CNN Implementation for object classification with Skip Connections an Batch Normalization

Vanishing Gradient Problem:

In the previous assignment, we were introduced how we can create deeper neural networks by chaining multiple layers. But the drawback of this approach was the vanishing gradient problem. It occurs when the gradients of the loss function with respect to the parameters of the network become extremely small as they are propagated backward through the network during the training process. This causes a myriad of issues such as overfitting on the training data, a decrease in the overall level of generalization of the model and very slow convergence to name a few.

Methods to Overcome Vanishing Gradient:

1. Residual Networks/Skip Connections:

The basic idea that follows here is that the input will bypass the convolutional layers and will be added to the output of the layer. This allows for an additional pathway for the gradient to flow during the backpropogation process that bypasses the path through additional layer, hence conserving the gradient without it getting infinitesimally small.

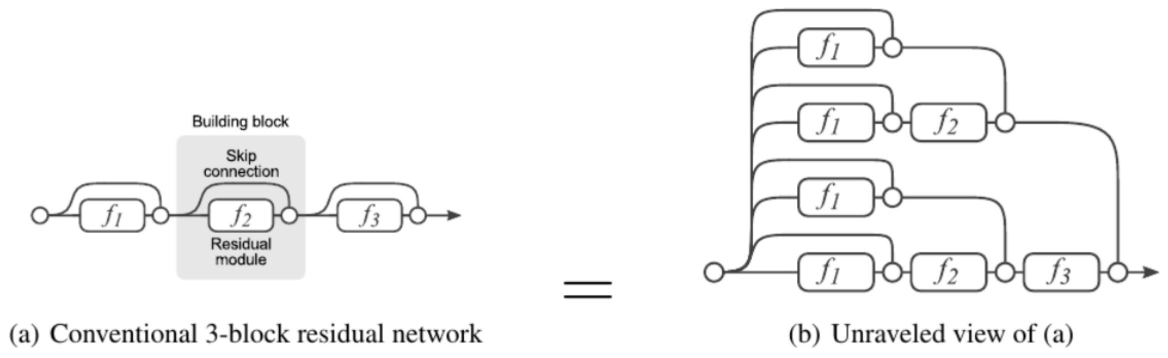

Building block
Skip connection
Residual module
f_1
f_2
f_3
=
(a) Conventional 3-block residual network
(b) Unraveled view of (a)

Fig1. From lecture 6 slide 60

2. Batch Normalization:

This is the process of normalization of a the pixel values in each channel basis the number of images in a batch. Batch normalization normalizes the inputs to each layer by subtracting the batch mean and dividing by the batch standard deviation.

$$\mu_c = \frac{1}{BHW} \sum_{n=1}^{B} \sum_{h=1}^{H} \sum_{w=1}^{W} x_{nchw}$$

$$\sigma_c = \sqrt{\frac{1}{BHW} \sum_{n=1}^{B} \sum_{h=1}^{H} \sum_{w=1}^{W} (x_{nchw} - \mu_c)^2 + \epsilon}$$

Fig2. Batch mean(mu) and Batch SD(sigma)

$$BN(x_{nchw}) = \gamma_c \frac{x_{nchw} - \mu_c}{\sigma_c} + \beta_c$$

Fig3. Pixel value calculation for a pixel in a channel, based on the mean and the SD.

Batch normalization generally helps by keeping the values on the basis of which the gradient is calculated inside a controlled range, hence keeping a check on the gradients that are calculated.

Approach:

For our network, we will be using the skip connection architecture as shown in the DL studio module, which combines the Residual Networks/Skip Connections and Batch Normalization, into a layer that we can use directly in our network. The code for the skip block that utilizes these concepts can be found below:

```
## This code has been taken from Professor Kak' DL Studio Mod
el and
## and in part also configured by referenceing previous yea
r's homework.

class SkipBlock(nn.Module):
     def __init__(self, in_ch, out_ch, downsample=False, skip
_connections=True):
        super(SkipBlock, self).__init__()
        self.downsample = downsample
        self.skip_connections = skip_connections
        self.in_ch = in_ch
        self.out_ch = out_ch
        self.convo1 = nn.Conv2d(in_ch, out_ch, 3, stride=1, p
adding=1)
        self.convo2 = nn.Conv2d(in_ch, out_ch, 3, stride=1, p
adding=1)
        self.bn1 = nn.BatchNorm2d(out_ch)
        self.bn2 = nn.BatchNorm2d(out_ch)
        if downsample:
             ##  Setting stride to 2 and kernel_size to 1 amo
unts to retaining every
            ##  other pixel in the image --- which halves the
size of the image:
            self.downsampler = nn.Conv2d(in_ch, out_ch, 1, st
ride=2)

     def forward(self, x):
        identity = x      #store the incoming data, image dat
a so that it can by pass the network is skip connection is en
abled
        out = self.convo1(x)
        out = self.bn1(out)
        out = nn.functional.relu(out)
        if self.in_ch == self.out_ch:
            out = self.convo2(out)
            out = self.bn2(out)
            out = nn.functional.relu(out)
        if self.downsample:
            out = self.downsampler(out)
            identity = self.downsampler(identity)
        if self.skip_connections:
            if self.in_ch == self.out_ch:
#               out += identity
                out = out + identity
            else:
                        ## To understand the following assign
ments, recall that the data has the
                        ## shape [B,C,H,W]. So it is the seco
nd axis that corresponds to the channels
#                        out[:,:self.in_ch,:,:] += identity
#                        out[:,self.in_ch:,:,:] += identity
```

```
                out = out + torch.cat((identity, identity), d
im=1)
        return out
```

And this block will be directly utilized as a layer when creating the neural net for this Homework, as can be seen below:

```
## This code has been taken from Professor Kak' DL Studio Mod
el and
## and in part also configured by referenceing previous yea
r's homework.

class HW5Net(nn.Module):
    def __init__(self, skip_connections=True, depth=32):
        super(HW5Net, self).__init__()
        self.depth = depth // 8
        self.conv = nn.Conv2d(3, 64, 3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.skip64_arr = nn.ModuleList()
        for i in range(self.depth):
            self.skip64_arr.append(SkipBlock(64, 64,skip_conn
ections=skip_connections))
        self.skip64ds1 = SkipBlock(64, 64, downsample=True, s
kip_connections=skip_connections) # Bug fix as professor ment
ioned in the class
        self.skip64ds2 = SkipBlock(64, 64, downsample=True, s
kip_connections=skip_connections) # Bug fix as professor ment
ioned in the class
        self.skip64to128 = SkipBlock(64, 128,skip_connections
=skip_connections )
        self.skip128_arr = nn.ModuleList()
        for i in range(self.depth):
            self.skip128_arr.append(SkipBlock(128, 128,skip_c
onnections=skip_connections))
```

```
        self.skip128ds = SkipBlock(128,128,downsample=True, s
kip_connections=skip_connections)
                ##  The following declaration is predicated o
n the assumption that the number of
                ##  output nodes (CxHxW) from the final convo
layer exactly 2048 for each
                ##  input image.  Depending on the size of th
e input image, this places a constraint
                ##  on how many downsampling instances of Ski
pBlock you can call in a network.
        self.fc1 =  nn.Linear(8192, 1000)
        self.fc2 =  nn.Linear(1000, 5)

    def forward(self, x):
        x = self.pool(nn.functional.relu(self.conv(x)))
        for i,skip64 in enumerate(self.skip64_arr[:self.dept
h//4]):
            x = skip64(x)
        x = self.skip64ds1(x) # Bug fix as professor mentione
d in the class
        # x = self.skip64ds2(x)
        for i,skip64 in enumerate(self.skip64_arr[self.dept
h//4:]):
            x = skip64(x)
        x = self.skip64ds2(x) # Bug fix as professor mentione
d in the class
        x = self.skip64to128(x)
        for i,skip128 in enumerate(self.skip128_arr[:self.dep
th//4]):
            x = skip128(x)
        for i,skip128 in enumerate(self.skip128_arr[self.dept
h//4:]):
            x = skip128(x)
        ## See the comment block above the "self.fc1" declara
tion in the constructor code.
        x  =  x.view( x.shape[0], - 1 )
        x = nn.functional.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

This code is pretty much identical to BMENet code defined in the DLStudio module, except to one fix that Professor Kak mentioned in the class that has been commented in the above code.

Note: I also tried to implement the network and skip connection with the new code Professor Kak had shared via email, but for the same the training took a much longer time and the validation accuracy was quite low and the vanishing gradient issue seemed to pop up again.

Number of Layers in the network:

```
len(list(net1.parameters()))
```

Using this we find that the number of layers is 98, when layer depth(defined in the init for HW5 Net) is 32.

I also tried running the network by changing the layer depth to 64, in that case that layer depth was 172

But I decided to revert back to 32 depth network as it was working faster and provided similar results as compared to the 64 depth network.

I have also initialized the weights for the network using Kaiming He initialization[1] to see if it got better results. As shown below:

```
def weights_init(m): # to help prevent vanishing gradient pro
blem
    if isinstance(m, nn.Conv2d):
        torch.nn.init.kaiming_uniform_(m.weight)
```

Results:

Training Loss:

1. Learning Rate = 1e-3

```
net1 = HW5Net()
net1.apply(weights_init)
epochs_net1 = 15
batch_size_net1 = 4
lr1 = 1e-3
net1_training_loss = model_training(net1, epochs_net1, train_
dataloader, torch.device("mps"), os.path.join('/Users/avnishk
anungo/Desktop/coco-dataset/train2017/hw5_dataset', 'net1.pt
h'),lr1)
confusion_matrix_net1, net1_testing_acc = model_testing(net1,
test_dataloader, batch_size_net1, torch.device("mps"), classe
s,os.path.join('/Users/avnishkanungo/Desktop/coco-dataset/tra
in2017/hw5_dataset', 'net1.pth'))
```

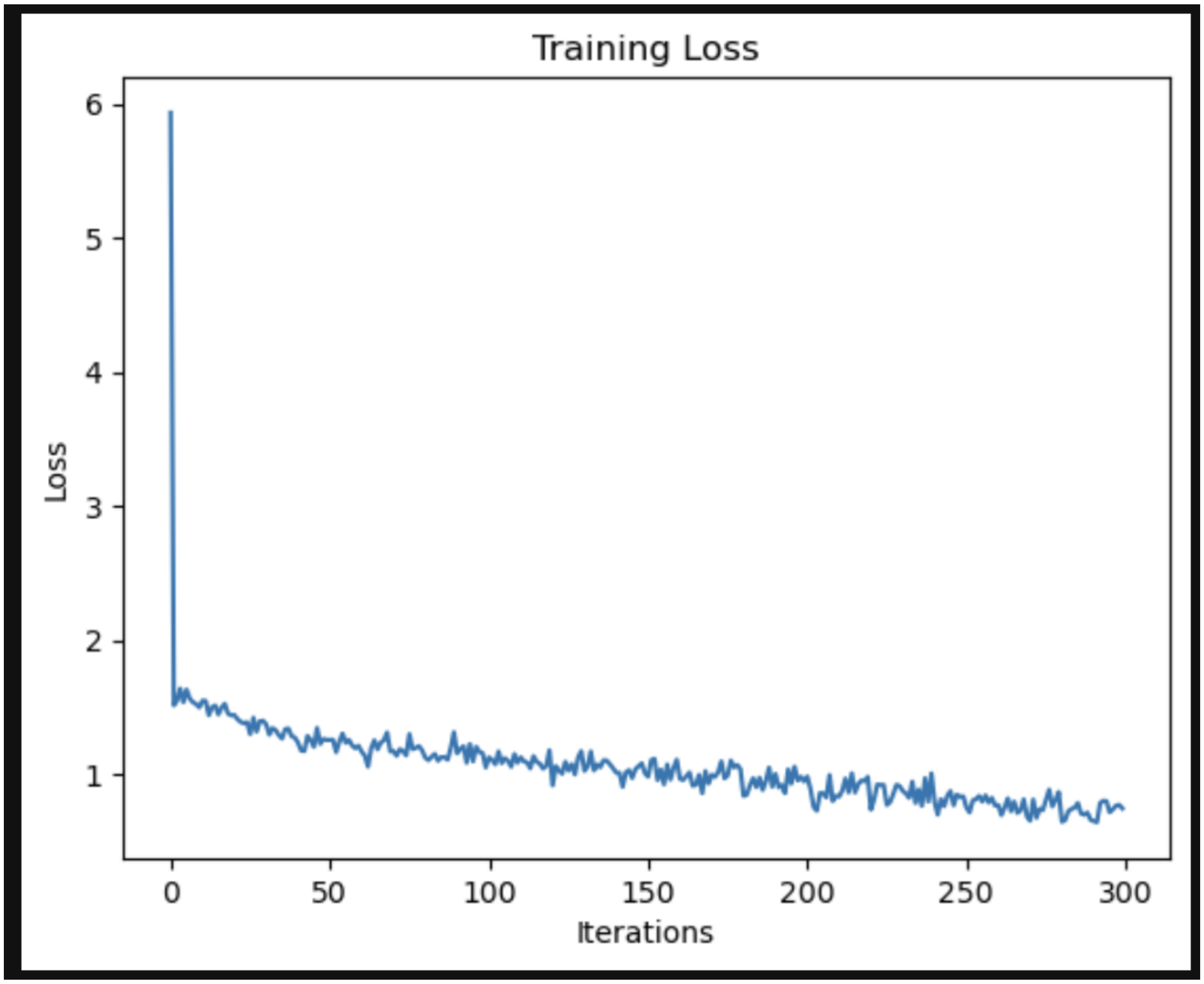


Fig. 4

2. Learning Rate = 2e-5

```
net2 = HW5Net()
net2.apply(weights_init)
epochs_net2 = 15
batch_size_net2 = 4
lr2 = 2e-5
net2_training_loss = model_training(net2, epochs_net2, train_
dataloader, torch.device("mps"), os.path.join('/Users/avnishk
anungo/Desktop/coco-dataset/train2017/hw5_dataset', 'net2.pt
h'),  lr2)
confusion_matrix_net2, net2_testing_acc = model_testing(net2,
test_dataloader, batch_size_net2, torch.device("mps"), classe
s,os.path.join('/Users/avnishkanungo/Desktop/coco-dataset/tra
in2017/hw5_dataset', 'net2.pth'))
```

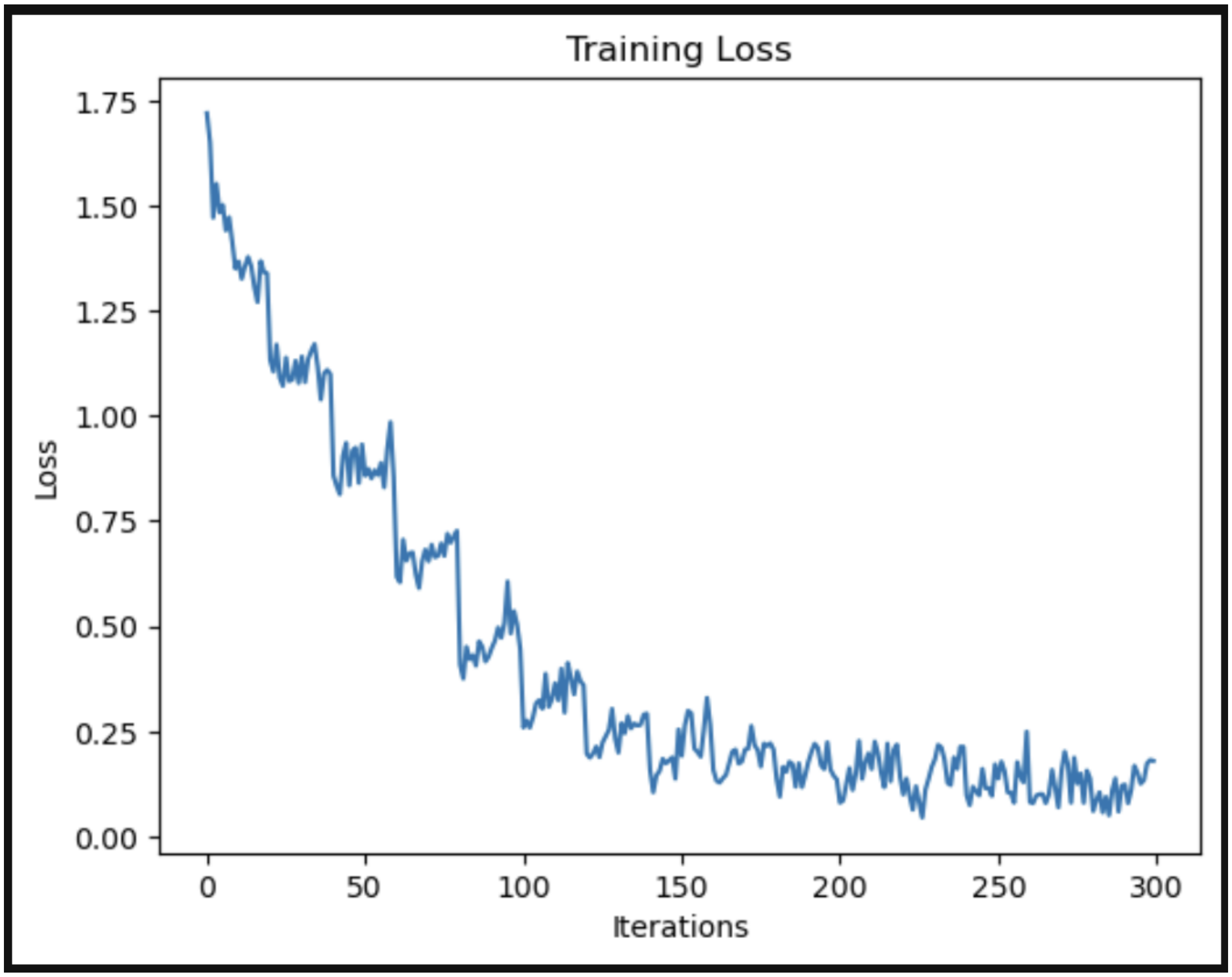
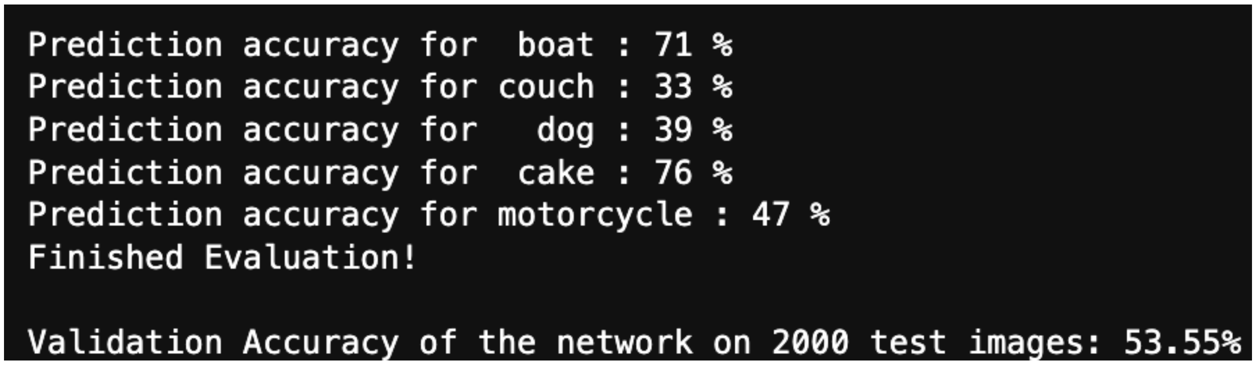


Fig. 5

Accuracy:

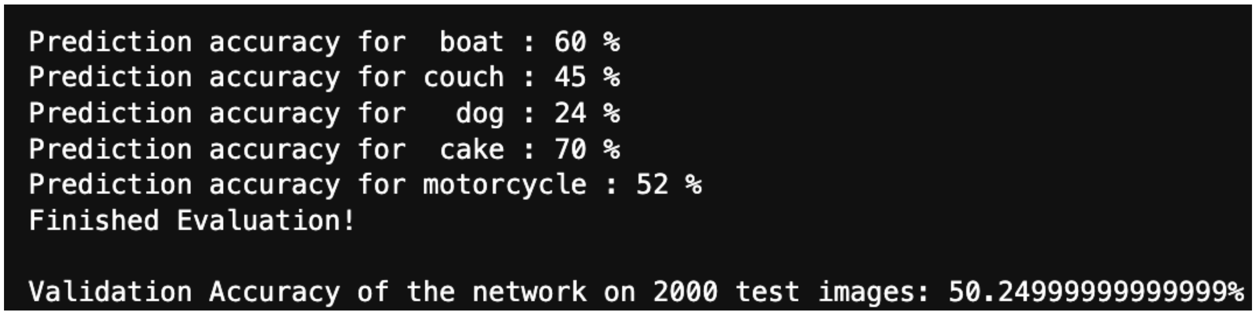
1. Learning Rate = 1e-3



```
Prediction accuracy for  boat : 71 %
Prediction accuracy for couch : 33 %
Prediction accuracy for   dog : 39 %
Prediction accuracy for  cake : 76 %
Prediction accuracy for motorcycle : 47 %
Finished Evaluation!

Validation Accuracy of the network on 2000 test images: 53.55%
```

2. Learning Rate = 2e-5



```
Prediction accuracy for  boat : 60 %
Prediction accuracy for couch : 45 %
Prediction accuracy for   dog : 24 %
Prediction accuracy for  cake : 70 %
Prediction accuracy for motorcycle : 52 %
Finished Evaluation!

Validation Accuracy of the network on 2000 test images: 50.24999999999999%
```

Confusion Matrix:

1. Learning Rate = 1e-3

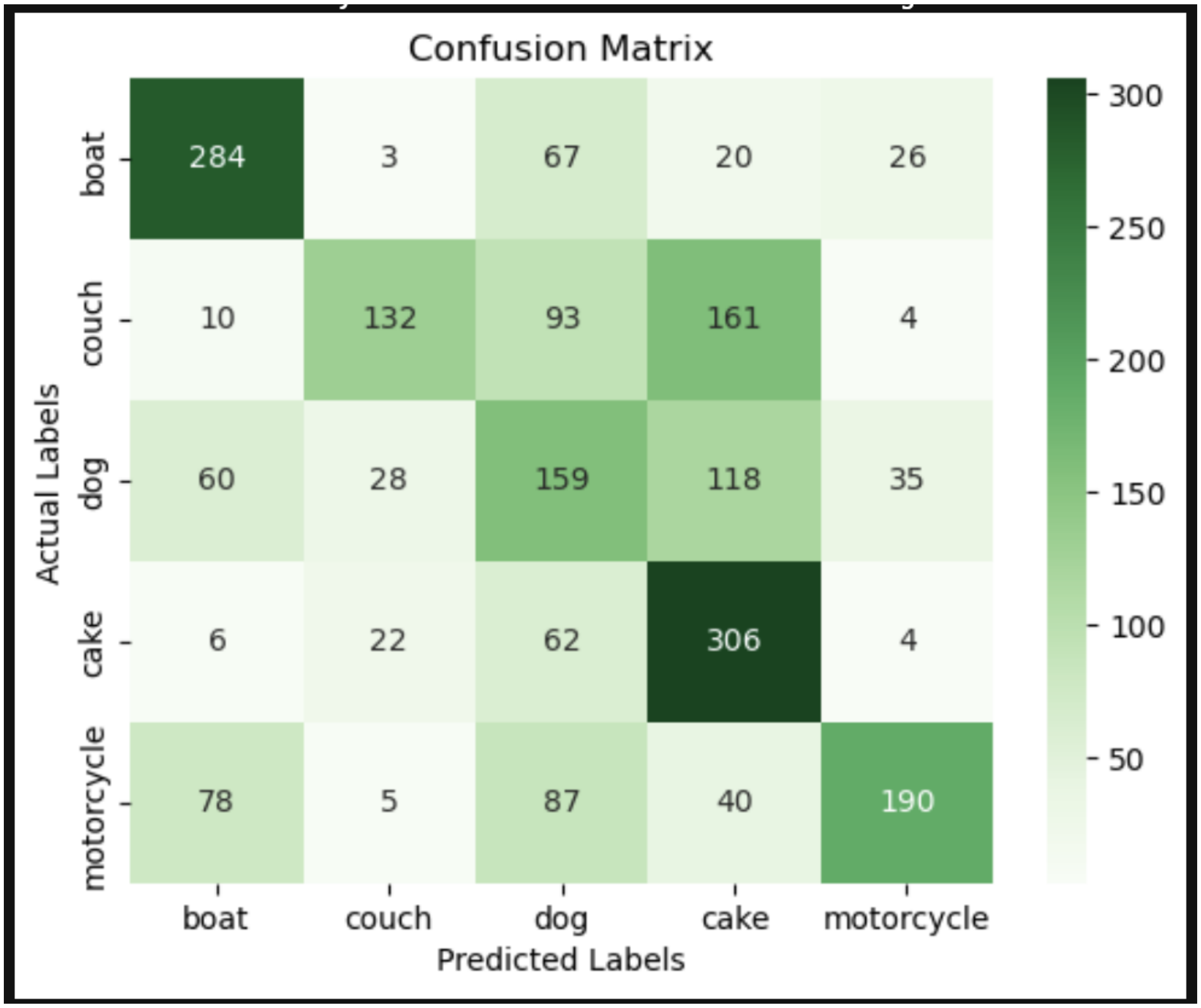


Fig. 6

2. Learning Rate = 2e-5

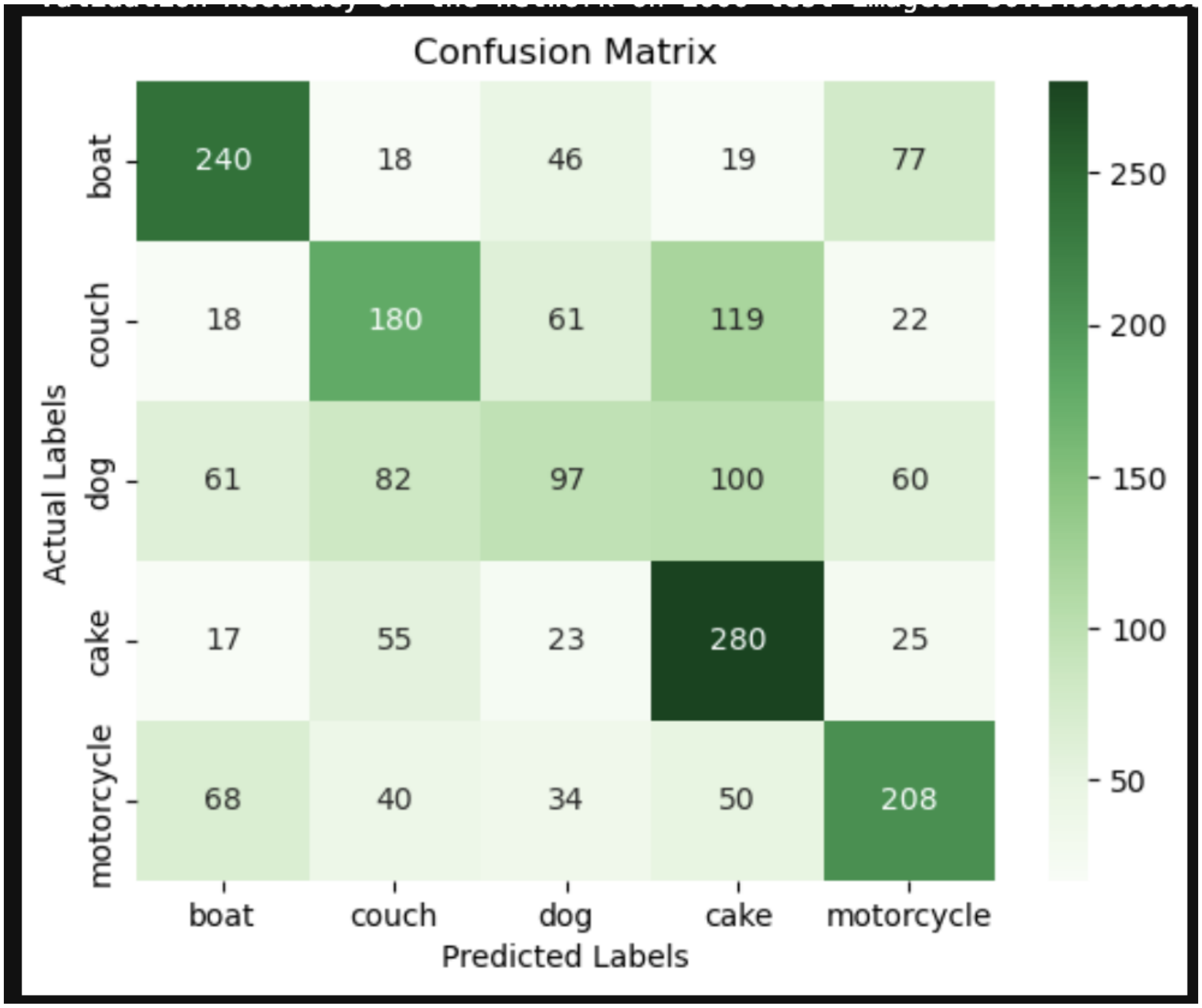


Fig. 7

- The validation accuracy for the network with lr(learning rate)=1e-3 is around 3-4% higher than the network with lr=2e-5. But we see that the running training loss for lr=2e-5 is much lower at the end than the network with lr=1e-3. So what we can observe is that the network with the higher lr is more generalizable as compared to the one with the lower lr and performs better on the validation dataset, while the training loss falls farther for the network with the lower lr, hence that seems to be performing better in the training part, which might be a sign pointing towards the fact that lower lrs, around 2e-5, might be overfitting on the training data.
- Another point to notice is that out of all classes, the “dog” class has the lowers classification accuracy and is the major cause of the lower accuracy. This

might be due to the fact that for images classified as dogs other classes are also likely to show up and our network is not able to correctly discriminate which is the dominant class.

Observations in comparison of HWNet5 to Net3(Homework 4):

- The implementation in HWNet5 with the skip blocks led to an increase in the validation accuracy by
- For my implementation of net 3 I was able to consistently prevent the vanishing gradient problem by using Xavier initialization, but the skip blocks(implementing skip connections and batch normalization) are much more efficient as they help prevent the vanishing gradient problem as well as led to an overall increase of validation accuracy as compared to Net3.
- With the Skip blocks we can make even deeper networks, while also preventing the Vanishing Gradient problem

Confusion Matrix of Net3(lr =1e-3):

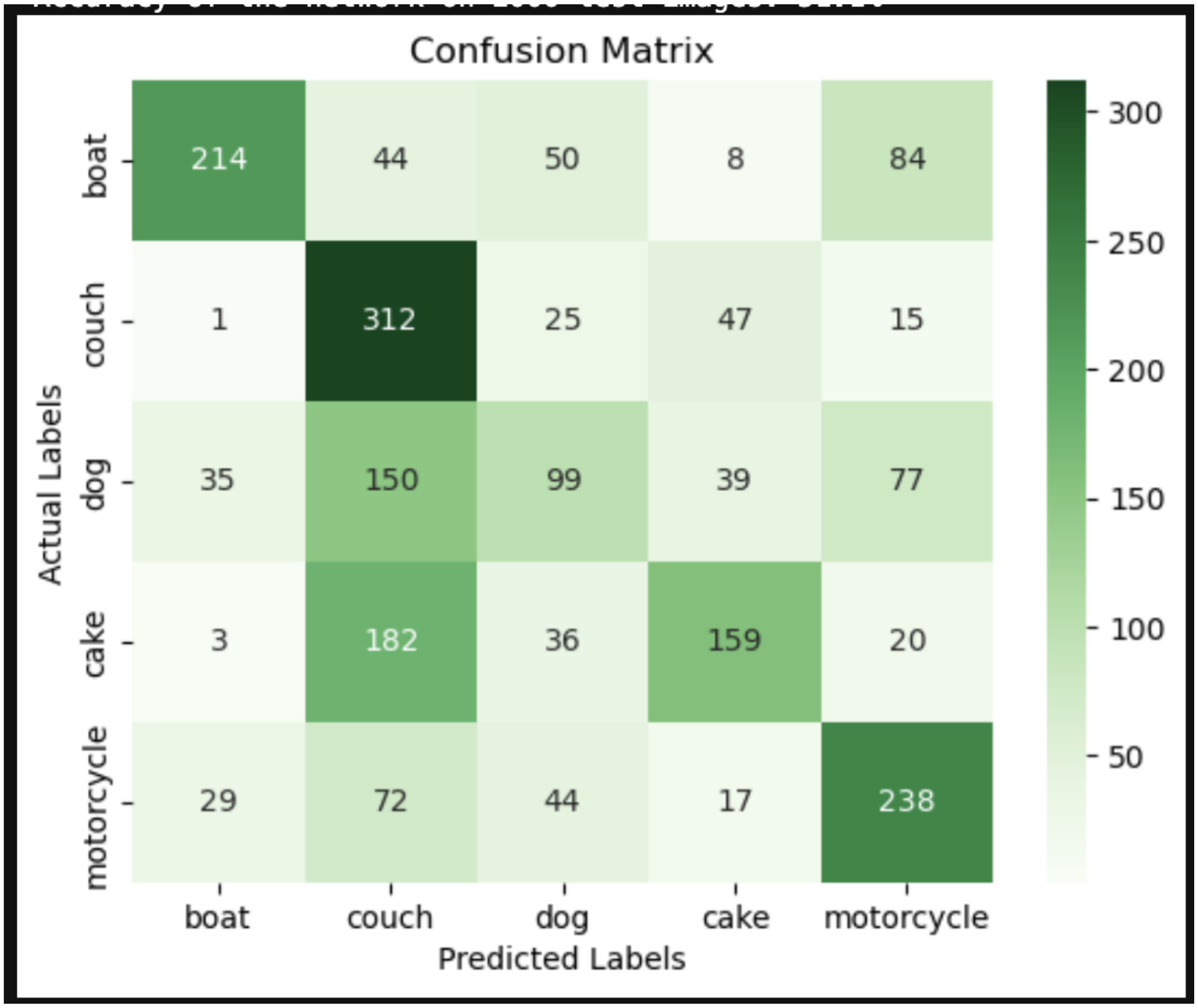

Confusion Matrix
Actual Labels
Predicted Labels
boat
couch
dog
cake
motorcycle
214
44
50
8
84
1
312
25
47
15
35
150
99
39
77
3
182
36
159
20
29
72
44
17
238
300
250
200
150
100
50

Fig. 8