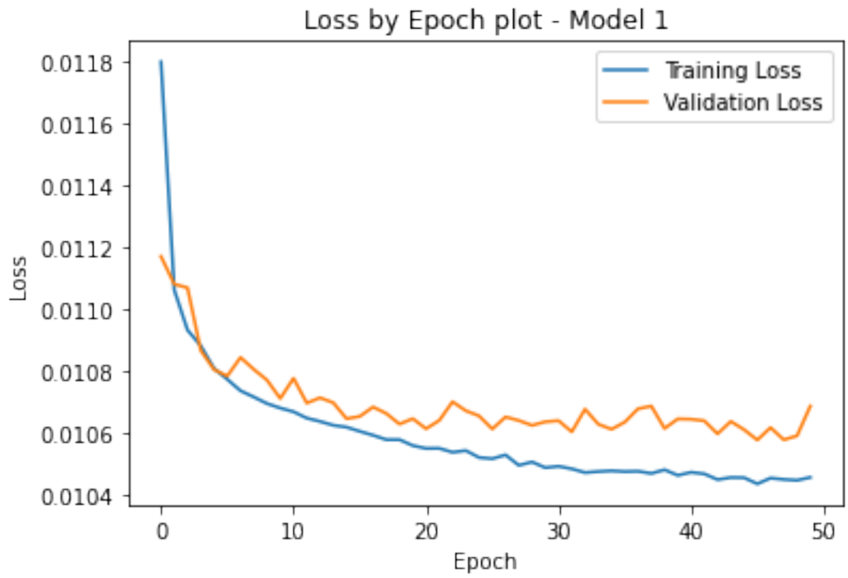
```
Epoch  1 -->  0.793
Epoch  2 -->  0.8001458333333333
Epoch  3 -->  0.8012291666666667
Epoch  4 -->  0.8219791666666667
Epoch  5 -->  0.828625
Epoch  6 -->  0.8310416666666667
Epoch  7 -->  0.8275
Epoch  8 -->  0.8319791666666667
Epoch  9 -->  0.8355
Epoch  10 -->  0.8405416666666666
Epoch  11 -->  0.8321875
Epoch  12 -->  0.8409583333333334
Epoch  13 -->  0.8414583333333333
Epoch  14 -->  0.8387916666666667
Epoch  15 -->  0.8468958333333333
Epoch  16 -->  0.8455208333333334
Epoch  17 -->  0.8419375
Epoch  18 -->  0.8466041666666667
Epoch  19 -->  0.8512916666666667
Epoch  20 -->  0.84575
Epoch  21 -->  0.8535416666666666
Epoch  22 -->  0.852
Epoch  23 -->  0.8468958333333333
Epoch  24 -->  0.8460208333333333
Epoch  25 -->  0.8494375
Epoch  26 -->  0.8529791666666666
Epoch  27 -->  0.8526458333333333
Epoch  28 -->  0.8529791666666666
Epoch  29 -->  0.8561875
Epoch  30 -->  0.8540208333333333
Epoch  31 -->  0.8549583333333334
Epoch  32 -->  0.8591041666666667
Epoch  33 -->  0.8511666666666666
Epoch  34 -->  0.8551666666666666
Epoch  35 -->  0.8571875
Epoch  36 -->  0.8565416666666666
Epoch  37 -->  0.850875
Epoch  38 -->  0.8475416666666666
Epoch  39 -->  0.8569791666666666
Epoch  40 -->  0.8555833333333334
Epoch  41 -->  0.8559583333333334
Epoch  42 -->  0.8534166666666667
Epoch  43 -->  0.8575625
Epoch  44 -->  0.8529375
Epoch  45 -->  0.8560416666666667
Epoch  46 -->  0.8602083333333334
Epoch  47 -->  0.8563333333333333
Epoch  48 -->  0.86275
Epoch  49 -->  0.861125
Epoch  50 -->  0.8459791666666666
```

In [ ]:
```
plt.plot(tr_acc, label = 'Training Acc')
plt.plot(np.array(val_acc)/100.0, label = 'Validation Acc')
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.title("Accuracy by Epoch plot - Model 1")
plt.legend()
plt.show()
```



In [ ]:
```
plt.plot(np.array(tr_loss)/100, label = 'Training Loss')
plt.plot(np.array(val_loss), label = 'Validation Loss')
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Loss by Epoch plot - Model 1")
plt.legend()
```

Out[ ]:
```
<matplotlib.legend.Legend at 0x7ff48b218d90>
```



In [ ]:
```
test_loss,test_acc = model.eval(loader=test_loader,BATCH_SIZE=100)
print("Test loss for Model 1 is ", test_loss)
print("Test accuracy for Model 1 is ", test_acc)
```

```
Test loss for Model 1 is  0.010652118769183854
Test accuracy for Model 1 is  83.85
```

Model 1 Performance:

Based on training and validation accuracy, we can see that there is no overfitting that is affecting the model as validation accuracy closely follows training accuracy. We conclude training of the model after 50 epochs. Test accuracy for this model after training for 50 epochs is 83.85%. The factors upon which the accuracy depends includes a lot of factors outside of network architecture. Some of those factors are learning rate, number of epochs, network initialization etc.