ECE 60146 Deep Learning **Name:** *Nikita Ravi (ravi30)*
Homework 4
02/20/2023

1 Introduction

The objective of this assignment was to classify images using shallow and deep neural network architectures. A total of three different architectures were implemented. The image dataset is a **COCO** dataset from 2014 and a downloader script was implemented to obtain all the training and validation images.

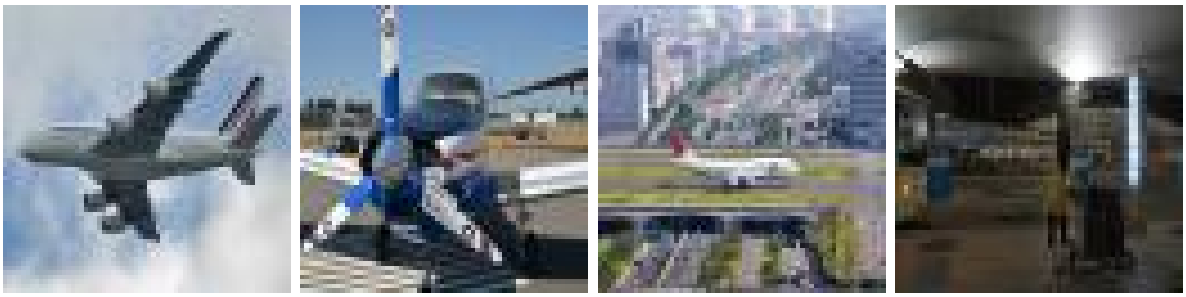
2 Methodology

2.1 COCO Downloader

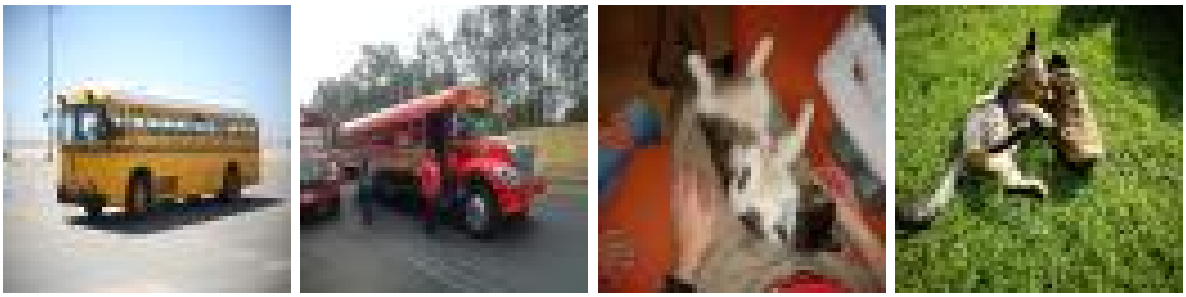
With the help of **pycocotools**, we created a downloader script to gather all the COCO training and validation dataset from 2014 and organize it locally in a specific file structure. The images are shown in the following section. The methodology to achieve this is as follows:

1. Iterate through the list of category names and create directories with the respective category names to store the associated images
2. Get the id associated to the category and then using this id, get all the image ids associated to the category id
3. Load all the images using the derived image ids and save it in its respective category directory

2.1.1 Images from Downloader



(a) Airplane 1 (b) Airplane 2 (c) Airplane 3 (d) Bus 1



(e) Bus 2 (f) Bus 3 (g) Cat 1 (h) Cat 2

(i) Cat 3 (j) Dog 1 (k) Dog 2 (l) Dog 3

(m) Pizza 1 (n) Pizza 2 (o) Pizza 3

Figure 1: Input Images from Five Different Classes

2.1.2 Source Code

```
# Import Libraries
from PIL import Image
```

```
from pycocotools.coco import COCO
import os
import requests

class COCODownloader:
    def __init__(self, root_path, coco_json_path, class_names, num_images_per_class):
        self.root_path = root_path
        self.coco_json_path = coco_json_path
        self.class_names = class_names
        self.num_images_per_class = num_images_per_class
        self.coco = COCO(self.coco_json_path)

    def __create_directories(self, class_name):
        path_to_class = os.path.join(self.root_path, class_name)
        if(not os.path.exists(path_to_class)):
            os.makedirs(path_to_class)

        return path_to_class

    def resize_image(self, image_path):
        try:
            image = Image.open(image_path)
            if(image.mode != "RGB"):
                image = image.convert(mode="RGB")

            resized_image = image.resize((64, 64), Image.BOX)
            resized_image.save(image_path)
            return True

        except Exception as e:
            print(e)

    def download_coco_images(self):
        for class_name in self.class_names:
            path_to_class = self.__create_directories(class_name=class_name)
            category_ids = self.coco.getCatIds(catNms=class_name)
            image_ids = self.coco.getImgIds(catIds=category_ids)

            images_from_class = self.coco.loadImgs(ids=image_ids)
            count = 0
            for image in images_from_class:
                try:
                    if(count < self.num_images_per_class):
                        image_data = requests.get(image["coco_url"])
                        path_to_image = os.path.join(path_to_class, image["file_name"])
                        with open(path_to_image, "wb") as fptr:
                            fptr.write(image_data.content)
                        if(self.resize_image(path_to_image)):
                            count += 1

                except Exception as e:
```

```
                    print(e)

def train():
    root_path = r"/Users/nikitaravi/Documents/Academics/ECE 60146/HW4/Train"
    coco_json_path = r"/Users/nikitaravi/Documents/Academics/ECE 60146/HW4/annotations/instances_train2014.json"
    class_names = ["airplane", "bus", "cat", "dog", "pizza"]
    train_images_per_class = 1500

    coco_downloader = COCODownloader(root_path, coco_json_path, class_names, train_images_per_class)
    coco_downloader.download_coco_images()


def val():
    root_path = r"/Users/nikitaravi/Documents/Academics/ECE 60146/HW4/Val"
    coco_json_path = r"/Users/nikitaravi/Documents/Academics/ECE 60146/HW4/annotations/instances_val2014.json"
    class_names = ["airplane", "bus", "cat", "dog", "pizza"]
    val_images_per_class = 500

    coco_downloader = COCODownloader(root_path, coco_json_path, class_names, val_images_per_class)
    coco_downloader.download_coco_images()

if __name__ == "__main__":
    mode = "val"
    if(mode == "train"):
        train()
    elif(mode == "val"):
        val()
```

Listing 1: COCO Downloader

2.2 Dataset Class and Dataloader

The purpose of this is to create a custom dataset class for our COCO images. The custom dataset class will store the meta information about the dataset and implement code that loads and augments the images. The transformations applied to the image is to convert it from a PIL object to a tensor object and normalize it.

2.2.1 Source Code

```
def get_images(root, category):
    category_path = os.path.join(root, category)
    image_files = [image for image in os.listdir(category_path) if image != ".DS_Store"]

    images_pil = [Image.open(os.path.join(category_path, image)).convert("RGB") for image in image_files]
    return images_pil
```

```
7
8  class GenerateDataset(torch.utils.data.Dataset):
9      def __init__(self, root, class_list, transform=None):
10         super().__init__()
11         self.root = root
12         self.class_list = class_list
13         self.transform = transform
14         self.data = []
15
16         for idx, category in enumerate(self.class_list):
17             images = get_images(self.root, category)
18             for image in images:
19                 self.data.append([image, idx])
20
21     def __len__(self):
22         return len(self.data)
23
24     def __getitem__(self, idx):
25         image = self.transform(self.data[idx][0]) if self.transform else
       self.data[idx][0]
26         label = torch.tensor(self.data[idx][1])
27
28         return image, label
29
30 if __name__ == "__main__":
31     train_root = r"/Users/nikitaravi/Documents/Academics/ECE 60146/HW4/
       Train"
32     val_root = r"/Users/nikitaravi/Documents/Academics/ECE 60146/HW4/Val"
33     class_list = ["airplane", "bus", "cat", "dog", "pizza"]
34     transform = tvt.Compose([tvt.ToTensor(), tvt.Normalize((0.5, 0.5, 0.5)
       , (0.5, 0.5, 0.5))])
35     epochs = 7
36
37     train_dataset = GenerateDataset(train_root, class_list, transform)
38     train_dataloader = torch.utils.data.DataLoader(train_dataset,
       batch_size=2, num_workers=2, shuffle=True)
39
40     test_dataset = GenerateDataset(val_root, class_list, transform)
41     test_dataloader = torch.utils.data.DataLoader(test_dataset, batch_size
       =2, num_workers=2, shuffle=True)
```

Listing 2: Dataset Class and Dataloader

2.3 Shallow Network Architecture

The objective of this task was to create and run a shallow convolutional network architecture and evaluate how well it does in classifying the COCO dataset. The network use instances of ***torch.nn.Conv2d*** in the mode in which it only uses valid pixels which forces the image to shrink as it goes down the architecture. Max-pooling was also implemented in the architecture by executing the command ***MaxPool2d*** to obtain the maximum pixel value in the kernel window over the image. The final layer is a linear fully connected network to highlight which class of the five categories the image belongs to. Which is why the out-features of the

last layer is five. The in-features of the previous fully connected network is the product of the new channel and dimensions of the image which is $32 \times 14 \times 14 = 6272$.

2.3.1 Source Code

```
class HW4Net(nn.Module):
    def __init__(self, task):
        super(HW4Net, self).__init__()
        self.task = task

        if(self.task == "task1"):
            self.conv1 = nn.Conv2d(3, 16, 3)
            self.pool = nn.MaxPool2d(2, 2)
            self.conv2 = nn.Conv2d(16, 32, 3)
            self.fc1 = nn.Linear(32*14*14, 64)
            self.fc2 = nn.Linear(64, 5) # x = 5 because there are 5
    classes

    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
```

Listing 3: Shallow Network

2.4 Shallow Network Architecture with Padding

The only difference between this network and the previous shallow network is that this network has a **padding** of one for all its convolutional layers. The padding changes the amount by which the kernel window moves across the window.

2.4.1 Source Code

```
class HW4Net(nn.Module):
    def __init__(self, task):
        super(HW4Net, self).__init__()
        self.task = task

        if(self.task == "task2"):
            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.fc1 = nn.Linear(32*16*16, 64)
            self.fc2 = nn.Linear(64, 5) # x = 5 because there are 5
    classes

    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
```

Listing 4: Shallow Network with Padding

2.5 Deep Network Architecture

The deep network architecture has the same initial and final layers as the previous two architectures. The middle component of the architecture comprises of 10 convolutional layers each with an in-channel of 32, out-channel of 32, kernel size of 3 and padding of 1. These ten additions make the architecture deep.

2.5.1 Source Code

```
class HW4Net(nn.Module):
    def __init__(self, task):
        super(HW4Net, self).__init__()
        self.task = task

        if(self.task == "task3"):
            self.conv1 = nn.Conv2d(3, 16, 3)
            self.pool = nn.MaxPool2d(2, 2)
            self.conv2 = nn.Conv2d(16, 32, 3)
            self.conv3 = nn.Conv2d(32, 32, 3, padding=1)
            self.fc1 = nn.Linear(32*14*14, 64)
            self.fc2 = nn.Linear(64, 5) # x = 5 because there are 5
   classes

    def forward(self, x):
        if(self.task == "task3"):
            x = self.pool(F.relu(self.conv1(x)))
            x = self.pool(F.relu(self.conv2(x)))
            x = F.relu(self.conv3(x))
            x = F.relu(self.conv3(x))
            x = F.relu(self.conv3(x))
            x = F.relu(self.conv3(x))
            x = F.relu(self.conv3(x))
            x = F.relu(self.conv3(x))
            x = F.relu(self.conv3(x))
            x = F.relu(self.conv3(x))
            x = F.relu(self.conv3(x))
            x = F.relu(self.conv3(x))
            x = x.view(x.shape[0], -1)
            x = F.relu(self.fc1(x))
            x = self.fc2(x)
            return x
```

Listing 5: Deep Network

2.6 Training

The optimizer used for this assignment was the ***Adam optimizer*** for its ability to keep a running average of both the first and second moment of gradients, and take both these moments into consideration for calculating the step size, thus adapting the learning rate and converging at the minimum quicker.

The criterion used for this assignment was the ***Cross-Entropy Loss*** because it measures the difference between the actual probability distribution of a classification model and the predicted distribution.

The training was conducted over seven epochs and the loss at each iteration was recorded. The final trained model was then used to test on the validation dataset and obtain a confusion matrix out of it. A snippet of the training and testing functions are shown below:

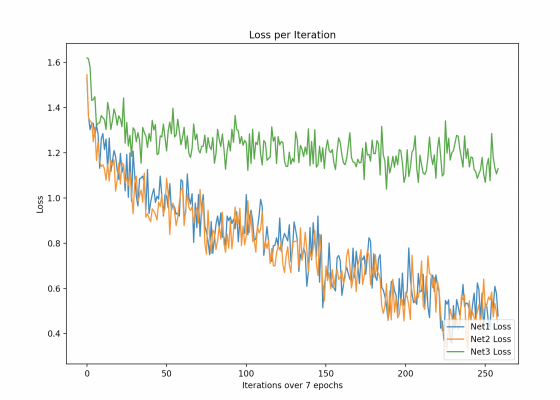
2.6.1 Source Code

```
# GLOBAL VARIABLES
device = 'cuda' if torch.cuda.is_available() else 'cpu'
device = torch.device(device)

def train(net, epochs, lr, betas, dataloader, save_to_path):
    # summary(net, input_size=(10, 3, 64, 64)); quit()
    net = net.to(device)
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(net.parameters(), lr=lr, betas=betas)
    loss_per_iteration = []

    for epoch in range(1, epochs + 1):
        running_loss = 0.0
        for batch_idx, (inputs, labels) in enumerate(dataloader):
            inputs = inputs.to(device)
            labels = labels.to(device)
            optimizer.zero_grad()
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
            if((batch_idx + 1) % 100 == 0):
                print("[epoch: %d, batch: %5d] loss: %.3f" % (epoch,
   batch_idx + 1, running_loss / 100))
                loss_per_iteration.append(running_loss / 100)
                running_loss = 0.0

    if(save_to_path):
        torch.save(net.state_dict(), save_to_path)
    return loss_per_iteration

def test(net, path_to_network, dataloader, num_classes):
    net.load_state_dict(torch.load(path_to_network))
    net = net.to(device)
    confusion_matrix = np.zeros((num_classes, num_classes))
```

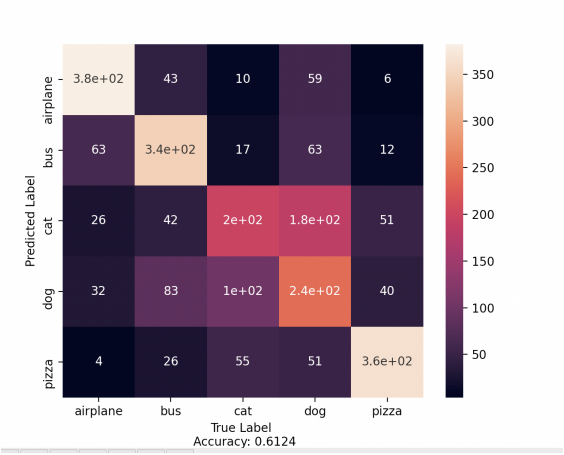
```
36
37      with torch.no_grad():
38          for inputs, labels in dataloader:
39              inputs = inputs.to(device)
40              labels = labels.to(device)
41              outputs = net(inputs)
42              _, predicted = torch.max(outputs, dim=1)
43              for label, prediction in zip(labels, predicted):
44                  confusion_matrix[label][prediction] += 1
45
46      accuracy = np.trace(confusion_matrix) / np.sum(confusion_matrix)
47      return confusion_matrix, accuracy
48
49  def plot_losses(loss1, loss2, loss3, epochs):
50      plt.plot(range(len(loss1)), loss1, label="Net1 Loss")
51      plt.plot(range(len(loss2)), loss2, label="Net2 Loss")
52      plt.plot(range(len(loss3)), loss3, label="Net3 Loss")
53
54      plt.title(f"Loss per Iteration")
55      plt.xlabel(f"Iterations over {epochs} epochs")
56      plt.ylabel("Loss")
57      plt.legend(loc="lower right")
58      plt.show()
59
60  def display_confusion_matrix(conf, class_list, accuracy):
61      sns.heatmap(conf, xticklabels=class_list, yticklabels=class_list,
        annot=True)
62      plt.xlabel(f"True Label \n Accuracy: {accuracy}")
63      plt.ylabel("Predicted Label")
64      plt.show()
```

Listing 6: Training

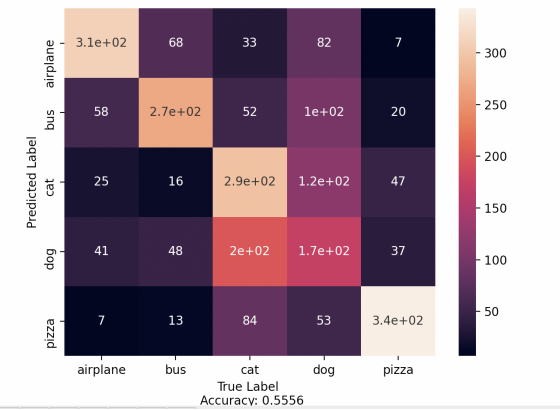
3 Results



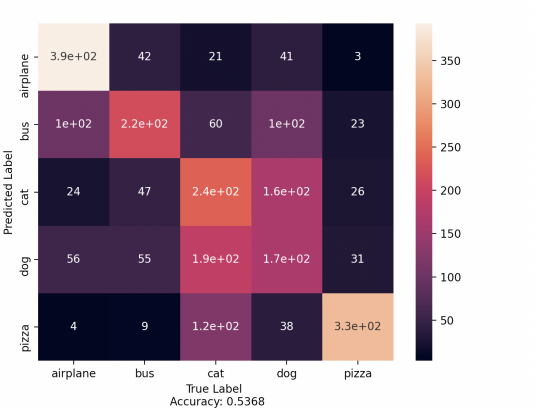
(a) Training Loss for all three networks



(b) Confusion Matrix for First Network



(c) Confusion Matrix for Second Network



(d) Confusion Matrix for Third Network

Figure 2: Loss Graph and Confusion Matrix for all Three Networks

4 Evaluation

1. Does adding padding to the convolutional layers make a difference in classification performance?
   Answer: With respect to the training phase, the training loss for network 1 and two were extremely close, as illustrated in figure 2(a) with network 2 (the one with the padding) performing slightly better than network 1. However, with respect to the testing phase, network 1 achieved a higher accuracy (61.24%) than network 2 (55.56%). For this particular assignment, adding padding made a slight difference.

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?
Answer: The vanishing gradient is slightly noticeable for network three because the loss curve isn't declining as much as the other two networks. This is happening because with the multiple ReLU activation functions used in the architecture, the input is compressed to a smaller input space thus causing a small change in the output which further decreases the derivative of the loss function.

3. Compare the classification results by all three networks, which CNN do you think is the best performer?
Answer: By examining just the validation accuracy of all three networks, network 1 had the highest accuracy in comparison to the others. So the best CNN is network 1.

4. By observing your confusion matrices, which class or classes do you think are more difficult to correctly differentiate and why?
Answer: The category that was most difficult to classify using this network was the cat class. This is probably because the cat was either in the background of the image or the network mistook it for a different pet: dog.

5. What is one thing that you propose to make the classification performance better?
Answer: Possibly changing the activation function to a sigmoid function or using different chained convolutional layers with different parameters would make the classification accuracy better.