Question 2

The question asks us to capture images of alphabets, then binarize them and resize them to 9x7. This was done for the alphabets 'A', 'B', 'C', 'D', 'E', and 'F'. This was achieved using the OpenCV library available with Python. Functions used:

- cv2.imread(): Reads image into a matrix
- cv2.cvtColor(): Converts to the necessary colour format. In this question, we convert the original image to grayscale. In a gray scale image the values of the pixels lie between 0 and 255. 0 corresponds to black and 255 corresponds to white.
- cv2.threshold(): Converts the grayscale image to a black and white image. This functions performs binarization. We have used a threshold of 100 to obtain the binarized image. If the pixel value is greater than the threshold, then that pixel is assigned a value of 255 i.e white, otherwise the pixel is assigned black or value 0.
- cv2.resize(): Resizes the binarized image to the required dimensions. This functions also performs interpolation to find out the values of pixels in the resized image. We have used the "Inter Area Interpolation" which is recommended for shrinking the image.

After the images are resized, the image in displayed and its pixel values are printed as a matrix with dimensions 9x7. Various observations were made.

```
import cv2
import os
width = 7
height = 9
dim = (width, height)
list_alpha = ['A','B','C','D','E','F']
for alpha in list_alpha:
    img = cv2.imread('F:\\notes\\8th sem\\Machine Learning-Jupyter
Codes\\images\\'+alpha+'_1.jpg')
    gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
    (thresh, bw) = cv2.threshold(gray, 100, 255, cv2.THRESH_BINARY)
    img = cv2.resize(bw, dim,interpolation = cv2.INTER_AREA)
    cv2.imshow('',img)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
    print('alphabet '+ alpha+' after resizing\n')
    print(img)
    print('\n')
```

```
alphabet A after resizing

[[255 255 255 255 255 255 255]
 [255 255 255 246 255 255 255]
```

```
 [255 255 253 247 254 255 255]
 [255 255 250 255 250 255 255]
 [255 255 245 249 246 255 255]
 [255 252 252 255 250 254 255]
 [255 249 255 255 255 249 255]
 [255 249 255 255 255 252 255]
 [255 255 255 255 255 255 255]]


alphabet B after resizing

[[255 255 255 255 255 255 255]
 [255 250 251 250 253 255 255]
 [255 250 255 255 250 255 255]
 [255 250 255 253 251 255 255]
 [255 247 246 246 253 255 255]
 [255 249 255 255 249 255 255]
 [255 248 255 255 250 255 255]
 [255 249 245 249 252 255 255]
 [255 255 255 255 255 255 255]]


alphabet C after resizing

[[255 255 255 252 250 254 255]
 [255 253 249 253 255 253 255]
 [255 249 255 255 255 255 255]
 [255 249 255 255 255 255 255]
 [252 252 255 255 255 255 255]
 [255 249 255 255 255 255 255]
 [255 248 255 255 255 255 255]
 [255 252 248 255 255 255 255]
 [255 255 255 249 249 252 255]]


alphabet D after resizing

[[255 250 250 250 252 255 255]
 [255 249 255 255 252 252 255]
 [255 249 255 255 255 250 255]
 [255 249 255 255 255 253 253]
 [255 249 255 255 255 255 250]
 [255 249 255 255 255 252 254]
 [255 249 255 255 255 251 255]
 [255 248 255 254 249 253 255]
 [255 247 249 250 255 255 255]]


alphabet E after resizing

[[255 255 255 255 255 254 255]
```

```
 [255 248 251 251 252 254 255]
 [255 249 255 255 255 255 255]
 [255 249 255 255 255 255 255]
 [255 247 251 253 255 255 255]
 [255 249 255 255 255 255 255]
 [255 249 255 255 255 255 255]
 [255 255 249 255 255 255 255]
 [255 255 248 250 250 252 255]]


alphabet F after resizing

[[255 255 255 255 255 255 255]
 [255 251 251 250 251 252 255]
 [255 251 253 255 255 255 255]
 [255 252 252 255 255 255 255]
 [255 254 244 255 255 255 255]
 [255 255 249 255 255 255 255]
 [255 255 249 255 255 255 255]
 [255 255 248 255 255 255 255]
 [255 255 253 255 255 255 255]]
```

Observations

- After resizing, the image was printed as a matrix of dimension 9x7.
- The values of pixels in each image ranges from 240 to 255. To the naked eye, the image will appear to be entirely white.
- The image after resizing is not binary because the cv2.resize() function performs interpolation. If we don't give any interpolation input to the cv2.resize() function, all the pixels become white.
- This method cannot be used to generate images of alphabets having size 9x7. We will have to use bigger images.

Question 3

Implementation of the single layer perceptron model which is trained on an alphabet dataset

1. PyTorch Deep Learning library is used for training the model.
2. Extended-MNIST dataset is used for training the model. This is character dataset which consists of about 125,000 characters of almost equally distributed 26 alphabets and about 40,000 characters for testing.
3. It is an open source dataset which is available on Kaggle.

Why use the the EMNIST dataset?

1. As observed in Question 2, we cannot get a `9*7` image by resizing an image captured using a phone.
2. Since we don't have access to the dataset created by other students, the data we have is very less which will result in a poor performance of the model.

```
import torch
from torch.utils.data.dataset import Dataset
import numpy as np
from torch.utils.data import DataLoader
import os
import random

import random
import scipy

class EMNIST_val(Dataset):
    def __init__(self, images_tr, labels_tr):
        self.images_tr = images_tr
        self.labels_tr = labels_tr
    def __len__(self):
        return len(self.labels_tr)
    def one_hot(self,gt):
        oh = np.zeros(26)
        oh[gt-1] = 1
        return oh
    def __getitem__(self, index):
        image = self.images_tr[index,:]
        gt = self.labels_tr[index]
        gt = self.one_hot(gt)
        sample = {'image': image, 'gt' : gt}
        return sample
```

```
import torch
from torch.utils.data.dataset import Dataset
import numpy as np
from torch.utils.data import DataLoader
import os
import random

import random
import scipy

class EMNIST(Dataset):
    def __init__(self, images_tr, labels_tr):
        self.images_tr = images_tr
        self.labels_tr = labels_tr
    def __len__(self):
        return len(self.labels_tr)
    def one_hot(self,gt):
        oh = np.zeros(26)
        oh[gt-1] = 1
        return oh
    def __getitem__(self, index):
        image = self.images_tr[index,:]
        gt = self.labels_tr[index]
```

```
        gt = self.one_hot(gt)
        sample = {'image': image, 'gt' : gt}
        return sample
```

```
import numpy as np
from emnist import *
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
import torch.optim as optim
images_tr, labels_tr = extract_training_samples('letters')
images_ts, labels_ts = extract_test_samples('letters')

loss_fn = torch.nn.MSELoss()


(n,h,w) = images_tr.shape
images_tr = images_tr.reshape(n,h*w)
images_val = images_tr[99840:,:]
labels_val = labels_tr[99840:]
num_class = 26
```

Given below is the architecture of the single layer perceptron neural network

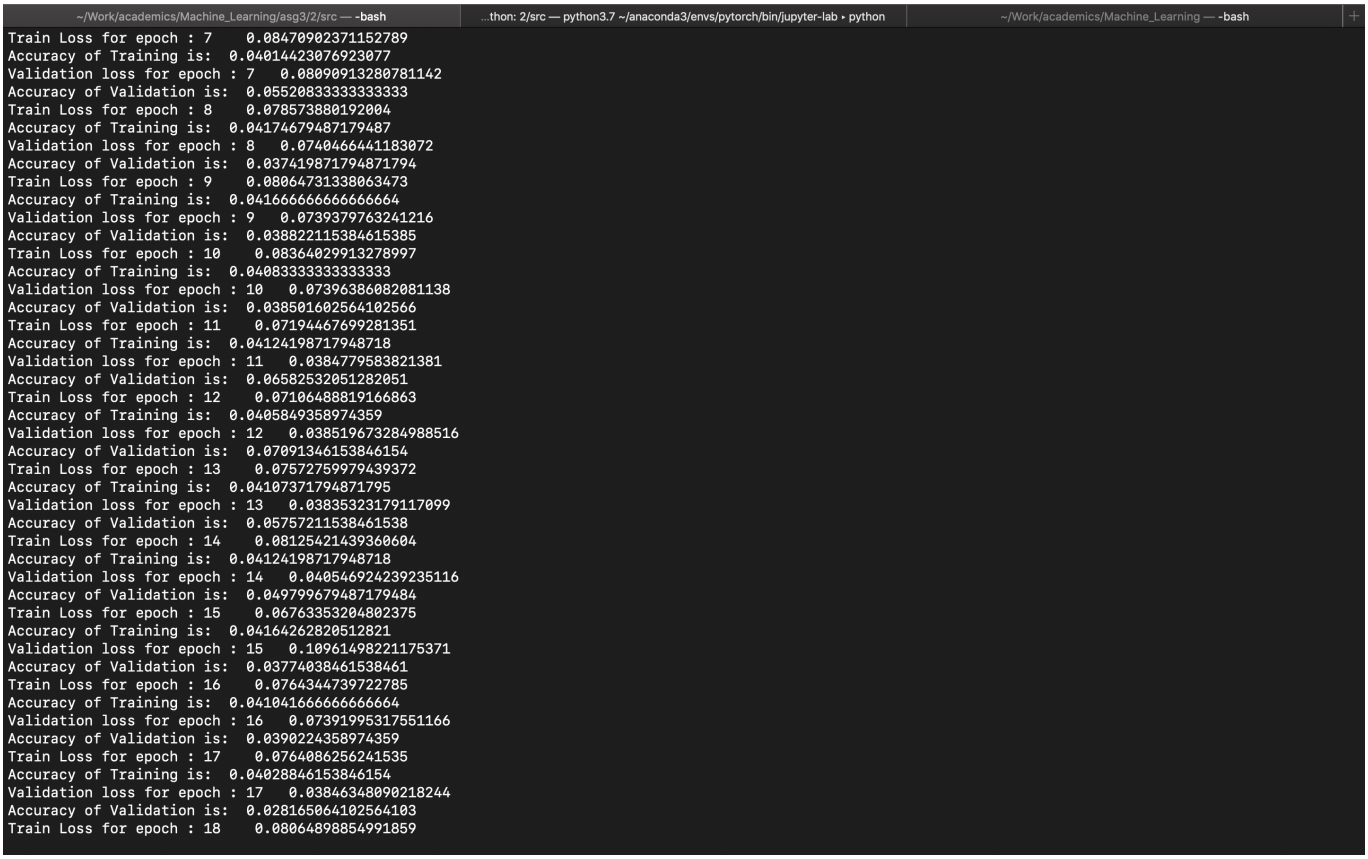
1. Each of the images of the dataset are of the size of `28 * 28` pixels.
2. These images are flattened to a `(784,1)` vector since we have to feed our image to a single layer linear perceptron.
3. The number of outputs after the perceptron is 26 since we have 26 classes, each for 1 alphabet.
4. The ground truth is one-hot encoded as a `(26,1)` vector.
5. Sigmoid activation function is used
6. Mean squared error loss is used to train the model
7. We use the adam optimizer to update the weights of the network
8. Training and validation accuracy is printed at the end of each epoch. The model is trained for a total of 100 epochs.

```
# define neural network
class olp(nn.Module):
    def __init__(self, input_neurons, output_neurons):
        nn.Module.__init__(self)
        self.fc = nn.Linear(input_neurons, output_neurons)
    def forward(self,x):
        x = self.fc(x)
        x = F.sigmoid(x)
        return x
```

```
model = olp(h*w,num_class)
optimizer = optim.Adam(model.parameters(), lr = 0.01, betas = (0.9,0.999),
weight_decay = 0.00005)
model.cpu()
num_epochs = 100
dataset_train = EMNIST(images_tr, labels_tr)
train_loader = DataLoader(dataset_train,batch_size= 1,shuffle=True, num_workers=4)
dataset_valid = EMNIST_val(images_val, labels_val)
val_loader = DataLoader(dataset_valid, batch_size = 1, shuffle = True, num_workers
= 4)
acc_tmp = 0
for ep in range(num_epochs):
    # Setting the model to train mode
    loss_agg = 0
    acc = 0
    model.train
    for batch_idx, (subject) in enumerate(train_loader):
        # Load the subject and its ground truth
        image = subject['image']
        mask = subject['gt']
        image.cpu()
        mask.cpu()
        optimizer.zero_grad()
        # Forward Propagation to get the output from the models
        output = model(image.float())
        # Computing the loss
        loss = loss_fn(output.double(), mask.double())
        # Back Propagation for model to learn
        loss.backward()
        #Updating the weight values
        optimizer.step()
        loss = loss.detach().cpu().numpy()
        loss_agg = loss_agg + loss
        output = output.detach().cpu().numpy()
        mask = mask.detach().cpu().numpy()
        tmp = np.zeros(output.shape)
        tmp[:,np.argmax(output)] = 1
        acc = acc + np.sum(tmp*mask)
    print("Train Loss for epoch :",ep,"  ",loss_agg/(batch_idx+1))
    print("Accuracy of Training is: ", acc/(batch_idx+1))
    model.eval
    loss_agg_val = 0
    acc = 0
    for batch_idx, (subject) in enumerate(val_loader):
        with torch.no_grad():
            image = subject['image']
            mask = subject['gt']
            output = model(image.float())
            loss = loss_fn(output.double(), mask.double())
            loss = loss.detach().cpu().numpy()
            loss_agg_val = loss_agg_val + loss
```

```
            output = output.detach().cpu().numpy()
            mask = mask.detach().cpu().numpy()
            tmp = np.zeros(output.shape)
            tmp[:,np.argmax(output)] = 1
            acc = acc + np.sum(tmp*mask)

    print("Validation loss for epoch :",ep," ",loss_agg_val/(batch_idx+1))
    print("Accuracy of Validation is: ", acc/(batch_idx+1))
    if acc>acc_tmp:
        acc_tmp = acc
        torch.save(model,"mod.pt")
```



Results of training

1. The training and validation accuracy are very low, at about 5 % when trained for 100 epochs
2. This tells us that the data can not be classfied using just a single layer perceptron.
3. This also tells us that the data is not linearly separable.
4. Hence, we need atleast 2 layer perceptron network to classify the alphabets