CV ASSIGNMENT-03

Karthikeya M (S20160010052)
Parkhi Mohan (S20160010061)
Sree Pragna V (S20160010106)

Question-1 :

Bag of visual words model and nearest neighbor classifier: Implement K-means cluster algorithm to compute visual word dictionary. The feature dimension of SIFT features is 128. Use the included SIFT word descriptors included in “train_sift_features” and “test_sift_features” to build bag of visual words as your image representation. Use nearest neighbor classifier (kNN) to categorize the test images. Work with different number of visual words. Display the confusion matrix and categorization accuracy.

```
import numpy as np
import cv2
import csv
import sys, os
import math
import operator
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn.model_selection import train_test_split
from sklearn.cluster import KMeans
from sklearn.metrics import classification_report
from scipy.spatial import distance
from google.colab import drive
drive.mount('/content/drive/')
```

All the requirements to be used have been imported

```
def siftDescriptorInput(path, name, imageCount):
    imageSiftFeatures=[]
    imageFeatureCount=[]
    for i in range(1, imageCount+1):
        print("-->",i,"/",imageCount + 1)
        train_file=path + '/' + str(i) + '_' + name + '_sift.csv'
        inputFile=open(train_file, 'rt')
        inputFile=csv.reader(inputFile)
        featureCount=0
#         print("CSV file:", inputFile)
        for x in inputFile:
#             print(x)
            imageSiftFeatures.append(list(map(int, x[4:])))
            featureCount+=1
        imageFeatureCount.append(featureCount)
    return imageSiftFeatures, imageFeatureCount
```

This function is used to read sift descriptors from the given input file and returns image sift features and their respective count

```
def calculateKMeans(siftFeaturesTrainData, siftFeaturesTestData, KMeansClusters):
#     print("define k means")
    k_means=KMeans(n_clusters=KMeansClusters)
    print("Fitting k means for:", KMeansClusters)
    k_means.fit(siftFeaturesTrainData + siftFeaturesTestData)
    print("Done. Calculating centroid and labels")
    centroids=k_means.cluster_centers_
    labels=k_means.labels_
#     print("Centroids -->", centroids)
#     print("Labels -->", labels)
    print("Centroid and label calculation done.")
    return centroids, labels
```

This function performs K means and returns centroids and their respective labels

```
def similiarityMeasure(oneImageSiftFeature, centroidsVisualWords):
  distanceIndex=0
  distances=[]
  for feature in centroidsVisualWords:
    d=distance.euclidean(feature, oneImageSiftFeature)
    distances.append(d)
  distanceIndex=distances.index(min(distances))
  return distanceIndex
```

Similarity Measure function calculates distance between a single vector and each of the calculated centroids and it returns index of the minimum distance of the calculated ones

```
def imageToVisualWords(completeSiftFeatures, imageFeatureCount, centroidsVisualWords, KMeansClusters):
  visualWords=[]
  i=0
  count=0
  imageFeature=[0]*KMeansClusters
  for oneImageSiftFeature in completeSiftFeatures:
    distanceIndex=similiarityMeasure(oneImageSiftFeature, centroidsVisualWords)
    imageFeature[distanceIndex] += 1
    count+=1
    if count==imageFeatureCount[i]:
      visualWords.append(imageFeature)
      imageFeature=[0]*KMeansClusters
      i+=1
      count=0
  return visualWords
```

It is used to compute cluster features and return the visual words

```
def labelsInputFunc(path):
  with open(path,'rU') as inputFile:
    inputFile=csv.reader(inputFile, delimiter=',')
    inputData=list(inputFile)
    return map(int, inputData[0])
```

labelsInputFunc takes input labels from given file

```
def printConfusionMatrix(testLabels, testPrediction):
    print(classification_report(testLabels, testPrediction, target_names=['0', '1', '2', '3', '4', '5', '6', '7']))
```

```
def euclideanDistance(instance1, instance2, length):
  distance=0
  for x in range(length):
    distance += pow((instance1[x] - instance2[x]), 2)
  return math.sqrt(distance)
```

This function computes euclidean distance between two instances

```
def getNeighbors(trainingSet, testInstance, k, trainLabels):
  distances=[]
  length=len(testInstance)
  for x in range(len(trainingSet)):
    dist=euclideanDistance(testInstance, trainingSet[x], length)
    distances.append((trainingSet[x], dist, trainLabels[x]))
  distances.sort(key=operator.itemgetter(1))
  neighbors=[]
  for x in range(k):
    neighbors.append(distances[x][2])
  return neighbors
```

This finds and returns k nearest neighbours

```
def getAccuracy(testSet, predictions):
  correct=0
  for x in range(len(testSet)):
    if testSet[x] == predictions[x]:
      correct += 1
  return (correct/float(len(testSet))) * 100.0
```

```
# sift features of training set
trainFeaturesPath='./drive/My Drive/HW3_data/train_sift_features'
print("Path set. Getting train features...")
siftFeaturesTrainData, perTrainImageFeatureCount=siftDescriptorInput(trainFeaturesPath, 'train', 1888)
print(len(perTrainImageFeatureCount))
print("done")
```

It takes train set features as input

```
# sift features of test set
testFeaturesPath='./drive/My Drive/HW3_data/test_sift_features'
print("Path set. Getting test features...")
siftFeaturesTestData, perTestImageFeatureCount=siftDescriptorInput(testFeaturesPath, 'test', 800)
print(len(perTestImageFeatureCount))
print("done")
```

It takes test set features as input

```
# Use K-means to compute visual words # Cluster descriptors
def func1(KMeansClusters):
  print("Making visual words for cluster size:", KMeansClusters)
  centroidsVisualWords, labels=calculateKMeans(siftFeaturesTrainData, siftFeaturesTestData, KMeansClusters)
#   print("done\n", centroidsVisualWords)
  print("done")
  return centroidsVisualWords, labels
```

It uses K means to compute visual words

```
# Training
# Represent each image by normalized counts of visual words
def func2(centroidsVisualWords, KMeansClusters):
  trainData=imageToVisualWords(siftFeaturesTrainData, perTrainImageFeatureCount, centroidsVisualWords, KMeansClusters)
  trainLabelsPath='./drive/My Drive/HW3_data/train_labels.csv'
  trainLabels=labelsInputFunc(trainLabelsPath)
  finalTrainLabels=list(trainLabels)
  return finalTrainLabels, trainData
```

It is used to represent each train image by normalised counts of visual words

```
# Testing
def func3(centroidsVisualWords, KMeansClusters):
  testData=imageToVisualWords(siftFeaturesTestData, perTestImageFeatureCount, centroidsVisualWords, KMeansClusters)
  # test_prediction=kNN_model.predict(testData)
  testLabelsPath='./drive/My Drive/HW3_data/test_labels.csv'
  testLabels=labelsInputFunc(testLabelsPath)
  finalTestLabels=list(testLabels)
  return finalTestLabels, testData
```

It is used to represent each test image by normalised counts of visual words

```
def func4(trainData,testData,k,finalTrainLabels,finalTestLabels):
  trainingSet=trainData
  testSet=testData
#   print('Train set: ' + repr(len(trainingSet)))
#   print('Test set: ' + repr(len(testSet)))
  predictions=[]
#   k=5
  for x in range(len(testSet)):
#     print(x+1,"/", len(testSet))
    neighbors=getNeighbors(trainingSet, testSet[x], k, finalTrainLabels)
    result=getResponse(neighbors)
    predictions.append(result)
#     print('> predicted=' + repr(result) + ', actual=' + repr(finalTestLabels[x]))
  accuracy=getAccuracy(finalTestLabels, predictions)
#   print("Accuracy:", accuracy)
  return accuracy, predictions
```

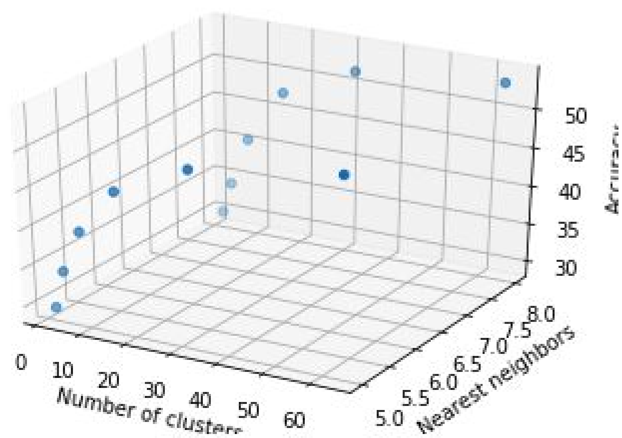
Used for prediction and accuracy using KNN

```
kMeansList = [2,4,8,16,32,64]
kNNList = [5,8]
x = []
y = []
z = []

for k1 in kMeansList:
  centroidsVisualWords, labels = func1(k1)
  finalTrainLabels, trainData = func2(centroidsVisualWords, k1)
  finalTestLabels, testData = func3(centroidsVisualWords, k1)
  for k2 in kNNList:
    accuracy, predictions = func4(trainData, testData, k2, finalTrainLabels, finalTestLabels)
    x.append(k1)
    y.append(k2)
    z.append(accuracy)
    print("Confusion matrix at kMeans:", k1, " and kNN:", k2)
    printConfusionMatrix(finalTestLabels, predictions)
    print("Accuracy:", accuracy, "%")
```

For each cluster (2,3,8,16,32,64) with nearest neighbors (5,8) code is run

```
fig = plt.figure("Kmeans and KNN accuracies")
ax = fig.add_subplot(111, projection='3d')

# x =[5, 10]
# y =[15, 30]
# z =[45, 90]



ax.scatter(x, y, z, marker='o')

ax.set_xlabel('Number of clusters') # k Means
ax.set_ylabel('Nearest neighbors')  # kNN
ax.set_zlabel('Accuracy')

plt.show()
```

Used to plot for each



Confusion matrices are :

```
Confusion matrix at kMeans: 2  and kNN: 5
              precision    recall  f1-score   support

           0       0.33      0.38      0.35       100
           1       0.65      0.64      0.65       100
           2       0.33      0.26      0.29       100
           3       0.35      0.32      0.33       100
           4       0.15      0.16      0.15       100
           5       0.18      0.21      0.19       100
           6       0.24      0.21      0.22       100
           7       0.18      0.19      0.19       100

   micro avg       0.30      0.30      0.30       800
   macro avg       0.30      0.30      0.30       800
weighted avg       0.30      0.30      0.30       800

Accuracy: 29.625 %
```

Confusion matrix at kMeans: 2 and kNN: 8

	precision	recall	f1-score	support
0	0.36	0.40	0.38	100
1	0.62	0.66	0.64	100
2	0.37	0.26	0.31	100
3	0.31	0.23	0.26	100
4	0.20	0.23	0.21	100
5	0.16	0.20	0.18	100
6	0.24	0.20	0.22	100
7	0.19	0.22	0.21	100
micro avg	0.30	0.30	0.30	800
macro avg	0.31	0.30	0.30	800
weighted avg	0.31	0.30	0.30	800

Accuracy: 30.0 %

Confusion matrix at kMeans: 4 and kNN: 5

	precision	recall	f1-score	support
0	0.45	0.45	0.45	100
1	0.76	0.72	0.74	100
2	0.42	0.32	0.36	100
3	0.35	0.30	0.32	100
4	0.18	0.21	0.20	100
5	0.21	0.28	0.24	100
6	0.32	0.32	0.32	100
7	0.18	0.17	0.18	100
micro avg	0.35	0.35	0.35	800
macro avg	0.36	0.35	0.35	800
weighted avg	0.36	0.35	0.35	800

Accuracy: 34.625 %

Confusion matrix at kMeans: 4 and kNN: 8

	precision	recall	f1-score	support
0	0.46	0.45	0.45	100
1	0.74	0.69	0.72	100
2	0.38	0.31	0.34	100
3	0.37	0.33	0.35	100
4	0.19	0.22	0.21	100
5	0.19	0.25	0.21	100
6	0.32	0.34	0.33	100
7	0.16	0.14	0.15	100
micro avg	0.34	0.34	0.34	800
macro avg	0.35	0.34	0.35	800
weighted avg	0.35	0.34	0.35	800

Accuracy: 34.125 %

Confusion matrix at kMeans: 8 and kNN: 5

	precision	recall	f1-score	support
0	0.55	0.48	0.51	100
1	0.76	0.72	0.74	100
2	0.44	0.35	0.39	100
3	0.37	0.30	0.33	100
4	0.31	0.38	0.34	100
5	0.18	0.24	0.21	100
6	0.40	0.46	0.43	100
7	0.33	0.29	0.31	100
micro avg	0.40	0.40	0.40	800
macro avg	0.42	0.40	0.41	800
weighted avg	0.42	0.40	0.41	800

Accuracy: 40.25 %

Confusion matrix at kMeans: 8 and kNN: 8

	precision	recall	f1-score	support
0	0.51	0.45	0.48	100
1	0.74	0.74	0.74	100
2	0.47	0.38	0.42	100
3	0.43	0.33	0.37	100
4	0.30	0.35	0.32	100
5	0.16	0.20	0.18	100
6	0.39	0.52	0.45	100
7	0.35	0.27	0.30	100
micro avg	0.41	0.41	0.41	800
macro avg	0.42	0.41	0.41	800
weighted avg	0.42	0.41	0.41	800

Accuracy: 40.5 %

Confusion matrix at kMeans: 16 and kNN: 5

	precision	recall	f1-score	support
0	0.47	0.46	0.46	100
1	0.80	0.80	0.80	100
2	0.57	0.44	0.50	100
3	0.54	0.42	0.47	100
4	0.41	0.41	0.41	100
5	0.26	0.34	0.29	100
6	0.41	0.47	0.44	100
7	0.36	0.36	0.36	100
micro avg	0.46	0.46	0.46	800
macro avg	0.48	0.46	0.47	800
weighted avg	0.48	0.46	0.47	800

Accuracy: 46.25 %

Confusion matrix at kMeans: 16 and kNN: 8

	precision	recall	f1-score	support
0	0.45	0.46	0.45	100
1	0.79	0.83	0.81	100
2	0.57	0.42	0.48	100
3	0.54	0.41	0.47	100
4	0.43	0.41	0.42	100
5	0.27	0.35	0.31	100
6	0.44	0.55	0.49	100
7	0.40	0.37	0.39	100
micro avg	0.47	0.47	0.48	800
macro avg	0.49	0.48	0.48	800
weighted avg	0.49	0.47	0.48	800

Accuracy: 47.5 %

Confusion matrix at kMeans: 32 and kNN: 5

	precision	recall	f1-score	support
0	0.54	0.50	0.52	100
1	0.79	0.85	0.82	100
2	0.47	0.42	0.44	100
3	0.63	0.46	0.53	100
4	0.45	0.49	0.47	100
5	0.28	0.33	0.30	100
6	0.47	0.60	0.53	100
7	0.49	0.41	0.45	100
micro avg	0.51	0.51	0.51	800
macro avg	0.52	0.51	0.51	800
weighted avg	0.52	0.51	0.51	800

Accuracy: 50.74999999999999 %

```
Confusion matrix at kMeans: 32  and kNN: 8
              precision    recall  f1-score   support

           0       0.59      0.51      0.55       100
           1       0.76      0.87      0.81       100
           2       0.52      0.49      0.51       100
           3       0.62      0.46      0.53       100
           4       0.46      0.51      0.48       100
           5       0.29      0.35      0.32       100
           6       0.48      0.60      0.54       100
           7       0.47      0.36      0.41       100

   micro avg       0.52      0.52      0.52       800
   macro avg       0.52      0.52      0.52       800
weighted avg       0.52      0.52      0.52       800

Accuracy: 51.87500000000001 %
Making visual words for cluster size: 64
Fitting k means for: 64
Done. Calculating centroid and labels
Centroid and label calculation done.
done

Confusion matrix at kMeans: 64  and kNN: 5
              precision    recall  f1-score   support

           0       0.58      0.60      0.59       100
           1       0.71      0.85      0.78       100
           2       0.56      0.55      0.56       100
           3       0.56      0.41      0.47       100
           4       0.52      0.45      0.48       100
           5       0.35      0.45      0.40       100
           6       0.50      0.58      0.54       100
           7       0.51      0.40      0.45       100

   micro avg       0.54      0.54      0.54       800
   macro avg       0.54      0.54      0.53       800
weighted avg       0.54      0.54      0.53       800

Accuracy: 53.625 %
```

```
Confusion matrix at kMeans: 64  and kNN: 8
              precision    recall  f1-score   support

           0       0.57      0.59      0.58       100
           1       0.75      0.86      0.80       100
           2       0.57      0.54      0.56       100
           3       0.57      0.43      0.49       100
           4       0.52      0.47      0.49       100
           5       0.35      0.48      0.41       100
           6       0.50      0.57      0.54       100
           7       0.48      0.35      0.40       100

   micro avg       0.54      0.54      0.54       800
   macro avg       0.54      0.54      0.53       800
weighted avg       0.54      0.54      0.53       800

Accuracy: 53.625 %
```

Observations:
With increase in k-value error decreases.

Question-2 :

Transfer Learning and Fine-tuning: Apply transfer learning with pre-trained AlexNet model trained over ImageNet database. Replace only class score layer with a new fully connected layer having 8 nodes for 8 categories. Freeze the weights of all layers except last replaced layer. Fine tune only last layer (i.e. retrain only weights of last layer). Report the accuracy Compare the results with previous nearest neighbor approach

```
import os
import torch
import imageio
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
from torchvision import datasets, models, transforms
from PIL import Image
import csv
from torch.utils.data import Dataset, DataLoader
import pandas as pd
import matplotlib.pyplot as plt
from google.colab import drive
drive.mount('/content/drive')
```

Importing all the requirements to be used for transfer learning and fine tuning

```
def readImages(imgPath, n):
  images = []
  for i in range(n):
#     print(i+1)
    path = imgPath + '/' + str(i+1) + '.jpg'
    image = imageio.imread(path)
    images.append(image)
  images = np.array(images)
#   print(images.shape)
  return images
```

The above function is to read images of the given dataset and return them

```
def readLabels(labelPath):
  with open(labelPath,'r') as csvfile:
    csvfile = csv.reader(csvfile, delimiter=',')
    data = list(csvfile)
    labels = list(map(int, data[0]))
    return labels
```

The above function is to read respective labels of images in the dataset and return them

```
def initialize(mn, n, bs, e, fe):
  modelName = mn
  numberOfClasses = n
  batchSize = bs
  epochs = e
  featureExtract = fe
  return modelName, numberOfClasses, batchSize, epochs, featureExtract
```

```
modelName, numberOfClasses, batchSize, epochs, featureExtract = initialize("alexnet", 8, 8, 5, True)
print("Model:", modelName)
print("Number of classes:", numberOfClasses)
print("Batch size:", batchSize)
print("Epochs:", epochs)
print("Feature extract:", featureExtract)
```

```
trainImagesPath='./drive/My Drive/HW3_data/train'
print("Loading 1888 train images data")
trainImagesData = readImages(trainImagesPath, 1888)
print("Done. Loading their labels")
trainLabelsPath='./drive/My Drive/HW3_data/train_labels.csv'
trainLabelsData = readLabels(trainLabelsPath)
print("Done\nNumber of train images:", len(trainImagesData))
print("Number of train labels:", len(trainLabelsData))
```

Loading train images dataset

```
testImagesPath='./drive/My Drive/HW3_data/test'
print("Loading 800 test images data")
testImagesData = readImages(testImagesPath, 800)
print("Done. Loading their labels")
testLabelsPath='./drive/My Drive/HW3_data/test_labels.csv'
testLabelsData = readLabels(testLabelsPath)
print("Done\nNumber of test images:", len(testImagesData))
print("Number of test labels:", len(testLabelsData))
```

Loading test images dataset

```
def initializeAlexnetModel(usePretrained=True):
  alexnetModel = models.alexnet(pretrained=usePretrained)
  for param in alexnetModel.parameters():
    if featureExtract==True:
      param.required_grad = False
  numberOfFeatures = alexnetModel.classifier[6].in_features
  alexnetModel.classifier[6] = nn.Linear(numberOfFeatures, numberOfClasses)
  inputSize = 224
  return alexnetModel, inputSize
```

The above function initialises the alexnet model

```
alexnetModel, inputSize = initializeAlexnetModel()
print(alexnetModel)
```

Here alexnet model is initialised

```
class FaceLandmarksDataset(Dataset):
    """Face Landmarks dataset."""

    def __init__(self,root_dir,total_count, csv_file, transform=None):
        """
        Args:
            csv_file (string): Path to the csv file with annotations.
            root_dir (string): Directory with all the images.
            transform (callable, optional): Optional transform to be applied
                on a sample.
        """
        self.root_dir = root_dir
        self.total_count = total_count
        self.csv_file = np.array(pd.read_csv(csv_file,header=None))[0]
        self.transform = transform

    def __len__(self):
        return self.total_count

    def __getitem__(self, image_no):
        img_name = os.path.join(self.root_dir,
                                str(image_no+1)+".jpg")
        image = Image.open(img_name)
        sample = {'image': image, 'label': self.csv_file[image_no]-1}

        if self.transform:
            sample['image'] = self.transform(sample['image'])

        return sample
```

Above function is used to get images with their respective labels

```
rootDirImage = './drive/My Drive/HW3_data/train'
rootDirLabel = './drive/My Drive/HW3_data/train_labels.csv'
trainData = FaceLandmarksDataset(rootDirImage, 1888, rootDirLabel, transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]))
```

Loading train images dataset and initialize the model

```
rootDirImage = './drive/My Drive/HW3_data/test'
rootDirLabel = './drive/My Drive/HW3_data/test_labels.csv'
testData = FaceLandmarksDataset(rootDirImage, 800, rootDirLabel, transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]))
```

Loading test images dataset

```
# Detect if we have a GPU available
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print("Initializing Datasets and Dataloaders...")
trainDataLoader = torch.utils.data.DataLoader(trainData,batch_size=batchSize, shuffle=True, num_workers=4)
testDataLoader = torch.utils.data.DataLoader(testData,batch_size=batchSize, shuffle=True, num_workers=4)
dataLoaderDictionary = {trainDataLoader, testDataLoader}
print(dataLoaderDictionary)
# Send the model to GPU
alexnetModel = alexnetModel.to(device)
```

```
# Gather the parameters to be optimized/updated in this run. If we are
#  finetuning we will be updating all parameters. However, if we are
#  doing feature extract method, we will only update the parameters
#  that we have just initialized, i.e. the parameters with requires_grad
#  is True.
paramsToUpdate = alexnetModel.parameters()
print("Params to learn:")
if featureExtract:
    paramsToUpdate = []
    for name, param in alexnetModel.named_parameters():
        if param.requires_grad == True:
            paramsToUpdate.append(param)
            print("-->",name)
else:
    for name, param in alexnetModel.named_parameters():
        if param.requires_grad == True:
            print("\t",name)
```

In the above function parameters to be learnt are found and appended to paramsToUpdate

```
# Observe that all parameters are being optimized
optimizer = optim.SGD(paramsToUpdate, lr=0.001, momentum=0.9)
print(optimizer)
# Setup the loss fxn
criterion = nn.CrossEntropyLoss()
print("Criterion:", criterion)
```

Stochastic gradient descent is used as optimizer with learning rate as 0.001, momentum as 0.9
Cross entropy loss is taken as our criteria.

```
for epoch in range(epochs):
    runningLoss = 0.0
    for i, data in enumerate(trainDataLoader, 0):
        # get the inputs
        inputs, labels = data['image'], data['label']
        inputs = inputs.to(device)
        labels = labels.to(device)
        # zero the parameter gradients
        optimizer.zero_grad()
        outputs = alexnetModel(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        runningLoss += loss.item()
        # print every 10 mini-batches
        if i % 10 == 9:
            print('(%d, %d) loss: %.3f' %(epoch+1, i+1, runningLoss/10))
            runningLoss = 0.0
print('Done')
```

Here we trained for 5 epochs

```
print("Running Loss:", runningLoss)
total = 0
correct = 0
with torch.no_grad():
    for i, data in enumerate(testDataLoader, 0):
        # get the inputs
        inputs, labels = data['image'], data['label']
        inputs = inputs.to(device)
        labels = labels.to(device)
        outputs = alexnetModel(inputs)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print("Accuracy:", correct/total*100, "%")
```

After training for 5 epochs the loss occurred is 1.856 and accuracy obtained is 91.125%

```
Running Loss: 1.8558482825756073
Accuracy: 91.125 %
```

Performs much better than the nearest neighbors approach.

Question-3 :

Training from Scratch with ResNet18 model .Designing ResNet18 model and training all layers from scratch. Comparing the results with previous two methods.
Extra Credit (Optional): Repeating this experiment over CIFAR10 dataset.

Importing the required Libraries:

```
import tensorflow as tf
import cv2
import matplotlib.pyplot as plt
import os
import pandas as pd
import numpy as np
import torch
import torchvision
import torch.nn as nn
from PIL import Image
```

Creating the ResNet model with 18 layers architecture:

Function for 1x1 convolution and 3x3 convolution are as follows:

```
def conv1x1(in_planes, out_planes, stride=1):
    """1x1 convolution"""
    return nn.Conv2d(in_planes, out_planes, kernel_size=1, stride=stride, bias=False)
```

```
def conv3x3(in_planes, out_planes, stride=1, groups=1):
    """3x3 convolution with padding"""
    return nn.Conv2d(in_planes, out_planes, kernel_size=3, stride=stride,
                     padding=1, groups=groups, bias=False)
```

Basic Block or Residual block where a single block contains two skip connections is as follows:

```
class BasicBlock(nn.Module):
    expansion = 1

    def __init__(self, inplanes, planes, stride=1, downsample=None, groups=1,
                 base_width=64, norm_layer=None):
        super(BasicBlock, self).__init__()
        if norm_layer is None:
            norm_layer = nn.BatchNorm2d
        if groups != 1 or base_width != 64:
            raise ValueError('BasicBlock only supports groups=1 and base_width=64')
        # Both self.conv1 and self.downsample layers downsample the input when stride != 1
        self.conv1 = conv3x3(inplanes, planes, stride)
        self.bn1 = norm_layer(planes)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(planes, planes)
        self.bn2 = norm_layer(planes)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x):
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)

        if self.downsample is not None:
            identity = self.downsample(x)

        out += identity
        out = self.relu(out)

        return out
```

Below code will linearly arrange the architecture of ResNet, this can be tailored for any number of layers.

```python
class ResNet(nn.Module):

    def __init__(self, block, layers, num_classes=1000, zero_init_residual=False,
                 groups=1, width_per_group=64, norm_layer=None):
        super(ResNet, self).__init__()
        if norm_layer is None:
            norm_layer = nn.BatchNorm2d

        self.inplanes = 64
        self.groups = groups
        self.base_width = width_per_group
        self.conv1 = nn.Conv2d(3, self.inplanes, kernel_size=7, stride=2, padding=3,
                               bias=False)
        self.bn1 = norm_layer(self.inplanes)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.layer1 = self._make_layer(block, 64, layers[0], norm_layer=norm_layer)
        self.layer2 = self._make_layer(block, 128, layers[1], stride=2, norm_layer=norm_layer)
        self.layer3 = self._make_layer(block, 256, layers[2], stride=2, norm_layer=norm_layer)
        self.layer4 = self._make_layer(block, 512, layers[3], stride=2, norm_layer=norm_layer)
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512 * block.expansion, num_classes)

        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
            elif isinstance(m, (nn.BatchNorm2d, nn.GroupNorm)):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)

        # Zero-initialize the last BN in each residual branch,
        # so that the residual branch starts with zeros, and each residual block behaves like an identity.
        # This improves the model by 0.2~0.3% according to https://arxiv.org/abs/1706.02677
        if zero_init_residual:
            for m in self.modules():
                if isinstance(m, Bottleneck):
                    nn.init.constant_(m.bn3.weight, 0)
                elif isinstance(m, BasicBlock):
                    nn.init.constant_(m.bn2.weight, 0)

    def _make_layer(self, block, planes, blocks, stride=1, norm_layer=None):
        if norm_layer is None:
            norm_layer = nn.BatchNorm2d
        downsample = None
        if stride != 1 or self.inplanes != planes * block.expansion:
```

```python
        return nn.Sequential(*layers)

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.avgpool(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)

        return x
```

Using the above Resnet model, to make it suitable for 18 layers below code is used.

```python
def resnet18(pretrained=False, **kwargs,):
    """Constructs a ResNet-18 model.
    Args:
        pretrained (bool): If True, returns a model pre-trained on ImageNet
    """
    model = ResNet(BasicBlock, [2, 2, 2, 2], **kwargs)
    if pretrained:
        model.load_state_dict(model_zoo.load_url(model_urls['resnet18']))
    return model
```

Running this model on the given Dataset:

Creating the Resnet 18 model for 8 Class Labels:

```python
device = torch.device("cuda:0" if (torch.cuda.is_available()) else "cpu")
model = resnet18(num_classes=8).to(device)
```

Reading the train and test datasets which has **8 Class Labels:**

```python
trans = torchvision.transforms.Compose([torchvision.transforms.Resize((224,224)),torchvision.transforms.ToTensor(),
                                        torchvision.transforms.Normalize((0.5,0.5,0.5),(0.5,0.5,0.5))])
trainData = ReadData("/content/drive/My Drive/CV3/train",trainLabels,transform = trans)
testData = ReadData("/content/drive/My Drive/CV3/test",testLabels,transform = trans)
```

Here the data is normalized and resized to 224x224

ReadData function is coded as follows:

```python
class ReadData(torch.utils.data.Dataset):
  def __init__(self,root,Ydata,transform):
    self.root = root
    self.transform = transform
    self.Ydata = Ydata
  def __len__(self):
    return self.Ydata.shape[0]
  def __getitem__(self,index):
#     print("index:", index)
    img = Image.open(self.root+"/"+str(index+1)+".jpg")
    img = self.transform(img)
    label = self.Ydata[index]
    dictImg = {'X':img,'Y':label}
    return dictImg
```

Here the image and its respective label are made into a dictionary and returned. The indexing is random in this case.

Data Loading of train and test by keeping the batch size as 128:

```
trainDataLoader = torch.utils.data.DataLoader(trainData,batch_size=128,shuffle=True,num_workers=2)
testDataLoader = torch.utils.data.DataLoader(testData,batch_size=128,shuffle=True,num_workers=2)
```

Using the Cross Entropy Loss function and Adam as Optimizer:

```
loss = nn.CrossEntropyLoss()
```

```
optim = torch.optim.Adam(model.parameters(),lr = 0.002, weight_decay = 1e-4)
epochs = 20
```

Training this model on the train Data for 20 epochs:

```
for i in range(epochs):
  truePos = 0
  te = 0
  print("epochs",i)
  print("Training .......")
  for data in trainDataLoader:
    Xbatch = data['X'].to(device)
    Ybatch = data['Y'].to(device)
#     Xbatch = data[0].to(device)
#     Ybatch = data[1].to(device)
#     print(Ybatch.shape)
    output = model(Xbatch)
#     print("output",output.shape)
    l = loss(output,Ybatch)
    te += Xbatch.shape[0]
    _,predict = torch.max(output,1)
    truePos+=(predict == Ybatch).sum().item()



    optim.zero_grad()
    l.backward()
    optim.step()
    print("accuracy of Train is :",(truePos/te)*100,"% ",", Loss : ",l.item())
```

Testing on the test data for every train batch in the following way:

```
    #test
  print("************************************************")
  print("Testing......")
  truePos = 0
  te = 0
  for data in testDataLoader:
    Xbatch = data['X'].to(device)
    Ybatch = data['Y'].to(device)
#     Xbatch = data[0].to(device)
#     Ybatch = data[1].to(device)
    output = model(Xbatch)
    te += Xbatch.shape[0]
    _,predict = torch.max(output,1)
    truePos+=(predict == Ybatch).sum().item()
  print("accuracy of Test is :",(truePos/te)*100,"%")
  print("************************************************")
```

RESULT:

After 20 epochs, the following result is obtained:

```
accuracy of Train is : 89.6484375 %  , Loss :  0.24729229509830475
accuracy of Train is : 89.90384615384616 %  , Loss :  0.2229180485010147
accuracy of Train is : 89.39732142857143 %  , Loss :  0.5096961259841919
accuracy of Train is : 89.24788135593221 %  , Loss :  0.30738478899002075
************************************************
Testing......
accuracy of Test is : 84.25 %
************************************************
```

Test Accuracy on the Given Dataset is : 84.25%

Running this model on CIFAR-10 Dataset:

Creating the Resnet 18 model for 10 class Labels:

```
device = torch.device("cuda:0" if (torch.cuda.is_available()) else "cpu")
model = resnet18(num_classes=10).to(device)
```

Reading the train and test datasets which has **10 Class Labels**:

```
trans = torchvision.transforms.Compose([torchvision.transforms.Resize((224,224)),torchvision.transforms.ToTensor(),
                                        torchvision.transforms.Normalize((0.5,0.5,0.5),(0.5,0.5,0.5))])
trainData = torchvision.datasets.CIFAR10(root="./cifar",train=True,download=True,transform=trans)
testData = torchvision.datasets.CIFAR10(root="./cifar",train=False,download=True,transform=trans)
```

Here the data is normalized and resized to 224x224

Data Loading of train and test by keeping the batch size as 128:

```
trainDataLoader = torch.utils.data.DataLoader(trainData,batch_size=128,shuffle=True,num_workers=2)
testDataLoader = torch.utils.data.DataLoader(testData,batch_size=128,shuffle=True,num_workers=2)
```

Using the Cross Entropy Loss function and Adam as Optimizer:

```
loss = nn.CrossEntropyLoss()
```

```
optim = torch.optim.Adam(model.parameters(),lr = 0.002, weight_decay = 1e-4)
epochs = 20
```

Training this model on the train Data for 20 epochs:

```
for i in range(epochs):
  truePos = 0
  te = 0
  print("epochs",i)
  print("Training .......")
  for data in trainDataLoader:
#     Xbatch = data['X'].to(device)
#     Ybatch = data['Y'].to(device)
    Xbatch = data[0].to(device)
    Ybatch = data[1].to(device)
#     print(Ybatch.shape)
    output = model(Xbatch)
#     print("output",output.shape)
    l = loss(output,Ybatch)
    te += Xbatch.shape[0]
    _,predict = torch.max(output,1)
    truePos+=(predict == Ybatch).sum().item()



    optim.zero_grad()
    l.backward()
    optim.step()
    print("accuracy of Train is :",(truePos/te)*100,"% ",", Loss : ",l.item())
```

Testing on the test data for every train batch in the following way:

```
    #test
  print("************************************************")
  print("Testing......")
  truePos = 0
  te = 0
  for data in testDataLoader:
#     Xbatch = data['X'].to(device)
#     Ybatch = data['Y'].to(device)
    Xbatch = data[0].to(device)
    Ybatch = data[1].to(device)
    output = model(Xbatch)
    te += Xbatch.shape[0]
    _,predict = torch.max(output,1)
    truePos+=(predict == Ybatch).sum().item()
  print("accuracy of Test is :",(truePos/te)*100,"%")
  print("************************************************")
```

RESULT:

After 20 epochs, the following result is obtained:

```
accuracy of Train is : 95.37275064267352 %  , Loss :  0.17626650631427765
accuracy of Train is : 95.37660256410257 %  , Loss :  0.10856685042381287
accuracy of Train is : 95.366 %  , Loss :  0.20923814177513123
************************************************
Testing......
accuracy of Test is : 83.22 %
************************************************
```

Test Accuracy on CIFAR-10 is : 83.22%

OBSERVATIONS:

The training images in the given dataset are very less to train and this effects the test accuracy. If this model is trained on a bigger dataset for more amount of time we will get much better accuracy.

OUTPUT COMPARISON:

If we compare the accuracy results of “**Pre-Trained ALEX NET**” and “**RESNET-18**” which is trained on the given training dataset of **1888** images, the test accuracy of **ALEX NET** (**91.125%**) is greater than that of the test accuracy of **RESNET18** (**84.25%**). This is because the Alex Net is pretrained on a bigger dataset (such as ImageNet) that is why the accuracy is higher in this case. Whereas RESNET18 is trained on rather small dataset of 1888 images. Even then it gave good results on the test dataset.