ASL_Project

May 11, 2022

1 Computer Vision for reading American Sign Language (ASL)

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

1.1 Importing libraries

[ ]:
```
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import torch
from tqdm import tqdm
import torch.nn as nn
from torch import optim
import torch.nn.functional as F
import torchvision.datasets as datasets
from torchvision.io import read_image
from torchvision import transforms
from torchvision import models
from torch.utils.data import Dataset, DataLoader,␣
 ↪SubsetRandomSampler,random_split

import glob
from mpl_toolkits.axes_grid1 import ImageGrid
from skimage.transform import resize
```

[ ]:
```
train_path='/content/drive/MyDrive/ASL_Project/dataset/asl_alphabet_train/
 ↪asl_alphabet_train'
test_path='/content/drive/MyDrive/ASL_Project/dataset/ASL_TEST/
 ↪asl_alphabet_test'
my_test_path='/content/drive/MyDrive/ASL_Project/My_Test_ASL'
```

1.1.1 Viewing the images from the training set

[ ]:
```
train_folders=np.array(glob.glob('/content/drive/MyDrive/ASL_Project/dataset/
 ↪asl_alphabet_train/asl_alphabet_train/*'))

fig = plt.figure(figsize=(30, 30))
grid = ImageGrid(fig, 111,
                 nrows_ncols=(8, 8),
                 axes_pad=0,
                 )
l=0
for img in train_folders:
        it=np.array(glob.glob(img+'/*'))
        im=plt.imread(it[0])
        class_label=it[0].split('/')[8]
        grid[l].imshow(im,cmap='gray',interpolation='nearest')
        grid[l].text(10,25, class_label,fontsize=40,color='red')
        l+=1
```

1.1.2 Viewing the images from self-generated test data

[ ]:
```
my_test_folders=np.array(glob.glob('/content/drive/MyDrive/ASL_Project/
 ↪My_Test_ASL/*'))

fig = plt.figure(figsize=(30, 30))
grid = ImageGrid(fig, 111,
                 nrows_ncols=(8, 8),
                 axes_pad=0,
                 )
l=0
for img in my_test_folders:
        it=np.array(glob.glob(img+'/*'))
        image=plt.imread(it[0])
        class_label=it[0].split('/')[6]
        image2=resize(image, (200, 200))
        grid[l].imshow(image2,cmap='gray',interpolation='nearest')
        grid[l].text(10,25, class_label,fontsize=40,color='red')
        l+=1
```

1.2 Transformations

Transformations on the Train dataset

[ ]:
```
train_transforms = transforms.Compose([transforms.Resize((96,96)),
                                       transforms.RandomHorizontalFlip(p=0.3),
                                       transforms.ToTensor()])
```

Transformations on the Test dataset

[ ]:
```
test_transforms = transforms.Compose([transforms.Resize((96,96)),
                                       transforms.ToTensor()])
```

[ ]:
```
# performing resize and horizontal flip transformations on the train data set

data= datasets.ImageFolder(train_path, transform=train_transforms)
```

[ ]:
```
# performing resize transformations on the two test data sets

test_data = datasets.ImageFolder(test_path, transform=test_transforms)
my_test_data=datasets.ImageFolder(my_test_path, transform=test_transforms)
```

1.2.1 Mapping of labels

[ ]:
```
class_labels=data.class_to_idx
class_labels = {value:key for key, value in class_labels.items()}
print(class_labels)
print(f"Class to index mapping: {data.class_to_idx}")
```

1.3 Splitting of train and validation data set

[ ]:
```
train_data_,val_data_= random_split(data, (69600, 17400),
                                    generator=torch.Generator().manual_seed(25))
#splits the dataset randomly
```

[ ]:
```
print('Total images in Train data set:', len(train_data_))
print('Total images in Validation data set:', len(val_data_))
print('Total images in Test data set:', len(test_data))
```

1.4 Loading images to data loaders

[ ]:
```
train_loader = torch.utils.data.DataLoader(train_data_, batch_size=128,␣
 ↪shuffle=True)
val_loader = torch.utils.data.DataLoader(val_data_, batch_size=128,␣
 ↪shuffle=True)
test_loader = torch.utils.data.DataLoader(test_data)
my_test_loader = torch.utils.data.DataLoader(my_test_data)
```

1.5 Checking if GPU is available

[ ]:
```
# use GPU to accelerate the training if available
device = "cuda" if torch.cuda.is_available() else "cpu"
print("Currently used device:", device)
```

1.6 Common functions for generating plots

[ ]:
```
def plot_accuracy(train_accuracy,validation_accuracy):
    '''
    Function to plot the accuracy for Training and Validation sets

    Parameters:
    train_accuracy: list of accuracy for training set
    validation_accuracy:list of accuracy for validation set


    Return:
    Plot of Train and Validation Accuracy

    '''
     # accuracy plots
    plt.figure(figsize=(8,8))
    plt.plot(train_accuracy, color='green', label='Train Accuracy')# plot train␣
 ↪accuracy
    plt.plot(validation_accuracy, color='orange', label='Validataion␣
 ↪Accuracy')# plot validation accuracy
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.show()
```

[ ]:
```
def plot_loss(train_loss,validation_loss):
    '''
    Function to plot the accuracy for Training and Validation sets

    Parameters:
    train_loss: list of loss for training set
    validation_loss:list of loss for validation set


    Return:
    Plot of Train and Validation Loss Curves

    '''
    plt.figure(figsize=(8,8))
    plt.plot(train_loss, color='blue', label='Train Loss')# plot train loss
    plt.plot(validation_loss, color='red', label='Validataion Loss')# plot␣
 ↪validation loss
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.show()
```

1.7 Common function for performing training on the model

```
[ ]: def train_model(model, dataloader,optimizer,criterion):
        '''
        Function to train the model on training set of images

        Parameters:
        model: the trained model
        dataloader: validation dataset
        optimizer: optimizer with a learning rate
        criterion: loss criterion

        Return:
        train_loss: train loss
        train_accuracy: train accuracy

        '''
        print('Training the model')
        model.train()
        batch_loss, batch_accuracy = [], []

        for k,(data,target) in tqdm(enumerate(dataloader),␣
     ↪total=int(len(train_loader)/train_loader.batch_size)):
            data, target = data.to(device), target.to(device)

            outputs = model(data)

            loss = criterion(outputs, target)
            acc = (outputs.argmax(1) == target).type(torch.float)
            acc = torch.mean(acc)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            # Append to lists
            batch_loss.append(loss.item())
            batch_accuracy.append(acc.item())

        train_loss = sum(batch_loss)/len(batch_loss)
        train_accuracy = sum(batch_accuracy)/len(batch_accuracy)

        print(f"Train Loss: {train_loss:.4f}, Train Accuracy: {train_accuracy:.2f}")

        return train_loss, train_accuracy
```

1.8 Common function for performing validation on the model

[ ]:
```
def validate_model(model, dataloader,criterion):
    '''
    Function to check performance on the validation dataset

    Parameters:
    model: the trained model
    dataloader: validation dataset
    criterion: loss criterion

    Return:
    val_loss: validation loss
    val_accuracy: validation accuracy

    '''

    print('Validating the model')
    model.eval()

    with torch.no_grad():
        for data,target in dataloader:
            data, target = data.to(device), target.to(device)
            outputs = model(data)
            val_loss = criterion(outputs, target)
            val_accuracy = (outputs.argmax(1) == target).type(torch.float).
 ↪mean()

        print(f'Validation Loss: {val_loss.item():.4f}, Validation Accuracy:␣
 ↪{val_accuracy.item():.2f}')

        return val_loss.item(), val_accuracy.item()
```

1.9 Common function for testing models

[ ]:
```
def test_model(model,test_loader,x):
    '''
    Function to test model's performance on the test set. The function receives␣
 ↪two types of test_loaders.
    One is the test set provided with the Kaggle dataset and the second one is␣
 ↪the dataset created by us.

    Parameters:
    model: the model to be used for testing
    test_loader: contains the test set images
    x: takes the value of 0 or 1. 0 indicates the test set sent to the function␣
 ↪was the Kaggle
```

```
    test data. 1 indicates the test set was the one created by us.

    Return:
    Prints accuracy on the test dataset

    '''
    true_labels=[] #correct labels
    pred_labels=[] #labels which would be predicted my model
    s=""

    with torch.no_grad():
        total = 0
        correct = 0
        for image, labels in test_loader:
            model.eval()
            image=image.to(device)
            labels=labels.to(device)
            output=model(image)

            predictions=torch.max(output,1)[1]
            correct+=(predictions==labels).sum().cpu().numpy()
            total+=len(labels)

            labels=labels.cpu().detach().numpy()
            predictions=predictions.cpu().detach().numpy()

            true_labels.extend(labels)
            pred_labels.extend(predictions)

        acc = correct*100/total
        if(x==0):
            s='Kaggle'
        else:
            s='self generated'
        print('Accuracy on '+s+' test dataset: %f' %(acc))
```

1.9.1 Common Optimizer and Criterion functions

[ ]:
```
def model_optimizer_criterion(model, learning_rate):
    '''
    Function to define the optimizer and criterion function for the model

    Parameters:
    learning_rate: learning rate for optimizer

    Return:
    optimizer: optimizer with a learning rate
```

```
    criterion: loss criterion
    '''
    # optimizer
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate,␣
 ↪weight_decay = 0.0001)
    # loss function
    criterion = nn.CrossEntropyLoss()

    return optimizer,criterion
```

1.10 Deep Neural Network

1.10.1 Defining the NN class

[ ]:
```
class NeuralNetwork(nn.Module):
    '''
    Defining a 4-layer Deep NN with Kaiming normal initialization and ReLU␣
 ↪activation functions
    '''

    def __init__(self):
        super(NeuralNetwork, self).__init__()

        self.flatten = nn.Flatten()
        #linear layers
        self.l1 = nn.Linear(3*96*96, 10000)# input image size is 96x96 and due␣
 ↪to RGB image there are 3 channels
        self.l2 = nn.Linear(10000, 2000)
        self.l3 = nn.Linear(2000, 250)
        self.l4 = nn.Linear(250, 29)

        nn.init.kaiming_normal_(self.l1.weight, mode='fan_in',
                                nonlinearity='relu')
        nn.init.kaiming_normal_(self.l2.weight, mode='fan_in',
                                nonlinearity='relu')
        nn.init.kaiming_normal_(self.l3.weight, mode='fan_in',
                                nonlinearity='relu')
        nn.init.kaiming_normal_(self.l4.weight, mode='fan_in',
                                nonlinearity='relu')

    def forward(self, x):
        #combining the entire model

        x = self.flatten(x)
        x = F.relu(self.l1(x))
        x = F.relu(self.l2(x))
        x = F.relu(self.l3(x))
```

```
        output = self.l4(x)

        return output
```

```
[ ]: nn_model = NeuralNetwork()
print('Layers of the updated pre-trained Neural Network model')
print(nn_model)

# trainable parameters
total_trainable_parameters = sum(para.numel() for para in nn_model.parameters()␣
 ↪if para.requires_grad)
print(f"{total_trainable_parameters:,} number of training parameters.")
```

```
[ ]: def neural_net_model():
    '''
    Function to train the model as well as check the performance on the␣
 ↪validation dataset. Performs early stopping when the
    validation loss exceeds the previous loss twice.
    '''
    num_epoch=40 #setting total number of epochs to be 40
    prev_loss=100 #initial loss value for early stopping
    max_penalty=3
    best_acc=0

    parameter_learning_rate=[0.001,0.0001,0.00001] # different values for␣
 ↪learning rate
    for i,learning_rate in enumerate(parameter_learning_rate): #Looping through␣
 ↪the learning rates
        train_loss , train_accuracy = [], []
        val_loss , val_accuracy = [], []
        counter=0
        model = NeuralNetwork().to(device)

        optimizer,criterion = model_optimizer_criterion(model, learning_rate)
        print("Training model on Learning Rate=",learning_rate)

        #model will train for a particular learning rate
        for epoch in range(num_epoch):# Looping through the epochs
            print(f"Epoch number {epoch+1} of {num_epoch}")

            #training the model
            train_epoch_loss, train_epoch_accuracy = train_model(model,
                                                                 train_loader,
                                                                 optimizer,
                                                                 criterion)
            #checking the performance on validation
            val_epoch_loss, val_epoch_accuracy = validate_model(model,
```

```
                                                            val_loader,
                                                            criterion)
            #append the loss and accuracy for every epoch for both train and␣
↪validation data set
            train_loss.append(train_epoch_loss)
            train_accuracy.append(train_epoch_accuracy)
            val_loss.append(val_epoch_loss)
            val_accuracy.append(val_epoch_accuracy)

            # Early Stopping
            if val_epoch_loss>prev_loss: # checking if validation loss for the␣
↪current epoch is greater than the previous loss
                counter+=1
                if counter>= max_penalty: # if the validation loss exceeds the␣
↪previous loss twice begin the early stopping process
                    print('Early Stopping Initiated')
                    break

            else:
                counter = 0

            prev_loss = val_epoch_loss # update the threshold loss to the␣
↪previous validation loss of the epoch

        #saving the best performing model
        if(val_epoch_accuracy>best_acc):
            best_acc=val_epoch_accuracy
            best_model=model

        #saving the model for a particular learning rate
        torch.save(model.state_dict(),'/content/drive/MyDrive/ASL_Project/
↪savedmodels/'+'model_neural_network'+str(learning_rate)+'.pth' )

        #plotting the accuracy and loss curves
        print('Plotting Validation and Training Accuracy Curves for Learning␣
↪Rate of ', learning_rate)
        plot_accuracy(train_accuracy,val_accuracy)
        print('Plotting Validation and Training Loss Curves for Learning Rate␣
↪of ', learning_rate)
        plot_loss(train_loss,val_loss)

    return best_model
```

1.10.2 Training the Neural Network model

[ ]:
```
trained_neural_net_model=neural_net_model()
```

1.10.3 Testing the Neural Network model

Testing the model on given test dataset

[ ]:
```
test_model(trained_neural_net_model,test_loader,0)
```

Testing the model on our test dataset

[ ]:
```
test_model(trained_neural_net_model,my_test_loader,1)
```

1.11 Convolutional Neural Network(CNN)

[ ]:
```
class CustomCNN(nn.Module):
    ''' Creating a CNN model from scratch'''

    def __init__(self):
        super(CustomCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 8, 5)
        self.conv2 = nn.Conv2d(8, 16, 5)
        self.conv3 = nn.Conv2d(16, 32, 3)
        self.conv4 = nn.Conv2d(32, 64, 5)
        self.fc1 = nn.Linear(64, 128)
        self.fc2 = nn.Linear(128, 29)#output 29 classes
        self.pool = nn.MaxPool2d(2, 2)
    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = self.pool(F.relu(self.conv4(x)))
        bs, _, _, _ = x.shape
        x = F.adaptive_avg_pool2d(x, 1).reshape(bs, -1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

[ ]:
```
model_cnn = CustomCNN().to(device)
print(model_cnn)
# total parameters and trainable parameters
total_params = sum(p.numel() for p in model_cnn.parameters())
print(f"{total_params:,} total parameters.")
total_trainable_params = sum(
    p.numel() for p in model_cnn.parameters() if p.requires_grad)
print(f"{total_trainable_params:,} training parameters.")
```

```
[ ]: def cnn_optimizer_criterion(model,learning_rate,wt_decay):
         '''
         Function to define the optimizer and criterion function for the CNN model

         Parameters:
         learning_rate: learning rate for optimizer

         Return:
         optimizer: optimizer with a learning rate
         criterion: loss criterion
         '''
         #optimizer
         optimizer = torch.optim.Adam(model.parameters(),␣
     ↪lr=learning_rate,weight_decay=wt_decay)
         # loss function
         criterion = nn.CrossEntropyLoss()
         return optimizer,criterion
```

```
[ ]: def my_cnn_model():
         '''
         Function to train the model as well as check the performance on the␣
     ↪validation dataset using CNN Architecture. Performs early stopping when the
         validation loss exceeds the previous loss thrice.
         '''
         num_epoch=40 #setting total number of epochs to be 40
         prev_loss=100 #initial loss value for early stopping
         max_penalty=3
         best_acc=0
         param_learning_rate=[0.1,0.01,0.001]
         param_weight_decay=[0.001,0.0001]
         # Hyper parameter tuning begins
         for i,learning_rate in enumerate(param_learning_rate):#Looping through␣
     ↪different learning rates values
             for j,wt_decay in enumerate(param_weight_decay):#Looping through␣
     ↪different weight decay values
                 model = CustomCNN().to(device)
                 ␣
     ↪optimizer,criterion=cnn_optimizer_criterion(model,learning_rate,wt_decay)
                 train_loss , train_accuracy = [], []
                 val_loss , val_accuracy = [], []
                 for epoch in range(num_epoch):
                     print(f"Epoch {epoch+1} of {num_epoch}")
                     #training the model
                     train_epoch_loss, train_epoch_accuracy = train_model(model,
                                                                          train_loader,
                                                                          optimizer,
                                                                          criterion)
```

```
                #checking the performance on validation
                val_epoch_loss, val_epoch_accuracy = validate_model(model,
                                                          val_loader,
                                                          criterion)
                # Early Stopping
                if val_epoch_loss>prev_loss: # checking if validation loss for␣
↪the current epoch is greater than the previous loss
                    counter+=1
                    if counter>= max_penalty: # if the validation loss exceeds␣
↪the previous loss twice begin the early stopping process
                        print('Early Stopping Initiated')
                        break

                else:
                    counter = 0

                prev_loss = val_epoch_loss # update the threshold loss to the␣
↪previous validation loss of the epoch

            #saving the best performing model
            if(val_epoch_accuracy>best_acc):
                best_acc=val_epoch_accuracy
                best_model=model

            #saving the model for a particular learning rate
            torch.save(model.state_dict(),'/content/drive/MyDrive/ASL_Project/
↪savedmodels/'+'model_cnn_lr'+str(learning_rate)+'_wt_decay_'+str(wt_decay)+'.
↪pth' )

            #plotting the accuracy and loss curves
            print('Plotting Validation and Training Accuracy Curves for␣
↪Learning Rate of ', learning_rate,' and Weight Decay of ',wt_decay)
            plot_accuracy(train_accuracy,val_accuracy)
            print('Plotting Validation and Training Loss Curves for Learning␣
↪Rate of ', learning_rate,' and Weight Decay of ',wt_decay)
            plot_loss(train_loss,val_loss)

    return best_model
```

1.11.1 Training the CNN model

[ ]:
```
trained_cnn_model=my_cnn_model()
```

1.11.2 Testing the CNN model

Testing the model on given test dataset

[ ]:
```
test_model(trained_cnn_model,test_loader,0)
```

Testing the model on our test dataset

[ ]:
```
test_model(trained_cnn_model,my_test_loader,1)
```

1.12 Resnet-50

1.12.1 Updating the fully connected layer of the pretrained Mobile Net V2 model

[ ]:
```
def Resnet_50():
    '''Using the concept of transfer learning on the pre-trained Resnet-50␣
 ↪model and retraining the fully connected layer '''

    model = models.resnet50(pretrained=True)

    for param in model.parameters():                  #freezing the pretrained␣
 ↪layers
        param.requires_grad = False

    model.fc = nn.Sequential(nn.Linear(2048, 1000),  #redefining the fully␣
 ↪connected classifier layer
                             nn.ReLU(),
                             nn.Dropout(0.3),
                             nn.Linear(1000, 500),
                             nn.ReLU(),
                             nn.Dropout(0.2),
                             nn.Linear(500, 29))
    return model
```

[ ]:
```
resnet_model = Resnet_50()
print('Layers of the updated pre-trained Resnet 50 model')
print(resnet_model)

# total parameters
total_parameters = sum(para.numel() for para in resnet_model.parameters())
print(f"{total_parameters:,} number of total parameters.")

# trainable parameters
total_trainable_parameters = sum(para.numel() for para in resnet_model.
 ↪parameters() if para.requires_grad)
print(f"{total_trainable_parameters:,} number of training parameters.")
```

[ ]:
```
def model_optimizer_criterion_res(model, learning_rate):
    '''
    Function to define the optimizer and criterion function for the ResNet model
```

```
    Parameters:
    learning_rate: learning rate for optimizer

    Return:
    optimizer: optimizer with a learning rate
    criterion: loss criterion
    '''
    # optimizer
    optimizer = torch.optim.Adam(model.fc.parameters(), lr=learning_rate,␣
↪weight_decay = 0.0001)
    # loss function
    criterion = nn.CrossEntropyLoss()

    return optimizer,criterion
```

[ ]:
```
def resnet_50_model():
    '''
    Function to train the model as well as check the performance on the␣
↪validation dataset. Performs early stopping when the
    validation loss exceeds the previous loss twice.
    '''

    num_epoch=40 #setting total number of epochs to be 40
    prev_loss=100 #initial loss value for early stopping
    max_penalty=3
    best_acc=0

    parameter_learning_rate=[0.1,0.01,0.001] # different values for learning␣
↪rate
    for i,learning_rate in enumerate(parameter_learning_rate): #Looping through␣
↪the learning rates
        train_loss , train_accuracy = [], []
        val_loss , val_accuracy = [], []
        counter=0
        model = Resnet_50().to(device)

        optimizer,criterion = model_optimizer_criterion_res(model,␣
↪learning_rate)
        print("Training model on Learning Rate=",learning_rate)

        #model will train for a particular learning rate
        for epoch in range(num_epoch):# Looping through the epochs
            print(f"Epoch number {epoch+1} of {num_epoch}")

            #training the model
            train_epoch_loss, train_epoch_accuracy = train_model(model,
                                                                 train_loader,
```

```
                                                           optimizer,
                                                           criterion)
          #checking the performance on validation
          val_epoch_loss, val_epoch_accuracy = validate_model(model,
                                                            val_loader,
                                                            criterion)
          #append the loss and accuracy for every epoch for both train and␣
↪validation data set
          train_loss.append(train_epoch_loss)
          train_accuracy.append(train_epoch_accuracy)
          val_loss.append(val_epoch_loss)
          val_accuracy.append(val_epoch_accuracy)

          # Early Stopping
          if val_epoch_loss>prev_loss: # checking if validation loss for the␣
↪current epoch is greater than the previous loss
              counter+=1
              if counter>= max_penalty: # if the validation loss exceeds the␣
↪previous loss twice begin the early stopping process
                  print('Early Stopping Initiated')
                  break

          else:
              counter = 0

          prev_loss = val_epoch_loss # update the threshold loss to the␣
↪previous validation loss of the epoch

      #saving the best performing model
      if(val_epoch_accuracy>best_acc):
          best_acc=val_epoch_accuracy
          best_model=model

      #saving the model for a particular learning rate
      torch.save(model.state_dict(),'/content/drive/MyDrive/ASL_Project/
↪savedmodels/'+'model_mobile_net'+str(learning_rate)+'.pth' )

      #plotting the accuracy and loss curves
      print('Plotting Validation and Training Accuracy Curves for Learning␣
↪Rate of ', learning_rate)
      plot_accuracy(train_accuracy,val_accuracy)
      print('Plotting Validation and Training Loss Curves for Learning Rate␣
↪of ', learning_rate)
      plot_loss(train_loss,val_loss)

  return best_model
```

1.12.2 Training the Resnet 50 model

[ ]:
```
trained_resnet_50_model = resnet_50_model()
```

1.12.3 Testing the Resnet 50 model

Testing the model on given test dataset

[ ]:
```
test_model(trained_resnet_50_model,test_loader,0)
```

Testing the model on our test dataset

[ ]:
```
test_model(trained_neural_net_model,my_test_loader,1)
```

1.13 Mobile-Net V2

1.13.1 Updating the pretrained Mobile Net V2 model

[ ]:
```
def ModelMob():
    '''Using the concept of transfer learning on the pre-trained Mobile Net V2␣
 ↪model and retraining the fully connected layer '''

    model = models.mobilenet_v2(pretrained=True)

    for param in model.parameters():                          #freezing the␣
 ↪pretrained layers
        param.requires_grad = False

    model.classifier = nn.Sequential(nn.Linear(1280, 128),    #redefining the␣
 ↪fully connected classifier layer
                             nn.ReLU(),
                             nn.Dropout(0.2),
                             nn.Linear(128, 64),
                             nn.ReLU(),
                             nn.Dropout(0.2),
                             nn.Linear(64, 29))
    return model
```

[ ]:
```
mob_model = ModelMob()
print('Layers of the updated pre-trained Mobile Net-V2 model')
print(mob_model)
# total parameters
total_parameters = sum(para.numel() for para in mob_model.parameters())
print(f"{total_parameters:,} number of total parameters.")

# trainable parameters
total_trainable_parameters = sum(para.numel() for para in mob_model.
 ↪parameters() if para.requires_grad)
```

```
print(f"{total_trainable_parameters:,} number of training parameters.")
```

1.13.2 Training the Mobile Net V2 model

```
[ ]: def mobile_net_model():
         '''
         Function to train the model as well as check the performance on the␣
      ↪validation dataset. Performs early stopping when the
         validation loss exceeds the previous loss twice.
         '''
         num_epoch=40 #setting total number of epochs to be 40
         prev_loss=100 # initial loss value for early stopping
         max_penalty=2
         best_acc=0

         parameter_learning_rate=[0.1,0.01,0.001]# different values for learning␣
      ↪rate
         for i,learning_rate in enumerate(parameter_learning_rate):#Looping through␣
      ↪the learning rates
             train_loss , train_accuracy = [], []
             val_loss , val_accuracy = [], []
             counter=0
             model = ModelMob().to(device)
             optimizer,criterion=model_optimizer_criterion(model, learning_rate)
             print("Training model on Learning Rate=",learning_rate)
             #model will train for a particular learning rate
             for epoch in range(num_epoch):# Looping through the epochs
                 print(f"Epoch number {epoch+1} of {num_epoch}")
                 #training the model
                 train_epoch_loss, train_epoch_accuracy = train_model(model,
                                                                  train_loader,
                                                                  optimizer,
                                                                  criterion)
                 #checking the performance on validation
                 val_epoch_loss, val_epoch_accuracy = validate_model(model,
                                                                   val_loader,
                                                                   criterion)
                 #append the loss and accuracy for every epoch for both train and␣
      ↪validation data set
                 train_loss.append(train_epoch_loss)
                 train_accuracy.append(train_epoch_accuracy)
                 val_loss.append(val_epoch_loss)
                 val_accuracy.append(val_epoch_accuracy)
                 # Early Stopping
                 if val_epoch_loss>prev_loss:# checking if validation loss is␣
      ↪greater than the previous loss
```

```
                counter+=1
                if counter>= max_penalty:# if the validation loss exceeds the␣
↪threshold loss twice begin the early stopping process
                    print('Early Stopping Initiated')
                    break
            else:
                counter = 0

            prev_loss=val_epoch_loss# update the previous loss to the previous␣
↪validation loss of the epoch

        #saving the best performing model
        if(val_epoch_accuracy>best_acc):
            best_acc=val_epoch_accuracy
            best_model=model
        #saving the model for a particular learning rate
        torch.save(model.state_dict(),'/content/drive/MyDrive/ASL_Project/
↪savedmodels/'+'model_mobile_net'+str(learning_rate)+'.pth' )
        #plotting the accuracy and loss curves
        print('Plotting Validation and Training Accuracy Curves for Learning␣
↪Rate of ', learning_rate)
        plot_accuracy(train_accuracy,val_accuracy)
        print('Plotting Validation and Training Loss Curves for Learning Rate␣
↪of ', learning_rate)
        plot_loss(train_loss,val_loss)

    return best_model
```

[ ]:
```
trained_mobile_net_model=mobile_net_model()
```

1.13.3 Testing the Mobile Net V2 model

Testing the model on given test dataset

[ ]:
```
test_model(trained_mobile_net_model,test_loader,0)
```

Testing the model on our test dataset

[ ]:
```
test_model(trained_mobile_net_model,my_test_loader,1)
```