Coursework 2: Neural Networks

This coursework covers the topics covered in class regarding neural networks for image classification.

This coursework includes both coding questions as well as written ones. Please upload the notebook, which contains your code, results and answers as a pdf file onto Cate.

Dependencies: If you work on a college computer in the Computing Lab, where Ubuntu 18.04 is installed by default, you can use the following virtual environment for your work, where relevant Python packages are already installed.

```
source /vol/bitbucket/wbai/virt/computer_vision_ubuntu18.04/bin/activate
```

Alternatively, you can use pip, pip3 or anaconda etc to install Python packages.

Note 1: please read the both the text and code comment in this notebook to get an idea what you are supposed to implement.

Note 2: If you are using the virtual environment in the Computing Lab, please run the following command in the command line before opening jupyter-notebook and importing tensorflow. This will tell tensorflow where the Nvidia CUDA libariries are.

```
export LD_LIBRARY_PATH=/vol/cuda/9.0.176/lib64/:"${LD_LIBRARY_PATH}}"
```

In [1]:
```
# Import libraries
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import tensorflow as tf
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout
```

```
Using TensorFlow backend.
```

Question 1 (20 points)

Throughout this coursework you will be working with the Fashion-MNIST dataset. If you are interested, you may find relevant information regarding the dataset in this paper.

[1] Fashion-MNIST: A novel image dataset for benchmarking machine learning algorithms. Han Xiao, Kashif Rasul, Roland Vollgraf. arXiv:1708.07747 (https://arxiv.org/abs/1708.07747)

Be sure that you have the following files in your working directory: data.tar.gz and reader.py. Loading the data can be done as follows:

```
from reader import get_images
(x_train, y_train), (x_test, y_test) = get_images()
```

The dataset is already split into a set of 60,000 training images and a set of 10,000 test images. The images are of size 28x28 pixels and stored as 784-D vector. So if you would like to visualise the images, you need to reshape the array.

There are in total 10 label classes, which are:

- 0: T-shirt/top
- 1: Trousers
- 2: Pullover
- 3: Dress
- 4: Coat
- 5: Sandal
- 6: Shirt
- 7: Sneaker
- 8: Bag
- 9: Ankle boot

1.1 Load data (6 points)

Load the dataset and print the dimensions of the training set and the test set.

In [12]:
```
from reader import get_images
(x_train, y_train), (x_test, y_test) = get_images()
print('dimension of x_train_set:' + str(x_train.shape) + '\ndimension of y_train_set:' + str(y_train.shape))
print('\ndimension of x_test_set:' + str(x_test.shape) + '\ndimension of y_test_set:' + str(y_test.shape))
```

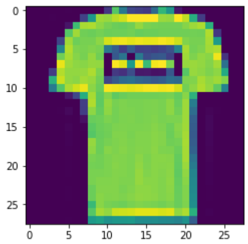
```
dimension of x_train_set:(60000, 784)
dimension of y_train_set:(60000,)

dimension of x_test_set:(10000, 784)
dimension of y_test_set:(10000,)
```

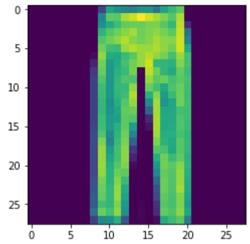
1.2 Visualize data (6 points)

Visualise 3 training images (T-shirt, trousers and pullover) and 3 test images (dress, coat and sandal).

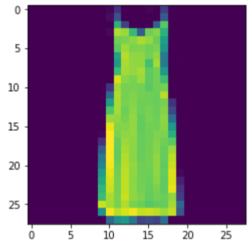
In [6]:
```
imgplot_train1 = plt.imshow(np.reshape(x_train[1],(28,28)))
```



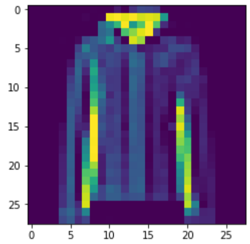
In [7]:
```
imgplot_train2 = plt.imshow(np.reshape(x_train[16],(28,28)))
```



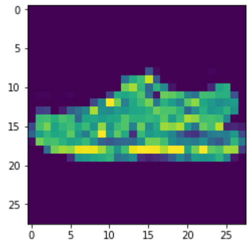
In [8]:
```
imgplot_test1 = plt.imshow(np.reshape(x_test[13],(28,28)))
```



In [9]:
```
imgplot_test2 = plt.imshow(np.reshape(x_test[6],(28,28)))
```



In [10]:
```
imgplot_test3 = plt.imshow(np.reshape(x_test[21],(28,28)))
```



1.3 Data balance (4 points)

Print out the number of training samples for each class.

In [13]:
```
unique, counts = np.unique(y_train, return_counts=True)
dict(zip(unique, counts))
```

Out[13]:
```
{0: 6000,
 1: 6000,
 2: 6000,
 3: 6000,
 4: 6000,
 5: 6000,
 6: 6000,
 7: 6000,
 8: 6000,
 9: 6000}
```

1.4 Discussion (4 points)

Is the dataset balanced? What would happen if the dataset is not balanced in the context of image classification?

In [ ]:
```
# Yes, this dataset is balanced, because the number of each class is same.
#
# If the dataset is not balanced, and assume that our training set has 1000 samples for dress and only 20 samples for
# trousers, then in this case, our algorithm will tends to classify all new examples as dress.
```

Question 2 (40 points)

Build a neural network and train it with the Fashion-MNIST dataset. Here, we use the keras library, which is a high-level neural network library built upon tensorflow.

In [16]:
```
# Convert the label class into a one-hot representation
num_classes = 10
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

2.1 Build a multi-layer perceptron, also known as multi-layer fully connected network. You need to define the layers, the loss function, the optimiser and evaluation metric. (30 points)

In [4]:
```
from keras.optimizers import SGD
opt = SGD(lr=0.01)
model = keras.models.Sequential()
M = 128 # hidden units
K = 10 # outputs

model.add(Dense(784, input_dim = 784,kernel_initializer= "uniform",
  bias_initializer= "zeros",activation= "relu"))
model.add(Dropout(0.3))

model.add(Dense(500,kernel_initializer= "uniform", bias_initializer=
  "zeros",activation= "relu"))
model.add(Dropout(0.3))

model.add(Dense(200,kernel_initializer= "uniform", bias_initializer=
  "zeros",activation= "relu"))
model.add(Dropout(0.3))

model.add(Dense(K,kernel_initializer= "uniform", bias_initializer=
   "zeros",activation= "softmax"))

model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
print(model.summary())
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_1 (Dense)              (None, 784)               615440    
_________________________________________________________________
dropout_1 (Dropout)          (None, 784)               0         
_________________________________________________________________
dense_2 (Dense)              (None, 500)               392500    
_________________________________________________________________
dropout_2 (Dropout)          (None, 500)               0         
_________________________________________________________________
dense_3 (Dense)              (None, 200)               100200    
_________________________________________________________________
dropout_3 (Dropout)          (None, 200)               0         
_________________________________________________________________
dense_4 (Dense)              (None, 10)                2010      
=================================================================
Total params: 1,110,150
Trainable params: 1,110,150
Non-trainable params: 0
_________________________________________________________________
None
```

2.2 Define the optimisation parameters including the batch size and the number of epochs and then run the optimiser. (10 points)

We have tested that for an appropriate network architecture, on a personal laptop and with only CPU, it takes about a few seconds per epoch to train the network. For 100 epochs, it takes about a coffee break's time to finish the training. If you run it on a powerful GPU, it would be even much faster.

In [5]:
```
batch_size = 200
epochs = 100
model.fit(x_train, y_train, batch_size, epochs )
```

```
Epoch 1/100
60000/60000 [==============================] - 9s 153us/step - loss: 2.0073 - acc: 0.6497
Epoch 2/100
60000/60000 [==============================] - 5s 89us/step - loss: 0.5815 - acc: 0.7884
Epoch 3/100
60000/60000 [==============================] - 6s 92us/step - loss: 0.5099 - acc: 0.8168
Epoch 4/100
60000/60000 [==============================] - 5s 91us/step - loss: 0.4773 - acc: 0.8274
Epoch 5/100
60000/60000 [==============================] - 5s 90us/step - loss: 0.4491 - acc: 0.8373
Epoch 6/100
60000/60000 [==============================] - 5s 90us/step - loss: 0.4310 - acc: 0.8443
Epoch 7/100
60000/60000 [==============================] - 5s 91us/step - loss: 0.4143 - acc: 0.8496
Epoch 8/100
60000/60000 [==============================] - 6s 94us/step - loss: 0.4002 - acc: 0.8552
Epoch 9/100
60000/60000 [==============================] - 5s 91us/step - loss: 0.3937 - acc: 0.8553
Epoch 10/100
60000/60000 [==============================] - 5s 90us/step - loss: 0.3814 - acc: 0.8600
Epoch 11/100
60000/60000 [==============================] - 5s 90us/step - loss: 0.3717 - acc: 0.8635
Epoch 12/100
60000/60000 [==============================] - 6s 92us/step - loss: 0.3668 - acc: 0.8660
Epoch 13/100
60000/60000 [==============================] - 5s 91us/step - loss: 0.3560 - acc: 0.8697
Epoch 14/100
60000/60000 [==============================] - 5s 91us/step - loss: 0.3532 - acc: 0.8705
Epoch 15/100
60000/60000 [==============================] - 5s 89us/step - loss: 0.3458 - acc: 0.8742
Epoch 16/100
60000/60000 [==============================] - 5s 91us/step - loss: 0.3417 - acc: 0.8757
Epoch 17/100
60000/60000 [==============================] - 6s 93us/step - loss: 0.3360 - acc: 0.8771
Epoch 18/100
60000/60000 [==============================] - 6s 92us/step - loss: 0.3295 - acc: 0.8796
Epoch 19/100
60000/60000 [==============================] - 5s 86us/step - loss: 0.3229 - acc: 0.8808
Epoch 20/100
60000/60000 [==============================] - 5s 91us/step - loss: 0.3210 - acc: 0.8811
Epoch 21/100
60000/60000 [==============================] - 6s 94us/step - loss: 0.3128 - acc: 0.8834
Epoch 22/100
60000/60000 [==============================] - 5s 90us/step - loss: 0.3114 - acc: 0.8849
Epoch 23/100
60000/60000 [==============================] - 5s 90us/step - loss: 0.3047 - acc: 0.8871
Epoch 24/100
60000/60000 [==============================] - 5s 91us/step - loss: 0.3009 - acc: 0.8886
Epoch 25/100
60000/60000 [==============================] - 5s 91us/step - loss: 0.3001 - acc: 0.8885
Epoch 26/100
60000/60000 [==============================] - 5s 88us/step - loss: 0.2957 - acc: 0.8907
Epoch 27/100
60000/60000 [==============================] - 6s 93us/step - loss: 0.2917 - acc: 0.8912
Epoch 28/100
60000/60000 [==============================] - 6s 94us/step - loss: 0.2872 - acc: 0.8934
Epoch 29/100
60000/60000 [==============================] - 6s 92us/step - loss: 0.2863 - acc: 0.8939
Epoch 30/100
60000/60000 [==============================] - 5s 91us/step - loss: 0.2798 - acc: 0.8950
Epoch 31/100
60000/60000 [==============================] - 6s 93us/step - loss: 0.2787 - acc: 0.8963
Epoch 32/100
60000/60000 [==============================] - 6s 95us/step - loss: 0.2741 - acc: 0.8968
Epoch 33/100
60000/60000 [==============================] - 6s 96us/step - loss: 0.2710 - acc: 0.8993
Epoch 34/100
60000/60000 [==============================] - 6s 100us/step - loss: 0.2707 - acc: 0.8994
Epoch 35/100
60000/60000 [==============================] - 6s 95us/step - loss: 0.2663 - acc: 0.8996
Epoch 36/100
60000/60000 [==============================] - 6s 98us/step - loss: 0.2657 - acc: 0.8992
Epoch 37/100
60000/60000 [==============================] - 6s 99us/step - loss: 0.2609 - acc: 0.9020
Epoch 38/100
60000/60000 [==============================] - 6s 92us/step - loss: 0.2602 - acc: 0.9016
Epoch 39/100
60000/60000 [==============================] - 6s 93us/step - loss: 0.2565 - acc: 0.9033
Epoch 40/100
60000/60000 [==============================] - 6s 92us/step - loss: 0.2541 - acc: 0.9043
Epoch 41/100
60000/60000 [==============================] - 5s 88us/step - loss: 0.2525 - acc: 0.9046
Epoch 42/100
60000/60000 [==============================] - 5s 89us/step - loss: 0.2501 - acc: 0.9056
Epoch 43/100
60000/60000 [==============================] - 5s 90us/step - loss: 0.2483 - acc: 0.9060
Epoch 44/100
60000/60000 [==============================] - 5s 91us/step - loss: 0.2443 - acc: 0.9079
Epoch 45/100
60000/60000 [==============================] - 5s 91us/step - loss: 0.2414 - acc: 0.9080
Epoch 46/100
60000/60000 [==============================] - 5s 87us/step - loss: 0.2384 - acc: 0.9093
Epoch 47/100
60000/60000 [==============================] - 5s 87us/step - loss: 0.2382 - acc: 0.9090
Epoch 48/100
60000/60000 [==============================] - 5s 87us/step - loss: 0.2348 - acc: 0.9114
Epoch 49/100
60000/60000 [==============================] - 5s 87us/step - loss: 0.2331 - acc: 0.9116
Epoch 50/100
60000/60000 [==============================] - 5s 88us/step - loss: 0.2352 - acc: 0.9100
Epoch 51/100
60000/60000 [==============================] - 5s 87us/step - loss: 0.2297 - acc: 0.9115
Epoch 52/100
60000/60000 [==============================] - 5s 87us/step - loss: 0.2272 - acc: 0.9131
Epoch 53/100
60000/60000 [==============================] - 6s 92us/step - loss: 0.2241 - acc: 0.9149
Epoch 54/100
60000/60000 [==============================] - 5s 90us/step - loss: 0.2231 - acc: 0.9149
Epoch 55/100
60000/60000 [==============================] - 6s 92us/step - loss: 0.2228 - acc: 0.9150
Epoch 56/100
```

Out[5]: <keras.callbacks.History at 0x7f09c81af128>

Question 3 (20 points)

Evaluate the performance of your network with the test data. Visualize the performance using appropriate metrics and graphs (eg. confusion matrix). Comment on your per class performance and how it could be better.

In [14]:

```
# This function is provided for you to display the confusion matrix.
# For more information about the confusion matrix, you can read at
# https://en.wikipedia.org/wiki/Confusion_matrix
def plot_confusion_matrix(cm, classes, normalize=False, title='Confusion matrix', cmap=plt.cm.Blues):
    """
        This function prints and plots the confusion matrix.

        cm: confusion matrix, default to be np.int32 data type
        classes: a list of the class labels or class names
        normalize: normalize the matrix so that each row amounts to one
        cmap: color map
    """
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')
    print(cm)

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.tight_layout()
```

3.1 Evaluate the classification accuracy on the test set (10 points)

In [18]:

```
scores = model.evaluate(x_test, y_test)
print('Test loss:', scores[0])
print('Test accuracy:', scores[1])
```

```
10000/10000 [==============================] - 0s 47us/step
Test loss: 0.32695946246385577
Test accuracy: 0.8948
```

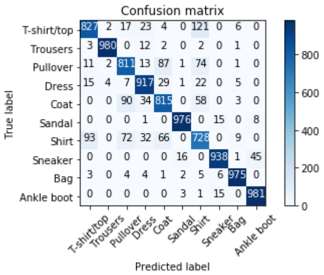
3.2 Calculate and plot the confusion matrix (10 points)

In [19]:

```
from sklearn.metrics import confusion_matrix
(Y_train, Y_train), (X_test, Y_test) = get_images()
Y_pred = model.predict_classes(x_test, batch_size=128)
```

In [20]:
```python
import itertools
cm = confusion_matrix(Y_test, Y_pred)
print('cm is:' + str(cm))
classes = ["T-shirt/top","Trousers","Pullover","Dress","Coat","Sandal","Shirt","Sneaker","Bag","Ankle boot"]
plot_confusion_matrix(cm, classes)
```

```
cm is:[[827   2  17  23   4   0 121   0   6   0]
 [  3 980   0  12   2   0   2   0   1   0]
 [ 11   2 811  13  87   1  74   0   1   0]
 [ 15   4   7 917  29   1  22   0   5   0]
 [  0   0  90  34 815   0  58   0   3   0]
 [  0   0   0   1   0 976   0  15   0   8]
 [ 93   0  72  32  66   0 728   0   9   0]
 [  0   0   0   0   0  16   0 938   1  45]
 [  3   0   4   4   1   2   5   6 975   0]
 [  0   0   0   0   0   3   1  15   0 981]]
Confusion matrix, without normalization
[[827   2  17  23   4   0 121   0   6   0]
 [  3 980   0  12   2   0   2   0   1   0]
 [ 11   2 811  13  87   1  74   0   1   0]
 [ 15   4   7 917  29   1  22   0   5   0]
 [  0   0  90  34 815   0  58   0   3   0]
 [  0   0   0   1   0 976   0  15   0   8]
 [ 93   0  72  32  66   0 728   0   9   0]
 [  0   0   0   0   0  16   0 938   1  45]
 [  3   0   4   4   1   2   5   6 975   0]
 [  0   0   0   0   0   3   1  15   0 981]]
```



Question 4 (20 points)

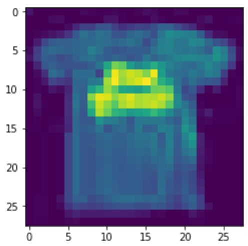
Take two photos, one of your clothes or shoes that belongs to one of 10 classes, the other that does not belong to any class.

Use either Python or other software (Photoshop, Gimp, or any image editer) to convert the photos into grayscale, crop the region of interest and reshape into the size of 28x28.

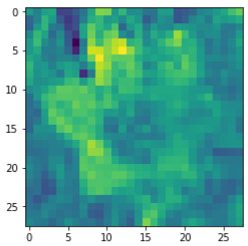
4.1 Load and visualise your own images (6 points)

In [31]:
```python
import matplotlib.image as mpimg
img1=mpimg.imread('T-shirt.jpg')
print(img1.shape)
imgplot = plt.imshow(img1)
```

```
(28, 28)
```



In [22]:
```python
img2=mpimg.imread('flower.jpg')
imgplot = plt.imshow(img2)
```



4.2 Test your network on the two images and show the classification results (10 points)

In [33]:
```python
img11 = np.reshape(img1,(1,784))
image1_pred = model.predict_classes(img11, batch_size=128)
print('predicted result of T-shirt is:' + classes[int(image1_pred)])
```

```
predicted result of T-shirt is:T-shirt/top
```

In [25]:
```
img22 = np.reshape(img2,(1,784))
image2_pred = model.predict_classes(img22, batch_size=128)
print('predicted result of flower is:' + classes[int(image2_pred)])
```

```
predicted result of flower is:Bag
```

4.3 Discuss the classification results and provide one method to improve real life performance of the network (4 points)

In [ ]:
```
# In our real life, there are lots of ways to improve the performance
#
# 1. Data:The first thing we can do to improve real life performance is to increase the size of training sets,
# we can simplely get more data, or we can invent more data, which is called data augmentation; also, we
# can rescale or transform our data, or maybe we can do the feature selection before training it, which makes it faster.
# 2. Algorithm:We can improve our algorithms, a better algorithm may improve our network's performance. And we can also
# simply tune our algotiyhms.
```

5. Survey

How long did the coursework take you to solve?

In [ ]:
```
# 4-5 hours.
```