Cover page for answers.pdf

CSE512 Spring 2021 - Machine Learning - Homework 6

Your Name: Venkata Ravi Teja Takkella

Solar ID: 113219890

NetID email address: venkataravite.takkella@stonybrook.edu

Names of people whom you discussed the homework with: NA

Setup:

Place the **test.py** and **utils.py** files under LearningToCountEverything. The parameters tuned are either mentioned as default values(weights) or given as part of the arguments(Grad steps). You can check the same in the Notebooks attached along with the submission

Q1.1)

a) **Randomly initiated tensor x => :**

tensor([[0.9334, 0.1717],
[0.7645, 0.7880],
[0.8824, 0.7022]])

b) **Tensor y => :**

tensor([[1., 1.],
[1., 1.],
[1., 1.]])

Tensor y size => :

torch.Size([3, 2])

c) **1) Out =>**
tensor([[1.9334, 1.1717],
[1.7645, 1.7880],
[1.8824, 1.7022]])

2) y=>
tensor([[1.9334, 1.1717],
[1.7645, 1.7880],
[1.8824, 1.7022]])

d) **Tensor** :
tensor([[0.9910, 0.2627],
[0.2259, 0.5102],
[0.4148, 0.0354]], dtype=torch.float64)

Numpy Array :
[[0.99104231 0.26271163]
[0.2259091 0.51024524]
[0.41481212 0.03542314]]

Q1.2)

a) **x => :**

tensor([[0.0720, 0.9404],
[0.2588, 0.5108],
[0.7300, 0.6649]], requires_grad=True)

b)

Multiplied by 10: tensor([[0.7200, 9.4043],
[2.5880, 5.1084],
[7.3003, 6.6487]], grad_fn=<MulBackward0>)

The grad_fn shows MulBackward0 since Multiplication (mathematical operator) is used for creating the variable.

Addition by 0.1: tensor([[0.8200, 9.5043],
[2.6880, 5.2084],
[7.4003, 6.7487]], grad_fn=<AddBackward0>)

The grad_fn shows AddBackward0 since Addition (mathematical operator) is used for creating the variable.

Out: tensor(9.5043, grad_fn=<MaxBackward1>)

The grad_fn shows MaxBackward1 since we are taking max for creating the variable.

c) **Gradient =>** :

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

d) **The message printed is as below :**

Multiplied by 10: tensor([[0.7200, 9.4043],
[2.5880, 5.1084],
[7.3003, 6.6487]])

Addition by 0.1: tensor([[0.8200, 9.5043],
[2.6880, 5.2084],
[7.4003, 6.7487]])

Out: tensor(9.5043)

While doing b, we have provided the gradients flag as True. Hence the grad_fn are stored so that we can use them while calculating the gradients. Whereas here we have used torch.no_grad() and so, there's no need of calculating the gradients and hence there are none seen in the output too.

Q1.3)

a) **Network :**

```
Network : Net(
  (conv1): Conv2d(3, 8, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(8, 24, kernel_size=(3, 3), stride=(1, 1))
  (dropout1): Dropout2d(p=0.25, inplace=False)
  (dropout2): Dropout2d(p=0.5, inplace=False)
  (fc1): Linear(in_features=864, out_features=240, bias=True)
  (fc2): Linear(in_features=240, out_features=120, bias=True)
  (fc3): Linear(in_features=120, out_features=84, bias=True)
  (fc4): Linear(in_features=84, out_features=10, bias=True)
)
```

Parameters:

torch.Size([8, 3, 5, 5])

b)

Output:

```
tensor([[ 0.1027,  0.0314, -0.0525, -0.0677,  0.0400, -0.0721,  0.0813,  0.0695,
        -0.0231,  0.0849]], grad_fn=<AddmmBackward>)
```

Size :

(1, 10)

c) **The bias gradients of the 2nd Convolutional layer are:**

```
tensor([ 0.0023, -0.0004,  0.0027, -0.0012,  0.0063, -0.0070,  0.0008, -0.0017,
        0.0032,  0.0008,  0.0032,  0.0000,  0.0015, -0.0003, -0.0013, -0.0002,
        0.0030,  0.0011, -0.0017, -0.0033,  0.0016,  0.0041, -0.0016,  0.0059])
```

Q1.4)

Accuracy on the entire test set **: 62 %**

Accuracy per class :

Accuracy for class **plane** is: 68.3 %
Accuracy for class **car** is: 68.1 %
Accuracy for class **bird** is: 54.5 %
Accuracy for class **cat** is: 33.8 %
Accuracy for class **deer** is: 50.3 %
Accuracy for class **dog** is: 53.2 %
Accuracy for class **frog** is: 68.3 %
Accuracy for class **horse** is: 74.4 %
Accuracy for class **ship** is: 81.9 %
Accuracy for class **truck** is: 74.6 %

Q1.5)

a)

Accuracy on the entire test set **: 32 %**

Accuracy per class :

Accuracy for class **plane** is: 31.4 %
Accuracy for class **car** is: 56.4 %
Accuracy for class **bird** is: 0.0 %
Accuracy for class **cat** is: 1.5 %
Accuracy for class **deer** is: 66.8 %
Accuracy for class **dog** is: 28.0 %
Accuracy for class **frog** is: 28.4 %
Accuracy for class **horse** is: 41.3 %
Accuracy for class **ship** is: 54.4 %
Accuracy for class **truck** is: 17.8 %

b)

Accuracy on the entire test set **: 24 %**

Accuracy per class :

Accuracy for class **plane** is: 47.7 %
Accuracy for class **car** is: 24.5 %
Accuracy for class **bird** is: 50.2 %
Accuracy for class **cat** is: 20.1 %
Accuracy for class **deer** is: 7.9 %
Accuracy for class **dog** is: 10.6 %
Accuracy for class **frog** is: 27.4 %
Accuracy for class **horse** is: 38.8 %
Accuracy for class **ship** is: 16.9 %
Accuracy for class **truck** is: 19.4 %

Q2.1)

Based on the paper, there are two major challenges in developing systems capable of counting large numbers of visual categories. First, most of the approaches treat counting as a supervised regression task, requiring thousands of labeled images to learn a fully convolutional regressor. Second, there are not any large enough unconstrained counting datasets with many visual categories for the development of a general counting method. This paper tries to solve these two challenges by a few-shot counting method which generalizes to completely novel classes using only the input image and a few exemplars i.e **FamNet**. Famnet consists of two key modules 1) **a multi-scale feature extraction module**, and 2) **a density prediction module**. The multi-scale feature extraction module consists of the first four blocks from a pre-trained ResNet-50 backbone and the density prediction module is designed to be agnostic to the visual categories by generating a **correlation map** between the exemplar features and image features as the input to the density prediction module.

We minimized the MSE loss between the predicted density map and the ground truth density map during Training. On top of that, we also included the test-time adaption which can improve the accuracy of the estimated count. The key idea is to harness the information provided by the locations of the exemplar bounding boxes. So the loss used is the weighted sum of **Min-Count Loss** and **Perturbation loss**. Zb denotes the crop from the density map Z for the exemplar bounding boxes.

$$\mathcal{L}_{MinCount} = \sum_{b \in B} \max(0, 1 - ||Z_b||_1). \qquad \mathcal{L}_{Per} = \sum_{b \in B} ||Z_b - G_{h \times w}||_2^2.$$

$$\mathcal{L}_{Adapt} = \lambda_1 \mathcal{L}_{MinCount} + \lambda_2 \mathcal{L}_{Per}$$

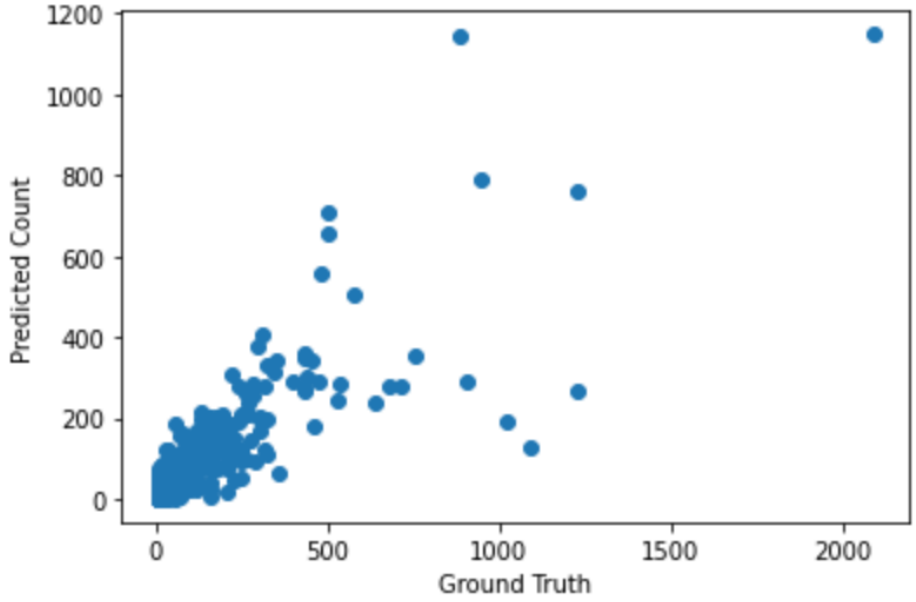
To train the FamNet, 6135 images across a diverse set of 147 object categories, from kitchen utensils and office stationery to vehicles and animals have been collected with three desired object instances marked as exemplar instances.

Finally, the data statistics obtained after testing on the validation and test sets provides the information that FamNet works much better than other baseline few-shot methods.

Q2.2)

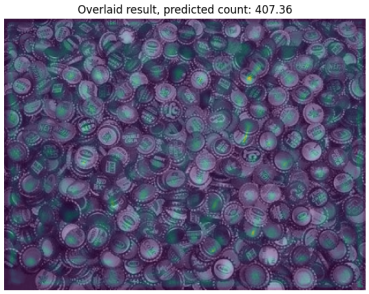
After running the test.py without test time adaption on the Val set.
The **MAE** obtained is : **24.32**
The **RMSE** obtained is : **70.94**

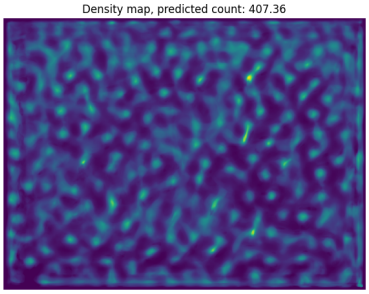
Scatter Plot :

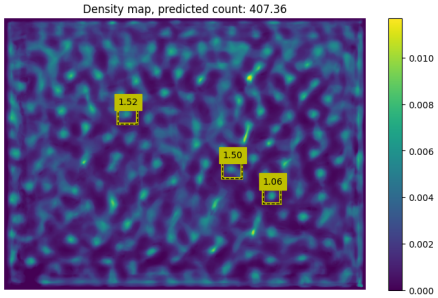


Highest Over Count Error :

1994.jpg



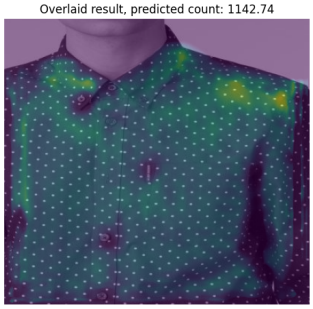




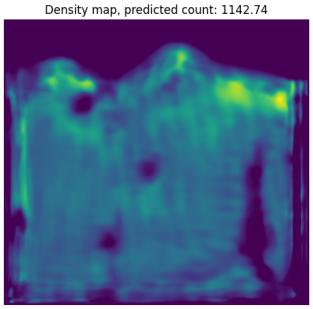
3425.jpg

Input image

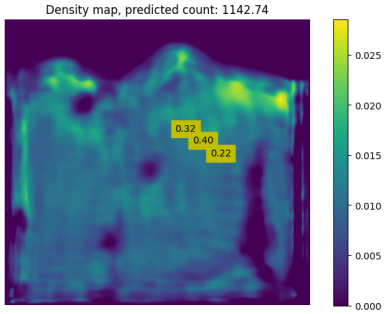
Overlaid result, predicted count: 1142.74



Density map, predicted count: 1142.74



Density map, predicted count: 1142.74

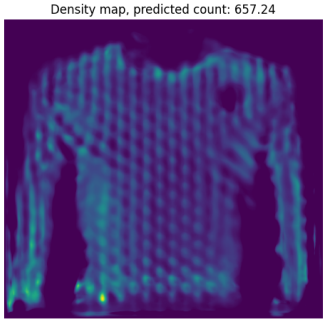


3437.jpg

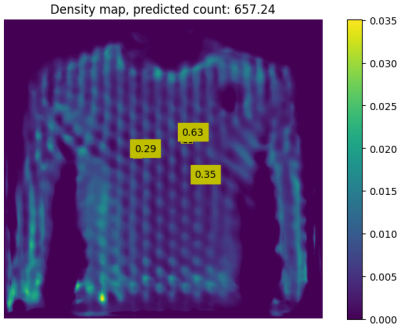
Input image

Overlaid result, predicted count: 657.24

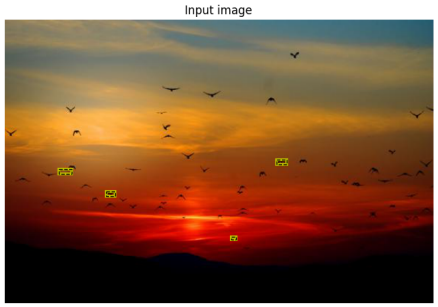
Density map, predicted count: 657.24

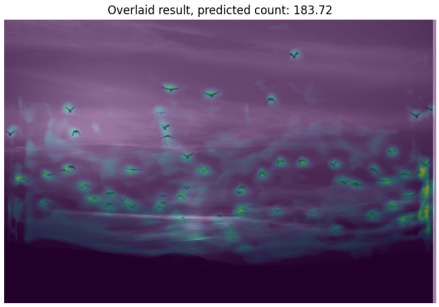


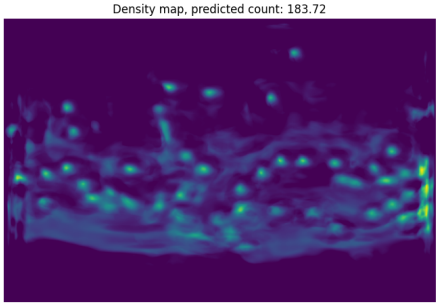
Density map, predicted count: 657.24

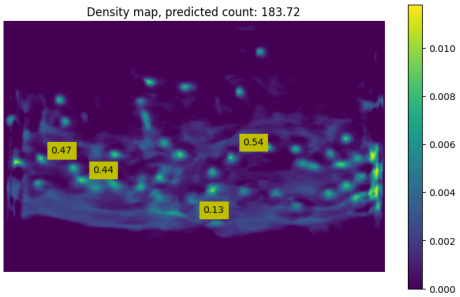


7477.jpg

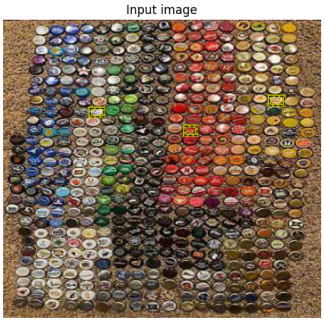


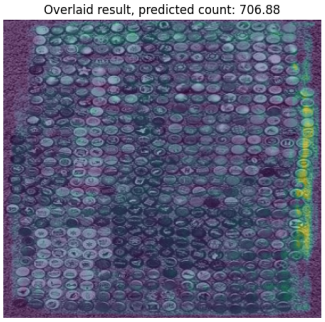


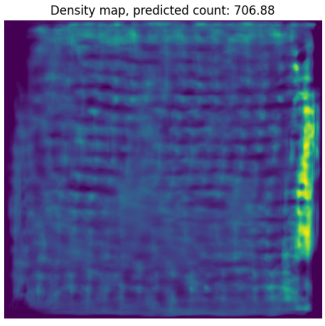


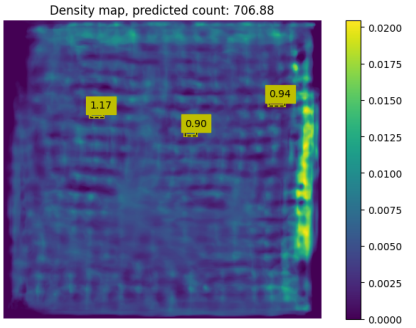


1936.jpg







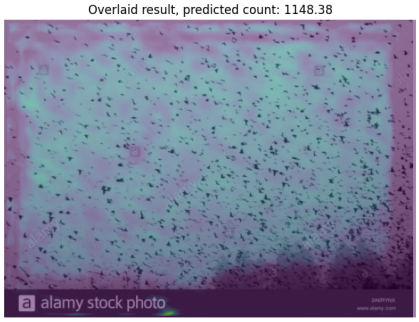


Highest Under Count Error:

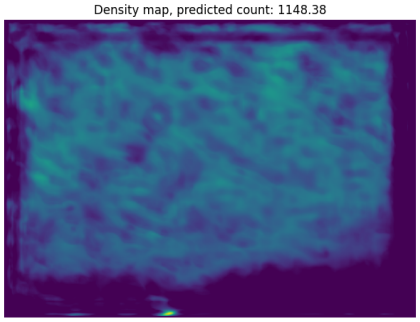
935.jpg

Input image

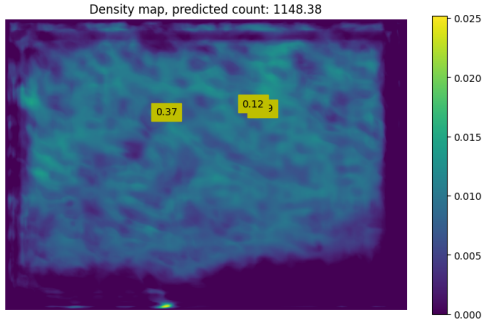
Overlaid result, predicted count: 1148.38



Density map, predicted count: 1148.38



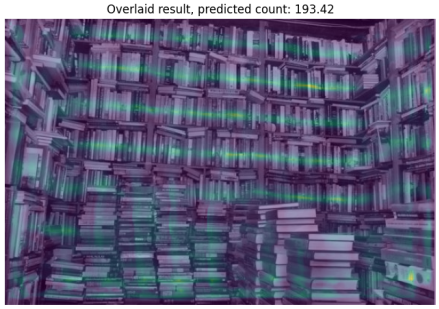
Density map, predicted count: 1148.38



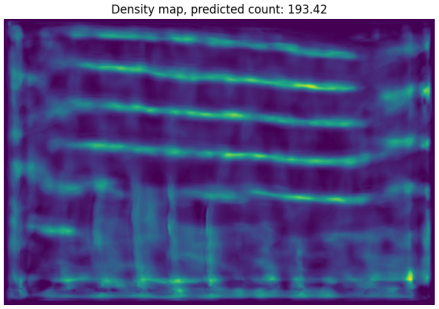
865.jpg

Input image

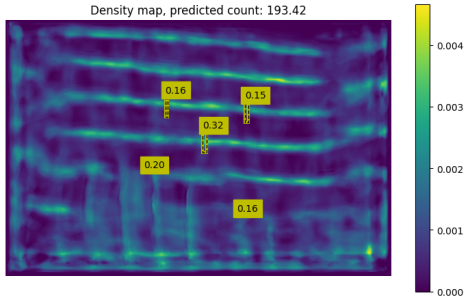
Overlaid result, predicted count: 193.42



Density map, predicted count: 193.42



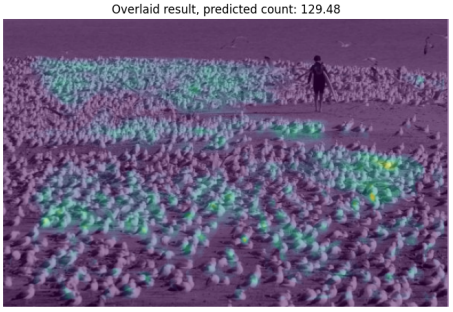
Density map, predicted count: 193.42



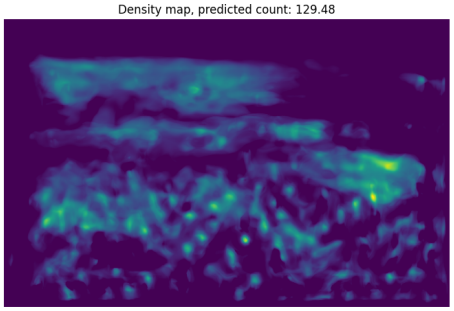
949.jpg

Input image

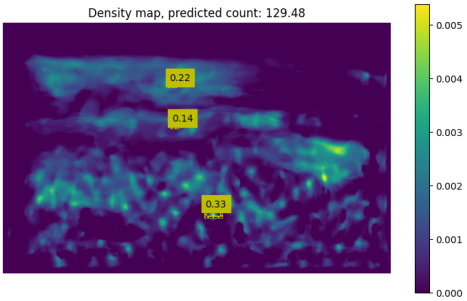
Overlaid result, predicted count: 129.48



Density map, predicted count: 129.48

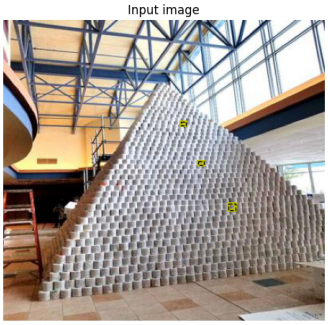


Density map, predicted count: 129.48

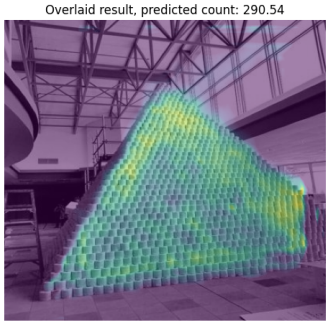


3665.jpg

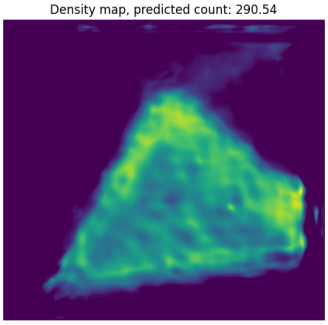
Input image



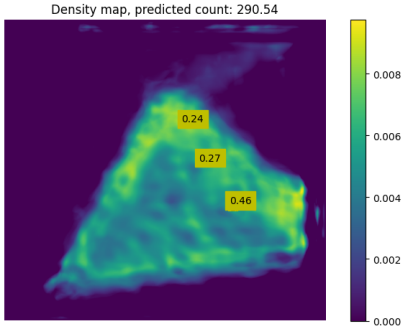
Overlaid result, predicted count: 290.54



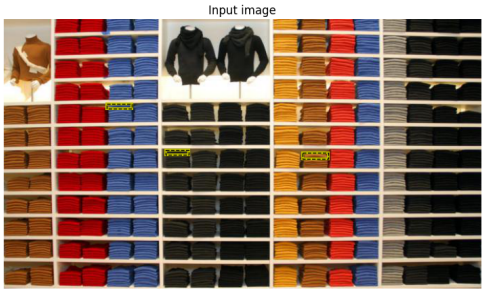
Density map, predicted count: 290.54

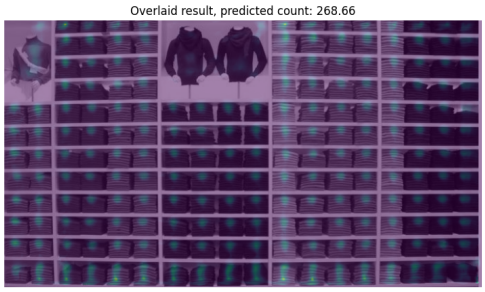


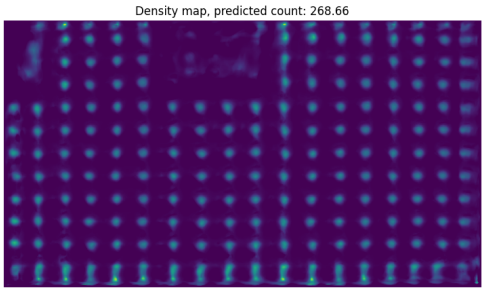
Density map, predicted count: 290.54

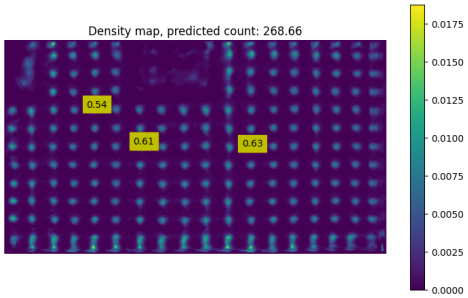


7656.jpg





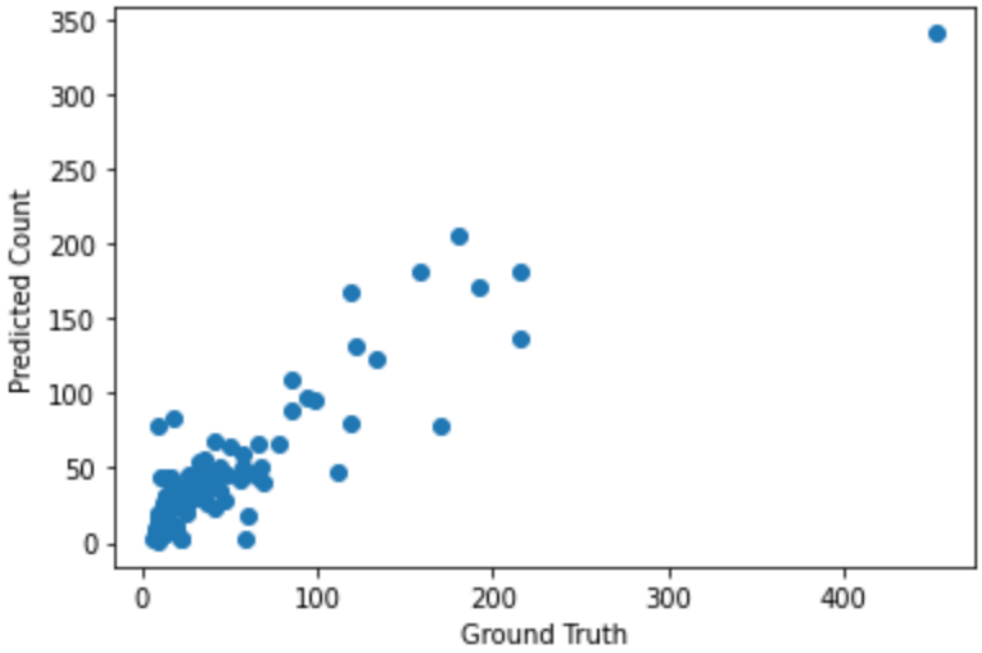




Q2.3)

Without Test Time Adaption:

MAE : **16.11**
RMSE: **25.56**

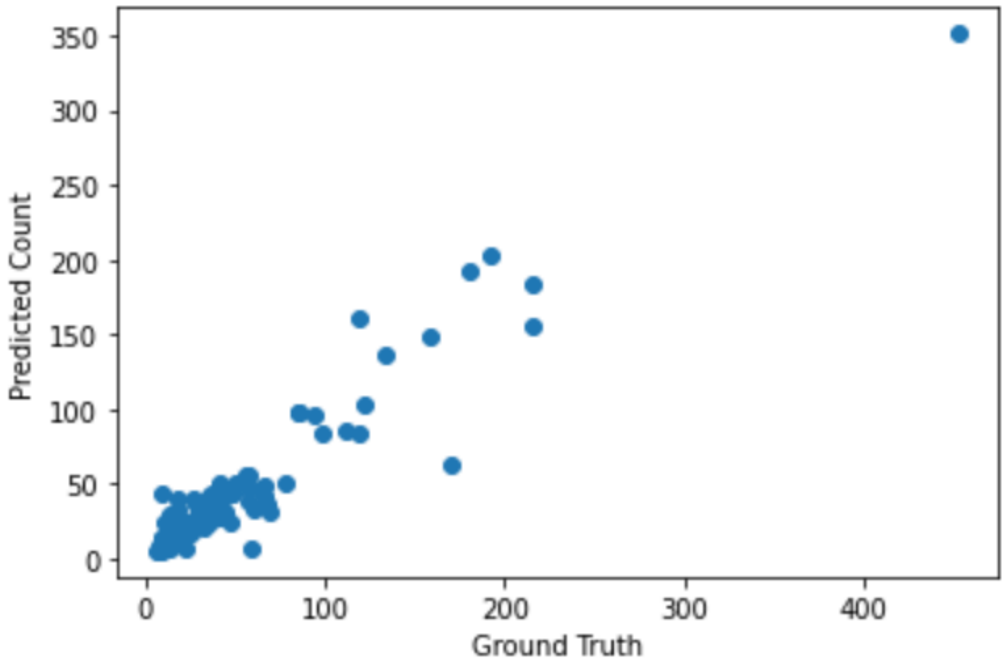


In addition to the existing Min-Count and Perturbation loss, I've included an extra Negative loss which was calculated based on the overlap of the masked image and the density outputs. After converting the masked image into 0s(blacks) and 1s(white), doing a cell wise multiplication with the density tensor and finally calculating the summation of all the positive values from the tensor gives us the loss. Finally, I've fine tuned the parameters of the existing two losses along with the new loss and number of grad steps too.

With Test Time Adaption:

MAE : **12.28**

RMSE: **12.28**


Predicted Count
Ground Truth
0
50
100
150
200
250
300
350
0
100
200
300
400