Deep Learning 2019

Assignment # 7

Report

Topic: Classification of Urdu MNIST Dataset

➢ VGG-16 Results:

I've performed three types of experiments with VGG-16:

1. I loaded pre-trained vgg16 model and performed experiments without removing last layer of 1000 classes and adding my own layer of 12 classes. I wanted to see if it works, and it does. I asserted the param.require_grad flag 'False' for parameters of all layers (feature and classifier), and used the pretrained model as it is without finetuning any of the layers.

2. I loaded pre-trained vgg16 model and then removed its last fc layer and added another fc layer with 12 classes. I asserted the param.require_grad flag 'False' for parameters of all layers (feature and classifier), and used the pretrained model as it is without finetuning any of the layers. The last layer in this case was initialized with random weights when I added it to the model and it wasn't finetuned either.

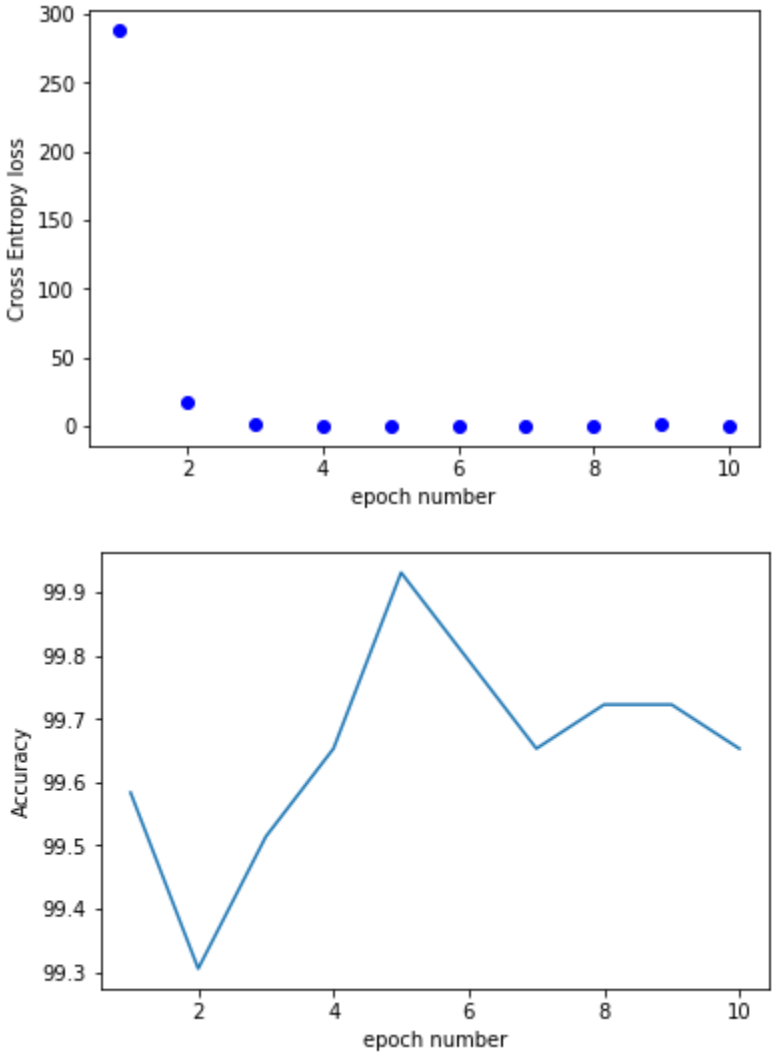
3. I loaded pre-trained vgg16 model and then removed its last fc layer and added another fc layer with 12 classes. I asserted the param.require_grad flag 'False' for parameters of feature layers only and finetuned the classifier layers for 10 epochs.

Experiment number	Accuracy
1.	99.3%
2.	98.9%
3.	100% *

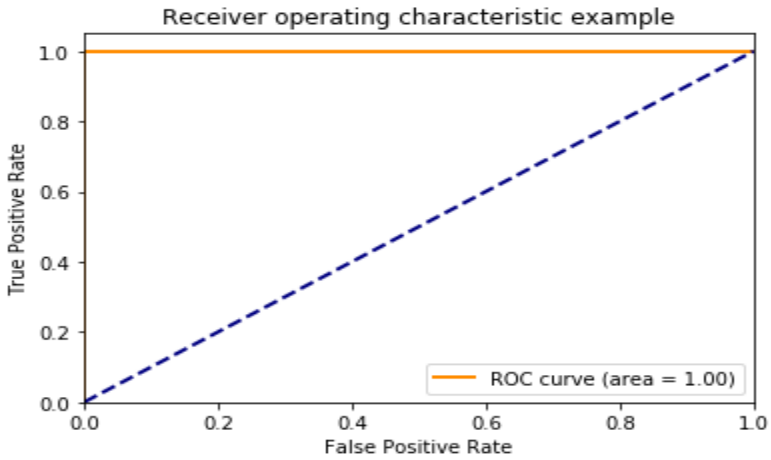
THE ACCURACY REACHED 100% IN THE 3RD EXPERIMENT BUT STARTED DECREASING, only slightly, IF TRAINING CONTIUED AFTER THAT.

Parameters and Results of finetuning experiment:

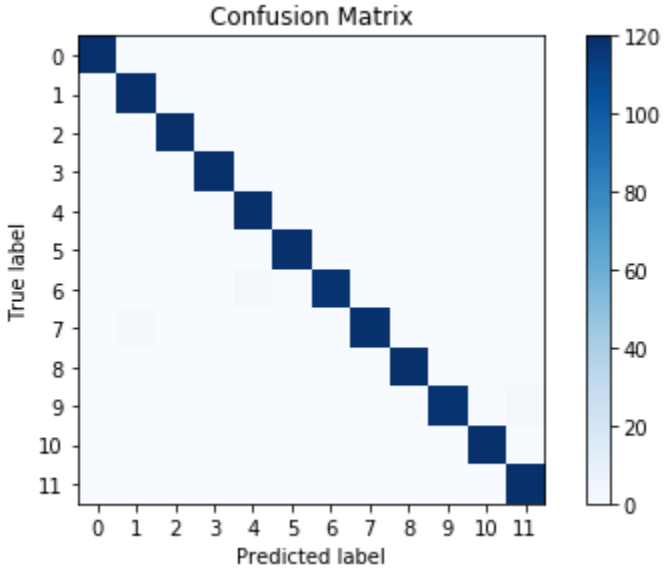
Learning Rate	0.001
Optimizer	SGD with momentum
momentum	0.9
Batch size	8
No. of epochs	10



It can be seen in above figure that accuracy reached 100%. These are results on test data.



```
[[120   0   0   0   0   0   0   0   0   0   0   0]
 [  0 120   0   0   0   0   0   0   0   0   0   0]
 [  0   0 120   0   0   0   0   0   0   0   0   0]
 [  0   0   0 120   0   0   0   0   0   0   0   0]
 [  0   0   0   0 120   0   0   0   0   0   0   0]
 [  0   0   0   0   0 120   0   0   0   0   0   0]
 [  0   0   0   0   2   0 118   0   0   0   0   0]
 [  0   1   0   0   0   0   0 119   0   0   0   0]
 [  0   0   0   0   0   0   0   0 120   0   0   0]
 [  0   0   0   0   0   0   0   0   0 118   0   2]
 [  0   0   0   0   0   0   0   0   0   0 120   0]
 [  0   0   0   0   0   0   0   0   0   0   0 120]]
```



➢RESNET-50 Results:

I've performed three types of experiments with Resnet-50:

4. I loaded pre-trained resnet-50 model and performed experiments without removing last layer of 1000 classes and adding my own layer of 12 classes. I wanted to see if it works, and it does. I asserted the param.require_grad flag 'False' for parameters of all layers (feature and classifier), and used the pretrained model as it is without finetuning any of the layers.

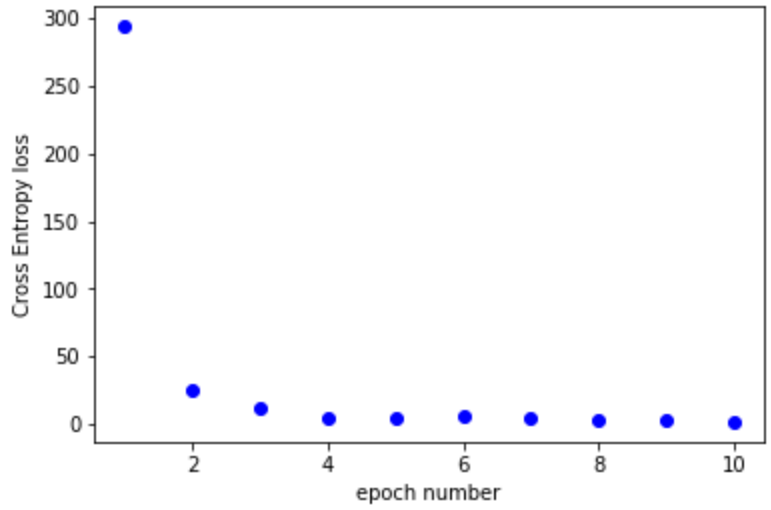
5. I loaded pre-trained resnet-50 model and then removed its last fc layer and added another fc layer with 12 classes. I asserted the param.require_grad flag 'False' for parameters of all layers (feature and classifier), and used the pretrained model as it is without finetuning any of the layers. The last layer in this case was initialized with random weights when I added it to the model and it wasn't finetuned either.

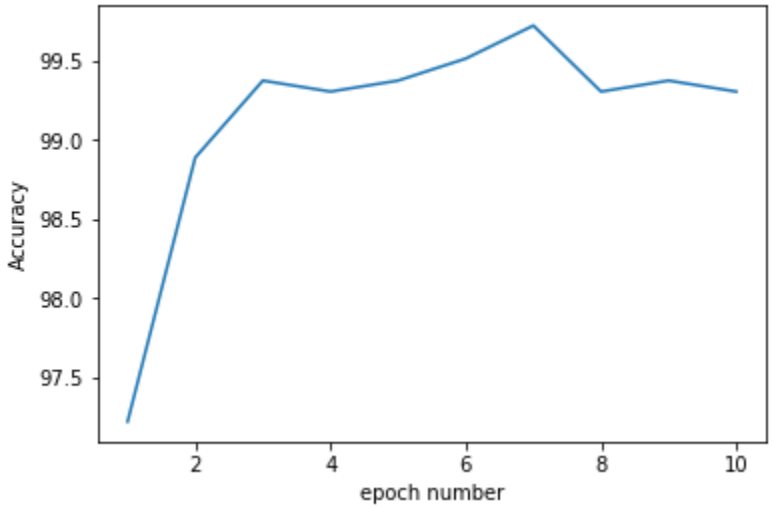
6. I loaded pre-trained resnet-50 model and then removed its last fc layer and added another fc layer with 12 classes. I finetuned the complete model for 10 epochs.

Experiment number	Accuracy
1.	98%
2.	98.8%
3.	99.5%

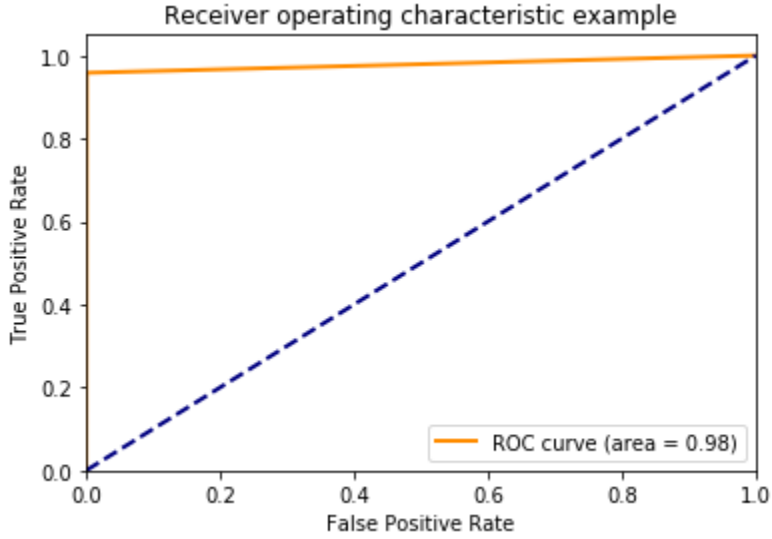
Parameters and Results of finetuning experiment:

Learning Rate	0.001
Optimizer	SGD with momentum
momentum	0.9
Batch size	8
No. of epochs	10

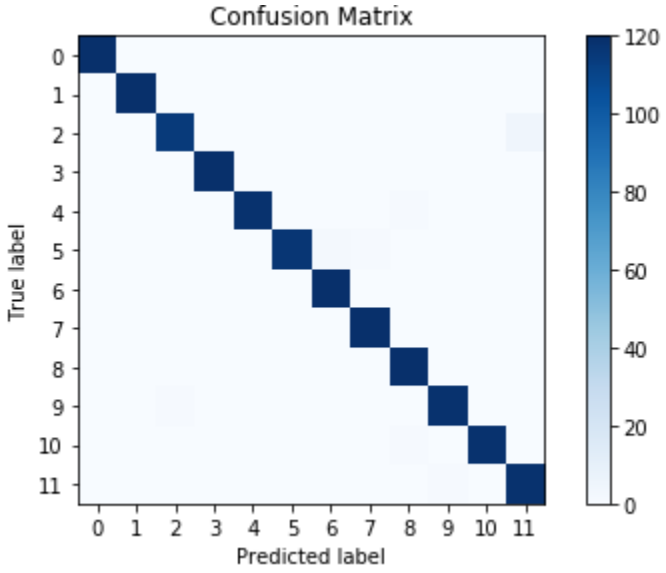




It can be seen in above figure that accuracy reached >99.5%. These are results on test data.



```
[[120   0   0   0   0   0   0   0   0   0   0   0]
 [  0 120   0   0   0   0   0   0   0   0   0   0]
 [  0   0 115   0   0   0   0   0   0   0   0   5]
 [  0   0   0 120   0   0   0   0   0   0   0   0]
 [  0   0   0   0 119   0   0   0   1   0   0   0]
 [  0   0   0   0   0 117   2   1   0   0   0   0]
 [  0   0   0   0   0   0 120   0   0   0   0   0]
 [  0   0   0   0   0   0   0 120   0   0   0   0]
 [  0   0   0   0   0   0   0   0 120   0   0   0]
 [  0   0   1   0   0   0   0   0   0 119   0   0]
 [  0   0   0   0   0   0   0   0   1   0 119   0]
 [  0   0   0   0   0   0   0   0   0   1   0 119]]
```



➢ My Own Network:

NOTE: I've resized images to size (64,64,3) using torch.transforms for all the experiments reported below.

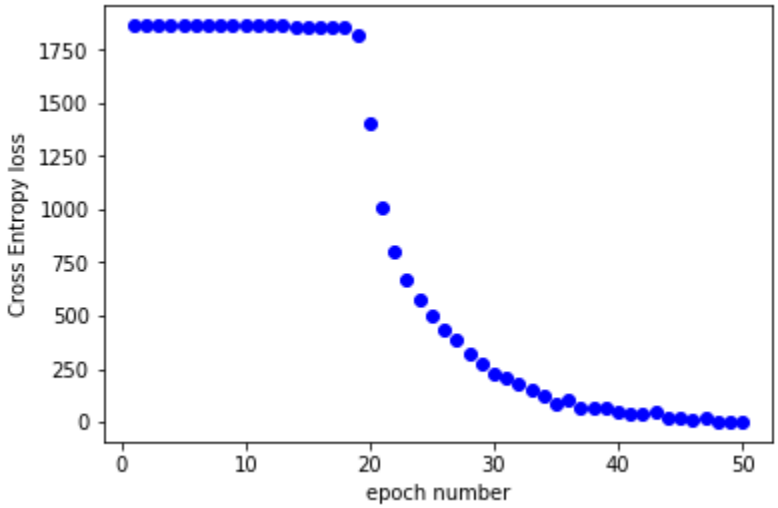
➢ Experiment 1 (Without Mean Image Subtraction & Dropout):

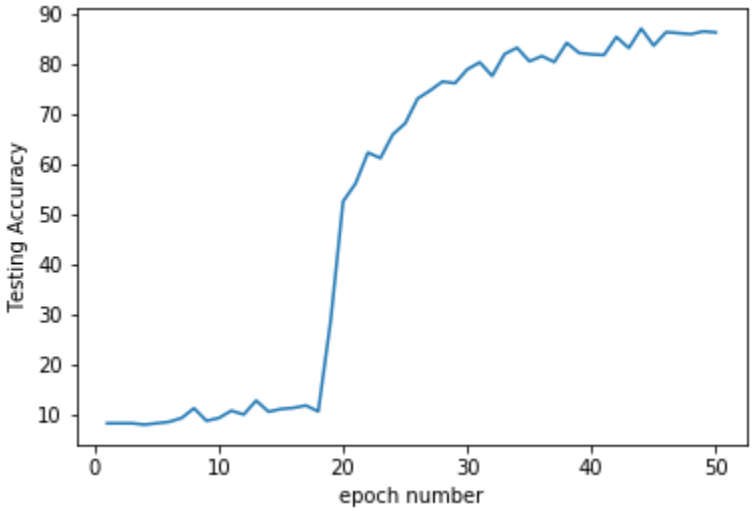
My first network has the following architecture:

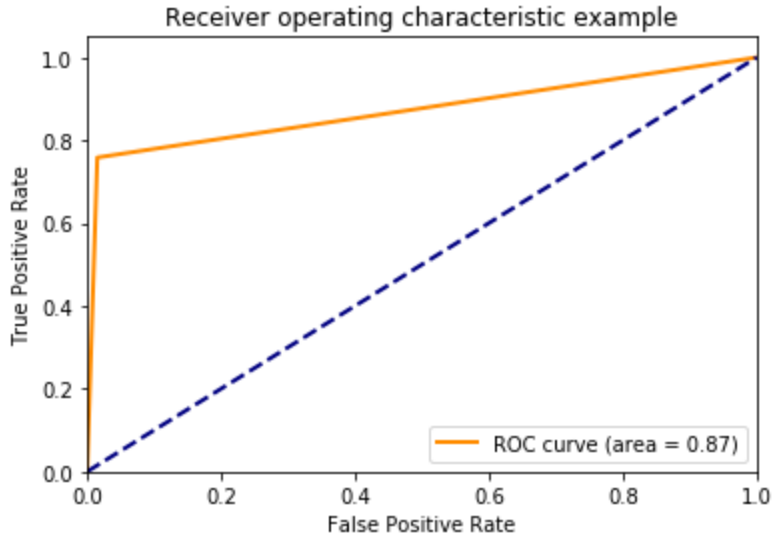
```
class My_Network(nn.Module):
    def __init__(self):
        super(My_Network, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 12, kernel_size=3),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),
            nn.Conv2d(12, 24, kernel_size=3),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),
            nn.Conv2d(24, 20, kernel_size=3),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),

        )
        self.fc = nn.Sequential(
            nn.Linear(20*6*6, 256),
            nn.ReLU(),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Linear(128, 12),
        )
        return

    def forward(self, x):
        x = self.features(x)
        #print((x.shape))
        x = x.view(-1, 20*6*6)
        x = self.fc(x)
        return x
```

Accuracy = 86.25

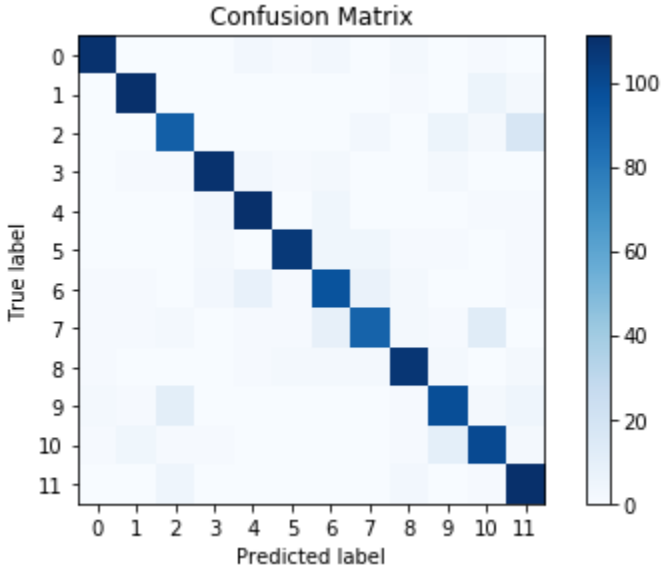
Learning Rate	0.001
Optimizer	SGD with momentum
momentum	0.9
Batch size	8
No. of epochs	50


Cross Entropy loss
epoch number
0
250
500
750
1000
1250
1500
1750
0
10
20
30
40
50


Testing Accuracy
epoch number
10
20
30
40
50
60
70
80
90
0
10
20
30
40
50




```
[[110   0   0   0   3   1   3   0   2   0   1   0]
 [  0 111   0   0   0   0   0   0   1   0   6   2]
 [  0   0  91   0   0   0   0   3   0   6   2  18]
 [  0   1   1 110   3   1   2   0   0   2   0   0]
 [  0   0   0   3 111   0   4   0   0   0   1   1]
 [  0   0   0   2   0 107   4   4   1   1   0   1]
 [  1   1   0   3   8   1  96   7   2   0   0   1]
 [  1   1   2   0   1   1   9  89   2   1  13   0]
 [  1   0   0   0   1   2   2   2 108   2   0   2]
 [  2   1  11   0   0   0   0   0   1  98   2   5]
 [  1   4   1   1   0   0   0   0   1  10 100   2]
 [  0   0   5   0   0   0   0   0   3   0   1 111]]
```



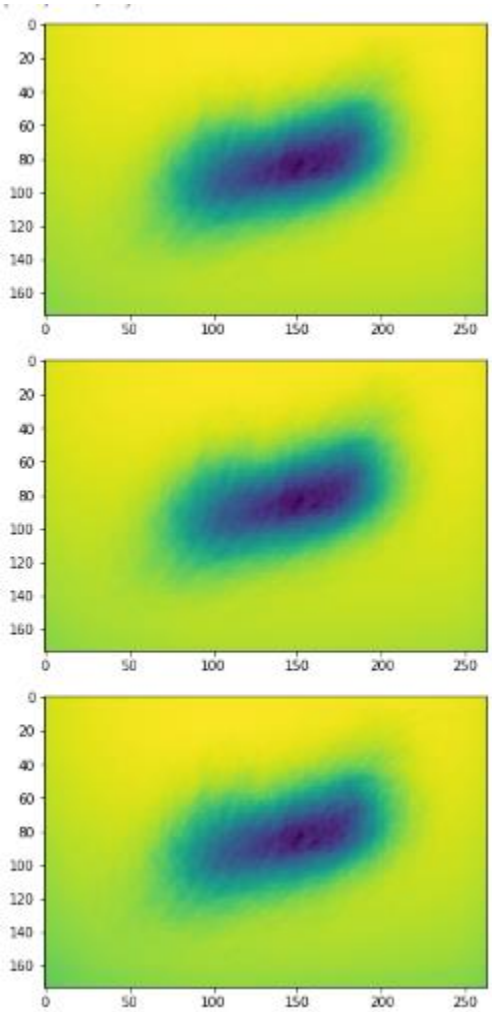
➢ Experiment 2 (With Mean Image Subtraction):

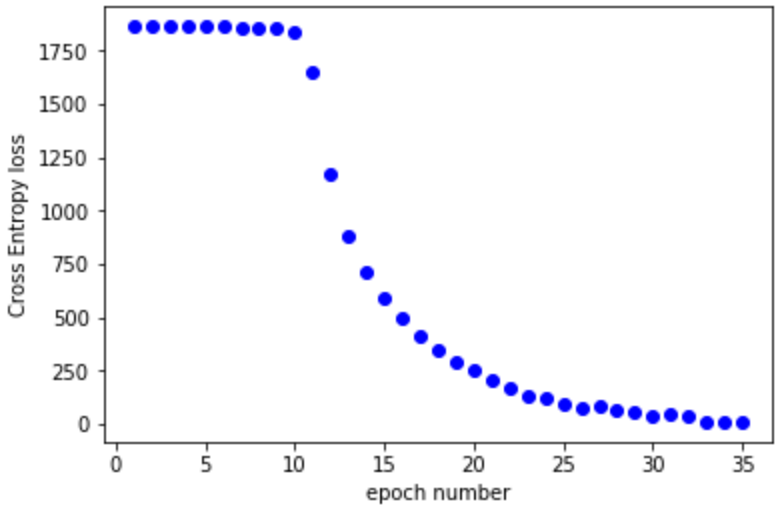
Findings: Accuracy has decreased

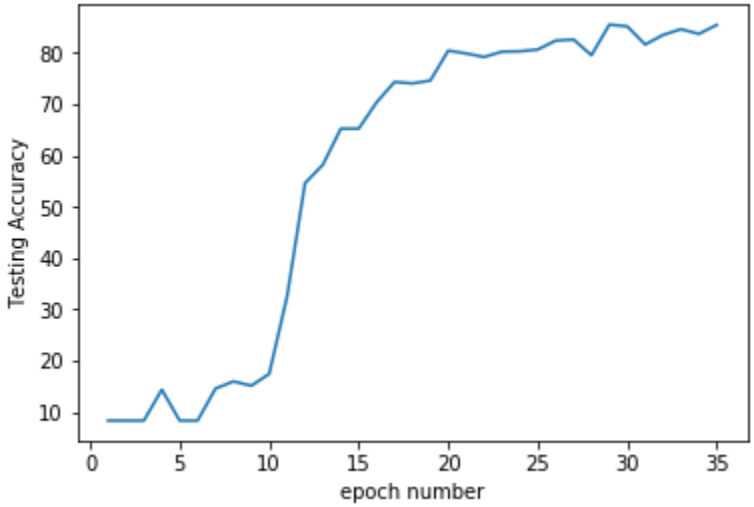
Accuracy = 83.88

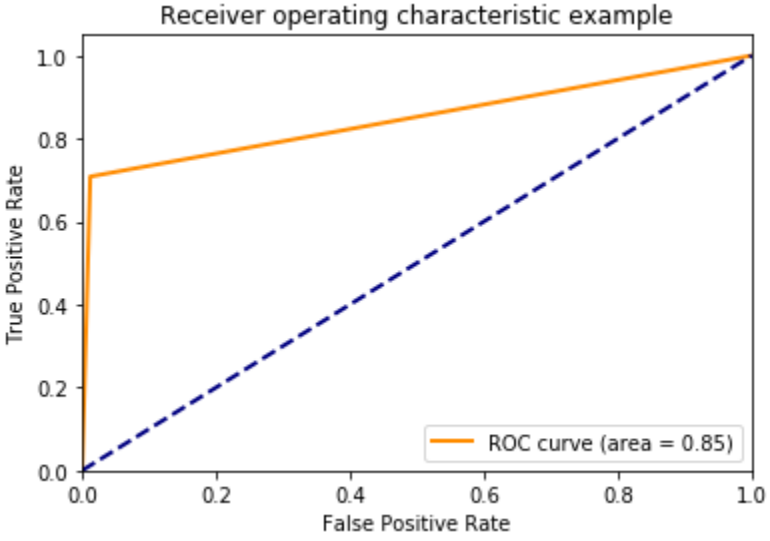
Learning Rate	0.001
Optimizer	SGD with momentum
momentum	0.9
Batch size	8
No. of epochs	35

Mean Images:

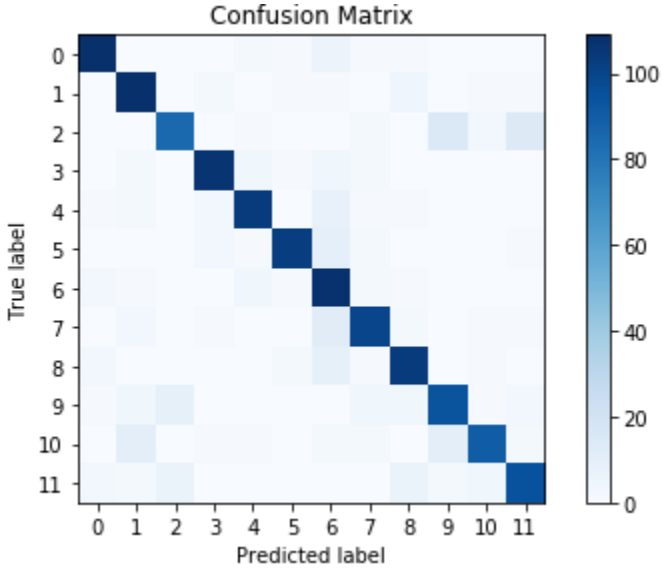



Cross Entropy loss
epoch number


Testing Accuracy
epoch number


Receiver operating characteristic example
True Positive Rate
False Positive Rate
ROC curve (area = 0.85)

```
[[109   0   0   0   2   1   6   1   1   0   0   0]
 [  0 109   0   2   0   1   1   0   5   0   1   1]
 [  0   0  85   0   1   0   0   2   0  15   3  14]
 [  0   2   0 107   4   1   4   2   0   0   0   0]
 [  1   2   0   3 104   0   8   1   1   0   0   0]
 [  0   0   0   3   1 103  10   2   0   0   0   1]
 [  3   1   0   0   4   1 108   2   1   0   0   0]
 [  0   3   0   1   0   0  12 100   2   0   1   1]
 [  3   0   0   0   0   2   9   1 104   0   1   0]
 [  1   4   9   0   0   0   0   4   4  94   1   3]
 [  0  11   0   1   1   0   2   2   0  11  90   2]
 [  3   2   7   0   0   0   0   0   7   2   4  95]]
```



➢ Experiment 3 (With Batch Normalization Layers):

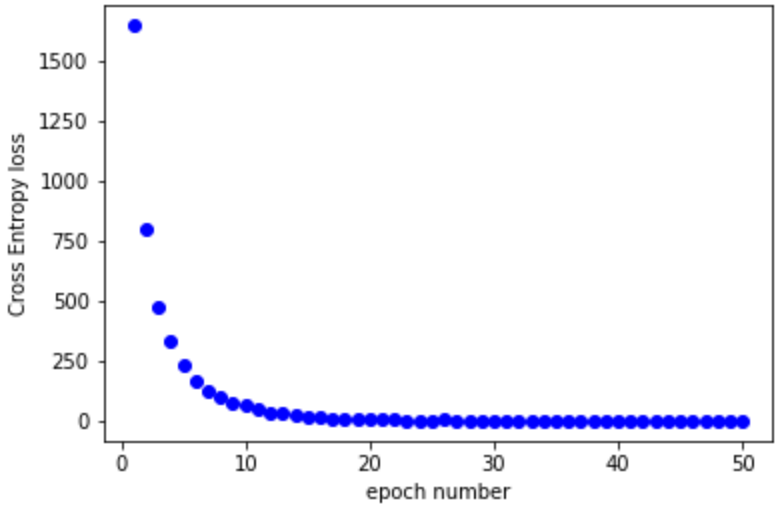
Following is the network architecture with BN-layers:

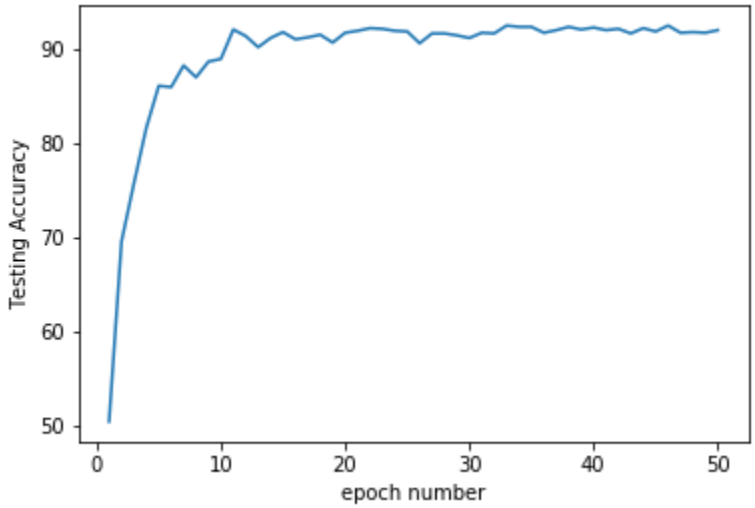
```
class My_Network_BN(nn.Module):
    def __init__(self):
        super(My_Network_BN, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 12, kernel_size=3),
            nn.BatchNorm2d(12),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),
            nn.Conv2d(12, 24, kernel_size=3),
            nn.BatchNorm2d(24),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),
            nn.Conv2d(24, 20, kernel_size=3),
            nn.BatchNorm2d(20),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),

        )
        self.fc = nn.Sequential(
            nn.Linear(20*6*6, 256),
            nn.ReLU(),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Linear(128, 12),
        )
        return

    def forward(self, x):
        x = self.features(x)
        #print((x.shape))
        x = x.view(-1, 20*6*6)
        x = self.fc(x)
        return x
```

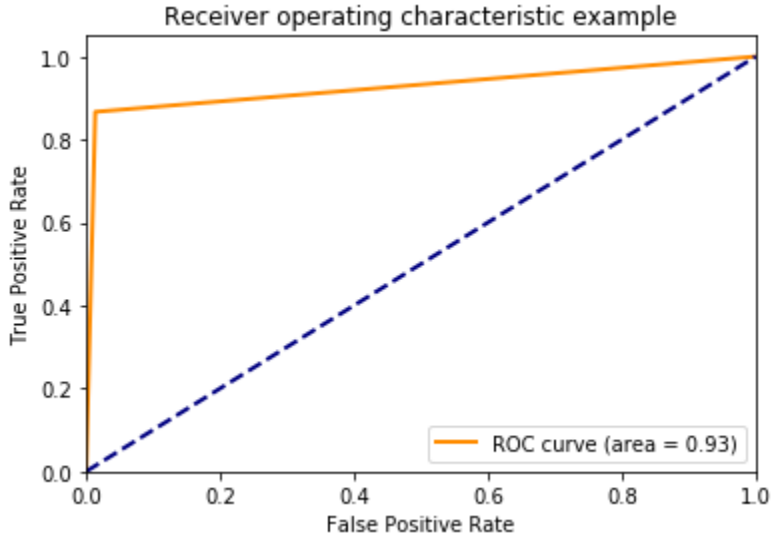
Findings: Accuracy has increased

Accuracy = 92.5

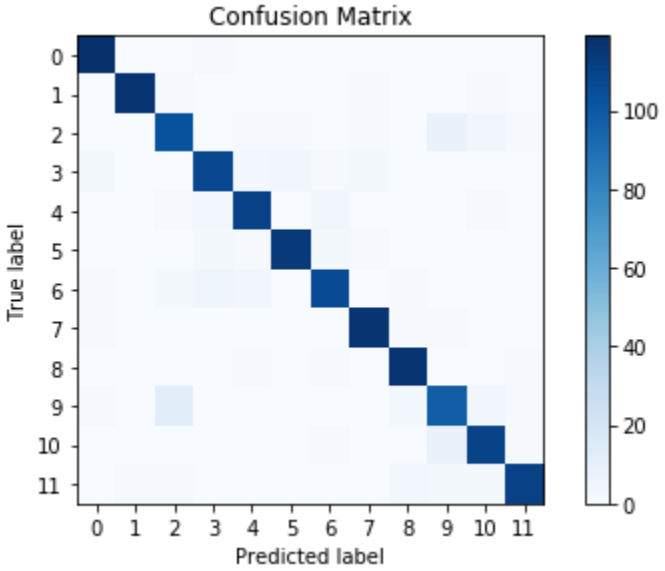
Learning Rate	0.001
Optimizer	SGD with momentum
momentum	0.9
Batch size	8
No. of epochs	50


Cross Entropy loss
epoch number


Testing Accuracy
epoch number


Receiver operating characteristic example
True Positive Rate
False Positive Rate
ROC curve (area = 0.93)

```
[[119   0   0   1   0   0   0   0   0   0   0   0]
 [  0 117   1   0   0   0   0   1   0   0   1   0]
 [  0   0 104   0   1   1   0   1   0   8   4   1]
 [  2   0   0 108   3   4   1   2   0   0   0   0]
 [  0   0   1   3 111   0   4   0   0   0   1   0]
 [  0   0   0   2   1 114   2   1   0   0   0   0]
 [  1   0   2   6   4   0 106   0   1   0   0   0]
 [  1   0   0   0   0   0   0 117   1   1   0   0]
 [  0   0   0   0   1   0   1   0 117   0   0   1]
 [  1   0  13   0   0   0   0   0   3  98   4   1]
 [  0   0   0   0   0   0   1   0   0   8 110   1]
 [  0   1   1   0   0   0   0   0   3   2   2 111]]
```



➢ Experiment 4 (With Dropout Layers):

Following is the network architecture with Dropout-layers:

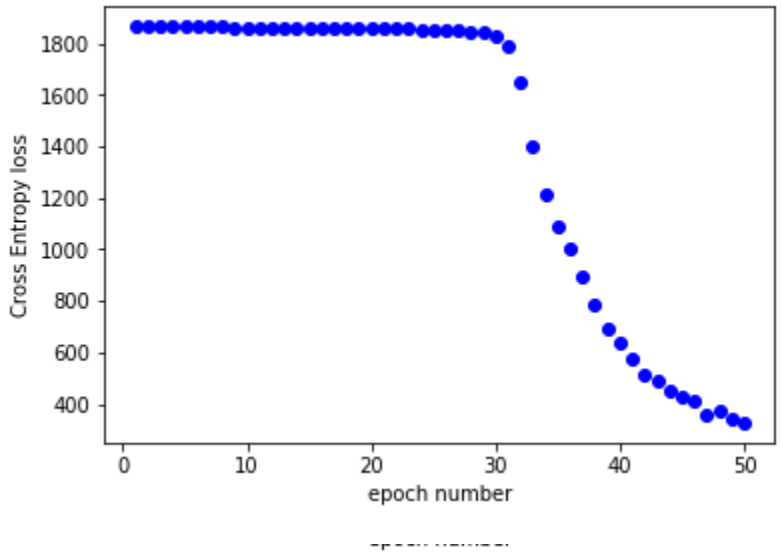
```
class My_Network_dropout(nn.Module):
    def __init__(self):
        super(My_Network_dropout, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 12, kernel_size=3),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),
            nn.Dropout(0.2),
            nn.Conv2d(12, 24, kernel_size=3),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),
            nn.Dropout(0.2),
            nn.Conv2d(24, 20, kernel_size=3),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),
            nn.Dropout(0.2),


        )
        self.fc = nn.Sequential(
            nn.Linear(20*6*6, 256),
            nn.ReLU(),
            nn.Dropout(0.1),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Linear(128, 12),
        )
        return

    def forward(self, x):
        x = self.features(x)
        #print((x.shape))
        x = x.view(-1, 20*6*6)
        x = self.fc(x)
        return x
```

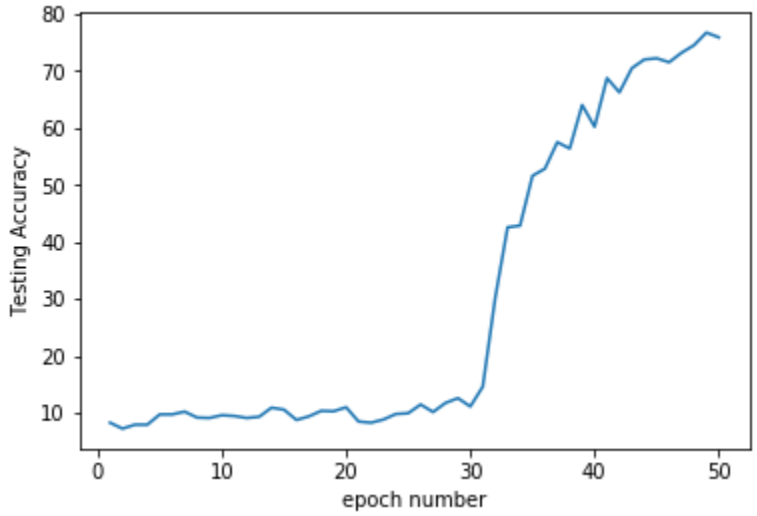
Findings: Accuracy has decreased. Model achieved accuracy of 75.8 in 50 epochs, though earlier it achieved accuracy of 86% without batch normalization in 50 epochs and an accuracy of 92.5% with batch normalization in 50 epochs. But with dropout accuracy didn't saturate at 50 epochs, while in other experiments it did. So, I trained the model for another 15 epochs and accuracy increased to 78% and I believe it would increase further if training is performed for more epochs but I didn't train the model any further.

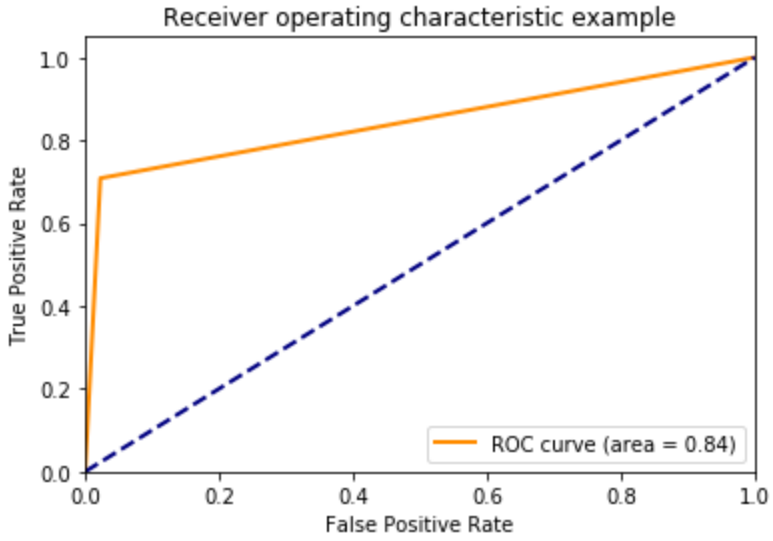
Also, I've used dropout after every convolution layer, I believe if I use dropout in fc layers only, accuracy will be more than the no-dropout case as convolutional layers learn features whereas fc layers learn classification, so adding dropout on fc-layers will help learning a more generic classifier which will perform better on unseen data.

Accuracy = 78.3%

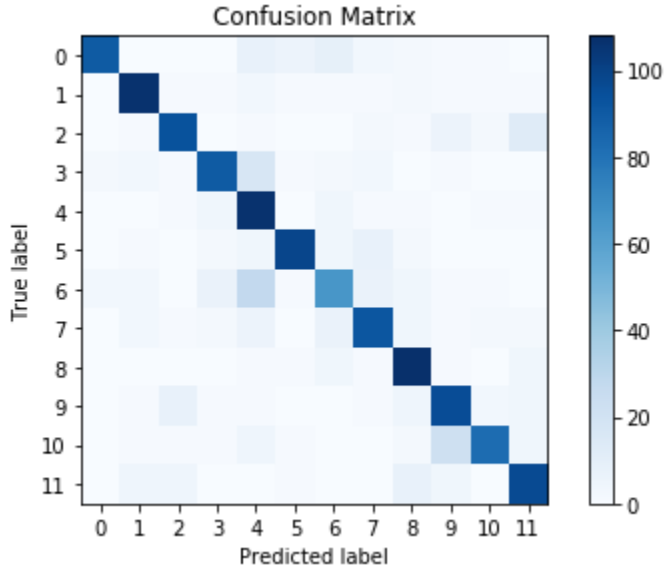
Learning Rate	0.001
Optimizer	SGD with momentum
momentum	0.9
Batch size	8
No. of epochs	50








```
[[ 90   0   0   0   8   6   9   3   2   1   1   0]
 [  0 107   1   1   3   1   1   1   2   1   1   1]
 [  0   1  94   0   1   0   0   2   1   6   2  13]
 [  2   3   1  90  17   1   2   3   0   1   0   0]
 [  0   0   1   4 107   0   4   1   1   0   1   1]
 [  0   1   0   2   4  99   4   8   2   0   0   0]
 [  3   3   0   7  28   1  65   7   4   1   1   0]
 [  0   3   1   2   6   0   7  92   4   1   2   2]
 [  0   0   0   0   1   1   4   1 108   1   0   4]
 [  0   1   8   1   1   0   0   1   5  96   3   4]
 [  0   1   1   1   5   1   0   0   2  22  83   4]
 [  0   5   5   0   0   1   0   0   8   4   0  97]]
```



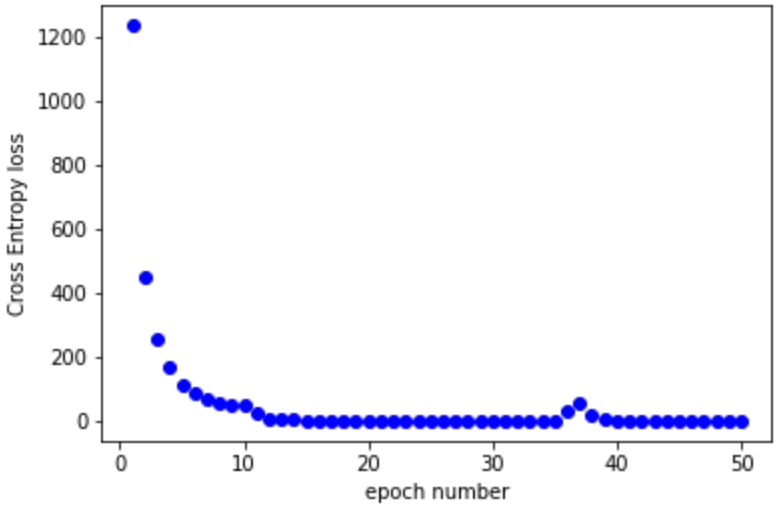
➢ Experiment 5 (Different Learning Rates):

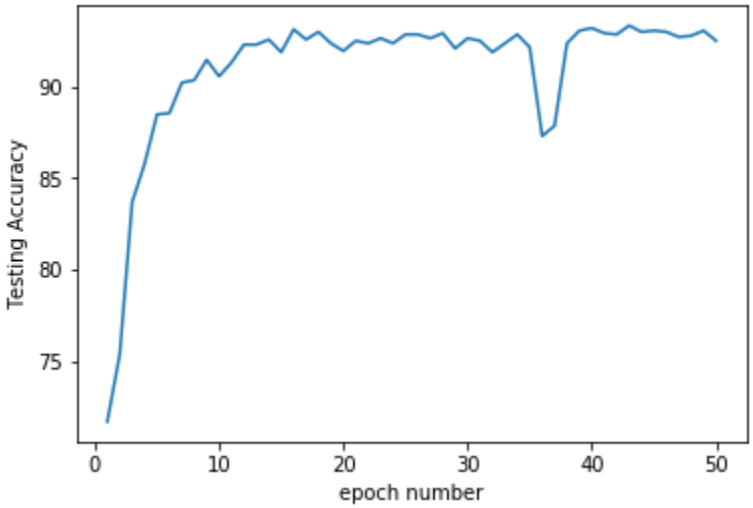
Here I'm using the network which includes batch-normalization layers.

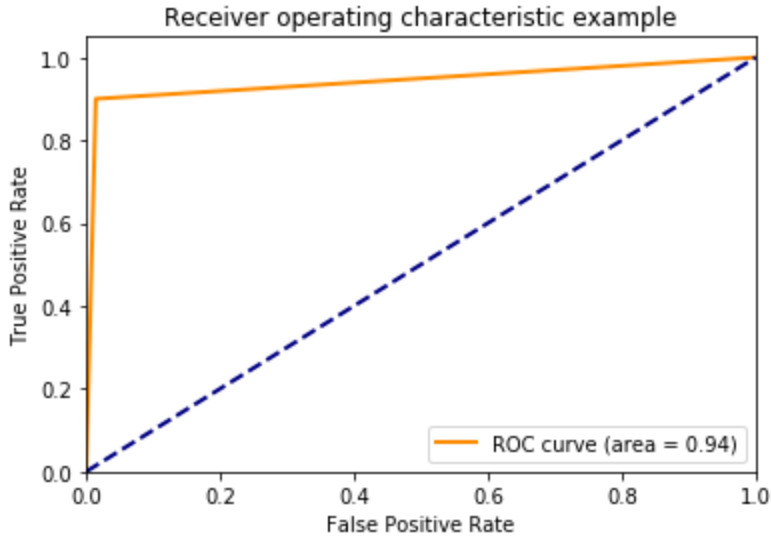
Experiment 5a. Learning rate = 0.003

Accuracy = 93.125% (higher accuracy achieved in 50 epochs)

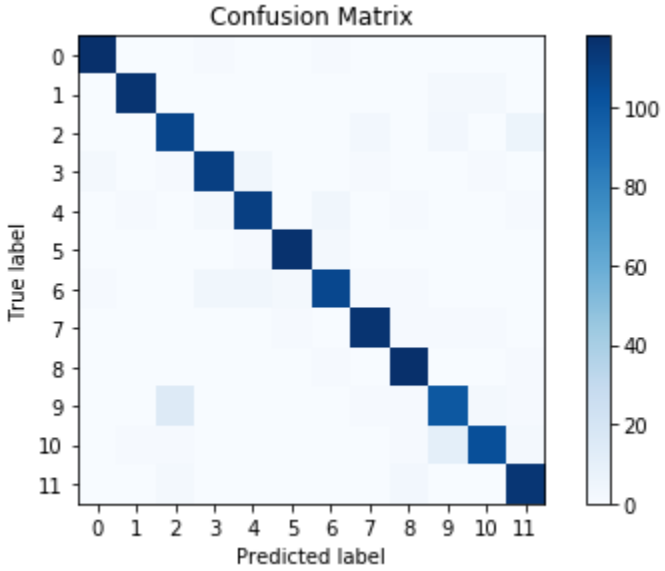
Learning Rate	0.003
Optimizer	SGD with momentum
momentum	0.9
Batch size	8
No. of epochs	50







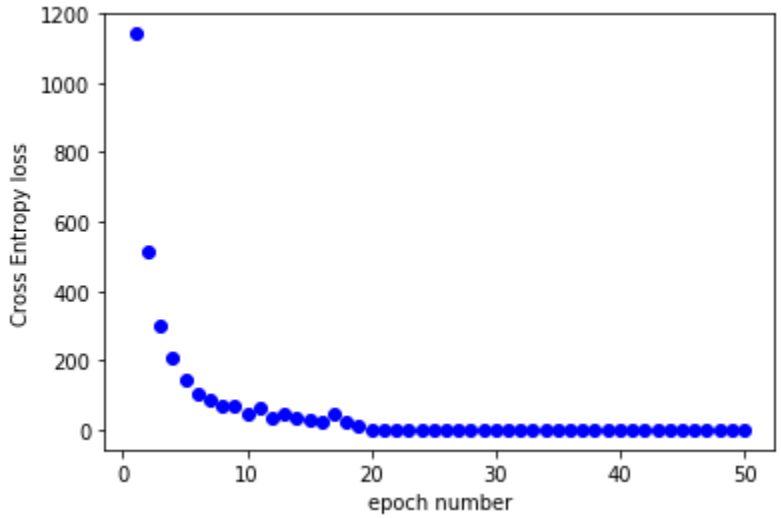

```
[[118   0   0   1   0   0   1   0   0   0   0   0]
 [  0 116   0   0   0   0   0   0   0   2   2   0]
 [  0   0 108   0   0   0   0   3   0   3   0   6]
 [  2   0   1 111   4   0   0   1   0   0   1   0]
 [  0   1   0   2 111   0   4   0   1   0   0   1]
 [  0   0   0   0   1 117   2   0   0   0   0   0]
 [  1   0   0   4   4   2 107   1   1   0   0   0]
 [  0   0   0   0   0   1   0 116   1   1   1   0]
 [  0   0   0   0   0   0   1   0 118   0   0   1]
 [  0   0  15   0   0   0   0   1   1 100   2   1]
 [  0   1   1   0   0   0   0   0   1  11 104   2]
 [  0   0   2   0   0   0   0   0   3   0   0 115]]
```

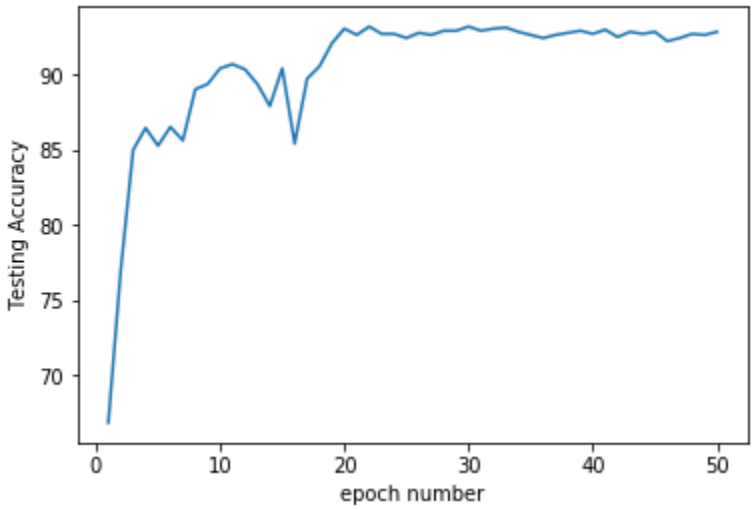


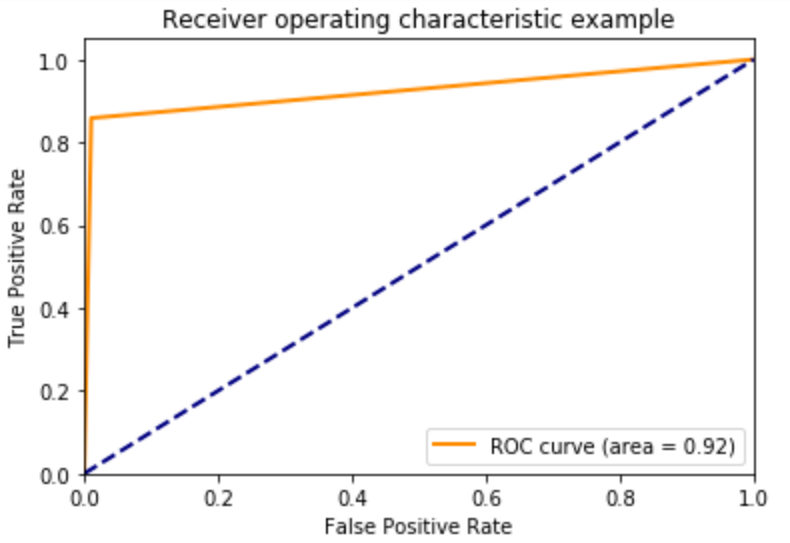
Experiment 5b. Learning rate = 0.009

Accuracy = 92.63%

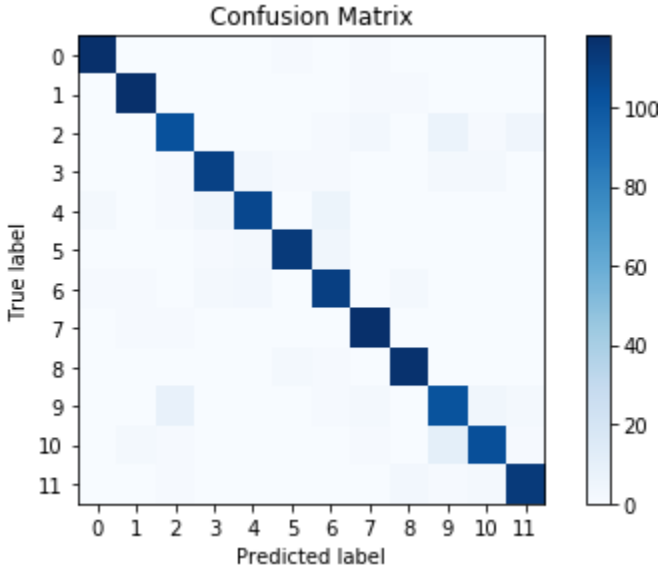
Learning Rate	0.009
Optimizer	SGD with momentum
momentum	0.9
Batch size	8
No. of epochs	50








```
[[118   0   0   0   0   1   0   1   0   0   0   0]
 [  0 118   0   0   0   0   0   1   1   0   0   0]
 [  0   0 103   0   0   0   1   3   0   7   1   5]
 [  0   0   1 110   3   1   1   0   0   2   2   0]
 [  2   0   1   4 107   0   6   0   0   0   0   0]
 [  0   0   0   1   2 113   4   0   0   0   0   0]
 [  1   1   0   2   3   0 111   0   2   0   0   0]
 [  0   1   1   0   0   0   0 118   0   0   0   0]
 [  0   0   0   0   0   2   1   0 117   0   0   0]
 [  0   0   9   0   0   0   1   2   0 102   4   2]
 [  0   2   1   0   0   0   0   1   0  11 104   1]
 [  0   0   1   0   0   0   0   0   3   1   2 113]]
```



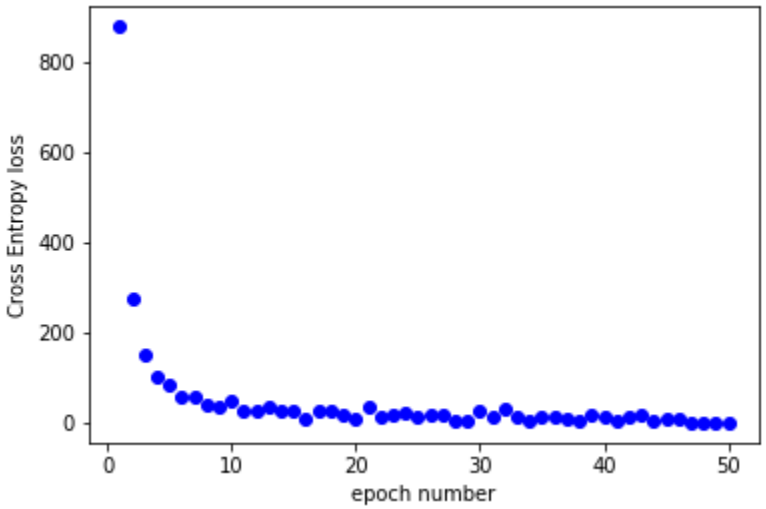
➢ Experiment 6 (Different Optimization Functions):

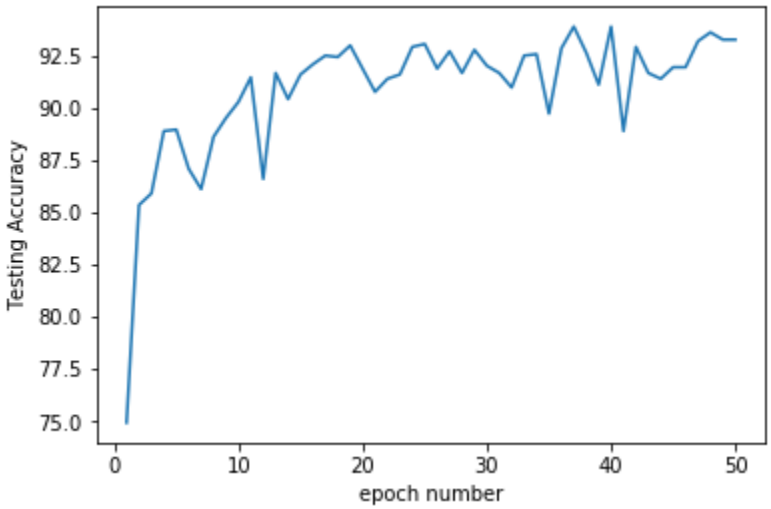
Here I'm using the network which includes batch-normalization layers.

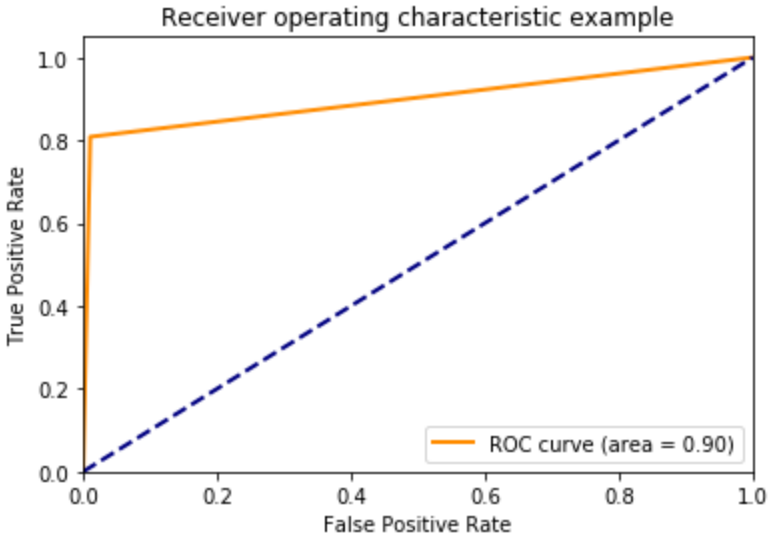
Experiment 6a. Optimizer = Adam

Accuracy = 93.47%

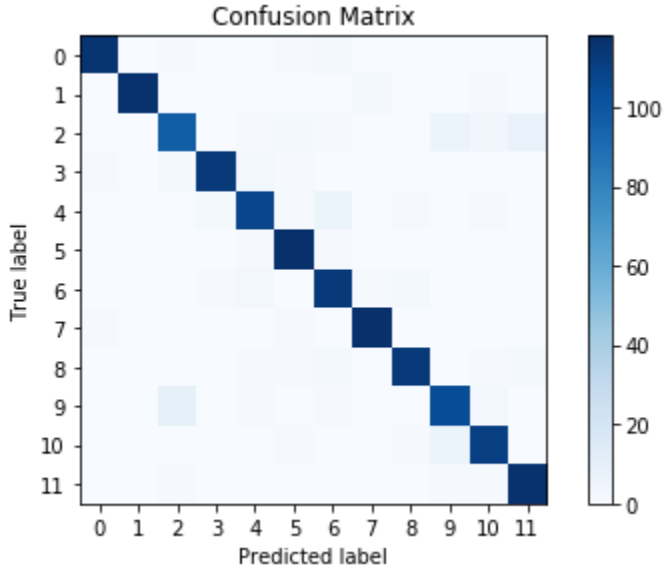
Learning Rate	0.001
Optimizer	Adam
Betas	0.9, 0.999
Batch size	8
No. of epochs	50







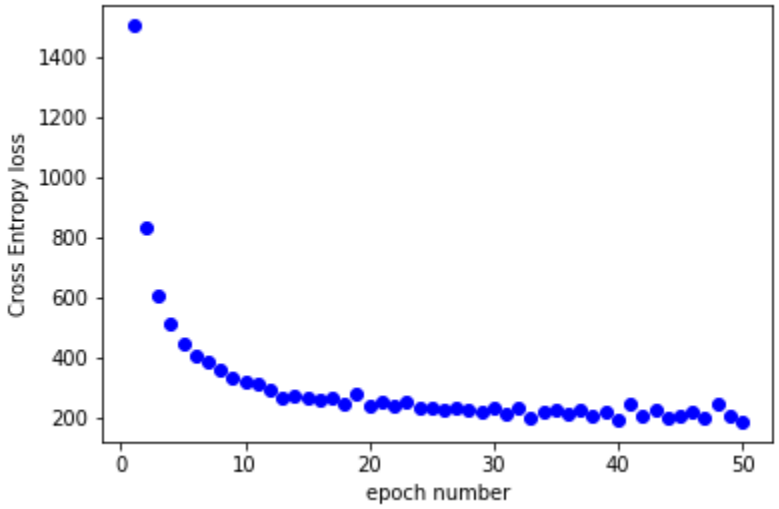

```
[[116   0   1   0   0   1   2   0   0   0   0   0]
 [  0 117   0   0   0   0   0   2   0   0   1   0]
 [  0   0  97   0   1   2   1   0   0   7   4   8]
 [  1   0   2 113   3   1   0   0   0   0   0   0]
 [  0   0   0   2 108   1   7   0   1   0   1   0]
 [  0   0   0   0   1 118   1   0   0   0   0   0]
 [  0   0   0   1   3   0 113   1   2   0   0   0]
 [  1   0   0   0   0   1   0 118   0   0   0   0]
 [  0   0   0   0   1   1   2   0 113   0   1   2]
 [  0   0  10   0   1   0   1   0   0 105   3   0]
 [  0   0   0   0   0   1   0   0   1   7 111   0]
 [  0   0   1   0   0   0   0   0   0   1   1 117]]
```

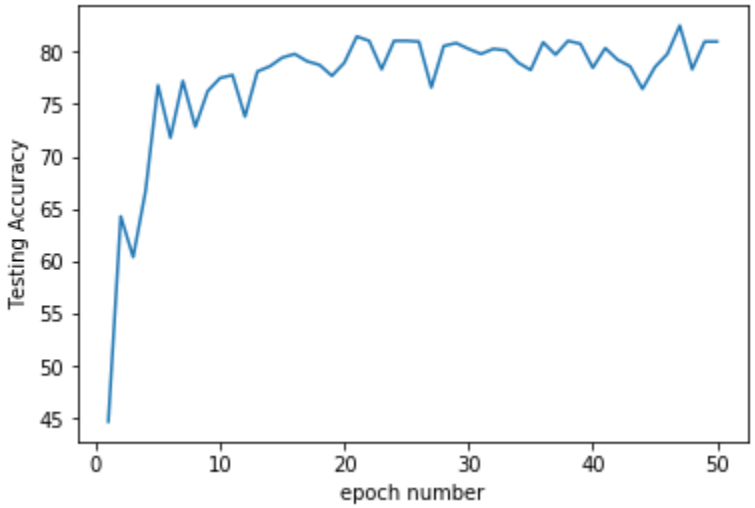


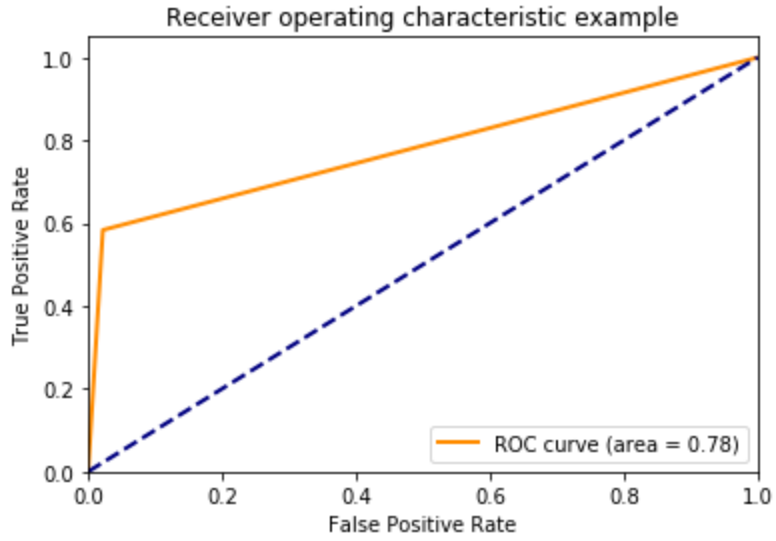
Experiment 6b. Optimizer = RMSprop

Accuracy = 81.52%

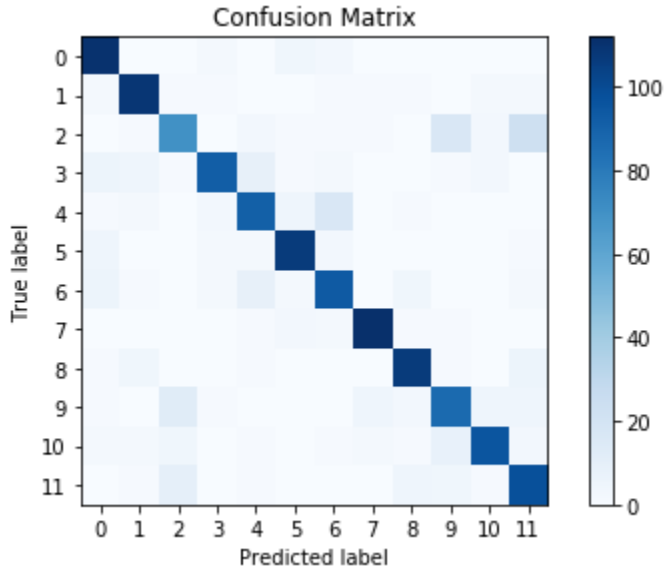
Learning Rate	0.001
Optimizer	RMSprop
Alpha	0.99
Momentum	0.9
Batch size	8
No. of epochs	50








```
[[111   0   0   2   0   4   3   0   0   0   0   0]
 [  2 109   1   1   0   0   1   1   1   0   2   2]
 [  0   1  70   0   3   1   1   1   0  17   3  23]
 [  6   5   1  92   9   1   2   0   0   1   3   0]
 [  1   2   0   3  91   5  17   0   1   0   0   0]
 [  5   0   0   2   2 107   3   0   0   0   0   1]
 [  6   1   0   2   9   2  94   0   4   0   0   2]
 [  0   0   0   0   1   3   2 112   1   1   0   0]
 [  1   4   0   0   1   0   0   0 107   1   0   6]
 [  1   0  13   1   0   0   0   5   3  87   5   5]
 [  2   2   4   0   1   0   1   2   1   8  96   3]
 [  0   1  10   0   1   0   0   0   5   4   1  98]]
```



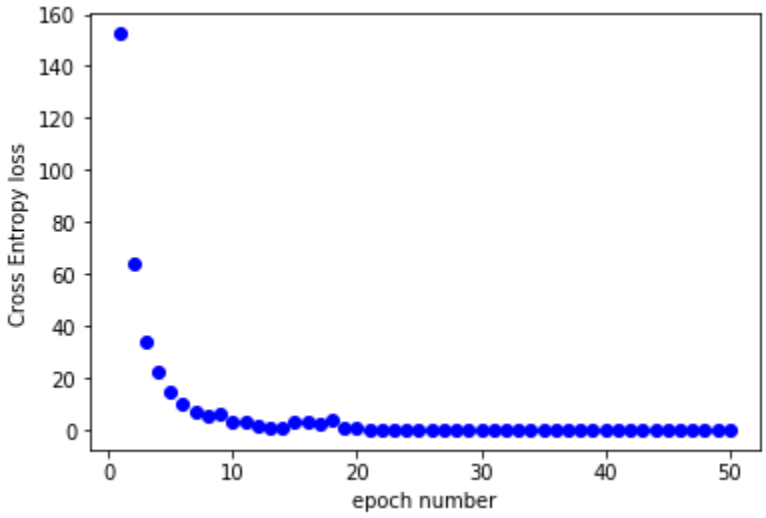
➢ Experiment 7 (Different Batch Sizes):

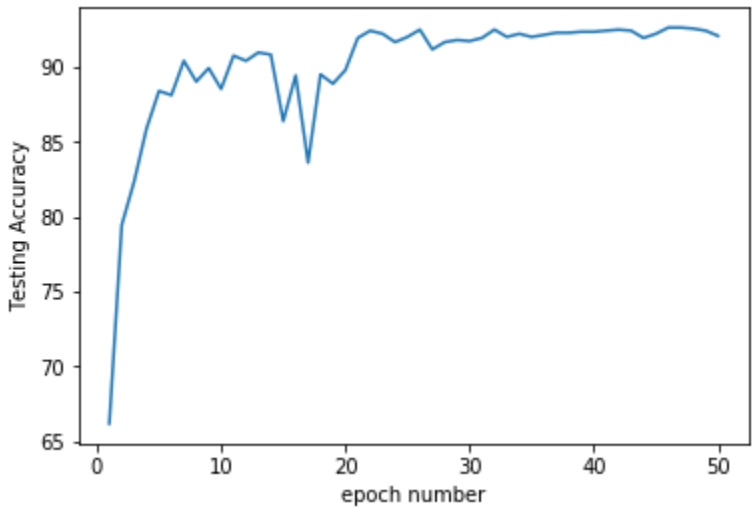
Here I'm using the network which includes batch-normalization layers. Earlier I performed same experiment with batch size-8, optimizer-Adam and achieved accuracy of 93.47% and here I'll report results of two more batch sizes.
ACCURACY IS DECREASING BY INCREASING BATCH SIZE. (POSSIBLE REASON COULD BE ONLY 500 IMAGES/CLASS.)

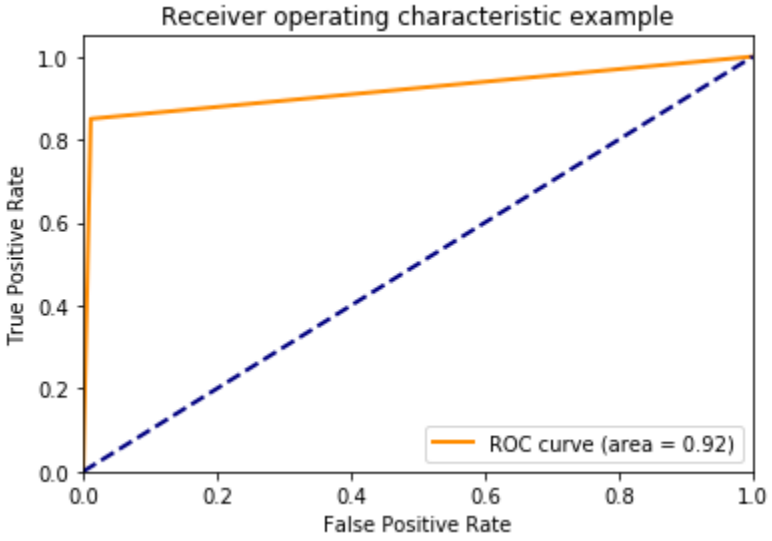
Experiment 7a. Batch Size = 64

Accuracy = 92.5%

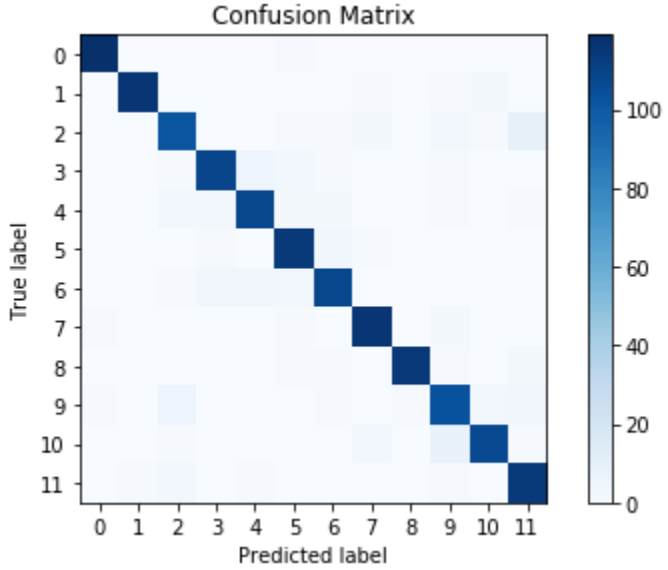
Learning Rate	0.001
Optimizer	Adam
Betas	0.9, 0.999
Batch size	64
No. of epochs	50







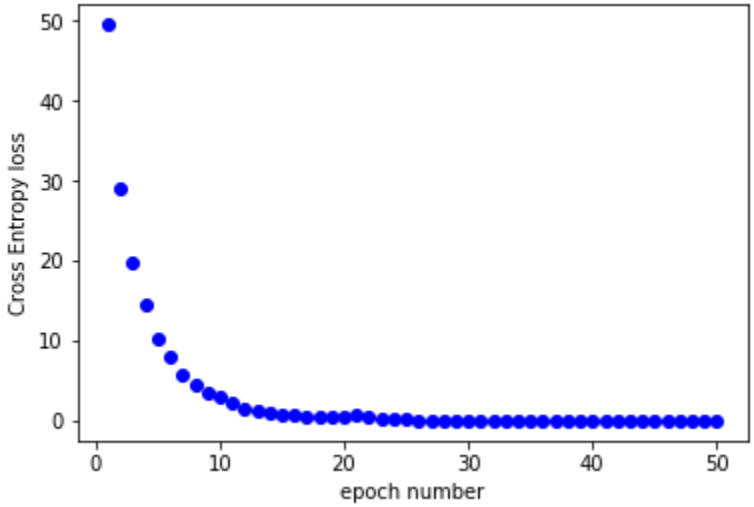

```
[[119   0   0   0   0   1   0   0   0   0   0   0]
 [  0 116   0   0   0   0   0   1   0   1   2   0]
 [  0   0 102   0   0   1   1   2   0   3   1  10]
 [  0   0   1 109   5   3   1   0   0   1   0   0]
 [  0   0   3   3 108   2   2   0   0   1   0   1]
 [  0   0   0   1   0 114   4   1   0   0   0   0]
 [  0   0   1   4   4   3 108   0   0   0   0   0]
 [  1   0   0   0   0   1   0 116   0   2   0   0]
 [  0   0   0   0   0   1   1   0 115   1   0   2]
 [  1   0   6   0   0   0   1   0   1 104   3   4]
 [  0   0   1   0   0   0   0   3   0   8 107   1]
 [  0   1   3   0   1   0   0   0   0   1   0 114]]
```

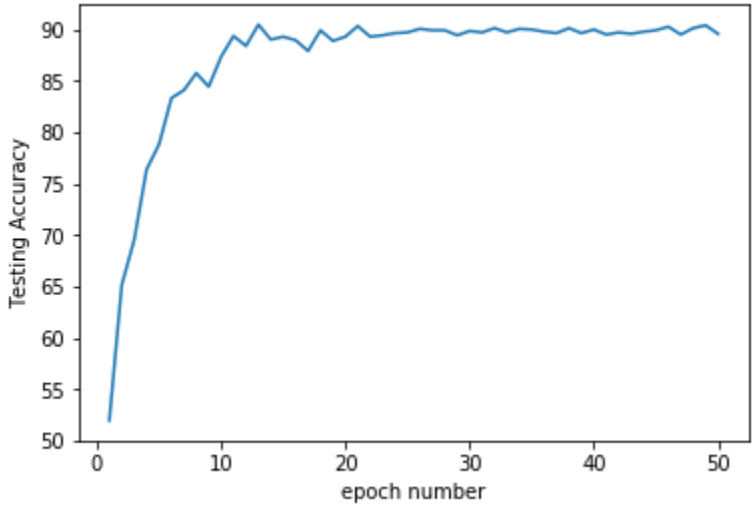


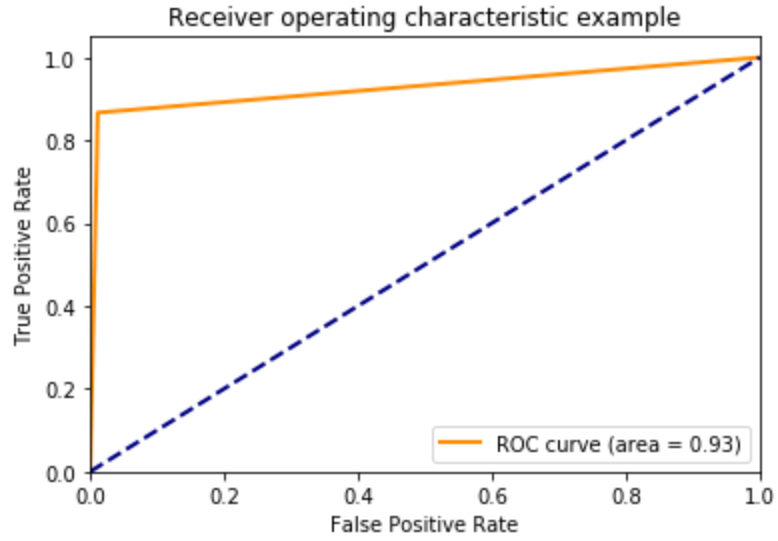
Experiment 7b. Batch Size = 256

Accuracy = 90%

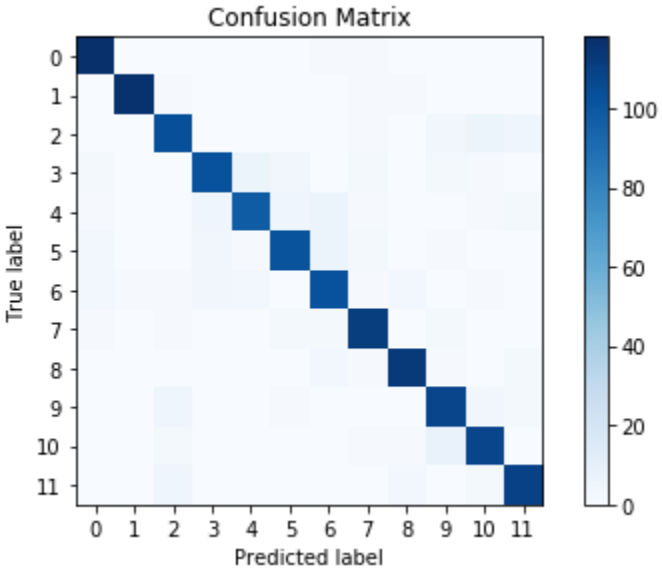
Learning Rate	0.001
Optimizer	Adam
Betas	0.9, 0.999
Batch size	256
No. of epochs	50








```
[[118   0   0   0   0   0   1   1   0   0   0   0]
 [  0 117   1   0   0   0   0   1   1   0   0   0]
 [  0   0 104   0   0   0   0   1   0   4   6   5]
 [  2   0   0 103   6   4   0   2   0   2   1   0]
 [  1   0   0   5  98   5   7   1   0   0   1   2]
 [  3   0   0   4   1 102   7   2   0   1   0   0]
 [  3   1   1   4   3   0 103   1   3   0   1   0]
 [  1   0   1   0   0   2   2 112   0   2   0   0]
 [  0   0   0   0   0   0   3   1 113   1   0   2]
 [  0   0   5   0   0   1   0   0   0 108   4   2]
 [  0   0   2   0   0   0   0   1   1   8 108   0]
 [  0   0   5   0   0   0   0   0   3   0   2 110]]
```



➢ Final Comments:

All the experiments and results have been reported above already, but here I'm giving a few comments based on comparison of different experimental results reported above.

1. **Use pre-trained models of ResNet-50 and VGG16 and report their accuracies. Which one performs better and why?**
   Both achieved accuracy of 99-100% on test data. Resnet50 gave a little lower accuracy in some of the experiments. The reason could be that resnet has learned highly Imagenet specific features in deep layers.

2. **Add batch normalization layers in your network and report your findings in the report.**
   Adding batch normalization to my defined network improved accuracy from 86 to 93%. This is a huge increase. I believe by applying a few more tricks accuracy can be increased further.

3. **Mean Subtract your images and now train your network**.
   Mean image subtraction has reduced accuracy from 86 to 83%. I observed the same in assignment 2 and 3. On the other hand normalization of image (dividing by 255) enhances performance. Also, I believe using per channel means (one value per channel) will enhance performance, but I didn't do that experiment in this assignment as it wasn't asked.

4. **Add some dropout layer in your network and report the comparison of results with and without the dropout layer.**
   I added dropout layers after every convolution layer (as can be seen from the network given in experiment 4 above) and a dropout layer after one of the fc layers. Accuracy dropped. Reason could be the dropout layers after convolution layers as they are learning the features. Adding dropout after convolution will decrease the features.
   On the other hand, dropout layers after fc layers (which are learning to classify the learned features) will enhance performance as a more generic classifier will be learned.

5. **Perform few experiments with different value of learning rate and you may also use the learning rate decay and momentum options**.
   I experimented with 3 values of learning rate i.e. 0.001, 0.003,0.009. 0.001 performed the best among these. I also performed an experiment with 0.1, but model didn't learn anything in 50 epochs even.

6. **Cross-validate for appropriate batch size and report the best batch size.**
   I tried three different batch sizes (using the batch normalization network, given in experiment 3 above). Batch sizes were 8, 64 & 256. Optimizer was Adam in all cases. Batch size 8 achieved an accuracy of 93.5%, 64 of 92% and 256 of 90%. Accuracy decreased by increasing batch size. I think the reason for this the lesser number of images per class.

Another observation that I made was the increase in computation efficiency. Time taken by these batch sizes was in following order:
256 < 64 < 8
The reason could be the efficient use of GPU in case of larger batch size and GPU handling experiments in parallel processes.

7. **Try different optimizers and compare the results.**
   I performed experiments with three optimizers: SGD with momentum, RMSprop & Adam. Adam performed best and achieved an accuracy of 93.47 (BN –layer included). SGD with momentum came second with 92% accuracy and RMSprop achieved surprisingly low accuracy of ~82%.

➢ References:

https://pytorch.org/docs/stable/optim.html

https://scikit-learn.org/stable/auto_examples/model_selection/plot_roc.html

https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html