ASSIGNMENT 3

COMPUTER VISION _ SPRING 2019

B S N V Chaitanya(S20160020115) D Sumanth(S20160020125)
K Anjali Poornima(S20160020132)

Question 1 : Bag of visual words model and nearest neighbor classifier

Analysing the steps to implement Bag of Visual Words and KNN classifier:

1. Firstly, all the Keypoints and descriptors for all training images are read from the respective csv file and are saved in a pickle file.
2. Next, Cluster descriptors are obtained using K-means clustering. This is done by using **sklearn.cluster. KMeans ,** where KMeans model is initialized and then fitted to the training data(sift features).
3. These descriptors are quantized using the cluster centroids to get 'Visual Words'
4. Then, each image is represented by normalized counts of 'Visual Words' , i.e. , the image is represented as a histogram(frequencies). For this, the images are ranked by nearest neighbour search for similar images.
5. Then, a KNN classifier is trained on the labeled examples using histogram values as features.
6. KNN classifier, K nearest neighbour classifier finds the K closest points from training data for any new point. Then vote for class label with labels of the K points. This is done using **sklearn.neighbors.KNeighborsClassifier .**
7. For **Testing,** the given sift descriptors are quantized into visual words and a visual word histogram is computed.
8. Then that label is assigned for which the confidence of classifier is higher.

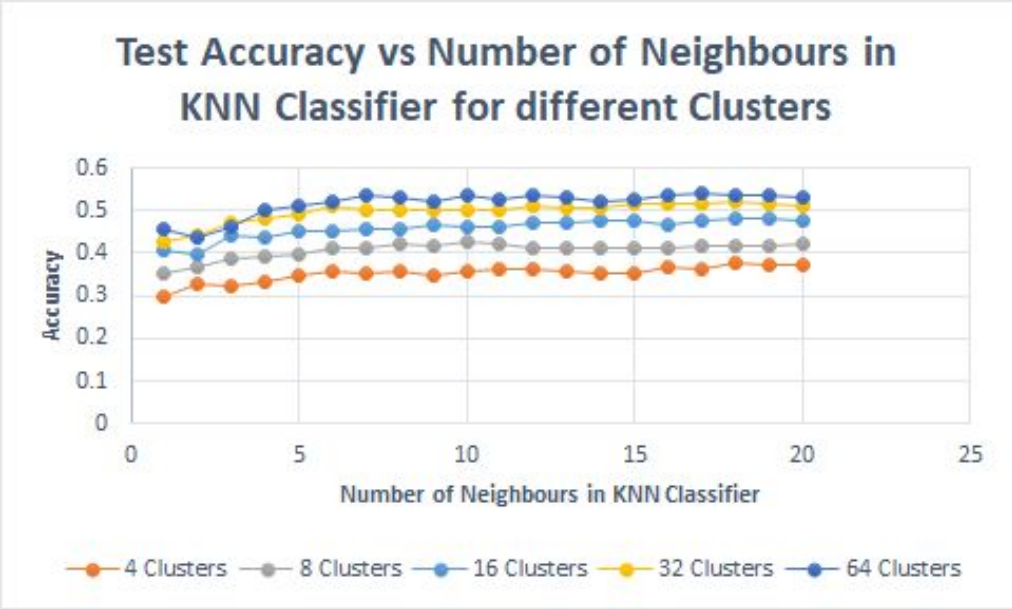
Code:

```
import csv
import time
from sklearn.cluster import KMeans
from scipy.spatial import distance
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report
from sklearn.metrics import accuracy_score
import numpy as np
import pickle
import warnings
warnings.filterwarnings('ignore')
def getFeatures(name,count):
    try:
```

```
        siftFeatures = pickle.load(open("pickleFiles/Features_"+name+str(count)+".pickle", "rb"))
        detectedRegions = pickle.load(open("pickleFiles/Regions_"+name+str(count)+".pickle", "rb"))
        print("Loaded Features and Regions Pickle file for "+name+"!!!")
    except (OSError, IOError) as e:
        siftFeatures = []
        detectedRegions = []
        for i in range(count):
            file = './' + name +'_sift_features/' + str(i+1) + '_' + name + '_sift.csv'
            csvFile = csv.reader(open(file))
            regions = 0
            for line in csvFile:
                siftFeatures.append(line[4:])
                regions += 1
            detectedRegions.append(regions)
        pickle.dump(siftFeatures, open("pickleFiles/Features_"+name+str(count)+".pickle", "wb"))
        pickle.dump(detectedRegions, open("pickleFiles/Regions_"+name+str(count)+".pickle", "wb"))
    return detectedRegions , siftFeatures
def getClusterDescriptors(siftFeatures,clusters,x):
    try:
        kmeans = pickle.load(open("pickleFiles/Kmeans_"+str(clusters)+"_"+str(x)+".pickle", "rb"))
        print("Loaded kmeans Pickle file!!!")
    except (OSError, IOError) as e:
        print("no kmeans pickle file found!!! Creating one")
        kmeans = KMeans(n_clusters = clusters)
        kmeans.fit(siftFeatures)
        pickle.dump(kmeans, open("pickleFiles/Kmeans_"+str(clusters)+"_"+str(x)+".pickle", "wb"))
        print("Saved kmeans pickle file!!     "+ "pickleFiles/Kmeans_"+str(clusters)+"_"+str(x)+".pickle")
    return kmeans.cluster_centers_
def getVisualWords(siftFeatures,regions_train,clusterCenters,clusters,name,save = False):
    try:
        vocab = pickle.load(open("pickleFiles/vocab_"+name+"_"+str(clusters)+"_"+str(len(regions_train))+".pickle",
"rb"))
        print("loaded "+name+" vocab pickle file!!!")
    except (OSError, IOError) as e:
        print("no "+name+" vocab pickle file found!!! Creating one")
        def getDistance(f):
            f1 = [float(i) for i in f]
            dist = []
            for i in clusterCenters:
                d = np.linalg.norm(i-f1)
                dist.append(d)
            return dist
        vocab = []
        features = [0] * clusters
        count = 0
        temp = 0
        for f in siftFeatures:
            dist = getDistance(f)
            i = dist.index(min(dist))
            features[i] += 1
            count += 1
            if count == regions_train[temp]:
                vocab.append(features)
                features = [0] * clusters
                count = 0
                temp += 1
        if(save == True):
            pickle.dump(vocab,open("pickleFiles/vocab_"+name+"_"+str(clusters)+"_"+str(len(regions_train))+".pickle",
"wb"))
            print("Saved "+name+" vocab pickle file!!")
    return vocab
```

```
print("Reading Files!!!")
numOfTrainSamples = 1888
numOfTestSamples = 800
regions_train,features_train = getFeatures('train' , numOfTrainSamples)
regions_test,features_test = getFeatures('test' , numOfTestSamples)
print("Number of Images in train " + str(len(regions_train)))
print(len(features_train))
print("Number of Images in test " + str(len(regions_test)))
tic = time.time()
clusters = 32
clusterCenters = getClusterDescriptors(features_train,clusters,numOfTrainSamples)
print(clusterCenters)
print("Time taken to calculate Clusters: " + str(time.time() - tic))
tic = time.time()
vocab_Train = getVisualWords(features_train,regions_train,clusterCenters,clusters,"train",save = True)
vocab_Test = getVisualWords(features_test,regions_test,clusterCenters,clusters,"test",save = True)
print("Time taken to represent image by normalized counts of visual words: " + str(time.time()-tic))
with open( './train_labels.csv','rU') as csvfile:
        csvfile = csv.reader(csvfile, delimiter=',')
        train_labels = [int(i) for i in list(csvfile)[0]]
with open( './test_labels.csv','rU') as csvfile:
        csvfile = csv.reader(csvfile, delimiter=',')
        test_labels = [int(i) for i in list(csvfile)[0]]
#print(train_labels)
def kNN_classifer(train_images, train_labels, k_NN):
    knn = KNeighborsClassifier(n_neighbors = k_NN)
    knn.fit(train_images, train_labels)
    return knn
accuracies = []
for i in range(1,21):
    k_NN = i
    kNN_model = kNN_classifer(vocab_Train, train_labels, k_NN)
    test_prediction = kNN_model.predict(vocab_Test)
    #print (classification_report(test_labels, test_prediction, target_names=['1', '2', '3', '4', '5', '6', '7', '8']))
    categorization_accuracy = accuracy_score(test_labels, test_prediction)
    print (categorization_accuracy)
    accuracies.append((i,categorization_accuracy))
```

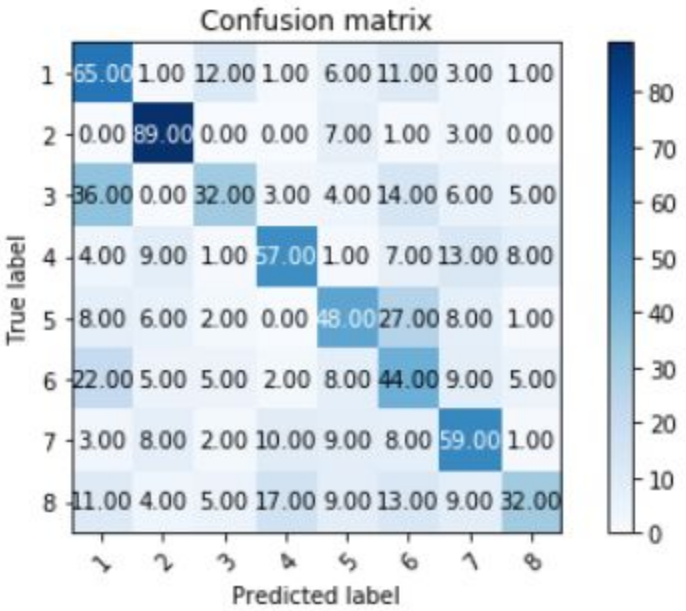
Observations:



The above graph represents the Test accuracy and the ‘K’ in K Nearest Neighbours as the number for clusters in KMeans clustering Varies. The observation is that, as the number of clusters increase, the accuracy increases. And also as the number of neighbors increase, the accuracy increases and finally stabilizes.

Clusters	Value of K (KNN)	Test Accuracy	Time for KMeans(in sec)
4	18	0.3775	279.697
8	10	0.4275	466.607
16	18	0.48375	1071.587
32	18	0.52	1348.79
64	17	0.54	8881.086

The above table illustrates the Maximum frequency and Time taken to compute KMeans clustering for given Clusters and K(neighbours in KNN Classifier).
The observation is that, as lusters increase accuracy increases, but the time taken also increases.



Confusion matrix for the prediction for 64 clusters and 20 Nearest Neighbours

Question 2 : Transfer Learning with AlexNet model

Analysing the steps to train AlexNet model:

1. Initially, the AlexNet (pretrained on ImageNet) is loaded and the top layers are freezed.
2. We loaded the 1888 Images of training data and 800 Images of test data for transfer learning on the pre-trained model
3. We trained the last layer on the AlexNet to output probabilities for 8 classes
4. The pre-trained model was downloaded from pytorch
5. Coming to the training, we applied random crop method to reduce the size of image from a dimension of 256 x 256 to 224 x 224
6. We used CrossEntropyLoss and SGD Optimizer with momentum 0.9 and learning rate 0.001

Code:

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

import os
import cv2
import torch
import scipy
import torch.nn as nn
import torch.nn.functional as F
from torchvision import models
from __future__ import print_function
from __future__ import division
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import torchvision
from torchvision import datasets, models, transforms
import matplotlib.pyplot as plt
import time
import os
import copy
import torchvision.transforms.functional as TF
from PIL import Image
import matplotlib.pyplot as plt
from scipy import io as sio
from scipy import ndimage, misc
import csv
from torch.utils.data import Dataset, DataLoader
import pandas as pd

data_dir = "ResNet18_data/train"

# Models to choose from [resnet, alexnet, vgg, squeezenet, densenet, inception]
model_name = "alexnet"

# Number of classes in the dataset
```

```
num_classes = 8

# Batch size for training (change depending on how much memory you have)
batch_size = 8

# Number of epochs to train for
num_epochs = 15

# Flag for feature extracting. When False, we finetune the whole model,
#   when True we only update the reshaped layer params
feature_extract = True

def read_images(images_path, number_of_images):
  images = []
  for i in range(number_of_images):
            print(i+1)
            path = images_path + '/' + str(i+1) + '.jpg'
            image = scipy.misc.imread(path)
            images.append(image)
  images = np.array(images)
  print(images.shape)
  return images

def read_labels(labels_path):
    with open(labels_path,'rU') as csvfile:
             csvfile = csv.reader(csvfile, delimiter=',')
             csvdata = list(csvfile)
             labels = map(int, csvdata[0])
             return list(labels)

def set_parameter_requires_grad(model, feature_extracting):
            if feature_extracting:
            for param in model.parameters():
            param.requires_grad = False

alexnet = models.alexnet(pretrained=True)
alexnet.classifier[6] = nn.Linear(4096,8)
alexnet.eval()

def initialize_model(model_name, num_classes, feature_extract, use_pretrained=True):
            # Initialize these variables which will be set in this if statement. Each of these
            #   variables is model specific.
            model_ft = None
            input_size = 0

            if model_name == "resnet":
            """ Resnet18
            """
            model_ft = models.resnet18(pretrained=use_pretrained)
            set_parameter_requires_grad(model_ft, feature_extract)
            num_ftrs = model_ft.fc.in_features
            model_ft.fc = nn.Linear(num_ftrs, num_classes)
            input_size = 224

            elif model_name == "alexnet":
            """ Alexnet
            """
            model_ft = models.alexnet(pretrained=use_pretrained)
            set_parameter_requires_grad(model_ft, feature_extract)
            num_ftrs = model_ft.classifier[6].in_features
            model_ft.classifier[6] = nn.Linear(num_ftrs,num_classes)
```

```
        input_size = 224

        else:
        print("Invalid model name, exiting...")
        exit()

        return model_ft, input_size

train_labels = np.array(pd.read_csv('/content/drive/My Drive/ResNet18_data/train_labels.csv',header=None))[0]
test_labels = np.array(pd.read_csv('/content/drive/My Drive/ResNet18_data/test_labels.csv',header=None))[0]

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

trainDataset = FaceLandmarksDataset('/content/drive/My Drive/ResNet18_data/train',1888,'/content/drive/My
Drive/ResNet18_data/train_labels.csv',transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
        ]))

testDataset = FaceLandmarksDataset('/content/drive/My Drive/ResNet18_data/test',800,'/content/drive/My
Drive/ResNet18_data/test_labels.csv',transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
        ]))



# Initialize the model for this run
model_ft, input_size = initialize_model("alexnet", 8, feature_extract, use_pretrained=True)
```

```
# Print the model we just instantiated
print(model_ft)

print("Initializing Datasets and Dataloaders...")

trainDataloader = torch.utils.data.DataLoader(trainDataset,batch_size = 8, shuffle = True, num_workers=4)
testDataloader = torch.utils.data.DataLoader(testDataset,batch_size = 8, shuffle = True, num_workers=4)
Dataloaders_dict = {trainDataloader,testDataloader}
# Detect if we have a GPU available
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# Send the model to GPU
model_ft = model_ft.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.
params_to_update = model_ft.parameters()
print("Params to learn:")
if feature_extract:
        params_to_update = []
        for name,param in model_ft.named_parameters():
        if param.requires_grad == True:
        params_to_update.append(param)
        print("\t",name)
else:
        for name,param in model_ft.named_parameters():
        if param.requires_grad == True:
        print("\t",name)

# Observe that all parameters are being optimized
optimizer_ft = optim.SGD(params_to_update, lr=0.001, momentum=0.9)

# Setup the loss fxn
criterion = nn.CrossEntropyLoss()

# Train and evaluate
#model_ft, hist = train_model(model_ft,trainDataloader,testDataloader, criterion, optimizer_ft,
num_epochs=num_epochs, is_inception=(model_name=="alexnet"))

for epoch in range(10):  # loop over the dataset multiple times

        running_loss = 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_ft.zero_grad()
        outputs = model_ft(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer_ft.step()

        # print statistics
        running_loss += loss.item()
        if i % 10 == 9:       # print every 10 mini-batches
```

```
            print('[%d, %5d] loss: %.3f' %(epoch + 1, i + 1, running_loss / 10))
            running_loss = 0.0

print('Finished Training')
running_loss
correct = 0
total = 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 = model_ft(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 800 test images: %d %%' % (
            100 * correct / total))
```

Accuracy: 90%
Epochs Trained: 10

Question 3 : Training from Scratch with ResNet18 model

Analysing the steps to train ResNet18 model:

1. Firstly, to describe ResNet18 Model, there are 18 convolution layers with 8 skip layers. After each conv layer, we do batch Normalization with Relu activation function.
2. For the first conv layer, we convolve with 64 filters of 7x7 kernel followed by a max pooling layer.
3. From then on, every conv layer uses same kernel size of 3x3.
4. Residual Network : For 3 consecutive conv layers, the input of the third layer is the combined input of the first layer and Relu of second layer.
5. Coming to the training, we applied random crop method to reduce the size of image from a dimension of 256 x 256 to 224 x 224
6. Training details : Batch size = 32 with regularization coefficient of 0.0001
7. Learning rate = 0.001 initially and is reduced to 0.01(by a factor of 10) after half the epochs are done.
8. Trained for 30 epochs
9. We used RMSProp optimizer with a momentum of 0.9

Layer Name	Output Size
Conv 1	112 x 112 x 64
Skip Conn_1 Conv_1	56 x 56 x 64
Skip Conn_1 Conv_2	56 x 56 x 64
Skip Conn_2 Conv_1	56 x 56 x 64
Skip Conn_2 Conv_2	56 x 56 x 64
Skip Conn_3 Conv_1	28 x 28 x 128
Skip Conn_3 Conv_2	28 x 28 x 128
Skip Conn_4 Conv_1	28 x 28 x 128
Skip Conn_4 Conv_2	28 x 28 x 128
Skip Conn_5 Conv_1	14 x 14 x 256
Skip Conn_5 Conv_2	14 x 14 x 256
Skip Conn_6 Conv_1	14 x 14 x 256
Skip Conn_6 Conv_2	14 x 14 x 256
Skip Conn_7 Conv_1	7 x 7 x 512
Skip Conn_7 Conv_2	7 x 7 x 512
Skip Conn_8 Conv_1	7 x 7 x 512
Skip Conn_8 Conv_2	7 x 7 x 512
Avg_pool (7 x 7)	1 x 1 x 512
Fully_conn_layer + softmax	8 classes labels

Code:

```
import tensorflow as tf
import cv2, numpy as np, os
import pandas as pd, time
import tensorflow.contrib as tf_contrib
DECAY_BN = 0.0001
EPSILON_BN = 1e-05

def get_mini_batches(X, y, crop = True, batch_size = 32):

    n_minibatches = X.shape[0] // batch_size
```

```
    data = list(zip(X, y))
    np.random.shuffle(data)
    X, y = zip(*data)
    X = np.array(X)
    y = np.array(y)
    mini_batches = []
    i = 0
    for i in range(n_minibatches):
        X_mini = X[i * batch_size:(i + 1)*batch_size]
        if(crop):
            X_mini = data_augmentation(X_mini, 224)
        y_mini = y[i * batch_size:(i + 1)*batch_size]
        mini_batches.append((X_mini, y_mini))
    if(X.shape[0] % batch_size != 0):
        X_mini = X[i * batch_size:X.shape[0]]
        if(crop):
            X_mini = data_augmentation(X_mini, 224)
        y_mini = y[i * batch_size:y.shape[0]]
        mini_batches.append((X_mini, y_mini))
        n_minibatches += 1
    return mini_batches, n_minibatches

def _random_crop(batch, crop_shape, padding=None):

    oshape = np.shape(batch[0])
    if padding:
        oshape = (oshape[0] + 2 * padding, oshape[1] + 2 * padding)
    new_batch = []
    npad = ((padding, padding), (padding, padding), (0, 0))
    for i in range(len(batch)):
        new_batch.append(batch[i])
        if padding:
            new_batch[i] = np.lib.pad(batch[i], pad_width=npad, mode='constant', constant_values=0)
        nh = np.random.randint(0, oshape[0] - crop_shape[0])
        nw = np.random.randint(0, oshape[1] - crop_shape[1])
        new_batch[i] = new_batch[i][nh:nh + crop_shape[0], nw:nw + crop_shape[1]]
    return new_batch


def data_augmentation(batch, img_size):
    return _random_crop(batch, [img_size, img_size, 3])

def read_images(path, ran, phase):
    save_path = phase+"_images.npy"
    if(os.path.exists(save_path)):
        print("npy file available...")
        return np.load(save_path)
    images = []
    for r, d, f in os.walk(path):
        for i in range(1, ran+1):
            img_name = os.path.join(r, str(i)+".jpg")
            images.append(cv2.imread(img_name)/255.)
    if(phase == 'test'):
        print("augumenting to 224")
        images = data_augmentation(images, 224)
    print("done loading from " + path)
    images = np.array(images)
    np.save(save_path, images)
    return images

def read_labels(csv_file, phase, nb_classes = 8):
```

```
    labels = np.array(pd.read_csv(csv_file, header = None).values)
    labels = np.reshape(labels, (labels.shape[1], labels.shape[0]))
    one_hot_targets = np.eye(nb_classes)[[i-1 for i in labels]]
    one_hot_targets = np.reshape(one_hot_targets, (one_hot_targets.shape[0], 8))

    return one_hot_targets


def get_num_trainable_params():
    total_parameters = 0
    for variable in tf.trainable_variables():
        shape = variable.get_shape()
        variable_parameters = 1
        for dim in shape:
            variable_parameters *= dim.value
        total_parameters += variable_parameters
    print(total_parameters)

train_images = read_images("../train/", 1888, "train")
test_images = read_images("../test/", 800, "test")

train_labels = read_labels("../train_labels.csv", "train")
test_labels = read_labels("../test_labels.csv", "test")

print("Train images: " + str(train_images.shape))
print("Test images:  " + str(test_images.shape))

print("Train labels: " + str(train_labels.shape))
print("Test labels:  " + str(test_labels.shape))

def residual_block(x, filter_size, wt_init, weight_regularizer, conv_number, instance, is_train, stride = 1, reuse =
False):
    prev = x

    x = tf.layers.conv2d(x, filters=filter_size, kernel_size=3, kernel_initializer=wt_init, strides=stride,
                         kernel_regularizer=weight_regularizer, padding="SAME", trainable=is_train,
name="shortcut_conv"+conv_number+"_"+instance, reuse=reuse)
    x = tf_contrib.layers.batch_norm(x, decay=DECAY_BN, epsilon=EPSILON_BN, center=True, scale=True,
trainable=is_train, reuse=reuse, scope="short_batch_norm"+conv_number+"_"+instance)
    x = tf.nn.relu(x)
    x = tf.layers.conv2d(x, filters=filter_size, kernel_size=3, kernel_initializer=wt_init, trainable=is_train,
                     kernel_regularizer=weight_regularizer, name="conv"+conv_number+"_"+instance, padding="SAME",
reuse=reuse)

    x = tf_contrib.layers.batch_norm(x, decay=DECAY_BN, epsilon=EPSILON_BN, center=True, scale=True,
trainable=is_train, reuse=reuse, scope="batch_norm"+conv_number+"_"+instance)

    if(stride == 2):
        #x = prev + x
        prev = tf.layers.conv2d(prev, filters=filter_size, kernel_size=1, kernel_initializer=wt_init, strides=stride,
                         kernel_regularizer=weight_regularizer, padding="SAME", trainable=is_train,
name="1x1_"+conv_number+"_"+instance, reuse=reuse)

    x = prev + x
    x = tf.nn.relu(x)

    return x

def build_model(train_input, is_train = False, reuse = False):
    wt_init = tf_contrib.layers.xavier_initializer()
```

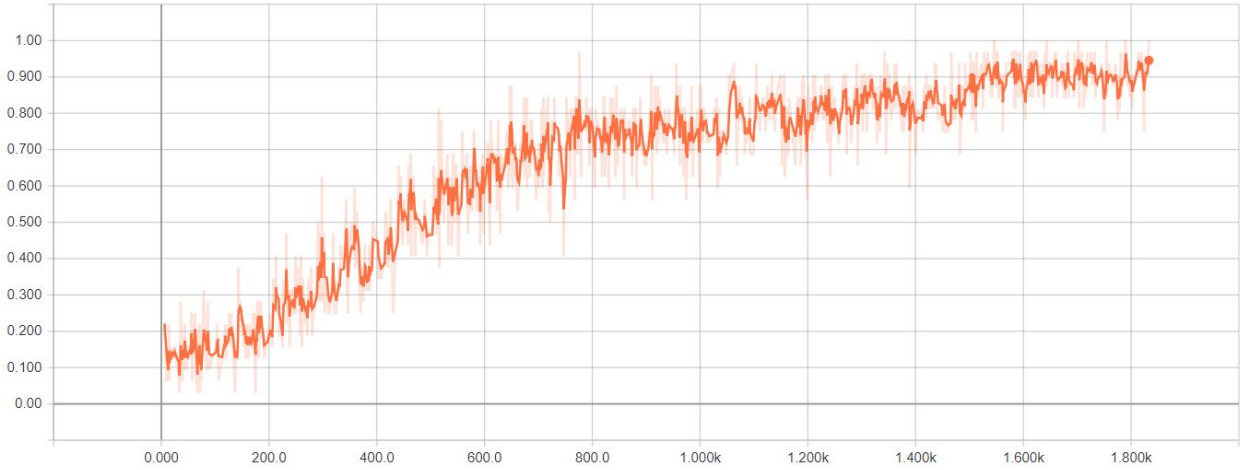
```
    weight_regularizer = tf_contrib.layers.l2_regularizer(0.0001)

    x = tf.layers.conv2d(train_input, filters=64, kernel_size=7, kernel_initializer=wt_init, strides=2,
                         kernel_regularizer=weight_regularizer, trainable=is_train, name="conv1", padding='SAME',
reuse=reuse)
    x = tf_contrib.layers.batch_norm(x, decay=DECAY_BN, epsilon=EPSILON_BN, center=True, scale=True,
trainable=is_train, reuse=reuse, scope="batch_norm1")
    x = tf.nn.relu(x)

    x = tf.layers.max_pooling2d(x, pool_size = 2, strides = 2, name="maxpool") # 56x56x64

    x = residual_block(x, 64, wt_init, weight_regularizer, "2", "1", is_train, reuse=reuse)
    x = residual_block(x, 64, wt_init, weight_regularizer, "2", "2", is_train, reuse=reuse)

    x = residual_block(x, 128, wt_init, weight_regularizer, "3", "1", is_train, stride=2, reuse=reuse)
    x = residual_block(x, 128, wt_init, weight_regularizer, "3", "2", is_train, reuse=reuse)

    x = residual_block(x, 256, wt_init, weight_regularizer, "4", "1", is_train, stride=2, reuse=reuse)
    x = residual_block(x, 256, wt_init, weight_regularizer, "4", "2", is_train, reuse=reuse)

    x = residual_block(x, 512, wt_init, weight_regularizer, "5", "1", is_train, stride=2, reuse=reuse)
    x = residual_block(x, 512, wt_init, weight_regularizer, "5", "2", is_train, reuse=reuse)

    x = tf.layers.average_pooling2d(x, pool_size=7, strides=2, name="avgpool")
    x = tf.contrib.layers.flatten(x)

    x = tf.layers.dense(x, 8, kernel_initializer=wt_init, trainable=is_train, name="output", reuse=reuse)
    #x = tf.nn.softmax(x, name="output_logits")

    return x

def get_losses(output, label):
    loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(logits=output, labels=label))
    prediction = tf.equal(tf.argmax(output, -1), tf.argmax(label, -1))
    accuracy = tf.reduce_mean(tf.cast(prediction, tf.float32))

    return loss, accuracy

tf.reset_default_graph()
train_input = tf.placeholder(tf.float32, (None, 224, 224, 3), name = "train_input")
train_label = tf.placeholder(tf.int32, (None, 8), name = "train_label")
test_input = tf.placeholder(tf.float32, (None, 224, 224, 3), name = "test_input")
test_label = tf.placeholder(tf.int32, (None, 8), name = "test_label")

batch = 32

lr = tf.placeholder(tf.float32, name='learning_rate')
train_output = build_model(train_input, is_train = True)
test_output = build_model(test_input, reuse = True)

train_loss, train_acc = get_losses(train_output, train_label)
test_loss, test_acc = get_losses(test_output, test_label)

optimizer = tf.train.MomentumOptimizer(learning_rate=lr, momentum=0.9).minimize(train_loss)

#### Summary #####
summary_train_loss = tf.summary.scalar("train_loss", train_loss)
summary_test_loss = tf.summary.scalar("test_loss", test_loss)

summary_train_accuracy = tf.summary.scalar("train_accuracy", train_acc)
summary_test_accuracy = tf.summary.scalar("test_accuracy", test_acc)
```

```
train_summary = tf.summary.merge([summary_train_loss, summary_train_accuracy])
test_summary = tf.summary.merge([summary_test_loss, summary_test_accuracy])

#get_num_trainable_params()

epochs = 50
epoch_lr = 0.1
counter = 0
test_counter = 0

sess = tf.Session()
sess.run(tf.global_variables_initializer())

checkpoint_dir = "./checkpoints"
writer = tf.summary.FileWriter("./logs", sess.graph)
saver = tf.train.Saver()

print("Starting Train!!!")
start_time = time.time()


test_accs = []
test_losses = []
for epoch in range(epochs):
    if epoch == int(epochs * 0.5) or epoch == int(epochs * 0.75):
        epoch_lr = epoch_lr * 0.1
    minibatches, n_batches = get_mini_batches(train_images, train_labels, batch_size = batch)
    i = 0
    for minibatch in minibatches:
        i += 1
        X, y = minibatch

        train_feed_dict = {train_input:X, train_label: y, lr: epoch_lr}

        acc, _, l, summary1 = sess.run([train_acc, optimizer, train_loss, train_summary], feed_dict=train_feed_dict)
        writer.add_summary(summary1, counter)
        counter += 1

        print("Epoch: [%2d] [%5d/%5d], train_accuracy: %.2f, learning_rate : %.4f, train_loss: %.2f" \
                    % (epoch, i, n_batches, acc, epoch_lr, l))
    if(epoch%5 == 0):
        if not os.path.exists(checkpoint_dir):
            os.makedirs(checkpoint_dir)
        saver.save(sess, os.path.join(checkpoint_dir, 'ResNet18.model'), global_step=counter)

    minibatches, n_batches = get_mini_batches(test_images, test_labels, batch_size = batch, crop=False)
    for minibatch in minibatches:
        X, y = minibatch

        test_feed_dict = {test_input:X, test_label: y}

        t_acc, summary2, t_loss = sess.run([test_acc, test_summary, test_loss], feed_dict=test_feed_dict)

        test_accs.append(t_acc)
        test_losses.append(t_loss)
        writer.add_summary(summary2, test_counter)
        print("Epoch: [%2d], test_accuracy: %.2f, test_loss: %.2f"%(epoch, t_acc, t_loss))
        test_counter += 1

sess.close()
```

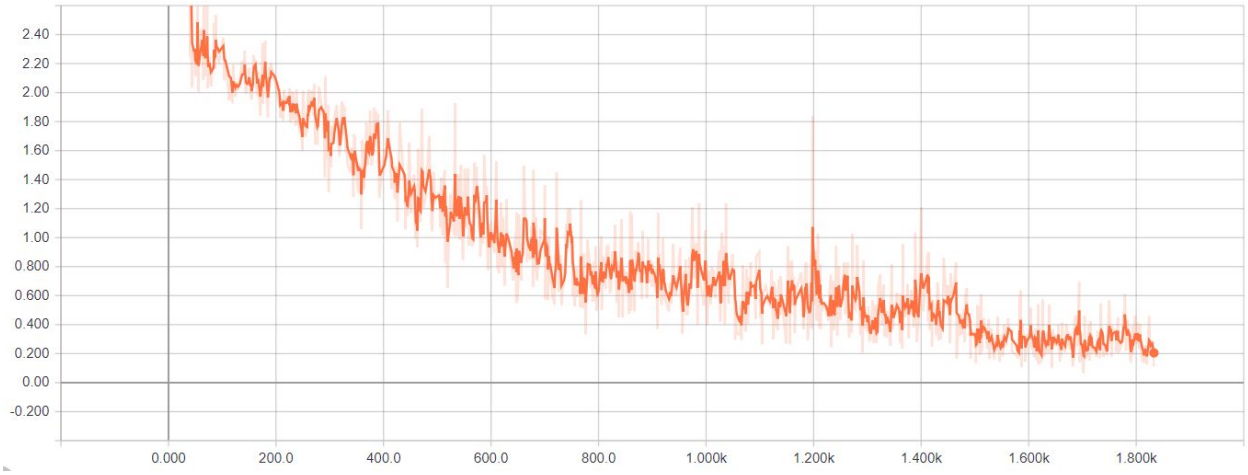
```
num_test_batches = test_images.shape[0]//batch

for i in range(len(test_accs)//num_test_batches):
    print(np.mean(test_accs[i*num_test_batches: (i+1)*num_test_batches]))
```

Observations:

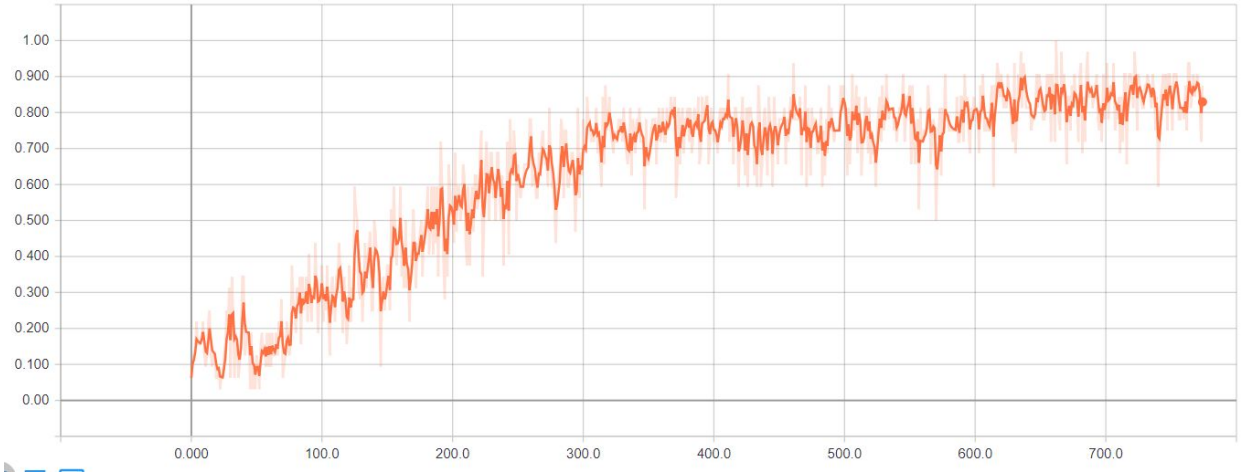
TRAINING ACCURACY



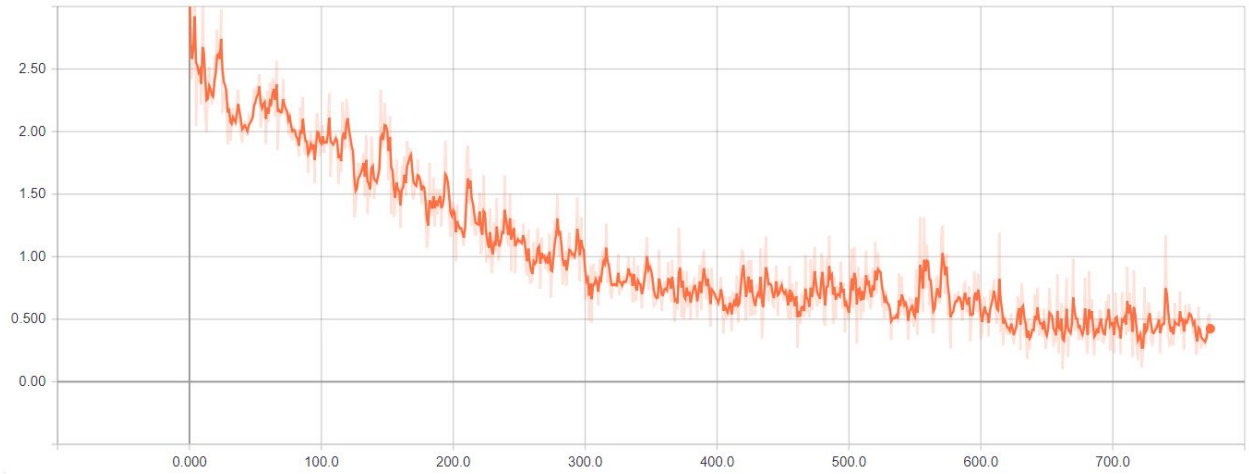
TRAINING LOSS



TEST ACCURACY



TEST LOSS



Final train accuracy: 0.9454
Final test accuracy: 0.8425

Final Accuracies for different models on test set:

Bag Of Visual Words with KNN	**0.54**
AlexNet (Refined)	**0.8972**
ResNet18	**0.8425**