ECE60146: Homework 5

Manish Kumar Krishne Gowda, 0033682812

(Spring 2024)

1 Introduction

In this assignment, we study and program Skip Connections and Residual Blocks, commonly used in CNNs to address the vanishing gradient problem. This problem occurs in deep neural networks, where gradients weaken as they propagate through many layers, making it difficult for the network to learn effectively.

A residual block consists of several stacked layers (e.g., convolutional layers, activation functions) that learn a residual function. This function represents the difference between the input and the desired output at that point in the network. Instead of learning the entire output directly, the block focuses on learning this residual, which often requires less complex transformations compared to learning the entire output from scratch.

The key element of a residual block is a skip connection. This connection directly adds the input of the block to its output before applying an activation function. This ensures that the information from the input can flow directly through the block, regardless of the complexity of the learned residual function.

Skip connections allow gradients to flow more easily through the network, improving learning in deeper architectures. In programming tasks we realise a custom Network invoolving Residual Blocks and the these tasks help to provide a overview of Residual networks which have consistently achieved state-of-the-art results in various image classification and computer vision tasks.

2 Programming Tasks

In this report, the study focused on the implementation and understanding of two key components within the DLStudio module: SkipBlock and BMEnet, with a particular emphasis on the design improvements introduced by Prof. Avinash Kak.

2.1 SkipBlock Implementation :

The SkipBlock was studied, and a new implementation was adopted, incorporating a cleaner design provided by Prof. Avinash Kak. The design philosophy behind this new SkipBlock revolves around the concept that when downsampling the height and width (H, W) axes of the input tensor, the number of output channels must be simultaneously doubled. This ensures a consistent and efficient handling of spatial dimensions.

2.2 BMEnet Class Utilization :

A modified BMEnet class (called MyNet), with its depth set to 16, was utilized in the study. This class follows a more transparent architecture. After the initial convolutional layer, which transforms the input 3-channel image into a 64-channel output tensor, the MyNet class incorporates a series of SkipBlocks.

Specifically, the structure includes 16 64-channel SkipBlocks, followed by a 64-to-128 channel converter SkipBlock. This converter not only increases the number of channels but also downsamples the input by a factor of 2. Subsequently, another converter is employed to transition from 128 to 256 channels, further downsampling the input by a factor of 2.

At the end of the convolutional chain in MyNet, a 4x4 image (for a 32x32 input) is obtained, comprising 256 channels. This implies that, at this stage, all the inter-class discriminatory information has been consolidated into the channel dimension. The thoughtful design of the MyNet class reflects a strategic approach to information representation and extraction through the convolutional layers, ultimately contributing to the network's effectiveness in capturing relevant features and patterns.

In this study, the implementation leveraged existing components such as SkipBlock and MyNet from the DLStudio module. The remaining code base, including the training and evaluation pipelines, retains the structures and methodologies that were used for HW4.

3 Layers and Learnable Parameters

Mynet contains a total of 514 layers and 31006850 learnable parameters.

The actual learnable parameters' calculation (i.e. those contributing to the forward layer) is given in Figure 1

4 Outputs and Discussion

Learning Rate	Training Loss (20 epochs)	Validation Loss	Accuracy
0.001	0.435	1.5520	54.52%
0.0001	0.101	2.3992	49.00%

Table 1: Performance Metrics for Different Learning Rates

Two learning rates, 0.001 and 0.0001, were employed during the training process. For a learning rate of 0.001, the training loss at the end of 20 epochs reached 0.435, the validation loss was 1.5520, and the accuracy achieved was 54.52%. Conversely, when using a learning rate of 0.0001, the training loss at the same epoch was reduced to 0.101. However, this decrease in training loss did not translate to improved validation performance, as the validation loss increased to 2.3992, resulting in an accuracy of 49.00%. These results highlight the sensitivity of the model's performance to the choice of learning rate, demonstrating the need for careful tuning to achieve optimal training outcomes. The details are summarized in Table 1.

The confusion matrix for MyNet with lr=0.001 and lr=0.0001 are reported in Figures 2 and 3 respectively. Further, as indicated in task instructions, the confusion matrix for Net3 of HW4 is also shown in Figure 4. The training Loss curves for MyNet training with lr=0.001 and lr=0.0001 are shown in Figure 5.

In the evaluation of model architectures, a comparative analysis was conducted between Net3 from HW4 and MyNet, shedding light on their respective complexities, parameter counts, and performance metrics.

Net3 from HW4, characterized by a simpler structure comprising 16 layers (Conv2d -¿ MaxPool -¿ Conv2d -¿ MaxPool -¿ 10 Conv2d -¿ Linear -¿ Linear), demonstrated modest model complexity. With a parameter count of 622,245 (approximately 0.6 million), Net3 achieved an accuracy of 56.32%. This accuracy was achieved with a relatively compact model design, highlighting its efficiency.

On the other hand, MyNet, inspite of skip connections and residual blocks, exhibited a more complex architecture with 514 layers and a substantially larger parameter count of 31,006,850 (approximately 31 million). Despite the increased model complexity, the accuracy achieved by MyNet was 54.52%. Notably, the accuracy did not witness improvement; in fact, there was a decrease in performance!

The decrease in accuracy observed in MyNet can be attributed to several factors, notably the heightened complexity of the model, potentially leading to challenges associated with overfitting. This concern is particularly pronounced given the limited dataset size, which may not be sufficient to effectively leverage the increased capacity of MyNet.

This overfitting hypothesis gains support from the training and accuracy results of MyNet when trained with a learning rate (lr) of 0.0001. A smaller learning rate often implies more meticulous model training, resulting in lower training loss. In the case of lr=0.0001, the training loss achieved a notably lower value of 0.1 compared to 0.4 with lr=0.001. However, paradoxically, the accuracy decreased.

This discrepancy suggests that, despite achieving a lower training loss with a smaller learning rate, the model's generalization to the validation set diminished, indicating an overfitting scenario. The heightened model complexity, combined with a relatively constrained dataset, may lead to MyNet capturing intricate patterns within the training data that do not necessarily generalize well to unseen instances. Therefore, the observed decrease in accuracy can be reasonably attributed to the model's susceptibility to overfitting under the given conditions.

Furthermore, the introduction of skip connections and residual blocks might not have yielded the anticipated benefits for the given task. The benefits of skip connections and residual blocks are often more pronounced in deeper networks or complex tasks. If the task at hand is relatively simple or the dataset size is limited, the added complexity introduced by these architectural features may not contribute significantly to improved performance. The increased learning capacity of the model due to skip connections and residual blocks might have lead to overfitting, especially since the dataset is not large enough to support the enhanced model capacity.

Among the various factors investigated as potential contributors to the suboptimal performance of general SkipConnection networks, gradient explosion could be a possible reason. Gradient explosion is a phenomenon that can occur during the training of deep neural networks, particularly during backpropagation. It is the opposite of the more commonly discussed vanishing gradient problem. Gradient explosion occurs when the gradients become extremely large, causing the updates to the model parameters to become excessively large as well. This can lead to numeric instability and erratic training behavior. However the decrease in performance in MyNet may not be attributed to the phenomenon called gradient explosion, since the loss is steadily decreasing for both learning rates.

An interesting observation pertains to the consistent underperformance of the "dog" class across both Net3 and MyNet. This trend could be attributed to the shared visual context between dogs and other objects, such as cakes and couches, which often coexist within a home setting. The model might face challenges in distinguishing between these visually similar elements, contributing to the lower accuracy for the "dog" class.

```
# -*- coding: utf-8 -*-
"""HW5.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1r4CiraRsUJwN9pHq8p_o70dXEQBOaBW2
"""
```

```
from google.colab import drive
drive.mount('/content/drive')

# Commented out IPython magic to ensure Python compatibility.
# %cd /content/drive/My Drive/hw4_data/
# %cd DLStudio-2.2.3
!pip install pymsgbox
!python setup.py install

import os
import torch
import random
import numpy as np
import requests
import matplotlib.pyplot as plt

from tqdm import tqdm
from PIL import Image
seed = 60146
random.seed(seed)
np.random.seed(seed)

from DLStudio import *

classes = ['boat', 'cake', 'couch', 'dog', 'motorcycle']
train_imgs_per_class = 1500
val_imgs_per_class = 500

import os
import torch

# Custom dataset class for COCO
class CocoDataset(torch.utils.data.Dataset):
  def __init__(self, root, transforms=None):
    super().__init__()
    self.root_dir = root
    self.classes = os.listdir(self.root_dir)
    self.transforms = transforms
    self.img_paths = []
    self.img_labels = []
    self.class_to_idx = {'boat':0, 'cake':1, 'couch':2, 'dog':3, 'motorcycle': 4}
    self.idx_to_class = {i:c for c, i in self.class_to_idx.items()}

    for cls in self.classes:
      cls_dir = os.path.join(self.root_dir, cls)
      paths = os.listdir(cls_dir)
      self.img_paths+= [os.path.join(cls_dir, path) for path in paths]
      self.img_labels+=[self.class_to_idx[cls]]*len(paths)

  def __len__(self):
    # Return the total number of images
    return len(self.img_paths)

  def __getitem__(self, index):
    index = index % len(self.img_paths)
    img_path = self.img_paths[index]
    img_label = self.img_labels[index]
    img = Image.open(img_path)
    img_transformed = self.transforms(img)
    return img_transformed, img_label

import torchvision.transforms as tvt
import torch.utils.data

# Define image transformations
reshape_size = 32 #unlike HW4 a 32x32 image is considered
transforms = tvt.Compose([
    tvt.ToTensor(),  # Convert images to PyTorch tensors
    tvt.Resize((reshape_size, reshape_size)),  # Resize to 64x64
    tvt.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))  # Normalize for better
                                              training
])

# Assuming CocoDataset is a custom class you've defined
train_dataset = CocoDataset('/content/drive/MyDrive/hw4_data/coco/train2017/',
                                          transforms=transforms)
val_dataset = CocoDataset('/content/drive/MyDrive/hw4_data/coco/val2017/',
                                          transforms=transforms)

# Create dataloaders with appropriate settings
train_data_loader = torch.utils.data.DataLoader(train_dataset,
                                                 batch_size=128,
                                                 shuffle=True,  # Shuffle for
                                                                                    train

                                                 num_workers=2)  # Use multiple
                                                                                    worke

                                                                                    for

                                                                                    faste

                                                                                    loadi

val_data_loader = torch.utils.data.DataLoader(val_dataset,
                                                batch_size=128,
                                                shuffle=False,  # No shuffling for

                                                                                   valida


                                                num_workers=2)

# Check dataloader output
train_loader_iter = iter(train_data_loader)
img, target = next(train_loader_iter)

print('img has length: ', len(img))  # Print batch size
print('target has length: ', len(target))  # Print batch size (should match img
                                          length)
print(img[0].shape)  # Print shape of a single image (should be torch.Size([3, 64,
                                           64]))

import os

def train_net(device, net, optimizer, criterion, data_loader, model_name,
              epochs=10, display_interval=100):
    """Trains a neural network classifier.
    Args:
        device (torch.device): Device to use for training.
        net (torch.nn.Module): Neural network model to train.
        optimizer (torch.optim.Optimizer): Optimizer to use for training.
        criterion (torch.nn.Module): Loss function to use for training.
        data_loader (torch.utils.data.DataLoader): Data loader for training data.
        model_name (str): Name of the model to save.
        epochs (int, optional): Number of training epochs. Defaults to 10.
        display_interval (int, optional): Interval to display training progress.
                                          Defaults to 100.
    """

    net = net.to(device)  # Move model to device
    loss_running_record = []  # Track loss for visualization

    for epoch in range(epochs):
        running_loss = 0.0

        for i, data in enumerate(data_loader):
            inputs, labels = data
            inputs = inputs.to(device)
            labels = labels.to(device)

            optimizer.zero_grad()  # Clear gradients
            outputs = net(inputs)  # Forward pass
            loss = criterion(outputs, labels)  # Calculate loss
            loss.backward()  # Backward pass
            optimizer.step()  # Update model parameters

            running_loss += loss.item()  # Accumulate loss

            if (i + 1) % display_interval == 0:
                avg_loss = running_loss / display_interval
                print(f"[epoch : {epoch + 1}, batch : {i + 1}] loss : {avg_loss:.
                                                       3f}")
                loss_running_record.append(avg_loss)
                running_loss = 0.0  # Reset running loss

    # Save model checkpoint
    checkpoint_path = os.path.join('/content/drive/MyDrive/hw5_data/saved_models',
                                   f'{model_name}.pth')
    torch.save(net.state_dict(), checkpoint_path)

    return loss_running_record

import matplotlib.pyplot as plt

def plot_loss(loss, display_interval, model_name):
    """Plots the training loss curve.

    Args:
        loss (list): List of loss values recorded during training.
        display_interval (int): Interval at which loss values were recorded.
        model_name (str): Name of the model being trained.
    """

    iterations = np.arange(len(loss)) * display_interval  # Calculate iterations
    plt.figure(figsize=(8, 6))  # Set plot size
    plt.plot(iterations, loss, label='Training Loss')

    plt.title(f'Training Loss for {model_name}', fontsize=16)  # Set title
    plt.xlabel('Iterations', fontsize=14)  # Set x-axis label
    plt.ylabel('Loss', fontsize=14)  # Set y-axis label

    plt.xlim(0, iterations.max())  # Adjust x-axis limits
    plt.grid(True)  # Add grid lines
    plt.legend()  # Show legend
    plt.tight_layout()  # Adjust layout for better spacing
    plt.show()

def validate_net(device, net, data_loader, model_path=None):
    """Validates a neural network classifier.

    Args:
        device (torch.device): Device to use for validation.
        net (torch.nn.Module): Neural network model to validate.
        data_loader (torch.utils.data.DataLoader): Data loader for validation data
                                                    .
        model_path (str, optional): Path to load model weights from. Defaults to
                                                    None.

    Returns:
        tuple: A tuple containing:
            - imgs (list): List of validation images.
            - all_labels (list): List of ground truth labels.
            - all_pred (list): List of predicted labels.
    """

    if model_path is not None:
        net.load_state_dict(torch.load(model_path))

    net = net.to(device)  # Move model to device
    net.eval()  # Set model to evaluation mode

    running_loss = 0.0
    iters = 0
    imgs = []
    all_labels = []
    all_pred = []

    with torch.no_grad():  # Disable gradient calculation for validation
        for i, data in enumerate(data_loader):
            inputs, labels = data
            inputs = inputs.to(device)
            labels = labels.to(device)

            outputs = net(inputs)  # Forward pass
            loss = criterion(outputs, labels)  # Calculate loss

            running_loss += loss.item()  # Accumulate loss
            iters += 1

            pred_labels = torch.argmax(outputs.data, axis=1)  # Get predicted
                                                    labels
            all_labels.extend(labels.tolist())  # Collect ground truth labels
            all_pred.extend(pred_labels.tolist())  # Collect predicted labels
            imgs.extend(inputs.cpu().detach().numpy())  # Collect images (move to
                                                         CPU and detach)

    avg_loss = running_loss / iters  # Calculate average validation loss
    print(f"Validation Loss: {avg_loss:.4f}")

    return imgs, all_labels, all_pred

def calc_confusion_matrix(num_classes, actual, predicted):
    """Calculates the confusion matrix.

    Args:
        num_classes (int): Number of classes in the dataset.
        actual (list): List of ground truth labels.
        predicted (list): List of predicted labels.

    Returns:
        np.ndarray: The confusion matrix as a NumPy array.
    """

    conf_mat = np.zeros((num_classes, num_classes), dtype=int)  # Initialize with
                                                 integer dtype

    for a, p in zip(actual, predicted):
        conf_mat[a, p] += 1  # Increment corresponding cell using double indexing

    return conf_mat

import seaborn as sns

def plot_conf_mat(conf_mat, classes, model_name):
    """Plots the confusion matrix.

    Args:
        conf_mat (np.ndarray): The confusion matrix to plot.
        classes (list): List of class names.
        model_name (str): Name of the model being evaluated.
    """

    num_classes = len(classes)

    # Calculate normalized counts and percentages for annotations
    labels = np.asarray([
        f"{count}\n{percent:.1f}%" for row in conf_mat for count, percent in zip(
                                                   row, row / np.sum(row) * 100)
    ]).reshape(num_classes, num_classes)

    plt.figure(figsize=(8, 6))  # Adjust figure size for better readability
    sns.heatmap(
        conf_mat,
        annot=labels,
        fmt="",
        cmap="YlOrBr",
        cbar=True,
        xticklabels=classes,
        yticklabels=classes,
        linewidths=0.5,  # Add thin lines between cells for clarity
    )
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.title(f'Confusion Matrix for {model_name}', fontsize=16)  # Increase title
                                            font size
    plt.tight_layout()  # Adjust layout for better spacing
    plt.show()

import torch.nn as nn
class SkipBlock(nn.Module):
  """
  Class Path:   DLStudio  ->  SkipConnections  ->  SkipBlock
  """
  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, padding=1)
      self.convo2 = nn.Conv2d(in_ch, out_ch, 3, stride=1, padding=1)
      self.bn1 = nn.BatchNorm2d(in_ch)
      self.bn2 = nn.BatchNorm2d(out_ch)

      self.in2out  =  nn.Conv2d(in_ch, out_ch, 1)                              ###
                                             <<<<<  from  Cheng-Hao Chen

      if downsample:
          ##  Setting stride to 2 and kernel_size to 1 amounts to retaining every
          ##  other pixel in the image --- which halves the size of the image:
          self.downsampler1 = nn.Conv2d(in_ch, in_ch, 1, stride=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 = nn.functional.relu(out)
      out = self.convo2(out)
      out = self.bn2(out)
      out = nn.functional.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 (self.downsample is False):
              out = out + identity
          elif (self.in_ch != self.out_ch) and (self.downsample is False):

              identity = self.in2out( identity )                               ###
                                                     <<<<  from  Cheng-Hao Chen

              out = out + identity
          elif (self.in_ch != self.out_ch) and (self.downsample is True):
              out = out + torch.cat((identity, identity), dim=1)
      return out

class Mynet(nn.Module):
  """
  Class Path:   DLStudio  ->  SkipConnections  ->  BMENet
```

```
    """
    def __init__(self, skip_connections=True, depth=8):
        super(Mynet, self).__init__()
        self.depth = depth
        self.conv = nn.Conv2d(3, 64, 3, padding=1)
        self.skip64_arr = nn.ModuleList()
        for i in range(self.depth):
            self.skip64_arr.append(SkipBlock(64, 64, skip_connections=
                                                    skip_connections))
        self.skip64to128ds = SkipBlock(64, 128, downsample=True, 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.skip128to256ds = SkipBlock(128, 256, downsample=True, skip_connections=
                                                    skip_connections )
        self.skip256_arr = nn.ModuleList()
        for i in range(self.depth):
            self.skip256_arr.append(SkipBlock(256, 256, skip_connections=
                                                    skip_connections))
        self.fc1 =  nn.Linear(4*4*256, 1000)
        self.fc2 =  nn.Linear(1000, 10)

    def forward(self, x):
        x = nn.MaxPool2d(2,2)(nn.functional.relu(self.conv(x)))
        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 = nn.functional.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# Use GPU if available
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
net16_lr001 = Mynet(skip_connections=True, depth=16).to(device)  # Move model to
                                            device
num_layers = len(list(net16_lr001.parameters()))
print(num_layers)

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net16_lr001.parameters(), lr=1e-3, betas=(0.9, 0.999)
                                            )
# Training
epochs = 20
display_interval = 5
net16_lr001_losses = train_net(device, net16_lr001, optimizer=optimizer, criterion
                                            =criterion,
                        data_loader=train_data_loader, model_name='MyNet16_lr001',
                        epochs=epochs, display_interval=display_interval)

# Validation
save_path = '/content/drive/MyDrive/hw5_data/saved_models/MyNet16_lr001.pth'
```

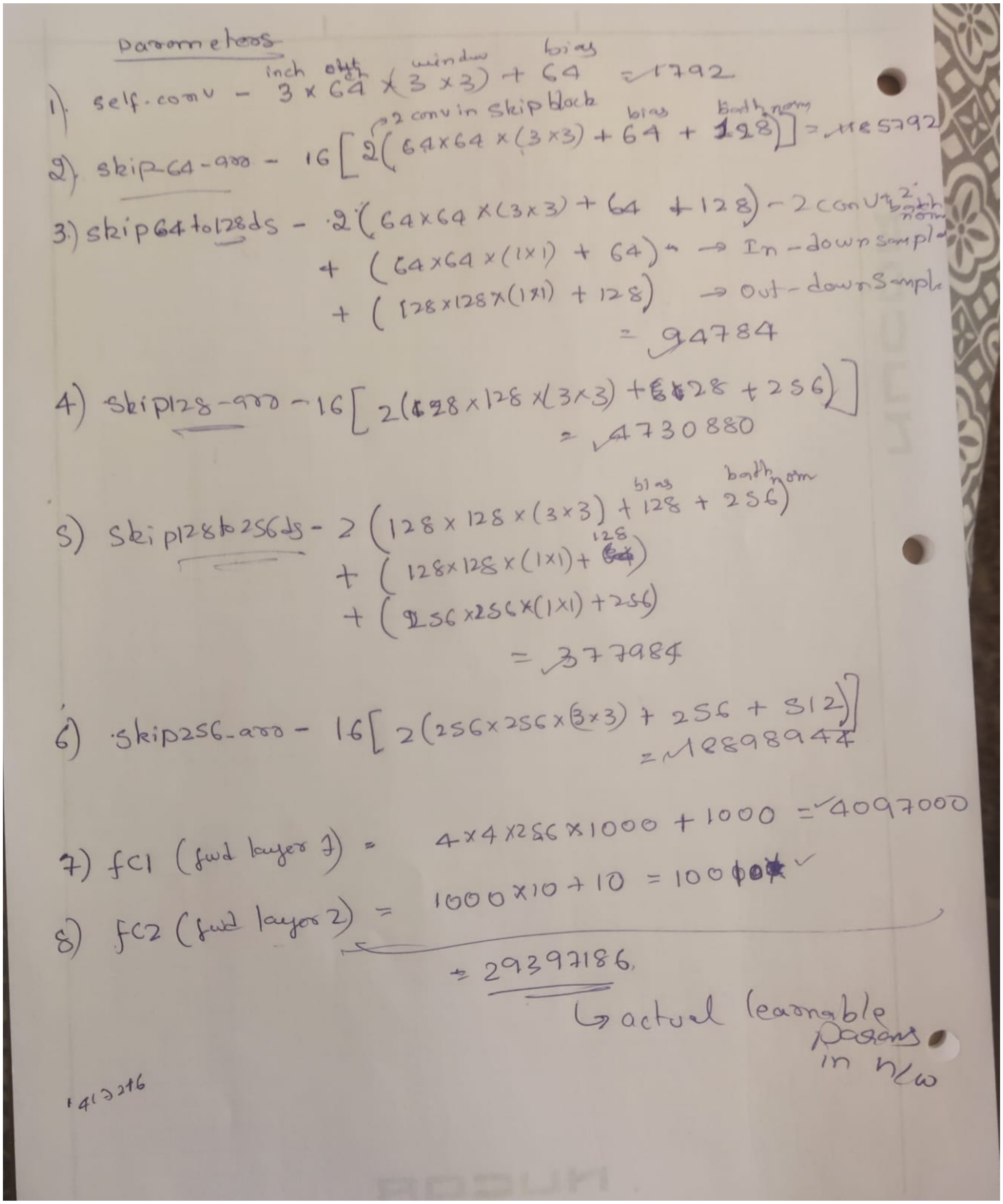
```
imgs, actual, predicted = validate_net(device, net16_lr001, val_data_loader,
                                        model_path=save_path)

# Calculate confusion matrix and accuracy
conf_mat16_lr001 = calc_confusion_matrix(5, actual, predicted)
print(conf_mat16_lr001)
accuracy16_lr001 = np.trace(conf_mat16_lr001) / float(np.sum(conf_mat16_lr001))
print(f"Accuracy: {accuracy16_lr001:.4f}")

# Plot confusion matrix
plot_conf_mat(conf_mat16_lr001, classes, 'MyNet depth=16 lr=0.001')

# Use GPU if available
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
net16_lr0001 = Mynet(skip_connections=True, depth=16).to(device)  # Move model to
                                        device
num_layers = len(list(net16_lr0001.parameters()))
print(num_layers)

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net16_lr0001.parameters(), lr=1e-4, betas=(0.9, 0.999
                                        ))
# Training
epochs = 20
display_interval = 5
net16_lr0001_losses = train_net(device, net16_lr0001, optimizer=optimizer,
                                        criterion=criterion,
                        data_loader=train_data_loader, model_name='MyNet16_lr0001'
                                                              ,
                        epochs=epochs, display_interval=display_interval)

# Validation
save_path = '/content/drive/MyDrive/hw5_data/saved_models/MyNet16_lr0001.pth'
imgs, actual, predicted = validate_net(device, net16_lr0001, val_data_loader,
                                        model_path=save_path)

# Calculate confusion matrix and accuracy
conf_mat16_lr0001 = calc_confusion_matrix(5, actual, predicted)
print(conf_mat16_lr0001)
accuracy16_lr0001 = np.trace(conf_mat16_lr0001) / float(np.sum(conf_mat16_lr0001))
print(f"Accuracy: {accuracy16_lr0001:.4f}")

# Plot confusion matrix
plot_conf_mat(conf_mat16_lr0001, classes, 'MyNet depth=16 lr=0.0001')

# Ensure consistent color scheme across plots
plt.style.use('seaborn')  # Consistent and aesthetically pleasing colors

# Create the plot
plt.figure(figsize=(8, 6))  # Set appropriate figure size
plt.ylim(0, 5) #set the y-axis limit to 5 to ensure that large values don't overly
                                         compress the rest of the graph
x = np.arange(len(net16_lr001_losses)) * display_interval
plt.plot(x, net16_lr001_losses, label='MyNet depth=16 lr=0.001', linewidth=2)  #
                                        Adjust linewidth
plt.plot(x, net16_lr0001_losses, label='MyNet depth=16 lr=0.0001', linewidth=2)

# Customize plot elements
plt.title('Training Loss for Skip Connections', fontsize=16)  # Increase title
```

```
                                          font size
plt.xlabel('Iterations', fontsize=14)  # Set x-axis label
plt.ylabel('Loss', fontsize=14)  # Set y-axis label
plt.xticks(fontsize=12)  # Adjust tick label size
plt.yticks(fontsize=12)
plt.grid(True)  # Add grid lines for readability
plt.legend(fontsize=12)  # Show legend

# Ensure tight layout for optimal spacing
plt.tight_layout()

# Display the plot
plt.show()

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
net16_lr001 = Mynet(skip_connections=True, depth=16).to(device)  # Move model to
                                          device
net16_lr001_params = sum(p.numel() for p in net16_lr001.parameters() if p.
                                          requires_grad)
print(net16_lr001_params)
```

5 Conclusion

Different neural network architectures were explored in this HW, specifically Net3 from HW4 and MyNet with skip connections, has provided valuable insights into their respective performances. While Net3 demonstrated efficiency with a simpler structure and achieved a commendable accuracy of 56.32%, MyNet's increased complexity and incorporation of skip connections did not yield the expected improvements, resulting in a slightly lower accuracy of 54.52%.

Several factors, including overfitting due to heightened model complexity and inadequate dataset size may have contributed to the observed discrepancies in performance. Additionally, the use of skip connections and residual blocks did not necessarily translate into a performance boost, emphasizing the importance of aligning architectural choices with the specific characteristics of the task at hand.



Parameters

1) self-conv - $\overset{\text{inch}}{3} \times \overset{\text{outch}}{64} \times \overset{\text{window}}{(3\times3)} + \overset{\text{bias}}{64}$ = 1792

2) skip-64-arr - $16\left[\overset{\text{2 conv in skip block}}{2}\left(64\times64\times(3\times3) + \overset{\text{bias}}{64} + \overset{\text{batch norm}}{128}\right)\right]$ = 1185792

3) skip64to128ds - $2\left(64\times64\times(3\times3) + 64 + 128\right)$ - 2 conv + 2 batch norm
+ $(64\times64\times(1\times1) + 64)$ → In-downsample
+ $(128\times128\times(1\times1) + 128)$ → Out-downSample
= 94784

4) skip128-arr - $16\left[2\left(128\times128\times(3\times3) + 128 + 256\right)\right]$
= 4730880

5) Skip128to256ds - $2\left(128\times128\times(3\times3) + \overset{\text{bias}}{128} + \overset{\text{batch norm}}{256}\right)$
+ $(128\times128\times(1\times1) + 128)$
+ $(256\times256\times(1\times1) + 256)$
= 377984

6) skip256-arr - $16\left[2\left(256\times256\times(3\times3) + 256 + 512\right)\right]$
= 18898944

7) fc1 (fwd layer 1) = $4\times4\times256\times1000 + 1000$ = 4097000

8) fc2 (fwd layer 2) = $1000\times10 + 10$ = 10010

= 29397186

↳ actual learnable params in n/w

1413276

Figure 1: Parameter Calculation for MyNet with depth 16

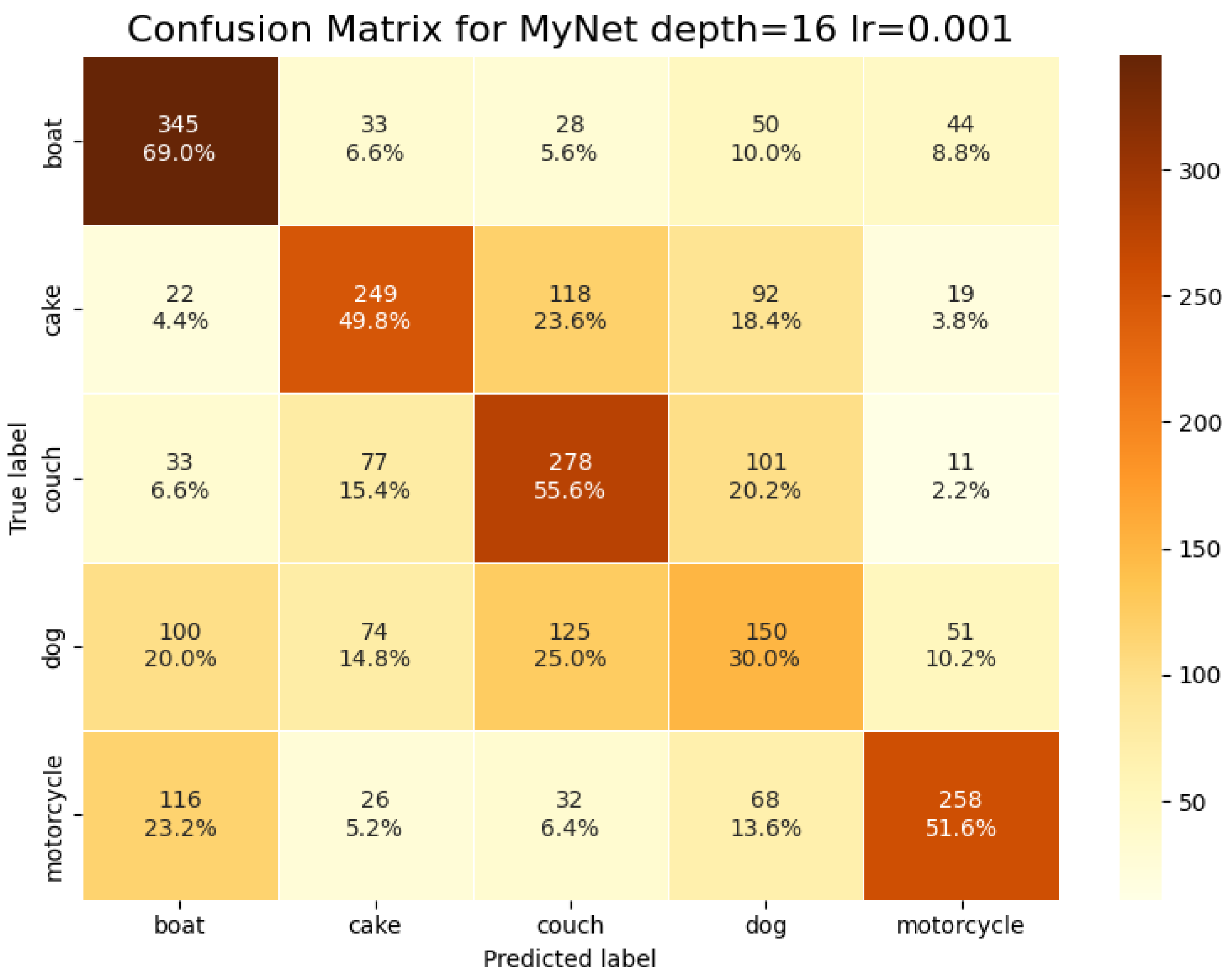


Figure 2: Confusion Matrix for MyNet with lr=0.001

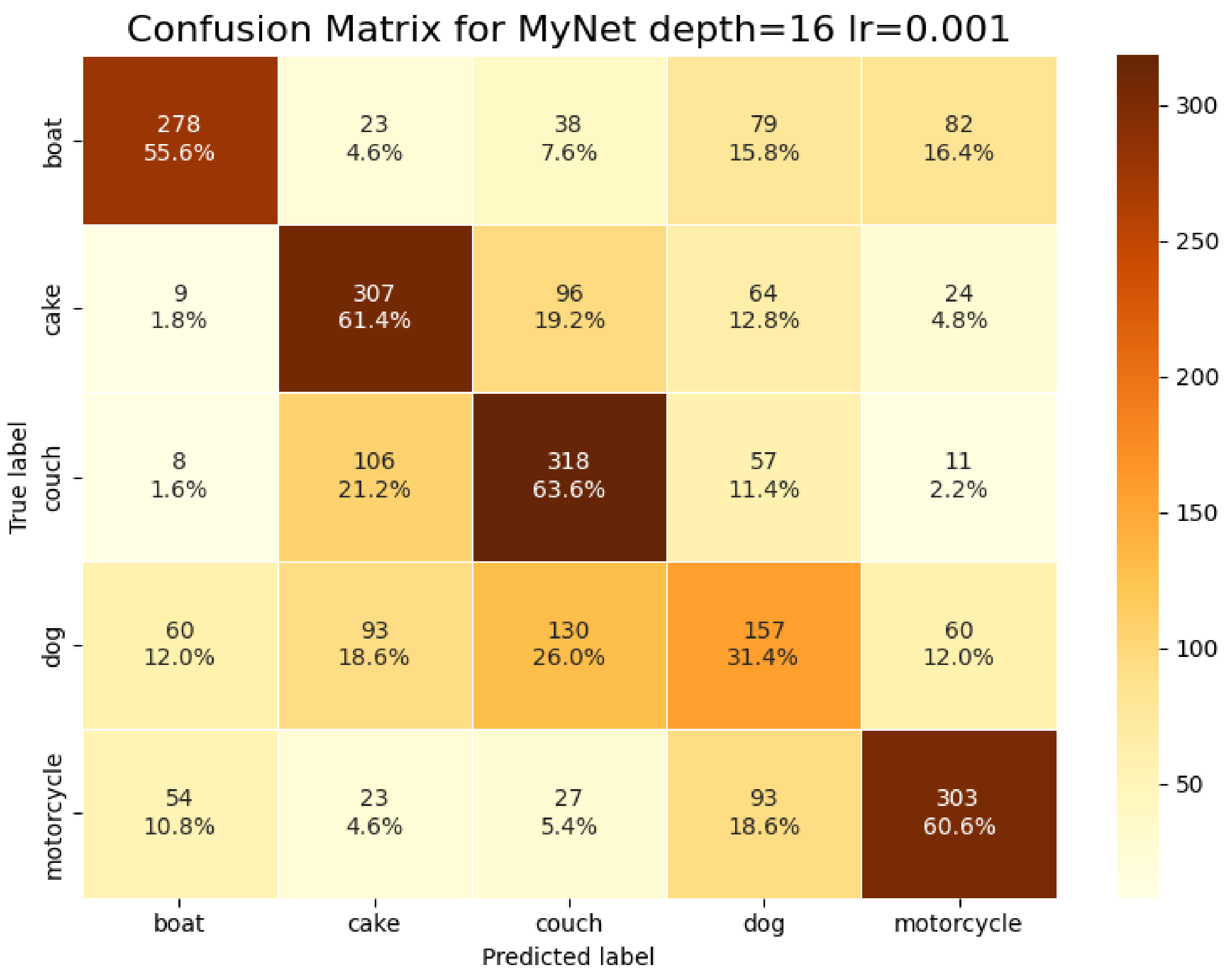


Figure 3: Confusion Matrix for MyNet with lr=0.0001

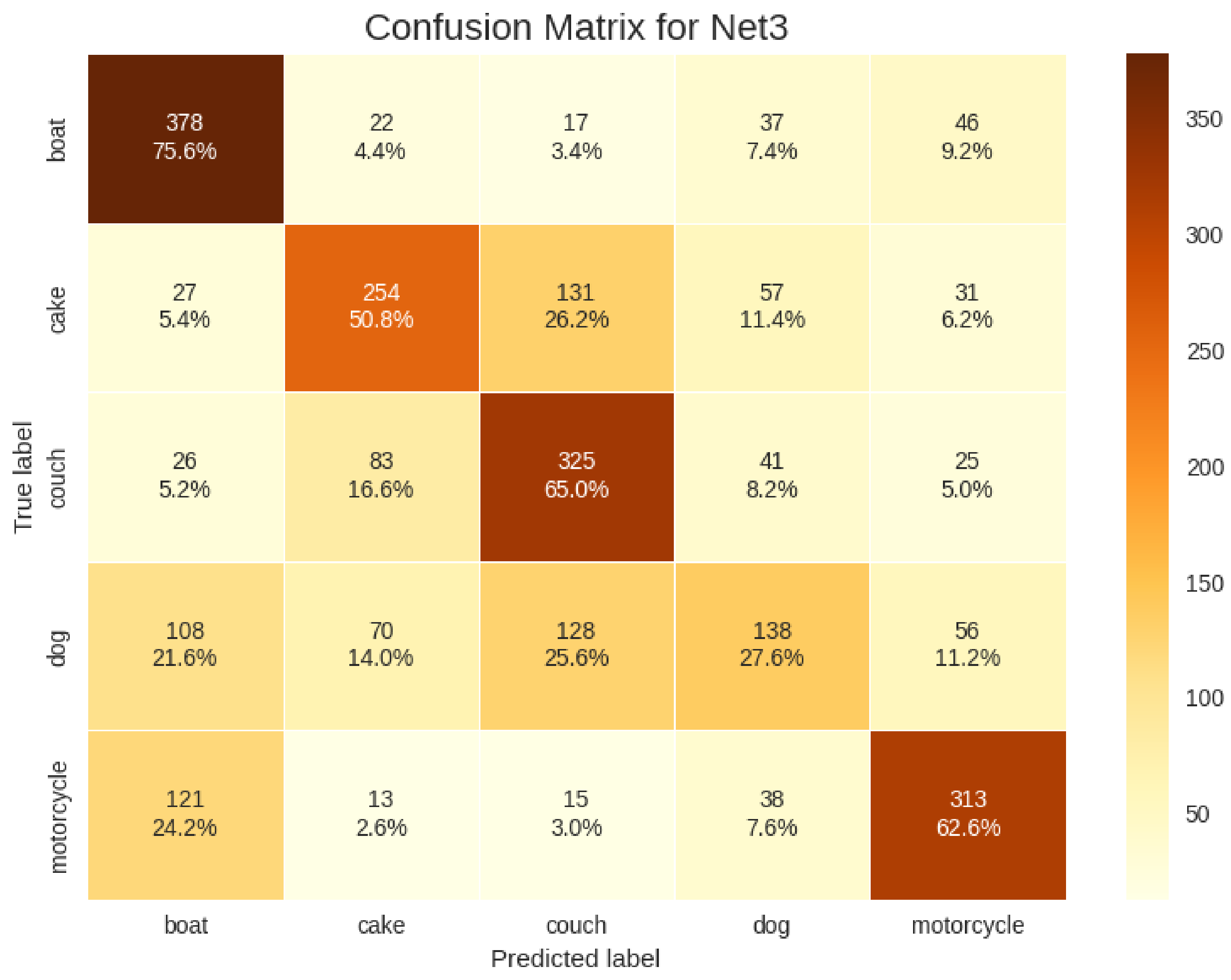


Figure 4: Confusion Matrix for Net3 of HW4

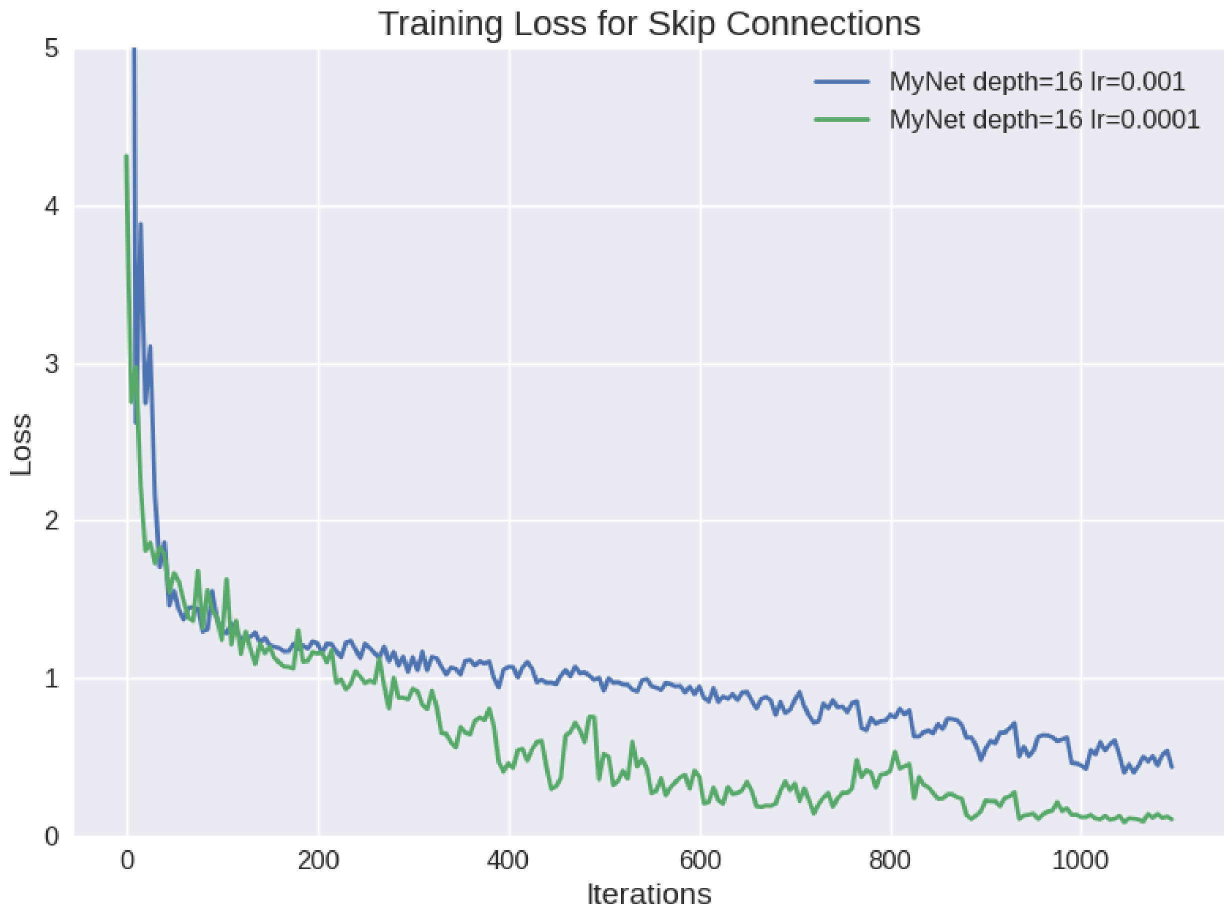


Figure 5: Training Loss for MyNet