HOMEWORK 4

ALEXANDRE OLIVE PELLICER

3.1. Using COCO to Create Your Own Image Classification Dataset

To create the dataset, we use the COCO API and the instances_train2017.json file to obtain the labels and ids of the images. The basic idea behind the code implemented to create both the Training and Validation dataset by no repeating any image has been keeping the ids of the images that have been already added to both the Training and Validation dataset in a list called "used_ids" so that before adding a new image, we will first check if its id has already been stored in "used_ids". If and only if it hasn't been stored in "used_ids" previously, then we will add it in the dataset.

This is the code used:

```
dataDir='.'

dataTypes=['train2017', 'train2017']
num_samples = [1600, 400]
saving_paths = ['../COCOTraining', '../COCOValidation']

classes = ['boat','couch','dog', 'cake', 'motorcycle']
used_ids = []

k=0
added = 0

# This for iterates 2 times. First to create the training dataset and
second to create the training dataset
for i, dataType in enumerate(dataTypes):
    annFile='{}/annotations/instances_{}.json'.format(dataDir,dataType)
    coco=COCO(annFile)

    # For each class, we first get all the ids of the images that have
been labeled with the specific class
    for cls in classes:
        catIds = coco.getCatIds(catNms=[cls])
        imgIds = coco.getImgIds(catIds=catIds )
```

```
        # We go across all the ids stored until reaching the desired
number of samples for the corresponding class
        while added < num_samples[i]:
            img_name = str(imgIds[k]).zfill(12)
            file_path =
f"../../../../Downloads/train2017/train2017/{img_name}.jpg"
            # If the image with specific id hasn't been previously added
to either the training dataset or the validation dataset, then we load
it, resize it and add it in the corresponding dataset
            if imgIds[k] not in used_ids and os.path.exists(file_path):
                original_image = cv2.imread(file_path)

                # Convert the image from BGR to RGB (OpenCV uses BGR by
default)
                original_image = cv2.cvtColor(original_image,
cv2.COLOR_BGR2RGB)

                # Read the image using skimage
                resized_image_opencv = cv2.resize(original_image, (64,
64))

                # Save the resized image using PIL
                resized_image_pil = Image.fromarray(resized_image_opencv)

resized_image_pil.save(f'{saving_paths[i]}/{cls}_{added}_{imgIds[k]}.jpg'
)

                # We add the id of the image we have just added to our
dataset to not using it again and increase the counter for added images
                used_ids.append(imgIds[k])
                added +=1

            # We move to the next id
            k += 1

        # For each class we reset the index for the array of ids "k" and
the counter of added images "added"
        k=0
        added = 0

# If the code has been correctly implemented, the length of used ids
should be 1600*5 + 400*5 = 10000
print(len(used_ids))
```

This is a selection of images from my dataset:

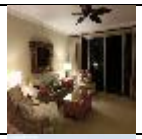
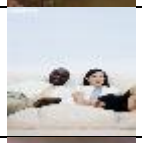
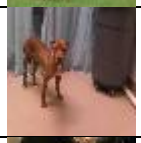
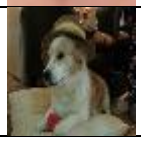
boat	cake	couch	dog	motorcycle
				
				
				

Fig 1: Selection of images from my dataset

3.2 Image Classification using CNNs– Training and Validation

In order the load the images for training or testing, we will use a dataloader taking the data from a dataset class that we have created following the structure of the class created in homework 2. It is this one:

```
class MyDataset ( torch.utils.data.Dataset ):

    def __init__ ( self , root ):
        super().__init__()
        # Obtain meta information (e.g. list of file names )
        self.root = root
        self.filenames = os.listdir(self.root)

        # Initialize data transforms , etc.
        self.transform = tvt.Compose([
            tvt.ToTensor(),
            tvt.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
        ])

    def __len__ ( self ):
        # Return the total number of images
        # the number is a place holder only
        return len(self.filenames)

    def __getitem__ ( self , index ):
        # Read an image at index and perform processing
        path = os.path.join(self.root, self.filenames[index])
        cls = self.filenames[index].split('_')[0]
        img = Image.open(path)

        # Get the normalized tensor
        transformed_img = self.transform(img)
```

```
        if cls == "boat":
            label = torch.tensor([1.0, 0.0, 0.0, 0.0, 0.0])
        elif cls == "couch":
            label = torch.tensor([0.0, 1.0, 0.0, 0.0, 0.0])
        elif cls == "dog":
            label = torch.tensor([0.0, 0.0, 1.0, 0.0, 0.0])
        elif cls == "cake":
            label = torch.tensor([0.0, 0.0, 0.0, 1.0, 0.0])
        elif cls == "motorcycle":
            label = torch.tensor([0.0, 0.0, 0.0, 0.0, 1.0])

        # Return the tuple : ( normalized tensor, label )
        return transformed_img, label
```

For all the CNN that we build in this assignment, we want them to classify images in 5 different classes. For this reason, the final linear layer needs to have 5 output features. Thus, “XX” = 5. The value of “XXXX” will depend on the size of the input tensor and the architecture of each CNN.

For training all our CNN we used the following:

- Cross Entropy Loss
- Adam optimizer with lr = 1e-3 and beta1= 0.9 and beta2 = 0.99
- 15 epochs
- Batch size 4

CNN Task 1:

Following the code structure provided in the assignment, we can define the HW4Net1 class and train it as follows:

```
class HW4Net1(nn.Module):
    def __init__(self):
        super(HW4Net1, self).__init__()
        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)

    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
```

```
device = "cuda:0"

# Dataloader
my_dataset = MyDataset ("../COCOTraining")
batch_size = 4
train_data_loader = DataLoader(my_dataset, batch_size = batch_size,
shuffle = True )

net1 = HW4Net1()

# Print num of learnable parameters
total_params = sum(p.numel() for p in net1.parameters())
print(f"Number of learnable parameters: {total_params}")

# List where the loss values will be stored
loss_net1 = []

# Training routine provided by the assignment
net1 = net1.to(device)
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(
net1.parameters(), lr=1e-3, betas=(0.9, 0.99))
epochs = 15
for epoch in tqdm(range(epochs)):
    running_loss = 0.0
    for i, data in enumerate(train_data_loader):
        inputs, labels = data
        inputs = inputs.to(device)
        labels = labels.to(device)
        optimizer.zero_grad()
        outputs = net1(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        if (i+1) % 100 == 0:
            # print("[epoch: %d, batch: %5d] loss: %.3f" \
            # % (epoch + 1, i + 1, running_loss / 100))
            loss_net1.append(running_loss/100)

            running_loss = 0.0

# Save the learned parameters of the model to do inference afterwards

torch.save(net1.state_dict(), './net1_normalization.pth')
```

In order to define the value of "XXXX", we follow these formulas which apply for the nn.MaxPool2d and nn.Conv2d layers:

- Input: $(N, C_{in}, H_{in}, W_{in})$ or (C_{in}, H_{in}, W_{in})
- Output: $(N, C_{out}, H_{out}, W_{out})$ or $(C_{out}, H_{out}, W_{out})$, where

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

Fig 2: Formulas to specify the output size of the nn.MaxPool2d and nn.Conv2d layers

In this case, as the input size of images is 64 by 64 with 3 channels, following the **black** line from the image below, we conclude that the value of "XXXX" for the first CNN will be 14x14x32.

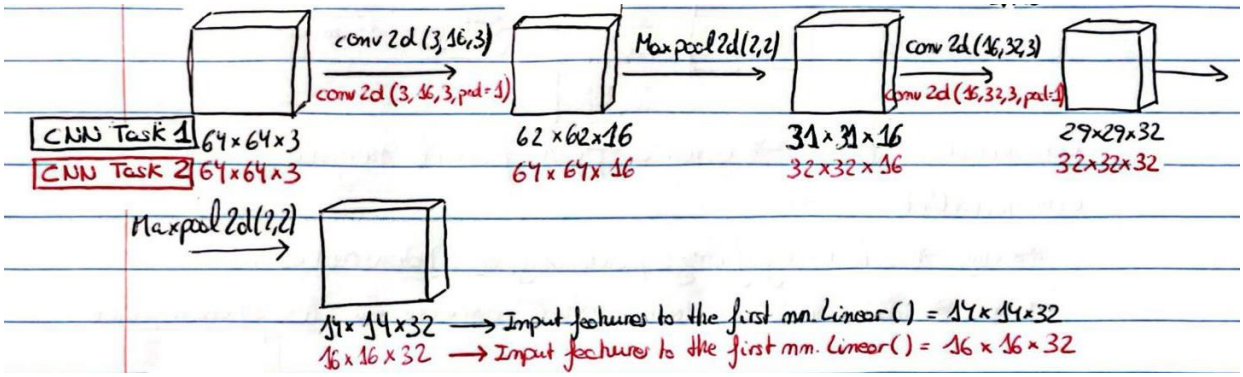


Fig 3: Step by step about how the size of the tensor decreases in each layer

CNN Task 2:

Now we added a padding of 1 to all the convolution layers from Net1.

Following the code structure provided in the assignment, we can define the HW4Net2 class and train it as follows:

```
class HW4Net2(nn.Module):
    def __init__(self):
        super(HW4Net2, self).__init__()
        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) #Now here there will be an
input of 16x16x32 features
        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

device = "cuda:0"

# Dataloader
my_dataset = MyDataset ("../COCOTraining")
batch_size = 4
train_data_loader = DataLoader(my_dataset, batch_size = batch_size,
shuffle = True )

net2 = HW4Net2()

# Print num of learnable parameters
total_params = sum(p.numel() for p in net2.parameters())
print(f"Number of learnable parameters: {total_params}")

# List where the loss vañues will be stored
loss_net2 = []

# Training routine provided by the assignment
net2 = net2.to(device)
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(
net2.parameters(), lr=1e-3, betas=(0.9, 0.99))
epochs = 15
for epoch in tqdm(range(epochs)):
    running_loss = 0.0
    for i, data in enumerate(train_data_loader):
        inputs, labels = data
        inputs = inputs.to(device)
        labels = labels.to(device)
        optimizer.zero_grad()
        outputs = net2(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        if (i+1) % 100 == 0:
            # print("[epoch: %d, batch: %5d] loss: %.3f" \
            # % (epoch + 1, i + 1, running_loss / 100))
            loss_net2.append(running_loss/100)
```

```
            running_loss = 0.0

# Save the learned parameters of the model to do inference afterwards
torch.save(net2.state_dict(), './net2_normalization.pth')
```

In this case, as the input size of images is 64 by 64 with 3 channels, following the **red** line from the image below, we conclude that the value of "XXXX" for the first CNN will be 16x16x32.

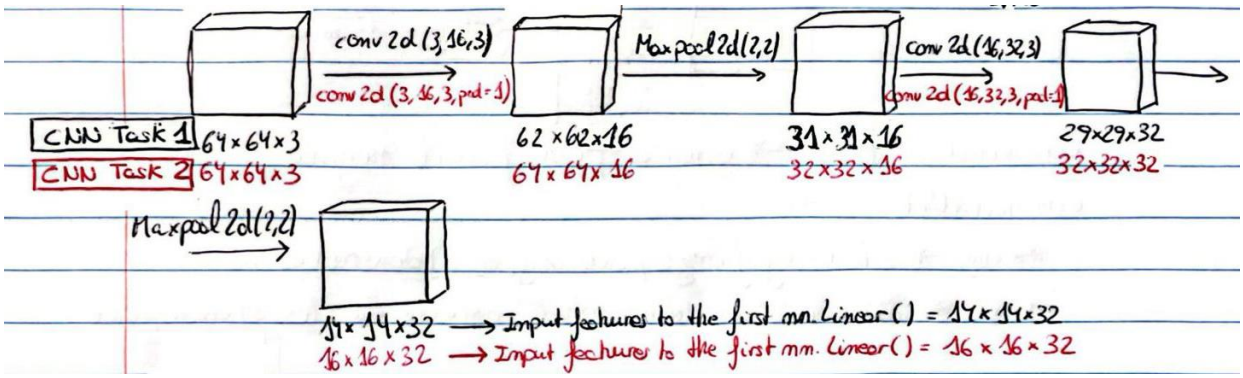


Fig 3: Step by step about how the size of the tensor decreases in each layer

CNN Task 3:

Now we added 10 extra convolutional layers between the second conv layer and the first linear layer following the instructions from the assignment.

In this case, the value of "XX" will be the same as the one used in Net2 since the convolutional layers added have the same number of input features as output features.

Following the code structure provided in the assignment, we can define the HW4Net3 class and train it as follows:

```
class HW4Net3(nn.Module):
    def __init__(self):
        super(HW4Net3, self).__init__()
        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.conv_repeat = nn.Conv2d(32, 32, 3, padding=1)
        self.fc1 = nn.Linear(32*16*16, 64) #Now here there will be an
input of 16x16x32 features
        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 = F.relu(self.conv_repeat(x))
        x = F.relu(self.conv_repeat(x))
```

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

device = "cuda:0"

# Dataloader
my_dataset = MyDataset ("../COCOTraining")
batch_size = 4
train_data_loader = DataLoader(my_dataset, batch_size = batch_size,
shuffle = True )

net3 = HW4Net3()

# Print num of learnable parameters
total_params = sum(p.numel() for p in net3.parameters())
print(f"Number of learnable parameters: {total_params}")

# List where the loss values will be stored
loss_net3 = []

# Training routine provided by the assignment
net3 = net3.to(device)
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(
net3.parameters(), lr=1e-3, betas=(0.9, 0.99))
epochs = 15
for epoch in tqdm(range(epochs)):
    running_loss = 0.0
    for i, data in enumerate(train_data_loader):
        inputs, labels = data
        inputs = inputs.to(device)
        labels = labels.to(device)
        optimizer.zero_grad()
        outputs = net3(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
```

```
        if (i+1) % 100 == 0:
            # print("[epoch: %d, batch: %5d] loss: %.3f" \
            # % (epoch + 1, i + 1, running_loss / 100))
            loss_net3.append(running_loss/100)

            running_loss = 0.0

# Save the learned parameters of the model to do inference afterwards
torch.save(net3.state_dict(), './net3_normalization.pth')
```

Plot of the 3 lost functions:

We use the following code to plot the 3 loss functions:

```
plt.plot(loss_net1)
plt.plot(loss_net2)
plt.plot(loss_net3)

plt.legend(["loss_net1", "loss_net2", "loss_net3"])

# Adding labels and title
plt.xlabel('PIterations')
plt.ylabel('Loss')
plt.title('Loss comparisson for the 3 nets')

# Display the plot
plt.show()
```

This is the obtained result:

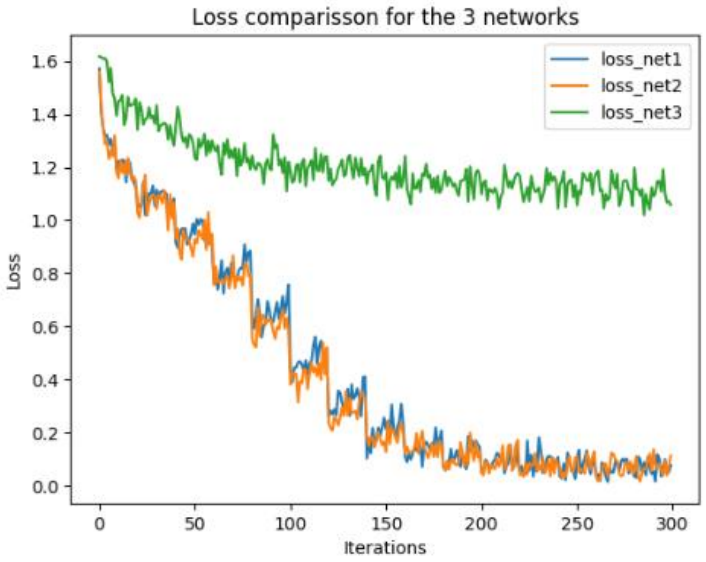


Fig 4: Loss comparison for the 3 networks

Mention that we are plotting the “running_loss” which is stored and displayed every 100 batches.

Confusion matrix for each CNN:

To compute the confusion matrixes, we use the Validation dataset and do inference over the model with the trained parameters loaded. This is the code we have used:

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report

device = "cuda:0"

# Dataloader loading the Validation dataset
my_dataset = MyDataset("../COCOValidation")
batch_size = 4
train_data_loader = DataLoader(my_dataset, batch_size = batch_size,
shuffle = True )

# Lists where the labels will be stored for each of the images from the
Validation dataset
predictions = []
real_labels = []

# Load the trained weights
net3 = HW4Net3()
net3.load_state_dict(torch.load('net3_normalization.pth'))
net3 = net3.to(device)

# Set the model to evaluation mode
net3.eval()

# Get the predicted label and the real label from each image and store
them to the lists mentioned before
for i, data in enumerate(train_data_loader):
    inputs, labels = data
    inputs = inputs.to(device)
    labels = labels.to(device)
    with torch.no_grad():
        outputs = net3(inputs)
    outputs = outputs.split(1)
    labels = labels.split(1)
    for lbl in labels:
        real_labels.append(torch.argmax(lbl.squeeze()).item())
    for pred in outputs:
```

```
        predictions.append(torch.argmax(pred.squeeze()).item())

# Compute the confusion matrix
cm = confusion_matrix(real_labels, predictions)

# Create a heatmap for visualization
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='g', cmap='Blues', cbar=False,
xticklabels=['boat','couch','dog', 'cake', 'motorcycle'],
yticklabels=['boat','couch','dog', 'cake', 'motorcycle'])
plt.xlabel('Predicted labels')
plt.ylabel('True labels')
plt.title('Confusion Matrix')
plt.show()

# Print classification report for additional metrics like accuracy
print("Classification Report:\n", classification_report(real_labels,
predictions))
```

These are the obtained results:

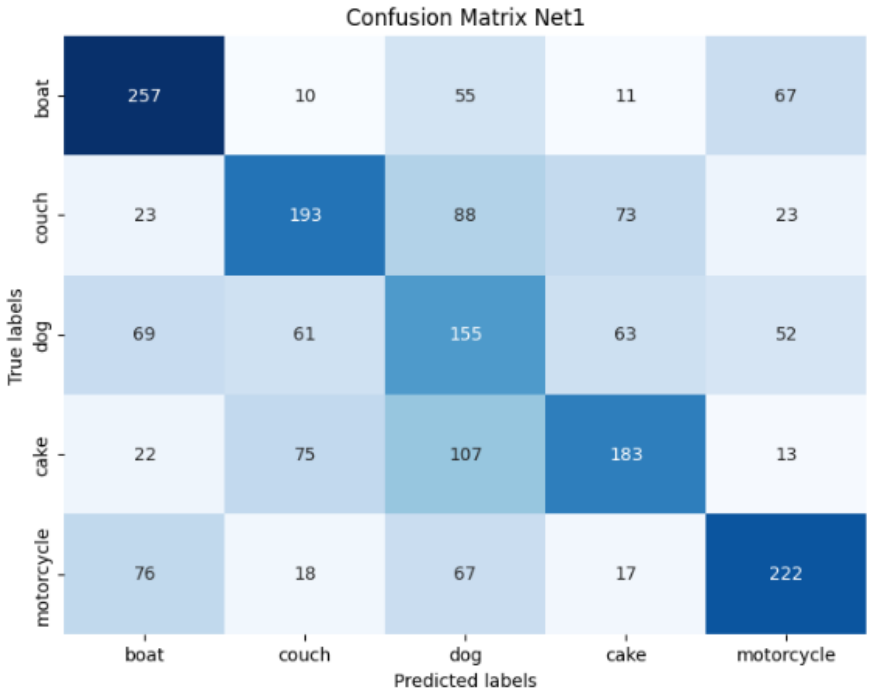


Fig 5: Confusion matrix for Net1

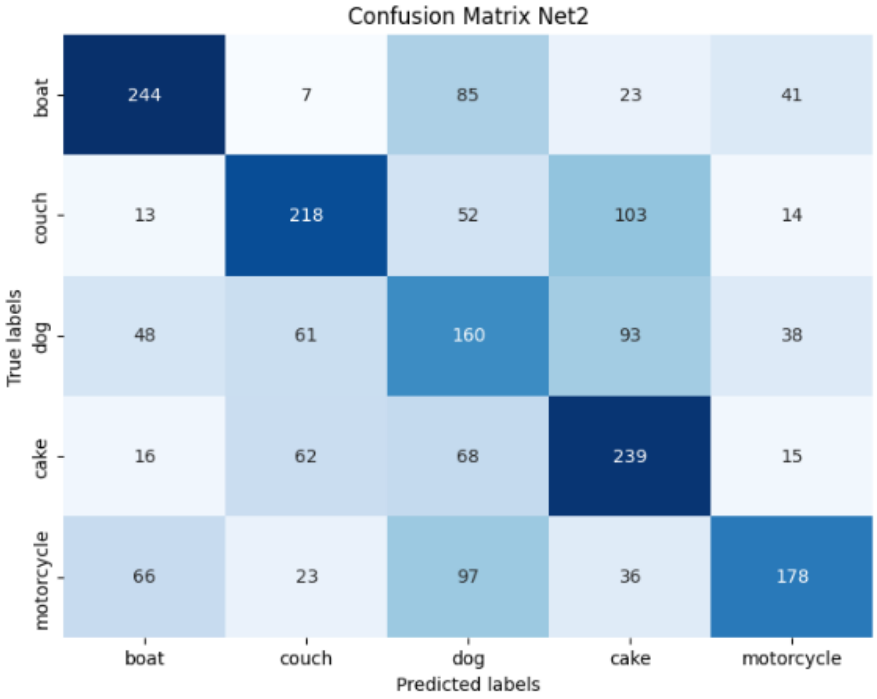


Fig 6: Confusion matrix for Net2

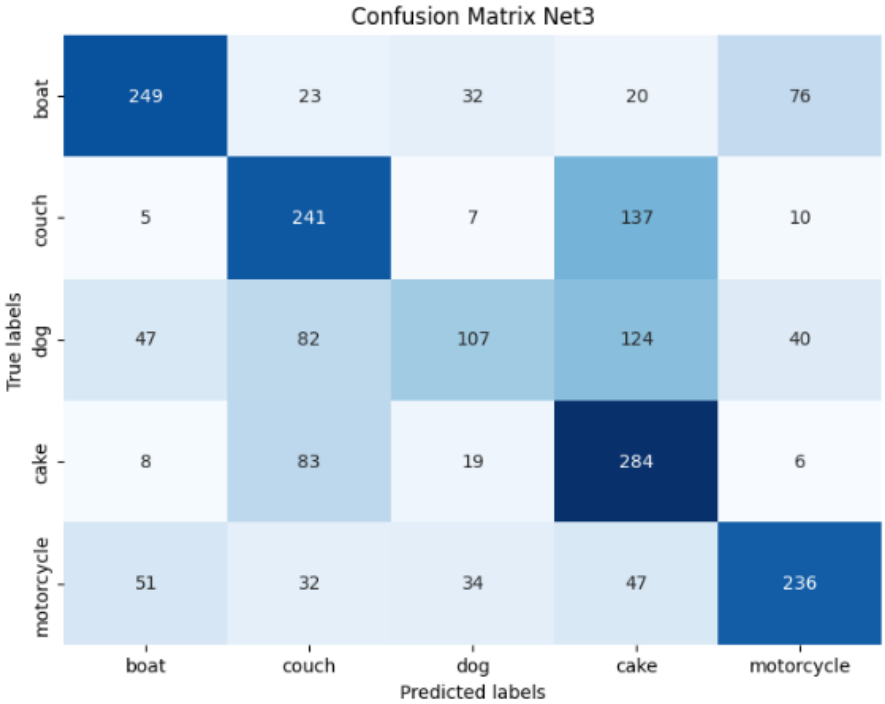


Fig 7: Confusion matrix for Net3

Number of learnable parameters and accuracy for each CNN:

To print the number of learnable parameters in each CNN we have used the following code snippet:

```
total_params = sum(p.numel() for p in net1.parameters())
print(f"Number of learnable parameters: {total_params}")
```

To print the accuracy, we have used the classification_report method from sklearn.metrics:

```
print("Classification Report:\n", classification_report(real_labels,
predictions))
```

These are the obtained results:

Net #	Number of learnable parameters	Classification accuracy with Validation dataset	Classification accuracy with Training dataset
Net1	406885	51%	98%
Net2	529765	54%	99%
Net3	539013	56%	58%

Fig 8: Table of number of parameters and accuracy for each network

Answering the questions:

1. **Does adding padding to the convolutional layers make a difference in classification performance?**
   Yes, it does it. We can see that the classification accuracy is higher in Net2 than in Net1 for both the Training and Validation dataset. Using padding avoids reducing the size of the tensors at the output of the convolutional layers and we can conclude that it has a positive effect in the classification performance.

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?**
   Looking at the plot of the 3 losses for each net against the number of iterations, we can see that the loss from Net3 doesn't converge to a minimum value between 0 and 0.2 as it is done in Net1 and Net2. We can see that the accuracy for Net3 (58%) for the training dataset is very far from the accuracies obtained for Net1 and Net2 for the training dataset too (98% and 99% respectively). It seems like adding extra convolutional layers to the network makes it difficult to correctly learn the parameters. Surprisingly, we don't see this behavior for the validation dataset (the best accuracy is achieved by Net3 although is close to the accuracies from Net1 and Net2). This probably happens because our Net1 and Net2 are overfitting. This implies that the network overly focuses on learning the intricacies of the training data, resulting in excessively complex decision boundaries. Therefore, the model struggles to generalize and tends to make errors when classifying unseen data not encountered during training.

3. **Compare the classification results by all three networks, which CNN do you think is the best performer?**

   I think that the network that offers a better performance is Net2. As we have mentioned in question 2 it is difficult to have a clear opinion by looking at the accuracies obtained in the validation dataset since they are all close and probably the number of samples used for testing are not representative at all. Considering that Net1 and Net2 are the ones that converge to a minimum loss value and the ones that offer a better accuracy for the training dataset, I would discard Net3 as a network with good performance. Finally, looking at the accuracies from Net1 and Net2 and their confusion matrixes it looks like Net2 slightly outperforms Net1. It is good the mention that Net1 and Net2 seem to be overfitting because of the high accuracies obtained with the training dataset and lower accuracy with validation dataset. This means that the network learns too much the features of the training data and the decision boundaries turn to be so complex. Thus, it doesn't generalize and then it makes mistakes when classifying data that it hasn't been seen during training. That's why we see a better accuracy, better performance with Net3.

4. **By observing your confusion matrices, which class or classes do you think are more difficult to correctly differentiate and why?**

   In the 3 confusion matrixes the class that has been labeled incorrectly more times is "dog". The COCO dataset contains different class labels for 1 image. For example, if an image has a dog and a couch, it will appear in the COCO dataset with both labels but, by the way we created our dataset, we will only have one label for it. Thus, in the case of having an image with a couch and a dog, we have it in our training dataset labeled with only one class, for example as a dog. Nevertheless, when doing inference our model may predict that it is a couch. In theory the model could be considered to work correctly, but in practice we would compute it as incorrect since we had that image labeled as "dog". Therefore, as an extension of this example, I think that "dog" is the class that is more difficult to differentiate because probably it is more prompt to appear to images where there is also a "boat" or a "couch" or a "cake" or a "motorcycle" and we are considering as a bad classification although it is correctly done.

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

   Because of the reasoning done in question 4, a good way of improving the performance of our network would be improving the training dataset and the labeling. For example, avoiding images where there appear one of the classes that we are targeting but the image is not labeled with it. Thus, avoiding the COCO images which are labeled with 2 or more of the classes [boat, couch, dog, cake, motorcycle]. Furthermore, increasing the number of training samples would also help to improve the performance since the network could learn more varied features for each class.