Coursework 2: Image Classification

In this coursework, we are going to develop a neural network model for for image classification.

What to do?

- The coursework includes both coding questions and written questions. Please read both the text and code comment in this notebook to get an idea what you are supposed to implement.
- First, run `jupyter-lab` or `jupyter-notebook` in the terminal to start the Jupyter notebook.
- Then, complete and run the code to get the results.
- Finally, please export (File | Export Notebook As...) or print (using the print function of your browser) the notebook as a pdf file, which contains your code, results and answers, and upload the pdf file onto Cate.

Dependencies:

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

```
source /vol/bitbucket/wbai/virt/computer_vision_2020/bin/activate
```

When you no longer need the virtual environment, you can exit it by running `deactivate`.

- If you work on your own laptop using either Anaconda or plain Python, you can install new packages (such as numpy, imageio etc) running `conda install [package_name]` or `pip3 install [package_name]` in the terminal.

In [21]:

```
# Import libraries (provided)
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
import time
import random
from sklearn import metrics
import torch
import torch.nn as nn
import torch.nn.functional as FSS
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
```

1. Load and visualise data. (25 marks)

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

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

The dataset is prepared in a similar way to MNIST. It is split into a set of 60,000 training images and a set of 10,000 test images. The images are of size 28x28 pixels.

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

In [22]:

```
# Load data (provided)
train_set = torchvision.datasets.FashionMNIST(root='.', download=True, train=True,transform=tra
nsforms.ToTensor())
train_image = np.array(train_set.data)
train_label = np.array(train_set.targets)
class_name = train_set.classes

test_set = torchvision.datasets.FashionMNIST(root='.', download=True, train=False,transform=tra
nsforms.ToTensor())
test_image = np.array(test_set.data)
test_label = np.array(test_set.targets)
```

1.1 Display the dimension of the training and test sets. (5 marks)

In [23]:

```
print(f"Training image dim: {train_image.shape}")
print(f"Testing image dim: {test_image.shape}")

print(f"Training label: {train_label[0:300]}")

arr = np.where(train_label[0:300]==0)
print(f"Index: {arr[0][1]}")
```

```
Training image dim: (60000, 28, 28)
Testing image dim: (10000, 28, 28)
Training label: [9 0 0 3 0 2 7 2 5 5 0 9 5 5 7 9 1 0 6 4 3 1 4 8 4 3 0 2 4 4 5 3 6
6 0 8 5
 2 1 6 6 7 9 5 9 2 7 3 0 3 3 3 7 2 2 6 6 8 3 3 5 0 5 5 0 2 0 0 4 1 3 1 6 3
 1 4 4 6 1 9 1 3 5 7 9 7 1 7 9 9 9 3 2 9 3 6 4 1 1 8 8 0 1 1 6 8 1 9 7 8 8
 9 6 6 3 1 5 4 6 7 5 5 9 2 2 2 7 6 4 1 8 7 7 5 4 2 9 1 7 4 6 9 7 1 8 7 1 2
 8 0 9 1 8 7 0 5 8 6 7 2 0 8 7 1 6 2 1 9 6 0 1 0 5 5 1 7 0 5 8 4 0 4 0 6 6
 4 0 0 4 7 3 0 5 8 4 1 1 2 9 2 8 5 0 6 3 4 6 0 9 1 7 3 8 5 8 3 8 5 2 0 8 7
 0 3 5 0 6 5 2 7 5 2 6 8 2 6 8 0 4 4 4 4 4 1 5 6 5 3 3 7 3 3 6 2 8 4 6 5 9
 3 2 3 2 4 4 8 2 5 3 0 7 2 0 2 5 7 2 3 1 7 6 2 9 1 9 1 1 8 7 8 4 2 6 6 7 9
 4 6 1 9]
Index: 2
```

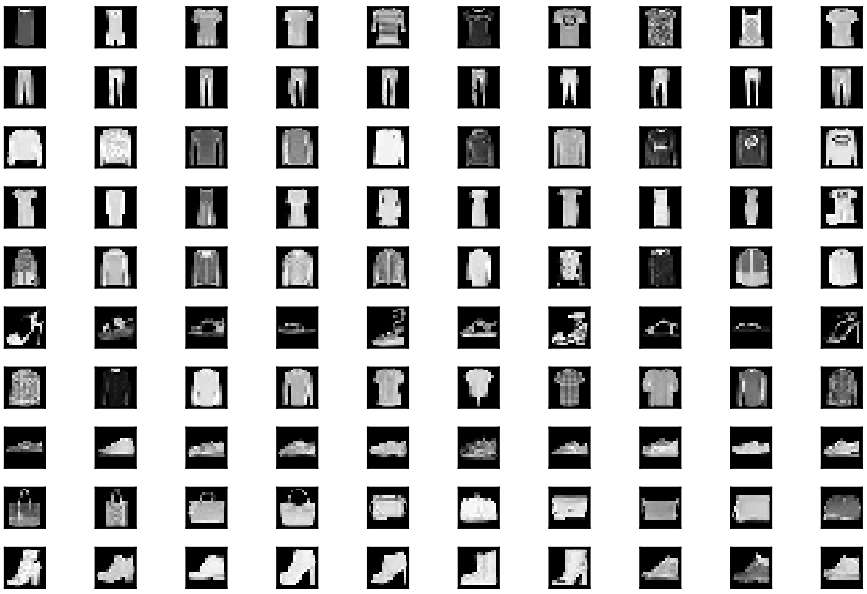
1.2 Visualise sample images for each of the 10 classes. (10 marks)

Please plot 10 rows x 10 columns of images. Each row shows 10 samples for one class. For example, row 1 shows 10 `T-shirt/top` images, row 2 shows 10 `Trousers` images.

In [24]:

```
# Let's visualize some examples
N=10
start_val = 0 # pick an element for the code to plot the following N**2 values
fig, axes = plt.subplots(N,N)
# items = list(range(0, 10))

for row in range(N):
  arr = np.where(train_label[start_val:start_val+300]==row)
  for col in range(N):
    # idx = start_val+row+N*col
    idx = arr[0][col+1]
    axes[row,col].imshow(train_image[idx], cmap='gray')
    fig.subplots_adjust(hspace=0.5)
    # fig.subplots_adjust(xspace=0.5)
    target = str(train_label[idx])
    # axes[row,col].set_title(target)
    axes[row,col].set_xticks([])
    axes[row,col].set_yticks([])
```



1.3 Display the number of training samples for each class. (5 marks)

In [25]:

```
name_list = ['top','Trousers','Pullover','Dress','Coat','Sandal','Shirt','Sneaker','Bag','Ankle
 boot']

for i in range(10):
    arr = np.where(train_label[0:]==i)
    print(f"{name_list[i]}: {len(arr[0])} \n")
# * 0: T-shirt/top
# * 1: Trousers
# * 2: Pullover
# * 3: Dress
# * 4: Coat
# * 5: Sandal
# * 6: Shirt
# * 7: Sneaker
# * 8: Bag
# * 9: Ankle boot
```

```
top: 6000

Trousers: 6000

Pullover: 6000

Dress: 6000

Coat: 6000

Sandal: 6000

Shirt: 6000

Sneaker: 6000

Bag: 6000

Ankle boot: 6000
```

1.4 Discussion. (5 marks)

Is the dataset balanced? What would happen for the image classification task if the dataset is not balanced?
The dataset is balanced in terms of number of images per class. If the dataset is not balanced, the trained model would be biased such that the test output would be biased as well.

In [ ]:

```

```

2. Image classification. (60 marks)

2.1 Build a convolutional neural network using the PyTorch library to perform classification on the Fashion-MNIST dataset. (15 marks)

You can use a network architecture similar to LeNet (shown below), which consists a number of convolutional layers and a few fully connected layers at the end.

In [26]:

```
class cnnNet(nn.Module):
    def __init__(self):
        super(cnnNet,self).__init__()
        self.conv1 = nn.Conv2d(1, 6, kernel_size=5,padding=2)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5)
        self.conv3 = nn.Conv2d(16,120,kernel_size=5)
        self.mp = nn.MaxPool2d(2)
        self.relu = nn.ReLU()
        self.fc1 = nn.Linear(120,84)
        self.fc2 = nn.Linear(84,10)
        self.logsoftmax = nn.LogSoftmax()

    def forward(self,x):
        in_size = x.size(0)
        out = self.relu(self.mp(self.conv1(x)))
        out = self.relu(self.mp(self.conv2(out)))
        out = self.relu(self.conv3(out))
        out = out.view(in_size, -1)
        out = self.relu(self.fc1(out))
        out = self.fc2(out)
        return self.logsoftmax(out)
```

In [27]:

```
model = cnnNet()
print(model)
```

```
cnnNet(
  (conv1): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (conv3): Conv2d(16, 120, kernel_size=(5, 5), stride=(1, 1))
  (mp): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (relu): ReLU()
  (fc1): Linear(in_features=120, out_features=84, bias=True)
  (fc2): Linear(in_features=84, out_features=10, bias=True)
  (logsoftmax): LogSoftmax()
)
```

2.2 Define the loss function, optimiser and hyper-parameters such as the learning rate, number of iterations, batch size etc. (5 marks)

In [28]:

```
# Define optimizer used
optimizer = optim.Adam(model.parameters(),lr=0.0001)
# Define Loss function used
lossfunc = nn.CrossEntropyLoss()
# Batch size
batch_size = 64
# Number of epochs
n_epochs = 25
```

2.3 Start model training. (20 marks)

At each iteration, get a random batch of images and labels from train_image and train_label, convert them into torch tensors, feed into the network model and perform gradient descent. Please also evaluate how long it takes for training.

In [29]:

```
train_losses = []

train_loader = torch.utils.data.DataLoader(dataset=train_set,batch_size=batch_size, shuffle=Tru
e)
total_step = len(train_loader)
for epoch in range(n_epochs):
    for i,(images,labels) in enumerate(train_loader):
        model.train() # Enables Training mode
        tr_loss = 0

        # print()
        output_train = model(images)

        # compute the loss
        loss_train = lossfunc(output_train,labels)
        # Keep all loss
        train_losses.append(loss_train)

        # Compute and update weights
        loss_train.backward()
        optimizer.step() # Performs single opt step
        # print(f"Epoch:{epoch} \t loss:{loss_train}")
        if (i+1) % 100 == 0:
            print ('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'.format(epoch+1, n_epochs, i+1,
total_step, loss_train.item()))
```

```
Epoch [1/25], Step [100/938], Loss: 1.9961
Epoch [1/25], Step [200/938], Loss: 1.4395
Epoch [1/25], Step [300/938], Loss: 1.3731
Epoch [1/25], Step [400/938], Loss: 1.7191
Epoch [1/25], Step [500/938], Loss: 1.2267
Epoch [1/25], Step [600/938], Loss: 1.4416
Epoch [1/25], Step [700/938], Loss: 0.9199
Epoch [1/25], Step [800/938], Loss: 0.8818
Epoch [1/25], Step [900/938], Loss: 0.8904
Epoch [2/25], Step [100/938], Loss: 0.7883
Epoch [2/25], Step [200/938], Loss: 0.6627
Epoch [2/25], Step [300/938], Loss: 0.9857
Epoch [2/25], Step [400/938], Loss: 0.7718
Epoch [2/25], Step [500/938], Loss: 0.9828
Epoch [2/25], Step [600/938], Loss: 0.8058
Epoch [2/25], Step [700/938], Loss: 0.7607
Epoch [2/25], Step [800/938], Loss: 0.6833
Epoch [2/25], Step [900/938], Loss: 0.7223
Epoch [3/25], Step [100/938], Loss: 0.7826
Epoch [3/25], Step [200/938], Loss: 0.7359
Epoch [3/25], Step [300/938], Loss: 0.8406
Epoch [3/25], Step [400/938], Loss: 0.6119
Epoch [3/25], Step [500/938], Loss: 0.6354
Epoch [3/25], Step [600/938], Loss: 0.5594
Epoch [3/25], Step [700/938], Loss: 0.6920
Epoch [3/25], Step [800/938], Loss: 0.6310
Epoch [3/25], Step [900/938], Loss: 0.4329
Epoch [4/25], Step [100/938], Loss: 0.5830
Epoch [4/25], Step [200/938], Loss: 0.4121
Epoch [4/25], Step [300/938], Loss: 0.5426
Epoch [4/25], Step [400/938], Loss: 0.4911
Epoch [4/25], Step [500/938], Loss: 0.6140
Epoch [4/25], Step [600/938], Loss: 0.5298
Epoch [4/25], Step [700/938], Loss: 0.8527
Epoch [4/25], Step [800/938], Loss: 0.8238
Epoch [4/25], Step [900/938], Loss: 0.5646
Epoch [5/25], Step [100/938], Loss: 0.6559
Epoch [5/25], Step [200/938], Loss: 0.7000
Epoch [5/25], Step [300/938], Loss: 0.6910
Epoch [5/25], Step [400/938], Loss: 0.7610
Epoch [5/25], Step [500/938], Loss: 0.5337
Epoch [5/25], Step [600/938], Loss: 0.5362
Epoch [5/25], Step [700/938], Loss: 0.8162
Epoch [5/25], Step [800/938], Loss: 0.6640
Epoch [5/25], Step [900/938], Loss: 0.8323
Epoch [6/25], Step [100/938], Loss: 0.6775
Epoch [6/25], Step [200/938], Loss: 0.7629
Epoch [6/25], Step [300/938], Loss: 0.4872
Epoch [6/25], Step [400/938], Loss: 0.6601
Epoch [6/25], Step [500/938], Loss: 0.5648
Epoch [6/25], Step [600/938], Loss: 0.5545
Epoch [6/25], Step [700/938], Loss: 0.7000
Epoch [6/25], Step [800/938], Loss: 0.7799
Epoch [6/25], Step [900/938], Loss: 0.7260
Epoch [7/25], Step [100/938], Loss: 0.3978
Epoch [7/25], Step [200/938], Loss: 0.4849
Epoch [7/25], Step [300/938], Loss: 0.8933
Epoch [7/25], Step [400/938], Loss: 0.5361
Epoch [7/25], Step [500/938], Loss: 0.4575
Epoch [7/25], Step [600/938], Loss: 0.6776
Epoch [7/25], Step [700/938], Loss: 0.4415
```

```
Epoch [7/25], Step [800/938], Loss: 0.3462
Epoch [7/25], Step [900/938], Loss: 0.6107
Epoch [8/25], Step [100/938], Loss: 0.5137
Epoch [8/25], Step [200/938], Loss: 0.4611
Epoch [8/25], Step [300/938], Loss: 0.4835
Epoch [8/25], Step [400/938], Loss: 0.5345
Epoch [8/25], Step [500/938], Loss: 0.3884
Epoch [8/25], Step [600/938], Loss: 0.4867
Epoch [8/25], Step [700/938], Loss: 0.5402
Epoch [8/25], Step [800/938], Loss: 0.2485
Epoch [8/25], Step [900/938], Loss: 0.5959
Epoch [9/25], Step [100/938], Loss: 0.2675
Epoch [9/25], Step [200/938], Loss: 0.5274
Epoch [9/25], Step [300/938], Loss: 0.5123
Epoch [9/25], Step [400/938], Loss: 0.2805
Epoch [9/25], Step [500/938], Loss: 0.5144
Epoch [9/25], Step [600/938], Loss: 0.4685
Epoch [9/25], Step [700/938], Loss: 0.3416
Epoch [9/25], Step [800/938], Loss: 0.4347
Epoch [9/25], Step [900/938], Loss: 0.5233
Epoch [10/25], Step [100/938], Loss: 0.4341
Epoch [10/25], Step [200/938], Loss: 0.6681
Epoch [10/25], Step [300/938], Loss: 0.6224
Epoch [10/25], Step [400/938], Loss: 0.5469
Epoch [10/25], Step [500/938], Loss: 0.3868
Epoch [10/25], Step [600/938], Loss: 0.4085
Epoch [10/25], Step [700/938], Loss: 0.3965
Epoch [10/25], Step [800/938], Loss: 0.2487
Epoch [10/25], Step [900/938], Loss: 0.7601
Epoch [11/25], Step [100/938], Loss: 0.3735
Epoch [11/25], Step [200/938], Loss: 0.4303
Epoch [11/25], Step [300/938], Loss: 0.4427
Epoch [11/25], Step [400/938], Loss: 0.5291
Epoch [11/25], Step [500/938], Loss: 0.3444
Epoch [11/25], Step [600/938], Loss: 0.5518
Epoch [11/25], Step [700/938], Loss: 0.4586
Epoch [11/25], Step [800/938], Loss: 0.4213
Epoch [11/25], Step [900/938], Loss: 0.3923
Epoch [12/25], Step [100/938], Loss: 0.5084
Epoch [12/25], Step [200/938], Loss: 0.7529
Epoch [12/25], Step [300/938], Loss: 0.4052
Epoch [12/25], Step [400/938], Loss: 0.6663
Epoch [12/25], Step [500/938], Loss: 0.5652
Epoch [12/25], Step [600/938], Loss: 0.4206
Epoch [12/25], Step [700/938], Loss: 0.6980
Epoch [12/25], Step [800/938], Loss: 0.3999
Epoch [12/25], Step [900/938], Loss: 0.5102
Epoch [13/25], Step [100/938], Loss: 0.5307
Epoch [13/25], Step [200/938], Loss: 0.3519
Epoch [13/25], Step [300/938], Loss: 0.2919
Epoch [13/25], Step [400/938], Loss: 0.3697
Epoch [13/25], Step [500/938], Loss: 0.3759
Epoch [13/25], Step [600/938], Loss: 0.6876
Epoch [13/25], Step [700/938], Loss: 0.4911
Epoch [13/25], Step [800/938], Loss: 0.4859
Epoch [13/25], Step [900/938], Loss: 0.4749
Epoch [14/25], Step [100/938], Loss: 0.5054
Epoch [14/25], Step [200/938], Loss: 0.4953
Epoch [14/25], Step [300/938], Loss: 0.4946
Epoch [14/25], Step [400/938], Loss: 0.5368
Epoch [14/25], Step [500/938], Loss: 0.3716
```

```
Epoch [14/25], Step [600/938], Loss: 0.3300
Epoch [14/25], Step [700/938], Loss: 0.5804
Epoch [14/25], Step [800/938], Loss: 0.3787
Epoch [14/25], Step [900/938], Loss: 0.4319
Epoch [15/25], Step [100/938], Loss: 0.4974
Epoch [15/25], Step [200/938], Loss: 0.4787
Epoch [15/25], Step [300/938], Loss: 0.3240
Epoch [15/25], Step [400/938], Loss: 0.3603
Epoch [15/25], Step [500/938], Loss: 0.4096
Epoch [15/25], Step [600/938], Loss: 0.4653
Epoch [15/25], Step [700/938], Loss: 0.5941
Epoch [15/25], Step [800/938], Loss: 0.4654
Epoch [15/25], Step [900/938], Loss: 0.4081
Epoch [16/25], Step [100/938], Loss: 0.5439
Epoch [16/25], Step [200/938], Loss: 0.4112
Epoch [16/25], Step [300/938], Loss: 0.3861
Epoch [16/25], Step [400/938], Loss: 0.5162
Epoch [16/25], Step [500/938], Loss: 0.2363
Epoch [16/25], Step [600/938], Loss: 0.4926
Epoch [16/25], Step [700/938], Loss: 0.4948
Epoch [16/25], Step [800/938], Loss: 0.4777
Epoch [16/25], Step [900/938], Loss: 0.5353
Epoch [17/25], Step [100/938], Loss: 0.4971
Epoch [17/25], Step [200/938], Loss: 0.4583
Epoch [17/25], Step [300/938], Loss: 0.2665
Epoch [17/25], Step [400/938], Loss: 0.6546
Epoch [17/25], Step [500/938], Loss: 0.5516
Epoch [17/25], Step [600/938], Loss: 0.6297
Epoch [17/25], Step [700/938], Loss: 0.3910
Epoch [17/25], Step [800/938], Loss: 0.5102
Epoch [17/25], Step [900/938], Loss: 0.4373
Epoch [18/25], Step [100/938], Loss: 0.4486
Epoch [18/25], Step [200/938], Loss: 0.3849
Epoch [18/25], Step [300/938], Loss: 0.4134
Epoch [18/25], Step [400/938], Loss: 0.2656
Epoch [18/25], Step [500/938], Loss: 0.3715
Epoch [18/25], Step [600/938], Loss: 0.3020
Epoch [18/25], Step [700/938], Loss: 0.4073
Epoch [18/25], Step [800/938], Loss: 0.5356
Epoch [18/25], Step [900/938], Loss: 0.7010
Epoch [19/25], Step [100/938], Loss: 0.4889
Epoch [19/25], Step [200/938], Loss: 0.5462
Epoch [19/25], Step [300/938], Loss: 0.6288
Epoch [19/25], Step [400/938], Loss: 0.3301
Epoch [19/25], Step [500/938], Loss: 0.3900
Epoch [19/25], Step [600/938], Loss: 0.2979
Epoch [19/25], Step [700/938], Loss: 0.4226
Epoch [19/25], Step [800/938], Loss: 0.3784
Epoch [19/25], Step [900/938], Loss: 0.3249
Epoch [20/25], Step [100/938], Loss: 0.2695
Epoch [20/25], Step [200/938], Loss: 0.4122
Epoch [20/25], Step [300/938], Loss: 0.3371
Epoch [20/25], Step [400/938], Loss: 0.4426
Epoch [20/25], Step [500/938], Loss: 0.6996
Epoch [20/25], Step [600/938], Loss: 0.3016
Epoch [20/25], Step [700/938], Loss: 0.2852
Epoch [20/25], Step [800/938], Loss: 0.3882
Epoch [20/25], Step [900/938], Loss: 0.6233
Epoch [21/25], Step [100/938], Loss: 0.3537
Epoch [21/25], Step [200/938], Loss: 0.4454
Epoch [21/25], Step [300/938], Loss: 0.5678
```

```
Epoch [21/25], Step [400/938], Loss: 0.2785
Epoch [21/25], Step [500/938], Loss: 0.2137
Epoch [21/25], Step [600/938], Loss: 0.2829
Epoch [21/25], Step [700/938], Loss: 0.7400
Epoch [21/25], Step [800/938], Loss: 0.5358
Epoch [21/25], Step [900/938], Loss: 0.2474
Epoch [22/25], Step [100/938], Loss: 0.4997
Epoch [22/25], Step [200/938], Loss: 0.4149
Epoch [22/25], Step [300/938], Loss: 0.4289
Epoch [22/25], Step [400/938], Loss: 0.5082
Epoch [22/25], Step [500/938], Loss: 0.5637
Epoch [22/25], Step [600/938], Loss: 0.5530
Epoch [22/25], Step [700/938], Loss: 0.6968
Epoch [22/25], Step [800/938], Loss: 0.3012
Epoch [22/25], Step [900/938], Loss: 0.4581
Epoch [23/25], Step [100/938], Loss: 0.4270
Epoch [23/25], Step [200/938], Loss: 0.5627
Epoch [23/25], Step [300/938], Loss: 0.4777
Epoch [23/25], Step [400/938], Loss: 0.6773
Epoch [23/25], Step [500/938], Loss: 0.5310
Epoch [23/25], Step [600/938], Loss: 0.4443
Epoch [23/25], Step [700/938], Loss: 0.4959
Epoch [23/25], Step [800/938], Loss: 0.4621
Epoch [23/25], Step [900/938], Loss: 0.5548
Epoch [24/25], Step [100/938], Loss: 0.4842
Epoch [24/25], Step [200/938], Loss: 0.2810
Epoch [24/25], Step [300/938], Loss: 0.5600
Epoch [24/25], Step [400/938], Loss: 0.4503
Epoch [24/25], Step [500/938], Loss: 0.5527
Epoch [24/25], Step [600/938], Loss: 0.5357
Epoch [24/25], Step [700/938], Loss: 0.5360
Epoch [24/25], Step [800/938], Loss: 0.4631
Epoch [24/25], Step [900/938], Loss: 0.4033
Epoch [25/25], Step [100/938], Loss: 0.6152
Epoch [25/25], Step [200/938], Loss: 0.6222
Epoch [25/25], Step [300/938], Loss: 0.4675
Epoch [25/25], Step [400/938], Loss: 0.4538
Epoch [25/25], Step [500/938], Loss: 0.4382
Epoch [25/25], Step [600/938], Loss: 0.2858
Epoch [25/25], Step [700/938], Loss: 0.4625
Epoch [25/25], Step [800/938], Loss: 0.6271
Epoch [25/25], Step [900/938], Loss: 0.3564
```

2.4 Deploy the trained model onto the test set. (10 marks)

Please also evaluate how long it takes for testing.

In [30]:

```
acc_cof =0
test_loader = torch.utils.data.DataLoader(dataset=test_set,batch_size=batch_size, shuffle=False
)
model.eval()  # eval mode (batchnorm uses moving mean/variance instead of mini-batch mean/varian
ce)
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in test_loader:
        outputs = model(images)
#         print(images.shape)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
        cof_mat = np.array(metrics.confusion_matrix(labels,predicted))
        if cof_mat.shape != (10,10):
            continue
        else:
            acc_cof+=cof_mat
```

2.5 Evaluate the classification accuracy on the test set. (5 marks)

In [31]:

```
print('Test Accuracy of the model on the 10000 test images: {} %'.format(100 * correct / total))
```

```
Test Accuracy of the model on the 10000 test images: 81.87 %
```

2.6 Print out and visualise the confusion matrix. (5 marks)

You can use relevant functions in [scikit-learn (https://scikit-learn.org/stable/modules/classes.html#module-sklearn.metrics)](https://scikit-learn.org/stable/modules/classes.html#module-sklearn.metrics).

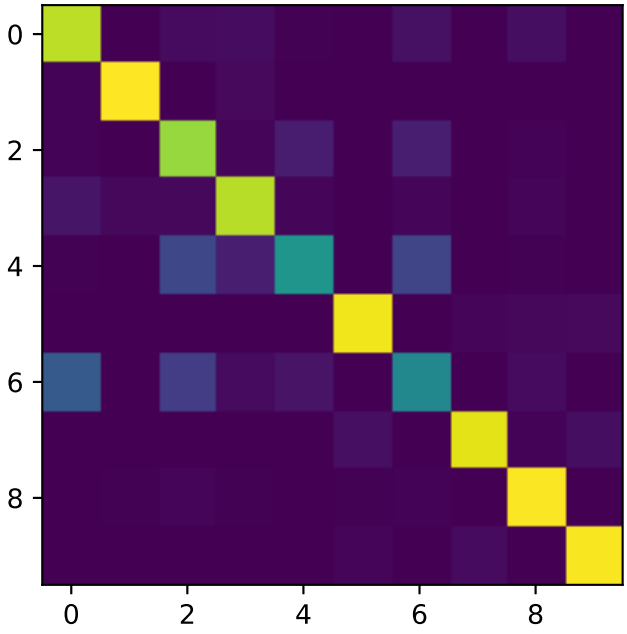
In [32]:

```
print(acc_cof)
plt.imshow(acc_cof)
```

```
[[854   0  28  33   4   1  43   0  34   0]
 [  9 950   3  25   3   0   1   0   2   0]
 [ 10   1 803  14  77   0  79   0   8   0]
 [ 55  21  21 847  17   0  14   0  12   0]
 [  4   1 207  79 496   0 199   0   4   0]
 [  1   0   0   0   0 930   0  14  19  24]
 [268   3 173  29  50   1 442   0  29   0]
 [  0   0   0   0   0  39   0 912  11  37]
 [  3   4  12   4   2   7   9   3 944   1]
 [  0   0   0   0   0  18   0  28   3 941]]
```

Out[32]:

```
<matplotlib.image.AxesImage at 0x7f1ee5893128>
```



3. Deploy in real world. (15 marks)

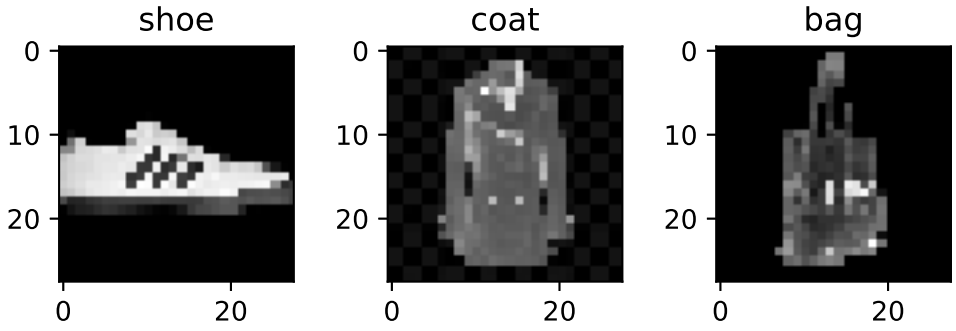
Take 3 photos that belongs to the 10 classes (e.g. clothes, shoes) in your real life. Use either Python or other software (Photoshop, Gimp etc) to convert the photos into grayscale, negate the intensities so that background becomes black or dark, crop the region of interest and reshape into the size of 28x28.

3.1 Load and visualise your own images (5 marks)

In [40]:

```
# img = matplotlib.image.imread('~/winDesktop/Computer-Vision/coursework_2/shoe.png')
# plt.imshow(img)
import cv2
inputs = ['shoe', 'coat' ,'bag']
figs = np.array([])
# fig, axes = plt.subplots(1,3)
fig = plt.figure()
fig.subplots_adjust(hspace=0.4, wspace=0.4)

for idx,input in enumerate(inputs):
    img = cv2.imread(str(input)+'.png')
    gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    neg = np.subtract(255,gray_img)
    re_neg = cv2.resize(neg,(28,28))
    figs = np.append(figs,re_neg)
    ax = fig.add_subplot(1, 3, idx+1)
    ax.imshow(re_neg,cmap='gray')
    ax.title.set_text(str(input))
```



3.2 Test your network on the real images and display the classification results. (5 marks)

In [49]:

```
shaped = figs.reshape(-1,1,28,28)
tens = torch.from_numpy(shaped).float()
outputs = model(tens)
_, predicted = torch.max(outputs.data, 1)
for i in range(3):
    print(f"Found class: {class_name[predicted[i]]}")
```

```
Found class: Bag
Found class: Coat
Found class: Bag
```

3.3 Discuss the classification results. (5 marks)

Does the model work? Is there anyway to improve the real life performance of the model? The model works as predicted. Howver, since the classification is not perfect, misclassification occured. Increasing the dimensions of data might be useful but would increase the amount of computation required.

In [ ]: