Image Classification on MS-COCO dataset

Shubham Kose (skose@purdue.edu)

Introduction:

The focus of this project is to introduce us to the MS-COCO dataset and use the subset of this vast dataset to run on a bunch of Neural networks and understand the image classification process using CNN.

Got familiar with COCO-API before starting to create my own dataset. Used the command "conda install -c conda-forge pycocotools" to download COCO Api. Downloaded the datasets for the 2017 Train/Val annotations and the 2017 Train images from the link to the official website.

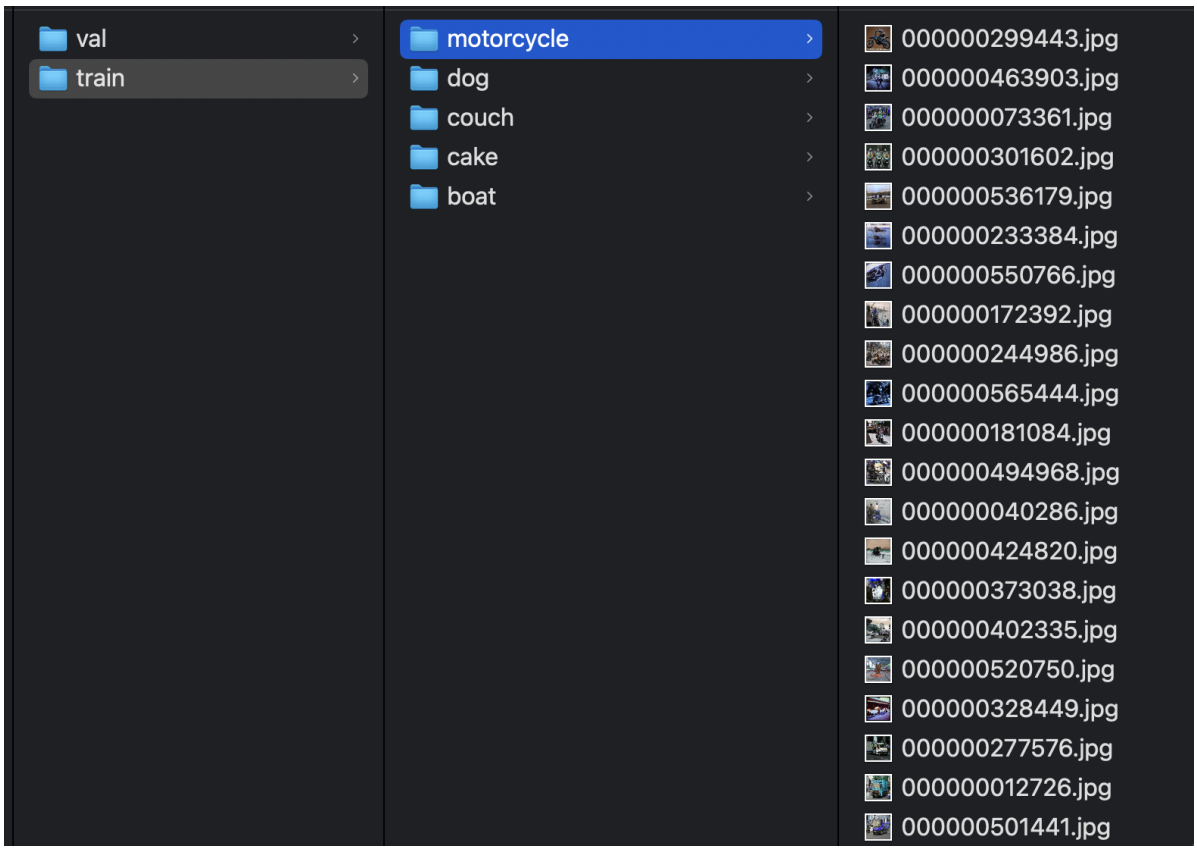
Task 3.1 : Using COCO to Create Your Own Image Classification Dataset

Created a subset of the COCO dataset for image classification, focusing on specific classes ('boat', 'cake', 'couch', 'dog', 'motorcycle'). The below process is followed to create a subset of COCO dataset:

1. Initialized the COCO API using the specified annotations file ‘/Users/skose/Downloads/CocoDataset/annotations/instances_train2017.json’ and set up the output directory '/Users/skose/Downloads/CocoSubset’ where the new image classification subset will be saved. I have3 also defined the desired classes ['boat', 'cake', 'couch', 'dog', 'motorcycle'] for image classification.
2. Created separate directories for each desired class within the output directory, distinguishing between training and validation sets.
3. **Iteration through COCO Images:** Looped through all images in the COCO dataset, extracting image information, and loading annotations. initializing

counters for tracking the number of training and validation images.

4. Checked for annotations that match the desired classes, and for each matching class, loaded and resized the image. The resized image is then saved to the appropriate directory based on whether it's for training or validation. The loop breaks if the required number of images is reached for all classes.
5. Break the loop when the required number of images (1600 for training and 400 for validation) is reached for all desired classes.



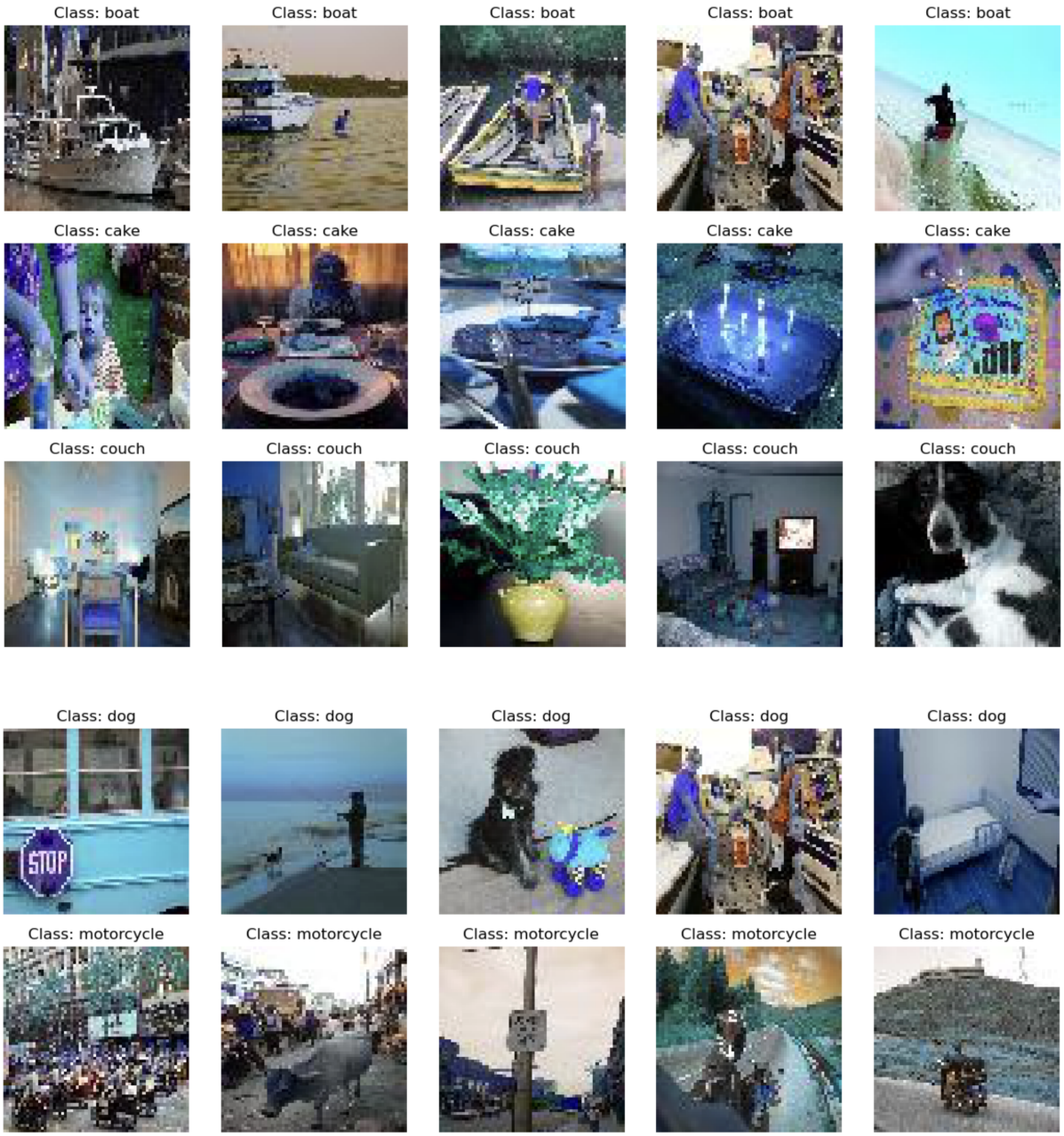
Directory Structure for Train and validation data for the COCO dataset subset

Next, created a custom dataset class named `MyCocoDataset` for image classification using a subset of the COCO dataset. The class inherits from PyTorch's `Dataset` class. The dataset is initialized with the root directory containing the organized image subsets for specific classes ('boat', 'cake', 'couch', 'dog', 'motorcycle'). The class keeps track of file names, where each file is represented by a dictionary containing the image path and its corresponding label. The labels are assigned

based on the index of the class in the predefined list of classes. The len method is overridden to return length of the datatset, allowing users to determine the dataset's size and obtain image-label pairs using the getitem method. The images are loaded as PIL images, and undergo a predefined transformation (scaling and normalization), and are returned along with their associated labels when requested through the index.

[5]:
```python
#Custom dataset class for image classification using COCO subset dataset
class MyCocoDataset(Dataset):
    def __init__(self, root):
        super().__init__()
        self.root = root
        self.file_names = []  #storing filename paths for images and associated labels
        self.classes = ['boat', 'cake', 'couch', 'dog', 'motorcycle']
        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_path, image_file), 'label': self.classes.index(label)} #creating a dictionary to sto
                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
```

Below is a plot of 5 images from each of the five classes of the subset dataset curated from the original COCO dataset.



Task 3.2: Image Classification using CNNs – Training and Validation

$$W_{out} = \frac{W - F + 2P}{S} + 1$$

Formula for Convolution Layer

The above formula is used to calculate the input tensor dimensions for the fully connected layer. Here Wout is the output dimension for the image, W is the image input dimension, F is the kernal dimension, P is the padding, and S is the stride used.

CNN Task 1:

A single-layer CNN network named `HW4Net1` for image classification is created. Below is an explanation of the calculations and formulas involved in the model:

1. **Input Image Size (conv1):**
    - The first convolutional layer (`conv1`) takes input images of size 64×64 pixels that we resized and saved .
    - The convolution operation is performed with a kernel size of 3×3.
    - Output size: (64 - 3 + 1) = 62×62 pixels.
2. **Pooling Layer (pool):**
    - A max-pooling layer (`pool`) with a kernel size of 2×2 and a stride of 2 is applied.
    - Input size: 62×62 pixels.
    - Output size: 62 / 2 = 31×31 pixels.
3. **Second Convolutional Layer (conv2) and pooling layer:**
    - The second convolutional layer (`conv2`) takes the 31×31-pixel output from the pooling layer.

- Convolution is performed with a kernel size of 3×3.
- Output size: (31 - 3 + 1) = 29×29 pixels.
- **Pooling Layer (pool):**
  - A max-pooling layer (`pool`) with a kernel size of 2×2 and a stride of 2 is again applied.
  - Input size: 29×29 pixels.
  - Output size: 29/ 2 = 14×14 pixels.

4. **Flatten Layer:**
   - The output from the second convolutional layer is flattened using `x.view`.
   - The input to the fully connected layer (`fc1`) is a 1D tensor with a size of 32 * 14 * 14 (computed from the output size of the previous layer).
5. **First Fully Connected Layer (fc1):**
   - A fully connected layer (`fc1`) with input features equal to the flattened tensor size (XXXX = 32*14*14 = 6272) and output features set to 64.
   - Output size: 64.
6. **Second Fully Connected Layer (fc2):**
   - The final fully connected layer (`fc2`) takes the output from the previous layer.
   - Output features are set to 5 (X=5), representing the number of desired classes.
7. **Activation Function:**
   - ReLU activation functions (`F.relu`) are applied after the convolutional and fully connected layers.

[128]:
```python
#class for Single layer CNN Net1
#implementation is based on the example HW4Net network provided in the homework4 document
class HW4Net1(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3) #Input image size :64x64, Output: (64-3+1) = 62x62
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)  # Input: 62x62, Output: 62/2 = 31x31
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3)  # Input: 31x31, Output: (31-3+1) = 29x29
        self.fc1 = nn.Linear(in_features=32 * 14 * 14, out_features=64) # Input: XXXX= 32*14*14 = 6272, Output: 64
        self.fc2 = nn.Linear(in_features=64, out_features=5) # Output: X=5 desired classes
    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x))) # Input: 64x64, Output: 31x31
        x = self.pool(F.relu(self.conv2(x))) # Input: 31x31, Output: 14x14
        x = x.view(x.shape[0], -1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

CNN Task 2:

1. **Input Image Size (conv1):**
    - The first convolutional layer (`conv1`) takes input images of size 64×64 pixels.
    - The convolution operation is performed with a kernel size of 3×3 and padding of 1.
    - Output size: ((64 - 3 + 2 * 1) / 1) + 1 = 64; After max-pooling (2×2), the output size becomes 32×32 pixels.
2. **Pooling Layer (pool):**
    - A max-pooling layer (`pool`) with a kernel size of 2×2 and a stride of 2 is applied.
    - Input size: 64×64 pixels.
    - Output size: (64 / 2) = 32×32 pixels.
3. **Second Convolutional Layer (conv2):**
    - The second convolutional layer (`conv2`) takes the 32×32-pixel output from the pooling layer.
    - Convolution is performed with a kernel size of 3×3 and padding of 1.
    - Output size: ((32 - 3 + 2 * 1) / 1) + 1 = 32; After max-pooling (2×2), the output size becomes 16×16 pixels.
    - **Pooling Layer (pool):**

- A max-pooling layer (`pool`) with a kernel size of 2×2 and a stride of 2 is again applied.
- Input size: 32×32 pixels.
- Output size: 32/ 2 = 16×16 pixels.

4. **Flatten Layer:**
   - The output from the second convolutional layer is flattened using `x.view`.
   - The input to the fully connected layer (`fc1`) is a 1D tensor with a size of 32 * 16 * 16 (computed from the output size of the previous layer).
5. **First Fully Connected Layer (fc1):**
   - A fully connected layer (`fc1`) with input features equal to the flattened tensor size (XXXX = 32*16*16 = 8192) and output features set to 64.
   - Output size: 64.
6. **Second Fully Connected Layer (fc2):**
   - The final fully connected layer (`fc2`) takes the output from the previous layer.
   - Output features are set to 5 (X = 5), representing the number of desired classes.
7. **Activation Function:**
   - ReLU activation functions (`F.relu`) are applied after the convolutional and fully connected layers.

[141…

```python
class HW4Net2(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3,padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3,padding=1)
        self.fc1 = nn.Linear(in_features=32 * 16 * 16, out_features=64) # Input: XXXX= 32*16*16 = 8192, Output: 64
        self.fc2 = nn.Linear(in_features=64, out_features=5) # Output: X=5 desired classes
    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x))) # Output Size: (64 - 3 + 2*1)/1 + 1 = 64; After Max Pooling: 32x32
        x = self.pool(F.relu(self.conv2(x))) # Output Size: (32 - 3 + 2*1)/1 + 1 = 32; After Max Pooling: 16x16
        x = x.view(x.shape[0], -1) #Flattened Output Size: 32 * 16 * 16
        x = F.relu(self.fc1(x)) # Output Size: 64
        x = self.fc2(x)   # Output Size: 5 (for 5 classes)
        return x
```

CNN Task 3:

1. **Input Image Size (conv1):**
    - The first convolutional layer (`conv1`) takes input images of size 64×64 pixels.
    - The convolution operation is performed with a kernel size of 3×3 and padding of 1.
    - Output size: ((64 - 3 + 2 * 1) / 1) + 1 = 64; After max-pooling (2×2), the output size becomes 32×32 pixels.
2. **Pooling Layer (pool):**
    - A max-pooling layer (`pool`) with a kernel size of 2×2 and a stride of 2 is applied.
    - Input size: 64×64 pixels.
    - Output size: (64 / 2) = 32×32 pixels.
3. **Second Convolutional Layer (conv2):**
    - The second convolutional layer (`conv2`) takes the 32×32-pixel output from the pooling layer.
    - Convolution is performed with a kernel size of 3×3 and padding of 1.
    - Output size: ((32 - 3 + 2 * 1) / 1) + 1 = 32; After max-pooling (2×2), the output size becomes 16×16 pixels.
4. **Consecutive Convolutional Layers (`conv_layers`):**
    - A list (`conv_layers`) contains 10 consecutive convolutional layers.
    - Each layer takes the 16×16-pixel output from the previous layer and performs convolution with a kernel size of 3×3 and padding of 1.
    - Output size for each layer: 16×16 pixels.
5. **Flatten Layer:**
    - The output from the last convolutional layer is flattened using `x.view`.
    - The input to the fully connected layer (`fc1`) is a 1D tensor with a size of 32 * 16 * 16 (computed from the output size of the previous layer).
6. **First Fully Connected Layer (fc1):**

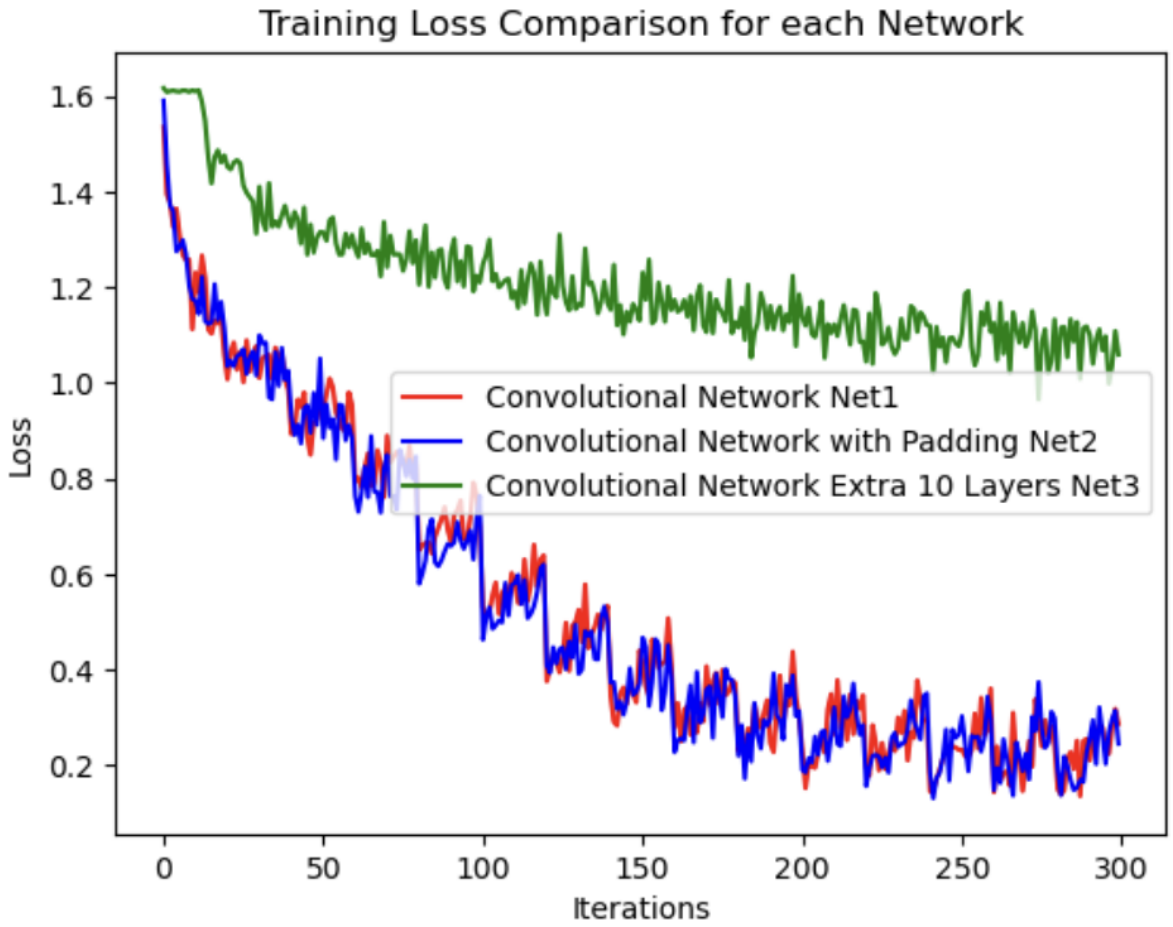
- A fully connected layer (`fc1`) with input features equal to the flattened tensor size (XXXX = 8192) and output features set to 64.
- Output size: 64.

7. **Second Fully Connected Layer (fc2):**
   - The final fully connected layer (`fc2`) takes the output from the previous layer.
   - Output features are set to 5 (X=5) , representing the number of desired classes.
8. **Activation Function:**
   - ReLU activation functions (`F.relu`) are applied after the convolutional and fully connected layers.

[186…

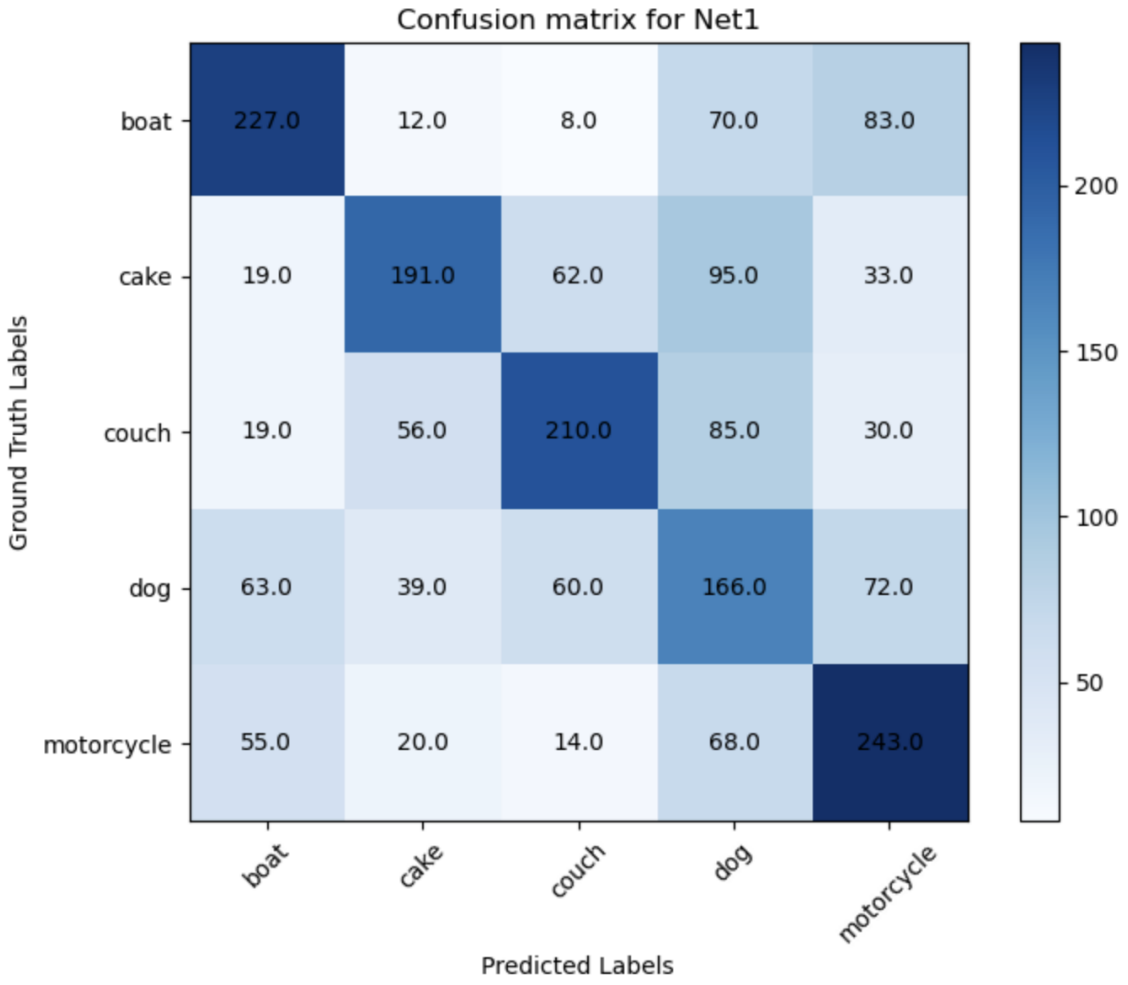
```python
class HW4Net3(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, padding=1)
        # List of 10 Consecutive Convolutional Layers
        self.conv_layers = nn.ModuleList([nn.Conv2d(in_channels=32, out_channels=32, kernel_size=3, padding=1) for _ in range(10)])
        self.fc1 = nn.Linear(in_features=32 * 16 * 16, out_features=64)  # Input: XXXX= 32*16*16 = 8192, Output: 64
        self.fc2 = nn.Linear(in_features=64, out_features=5)  # Output: X=5 desired classes
    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x))) # Output Size: (64 - 3 + 2*1)/1 + 1 = 64; After Max Pooling: 32x32
        x = self.pool(F.relu(self.conv2(x))) # Output Size: (32 - 3 + 2*1)/1 + 1 = 32; After Max Pooling: 16x16
        # Consecutive Convolutional Layers in the List
        for m in self.conv_layers:
            x = F.relu(m(x))  # Output Size: 16x16 for each layer
        x = x.view(x.shape[0], -1) # Flattened Output Size: 32 * 16 * 16
        x = F.relu(self.fc1(x)) # Output Size: 64
        x = self.fc2(x)    # Output Size: 5 (for 5 classes)
        return x
```

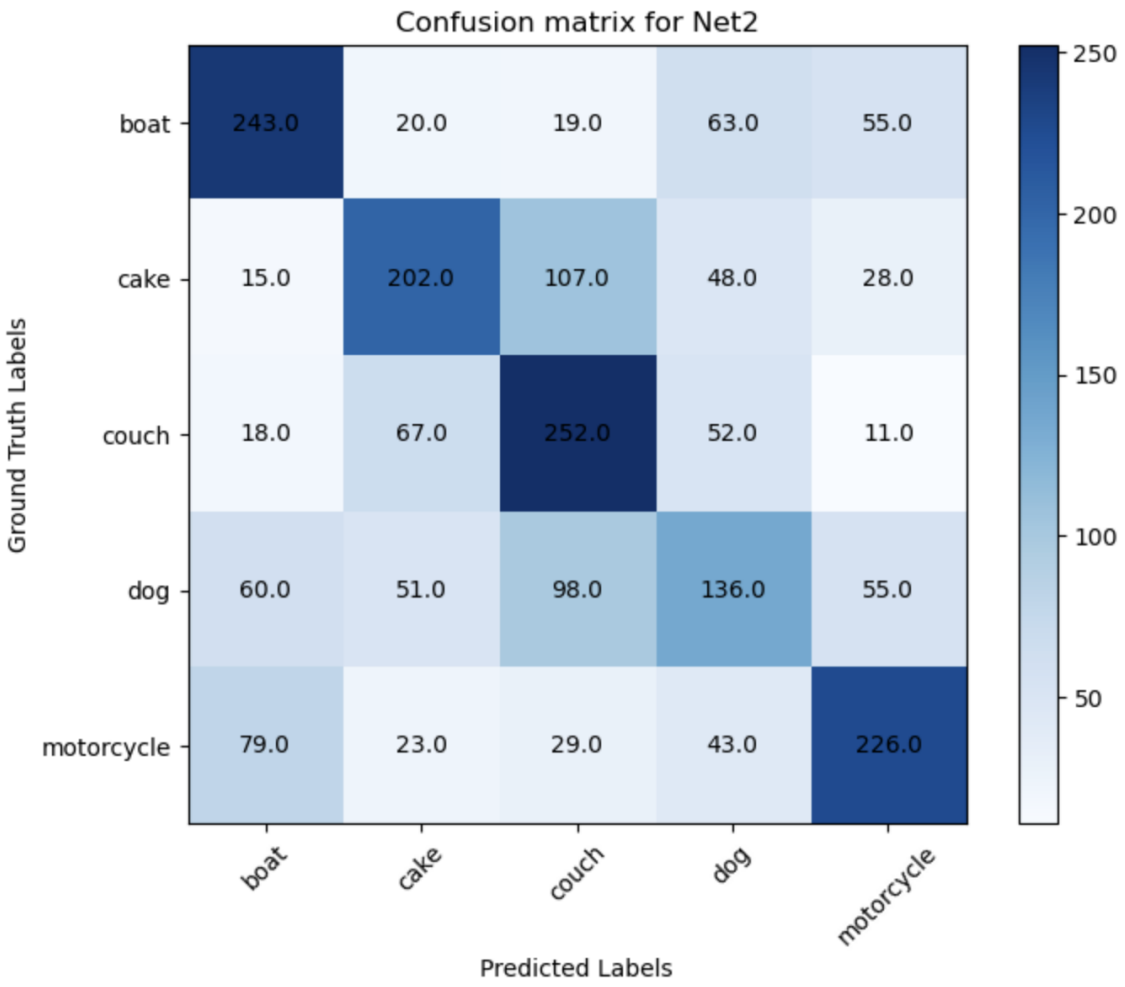
Training loss comparison for the three networks:

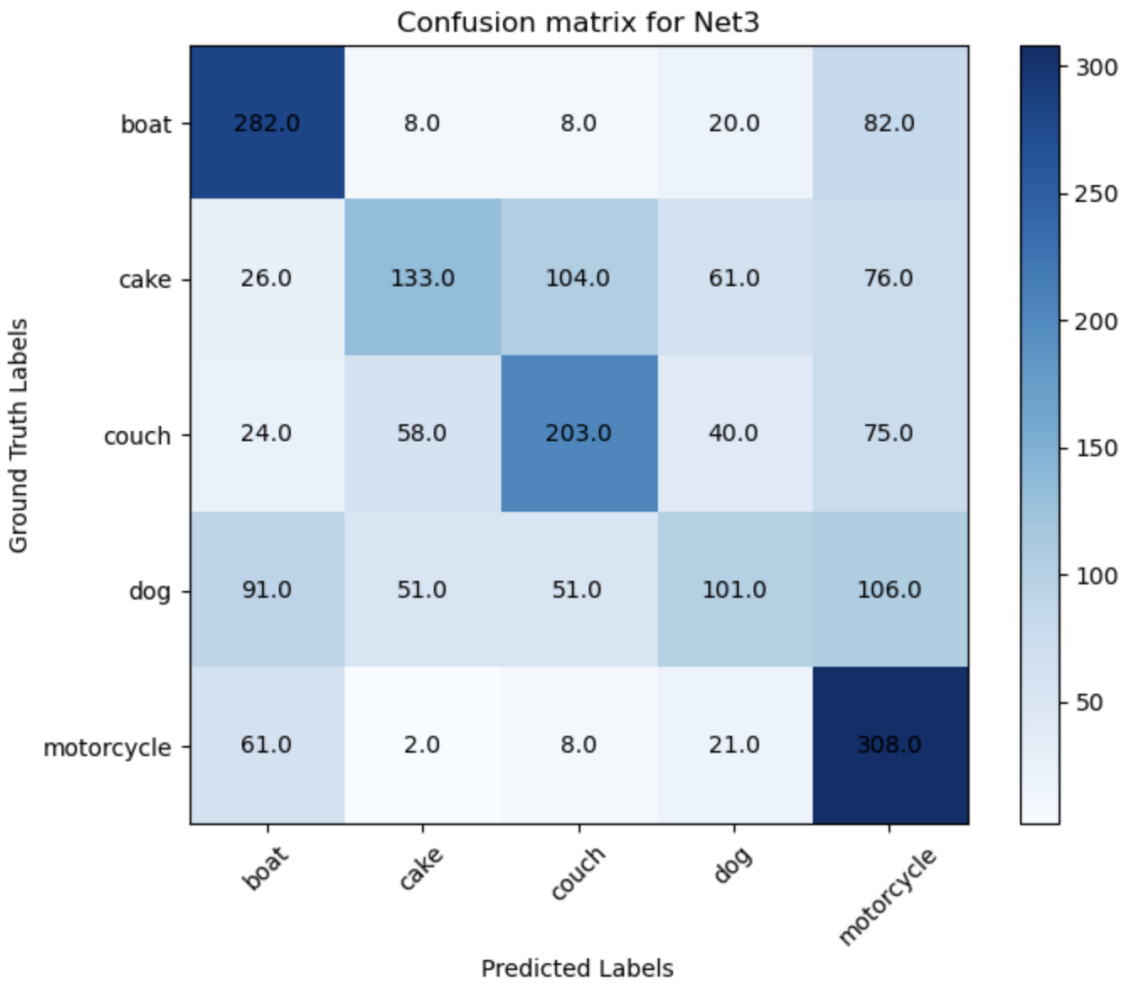


In the above loss curve vs iterations, we can see that both Network 1 and Network 2 have similar performance with Network 2 only performing slightly better.
However, Network 3 does not perform as good as the other two networks.

Confusion Matrix plot for the 3 Networks:


Confusion matrix for Net1
boat
cake
couch
dog
motorcycle
227.0
12.0
8.0
70.0
83.0
19.0
191.0
62.0
95.0
33.0
19.0
56.0
210.0
85.0
30.0
63.0
39.0
60.0
166.0
72.0
55.0
20.0
14.0
68.0
243.0
200
150
100
50
Ground Truth Labels
Predicted Labels


Confusion matrix for Net2
boat
cake
couch
dog
motorcycle
Ground Truth Labels
243.0
20.0
19.0
63.0
55.0
15.0
202.0
107.0
48.0
28.0
18.0
67.0
252.0
52.0
11.0
60.0
51.0
98.0
136.0
55.0
79.0
23.0
29.0
43.0
226.0
boat
cake
couch
dog
motorcycle
Predicted Labels
250
200
150
100
50



Classification Accuracy and Number of Parameters for the 3 Networks:

	Net1	Net2	Net3
No. of Parameters	`406885`	`529765`	`622245`
Classification Accuracy	`51.8499%`	`52.9499%`	`51.3499%`

As can be seen from above confusion matrix images for the 3 networks. The accuracy for the networks are as follows:
Network 1: 51.8499%
Network 2: 52.9499%
Network 3: 51.3499%

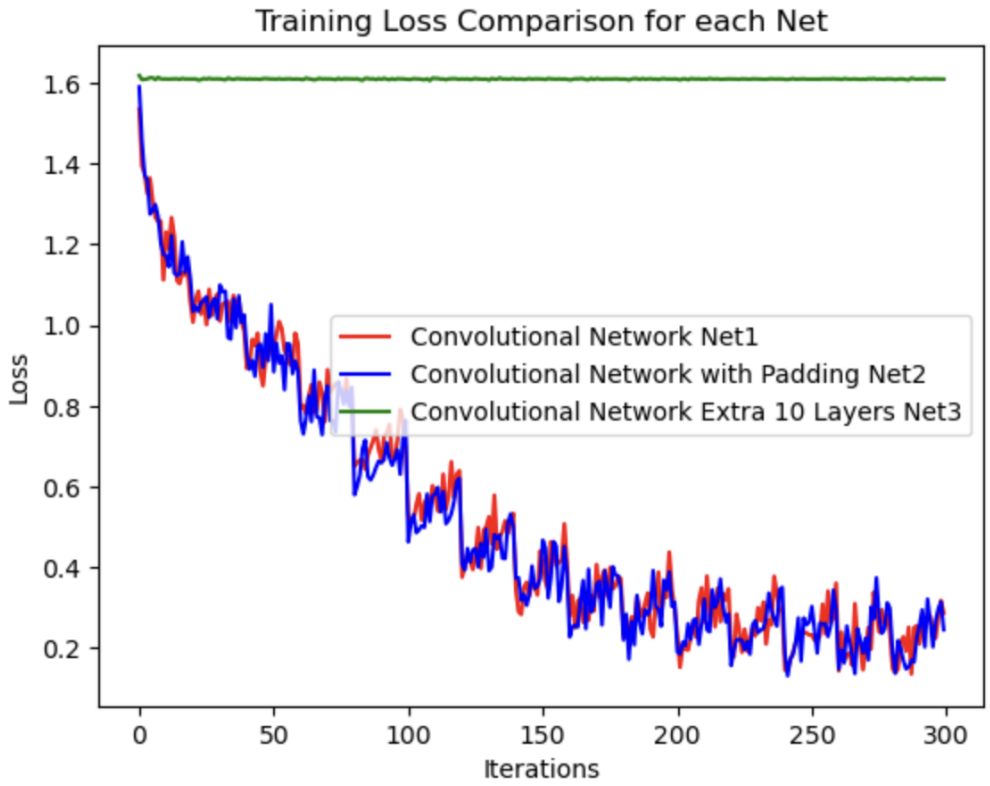
The respective training losses show that Network 3 had the highest loss and lowest validation accuracy among all the 3 networks. Network 2 with padding performs slightly better than Network 1 without padding

Answers to the questions asked:

1. Does adding padding to the convolutional layers make a difference in classification performance?
   >> Based on the confusion matrix results and the loss curve, adding padding to convolutional layers increase the classification performance and decrease the loss curve slightly, as seen in
   `HW4Net2.` Padding can help in maintaining spatial information especially when dealing with features at the edges of the input image, and the network can learn more from the edges as well.

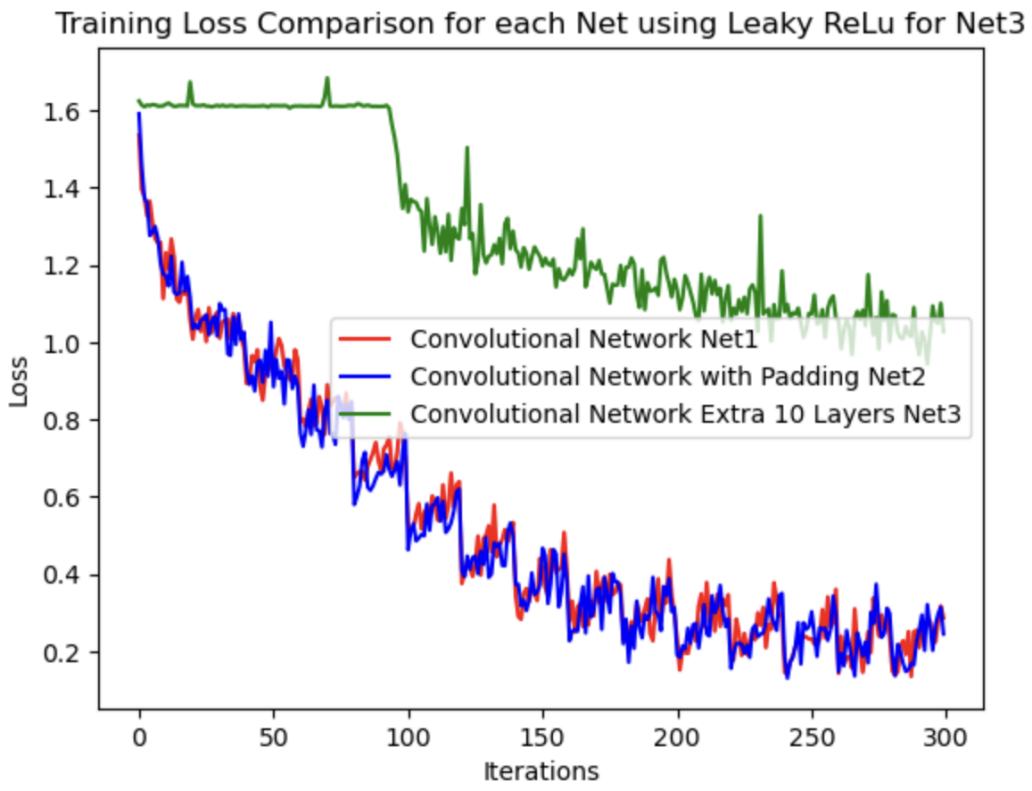
2. As you may have known, naively chaining a large number of layers can result in difficulties in training. This phenomenon is often referred to as vanishing gradient. Do you observe something like that in Net3?
   >> Yes, the above loss curve for the Network 3 was obtained randomly only a few times when I repeatedly kept training the Network 3. The below loss curve was obtained when the same Network 3 was trained again on the same data and plotted the loss curve.

   In the provided `HW4Net3` architecture, there's a chain of consecutive convolutional layers (`conv_layers`) with a total of 10 layers. The vanishing gradient problem occurs when gradients are propagated back through multiple layers during backpropagation, eventually becoming incredibly small. This makes training difficult for deep networks.



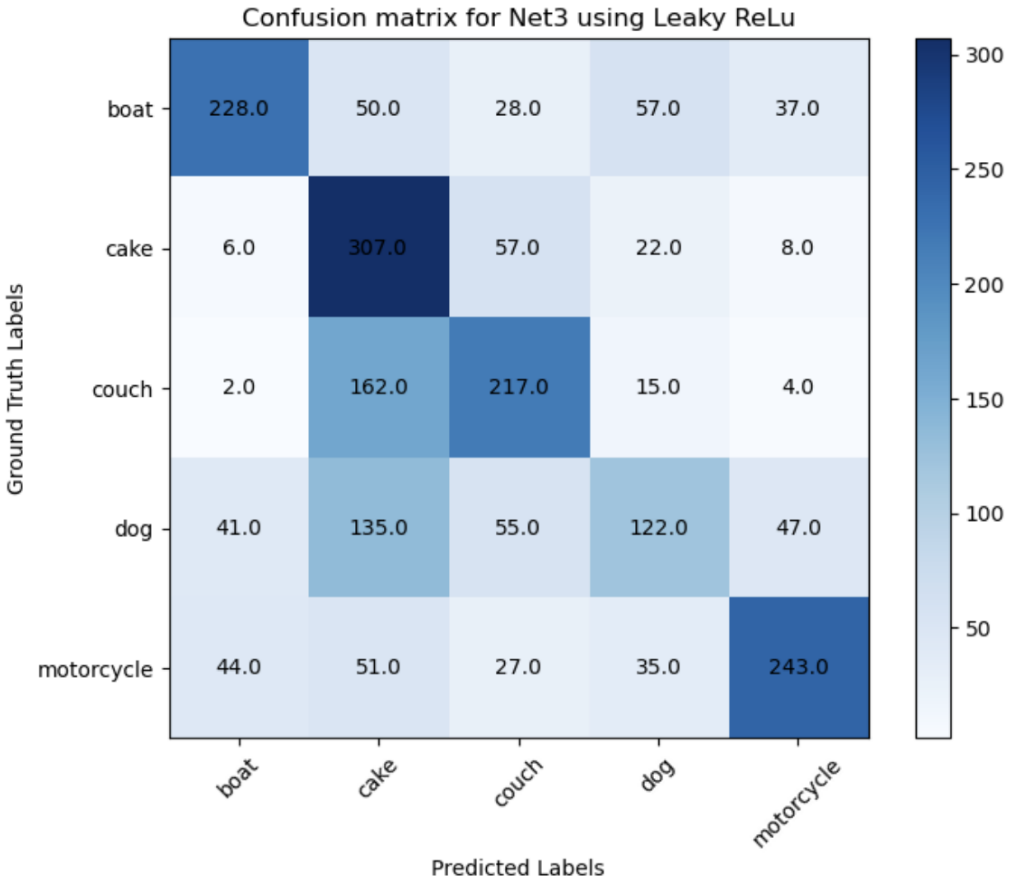
In the above loss curve shown, it shows that the loss for Network 3 does not ever decrease. Since we have stacked 10 convolutional layers in the `HW4Net3` network, the vanishing gradient problem can occur due to repeated application of ReLu activation functions, as some neurons can output zero for all inputs.

To address the vanishing gradient problem, techniques such as batch normalization, skip connections (residual connections), and careful weight initialization can be used. To improve the training stability and convergence of `HW4Net3`, I tried experimenting with changing the activation function from ReLU to Leaky ReLu layer for all the layers. The resulting loss curve performed a bit better and decreased a bit as shown below:



Below is the confusion matrix and classification accuracy of 55.85% achieved for Net3 network when using a Leaky ReLu instead of ReLu layer:

```
Begin...

Prediction accuracy for  boat : 57 %
Prediction accuracy for  cake : 76 %
Prediction accuracy for couch : 54 %
Prediction accuracy for   dog : 30 %
Prediction accuracy for motorcycle : 60 %
Finished!

Accuracy of the network on validation images: 55.85%
```



3. Compare the classification results by all three networks, which CNN do you think is the best performer?
   >> On comparing classification results, Network 2 performed the best on my dataset. As shown in the above loss curve that Network 2 has comparatively the best performance out of all the 3 networks and its classification accuracy is also the highest, followed by Network 1 and then Network 3 performed the worst among all due to the problem of vanishing gradient descent.

4. By observing your confusion matrices, which class or classes do you think are more difficult to correctly differentiate and why?
   >> Looking at the confusion matrix for all the 3 Networks, it is observed that

the dog class score comparatively low across the networks when compared to other classes. One of the reasons might be that they often appear in similar contextual backgrounds as couch and motorcycle and boat. Also a dog on a boat and a dog on a motorcycle or a dog alongside a cake can be a few possible examples of images where dogs appear in the same context as other classes and hence it might be difficult for better classification of the Dog class.

5. What is one thing that you propose to make the classification performance better? >> To make the classification performance better, there are a lot of different ways we can experiment, such as augmenting more data using different transformations so that the model learns to classify more generic objects. Fine tuning the exisiting networks and trying to find the best performance. Increasing the resolution can help retaining more information in the image and help model learn better using more features. To solve the problem of vanishing gradient descent in Net 3, we can try initializing weights, using batch normalization and residual networks, using different activation functions (like I used leaky ReLu instead of ReLu and achieved better performance).

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
```

```
#setting seed value for reproducibility
# The code is Borrowed from Dr. Avinash Kak's DL Studio Modul
e
seed = 63
random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
numpy.random.seed(seed)
torch.backends.cudnn.deterministic=True
torch.backends.cudnn.benchmarks=False
os.environ["PYTHONHASHSEED"] = str(seed)
```

```
#Created a subset of the COCO dataset for image classificatio
n, focusing on specific classes ('boat', 'cake', 'couch', 'do
g', 'motorcycle')
# Path to the COCO annotations file
coco_ann_file = '/Users/skose/Downloads/CocoDataset/annotatio
ns/instances_train2017.json'

# Initialize COCO API
coco = COCO(coco_ann_file)

# Output directory for saving the new image classification su
bset of COCO dataset
```

```
output_dir = '/Users/skose/Downloads/CocoSubset'
os.makedirs(output_dir, exist_ok=True)

# Desired classes for image classification as mentioned in Ho
mework 4
desired_classes = ['boat', 'cake', 'couch', 'dog', 'motorcycl
e']

# Create directories for each class in the output directory
for label in desired_classes:
    class_train_dir = os.path.join(output_dir, 'train', labe
l)
    os.makedirs(class_train_dir, exist_ok=True)
    class_val_dir = os.path.join(output_dir, 'val', label)
    os.makedirs(class_val_dir, exist_ok=True)

# Counters for tracking the number of training and validation
images
train_counter = {label: 0 for label in desired_classes}
val_counter = {label: 0 for label in desired_classes}

# Iterate through all images in the COCO dataset
for img_id in coco.getImgIds():
    img_info = coco.loadImgs(img_id)
    img_path = os.path.join('/Users/skose/Downloads/CocoDatas
et/train2017', img_info[0]['file_name'])

    # Get annotations for the image
    ann_ids = coco.getAnnIds(imgIds=img_id)
    annotations = coco.loadAnns(ann_ids)

    # Check for annotations that matches the desired classes
    matching_classes = set()  #set for containing the names o
f classes (included only once) for annotations associated wit
h the current image
    for ann in annotations:
```

```
        category_id = ann['category_id']
        category_info = coco.loadCats(category_id)[0]
        category_name = category_info['name']
        matching_classes.add(category_name)

    for label in matching_classes:
        if label in desired_classes:
            # Loading and resizing the image using OpenCV
            image = cv2.imread(img_path)
            resized_image = cv2.resize(image, (64, 64)) #64x6
4 size image as asked in the homework

            # Saving the resized image using PIL
            pil_image = Image.fromarray(resized_image)

            # Determine whether to use the image for training
or validation
            if train_counter[label] < 1600: #1600 training im
ages required
                pil_image.save(os.path.join(output_dir, 'trai
n', label, img_info[0]['file_name']))
                train_counter[label] += 1
            elif val_counter[label] < 400:  #400 validation i
mages required
                pil_image.save(os.path.join(output_dir, 'va
l', label, img_info[0]['file_name']))
                val_counter[label] += 1

            # Break the loop if the required number of images
is reached for all classes
            if all(train_counter[label] == 1600 and val_count
er[label] == 400 for label in desired_classes):
                break

    # Break the loop if the required number of images is reac
hed for all classes
```

```
    if all(train_counter[label] == 1600 and val_counter[labe
l] == 400 for label in desired_classes):
        break
```

```
#Custom dataset class for image classification using COCO sub
set dataset
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
trainDataset = MyCocoDataset('/Users/skose/Downloads/CocoSubs
et/train')
valDataset = MyCocoDataset('/Users/skose/Downloads/CocoSubse
t/val')
```

```
#Plotting 5 images from each of the five classes
import numpy as np

# Create a figure with subplots
fig, axes = plt.subplots(nrows=5, ncols=5, figsize=(12, 12))

# Loop through each class
for i, label in enumerate(trainDataset.classes):
    # Filter images of the current class
    class_images = [item for item in trainDataset.file_names
if item['label'] == i][:5]

    # Loop through 5 images of the current class
    for j, item in enumerate(class_images):
        image_path = item['image_path']
        label = item['label']

        # Read and display the image
```

```
        image = Image.open(image_path)
        axes[i, j].imshow(np.asarray(image))
        axes[i, j].set_title(f'Class: {trainDataset.classes[l
abel]}')

        # Hide axes ticks and labels
        axes[i, j].axis('off')

# Adjust layout for better visualization
plt.tight_layout()
plt.show()
```

```
#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
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-3, 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)
```

```
# Plotting confusion matrix heatmap using matplotlib
import numpy as np
from sklearn.metrics import confusion_matrix

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()
```

```
#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))
```

```
#class for Single layer CNN Net1
#implementation is based on the example HW4Net network provid
ed in the homework4 document
class HW4Net1(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=1
6, kernel_size=3) #Input image size :64x64, Output: (64-3+1)
= 62x62
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)  #
Input: 62x62, Output: 62/2 = 31x31
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=3
2, kernel_size=3)  # Input: 31x31, Output: (31-3+1) = 29x29
        self.fc1 = nn.Linear(in_features=32 * 14 * 14, out_fe
atures=64) # Input: XXXX= 32*14*14 = 6272, Output: 64
        self.fc2 = nn.Linear(in_features=64, out_features=5)
# Output: X=5 desired classes
    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x))) # Input: 64x64,
Output: 31x31
```

```
        x = self.pool(F.relu(self.conv2(x))) # Input: 31x31,
Output: 14x14
        x = x.view(x.shape[0], -1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

```
#run the training loop and save the training loss for plottin
g later
net1 = HW4Net1()
epochs_net1 = 15
model_path1=os.path.join('/Users/skose/Downloads/path', 'net1
1.pth')
training_loss_net1 = training_routine(net1, epochs_net1, trai
nDataLoader, device, model_path1)
```

```
#run the evaluation on validation dataset and save the valida
tion accuracy and confusion matrix
batch_size1=4
confusion_matrix_net1, validation_acc_net1 = model_evaluation
(net1, valDataLoader, batch_size1, device, desired_classes,mo
del_path1)
```

```
#Plot confusion matrix
title= 'Confusion matrix for Net1'
plot_confusion_matrix(confusion_matrix_net1, title)
```

```
#class for Single layer CNN Net2
class HW4Net2(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=1
6, kernel_size=3,padding=1)
```

```
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=3
2, kernel_size=3,padding=1)
        self.fc1 = nn.Linear(in_features=32 * 16 * 16, out_fe
atures=64)
        self.fc2 = nn.Linear(in_features=64, out_features=5)
    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x))) # Output Size:
(64 - 3 + 2*1)/1 + 1 = 64; After Max Pooling: 32x32
        x = self.pool(F.relu(self.conv2(x))) # Output Size:
(32 - 3 + 2*1)/1 + 1 = 32; After Max Pooling: 16x16
        x = x.view(x.shape[0], -1) #Flattened Output Size: 32
* 16 * 16
        x = F.relu(self.fc1(x)) # Output Size: 64
        x = self.fc2(x)   # Output Size: 5 (for 5 classes)
        return x
```

```
#run the training loop and save the training loss for plottin
g later
net2 = HW4Net2()
epochs_net2 = 15  # Number of training epochs
model_path2 = os.path.join('/Users/skose/Downloads/path', 'ne
t21.pth')  # Path to save the trained model
training_loss_net2 = training_routine(net2, epochs_net2, trai
nDataLoader, device, model_path2)
```

```
#run the evaluation on validation dataset and save the valida
tion accuracy and confusion matrix
batch_size2 = 4
confusion_matrix_net2, validation_acc_net2 = model_evaluation
(net2, valDataLoader, batch_size2, device, desired_classes, m
odel_path2)
```

```
#Plot confusion matrix
title= 'Confusion matrix for Net2'
plot_confusion_matrix(confusion_matrix_net2, title)
```

```
#class for CNN Network with 10 additional convolutional layer
Net3
class HW4Net3(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=1
6, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=3
2, kernel_size=3, padding=1)
        # List of 10 Consecutive Convolutional Layers
        self.conv_layers = nn.ModuleList([nn.Conv2d(in_channe
ls=32, out_channels=32, kernel_size=3, padding=1) for _ in ra
nge(10)])
        self.fc1 = nn.Linear(in_features=32 * 16 * 16, out_fe
atures=64)  # Input: XXXX= 32*16*16 = 8192, Output: 64
        self.fc2 = nn.Linear(in_features=64, out_features=5)
# Output: X=5 desired classes
    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x))) # Output Size:
(64 - 3 + 2*1)/1 + 1 = 64; After Max Pooling: 32x32
        x = self.pool(F.relu(self.conv2(x))) # Output Size:
(32 - 3 + 2*1)/1 + 1 = 32; After Max Pooling: 16x16
        # Consecutive Convolutional Layers in the List
        for m in self.conv_layers:
            x = F.relu(m(x))  # Output Size: 16x16 for each l
ayer
        x = x.view(x.shape[0], -1) # Flattened Output Size: 3
2 * 16 * 16
        x = F.relu(self.fc1(x)) # Output Size: 64
```

```
        x = self.fc2(x)    # Output Size: 5 (for 5 classes)
        return x
```

```
#run the training loop and save the training loss for plottin
g later
net3 = HW4Net3()
epochs_net3 = 15
save_path_net3 = os.path.join('/Users/skose/Downloads/path',
'net3.pth')
training_loss_net3 = training_routine(net3, epochs_net3, trai
nDataLoader, device, save_path_net3)
```

```
#run the evaluation on validation dataset and save the valida
tion accuracy and confusion matrix
batch_size_net3 = 4
confusion_matrix_net3, testing_acc_net3 = model_evaluation(ne
t3, valDataLoader, batch_size_net3, device, desired_classes,
save_path_net3)
```

```
#Plot confusion matrix
title= 'Confusion matrix for Net3'
plot_confusion_matrix(confusion_matrix_net3, title)
```

```
#class for CNN Network with 10 additional convolutional layer
Net3 using leaky ReLu instead of ReLu, to address the problem
of vanishing gradient descent
class HW4Net31(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=1
6, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=3
```

```
2, kernel_size=3, padding=1)
        # List of 10 Consecutive Convolutional Layers
        self.conv_layers = nn.ModuleList([nn.Conv2d(in_channe
ls=32, out_channels=32, kernel_size=3, padding=1) for _ in ra
nge(10)])
        self.fc1 = nn.Linear(in_features=32 * 16 * 16, out_fe
atures=64)
        self.fc2 = nn.Linear(in_features=64, out_features=5)
    def forward(self, x):
        x = self.pool(F.leaky_relu(self.conv1(x), negative_sl
ope=0.01))
        x = self.pool(F.leaky_relu(self.conv2(x), negative_sl
ope=0.01))
        for m in self.conv_layers:
            x = F.leaky_relu(m(x), negative_slope=0.01)
        x = x.view(x.shape[0], -1)
        x = F.leaky_relu(self.fc1(x), negative_slope=0.01)
        x = self.fc2(x)
        return x
```

```
#run the training loop and save the training loss for plottin
g later
net6 = HW4Net31()
epochs_net6 = 15
save_path6 = os.path.join('/Users/skose/Downloads/path', 'net
31.pth')
training_loss_net6 = training_routine(net6, epochs_net6, trai
nDataLoader, device, save_path6)
```

```
#function to plot the training loss curve for the three netwo
rks
def train_loss_plot(train_loss, title):
    plt.plot(train_loss[0], 'r', label='Convolutional Network
Net1')
```

```
    plt.plot(train_loss[1], 'b', label='Convolutional Network
with Padding Net2')
    plt.plot(train_loss[2], 'g', label='Convolutional Network
Extra 10 Layers Net3')
    plt.title(title)
    plt.legend(loc="center right")
    plt.xlabel("Iterations")
    plt.ylabel("Loss")
    plt.show()
```

```
train_loss = []
train_loss.append(training_loss_net1)
train_loss.append(training_loss_net2)
train_loss.append(training_loss_net3)
train_loss_plot(train_loss, 'Training Loss Comparison for eac
h Network')
```

```
#function to count the total number of parameters for a netwo
rk
def count_parameters(model):
    total_params = sum(p.numel() for p in model.parameters())
    return total_params
```

```
#total number of parameters for a Net1
total_parameters = count_parameters(net1)
print(f"Total parameters in the Net1 model: {total_parameter
s}")
```

```
#total number of parameters for a Net2
total_parameters = count_parameters(net2)
print(f"Total parameters in the Net2 model: {total_parameter
s}")
```

```
#total number of parameters for a Net3
total_parameters = count_parameters(net3)
print(f"Total parameters in the Net3 model: {total_parameter
s}")
```