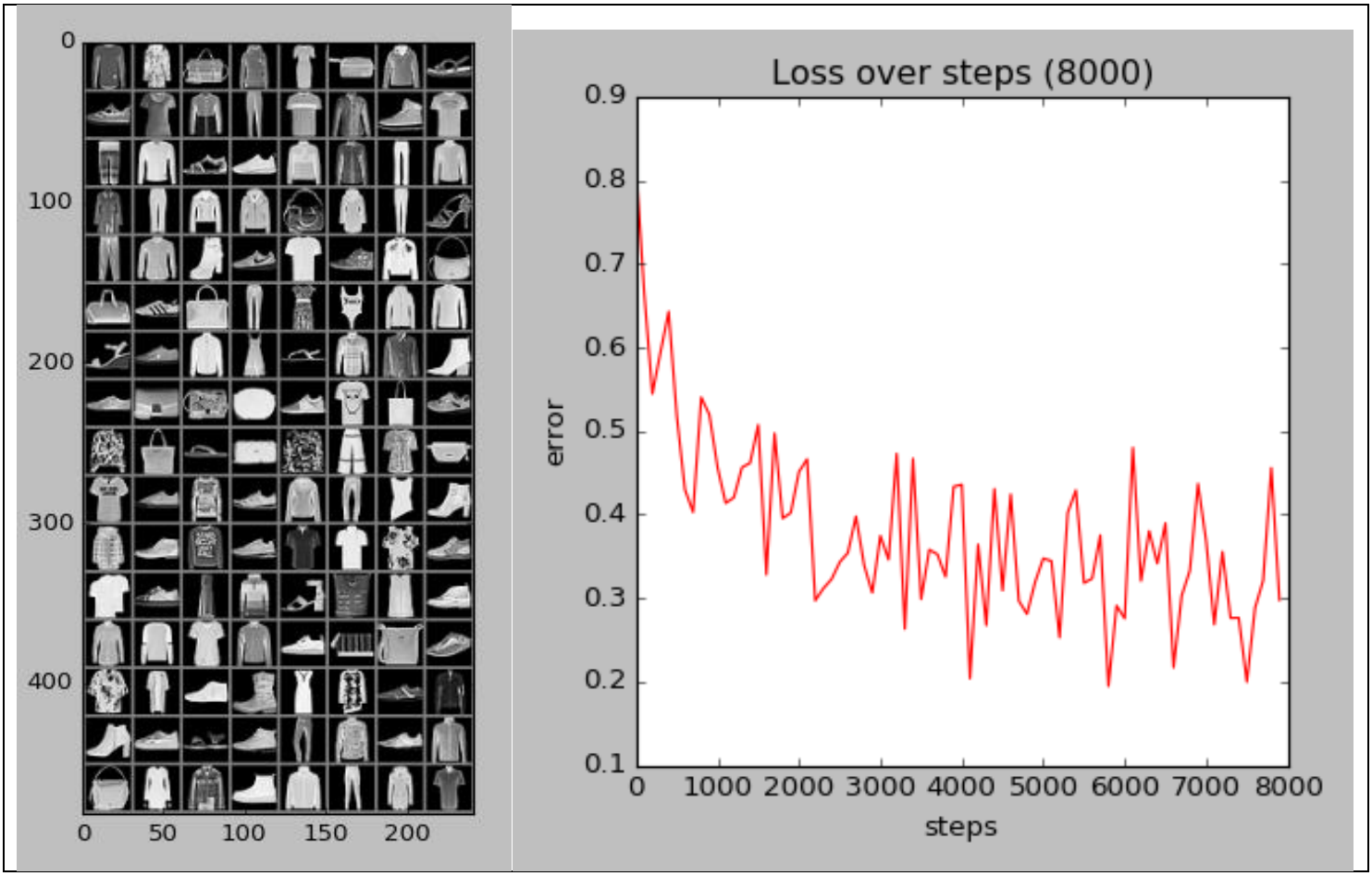
Exam 1

Q1

Run the program in PyCharm and investigate the errors. I have manually added some bugs into the sample program. The first step is to get an understanding of the code and fix the errors. Once you have fixed the errors, investigate and verify its performance.

Batch size output – **Loss Error –**



Accuracy of individual classes –

```
Accuracy of T-shirt/top : 87 %
Accuracy of Trouser : 93 %
Accuracy of Pullover : 80 %
Accuracy of Dress : 83 %
Accuracy of  Coat : 82 %
Accuracy of Sandal : 92 %
Accuracy of Shirt : 57 %
Accuracy of Sneaker : 100 %
Accuracy of   Bag : 98 %
Accuracy of Ankle boot : 96 %
```

++

Q2

Modify the program to run on the GPU.

++

Q3

Find out, if program is set up to perform stochastic gradient descent or mini-batch gradient descent. Explain your answer.

The training dataset is of 60,000 cases. Using a stochastic gradient descent algorithm on this data would involve setting the batch_size to 1 before executing the code. Suppose the number of epochs is 10, it would mean that this process would take 600,000 steps for the entire code to execute once. This is extremely time consuming & hence, it's better to use the mini-batch gradient descent since it's a more time-efficient process.

++

Q4

Experiment with different numbers of layers and different numbers of neurons (Deep Network vs Shallow Network). While increasing the number of layers in the network, make sure the total number of weights and biases in the network are roughly the same as the original. Describe how the performance changes as the number of layers increases – both in terms of training time and performance.

In order to compare the output, performance & training time of the models, I set a random seed (1122) & then proceeded with iterating with different number of layers & neurons.

All other parameters apart from Neurons & Layers were left untouched. I started the process with the layers & transfer function that was provided to us. (A 2 layer network with a ReLU transfer function in-between).

Then, while keeping the layers constant, I iterated over different sizes of Neurons [10, 20, 50, 100] & recorded the outputs. After one such iteration, I increased the layers to 3 & 5 & then proceeded with using the same set of Neurons that I used for the 2 Layer network.

Layers	Neurons	Accuracy	Precision	Recall	F1Score	Time (in mins)
2	10	85%	0.853	0.853	0.851	2.90435
2	20	85%	0.858	0.858	0.855	2.82618
2	50	87%	0.876	0.876	0.875	2.73348
2	100	87%	0.881	0.877	0.878	2.68658
3	10	84%	0.847	0.844	0.844	2.83093
3	20	86%	0.867	0.865	0.865	2.83859
3	50	87%	0.878	0.872	0.873	2.81081
3	100	88%	0.887	0.887	0.887	2.82889
5	10	84%	0.848	0.846	0.846	2.9599
5	20	86%	0.868	0.869	0.868	2.9896
5	50	87%	0.876	0.876	0.875	3.0236
5	100	87%	0.880	0.880	0.878	2.9809

Initially, with a shallow network & just 10 neurons, the F1Score was around .85 for all 3 sets of layers [2, 3, 5]. However, when I increased the number of layers keeping all other parameters constant, my F1Score went up. Also, adding more neurons, reduced the training time of the model.

++

Q5

Try several training functions from the list on this page: http://pytorch.org/docs/master/optim.html. Compare the performance with gradient descent. Hint: Use the same seed number.

input_size	28*28
hidden_size	100
num_classes	10
num_epochs	20
batch_size	128
learning_rate	0.001
Layers	3
Transfer Function	ReLU
Criterion	CrossEntropyLoss

Optimizer	ADAM
Test Accuracy - 10000 images	88%
Precision	0.8874896
Recall	0.8866000
F1Score	0.8867066
Time (in mins)	2.7780 mins

ADAGRAD	ADADELTA	RMSPROP	ASGD	ADAMAX	SGD
83%	73%	87%	77%	88%	77%
0.8377228	0.7214791	0.8764981	0.7681861	0.8853699	0.7682870
0.8386000	0.7314000	0.8763000	0.7707000	0.8834000	0.7708000
0.8379241	0.7240462	0.8753373	0.7688792	0.8840602	0.7689807
2.73236 mins	2.86897 mins	2.80662 mins	2.7594 mins	2.98076 mins	2.67699 mins

Keeping all other parameters constant & using a random_state, I fooled around with ADAM, ADAGRAD, ADADELTA, RMSPROP, ASGD, ADAMAX & SGD optimizers & my initial model from previous step using ADAM optimizer still stood out the best.

++

Q6

Try different transfer functions and do a comparison study between the transfer functions. Explain your results.

input_size	28*28
hidden_size	100
num_classes	10
num_epochs	20
batch_size	128
learning_rate	0.001
Optimizer	ADAM
Criterion	CrossEntropyLoss

Transfer Function	ReLU	LReLU	Softmax
Layers	2	2	2
Test Accuracy - 10000 images	87%	88%	75%
Precision	0.8806	0.8840	0.7525
Recall	0.8767	0.8800	0.7565
F1Score	0.8779	0.8814	0.7303
Time (in mins)	2.7071 mins	2.74711 mins	2.73286 mins

ReLU + ReLU	ReLU + Softmax	LeakyReLU + Softmax
3	3	3
88%	83%	86%
0.8875	0.8315	0.8667
0.8866	0.8359	0.8678
0.8867	0.8248	0.8657
2.7324 min	2.73132 mins	2.76537 mins

ReLU + ReLU + ReLU	ReLU + ReLU + Softmax
4	4
87%	86%
0.8782	0.8702
0.8777	0.8684
0.8772	0.8686
2.89982 mins	2.75033 mins

Initially the model had just 2 layers with the ReLU transfer function. Based on Q4, my model with the best output so far has been with 3 Layers, 100 Neurons, 20 epochs, 128 as batch size, using the ADAM optimizer and the CrossEntropyLoss as the criterion.

Keeping all other parameters constant, I changed the transfer function ReLU in the 2 layer model to Leaky ReLU & Softmax. My 2 layer model with Leaky ReLU had a similar accuracy score as my 3 layer model with 2 ReLU function. However, the later model has a slightly better F1Score. Keeping this model as my benchmark, I proceeded with adding more layers & more transfer functions. I did a 3 layer ReLU & a 3 layer 2 ReLUs & Softmax model, but the models performed poorly.

++

Q8

Try to use dropout nodes and check if the performance is better or not. Explain the advantages/disadvantages of using dropout nodes.

Dropout nodes are basically used to check if a model is being overfitted. In essence, what it does is check if my model has more parameters than necessary. The nn.Dropout(p) function shuts down p% of the neurons in my network randomly during each iteration. This helps to avoid over reliance on any particular neuron as the other neurons will have to compensate for the dropped neurons.
If my model accuracy is affected on using the dropout, it means that the model is not being overfitted as my model needs all the neurons to that are being shut down. If my model accuracy isn't affected, it means that I'm overfitting since shutting down the neurons did not affect the performance of my model.

Continuing with my current best model, I wanted to check if my model was being overfitted. So using the dropout feature (p = 0.2 & 0)

No Dropout
Neurons = **100**
Precision, Recall, F1Score & Support of this model is (0.8817, 0.8837116920550795, 0.8824558727866408, None)
Accuracy of the network on the 10000 test images: **88 %**

Using Dropout (p = .2)
Neurons = **100**
Precision, Recall, F1Score & Support of this model is (0.8837, 0.8849456380917516, 0.8838458405181306, None)

Accuracy of the network on the 10000 test images: **88 %**

Accuracy of the model remained the same after adding a dropout node. So I iteratively reduced the number of neurons till the time I found that the Accuracy started going down.

Using Dropout (p = .2)
Neurons = **60**
Precision, Recall, F1Score & Support of this model is (0.8737, 0.8749456380917516, 0.8738458405181306, None)
Accuracy of the network on the 10000 test images: **87 %**

As we can see, on introducing the dropout, the model accuracy went down, meaning the model was not being overfitted.

++

Q9

Make a chart and or table to compare the difference in performance between batch, minibatch, and stochastic gradient descent. Calculate the time it takes to train the network for each method by using the system (sys) clock command in Python.

Using different batch sizes, but reducing the epochs, I found that the mini batch process was the quickest, followed by the full batch method. The Stochastic method was the most time inefficient method.

Method	Batch	Epoch	Time (in mins)
Stochastic Method	1	2	4.15063
Mini batch	100	2	0.22927
	1,000	2	0.32702
	5,000	2	0.34888
	10,000	2	0.36918
Full Batch	60,000	2	0.56593

++

Q10

Plot the confusion matrix (as table or a figure) for your predictions and explain each element of the confusion matrix. Additionally, Calculate the accuracy and misclassification rate.

Since the calculation is being batch wise, I created a couple of lists to store the values of the predicted classes in the first list & the target values for the same batch in another list. Then, using sklearn.metrics I calculated a confusion matrix which is store in a numpy array. Then I converted the array to a dataframe with the feature names in the rows and columns. The rows have predicted values & the columns have the actual targets. Let us consider one element, say Tshirt/top. The target and predicted value for Tshirt/Top is 823, which means that the model correctly identified 823 Tshirts/tops. If we consider the value below that cell – 0, it means that the model did not misclassify any Tshirts/Tops as Trousers. The value to the right of 823, 4 – means that the model incorrectly identified 4 Trousers as Tshirt/top.

	T-shirt/top	Trouser	Pullover	Dress	Coat	Sandal	Shirt	Sneaker	Bag	Ankle boot
T-shirt/top	823	4	17	32	0	0	107	0	2	0
Trouser	0	965	0	5	2	0	1	0	0	0
Pullover	18	1	835	16	141	0	116	0	4	0
Dress	24	23	9	877	30	1	23	0	5	0
Coat	6	3	73	38	768	0	69	0	5	0
Sandal	0	0	0	0	0	935	0	13	3	6
Shirt	122	2	66	28	58	0	671	0	9	0
Sneaker	2	0	0	0	0	39	0	931	5	28
Bag	5	2	0	4	1	2	13	0	967	1
Ankle boot	0	0	0	0	0	23	0	56	0	965

```
Accuracy of the network on the 10000 test images: 87 %
Misclassification of the network on the 10000 test images: 12 %
```

Using sklearn.metrics, I also calculated the accuracy of the model, which is nothing but the ratio of Correctly classified Test cases and Total Test cases count. The misclassification is nothing but 1-Accuracy of the model.

++

Q13

Bonus: You can use torchvison capability of PyTorch to visualize the network. Try using tensorboard and look into weight and biases of network and find out how this information can be useful to improve your results.

Using the torchviz package, I tried to visualize the network. I could get the output, but not the image.

```
digraph {
graph [size="12,12"]
node [align=left fontsize=12 height=0.2 ranksep=0.1 shape=box style=filled]
139620986333896 [label=ThAddmmBackward fillcolor=darkolivegreen1]
139620986334008 -> 139620986333896
139620986334008 [label=ExpandBackward]
139620986333448 -> 139620986334008
139620986333448 [label="fc3.bias
 (10)" fillcolor=lightblue]
139620985749008 -> 139620986333896
139620985749008 [label=DropoutBackward]
139620986333504 -> 139620985749008
139620986333504 [label=ThresholdBackward0]
139620986332776 -> 139620986333504
139620986332776 [label=ThAddmmBackward]
139620986333280 -> 139620986332776
139620986333280 [label=ExpandBackward]
139620986331656 -> 139620986333280
139620986331656 [label="fc2.bias
 (60)" fillcolor=lightblue]
139622289400888 -> 139620986332776
139622289400888 [label=DropoutBackward]
139620986331376 -> 139622289400888
139620986331376 [label=ThresholdBackward0]
139620986580496 -> 139620986331376
139620986580496 [label=ThAddmmBackward]
139620986781480 -> 139620986580496
139620986781480 [label=ExpandBackward]
139620986780920 -> 139620986781480
139620986780920 [label="fc1.bias
 (60)" fillcolor=lightblue]
139620986781648 -> 139620986580496
139620986781648 [label=TBackward]
139620986780976 -> 139620986781648
```

139620986780976 [label="fc1.weight
(60, 784)" fillcolor=lightblue]
139620986331320 -> 139620986332776
139620986331320 [label=TBackward]
139620986597616 -> 139620986331320
139620986597616 [label="fc2.weight
(60, 60)" fillcolor=lightblue]
139620986334232 -> 139620986333896
139620986334232 [label=TBackward]
139620986333728 -> 139620986334232
139620986333728 [label="fc3.weight
(10, 60)" fillcolor=lightblue]
}

Q14

Bonus: The 3 highest performance on the test set (f1score and accuracy) will get a bonus point.

My best model had a dropout layer (.2) & I checked it for overfitting. Also, it had the following parameters.

```
input_size = 28*28
hidden_size = 60
num_classes = 10
num_epochs = 20
batch_size = 128
learning_rate = 0.001

Layers - 3
Transfer Functions - ReLU (x3)
Criterion - CrossEntropyLoss
Optimizer - ADAM
Dropout - .2
```

Model Performance -

Precision, Recall, F1Score & Support of this model is (0.8770999999999999, 0.8766929119100281, **0.8754653253993478**, None)
++
Accuracy of the network on the 10000 test images: **87.71%**
Misclassification of the network on the 10000 test images: **12.29%**
Predicted: Dress Pullover Sneaker Sandal
++
Accuracy of T-shirt/top : 86 %
Accuracy of Trouser : 93 %
Accuracy of Pullover : 76 %
Accuracy of Dress : 87 %
Accuracy of Coat : 81 %
Accuracy of Sandal : 96 %
Accuracy of Shirt : 56 %
Accuracy of Sneaker : 98 %
Accuracy of Bag : 98 %
Accuracy of Ankle boot : 97 %
++
Execution time is **2.73996 minutes**