ECE60146: Homework 4

Manish Kumar Krishne Gowda, 0033682812

(Spring 2024)

1 Introduction

This report focuses on the development and application of Convolutional Neural Networks (CNNs) using PyTorch for image classification. The primary dataset under consideration is the widely adopted MS-COCO (Microsoft Common Objects in COntext), serving as the primary resource for training and validating the designed CNNs.

Throughout the assignment, three distinct CNNs were crafted, each featuring a unique architecture comprising essential components such as convolutional layers, max pooling, and linear layers. The training and validation processes were conducted meticulously, utilizing a subset of images derived from the COCO dataset (2017 version). The performance evaluation of the CNNs involved accuracy comparison, achieved through the computation of a confusion matrix. It is noteworthy that the methodology incorporated insights derived from the source code of the DLStudio v2.3.3 module [1], as well as the previous year solution [3].

2 ExperimentsWithCIFAR

To gain a foundational understanding of training and testing classification networks, we commenced by executing the script associated with **playing_with_cifar10.py** found in the Examples directory within the downloaded DLStudio-2.2.3 folder. This script provided an insightful demonstration of the construction of neural networks, incorporating diverse layers such as convolutional layers with activation functions, pooling layers, and concluding with linear layers to generate outputs prior to loss calculation. Parts of the script like creating the Dataloader, training and validating the network and finding the number of parameters were incorporated into the solutions of HW4. A snippet of the sample output of the execution of this script is shown in Figure 1.

3 Programming Task Execution and Outputs

3.1 Using COCO to Create Your Own Image Classification Dataset

For this assignment, the 2017 version of the COCO dataset was utilized, necessitating the download of the 2017 annotation zip (**annotations_trainval2017.zip**)file from the MS-COCO website [2]. This file, containing a JSON file named **instances train2017.json**, provided comprehensive details about the images present in the **train2017.zip** file, including image IDs, classes, and other relevant information.

To streamline the dataset, five provided specific categories – 'dog,' 'boat,' 'cake,' 'couch,' and 'motorcycle' – were selected for analysis. Subsequently, the first 1500 image URLs from each category were allocated for training, while the last 500 were reserved for validation. Images were downloaded using these URLs through the requests Python library and saved in separate directories named after each category. Additionally, the images underwent downsampling to a smaller size of 64×64 using the PIL (pillow) library before being saved.

Figures 3 and 2 and 2 showcase 3 samples from from each of the 5 classes from the training and validation sets.

```
Overall accuracy of the network on the 10000 test images: 51 %


Displaying the confusion matrix:

           plane    car   bird    cat   deer    dog   frog  horse   ship  truck

 plane:    68.20   1.00   3.10   1.10   2.10   1.20   0.50   1.80  17.70   3.30
   car:     8.70  51.40   0.50   1.00   0.50   0.10   0.50   1.30  19.80  16.20
  bird:    14.80   1.00  36.70   4.50  14.80  12.30   3.20   6.40   5.10   1.20
   cat:     5.80   0.70   8.70  21.70  11.30  31.00   3.90   6.70   7.20   3.00
  deer:     9.70   1.30  12.50   2.60  44.40   8.40   3.20  13.80   3.50   0.60
   dog:     3.00   0.70   7.60   9.70   6.80  58.10   1.50   8.70   2.70   1.20
  frog:     2.60   4.90   7.30   5.20  24.20   6.10  38.00   5.30   4.00   2.40
 horse:     3.90   0.20   3.90   4.00   6.70  11.30   0.50  65.20   1.60   2.70
  ship:    10.50   2.40   0.50   1.00   0.70   1.20   0.10   1.60  80.20   1.80
 truck:     7.30  11.60   1.00   1.90   0.90   1.50   0.50   4.50  18.90  51.90
```

Figure 1: ExperimentsWithCIFAR

To efficiently handle and process these images, a custom **CocoDataset** class was developed based on the **torch.utils.data.Dataset** class. This class facilitated the loading of images from each category directory after applying necessary transformations. Finally, a Dataloader (similar to the **playing_with_cifar10.py** was instantiated to encapsulate the dataset, enabling the processing of images in batches of 128 during both training and validation phases.

3.2 Image Classification using CNNs – Training and Validation

3.2.1 CNN Task 1

For this particular task, a CNN, referred to as Net1, was constructed in accordance with the specifications outlined in the assignment. Net1 is characterized by its simplicity, featuring two convolutional layers, each accompanied by a max-pooling layer, culminating in two linear layers. The convolutional layers employ a kernel size of 3, and notably, Net1 operates without any padding. The output dimension for a convolutional layer is computed using the formula $\lfloor \frac{(W-F+2P)}{S} \rfloor + 1$ where W denotes the height or width, F represents the kernel size, P stands for padding, and S denotes the stride. This calculation ensured that the subsequent linear layers were appropriately configured to process the information derived from the convolutional layers in Net1.

Here's a simple breakdown of the HW4Net (Net1) network:

1. **Input layer:** Takes a 3-channel image of size 64x64 (B, 3, 64, 64)
2. **Conv1 (3x3, 16 filters):** Applies 16 convolutional filters of size 3x3 to extract features. (B, 3, 64, 64) → (B, 16, 62, 62)
3. **MaxPool2d (2x2):** Downsamples the feature maps by a factor of 2 in both dimensions. (B, 16, 62, 62) → (B, 16, 31, 31)
4. **Conv2 (3x3, 32 filters):** Applies 32 convolutional filters of size 3x3 to extract more complex features. (B, 16, 32, 31) → (B, 32, 29, 29)
5. **MaxPool2d (2x2):** Further downsamples the feature maps by a factor of 2. (B, 32, 29, 29) → (B, 32, 14, 14)

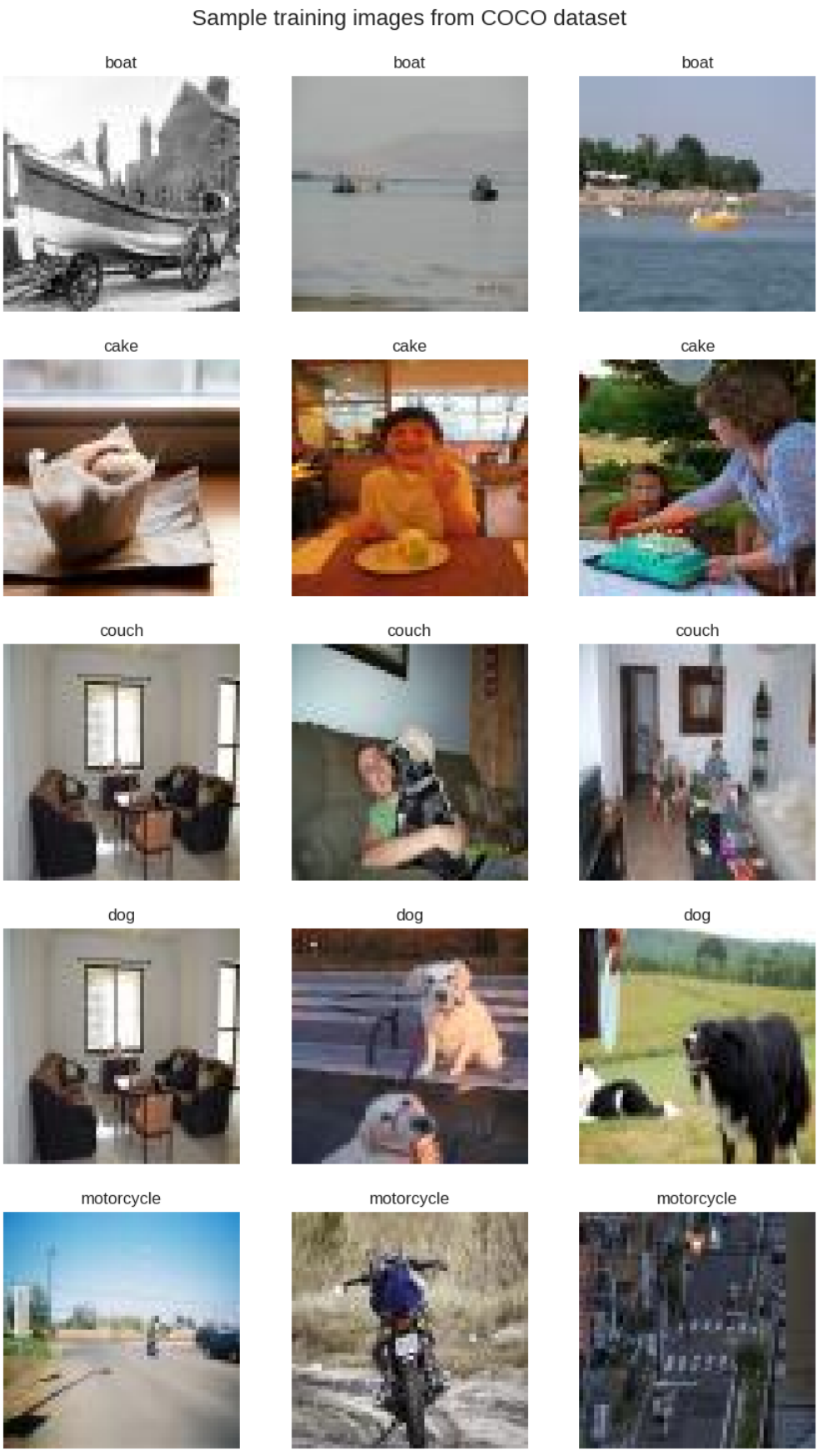

Figure 2: Sample Training Images

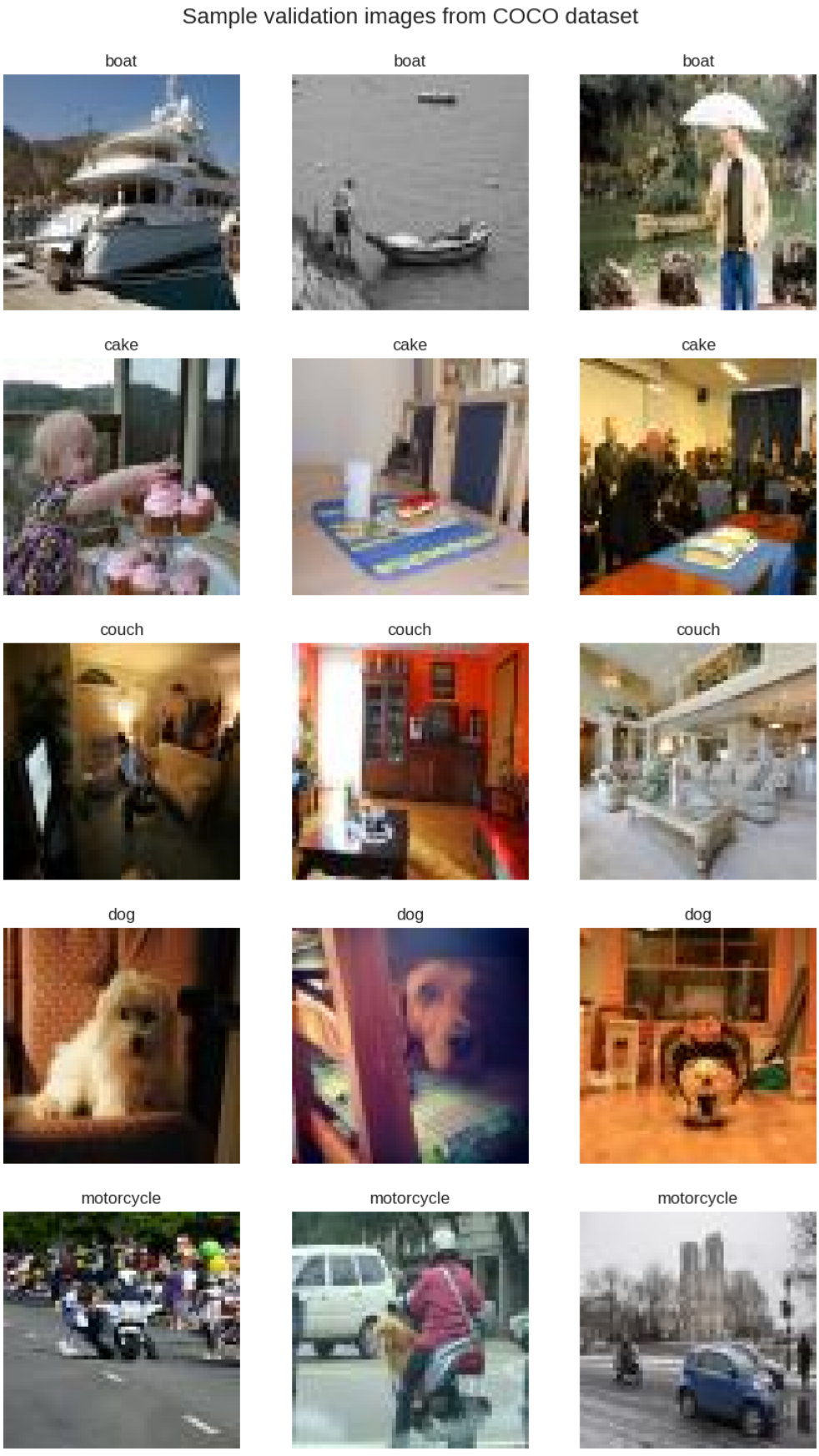

Figure 3: Sample Validation Images

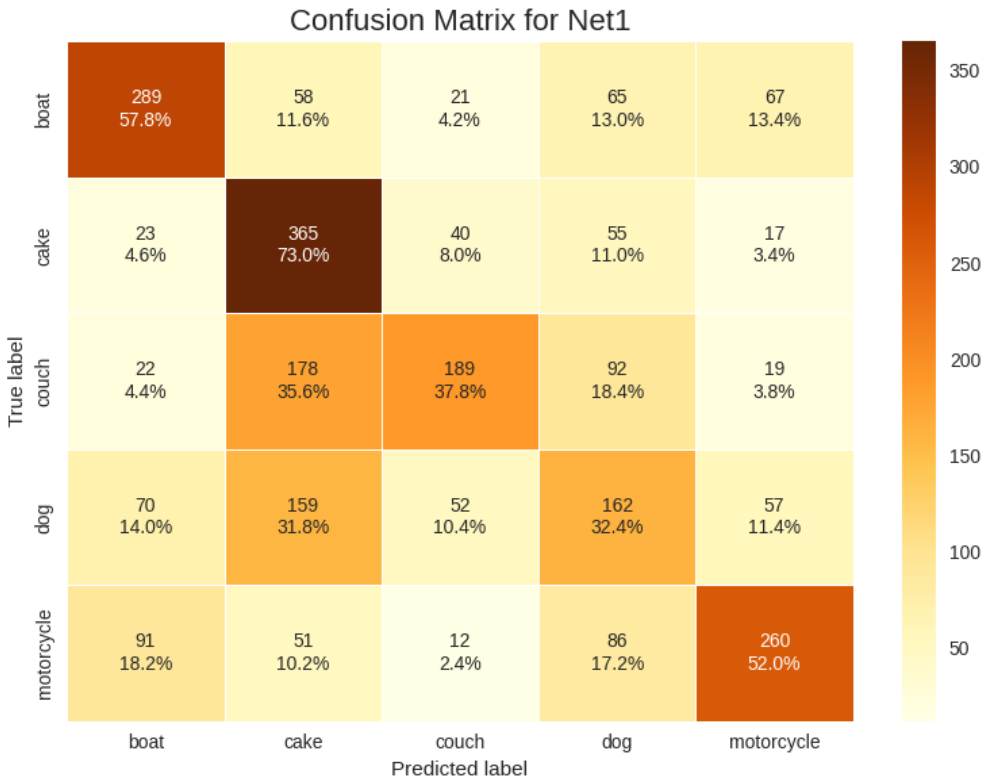

Figure 4: Net1 Confusion Matrix

6. **Flatten:** Reshapes the features into a 1D vector for the fully connected layers. (B, 32, 14, 14) → (B, 32x14x14)

7. **FC1:** A fully connected layer with 64 neurons for further feature learning. (B, 32x14x14) → (B, 64)

8. **FC2 :** The final fully connected layer, producing 5 output values (likely for 5 classes) (B, 64) → (B, 5)

9. **ReLU:** Applies ReLU activation to introduce non-linearity.

Here, B signifies the batch size, and as such, the value of **XXXX** is computed as $32 \times 14 \times 14 = 6272$. This value represents the flattened output size from the convolutional layers of Net1. The network underwent training for 20 epochs employing the provided training routine. The Adam optimizer, with parameters $\beta_1 = 0.9$ and $\beta_2 = 0.999$, and a learning rate of 0.001, was utilized for optimization purposes. This choice of optimizer and learning rate was aimed at enhancing the network's performance during the training process, taking insights from HW3.

To assess the effectiveness of Net1, a validation routine **validate_net** was realised. In this process, predicted labels were obtained through model evaluation on the downloaded validation dataset. During the validation loop, the function iterates through the validation images, calculating the loss for each batch. Ground truth labels, predicted labels, and input images are collected for subsequent analysis.Subsequently, these predictions were employed to compute the confusion matrix using the functions **calc_confusion_matrix** and **plot_conf_mat** functions.The visual representation of the confusion matrix for Net1 is presented in Figure 4.

3.2.2 CNN Task 2

Another neural network called Net2, similar to Net1 was created. The difference is that Net2 has convolutional layers with a padding value of 1. This padding helps keep the input and output image shapes consistent throughout the network. The 'padding' parameter was set to 1 in each convolutional layer (torch.nn.Conv2d). This ensures that when the network processes images, it maintains important spatial details. Similar approach as Net1 was followed to calculate the input and output dimensions for the linear layers in Net2. This consistent method allows us to smoothly connect the convolutional and linear layers, making the network effective in learning image features.

While the overall structure of both models is similar, with 2 convolutional layers, 2 pooling layers, 2 fully connected layers and ReLU activation after convolutional and fully connected layers, the key differences are

1. **Padding in convolutional layers:** Both conv1 and conv2 layers in Net2 have padding=1. As explained, this ensures consistent output size after convolutions, making feature map dimensions easier to predict.

2. **Dynamic input size calculation:** The fc1 layer in Net2 dynamically calculates its input size using the **_calculate_fc_input_size** helper function. This makes the model adaptable to different input dimensions and avoids hardcoding layer sizes.

The sizes of the output of the different layers now will be

1. **Input layer:** (B, 3, 64, 64)
2. **Conv1:** (B, 3, 64, 64) → (B, 16, 64, 64)
3. **MaxPool2d (2x2):** (B, 16, 64, 64) → (B, 16, 32, 32)
4. **Conv2:** (B, 16, 32, 32) → (B, 32, 32, 32)
5. **MaxPool2d (2x2):** (B, 32, 32, 32) → (B, 32, 16, 16)
6. **Flatten:** (B, 32, 16, 16) → (B, 32x16x16)
7. **FC1:** (B, 32x16x16) → (B, 64)
8. **FC2 :** (B, 64) → (B, 5)
9. **ReLU:** Does not change image dimension.

The epoch size and the hyperparameter values for Net2 are kept the same as that for Net1. The visual representation of the confusion matrix for Net2 is presented in Figure 5.

3.2.3 CNN Task 3:

Net3, was derived from Net1 by introducing a set of 10 intermediary convolutional layers. These additional layers, featuring a kernel size of 3 and a padding of 1, were strategically positioned between the first two convolutional layers and the last two linear layers. Each of these newly added convolutional layers had 32 output channels and 32 input channels. Remarkably, the choice of a kernel size of 3, coupled with a padding of 1 in these additional layers, ensured that the shape of the output remained unchanged compared to the input. Consequently, the value of **XXXX** for Net3 aligns with that of Net2, amounting to 32x16x16 = 8192. This consistency in output size reflects the successful preservation of spatial information throughout the network, illustrating

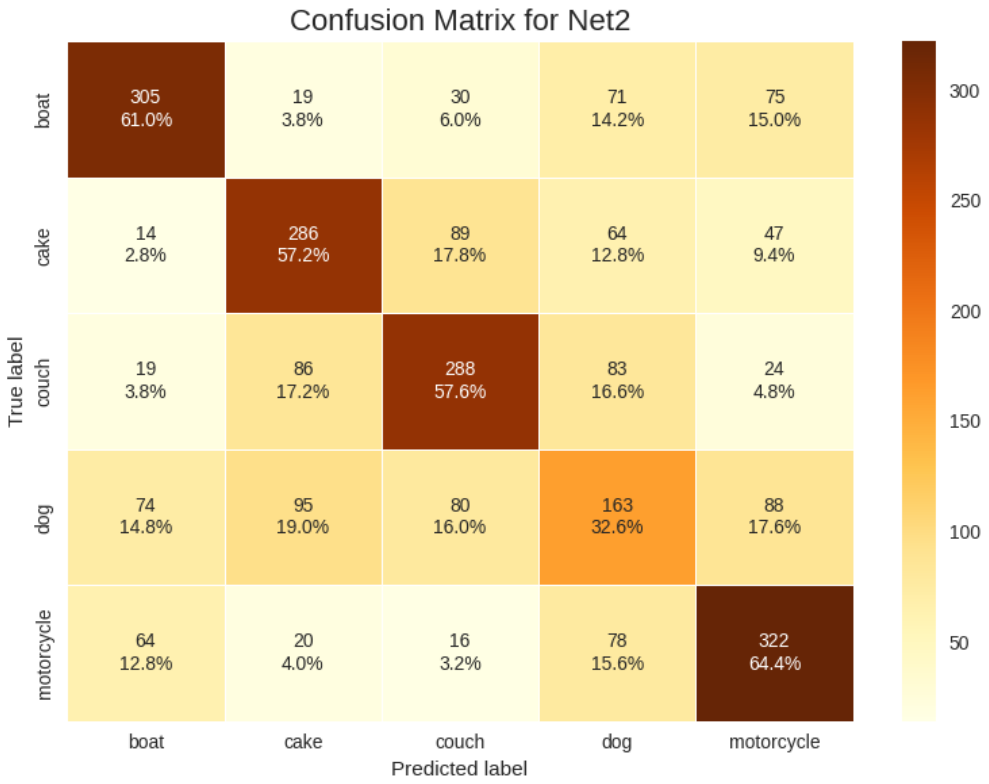

Figure 5: Net2 Confusion Matrix

Net3's capacity to handle complex image features while maintaining compatibility with the input dimensions.

While the general structure (conv-pool-conv-pool-flatten-fc-fc) is still present in Net3, but with the addition of the 10 Conv2d layers, the key differences with respect to previous Net1 and Net2 are:

1. **ModuleList for convolutional layers:** Net3 uses nn.ModuleList to store a sequence of convolutional and pooling layers, making it easier to manage multiple layers with varying configurations.

2. **10 additional Conv2d layers:** Net3 has 10 consecutive convolutional layers with 32 filters each, added after the initial 2 conv-pool blocks. This significantly deepens the network, potentially increasing its representational capacity.

3. **Looping in the forward pass:** The forward method iterates through the **conv_list** using a loop, applying layers sequentially. This flexibility accommodates the varying layers within **conv_list**.

The epoch size (=20) and the hyperparameter values for Net3 are kept the same as that for Net1 and Net2. The visual representation of the confusion matrix for Net3 is presented in Figure 6.

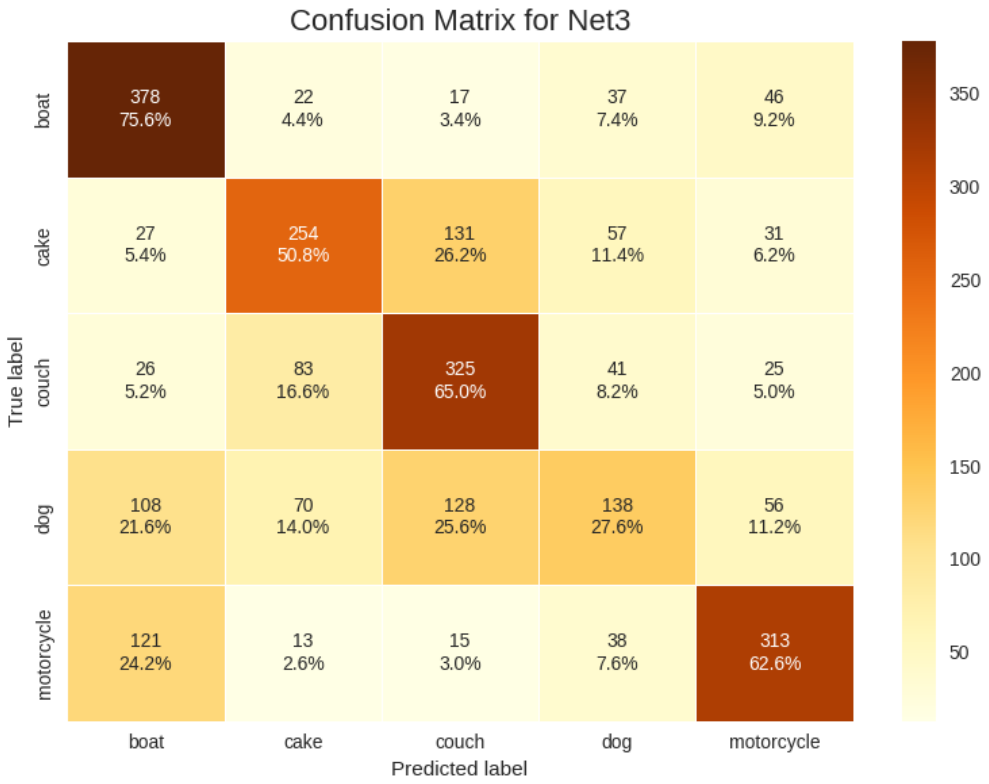

Figure 6: Net3 Confusion Matrix

3.2.4 Additional Network (Net4) with residual connection

An additional Net4 was realised to potentially overcome the vanishing gradient problem in Net3. It involves a residual connection to facilitate the flow of information directly across layers, often improving training of deeper networks.

In Net4, before the stack of Conv3 layers, the feature maps **x** are stored as **x_residual**. After the 10 Conv3 layers, Tte output **x** is added to the original **x_residual** using element-wise addition. The residual connection is a technique often used in deep neural networks to improve performance and training stability. It involves adding the input of a block of layers to its output, creating a "shortcut" for information to flow through the network.

For example, imagine **x** starts as a 3-dimensional tensor representing an image (e.g., shape (64, 64, 3)). The convolutional and pooling layers extract features and reduce dimensionality. Before the 10 iterations of self.conv3, **x_residual** stores a copy of this intermediate representation. After the iterations, **x** has undergone further transformations. The addition $\mathbf{x} = \mathbf{x} + \mathbf{x\_residual}$ merges the original features **x_residual** with the processed features **x**, potentially preserving valuable information, promoting feature reuse and mitigating vanishing gradients. This can be seen by comparing the training loss for Net3 vs Net4.

The visual representation of the confusion matrix for Net4 is presented in Figure 7.

3.2.5 Parameter calculation

Net1

1. **For Conv1**:

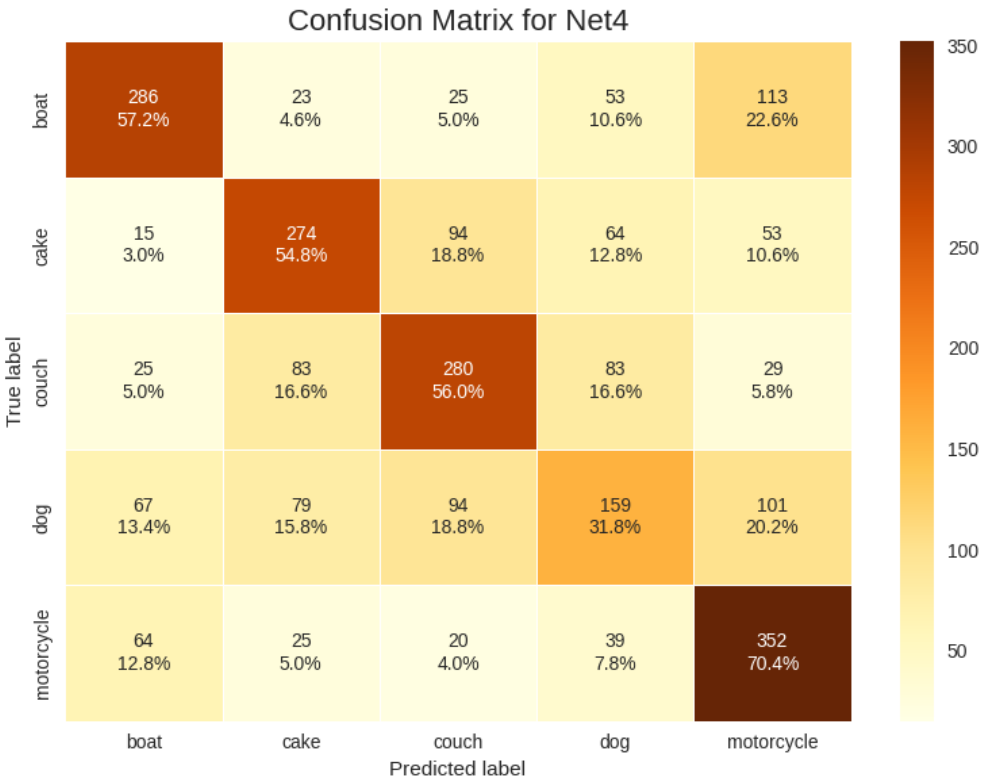

Figure 7: Net4 Confusion Matrix

- Input channels (3) multiplied by output channels (16) multiplied by the kernel size (3x3) gives the weights: $3 \times 16 \times 3 \times 3 = 432$.
- Adding the biases (16) gives a total of $432 + 16 = 448$ parameters.

2. **For Conv2:**

   - Input channels (16) multiplied by output channels (32) multiplied by the kernel size (3x3) gives the weights: $16 \times 32 \times 3 \times 3 = 4608$.
   - Adding the biases (32) gives a total of $4608 + 32 = 4640$ parameters..

3. **For FC1:**

   - The input size (6272) multiplied by the output size (64) gives the weights: $6272 \times 64 = 401408$.
   - Adding the biases (64) gives a total of $401408 + 64 = 401472$ parameters.

4. **For FC2:**

   - The input size (64) multiplied by the output size (5) gives the weights: $64 \times 5 = 320$.
   - Adding the biases (5) gives a total of $320 + 5 = 325$ parameters.

Summing up all the parameters from Conv1, Conv2, FC1, and FC2: $448 + 4640 + 401472 + 325 = 406885$ parameters in total for the HW4Net architecture.

Net2 Along the similar lines as Net1, we have

1. **For Conv1 :** (3×16×3×3)+16=448
2. **For Conv2 :** (16×32×3×3)+32=4640
3. **For FC1 :** (8192×64)+64=524352
4. **For FC2:** (64×5)+5=325

Summing up all the parameters from Conv1, Conv2, FC1, and FC2: 448 + 4640 + 524352 + 325 = 529765 parameters in total for the Net2 architecture.

Net3 Along the similar lines as Net2, we have

1. **For Conv1 :** (3×16×3×3)+16=448
2. **For Conv2 :** (16×32×3×3)+32=4640
3. **For 10 Conv3 :** (16×32×3×3)+32=92480
4. **For FC1 :** (8192×64)+64=524352
5. **For FC2:** (64×5)+5=325

Summing up all the parameters from Conv1, Conv2, 10 Conv3, FC1, and FC2: 448 + 4640 + 92480 + 524352 + 325 = 622245 parameters in total for the Net3 architecture.

Net4 is just adding the output of the first two conv layers to the output of the stacked conv layers. Hence it does not add any additional parameters to the network.

3.2.6 Training Loss and Comparison Table

he training loss profiles for the four aforementioned networks are visually depicted in Figure 8, offering insights into their learning behavior and convergence during the training process. Additionally, a comprehensive performance and parameter comparison table is presented in Figure 9. This table serves as a valuable reference for evaluating and contrasting the accuracy and model parameters. Together, these visualizations provide a succinct overview of the training dynamics and comparative aspects of the networks, aiding in the assessment and selection of the most suitable model for the task at hand.

3.2.7 Discussion

1. Does adding padding to the convolutional layers make a difference in classification performance?

    - *Yes, adding padding to convolutional layers can significantly impact classification performance. The evidence from the provided accuracy values for Net1 and Net2 supports this notion. Net2, which incorporates padding in its convolutional layers, demonstrates an improved accuracy of 54.56% compared to Net1, which has an accuracy of 50.6%. The introduction of padding in convolutional layers is crucial because it helps preserve spatial information at the borders of an image during the convolutional operations. Without padding, the dimensions of the feature maps gradually decrease through convolution and pooling layers, potentially leading to loss of valuable edge information. In contrast, by*

using padding, the spatial dimensions are maintained, allowing the network to capture more comprehensive features, especially at the image borders.The enhanced performance of Net2 might be attributed to its ability to retain important details at the edges of images such as couch and motorcycle

2. Do you observe vanishing gradient in Net3?

   - *Yes, it appears that Net3 is experiencing the issue of vanishing gradients. Despite having a comparable accuracy to other networks, Net3 exhibits the highest training loss among the four networks. The elevated training loss suggests that during the backpropagation process, the gradients of the network's parameters may be diminishing, making it challenging for the model to learn effectively. Vanishing gradients occur when the gradients of the loss with respect to the parameters become very small, causing minimal updates to the weights during training. One potential reason for vanishing gradients in Net3 could be the deep stack of convolutional layers, especially considering the presence of multiple layers (10) with the ReLU activation function. The ReLU function, while popular, can sometimes lead to dead neurons and vanishing gradients if not carefully handled.As discussed, Net4, employing a residual network (ResNet) architecture, may offer a potential solution to address the vanishing gradient problem. Residual connections allow for the direct flow of gradients during backpropagation, mitigating the vanishing gradient issue and enabling the effective training of deeper networks.*

3. Which CNN is the best performer?

   - *Among the three networks, Net3 emerges as the best performer based on classification accuracy. Net1 achieves an accuracy of 50.6%, Net2 improves to 54.56%, while Net3 achieves the highest accuracy at 56.32%. The enhanced performance of Net3 can be attributed to additional 10 intermediary convolutional layers, allowing for more intricate feature extraction. This increased depth could enable the network to capture more complex patterns in the data. Also padding in the convolutional layers could prevent the loss of edge details*

4. Which class(es) is/are more difficult to correctly differentiate?

   - *The analysis of the confusion matrix reveals that the class "Dog" is more challenging to correctly differentiate compared to other classes. The confusion matrix indicates that instances of dogs are being misclassified as "cake," "couch," or even "boat.". This could be because dogs and objects like cakes and couches could share the similar background of a home setting.*

5. What is one thing that you propose to make the classification performance better?

   - *One potential solution could be Data Augmentation. We can introduce variations in the training dataset by applying transformations like rotation, scaling, greying and flipping to generate additional diverse images. This helps the model become more robust and improves its ability to generalize to different scenarios.*

4 Conclusion

In this HW, we investigated Convolutional Neural Networks (CNNs) within the PyTorch framework, utilizing the MS-COCO dataset for image classification. Through the creation and analysis of

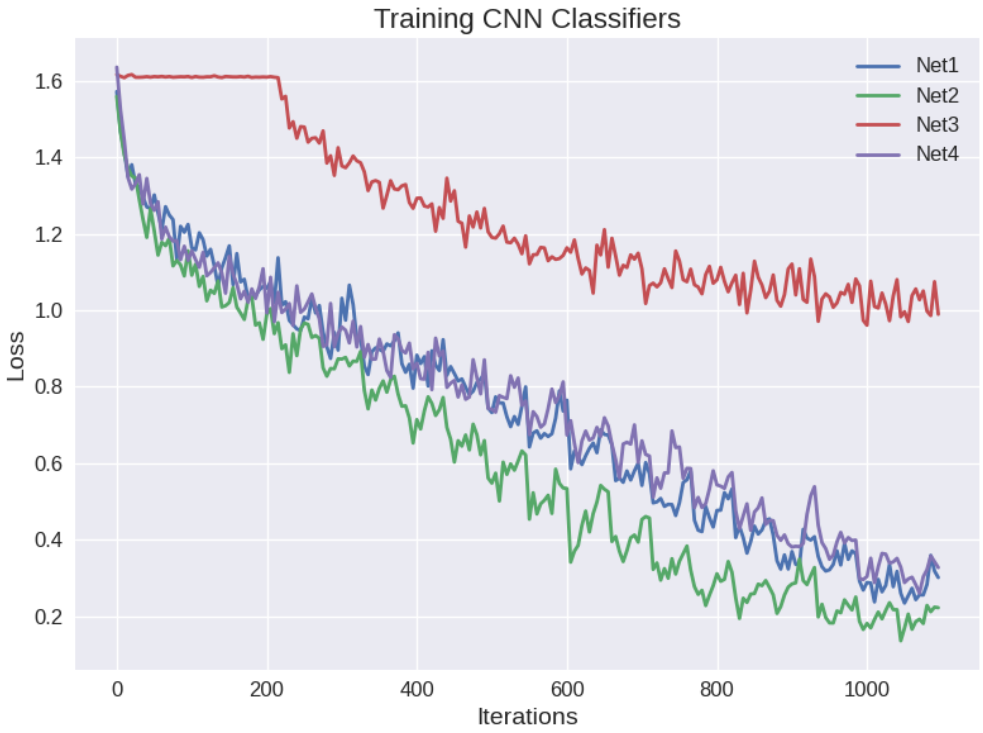

Figure 8: Training Loss

	Net Name	Number of Parameters	Classification Accuracy
0	Net1	406885	0.5060
1	Net2	529765	0.5456
2	Net3	622245	0.5632
3	Net4	622245	0.5404

Figure 9: Performance Comparison Table

three distinct CNN architectures, each tailored with convolutional, pooling, and linear layers, we explored the intricate process of training and validating these networks. We evaluated the CNNs' performance through the computation of a confusion matrix, providing valuable insights into the accuracy of image classifications across various classes. In our exploration, we learned about the importance of using padding in convolutional layers. We also took a close look at the calculation of the number of parameters in our model, which are the elements that the network adjusts during training. Another challenge we tackled was the issue of vanishing gradients, where information gets lost during training. We explored ways to address this problem, specifically using the residual network.

5 Source code

```
# -*- coding: utf-8 -*-
"""HW4.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1ql-hAMqItDuTMzTN7X8qfJYssZVMDj35
"""

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

# Commented out IPython magic to ensure Python compatibility.
# %cd /content/drive/My Drive/hw4_data/

!wget -O DLStudio-2.2.3.tar.gz \
    https://engineering.purdue.edu/kak/distDLS/DLStudio-2.2.3.tar.gz?download

!tar -xvf DLStudio-2.2.3.tar.gz

# Commented out IPython magic to ensure Python compatibility.
# %cd DLStudio-2.2.3

!pip install pymsgbox

!python setup.py install

!pip install pycocotools

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
from pycocotools.coco import COCO

!pwd

!python '/content/drive/My Drive/hw4_data/DLStudio-2.2.3/Examples/
                                        playing_with_cifar10.py'
```

```
seed = 60146
random.seed(seed)
np.random.seed(seed)

from DLStudio import *

!mkdir /content/drive/MyDrive/hw4_data/coco
!wget --no-check-certificate http://images.cocodataset.org/annotations/
                                            annotations_trainval2017.zip \
    -O /content/drive/MyDrive/hw4_data/coco/annotations_trainval2017.zip

!unzip /content/drive/MyDrive/hw4_data/coco/annotations_trainval2017.zip -d /
                                            content/drive/MyDrive/hw4_data/coco/

!mkdir /content/drive/MyDrive/hw4_data/coco/train2017
!mkdir /content/drive/MyDrive/hw4_data/coco/val2017
!mkdir /content/drive/MyDrive/hw4_data/saved_models

# ImageDownloader class for downloading COCO images
class ImageDownloader():
  def __init__(self, root_dir, annotation_path, classes, imgs_per_class):
    self.root_dir = root_dir
    self.annotation_path = annotation_path
    self.classes = classes
    self.images_per_class = imgs_per_class
    self.class_dir = {}
    self.coco = COCO(self.annotation_path)

  #Create directories same as category names to save images
  def create_dir(self):
    """Creates directories for each class."""
    for c in self.classes:
        dir_path = os.path.join(self.root_dir, c)
        os.makedirs(dir_path, exist_ok=True)  # Create only if not existing
        self.class_dir[c] = dir_path

  # Download images
  def download_images(self, download = True, val = False):
    img_paths = {}
    for c in tqdm(self.classes):
      class_id = self.coco.getCatIds(c)
      img_id = self.coco.getImgIds(catIds=class_id)
      imgs = self.coco.loadImgs(img_id)
      img_paths[c] = []

      # Get indices for training or validation split
      indices = np.arange(self.images_per_class) if not val else np.arange(len(
                                                  imgs) - self.images_per_class, len(
                                                  imgs))

      for i in indices:
        img_path = os.path.join(self.root_dir, c, imgs[i]['file_name'])
        if download:
          self.download_image(img_path, imgs[i]['coco_url'])
          self.resize_image(img_path)
        img_paths[c].append(img_path)

    return img_paths
```

```
  #Download image from URL using requests
  def download_image(self, path, url):
    try:
      img_data = requests.get(url).content
      with open(path, 'wb') as f:
          f.write(img_data)
      return True
    except Exception as e:
      print(f"Caught exception: {e}")
    return False

  # Resize image
  def resize_image(self, path, size = (64, 64)):
    im = Image.open(path)
    if im.mode != "RGB":
      im = im.convert(mode="RGB")
    im_resized = im.resize(size, Image.BOX)
    im_resized.save(path)

#First set to true to download images, later download=False to not re-download
classes = ['boat', 'cake', 'couch', 'dog', 'motorcycle']
train_imgs_per_class = 1500
val_imgs_per_class = 500

# Download training images
train_downloader = ImageDownloader('/content/drive/MyDrive/hw4_data/coco/train2017
                                   ',
                '/content/drive/MyDrive/hw4_data/coco/annotations/
                                                    instances_train2017.json'
                                                    ,
                classes, train_imgs_per_class)
train_downloader.create_dir()
train_img_paths = train_downloader.download_images(download = False)

# Download validation images
val_downloader = ImageDownloader('/content/drive/MyDrive/hw4_data/coco/val2017',
                '/content/drive/MyDrive/hw4_data/coco/annotations/
                                                    instances_train2017.json'
                                                    ,
                classes, val_imgs_per_class)
val_downloader.create_dir()
val_img_paths = val_downloader.download_images(download = False, val = True)

plt.style.use('seaborn')  # Consistent and visually appealing color scheme

# Plot sample training images
fig, axes = plt.subplots(5, 3, figsize=(9, 15))
fig.suptitle('Sample training images from COCO dataset', fontsize=16)

for i, cls in enumerate(classes):
    for j, path in enumerate(train_img_paths[cls][:3]):
        im = Image.open(path)
        axes[i][j].imshow(im)
        axes[i][j].set_title(cls)
        axes[i][j].axis('off')  # Remove axes for cleaner visualization

plt.tight_layout(pad=2)  # Adjust spacing between subplots
plt.show()
```

```
# Print a line separation
print("-" * 100)  # Print 80 hyphens to create a visual separator

# Plotting sample validation images
fig, axes = plt.subplots(5, 3, figsize=(9, 15))
fig.suptitle('Sample validation images from COCO dataset', fontsize=16)

for i, cls in enumerate(classes):
    for j, path in enumerate(val_img_paths[cls][:3]):
        im = Image.open(path)
        axes[i][j].imshow(im)
        axes[i][j].set_title(cls)
        axes[i][j].axis('off')  # Remove axes for cleaner visualization

plt.tight_layout(pad=2)  # Adjust spacing between subplots
plt.show()

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 = 64
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
                                                  num_workers=2)
val_data_loader = torch.utils.data.DataLoader(val_dataset,
                                                batch_size=128,
                                                shuffle=False,  # No shuffling
                                                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/hw4_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
import torch.nn.functional as F

class HW4Net(nn.Module):
    """Neural network model 1 for HW4."""

    def __init__(self):
        super().__init__()  # Use super() for conciseness

        # Convolutional layers
        self.conv1 = nn.Conv2d(3, 16, 3)  # Add padding for consistent output size
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(16, 32, 3) # default padding = 0

        # Fully connected layers
        self.fc1 = nn.Linear(6272, 64)  # Consider calculating input size
                                                dynamically
```

```
        self.fc2 = nn.Linear(64, 5)

    def forward(self, x): # (B, 3, 64, 64)
        x = self.pool(F.relu(self.conv1(x)))  # (B, 16, 62, 62) ->  (B, 16, 31, 31
                                                ) #(input_size + 2*padding -
                                                window_size)+1
        x = self.pool(F.relu(self.conv2(x))) # (B, 32, 29, 29) ->  (B, 32, 14, 14)
        x = x.view(x.shape[0], -1) # (B, 6272)
        x = F.relu(self.fc1(x)) #(B, 64)
        x = self.fc2(x) # (B, 5)
        return x

# Use GPU if available
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Initialize model, optimizer, and criterion
net = HW4Net().to(device)  # Move model to device
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net.parameters(), lr=1e-3, betas=(0.9, 0.999))

# Training
epochs = 20
display_interval = 5
net1_losses = train_net(device, net, optimizer=optimizer, criterion=criterion,
                        data_loader=train_data_loader, model_name='net1',
                        epochs=epochs, display_interval=display_interval)

# Validation
save_path = '/content/drive/MyDrive/hw4_data/saved_models/net1.pth'
imgs, actual, predicted = validate_net(device, net, val_data_loader, model_path=
                                       save_path)

# Calculate confusion matrix and accuracy
conf_mat = calc_confusion_matrix(5, actual, predicted)
print(conf_mat)
accuracy1 = np.trace(conf_mat) / float(np.sum(conf_mat))
print(f"Accuracy: {accuracy1:.4f}")

# Plot confusion matrix
plot_conf_mat(conf_mat, classes, 'Net1')

import torch.nn as nn
import torch.nn.functional as F

class Net2(nn.Module):
    """Neural network model with improved structure and clarity."""

    def __init__(self):
        super().__init__()  # Use super() for conciseness

        #Convolutional layers with padding for consistent output size
        self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)

        # Fully connected layers with dynamic input size calculation
        self.fc1 = nn.Linear(self._calculate_fc_input_size(), 64)  # Calculate
                                                input size dynamically
        self.fc2 = nn.Linear(64, 5)
```

```
    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(x.shape[0], -1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

    def _calculate_fc_input_size(self):
        """Helper function to calculate input size for the first fully connected
                                                    layer."""
        with torch.no_grad():
            x = torch.randn(1, 3, 64, 64)  # Assume input size of 3x64x64
            x = self.pool(F.relu(self.conv1(x))) #(B, 3, 64, 64) ->  (B, 16, 32,
                                                    32)
            x = self.pool(F.relu(self.conv2(x))) #(B, 16, 32, 32) ->  (B, 32, 16,
                                                    16)
            x = x.view(x.shape[0], -1) #(B,8192)
            return x.shape[1]

# Initialize Net2
net2 = Net2().to(device)  # Move model to device immediately

# Criterion and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net2.parameters(), lr=1e-3, betas=(0.9, 0.999))

# Training
epochs = 20
display_interval = 5
net2_losses = train_net(device, net2, optimizer=optimizer, criterion=criterion,
                        data_loader=train_data_loader, model_name='net2',
                        epochs=epochs, display_interval=display_interval)

# Validation
save_path = '/content/drive/MyDrive/hw4_data/saved_models/net2.pth'
imgs, actual, predicted = validate_net(device, net2, val_data_loader, model_path=
                                        save_path)

# Calculate confusion matrix and accuracy
conf_mat = calc_confusion_matrix(5, actual, predicted)
print(conf_mat)
accuracy2 = np.trace(conf_mat) / float(np.sum(conf_mat))
print(f"Accuracy: {accuracy2:.4f}")

# Plot confusion matrix
plot_conf_mat(conf_mat, classes, 'Net2')

import torch.nn as nn
import torch.nn.functional as F

class Net3(nn.Module):
    def __init__(self):
        super().__init__()

        # Convolutional layers using ModuleList
        self.conv_list = nn.ModuleList([
            nn.Conv2d(3, 16, 3, padding=1),
```

```
            nn.MaxPool2d(2, 2),
            nn.Conv2d(16, 32, 3, padding=1),
            nn.MaxPool2d(2, 2),
            *[nn.Conv2d(32, 32, 3, padding=1) for _ in range(10)]  # Define all 10
                                                                    Conv2d layers
        ])

        # Fully connected layers
        self.fc1 = nn.Linear(self._calculate_fc_input_size(), 64)  # Input size
                                                                     might need adjustment
        self.fc2 = nn.Linear(64, 5)

    def forward(self, x):
        for i, layer in enumerate(self.conv_list):
            if isinstance(layer, nn.Conv2d):
                x = F.relu(layer(x))
            else:  # Assuming it's a MaxPool2d layer
                x = layer(x)

            # Apply the 10 Conv2d layers after the initial 4 layers
            if i == 4:
                for conv_layer in self.conv_list[5:]:  # Iterate through layers 5
                                                         to 14
                    x = F.relu(conv_layer(x))
                break  # Exit the outer loop

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

    def _calculate_fc_input_size(self):
        """Helper function to calculate input size for the first fully connected
                                                    layer."""
        with torch.no_grad():
            x = torch.randn(1, 3, 64, 64)  # Assume input size of 3x64x64
            x = nn.MaxPool2d(2, 2)(F.relu(self.conv_list[0](x))) #(B, 3, 64, 64)
                                                                   ->  (B, 16, 32, 32)
            x = nn.MaxPool2d(2, 2)(F.relu(self.conv_list[2](x))) #(B, 16, 32, 32)
                                                                   ->  (B, 32, 16, 16)
            x = x.view(x.shape[0], -1) #(B,8192)
            return x.shape[1]

import torch.nn as nn
import torch.nn.functional as F

class Net4(nn.Module):
    """Neural network model with enhanced structure and potential for
                                                regularization."""

    def __init__(self):
        super().__init__()

        # Convolutional layers with padding for consistent output size
        self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
        self.conv3 = nn.Conv2d(32, 32, 3, padding=1)  # Add padding for consistent
                                                        output
```

```
        # Fully connected layers with dynamic input size calculation
        self.fc1 = nn.Linear(self._calculate_fc_input_size(), 64)  # Calculate
                                                    input size dynamically
        self.fc2 = nn.Linear(64, 5)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))

        # Residual connection for potential improvement
        x_residual = x  # Store residual for later addition
        for _ in range(10):
            x = F.relu(self.conv3(x))
        x = x + x_residual  # Add residual connection

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

    def _calculate_fc_input_size(self):
        """Helper function to calculate input size for the first fully connected
                                                    layer."""
        with torch.no_grad():
            x = torch.randn(1, 3, 64, 64)  # Assume input size
            x = self.pool(F.relu(self.conv1(x)))
            x = self.pool(F.relu(self.conv2(x)))
            x = self.conv3(x)  # Apply conv3 once for calculation
            x = x.view(x.shape[0], -1)
            return x.shape[1]

# Initialize Net3
net3 = Net3().to(device)  # Move model to device immediately

# Criterion and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net3.parameters(), lr=1e-3, betas=(0.9, 0.999))

# Training
epochs = 20
display_interval = 5
net3_losses = train_net(device, net3, optimizer=optimizer, criterion=criterion,
                        data_loader=train_data_loader, model_name='net3',
                        epochs=epochs, display_interval=display_interval)

# Validation
save_path = '/content/drive/MyDrive/hw4_data/saved_models/net3.pth'
imgs, actual, predicted = validate_net(device, net3, val_data_loader, model_path=
                                       save_path)

# Calculate confusion matrix and accuracy
conf_mat = calc_confusion_matrix(5, actual, predicted)
print(conf_mat)
accuracy3 = np.trace(conf_mat) / float(np.sum(conf_mat))
print(f"Accuracy: {accuracy3:.4f}")

# Plot confusion matrix
plot_conf_mat(conf_mat, classes, 'Net3')
```

```
# Initialize Net3
net4 = Net4().to(device)  # Move model to device immediately

# Criterion and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net4.parameters(), lr=1e-3, betas=(0.9, 0.999))

# Training
epochs = 20
display_interval = 5
net4_losses = train_net(device, net4, optimizer=optimizer, criterion=criterion,
                        data_loader=train_data_loader, model_name='net4',
                        epochs=epochs, display_interval=display_interval)

# Validation
save_path = '/content/drive/MyDrive/hw4_data/saved_models/net4.pth'
imgs, actual, predicted = validate_net(device, net4, val_data_loader, model_path=
                                       save_path)

# Calculate confusion matrix and accuracy
conf_mat = calc_confusion_matrix(5, actual, predicted)
print(conf_mat)
accuracy4 = np.trace(conf_mat) / float(np.sum(conf_mat))
print(f"Accuracy: {accuracy4:.4f}")

# Plot confusion matrix
plot_conf_mat(conf_mat, classes, 'Net4')

# 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
x = np.arange(len(net1_losses)) * display_interval
plt.plot(x, net1_losses, label='Net1', linewidth=2)  # Adjust linewidth
plt.plot(x, net2_losses, label='Net2', linewidth=2)
plt.plot(x, net3_losses, label='Net3', linewidth=2)
plt.plot(x, net4_losses, label='Net4', linewidth=2)

# Customize plot elements
plt.title('Training CNN Classifiers', 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()

import pandas as pd

# Calculate model parameters
net1_params = sum(p.numel() for p in net.parameters() if p.requires_grad)
```

```
net2_params = sum(p.numel() for p in net2.parameters() if p.requires_grad)
net3_params = sum(p.numel() for p in net3.parameters() if p.requires_grad)


# Create the table
data = {
    'Net Name': ['Net1', 'Net2', 'Net3', 'Net4'],
    'Number of Parameters': [net1_params, net2_params, net3_params, net3_params],
    'Classification Accuracy': [accuracy1, accuracy2, accuracy3, accuracy4]  #
                                                Replace with actual accuracy values
}

table = pd.DataFrame(data)
print(table.to_string())
```

References

[1] URL https://engineering.purdue.edu/kak/distDLS/

[2] URL https://cocodataset.org/#download

[3] URL http://tinyurl.com/34k7jt5n