CSE512 Spring 2021 - Machine Learning - Homework 6

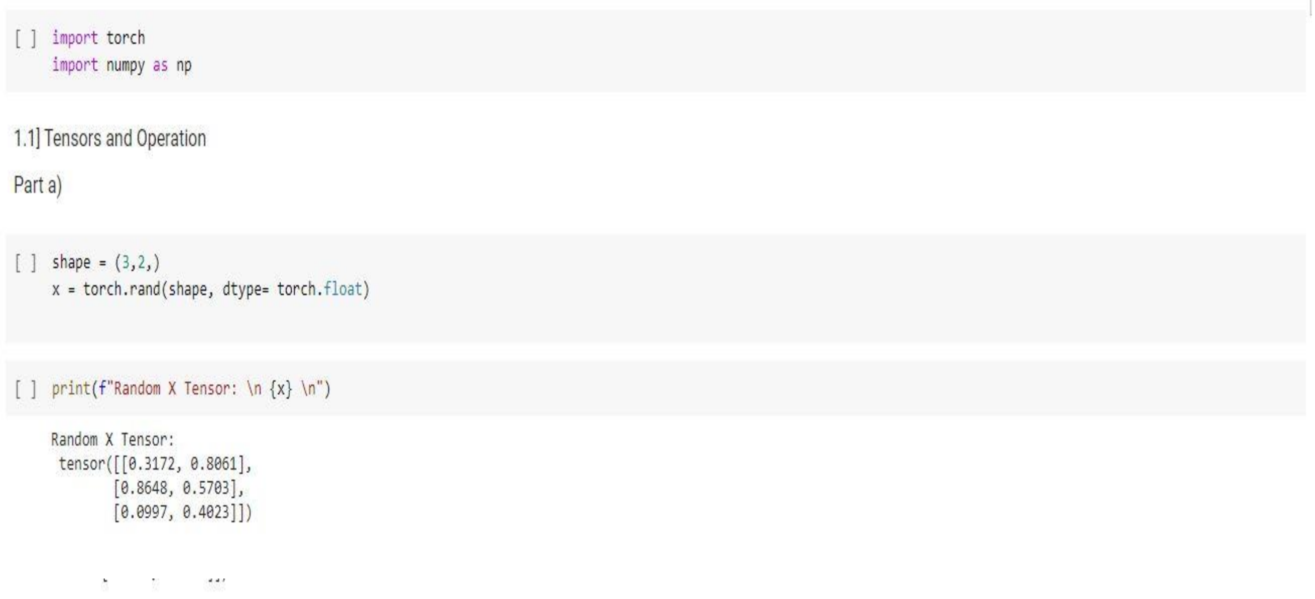
Your Name: Sagar Onkar Toshniwal

Solar ID: 113260061

NetID email address: sagaronkar.toshniwal@stonybrook.edu

Names of people whom you discussed the homework with: None, but I have taken reference from Pytorch, StackOverflow and Google.

1. Tensor Operation



```
[ ]  import torch
     import numpy as np
```

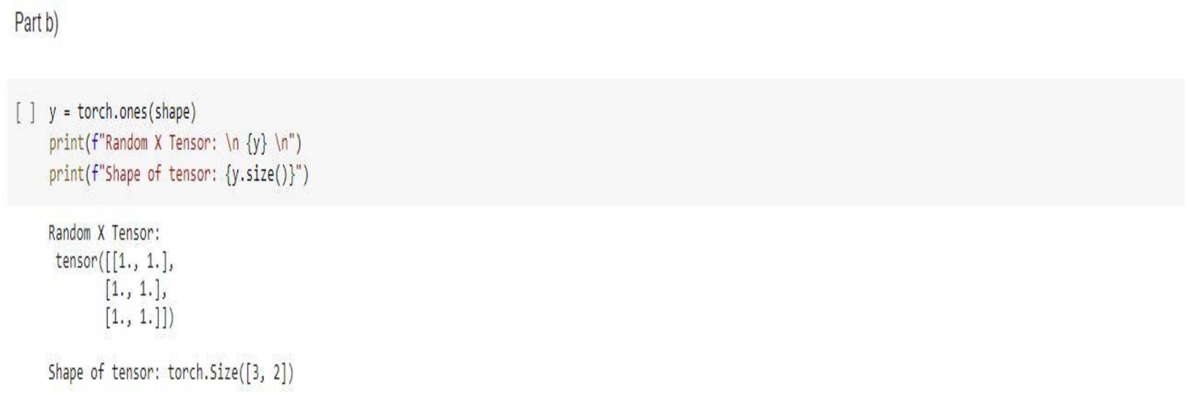
1.1] Tensors and Operation

Part a)

```
[ ]  shape = (3,2,)
     x = torch.rand(shape, dtype= torch.float)
```

```
[ ]  print(f"Random X Tensor: \n {x} \n")
```

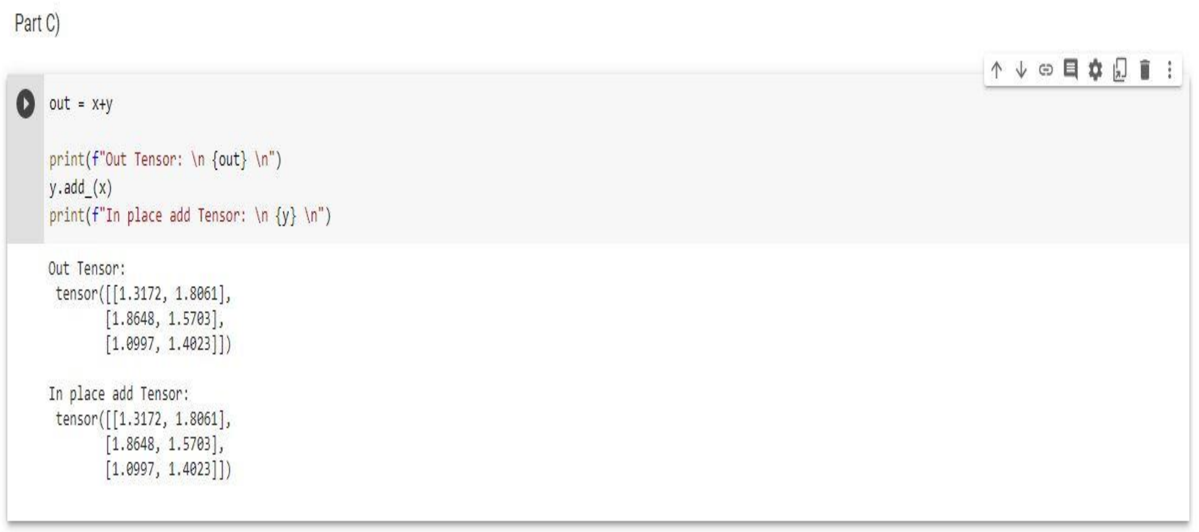
```
     Random X Tensor: 
      tensor([[0.3172, 0.8061],
             [0.8648, 0.5703],
             [0.0997, 0.4023]])
```



Part b)

```
[ ]  y = torch.ones(shape)
     print(f"Random X Tensor: \n {y} \n")
     print(f"Shape of tensor: {y.size()}")
```

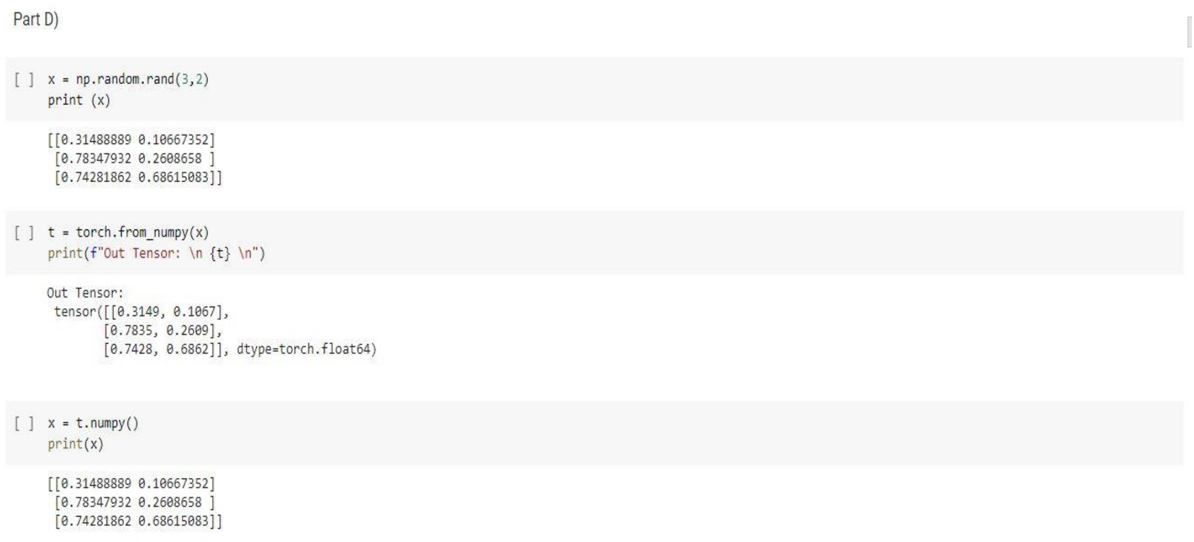
```
     Random X Tensor: 
      tensor([[1., 1.],
             [1., 1.],
             [1., 1.]])

     Shape of tensor: torch.Size([3, 2])
```



Part C)

```
out = x+y

print(f"Out Tensor: \n {out} \n")
y.add_(x)
print(f"In place add Tensor: \n {y} \n")
```

```
Out Tensor:
 tensor([[1.3172, 1.8061],
        [1.8648, 1.5703],
        [1.0997, 1.4023]])

In place add Tensor:
 tensor([[1.3172, 1.8061],
        [1.8648, 1.5703],
        [1.0997, 1.4023]])
```



Part D)

```
[ ]  x = np.random.rand(3,2)
     print (x)
```

```
[[0.31488889 0.10667352]
 [0.78347932 0.2608658 ]
 [0.74281862 0.68615083]]
```

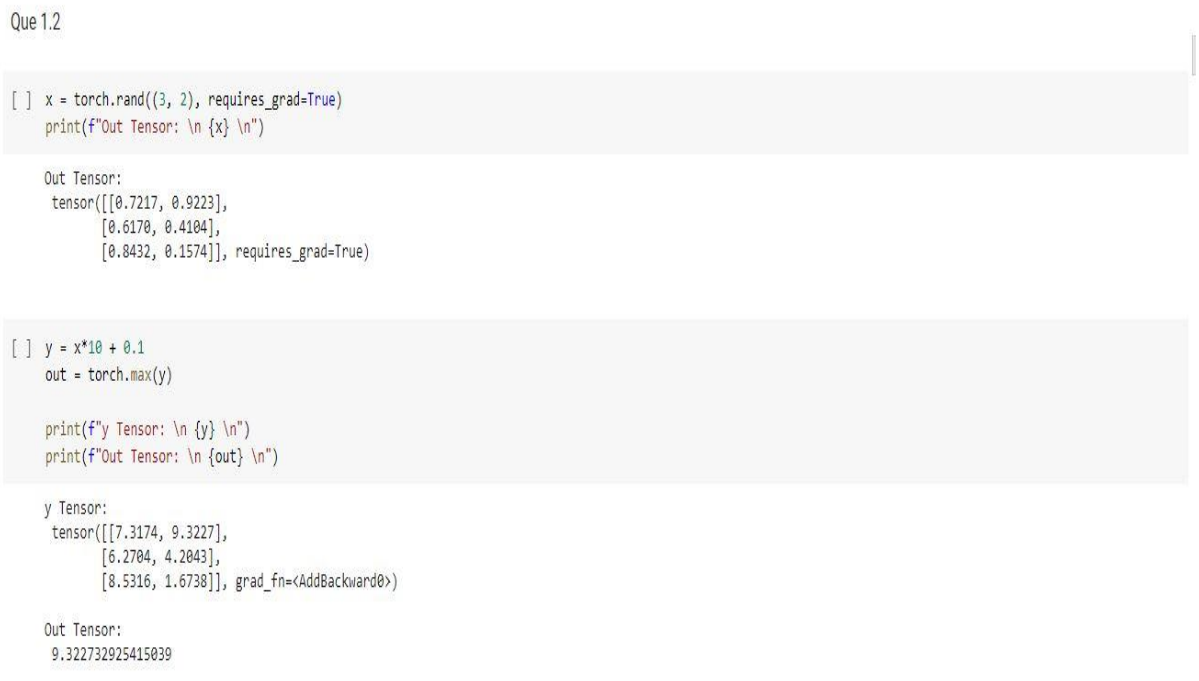
```
[ ]  t = torch.from_numpy(x)
     print(f"Out Tensor: \n {t} \n")
```

```
Out Tensor:
 tensor([[0.3149, 0.1067],
        [0.7835, 0.2609],
        [0.7428, 0.6862]], dtype=torch.float64)
```

```
[ ]  x = t.numpy()
     print(x)
```

```
[[0.31488889 0.10667352]
 [0.78347932 0.2608658 ]
 [0.74281862 0.68615083]]
```

1.2] Autograd

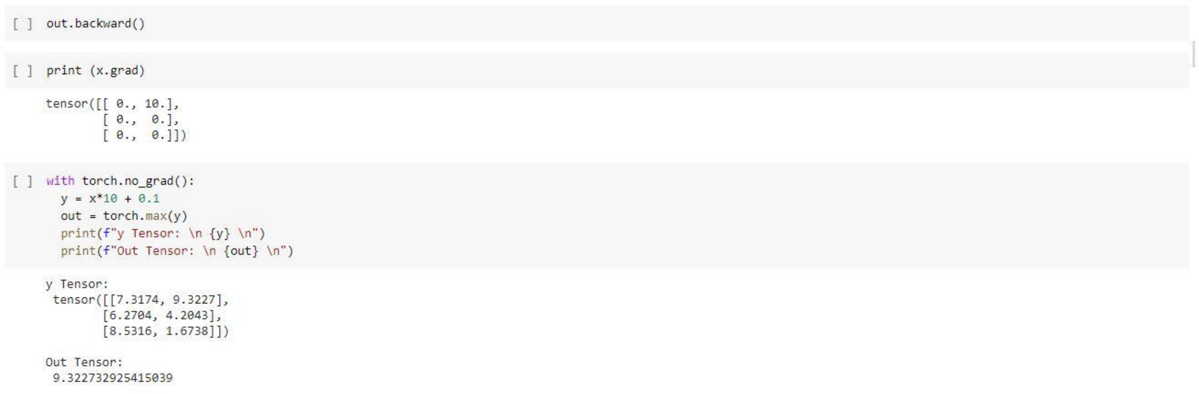


Que 1.2

```
[ ] x = torch.rand((3, 2), requires_grad=True)
    print(f"Out Tensor: \n {x} \n")

    Out Tensor:
     tensor([[0.7217, 0.9223],
             [0.6170, 0.4104],
             [0.8432, 0.1574]], requires_grad=True)
```

```
[ ] y = x*10 + 0.1
    out = torch.max(y)

    print(f"y Tensor: \n {y} \n")
    print(f"Out Tensor: \n {out} \n")

    y Tensor:
     tensor([[7.3174, 9.3227],
             [6.2704, 4.2043],
             [8.5316, 1.6738]], grad_fn=<AddBackward0>)

    Out Tensor:
     9.322732925415039
```

Grad_fn attributes stores the gradient while backpropagation. It stores the gradient for only those variables or intermediate variable whose requires_grad attritube is set to "True"



```
[ ] out.backward()
```

```
[ ] print (x.grad)

    tensor([[ 0., 10.],
            [ 0.,  0.],
            [ 0.,  0.]])
```

```
[ ] with torch.no_grad():
      y = x*10 + 0.1
      out = torch.max(y)
      print(f"y Tensor: \n {y} \n")
      print(f"Out Tensor: \n {out} \n")

    y Tensor:
     tensor([[7.3174, 9.3227],
             [6.2704, 4.2043],
             [8.5316, 1.6738]])

    Out Tensor:
     9.322732925415039
```

On printing gradient for x, since we performed maximum operation only the max is stored in out and rest all 0 and while calculating gradient d_out/d_x, we get the above output.

On using no_grad() attribute, the gradient are not stored and hence no memory is allocated. Thus we didn't get any gradient.

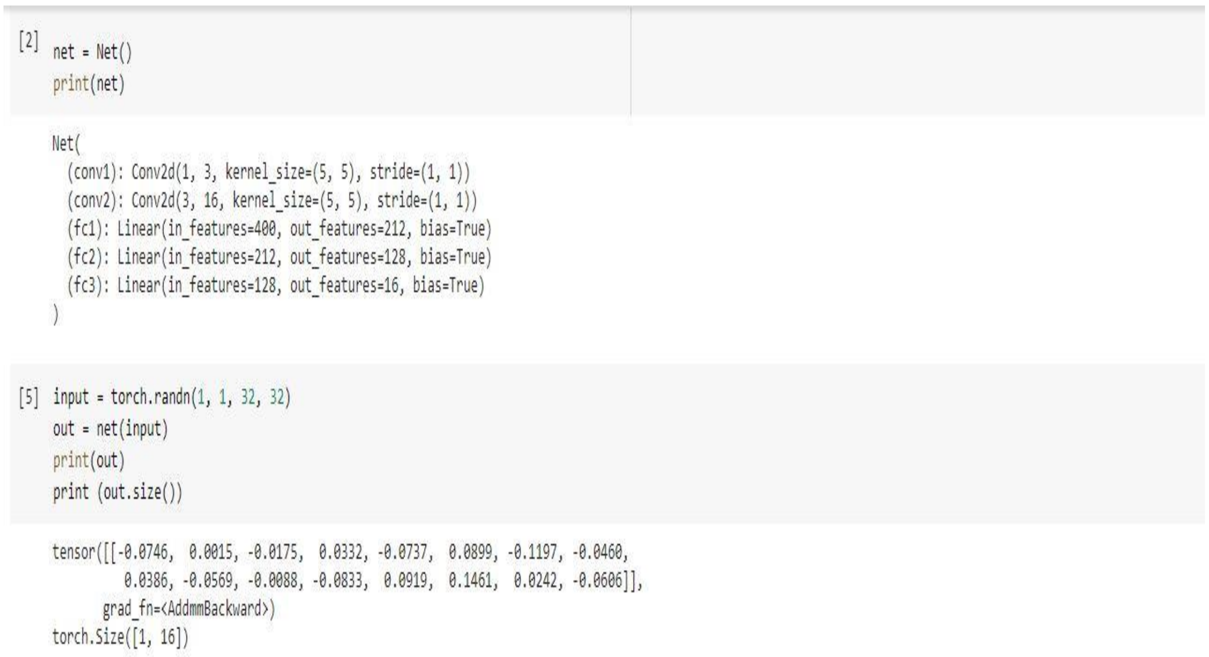
1.3] Neural Network:

I have tried multiple neural network architecture to get min loss and after lot of permuatation and combination I made one architecture but it was slightly similar to the given. But considering the loss I chose by changing other parameters.

1.3.a) parameters :- 2 convolution layer and 3 linear layers

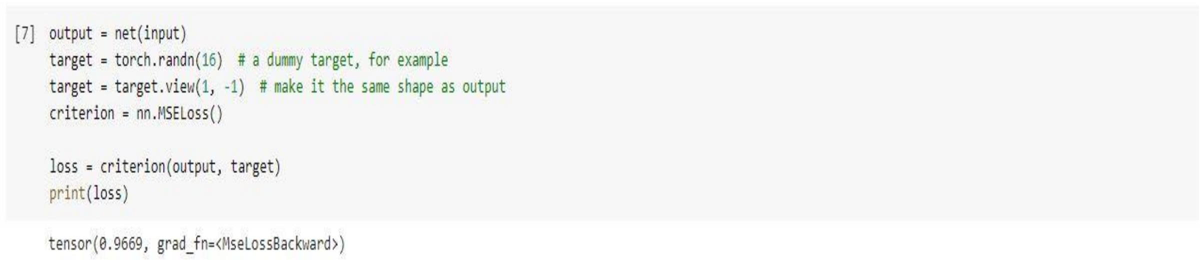
Total 5 parameters.

1.3.b) Output is shown in figure below



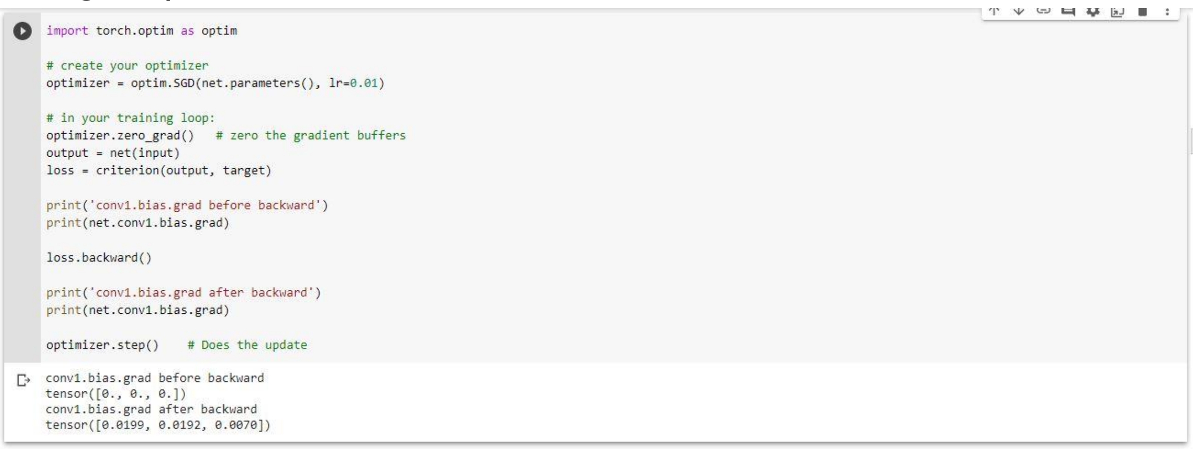
```
[2] net = Net()
    print(net)

    Net(
      (conv1): Conv2d(1, 3, kernel_size=(5, 5), stride=(1, 1))
      (conv2): Conv2d(3, 16, kernel_size=(5, 5), stride=(1, 1))
      (fc1): Linear(in_features=400, out_features=212, bias=True)
      (fc2): Linear(in_features=212, out_features=128, bias=True)
      (fc3): Linear(in_features=128, out_features=16, bias=True)
    )

[5] input = torch.randn(1, 1, 32, 32)
    out = net(input)
    print(out)
    print (out.size())

    tensor([[-0.0746,  0.0015, -0.0175,  0.0332, -0.0737,  0.0899, -0.1197, -0.0460,
              0.0386, -0.0569, -0.0088, -0.0833,  0.0919,  0.1461,  0.0242, -0.0606]],
           grad_fn=<AddmmBackward>)
    torch.Size([1, 16])
```

1.3.c) MSE loss – 0.9669



```
[7] output = net(input)
    target = torch.randn(16)  # a dummy target, for example
    target = target.view(1, -1)  # make it the same shape as output
    criterion = nn.MSELoss()

    loss = criterion(output, target)
    print(loss)

    tensor(0.9669, grad_fn=<MseLossBackward>)
```

Bias grad printed below



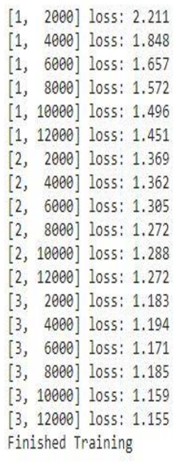
```
import torch.optim as optim

# create your optimizer
optimizer = optim.SGD(net.parameters(), lr=0.01)

# in your training loop:
optimizer.zero_grad()   # zero the gradient buffers
output = net(input)
loss = criterion(output, target)

print('conv1.bias.grad before backward')
print(net.conv1.bias.grad)

loss.backward()

print('conv1.bias.grad after backward')
print(net.conv1.bias.grad)

optimizer.step()    # Does the update
```

```
conv1.bias.grad before backward
tensor([0., 0., 0.])
conv1.bias.grad after backward
tensor([0.0199, 0.0192, 0.0070])
```

Que 1.4]

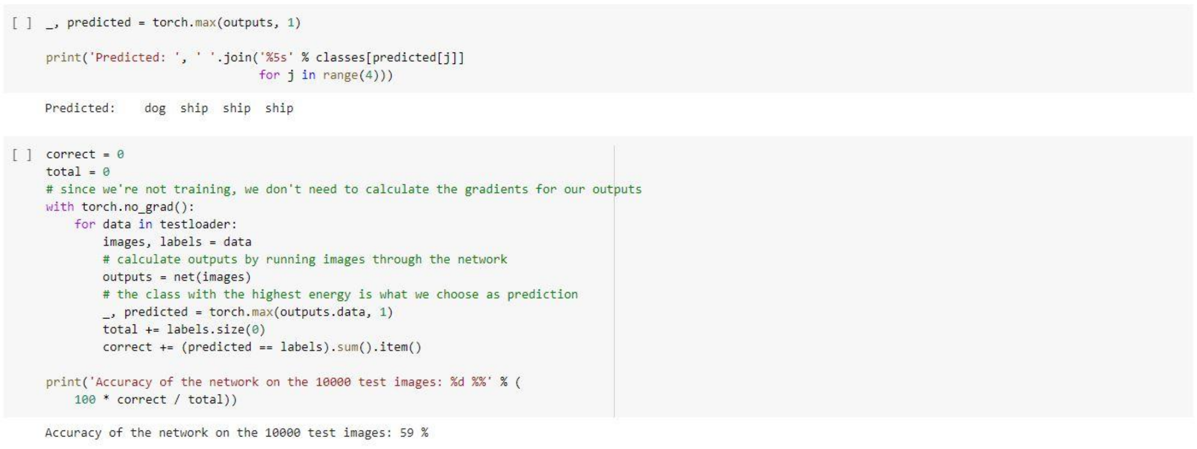
1.4] Training

Loss for 3 epoch



```
[1,  2000] loss: 2.211
[1,  4000] loss: 1.848
[1,  6000] loss: 1.657
[1,  8000] loss: 1.572
[1, 10000] loss: 1.496
[1, 12000] loss: 1.451
[2,  2000] loss: 1.369
[2,  4000] loss: 1.362
[2,  6000] loss: 1.305
[2,  8000] loss: 1.272
[2, 10000] loss: 1.288
[2, 12000] loss: 1.272
[3,  2000] loss: 1.183
[3,  4000] loss: 1.194
[3,  6000] loss: 1.171
[3,  8000] loss: 1.185
[3, 10000] loss: 1.159
[3, 12000] loss: 1.155
Finished Training
```

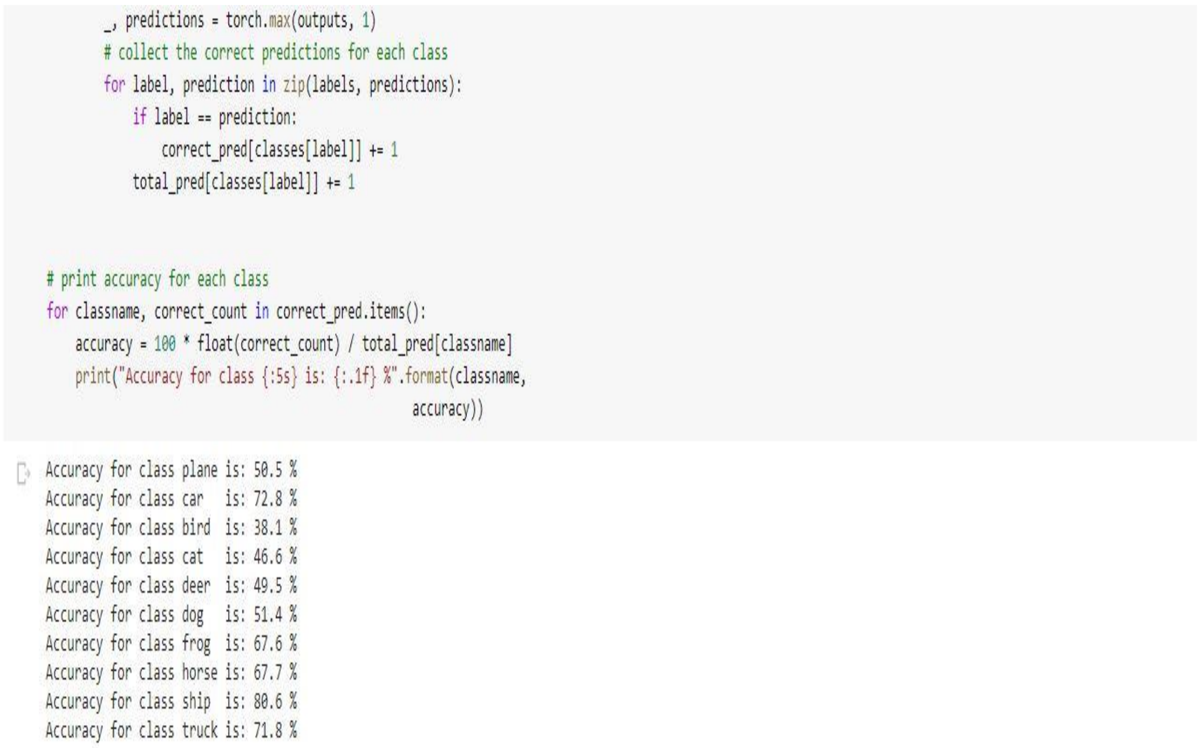
Accuracy on test Images is 59%



```
[ ]  _, predicted = torch.max(outputs, 1)

     print('Predicted: ', ' '.join('%5s' % classes[predicted[j]]
                                   for j in range(4)))

     Predicted:    dog  ship  ship  ship
```

```
[ ]  correct = 0
     total = 0
     # since we're not training, we don't need to calculate the gradients for our outputs
     with torch.no_grad():
         for data in testloader:
             images, labels = data
             # calculate outputs by running images through the network
             outputs = net(images)
             # the class with the highest energy is what we choose as prediction
             _, predicted = torch.max(outputs.data, 1)
             total += labels.size(0)
             correct += (predicted == labels).sum().item()

     print('Accuracy of the network on the 10000 test images: %d %%' % (
         100 * correct / total))

     Accuracy of the network on the 10000 test images: 59 %
```

Accuracy for each class given below

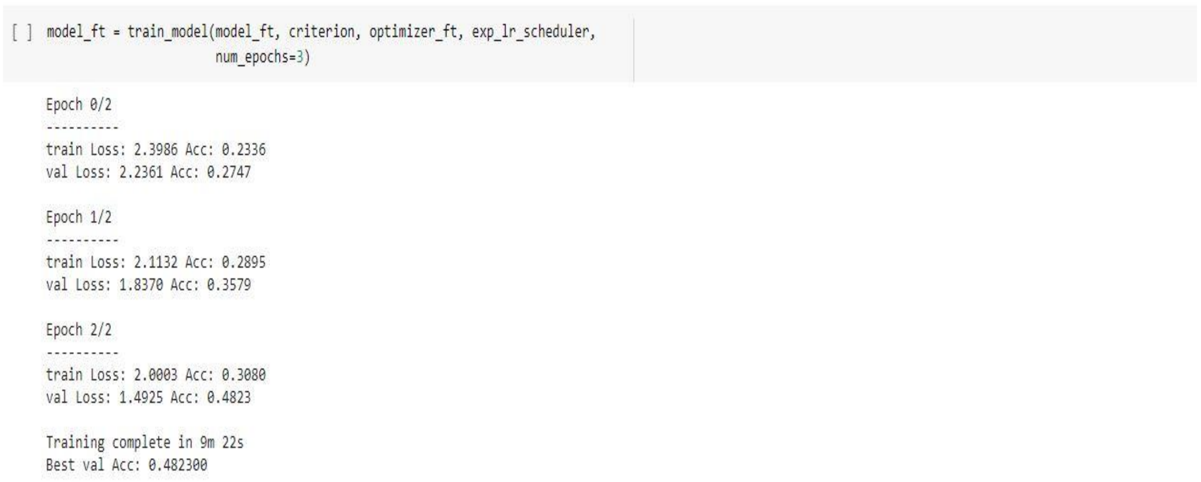


```
        _, predictions = torch.max(outputs, 1)
        # collect the correct predictions for each class
        for label, prediction in zip(labels, predictions):
            if label == prediction:
                correct_pred[classes[label]] += 1
            total_pred[classes[label]] += 1



# print accuracy for each class
for classname, correct_count in correct_pred.items():
    accuracy = 100 * float(correct_count) / total_pred[classname]
    print("Accuracy for class {:5s} is: {:.1f} %".format(classname,
                                                   accuracy))
```

```
Accuracy for class plane is: 50.5 %
Accuracy for class car   is: 72.8 %
Accuracy for class bird  is: 38.1 %
Accuracy for class cat   is: 46.6 %
Accuracy for class deer  is: 49.5 %
Accuracy for class dog   is: 51.4 %
Accuracy for class frog  is: 67.6 %
Accuracy for class horse is: 67.7 %
Accuracy for class ship  is: 80.6 %
Accuracy for class truck is: 71.8 %
```

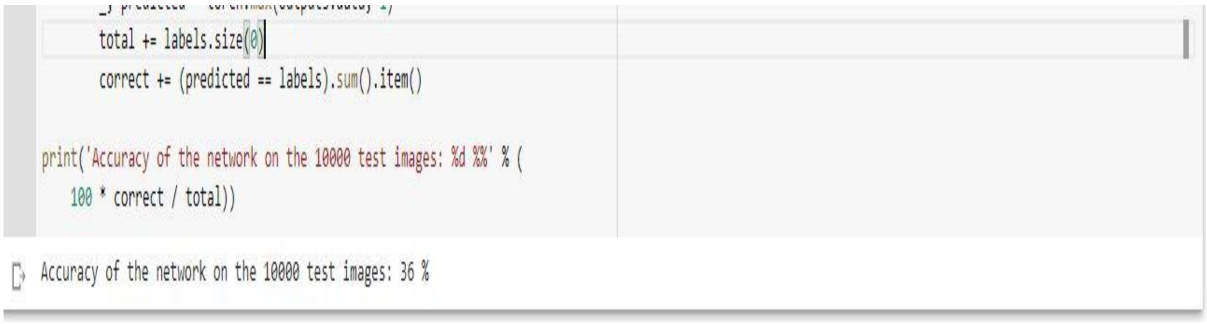
1.5] Transfer Learning

1.5.a] Training Loss for 3 epoch



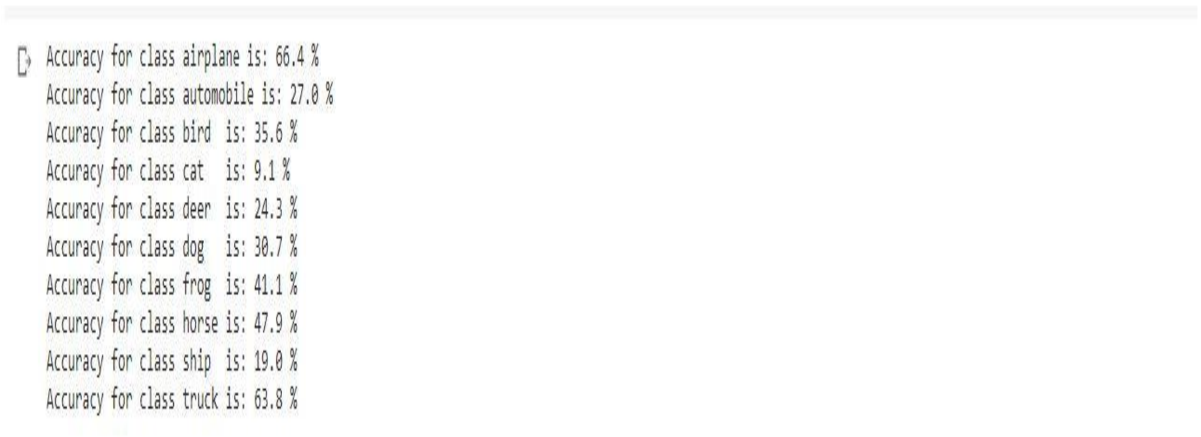
```
[ ] model_ft = train_model(model_ft, criterion, optimizer_ft, exp_lr_scheduler,
                       num_epochs=3)

Epoch 0/2
----------
train Loss: 2.3986 Acc: 0.2336
val Loss: 2.2361 Acc: 0.2747

Epoch 1/2
----------
train Loss: 2.1132 Acc: 0.2895
val Loss: 1.8370 Acc: 0.3579

Epoch 2/2
----------
train Loss: 2.0003 Acc: 0.3080
val Loss: 1.4925 Acc: 0.4823

Training complete in 9m 22s
Best val Acc: 0.482300
```

Accuracy on test Images – 36%



```
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %d %%' % (
    100 * correct / total))

Accuracy of the network on the 10000 test images: 36 %
```

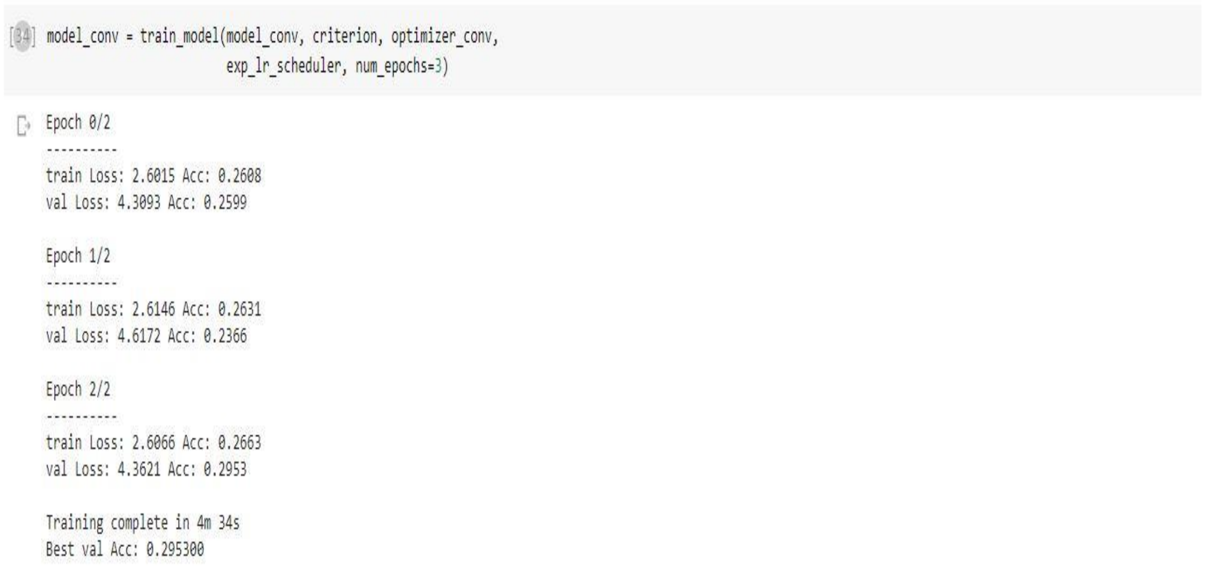
Accuracy for each class is given Below



```
Accuracy for class airplane is: 66.4 %
Accuracy for class automobile is: 27.0 %
Accuracy for class bird  is: 35.6 %
Accuracy for class cat   is: 9.1 %
Accuracy for class deer  is: 24.3 %
Accuracy for class dog   is: 30.7 %
Accuracy for class frog  is: 41.1 %
Accuracy for class horse is: 47.9 %
Accuracy for class ship  is: 19.0 %
Accuracy for class truck is: 63.8 %
```

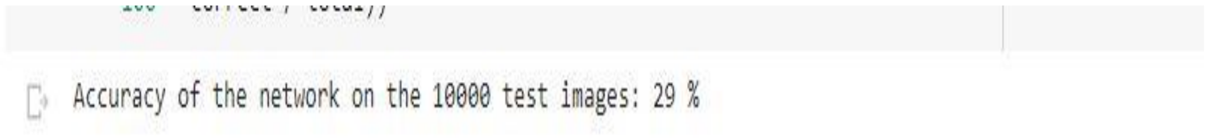
1.5.b] Transfer Learning Pre-Trained model

Loss for 3 epoch



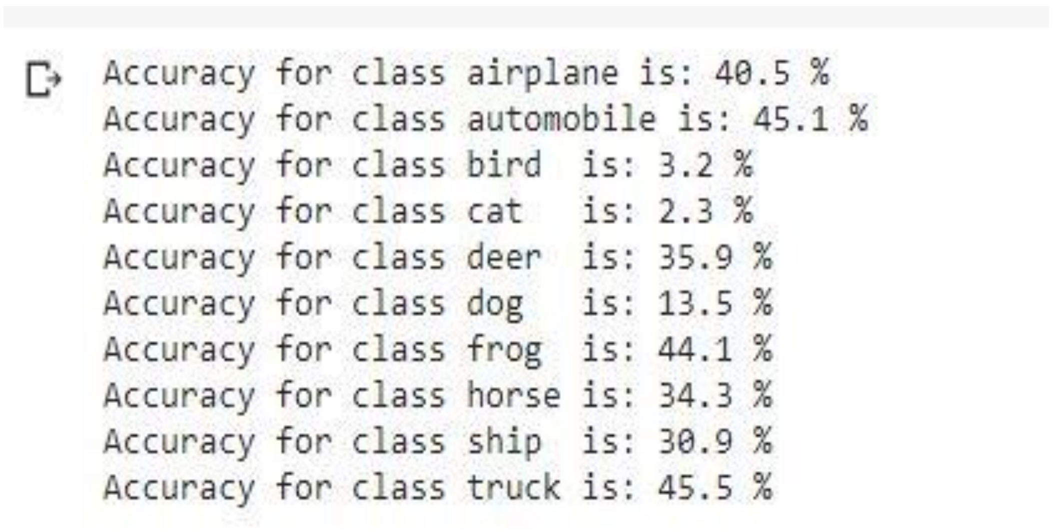
```
[34] model_conv = train_model(model_conv, criterion, optimizer_conv,
                          exp_lr_scheduler, num_epochs=3)

Epoch 0/2
----------
train Loss: 2.6015 Acc: 0.2608
val Loss: 4.3093 Acc: 0.2599

Epoch 1/2
----------
train Loss: 2.6146 Acc: 0.2631
val Loss: 4.6172 Acc: 0.2366

Epoch 2/2
----------
train Loss: 2.6066 Acc: 0.2663
val Loss: 4.3621 Acc: 0.2953

Training complete in 4m 34s
Best val Acc: 0.295300
```

Accuracy for Test Images – 29%



```
        100 * correct / total))

Accuracy of the network on the 10000 test images: 29 %
```

Accuracy for each class is given below



```
Accuracy for class airplane is: 40.5 %
Accuracy for class automobile is: 45.1 %
Accuracy for class bird  is: 3.2 %
Accuracy for class cat   is: 2.3 %
Accuracy for class deer  is: 35.9 %
Accuracy for class dog   is: 13.5 %
Accuracy for class frog  is: 44.1 %
Accuracy for class horse is: 34.3 %
Accuracy for class ship  is: 30.9 %
Accuracy for class truck is: 45.5 %
```

2.1] Short Summary

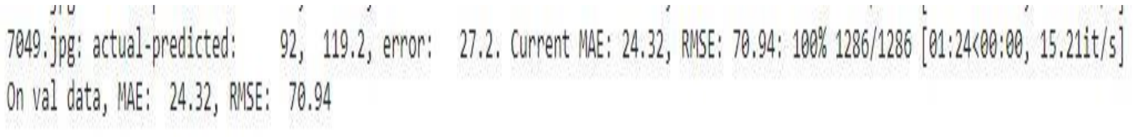
This paper focuses on counting objects from multiple categories from a given only a few annotated instances of that category. Existing work is this domain is focused on specific or few categories at one time. From the Existing work, it overcomes two important challenges – 1) it avoids Supervised regression task by implementing few shout counting model 2) The Dataset collection for multiple categories and its annotations.

Few Shot Adaptation and Matching Network (FamNet) is the proposed architecture for tackling the few-shot counting task. It consists of a 1) a feature extraction module, and 2) a density prediction module. The feature extraction module consists of a general feature extractor capable of handling a large number of visual categories. The density prediction module is designed to be agnostic to the visual category.

It implements few shot counting task which deals with counting instances within an image which are similar to the exemplars from the query image and then and predicts a density map for the presence of all objects of interest in the query image. The classes at test time are completely different from the ones during training. At test Time this adaptation scheme uses the provided exemplars and then adapts the counting network to them with a few gradient descent updates, where the gradients are computed based on two loss functions which utilize the locations of the exemplars to the fullest extent thus providing advantage over the existing work.

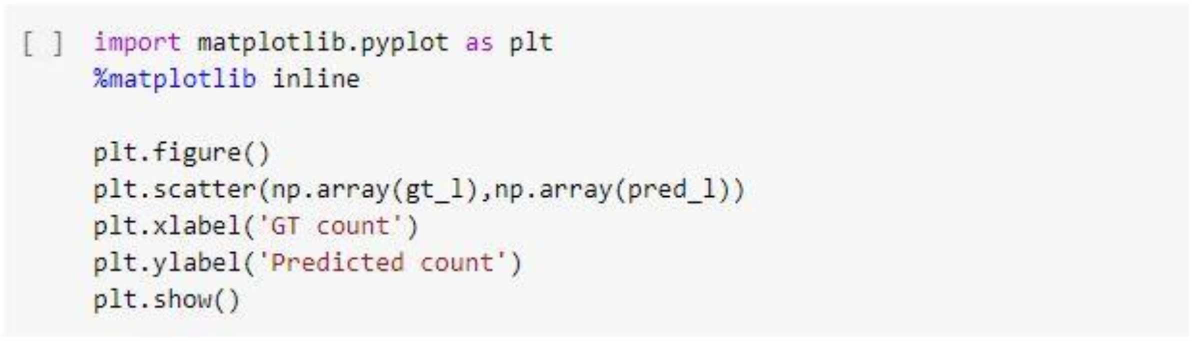
2.2] Result Analysis

```
- On val data, MAE:  24.32, RMSE:  70.94
```

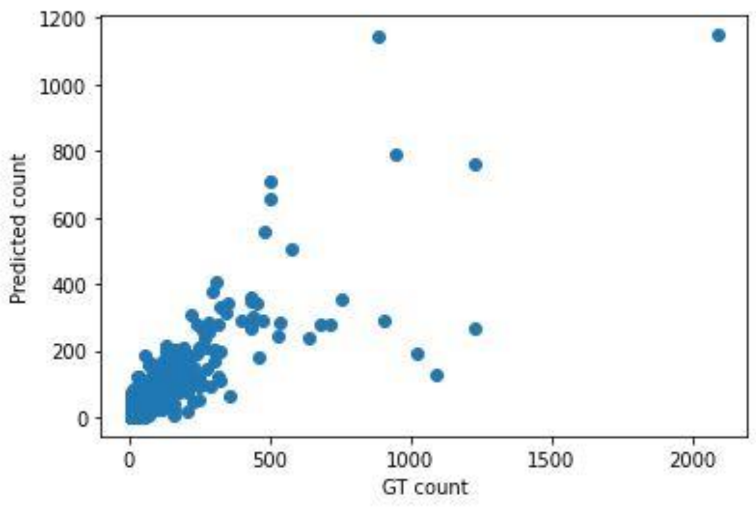


```
7049.jpg: actual-predicted:    92,  119.2, error:   27.2. Current MAE: 24.32, RMSE: 70.94: 100% 1286/1286 [01:24<00:00, 15.21it/s]
On val data, MAE:  24.32, RMSE:  70.94
```

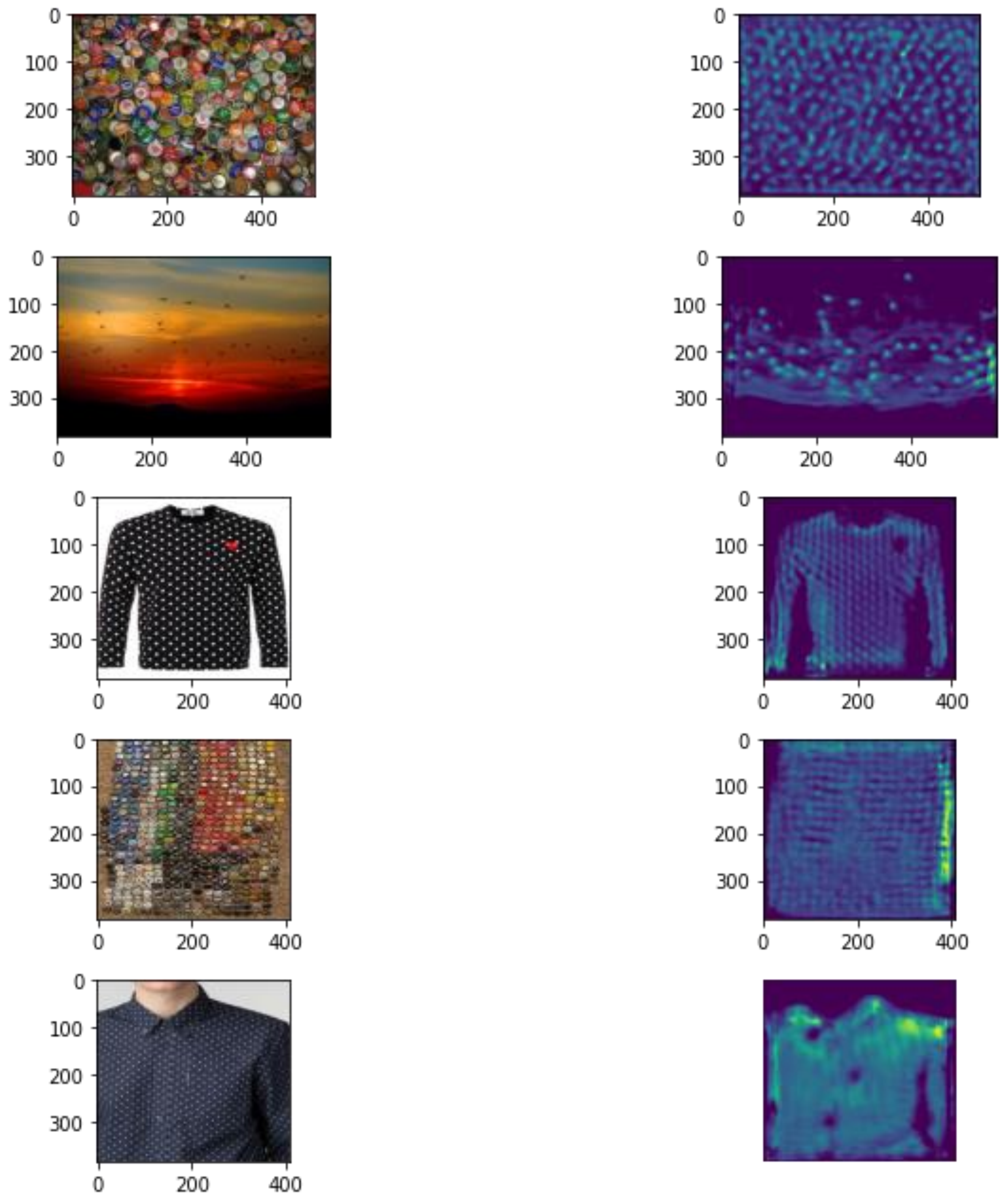
- Scatter Plot X-axis contains the ground truth count, and the Y-axis contains the predicted count.



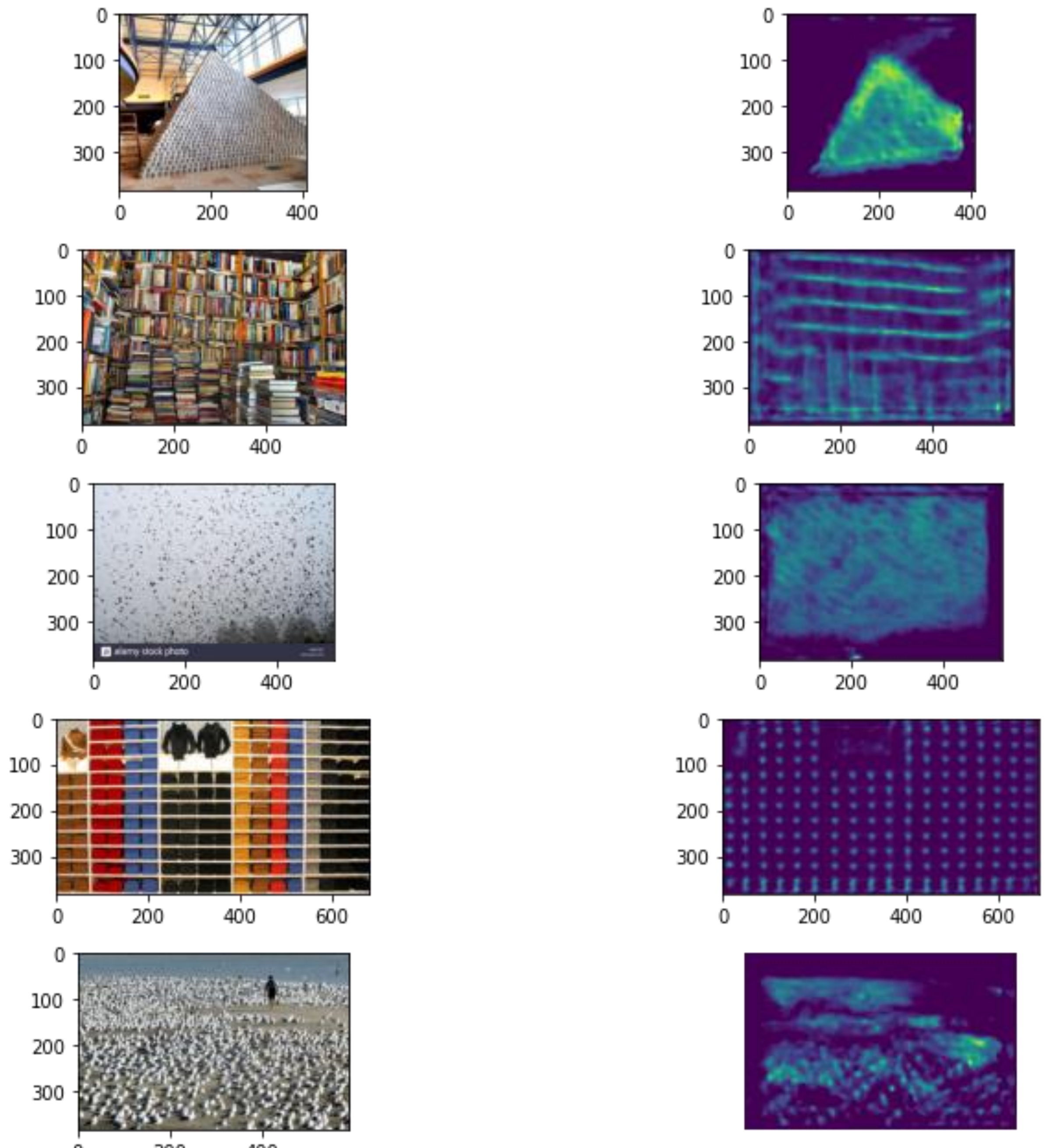
```
[ ] import matplotlib.pyplot as plt
    %matplotlib inline

    plt.figure()
    plt.scatter(np.array(gt_l),np.array(pred_l))
    plt.xlabel('GT count')
    plt.ylabel('Predicted count')
    plt.show()
```



5 images and corresponding predicted density maps (output of FamNet) with the highest over-count error (predicted-count - gt-count)



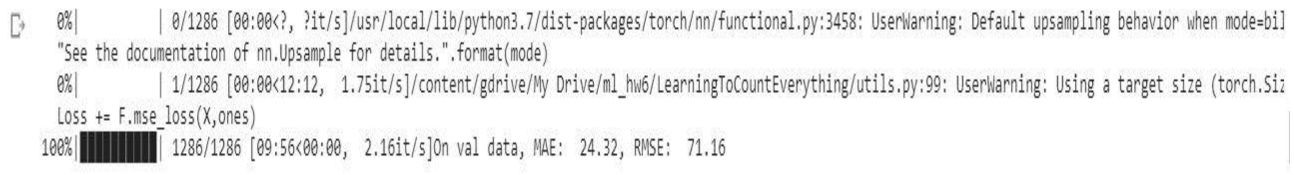
5 images and corresponding predicted density maps (output of FamNet) with the highest under-count error (gt-count - predicted-count)



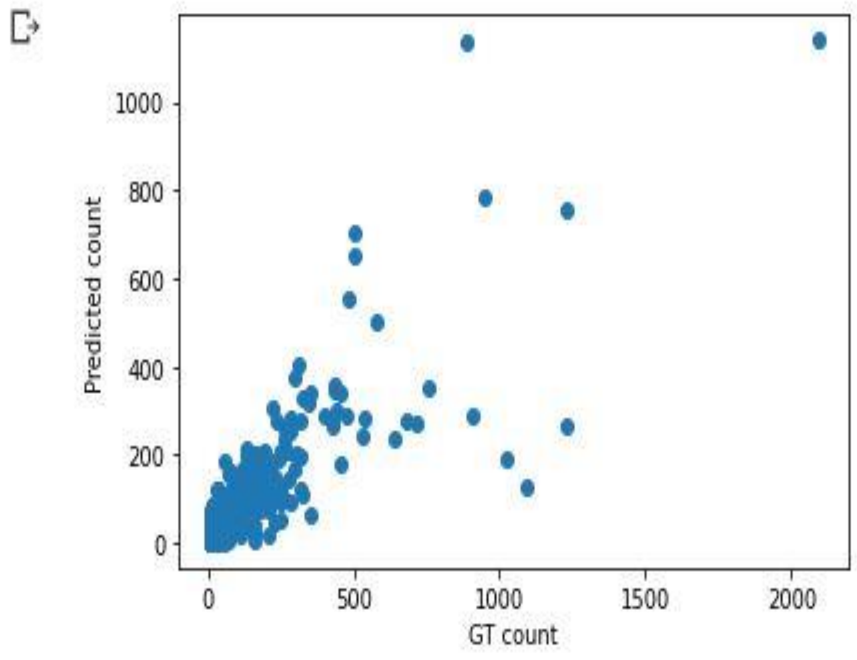
2.3] Adaptation :

1) With Adaptation :

```
MAE:  24.32, RMSE:  71.16
```

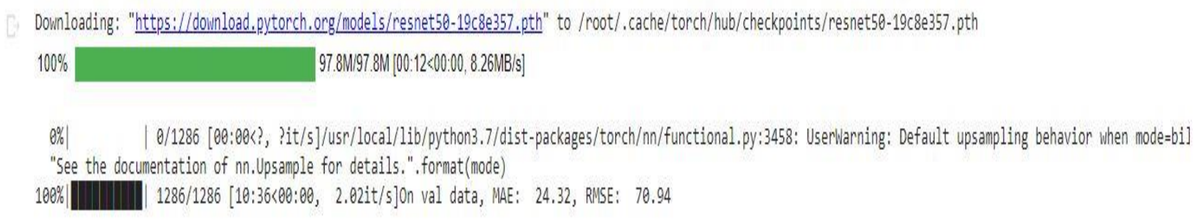


```
  0%|          | 0/1286 [00:00<?, ?it/s]/usr/local/lib/python3.7/dist-packages/torch/nn/functional.py:3458: UserWarning: Default upsampling behavior when mode=bil
  "See the documentation of nn.Upsample for details.".format(mode)
  0%|          | 1/1286 [00:00<12:12,  1.75it/s]/content/gdrive/My Drive/ml_hw6/LearningToCountEverything/utils.py:99: UserWarning: Using a target size (torch.Siz
  Loss += F.mse_loss(X,ones)
100%|██████████| 1286/1286 [09:56<00:00,  2.16it/s]On val data, MAE:  24.32, RMSE:  71.16
```

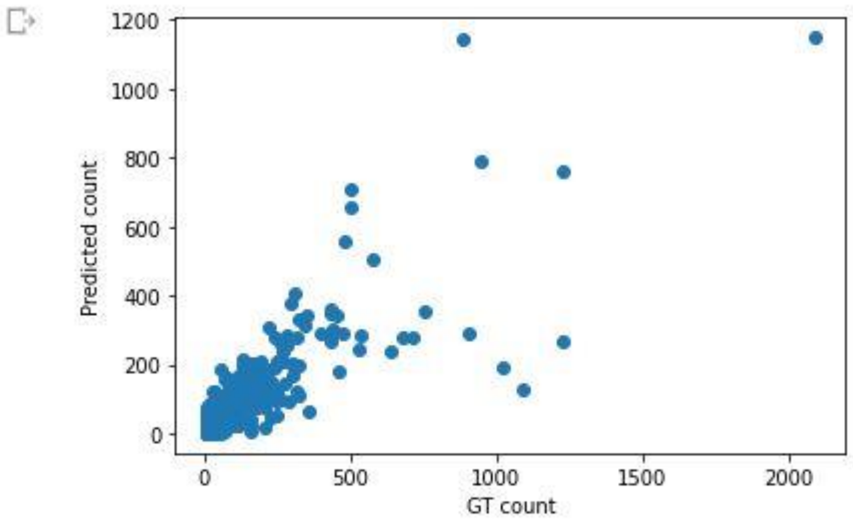


2] Without Adaptation :-

```
MAE:  24.32, RMSE:  70.94
```



```
Downloading: "https://download.pytorch.org/models/resnet50-19c8e357.pth" to /root/.cache/torch/hub/checkpoints/resnet50-19c8e357.pth
100%                                  97.8M/97.8M [00:12<00:00, 8.26MB/s]

  0%|          | 0/1286 [00:00<?, ?it/s]/usr/local/lib/python3.7/dist-packages/torch/nn/functional.py:3458: UserWarning: Default upsampling behavior when mode=bil
  "See the documentation of nn.Upsample for details.".format(mode)
100%|██████████| 1286/1286 [10:36<00:00,  2.02it/s]On val data, MAE:  24.32, RMSE:  70.94
```



2.4] Already submitted CSV file at Kaggle