Skip Connections and Batch Normalization in CNN

Shubham Kose (skose@purdue.edu)

Introduction:

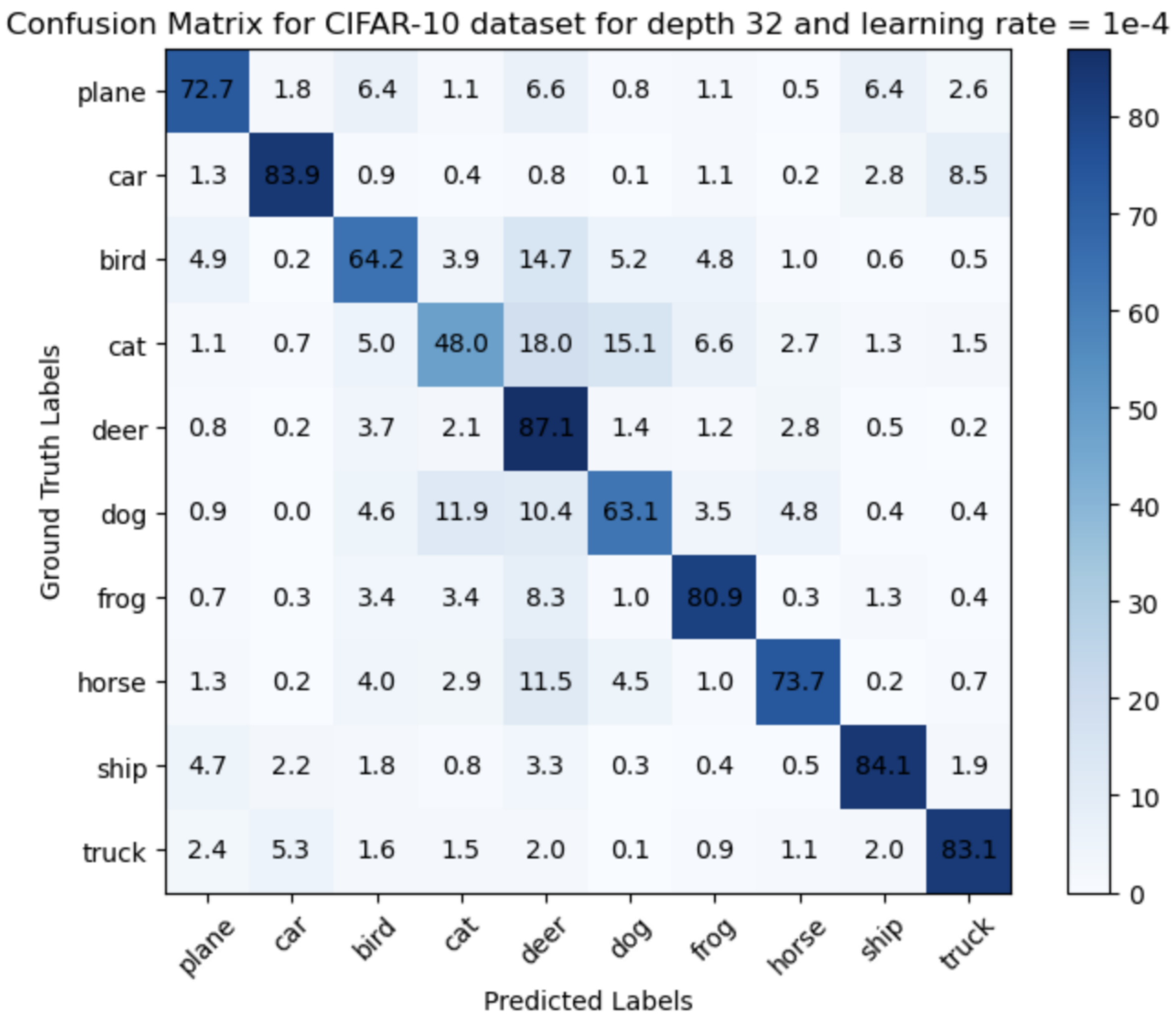
The goal of this homework is for us to acquire some preliminary experience with using a new network building-block element known as a SkipBlock or a ResBlock for mitigating the problem of vanishing gradients.

Skip connections are used to address the vanishing gradient problem in deep neural networks. The vanishing gradient problem occurs when training deep networks, and it becomes challenging for gradients to propagate through many layers, leading to slow learning and convergence.

Skip connections are also known as identity shortcuts and work by allowing the output from one layer to be added directly to the output of a layer that is not necessarily its immediate successor. The basic building block of skip connection is the residual block or skip block, which includes a shortcut connection. They allow the gradient to flow more easily during backpropagation by skipping one or more layers.

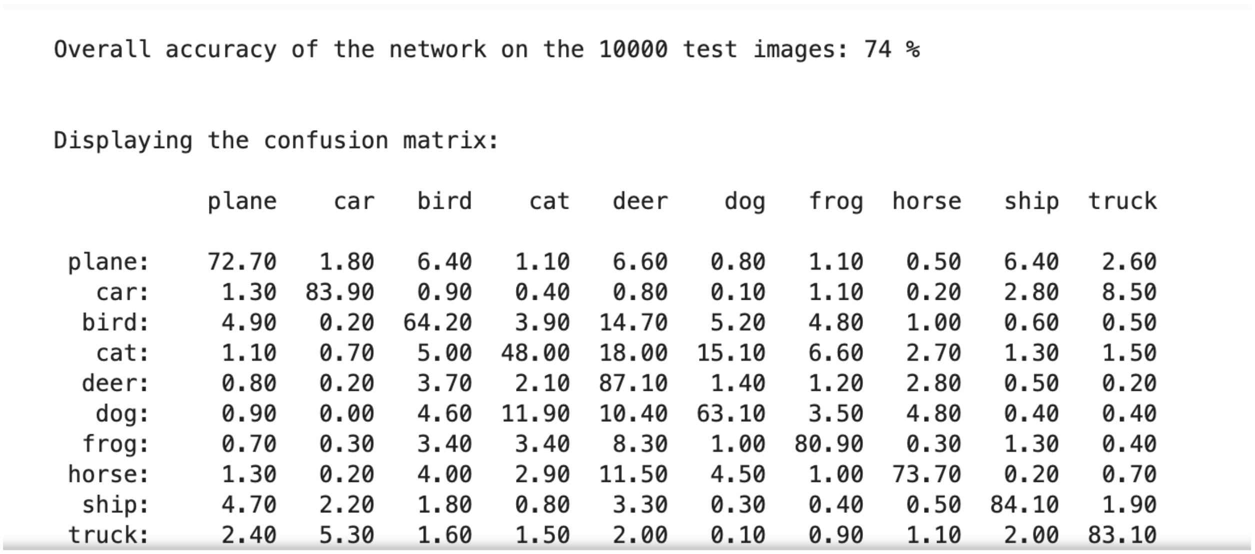
Task 2:

Confusion matrix generated for CIFAR dataset using the playing_with_skip_connections.py:



Ground Truth \ Predicted	plane	car	bird	cat	deer	dog	frog	horse	ship	truck
plane	72.7	1.8	6.4	1.1	6.6	0.8	1.1	0.5	6.4	2.6
car	1.3	83.9	0.9	0.4	0.8	0.1	1.1	0.2	2.8	8.5
bird	4.9	0.2	64.2	3.9	14.7	5.2	4.8	1.0	0.6	0.5
cat	1.1	0.7	5.0	48.0	18.0	15.1	6.6	2.7	1.3	1.5
deer	0.8	0.2	3.7	2.1	87.1	1.4	1.2	2.8	0.5	0.2
dog	0.9	0.0	4.6	11.9	10.4	63.1	3.5	4.8	0.4	0.4
frog	0.7	0.3	3.4	3.4	8.3	1.0	80.9	0.3	1.3	0.4
horse	1.3	0.2	4.0	2.9	11.5	4.5	1.0	73.7	0.2	0.7
ship	4.7	2.2	1.8	0.8	3.3	0.3	0.4	0.5	84.1	1.9
truck	2.4	5.3	1.6	1.5	2.0	0.1	0.9	1.1	2.0	83.1

Accuracy attained : 74%



```
Overall accuracy of the network on the 10000 test images: 74 %


Displaying the confusion matrix:

            plane    car   bird    cat   deer    dog   frog  horse   ship  truck

 plane:     72.70   1.80   6.40   1.10   6.60   0.80   1.10   0.50   6.40   2.60
   car:      1.30  83.90   0.90   0.40   0.80   0.10   1.10   0.20   2.80   8.50
  bird:      4.90   0.20  64.20   3.90  14.70   5.20   4.80   1.00   0.60   0.50
   cat:      1.10   0.70   5.00  48.00  18.00  15.10   6.60   2.70   1.30   1.50
  deer:      0.80   0.20   3.70   2.10  87.10   1.40   1.20   2.80   0.50   0.20
   dog:      0.90   0.00   4.60  11.90  10.40  63.10   3.50   4.80   0.40   0.40
  frog:      0.70   0.30   3.40   3.40   8.30   1.00  80.90   0.30   1.30   0.40
 horse:      1.30   0.20   4.00   2.90  11.50   4.50   1.00  73.70   0.20   0.70
  ship:      4.70   2.20   1.80   0.80   3.30   0.30   0.40   0.50  84.10   1.90
 truck:      2.40   5.30   1.60   1.50   2.00   0.10   0.90   1.10   2.00  83.10
```

Observations:

The results above are pretty good for the CIFAR- 10 dataset because of the number of images used for training which are substantially high as compared to the subset of images we are using below from the COCO dataset for our training.

Task 3.1: Building own Deep Convolutional Neural Network and training and evaluating the network:

Below is the created `SkipBlock` class that represents a residual block with optional skip connections. It takes input with `in_ch` channels and produces output with `out_ch` channels. The block consists of two convolutional layers, each followed by batch normalization and rectified linear unit (ReLU) activation. If the `downsample` flag is set, a 1×1 convolution is used to downsample the input. The skip connections are employed if specified, allowing the output to be added to the input, either directly or after concatenation in case of different channel dimensions.

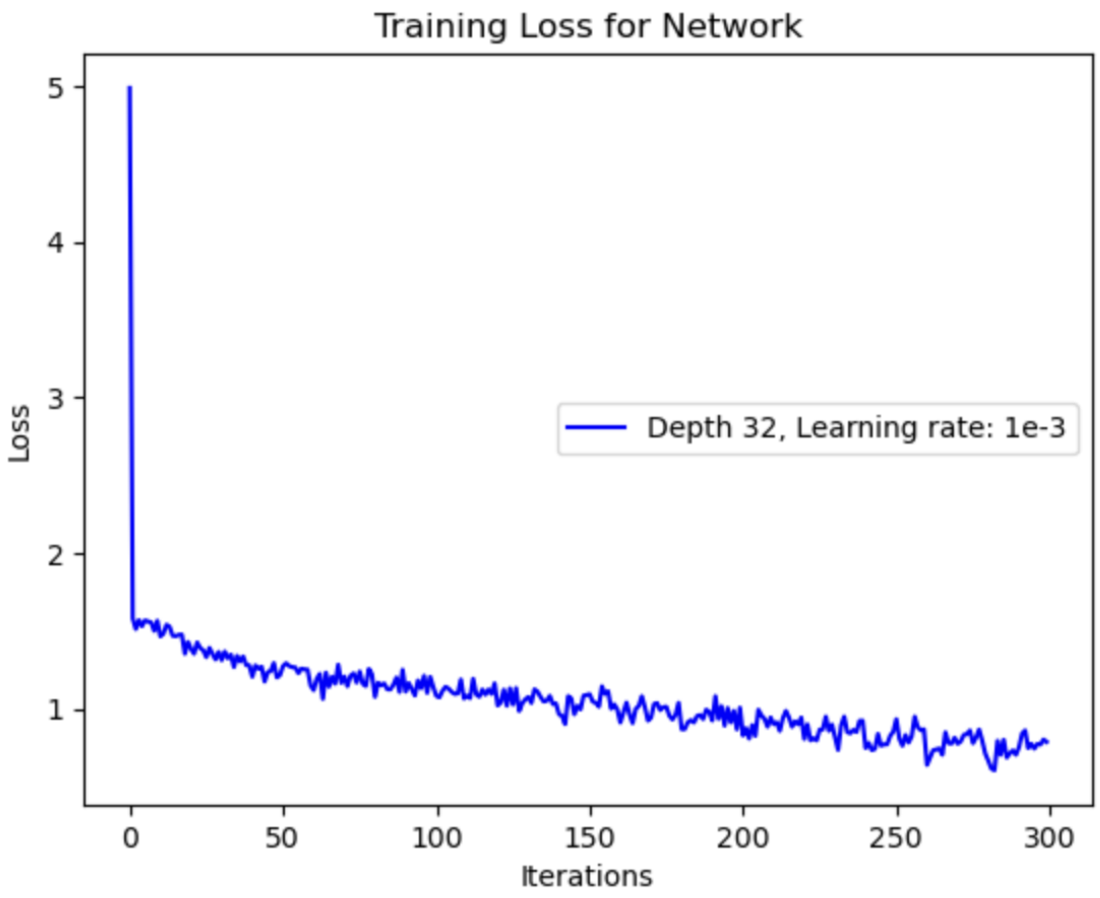
I created the `HW5Net` class along the lines of BMEnet class from DLStudio module by Dr. Kak, this class assembles a hierarchical network by stacking instances of `SkipBlock` with varying depths. The network starts with a convolutional layer followed by max-pooling. It then utilizes skip connections between residual blocks, allowing information to bypass multiple convolutional layers. The final layers include two fully connected layers (`fc1` and `fc2`) for classification. The depth of the network and skip connection usage is configured through the constructor parameters for different scenarios and the results are visualized in following pages:

```
class HW5Net(nn.Module):
    def __init__(self, skip_connections=True, depth=32):
        super().__init__()
        if depth not in [8, 16, 32, 64]:
            sys.exit("BMEnet has been tested for depth for only 8, 16, 32, and 64")
        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_connections=skip_connections))
        self.skip64ds = SkipBlock(64, 64, downsample=True, skip_connections=skip_connections)
        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_connections=skip_connections))
        self.skip128ds = SkipBlock(128, 128, downsample=True, skip_connections=skip_connections)

        # The following declaration is predicated on the assumption that the number of
        # output nodes (CxHxW) from the final convo layer is exactly 2048 for each
        # input image. Depending on the size of the input image, this places a constraint
        # on how many downsampling instances of SkipBlock you can call in a network.
        self.fc1 = nn.Linear(8192, 1000) # input size = 8192 for
        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.depth//4]):
            x = skip64(x)
        x = self.skip64ds(x)
        for i, skip64 in enumerate(self.skip64_arr[self.depth//4:]):
            x = skip64(x)
        x = self.skip64ds(x)
        x = self.skip64to128(x)
        for i, skip128 in enumerate(self.skip128_arr[:self.depth//4]):
            x = skip128(x)
        for i, skip128 in enumerate(self.skip128_arr[self.depth//4:]):
            x = skip128(x)
        # See the comment block above the "self.fc1" declaration in the constructor code.
        x = x.view(x.shape[0], - 1)
        x = nn.functional.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

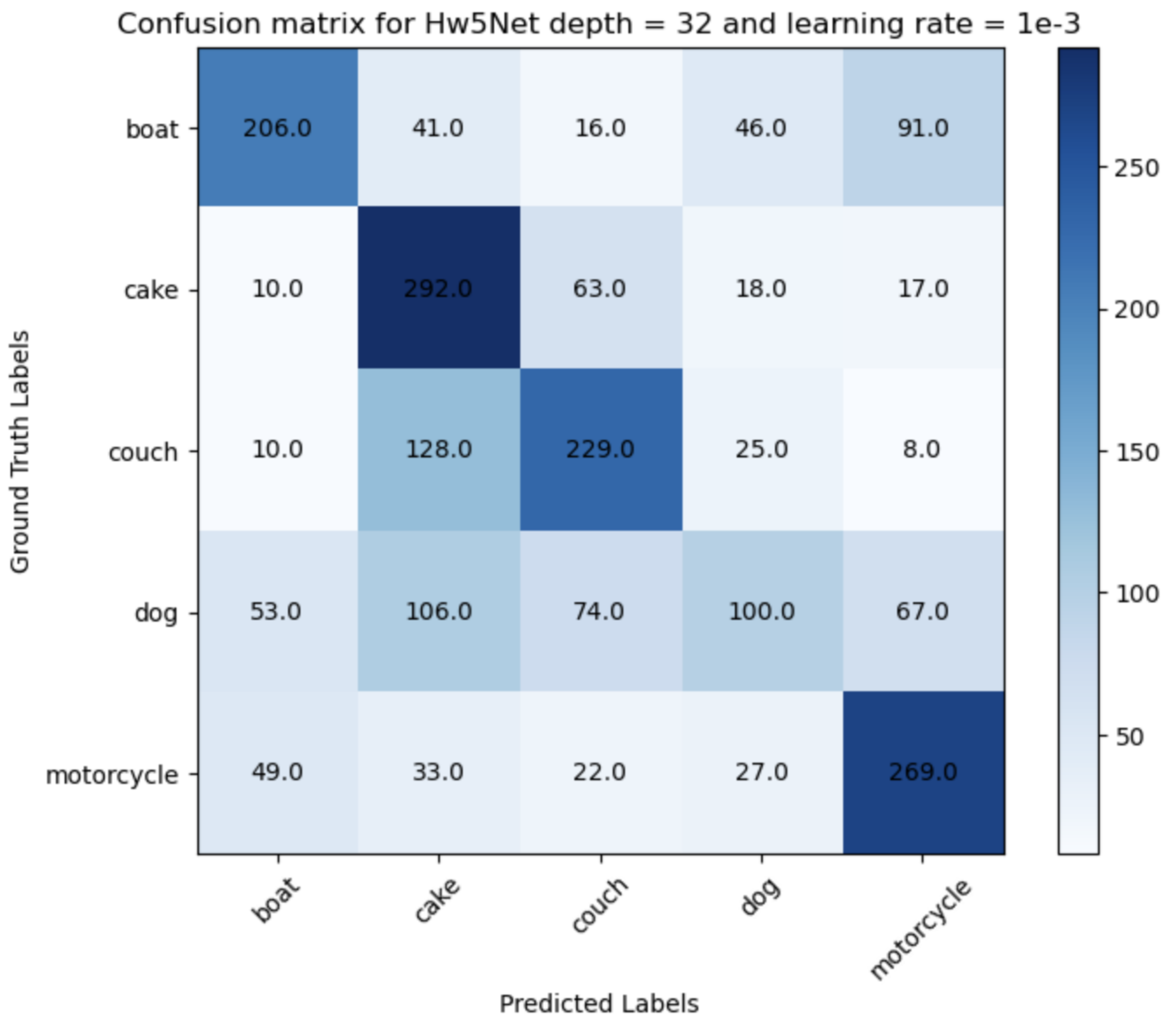
Training loss and accuracy metrics for two different learning rates for Skip block Depth = 32:

Total number of learnable layers for this depth : 98

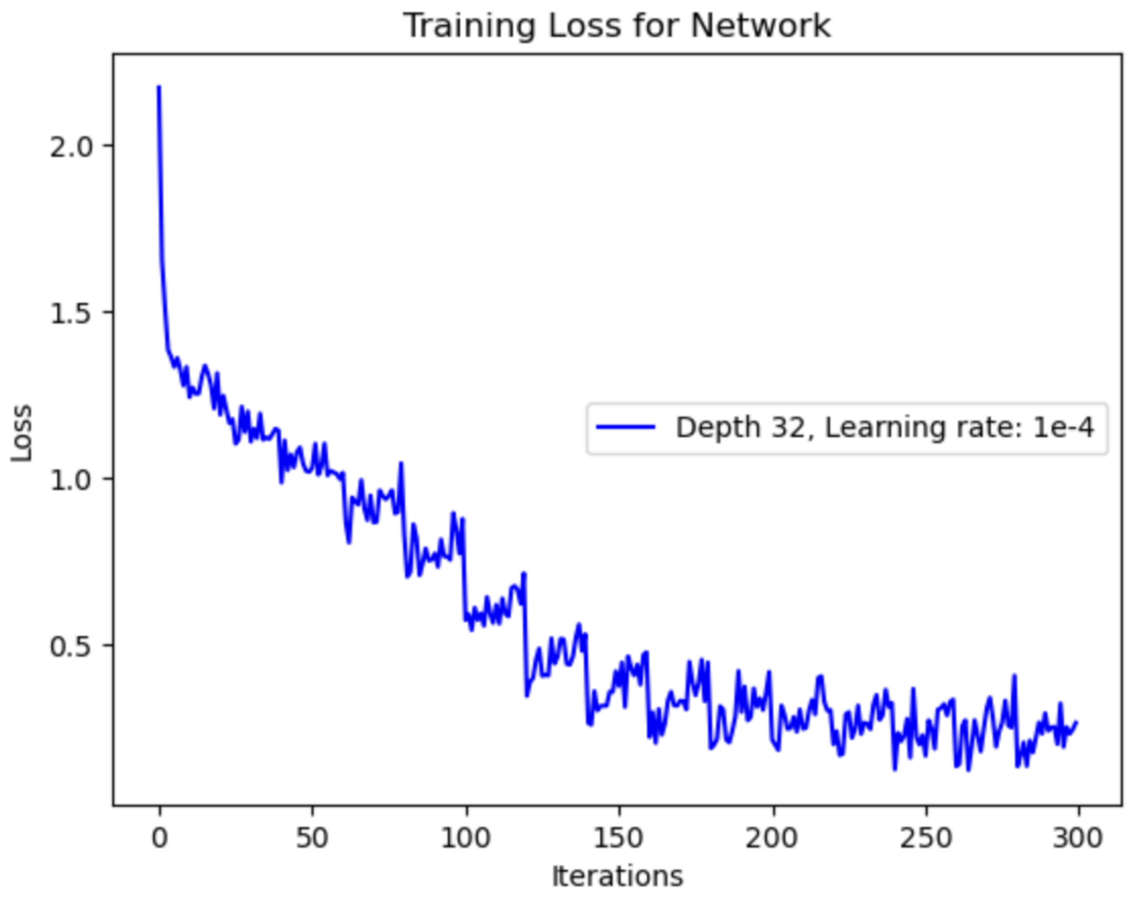
Learning rate = 1e-3:



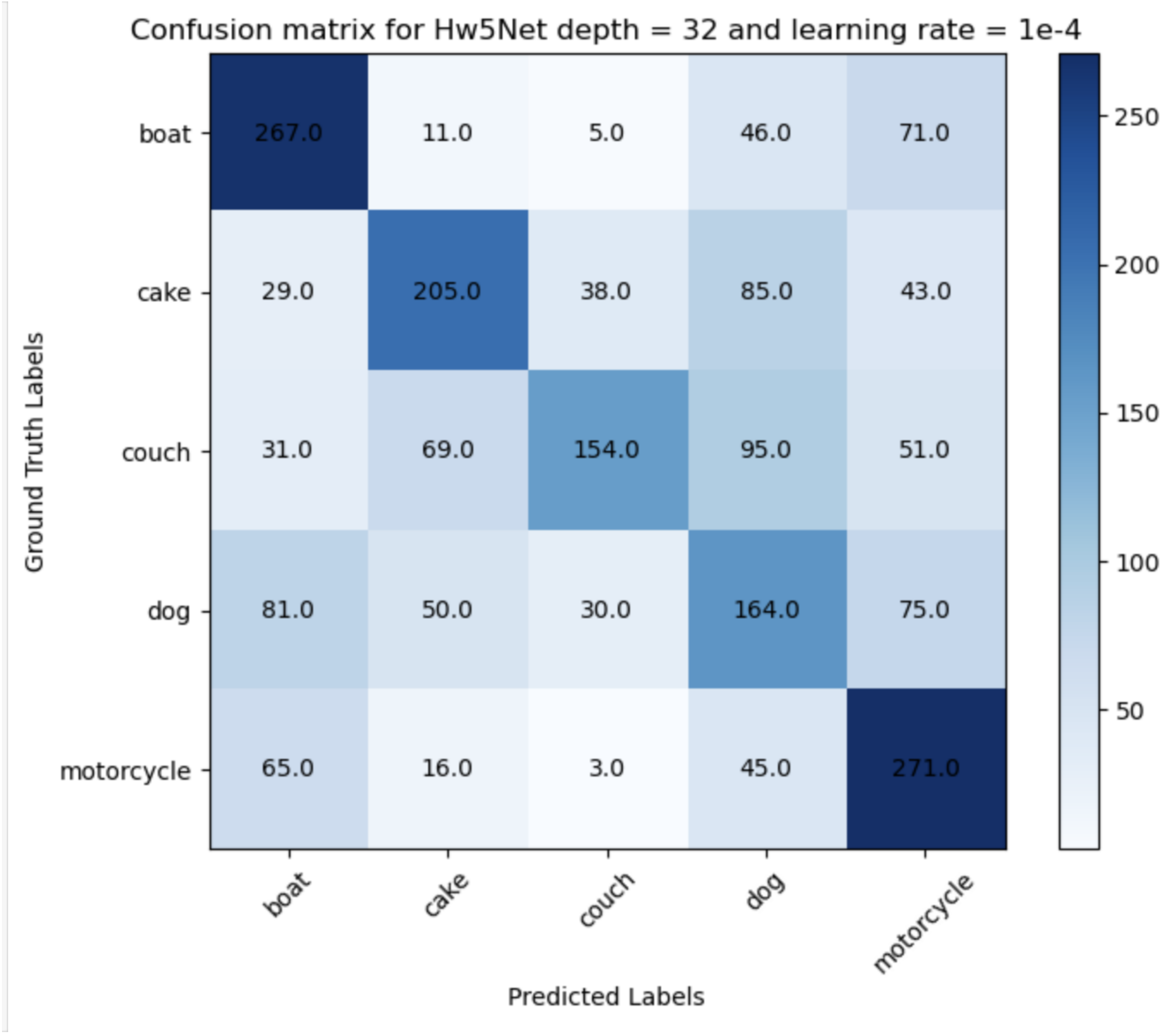
As can be seen from below confusion matrix the accuracy for the network is : 54.80%



Learning rate = 1e-4:



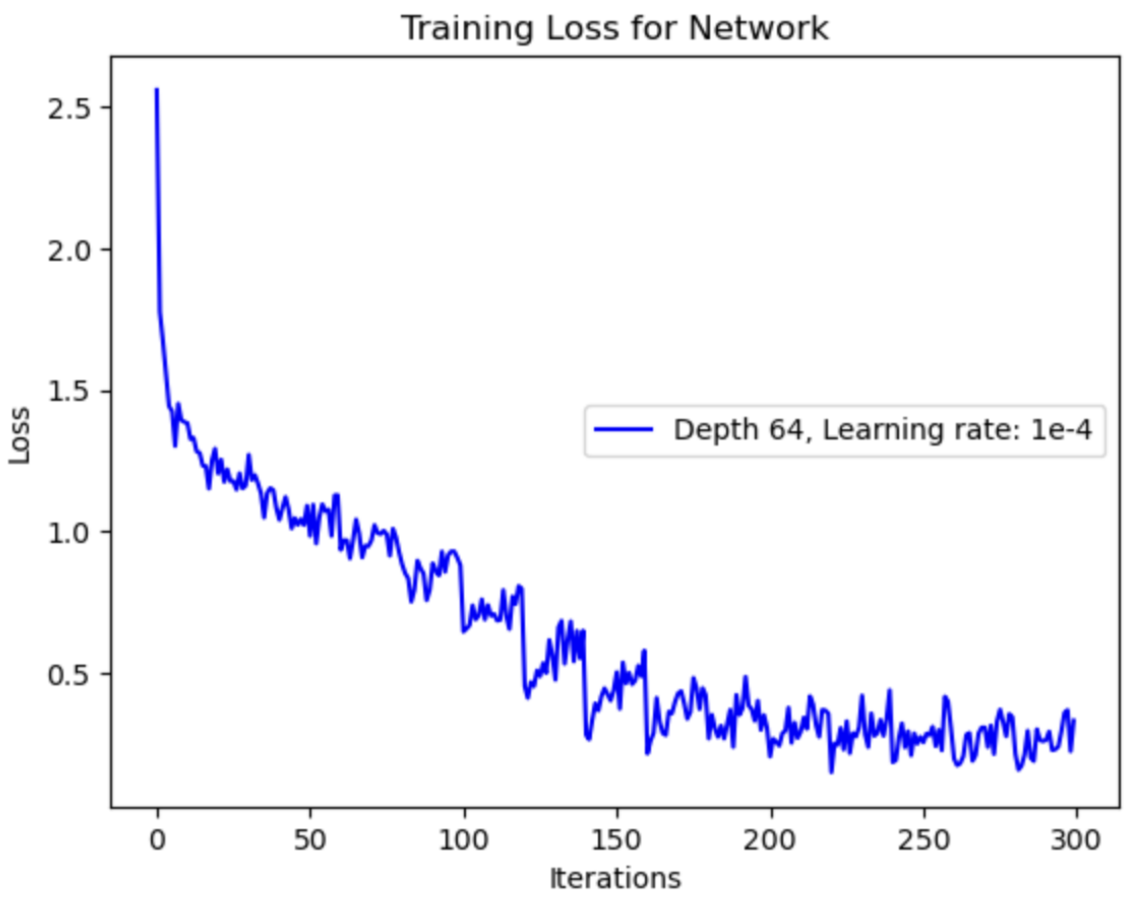
As can be seen from below confusion matrix the accuracy for the network is : 53.05%



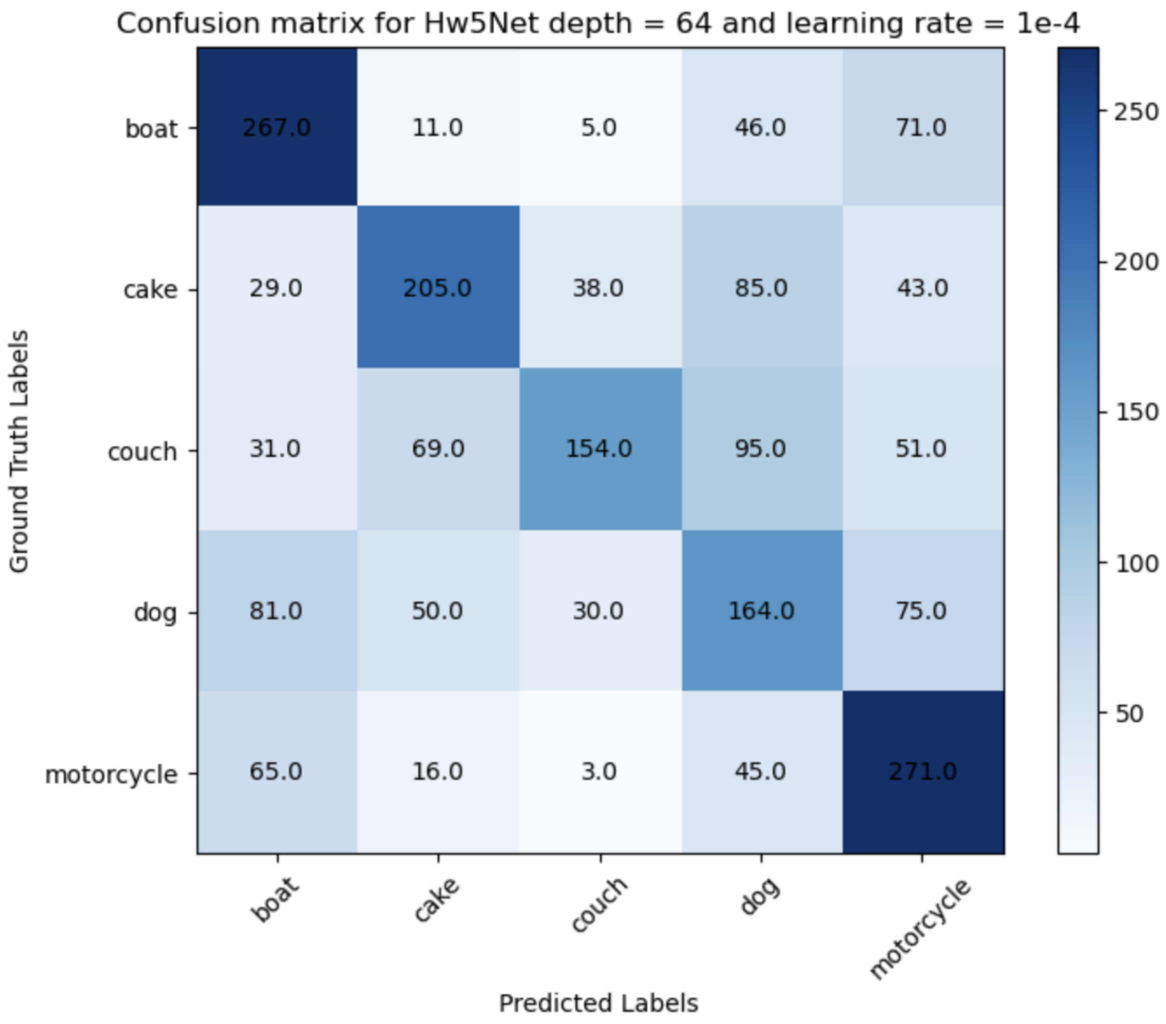
Training loss and accuracy metrics for two different learning rates for Skip block Depth = 64 and learning rate =1e-4:

I tried training the network using 64 depth but did not obtain significant improvement in the accuracy, as seen below:

Total number of learnable layers for this depth : 162



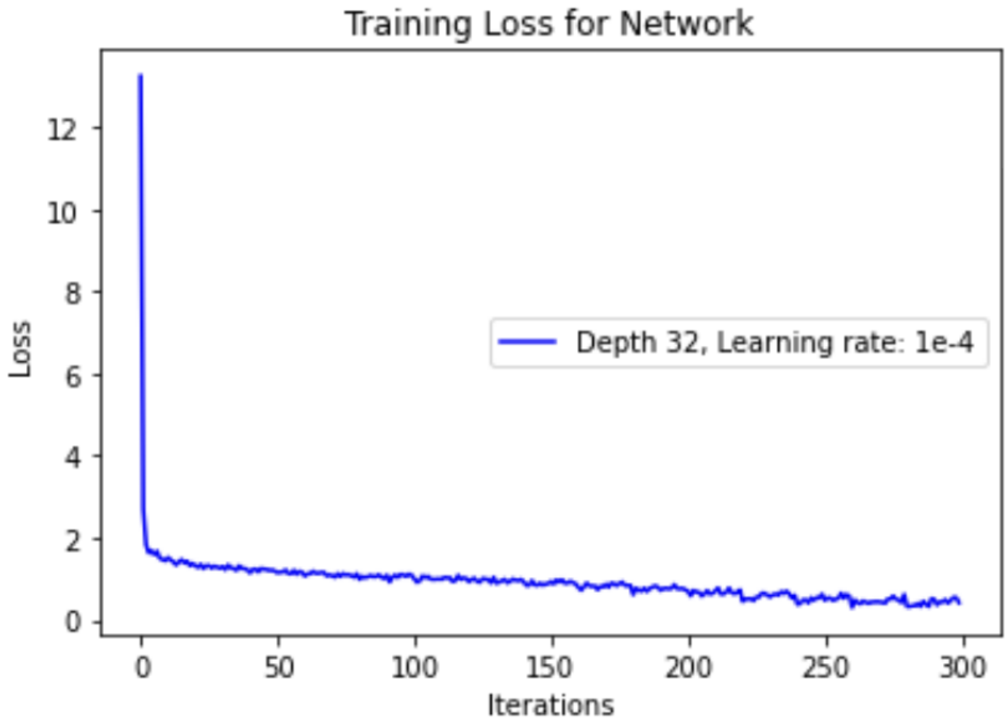
As can be seen from below confusion matrix the accuracy for the network is : 53.1%



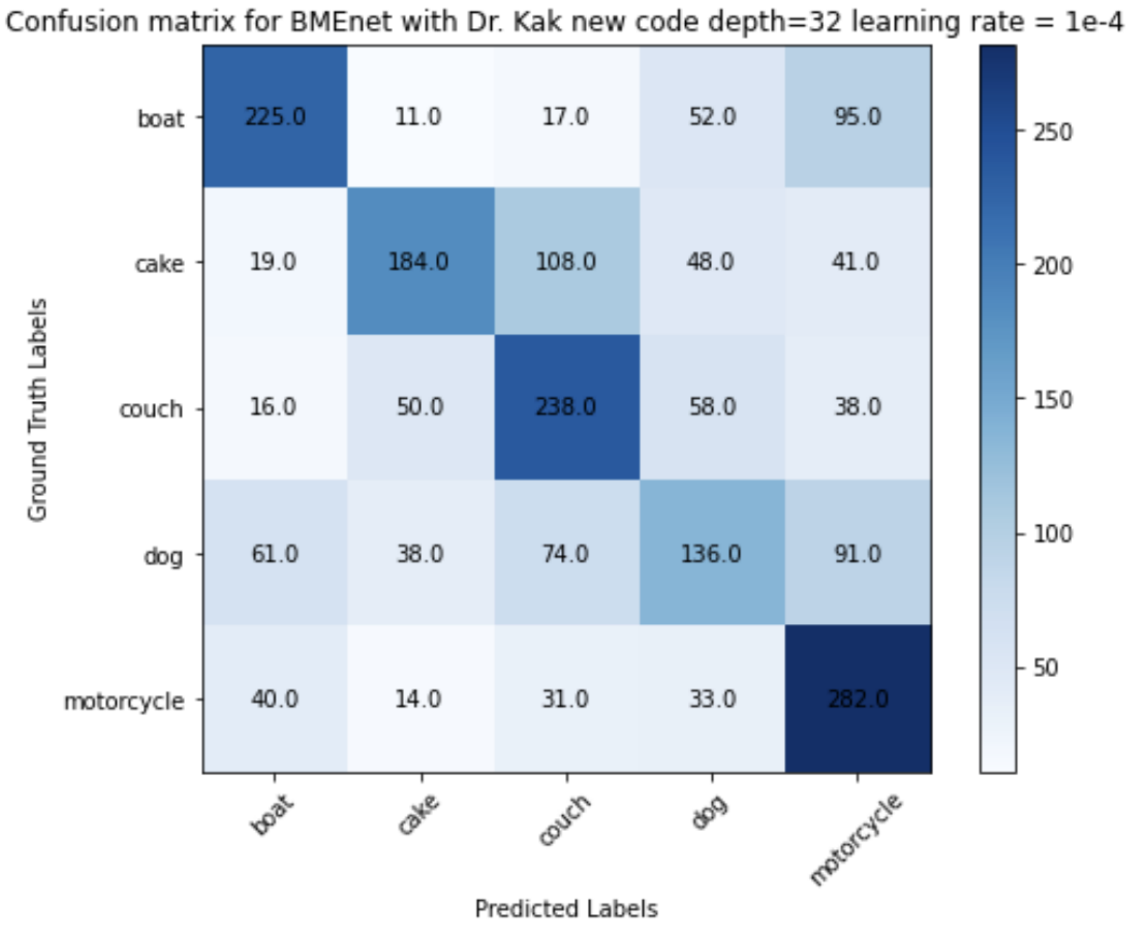
Training loss and accuracy metrics for two different learning rates for Skip block Depth = 32 using the last code provided by Professor Kak over email with the new modifications for skip block :

As seen in the below loss curve and confusion matrix the accuracy achieved was pretty much similar to the previous Network structure.

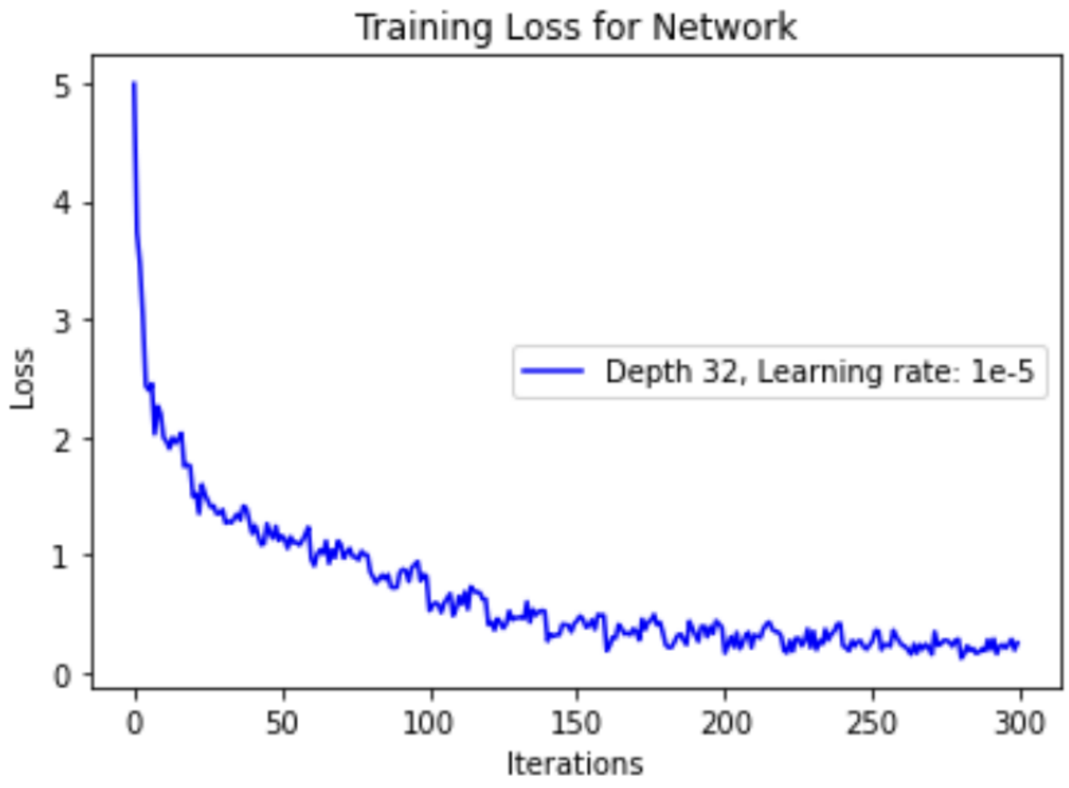
Learning rate = 1e-4:



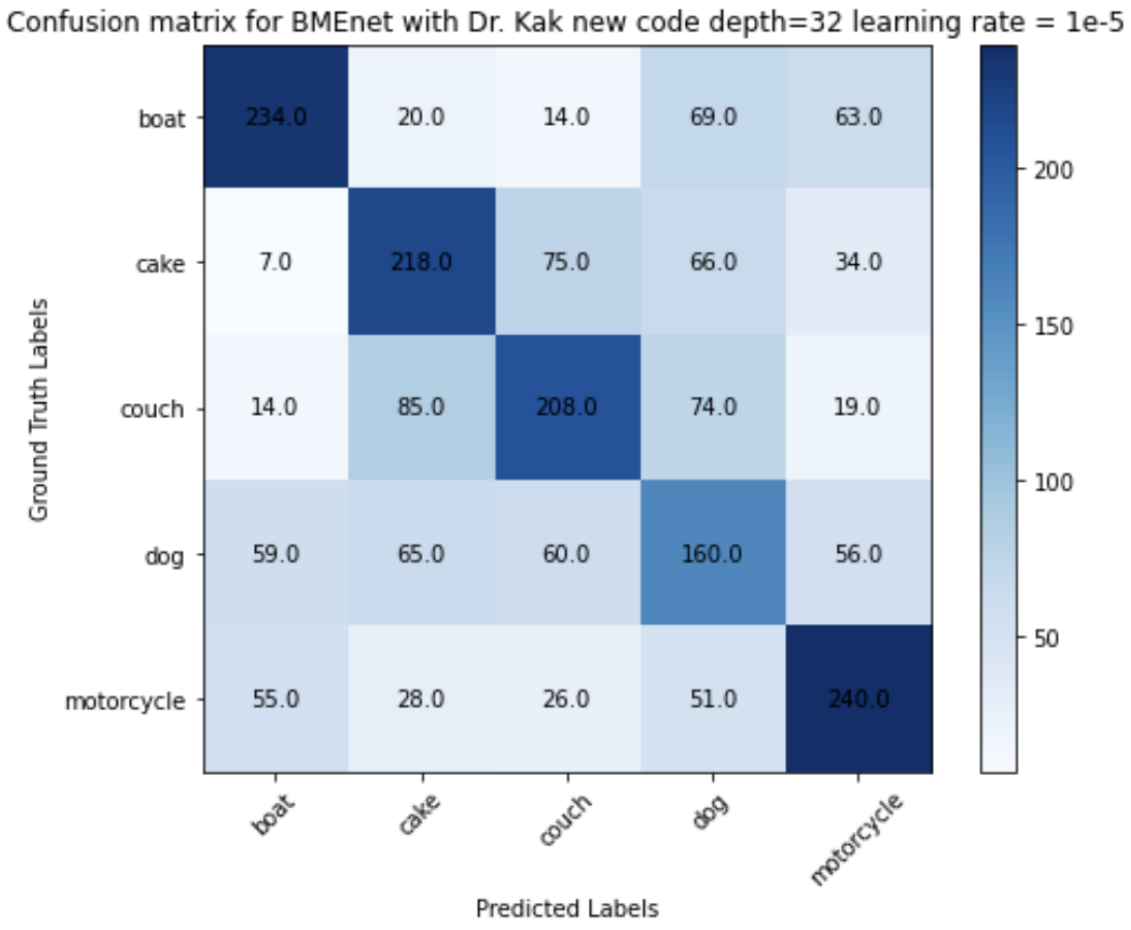
As can be seen from below confusion matrix the accuracy for the network is : 53.25%



Learning rate = 1e-5:

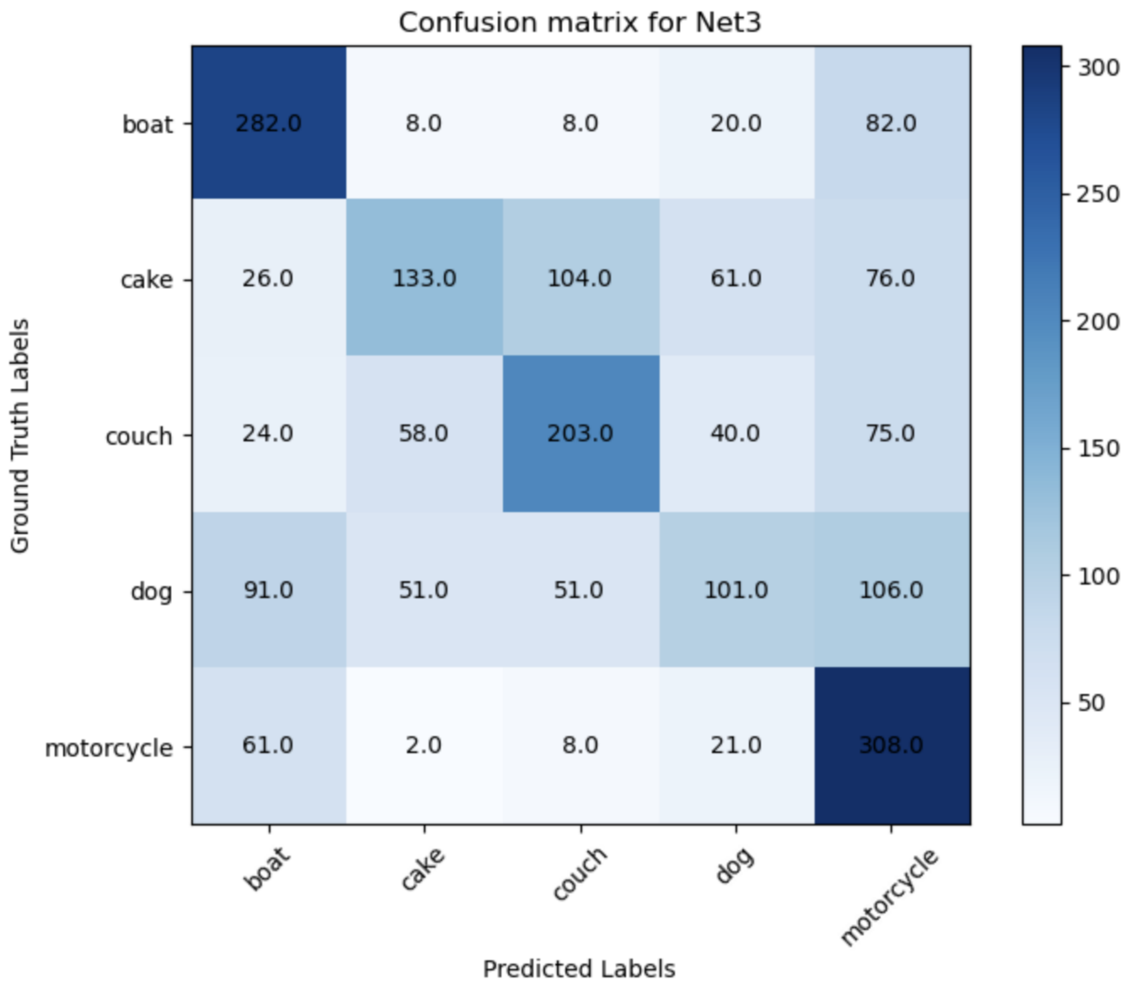


As can be seen from below confusion matrix the accuracy for the network is : 53%



Observations regarding the classification performance of
HWNet5 in comparison with Net3 from Homework 4:

As seen in the below confusion matrix from homework 4 Net3, the accuracy for the Net3 network was 51.35% before skip blocks were used (This was only achieved a few times, after running Net3 for a couple of times the problem of vanishing gradient descent arose and the accuracy further dropped to just 20%).



I used the skip blocks for the class HWNet5 class and as can be seen in above attached images the classification accuracy increased by about 2-4% to 53-54.8% (when experimented with different learning rates and number of layers). However, that isnt a considerable amount of improvement, I believe hyperparameter tuning and weight initialization can help improve the performance of this network more.

Code:

```
#import libraries that are used in the code below
```

```
import os
import torch
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
import torch.nn.functional as F
from torchvision import transforms
import torchvision.transforms as tvt
from PIL import Image
import numpy as np
import glob
import torch
import torch.nn as nn
import torch.nn.functional as F
import seaborn as sns
import matplotlib.pyplot as plt
import shutil
import random
import time
import numpy
import cv2
from pycocotools.coco import COCO
```

```
#Custom dataset class for image classification using COCO sub
set dataset same as homework 4
class MyCocoDataset(Dataset):
    def __init__(self, root):
        super().__init__()
        self.root = root
        self.file_names = []  #storing filename paths for ima
ges and associated labels
        self.classes = ['boat', 'cake', 'couch', 'dog', 'moto
rcycle']
        self.transform = tvt.Compose([
            tvt.ToTensor(),
            tvt.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
        ])
        for label in self.classes:
            folder_path = os.path.join(root, label)
            for image_file in os.listdir(folder_path):
                info = {'image_path': os.path.join(folder_pat
h, image_file), 'label': self.classes.index(label)} #creating
a dictionary to store image path and label for every image
                self.file_names.append(info) #creating a list
of dictionary

    def __len__(self):
        return len(self.file_names)

    def __getitem__(self, index):
        file_name = self.file_names[index]
        image_path = file_name['image_path']

        # Reading image as a PIL image
        image = Image.open(image_path)

        # Applying predefined transform
        augmented_image = self.transform(image)

        # Setting label
        label = file_name['label']

        return augmented_image, label
```

```
#creating instances for train and validation dataset classes,
using the same dataset from homework 4
trainDataset = MyCocoDataset('/Users/skose/Downloads/CocoSubs
et/train')
valDataset = MyCocoDataset('/Users/skose/Downloads/CocoSubse
t/val')
```

```
#creating instances of Train and validation dataloader
trainDataLoader = DataLoader(trainDataset, batch_size = 4,shu
ffle=True)
valDataLoader = DataLoader(valDataset, batch_size = 4,shuffle
=True)
```

```
# The code is derived from Dr. Avinash Kak's DL Studio Module
for the training loop and is same as my homework 4 training r
outine
def training_routine(net, epochs, trainDataLoader, device, mo
del_path):
    # Initialize a list to store training losses during each
epoch
    training_loss = []
    # Define the loss function (CrossEntropyLoss) and optimiz
er (Adam)
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(net.parameters(), lr=1e-5, b
etas=(0.9, 0.99))

    # Move the network to the specified device (CPU or GPU)
    net = net.to(device)

    # Set the network in training mode
    net.train()

    print("Begin Training...\n")

    # Training loop over epochs
    for epoch in range(epochs):
        running_loss = 0.0

        # Iterate over batches in the training data loader
        for i, data in enumerate(trainDataLoader):
            inputs, labels = data
            inputs = inputs.to(device)
            labels = labels.to(device)

            # Zero the gradients to accumulate new gradients
            optimizer.zero_grad()

            # Forward pass: compute predicted outputs by pass
ing inputs through the network
            outputs = net(inputs)

            # Compute the loss
            loss = criterion(outputs, labels)

            # Backward pass: compute gradient of the loss wit
h respect to model parameters
            loss.backward()

            # Update the parameters (weights) of the model
            optimizer.step()

            # Update running loss
            running_loss += loss.item()

            # Print running loss every 100 batches
            if (i + 1) % 100 == 0:
                print("[epoch: %d, batch: %5d] loss: %.3f" %
(epoch + 1, i + 1, running_loss / 100))
                training_loss.append(running_loss / 100)
                running_loss = 0.0

    print("Finished Training!\n")
    # Save the trained model state dictionary to the specifie
d path
    torch.save(net.state_dict(), model_path)

    # Return the list of training losses to plot the loss cur
ve
    return training_loss
```

```
#This testing loop derives ideas from the previous year's hom
ework 4 assignment at https://engineering.purdue.edu/DeepLear
n/2_best_solutions/2023/Homeworks/HW4/2BestSolutions/1.pdf
def model_evaluation(net, valDataLoader, batch_size, device,
desired_classes, model_path):
    # The part of code is derived from Dr. Avinash Kak's DL S
tudio Module for testing with small changes in it.
    print("Begin...\n")

    # Loading trained model state
    net.load_state_dict(torch.load(model_path))

    # Set the network in evaluation mode
    net.eval()

    # Initialize variables for accuracy calculation
    correct, total = 0, 0

    # Initialize a confusion matrix
    confusion_matrix = torch.zeros(5, 5)

    # Initialize variables for per-class accuracy calculation
    class_correct = [0] * 5
    class_total = [0] * 5
    with torch.no_grad():
        # Iterate over batches in the testing data loader
        for i, data in enumerate(valDataLoader):
            inputs, labels = data
            inputs = inputs.to(device)
            labels = labels.to(device)

            # Forward pass: compute predicted outputs by pass
ing inputs through the network
            output = net(inputs)

            # Find the predicted class (index with maximum pr
obability)
            _, predicted = torch.max(output.data, 1)

            # Update total and correct predictions
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

            # Update confusion matrix
            comp = predicted == labels
            for label, prediction in zip(labels, predicted):
                confusion_matrix[label][prediction] += 1

            # Update per-class accuracy
            for j in range(batch_size):
                label = labels[j]
                class_correct[label] += comp[j].item()
                class_total[label] += 1
    confusion_matrix_np = confusion_matrix.cpu().numpy()
    # Print per-class accuracy
    for j in range(5):
        print('Prediction accuracy for %5s : %2d %%' % (desir
ed_classes[j], 100 * class_correct[j] / class_total[j]))

    print("Finished!\n")
    # Print overall accuracy
    print('Accuracy of the network on validation images:
{}%'.format(100 * float(correct / total)))

    # Return confusion matrix and overall accuracy
    return confusion_matrix_np, float(correct / total
```

```
#set the device value to the currently used device
if torch.backends.mps.is_available() == True:
 device = torch.device("mps")
else:
 device = torch.device("cpu")
# Print out currently using device
print("Currently using: {}".format(device))
```

```
#function to plot the training loss curve for the network (th
e depth and learning rate values were changed each time while
plotting the graphs for every network)
def train_loss_plot(train_loss, title):
    plt.plot(train_loss[0], 'b', label='Depth 32, Learning ra
te: 1e-5')
    plt.title(title)
    plt.legend(loc="center right")
    plt.xlabel("Iterations")
    plt.ylabel("Loss")
    plt.show()
```

```
#function to plot the confusion matrix for the network (the d
epth and learning rate values were changed each time while pl
otting the graphs for every network)
def plot_confusion_matrix(confusion_matrix, title):
    plt.figure(figsize=(8, 6))
    plt.imshow(confusion_matrix, interpolation='nearest', cma
p=plt.cm.Blues)
    plt.title(title)
    plt.colorbar()
    class_names = ['boat', 'cake', 'couch', 'dog', 'motorcycl
e']
    # Set ticks and labels
    tick_marks = np.arange(len(class_names))
    plt.xticks(tick_marks, class_names, rotation=45)
    plt.yticks(tick_marks, class_names)

    # Display values in the matrix
    for i in range(len(class_names)):
        for j in range(len(class_names)):
            plt.text(j, i, str(confusion_matrix[i, j]), ha='c
enter', va='center', color='black')

    plt.ylabel('Ground Truth Labels')
    plt.xlabel('Predicted Labels')
    plt.show()
```

```
#This code is borrowed directly from the DL studio module cod
e provided by Professor Kak for skip connection class using s
kip block and BMEnet class
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(out_ch, out_ch, 3, stride=1,
padding=1)
        self.bn1 = nn.BatchNorm2d(out_ch)
        self.bn2 = nn.BatchNorm2d(out_ch)
        if downsample:
            self.downsampler = nn.Conv2d(in_ch, out_ch, 1, st
ride=2)

    def forward(self, x):
        identity = x
        out = self.convo1(x)
        out = self.bn1(out)
        out = nn.functional.relu(out)
        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 = out + identity
            else:
                out = out + torch.cat((identity, identity), d
im=1)
        return out

class HW5Net(nn.Module):
    def __init__(self, skip_connections=True, depth=32):
        super().__init__()
        if depth not in [8, 16, 32, 64]:
            sys.exit("BMEnet has been tested for depth for on
ly 8, 16, 32, and 64")
        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_con
nections=skip_connections))
        self.skip64ds = SkipBlock(64, 64, downsample=True, sk
ip_connections=skip_connections)
        self.skip64to128 = SkipBlock(64, 128, skip_connection
s=skip_connections)
        self.skip128_arr = nn.ModuleList()
        for i in range(self.depth):
            self.skip128_arr.append(SkipBlock(128, 128, skip_
connections=skip_connections))
        self.skip128ds = SkipBlock(128, 128, downsample=True,
skip_connections=skip_connections)

        # The following declaration is predicated on the assu
mption that the number of
        # output nodes (CxHxW) from the final convo layer is
exactly 2048 for each
        # input image. Depending on the size of the input ima
ge, this places a constraint
        # on how many downsampling instances of SkipBlock you
can call in a network.
        self.fc1 = nn.Linear(8192, 1000) # input size = 8192
for
        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.skip64ds(x)
        for i, skip64 in enumerate(self.skip64_arr[self.dept
h//4:]):
            x = skip64(x)
        x = self.skip64ds(x)
        x = self.skip64to128(x)
        for i, skip128 in enumerate(self.skip128_arr[:self.de
pth//4]):
            x = skip128(x)
        for i, skip128 in enumerate(self.skip128_arr[self.dep
th//4:]):
            x = skip128(x)
        # See the comment block above the "self.fc1" declarat
ion in the constructor code.
        x = x.view(x.shape[0], - 1)
        x = nn.functional.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

```
#run the training loop for depth =32 and learning rate = 1e-3
and save the training loss for plotting later
net= HW5Net(skip_connections=True, depth=32)
epochs_net = 15
model_path=os.path.join('/Users/skose/Downloads/path', 'nethw
5.pth')
training_loss_net = training_routine(net, epochs_net, trainDa
taLoader, device, model_path)
```

```
desired_classes=['boat', 'cake', 'couch', 'dog', 'motorcycl
e']
```

```


#run the evaluation on validation dataset for depth =32 and l
earning rate = 1e-3 and save the validation accuracy and conf
usion matrix
batch_size=4
confusion_matrix_net, validation_acc_net = model_evaluation(n
et, valDataLoader, batch_size, device, desired_classes,model_
path)
#Plot confusion matrix  for depth =32 and learning rate = 1e-
3
title= 'Confusion matrix for Hw5Net depth = 32 and learning r
ate = 1e-3'
plot_confusion_matrix(confusion_matrix_net, title)
# training loss for depth =32 and learning rate = 1e-3
train_loss = []
train_loss.append(training_loss_net)
train_loss_plot(train_loss, 'Training Loss for Network')
```

```
#Total number of layers for the network
num_layers = len( list ( net . parameters () ) )
num_layers
```

```
#run the training loop  for depth =32 and learning rate = 1e-
4 and save the training loss for plotting later
net= HW5Net(skip_connections=True, depth=32)
epochs_net = 15
model_path=os.path.join('/Users/skose/Downloads/path', 'nethw
55.pth')
training_loss_net2 = training_routine(net, epochs_net, trainD
ataLoader, device, model_path)
```

```
#run the evaluation on validation dataset for depth =32 and l
earning rate = 1e-4 and save the validation accuracy and conf
```

```
usion matrix
batch_size=4
confusion_matrix_net2, validation_acc_net2 = model_evaluation
(net, valDataLoader, batch_size, device, desired_classes,mode
l_path)
#Plot confusion matrix  for depth =32 and learning rate = 1e-
4
title= 'Confusion matrix for Hw5Net depth = 32 and learning r
ate = 1e-4'
plot_confusion_matrix(confusion_matrix_net2, title)
#train loss for depth = 32 and learning rate = 1e-4
train_loss = []
train_loss.append(training_loss_net2)
train_loss_plot(train_loss, 'Training Loss for Network')





#run the training loop  for depth =64 and learning rate = 1e-
4 and save the training loss for plotting later
net= HW5Net(skip_connections=True, depth=64)
model_path=os.path.join('/Users/skose/Downloads/path', 'nethw
53.pth')
training_loss_net3 = training_routine(net, epochs_net, trainD
ataLoader, device, model_path)





#run the evaluation on validation dataset  for depth =64 and
learning rate = 1e-4 and save the validation accuracy and con
fusion matrix
batch_size=4
confusion_matrix_net3, validation_acc_net3 = model_evaluation
```

```
(net, valDataLoader, batch_size, device, desired_classes,mode
l_path)
#Plot confusion matrix  for depth =64 and learning rate = 1e-
4
title= 'Confusion matrix for Hw5Net depth = 64 and learning r
ate = 1e-4'
plot_confusion_matrix(confusion_matrix_net2, title)
#train loss for depth 64 and learning rate 1e-4
train_loss = []
train_loss.append(training_loss_net3)
train_loss_plot(train_loss, 'Training Loss for Network')
```

```
#This code is borrowed directly from the last code provided b
y Professor Kak over email with the new modifications for ski
p block
#Tried to run this code for different learning rates in order
to just compare whether the performances improves as compared
#to my previous implementation

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, in_ch, 3, stride=1, pa
dding=1)
        self.convo2 = nn.Conv2d(in_ch, out_ch, 3, stride=1, p
adding=1)
        self.bn1 = nn.BatchNorm2d(in_ch)
        self.bn2 = nn.BatchNorm2d(out_ch)
        self.in2out = nn.Conv2d(in_ch, out_ch, 1)
        if downsample:
            self.downsampler1 = nn.Conv2d(in_ch, in_ch, 1, st
ride=2)
            self.downsampler2 = nn.Conv2d(out_ch, out_ch, 1,
stride=2)

    def forward(self, x):
        identity = x
        out = self.convo1(x)
        out = self.bn1(out)
        out = F.relu(out)
        out = self.convo2(out)
        out = self.bn2(out)
        out = F.relu(out)

        if self.downsample:
            identity = self.downsampler1(identity)
            out = self.downsampler2(out)

        if self.skip_connections:
            if (self.in_ch == self.out_ch) and (not self.down
sample):
                out = out + identity
            elif (self.in_ch != self.out_ch) and (not self.do
wnsample):
                identity = self.in2out(identity)
                out = out + identity
            elif (self.in_ch != self.out_ch) and self.downsam
ple:
                out = out + torch.cat((identity, identity), d
im=1)

        return out

class BMEnet(nn.Module):
    """
    Class Path: DLStudio -> SkipConnections -> BMENet
    """
    def __init__(self, skip_connections=True, depth=8):
        super(BMEnet, self).__init__()
        self.depth = depth
        self.conv = nn.Conv2d(3, 64, 3, padding=1)
        self.skip64_arr = nn.ModuleList([SkipBlock(64, 64, sk
ip_connections=skip_connections) for _ in range(depth)])
        self.skip64to128ds = SkipBlock(64, 128, downsample=Tr
ue, skip_connections=skip_connections)
        self.skip128_arr = nn.ModuleList([SkipBlock(128, 128,
skip_connections=skip_connections) for _ in range(depth)])
        self.skip128to256ds = SkipBlock(128, 256, downsample=
True, skip_connections=skip_connections)
        self.skip256_arr = nn.ModuleList([SkipBlock(256, 256,
skip_connections=skip_connections) for _ in range(depth)])
        self.fc1 = nn.Linear(16384, 1000)
        self.fc2 = nn.Linear(1000, 5)

    def forward(self, x):
        x = F.max_pool2d(F.relu(self.conv(x)), 2, 2)


        for skip64 in self.skip64_arr:
            x = skip64(x)
        x = self.skip64to128ds(x)

        for skip128 in self.skip128_arr:
            x = skip128(x)
        x = self.skip128to256ds(x)

        for skip256 in self.skip256_arr:
            x = skip256(x)

        x = x.view(x.shape[0], -1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x





#run the training loop for the borrowed BMEnet code from Dr.
Kak's email for depth 32 and learning rate 1e-5
#and save the training loss for plotting later
netNew= BMEnet(skip_connections=True, depth=32)
epochs_net = 15
model_path6=os.path.join('/Users/skose/Downloads/path', 'neth
w6.pth')
training_loss_net6 = training_routine(netNew, epochs_net, tra
inDataLoader, device, model_path6)




#run the evaluation on validation dataset for the borrowed BM
Enet code from Dr. Kak's email for depth 32 and learning rate
1e-5
#and save the validation accuracy and confusion matrix
batch_size=4
confusion_matrix_net6, validation_acc_net6 = model_evaluation
(netNew, valDataLoader, batch_size, device, desired_classes,m
odel_path6)
#Plot confusion matrix
title= 'Confusion matrix for BMEnet with Dr. Kak new code'
plot_confusion_matrix(confusion_matrix_net6, title)
#training loss for the borrowed BMEnet code from Dr. Kak's em
ail for depth 32 and learning rate 1e-5
train_loss = []
train_loss.append(training_loss_net6)
train_loss_plot(train_loss, 'Training Loss for Network')
```

```
#run the training loop for the borrowed BMEnet code from Dr.
Kak's email for depth 32 and learning rate 1e-4
#and save the training loss for plotting later
netNew1= BMEnet(skip_connections=True, depth=32)
epochs_net = 15
model_path7=os.path.join('/home/skose', 'nethw7.pth')
training_loss_net7 = training_routine(netNew1, epochs_net, tr
ainDataLoader, device, model_path7)



#run the evaluation on validation dataset for the borrowed BM
Enet code from Dr. Kak's email for depth 32 and learning rate
1e-4 and save the validation accuracy and confusion matrix
batch_size=4
confusion_matrix_net7, validation_acc_net7 = model_evaluation
(netNew1, valDataLoader, batch_size, device, desired_classes,
model_path7)
#Plot confusion matrix
title= 'Confusion matrix for BMEnet with Dr. Kak new code'
plot_confusion_matrix(confusion_matrix_net7, title)
#training loss for the borrowed BMEnet code from Dr. Kak's em
ail for depth 32 and learning rate 1e-4
train_loss = []
train_loss.append(training_loss_net7)
train_loss_plot(train_loss, 'Training Loss for Network')
```