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 [120]:

```
# 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
from keras.layers import Activation
import math
from reader import get_images
from keras.optimizers import SGD
from sklearn.metrics import confusion_matrix
import itertools
```

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 [121]:

```
(x_train, y_train), (x_test, y_test) = get_images()
print(x_train.dtype,x_test.dtype)
print("x_train shape:", x_train.shape, "y_train shape:", y_train.shape)
print("x_test shape:", x_test.shape, "y_test shape:", y_test.shape)
```

```
uint8 uint8
x_train shape: (60000, 784) y_train shape: (60000,)
x_test shape: (10000, 784) y_test shape: (10000,)
```

1.2 Visualize data (6 points)

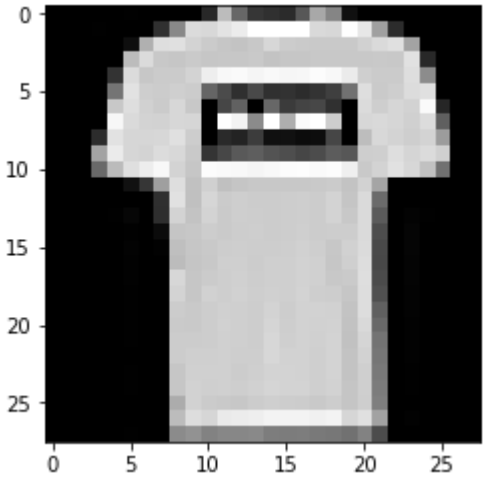
Visualise 3 training images (T-shirt, trousers and pullover) and 3 test images (dress, coat and sandal).

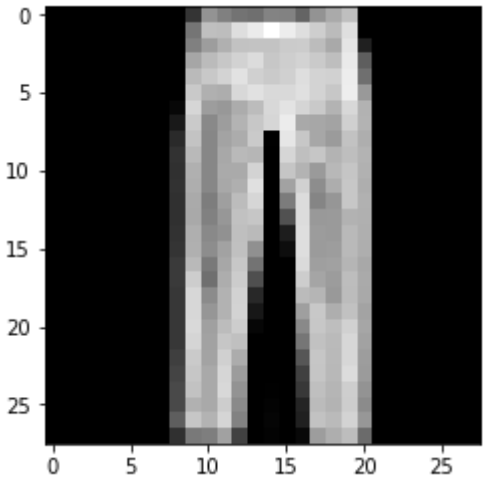
In [122]:

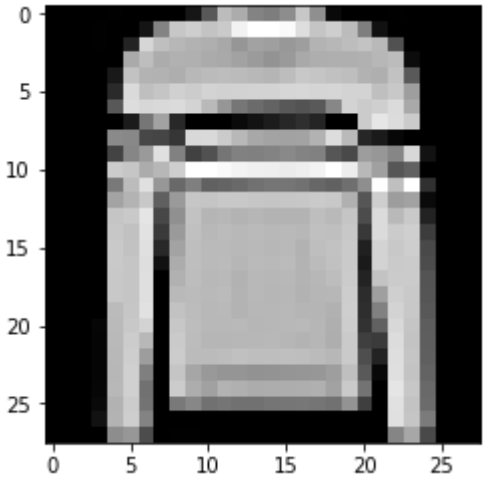
```
n_tshirt = np.argwhere(y_train == 0)[0]
n_trousers = np.argwhere(y_train == 1)[0]
n_pullover = np.argwhere(y_train == 2)[0]

img_tshirt = x_train[n_tshirt]
plt.imshow(np.reshape(img_tshirt,(28,28)), cmap="gray")
plt.figure()
img_trousers = x_train[n_trousers]
plt.imshow(np.reshape(img_trousers ,(28,28)),cmap="gray")
plt.figure()

img_pullovr = x_train[n_pullover]
plt.imshow(np.reshape(img_pullovr ,(28,28)),cmap="gray")
plt.figure()
#above is training images
#below is testing images

n_dress = np.argwhere(y_train == 3)[0]
n_coat = np.argwhere(y_train == 4)[0]
n_sandal = np.argwhere(y_train == 5)[0]

img_tdress = x_train[n_dress]
plt.imshow(np.reshape(img_tdress,(28,28)),cmap="gray")
plt.figure()

img_tcoat = x_train[n_coat]
plt.imshow(np.reshape(img_tcoat ,(28,28)),cmap="gray")
plt.figure()

img_tsandal = x_train[n_sandal]
plt.imshow(np.reshape(img_tsandal ,(28,28)),cmap="gray")
plt.figure()
```

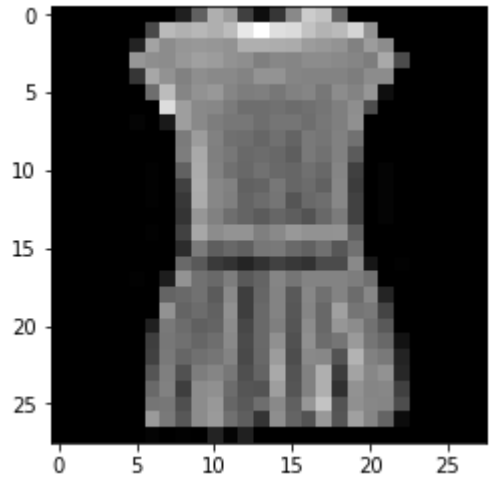
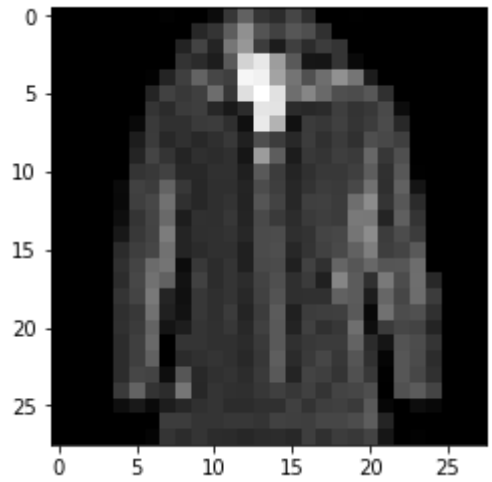
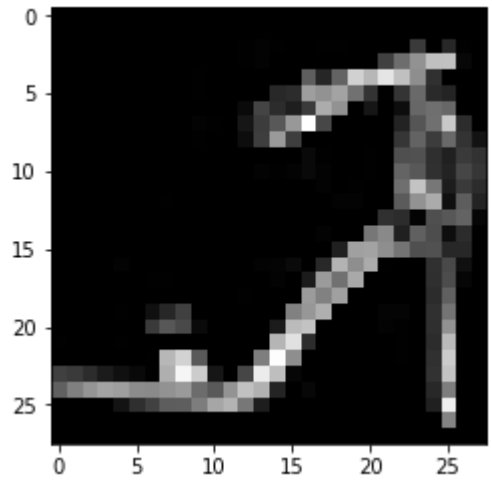
Out[122]:

<Figure size 432x288 with 0 Axes>







<Figure size 432x288 with 0 Axes>

1.3 Data balance (4 points)

Print out the number of training samples for each class.

In [123]:

```
t_shirt = len(np.argwhere(y_train == 0))
print("number of T shirts:", t_shirt)
t_trouser = len(np.argwhere(y_train == 1))
print("number of Toursers:", t_trouser)
t_pullover = len(np.argwhere(y_train == 2))
print("number of Pullover:", t_pullover)
t_dress = len(np.argwhere(y_train == 3))
print("number of Dress:", t_dress)
t_coat = len(np.argwhere(y_train == 4))
print("number of Coat:", t_coat)
t_sandal = len(np.argwhere(y_train == 5))
print("number of Sandal:", t_shirt)
t_shirt = len(np.argwhere(y_train == 6))
print("number of Shirts:", t_shirt)
t_sneaker = len(np.argwhere(y_train == 7))
print("number of Sneaker:", t_sneaker)
t_bag = len(np.argwhere(y_train == 8))
print("number of Bag:", t_bag)
t_ankleboot = len(np.argwhere(y_train == 9))
print("number of Ankle boot:", t_ankleboot)
```

```
number of T shirts: 6000
number of Toursers: 6000
number of Pullover: 6000
number of Dress: 6000
number of Coat: 6000
number of Sandal: 6000
number of Shirts: 6000
number of Sneaker: 6000
number of Bag: 6000
number of Ankle boot: 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 [ ]:

```
The dataset is perfectly balanced. Most machine learning classification algorith
ms are sensitive to unbalance in the predictor classes, and
an unbalanced dataset will bias the prediction model towards the more common cla
ss, which means the image classification can be
inaccurate with high probability.
```

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 [124]:

```
# 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 [125]:

```
model = keras.models.Sequential()
model.add(Dense(units=784, activation='relu', input_dim=len(x_train[0])))
model.add(Dropout(0.25))
model.add(Dense(units=500, activation='relu'))
model.add(Dropout(0.1))
model.add(Dense(units=150, activation='relu'))
model.add(Dense(units=10, activation='softmax'))
#sgd = SGD(lr=0.01, decay=0.0, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['accura
cy'])
print(model.summary())
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_21 (Dense)             (None, 784)               615440
_________________________________________________________________
dropout_11 (Dropout)         (None, 784)               0
_________________________________________________________________
dense_22 (Dense)             (None, 500)               392500
_________________________________________________________________
dropout_12 (Dropout)         (None, 500)               0
_________________________________________________________________
dense_23 (Dense)             (None, 150)               75150
_________________________________________________________________
dense_24 (Dense)             (None, 10)                1510
=================================================================
Total params: 1,084,600
Trainable params: 1,084,600
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 [150]:

```
batch_size = 1000
epochs = 100
model.fit(x_train, y_train, batch_size, epochs)
```

```
Epoch 1/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.3875 - acc: 0.8588
Epoch 2/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.3842 - acc: 0.8604
Epoch 3/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.3767 - acc: 0.8624
Epoch 4/100
60000/60000 [==============================] - 3s 53us/step - loss:
0.3711 - acc: 0.8641
Epoch 5/100
60000/60000 [==============================] - 3s 53us/step - loss:
0.3716 - acc: 0.8632
Epoch 6/100
60000/60000 [==============================] - 3s 53us/step - loss:
0.3656 - acc: 0.8649
Epoch 7/100
60000/60000 [==============================] - 3s 53us/step - loss:
0.3592 - acc: 0.8667
Epoch 8/100
60000/60000 [==============================] - 3s 53us/step - loss:
0.3604 - acc: 0.8677
Epoch 9/100
60000/60000 [==============================] - 3s 53us/step - loss:
0.3540 - acc: 0.8690
Epoch 10/100
60000/60000 [==============================] - 3s 53us/step - loss:
0.3537 - acc: 0.8687
Epoch 11/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.3518 - acc: 0.8703
Epoch 12/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.3484 - acc: 0.8711
Epoch 13/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.3462 - acc: 0.8731
Epoch 14/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.3434 - acc: 0.8738
Epoch 15/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.3399 - acc: 0.8741
Epoch 16/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.3374 - acc: 0.8772
Epoch 17/100
60000/60000 [==============================] - 3s 55us/step - loss:
0.3335 - acc: 0.8756
Epoch 18/100
60000/60000 [==============================] - 3s 56us/step - loss:
0.3308 - acc: 0.8767
Epoch 19/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.3292 - acc: 0.8763
Epoch 20/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.3289 - acc: 0.8780
Epoch 21/100
```

```
60000/60000 [==============================] - 3s 54us/step - loss:
0.3274 - acc: 0.8789
Epoch 22/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.3234 - acc: 0.8774
Epoch 23/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.3241 - acc: 0.8797
Epoch 24/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.3191 - acc: 0.8795
Epoch 25/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.3138 - acc: 0.8819
Epoch 26/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.3175 - acc: 0.8813
Epoch 27/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.3109 - acc: 0.8831
Epoch 28/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.3121 - acc: 0.8825
Epoch 29/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.3072 - acc: 0.8853
Epoch 30/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.3106 - acc: 0.8835
Epoch 31/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.3083 - acc: 0.8842
Epoch 32/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.3038 - acc: 0.8863
Epoch 33/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.3038 - acc: 0.8855
Epoch 34/100
60000/60000 [==============================] - 3s 53us/step - loss:
0.2984 - acc: 0.8885
Epoch 35/100
60000/60000 [==============================] - 3s 53us/step - loss:
0.2985 - acc: 0.8885
Epoch 36/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.2975 - acc: 0.8883
Epoch 37/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.2956 - acc: 0.8889
Epoch 38/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.2952 - acc: 0.8894
Epoch 39/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.2938 - acc: 0.8893
Epoch 40/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.2898 - acc: 0.8908
Epoch 41/100
60000/60000 [==============================] - 3s 55us/step - loss:
```

```
0.2901 - acc: 0.8912
Epoch 42/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.2895 - acc: 0.8914
Epoch 43/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.2857 - acc: 0.8917
Epoch 44/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.2872 - acc: 0.8926
Epoch 45/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.2825 - acc: 0.8931
Epoch 46/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.2795 - acc: 0.8941
Epoch 47/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.2793 - acc: 0.8941
Epoch 48/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.2792 - acc: 0.8958
Epoch 49/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.2797 - acc: 0.8941
Epoch 50/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.2757 - acc: 0.8947
Epoch 51/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.2752 - acc: 0.8965
Epoch 52/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.2732 - acc: 0.8955
Epoch 53/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.2727 - acc: 0.8966
Epoch 54/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.2702 - acc: 0.8969
Epoch 55/100
60000/60000 [==============================] - 3s 53us/step - loss:
0.2674 - acc: 0.8982
Epoch 56/100
60000/60000 [==============================] - 3s 53us/step - loss:
0.2698 - acc: 0.8968
Epoch 57/100
60000/60000 [==============================] - 3s 53us/step - loss:
0.2680 - acc: 0.8985
Epoch 58/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.2615 - acc: 0.8998
Epoch 59/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.2618 - acc: 0.8994
Epoch 60/100
60000/60000 [==============================] - 3s 53us/step - loss:
0.2643 - acc: 0.8996
Epoch 61/100
60000/60000 [==============================] - 3s 53us/step - loss:
0.2629 - acc: 0.9008
```

```
Epoch 62/100
60000/60000 [==============================] - 3s 53us/step - loss:
0.2596 - acc: 0.9011
Epoch 63/100
60000/60000 [==============================] - 3s 53us/step - loss:
0.2593 - acc: 0.9027
Epoch 64/100
60000/60000 [==============================] - 3s 53us/step - loss:
0.2587 - acc: 0.9019
Epoch 65/100
60000/60000 [==============================] - 3s 53us/step - loss:
0.2568 - acc: 0.9022
Epoch 66/100
60000/60000 [==============================] - 3s 53us/step - loss:
0.2550 - acc: 0.9023
Epoch 67/100
60000/60000 [==============================] - 3s 53us/step - loss:
0.2539 - acc: 0.9044
Epoch 68/100
60000/60000 [==============================] - 3s 53us/step - loss:
0.2526 - acc: 0.9044
Epoch 69/100
60000/60000 [==============================] - 3s 53us/step - loss:
0.2523 - acc: 0.9037
Epoch 70/100
60000/60000 [==============================] - 3s 53us/step - loss:
0.2525 - acc: 0.9036
Epoch 71/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.2505 - acc: 0.9031
Epoch 72/100
60000/60000 [==============================] - 3s 53us/step - loss:
0.2481 - acc: 0.9053
Epoch 73/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.2469 - acc: 0.9059
Epoch 74/100
60000/60000 [==============================] - 3s 53us/step - loss:
0.2450 - acc: 0.9067
Epoch 75/100
60000/60000 [==============================] - 4s 59us/step - loss:
0.2447 - acc: 0.9067
Epoch 76/100
60000/60000 [==============================] - 3s 56us/step - loss:
0.2455 - acc: 0.9066
Epoch 77/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.2428 - acc: 0.9076
Epoch 78/100
60000/60000 [==============================] - 3s 55us/step - loss:
0.2442 - acc: 0.9068
Epoch 79/100
60000/60000 [==============================] - 3s 56us/step - loss:
0.2402 - acc: 0.9083
Epoch 80/100
60000/60000 [==============================] - 3s 56us/step - loss:
0.2419 - acc: 0.9075
Epoch 81/100
60000/60000 [==============================] - 3s 56us/step - loss:
0.2420 - acc: 0.9066
Epoch 82/100
```

```
60000/60000 [==============================] - 3s 56us/step - loss:
0.2388 - acc: 0.9090
Epoch 83/100
60000/60000 [==============================] - 3s 55us/step - loss:
0.2387 - acc: 0.9076
Epoch 84/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.2372 - acc: 0.9102
Epoch 85/100
60000/60000 [==============================] - 3s 55us/step - loss:
0.2364 - acc: 0.9093
Epoch 86/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.2357 - acc: 0.9092
Epoch 87/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.2328 - acc: 0.9102
Epoch 88/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.2320 - acc: 0.9112
Epoch 89/100
60000/60000 [==============================] - 3s 54us/step - loss:
0.2311 - acc: 0.9099
Epoch 90/100
60000/60000 [==============================] - 3s 55us/step - loss:
0.2304 - acc: 0.9113
Epoch 91/100
60000/60000 [==============================] - 3s 56us/step - loss:
0.2269 - acc: 0.9129
Epoch 92/100
60000/60000 [==============================] - 3s 56us/step - loss:
0.2297 - acc: 0.9116
Epoch 93/100
60000/60000 [==============================] - 3s 55us/step - loss:
0.2262 - acc: 0.9141
Epoch 94/100
60000/60000 [==============================] - 3s 55us/step - loss:
0.2252 - acc: 0.9135
Epoch 95/100
60000/60000 [==============================] - 3s 56us/step - loss:
0.2255 - acc: 0.9134
Epoch 96/100
60000/60000 [==============================] - 3s 56us/step - loss:
0.2223 - acc: 0.9145
Epoch 97/100
60000/60000 [==============================] - 3s 57us/step - loss:
0.2274 - acc: 0.9122
Epoch 98/100
60000/60000 [==============================] - 3s 56us/step - loss:
0.2245 - acc: 0.9136
Epoch 99/100
60000/60000 [==============================] - 3s 56us/step - loss:
0.2236 - acc: 0.9133
Epoch 100/100
60000/60000 [==============================] - 3s 55us/step - loss:
0.2213 - acc: 0.9141
```

Out[150]:

```
<keras.callbacks.History at 0x7fab9c73f7b8>
```

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 [151]:

```
# 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 [152]:

```
score = model.evaluate(x_test, y_test, verbose=0)
print('\n', 'Test accuracy:', score[1])
```

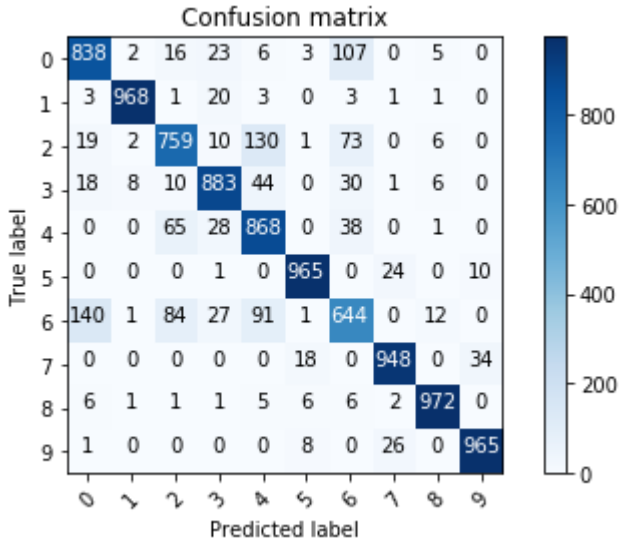
```
 Test accuracy: 0.881
```

3.2 Calculate and plot the confusion matrix (10 points)

In [153]:

```
ypred_onehot = model.predict(x_test)
# Convert predictions classes from one hot vectors to labels: [0 0 1 0 0 ...] --
> 2
ypred = np.argmax(ypred_onehot,axis=1)
# Convert validation observations from one hot vectors to labels
ytrue = np.argmax(y_test,axis=1)
# compute the confusion matrix
confusion_mtx = confusion_matrix(ytrue, ypred)
# plot the confusion matrix
plot_confusion_matrix(confusion_mtx, classes=range(num_classes))
```

```
Confusion matrix, without normalization
[[838   2  16  23   6   3 107   0   5   0]
 [  3 968   1  20   3   0   3   1   1   0]
 [ 19   2 759  10 130   1  73   0   6   0]
 [ 18   8  10 883  44   0  30   1   6   0]
 [  0   0  65  28 868   0  38   0   1   0]
 [  0   0   0   1   0 965   0  24   0  10]
 [140   1  84  27  91   1 644   0  12   0]
 [  0   0   0   0   0  18   0 948   0  34]
 [  6   1   1   1   5   6   6   2 972   0]
 [  1   0   0   0   0   8   0  26   0 965]]
```



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.

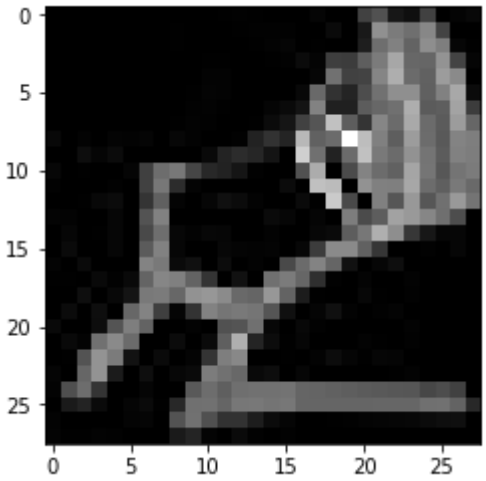
4.1 Load and visualise your own images (6 points)

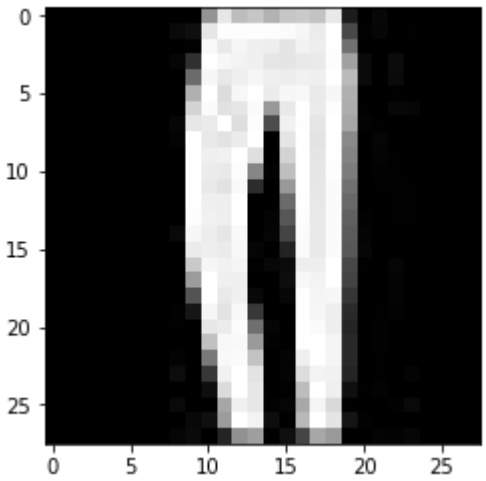
In [160]:

```
import imageio
data11 = imageio.imread('my_not.jpg')
plt.imshow(data11)
plt.figure()
data111 = imageio.imread('my_yes.jpg')
plt.imshow(data)
plt.figure()
```

Out[160]:

```
<Figure size 432x288 with 0 Axes>
```






```
<Figure size 432x288 with 0 Axes>
```

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

In [184]:

```
y_p = model.predict(data11.reshape(1,784))
y_p = np.argmax(y_p,axis=1)
print("data11 does not belong to any class, so if print class exist ->:",y_p,",c
lassfied wrong")
y_p = model.predict(data111.reshape(1,784))
y_p = np.argmax(y_p,axis=1)
print("data1111 belongs to class [1], by testing, it shows",y_p)
```

```
data11 does not belong to any class, so if print class exist ->:
[5] ,classfied wrong
data1111 belongs to class [1], by testing, it shows [1]
```

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

In [ ]:

```
The classification results is complety correct. Balance system bus load can impr
ove the performance, because we don not want all I/O for NIC and hard drives and
 tape drives, etc. on the same bus.
.Realize that data does not go directly from the hard drive to the NIC if they a
re on the same bus.
All the components still have to chat with CPU, so if there is contention,
it is faster if they are on separate busses.
```

5. Survey

How long did the coursework take you to solve?

In [125]:

```
6 hours
```

```
  File "<ipython-input-125-58241207e968>", line 1
    4 hours
      ^
SyntaxError: invalid syntax
```