CSC420

Assignment 5

Zhili Xu

Question 1:

a)

No, 5 images are not enough to train neural network. Usually, it needs more than 1000 images from each class.

b)

Tf-idf looks at a normalized count where each word count is divided by the number of documents this word appears in. Tf-idf makes rare words more prominent and effectively ignores common words.

c)

```
if __name__ == "__main__":
    load_dataset()
    a = np.load('faces.npy')
    model = load_facenet()
    f = open('embedding.txt', 'w')
    # print ('here')
    for i in range(a.shape[0]):
        img = a[i][..., ::-1]
        img = cv2.resize(img, (96, 96))
        # img = cv2.imread("saved_faces/face_image_"+str(i)+".jpg")
        # print (img_to_encoding(img, model))
        f.write("saved_faces/face_image_"+str(i)+".jpg\n")
        f.write(str(img_to_encoding(img, model)))
        f.write('\n')
```

Image embedding is store in ‘embedding.txt’.

d)

Clustering does not only make semantic image retrieval straightforward, but also enables the application of a wide variety of existing methods that rely on metric feature spaces

e) and f)

using the scikit-learn function to create 6 clusters. Then use dictionary to store their labels corresponding to their filename.

```
if __name__ == "__main__":
    # images = [image1]
    model = load_facenet()
    #q1e
```

```
    data = np.load('embedding.npy')
    kmeans = KMeans(n_clusters=6, random_state=0).fit(data)
    center = kmeans.cluster_centers_
    labels = kmeans.labels_

    #q1f
    index = 0
    d = {}
    for i in labels:
        name = 'saved_faces/face_image_'+str(index)+'.jpg'
        d[name] = i
        index += 1
```

g)

Use the similarity function to compare input image embedding with cluster centers, find the maximum value of with all 6 cluster. If the value is larger than the threshold, then the image is belong to that cluster and label it with the same label.

h)

The output is write to ‘match.txt’ file

```
if __name__ == "__main__":
    name1 = 'input_faces/face_image_97.jpg'
    name2 = 'input_faces/face_image_107.jpg'
    name3 = 'input_faces/face_image_109.jpg'
    name4 = 'input_faces/face_image_116.jpg'
    name5 = 'input_faces/face_image_119.jpg'
    name6 = 'input_faces/face_image_126.jpg'
    name7 = 'input_faces/face_image_600.jpg'

    image1 = cv2.imread('input_faces/face_image_97.jpg')
    image2 = cv2.imread('input_faces/face_image_107.jpg')
    image3 = cv2.imread('input_faces/face_image_109.jpg')
    image4 = cv2.imread('input_faces/face_image_116.jpg')
    image5 = cv2.imread('input_faces/face_image_119.jpg')
    image6 = cv2.imread('input_faces/face_image_126.jpg')
    image7 = cv2.imread('input_faces/face_image_600.jpg')
    images = [image1, image2,image3,image4,image5,image6,image7]
    names = [name1, name2, name3,name4,name5,name6,name7]
    # images = [image1]
    model = load_facenet()
```

```
data = np.load('embedding.npy')
kmeans = KMeans(n_clusters=6, random_state=0).fit(data)
center = kmeans.cluster_centers_
labels = kmeans.labels_

index = 0
d = {}
for i in labels:
    name = 'saved_faces/face_image_'+str(index)+'.jpg'
    d[name] = i
    index += 1

index = 0
f = open('matches.txt', 'w')
for im in images:
    img = cv2.resize(im, (96, 96))
    embedding = img_to_encoding(img, model)
    temp = []
    for c in center:
        sim = np.dot(embedding, c)/(np.linalg.norm(c)*np.linalg.norm(embedding))
        temp.append(sim)

    if (max(temp) > 0.8):
        cluster = temp.index(max(temp))
    else:
        cluster = None

    string = names[index] + ' ' + 'matches with ['
    for key,value in d.items():
        if value == cluster:
            string += key + ' '
    f.write(string)
    f.write(']\n\n')
    index+=1
f.close()
```

i)

In order to use clustering and face embedding, the challenge with an image that contains multiple people is that we need to separate single image into different cluster in order to identify the person. In cases which people are close to each other or a person face is partially blocked, then it is hard to identity that person. A more general way is to each person's face data then use object detect to find the person.

Question2:

a)

This problem makes it really hard to learn and tune the parameters of the earlier layers in the network. This problem becomes worse as the number of layers in the architecture increases.

We can avoid this problem by using activation functions which don't have this property of 'squashing' the input space into a small region. A popular choice is Rectified Linear Unit which maps xx to max(0,x)max(0,x).

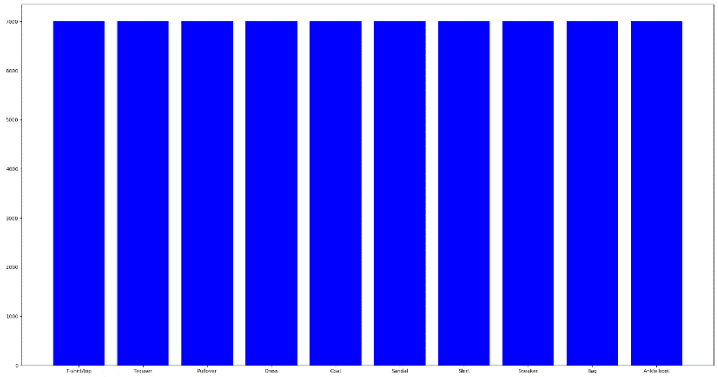
b)

Reduces the number of parameters within the model

It generalises the results from a convolutional filter - making the detection of features invariant to scale or orientation changes.

c)

```
unique, counts = np.unique(train_labels, return_counts=True)
d = dict(zip(unique, counts))
unique, counts = np.unique(test_labels, return_counts=True)
e = dict(zip(unique, counts))
for key,value in e.items():
    if key in d:
        d[key] += value
plt.bar(list(d.keys()),list(d.values()), color='b')
t1 = np.arange(-1, 10, 1)
plt.xticks(range(len(class_names)),class_names)
plt.show()
```



d)

```
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(128, activation=tf.nn.relu),
    keras.layers.Dense(10, activation=tf.nn.softmax)
])

model.compile(optimizer=tf.train.AdamOptimizer(),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

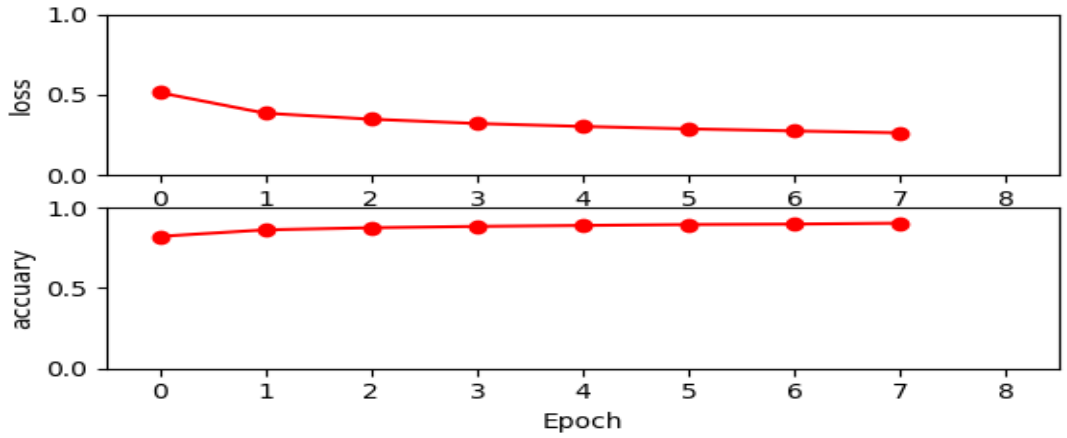
e)

f)

```
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(128, activation=tf.nn.relu),
    keras.layers.Dense(10, activation=tf.nn.softmax)
])

model.compile(optimizer=tf.train.AdamOptimizer(),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

history = model.fit(train_images, train_labels, epochs=8, batch_size = 64)

loss = list(history.history['loss'])
plt.figure(1)
plt.subplot(311)
plt.plot(range(len(loss)), loss, 'ro-')
plt.axis([-0.5, 8.5, 0.0, 1.0])
plt.xlabel('Epoch')
plt.ylabel('loss')

acc = list(history.history['acc'])
plt.subplot(312)
plt.plot(range(len(acc)), acc, 'ro-')
plt.axis([-0.5, 8.5, 0.0, 1.0])
plt.xlabel('Epoch')
plt.ylabel('accuary')
```

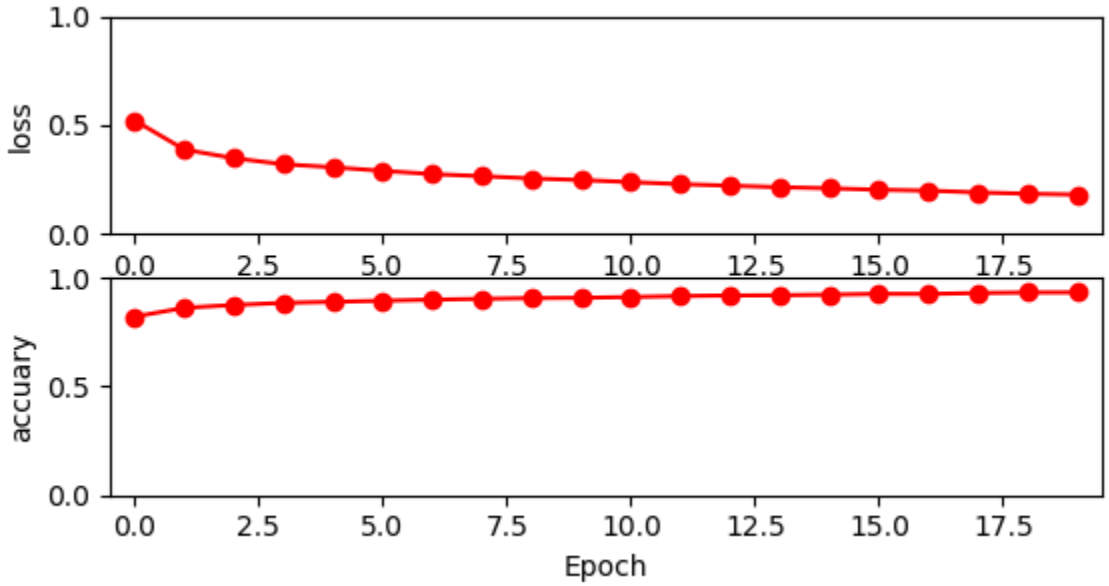
Using 8 epoch and 64 batch, the final loss and accuracy is:

- 1s 15us/step - loss: 0.2640 - acc: 0.9032



Using 20 epoch and 64 batch, the final loss and accuracy is:

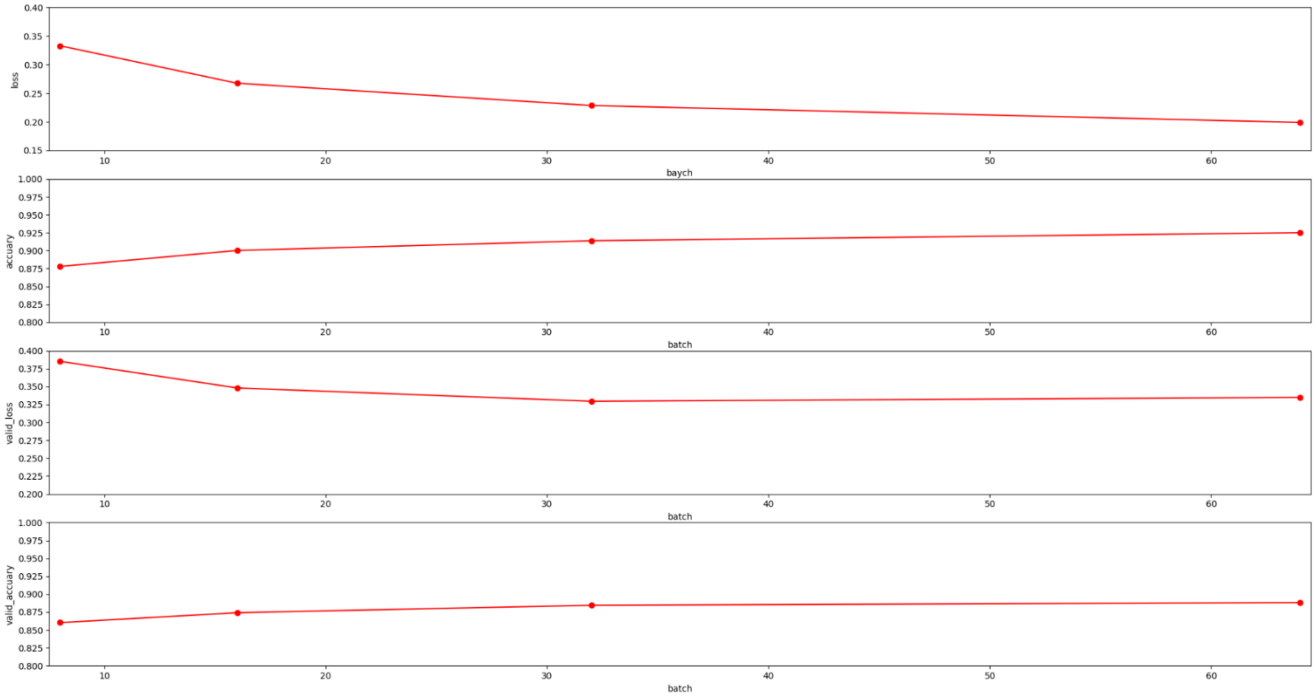
1s 15us/step - loss: 0.1812 - acc: 0.9320



The loss and accuracy functions start to plateau out, so we can stop training.

g)

```
batch = [8,16,32,64]
train_loss=[]
train_acc=[]
valid_loss=[]
valid_acc=[]
for i in batch:
    history = model.fit(train_images, train_labels, epochs=5, batch_size = i)
    test_loss, test_acc = model.evaluate(test_images, test_labels)
    train_loss.append(list(history.history['loss'])[-1])
    train_acc.append(list(history.history['acc'])[-1])
    valid_loss.append(test_loss)
    valid_acc.append(test_acc)


plt.figure(1)
plt.subplot(211)
plt.plot(batch, train_loss, 'ro-')
plt.axis([7.5, 64.5, 0.15, 0.4])
plt.xlabel('baych')
plt.ylabel('loss')

plt.subplot(212)
plt.plot(batch, train_acc, 'ro-')
plt.axis([7.5, 64.5, 0.8, 1.0])
plt.xlabel('batch')
plt.ylabel('accuary')


plt.subplot(413)
plt.plot(batch, valid_loss, 'ro-')
plt.axis([7.5, 64.5, 0.2, 0.4])
plt.xlabel('batch')
plt.ylabel('valid_loss')

plt.subplot(414)
plt.plot(batch, valid_acc, 'ro-')
plt.axis([7.5, 64.5, 0.8, 1.0])
plt.xlabel('batch')
plt.ylabel('valid_accuary')
```


loss
baych
accuary
batch
valid_loss
batch
valid_accuary
batch