ECE 60146 - Homework 4

Ali Almuallem, aalmuall@purdue.edu

February 2025

1 DL Studio

1.1 Install DLStudio

I installed the DLStudio [2] by following the instructions on the website by running the following command

```
python setup.py install
```

but this has resuledt in issues in my machine so I used the build commans and then imported the library like below. The system path should include the diurectory where DLStudio is located.

```
import sys
sys.path.append( "../" )
from DLStudio import *
```

1.2 Explore Example Networks

I ran both networks with the default parameters as noted bellow. Notice how the second network is much more complex which we anticipate will perform better in image classification tasks. **Net()** has only 6 layers of filters, while **Net2()** has 128 of those which would translate to more capacity to learn complex features.

1.2.1 Net() default model architecture

```
Net(
  (conv1): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=400, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

1.2.2 Net2() default model architecture

```
Net2(
  (relu): ReLU()
  (conv1): Conv2d(3, 128, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv3): Conv2d(128, 128, kernel_size=(2, 2), stride=(1, 1), padding=(1, 1))
  (pool3): MaxPool2d(kernel_size=2, stride=1, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=8192, out_features=150, bias=True)
  (fc2): Linear(in_features=150, out_features=100, bias=True)
  (fc3): Linear(in_features=100, out_features=10, bias=True)
)
```

1.2.3 Results for different parameters

We run the **Net()** with different parameters and we report the results below.

Original Architecture

```
Net(
  (conv1): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=400, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

Modified Architecture We changed the filters to 3 in the first convolution layer instead of 6, and 8 in the second convolutional layer instead of 16.

```
Net(
  (conv1): Conv2d(3, 3, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(3, 8, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=200, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

Class	Unmodified Net()	Modified Net() with less filters
plane	**53%**	51%
car	**72%**	60%
bird	**50%**	38%
cat	33%	**47%**
deer	32%	**39%**
dog	**55%**	20%
frog	**67%**	49%
horse	46%	**48%**
ship	**77%**	65%
truck	33%	**49%**
overall	**52%**	47%

Table 1: The accuracies for the original **Net()** and the modified version with less number of filters. As anticipated, the original one with more filters achieved higher overall accuracy as those filters enabled more features discovery. The **boldface** values are higher for each category.

1.3 Run CIFAR-10 Example

We run the following to run the CIFAR-10 [3] example.

Note: I had to modify the provided code because it caused issues with the multi-process on my MAC computer with Apple Silicon.

The edit was basically wrapping the functions inside the *playing_with_cifar10.py* into a main function

```
if __name__ == '__main__':
    dls = DLStudio(
    #                   dataroot = "/home/kak/ImageDatasets/CIFAR-10/",
                        dataroot = "./data/CIFAR-10/",
                        image_size = [32,32],
                        path_saved_model = "./saved_model",
                        momentum = 0.9,
                        learning_rate = 1e-3,
                        epochs = 2,
                        batch_size = 4,
                        classes = ('plane','car','bird','cat','deer','dog','frog','horse
                            ','ship','truck'),
                        use_gpu = True,
                    )
    ...
    ...
```

Input

```
python playing_with_cifar10.py
```

Output

```
...
...
[epoch:2/2  iter=11000  elapsed_time=  164 secs]   Predicted Labels:        cat
    ship        dog        dog
[epoch:2/2  iter=11000  elapsed_time=  164 secs]   Loss: 1.286


[epoch:2/2  iter=12000  elapsed_time=  169 secs]   Ground Truth:            car
    car       bird        cat
[epoch:2/2  iter=12000  elapsed_time=  169 secs]   Predicted Labels:        car
    frog       deer        cat
[epoch:2/2  iter=12000  elapsed_time=  169 secs]   Loss: 1.276

Finished Training
```

CIFAR10 Example - Loss with Net()

The loss as a function of iterations for the example code for training CIFAR10. The loss has monitonically decreased which suggests the network is learning successfully.

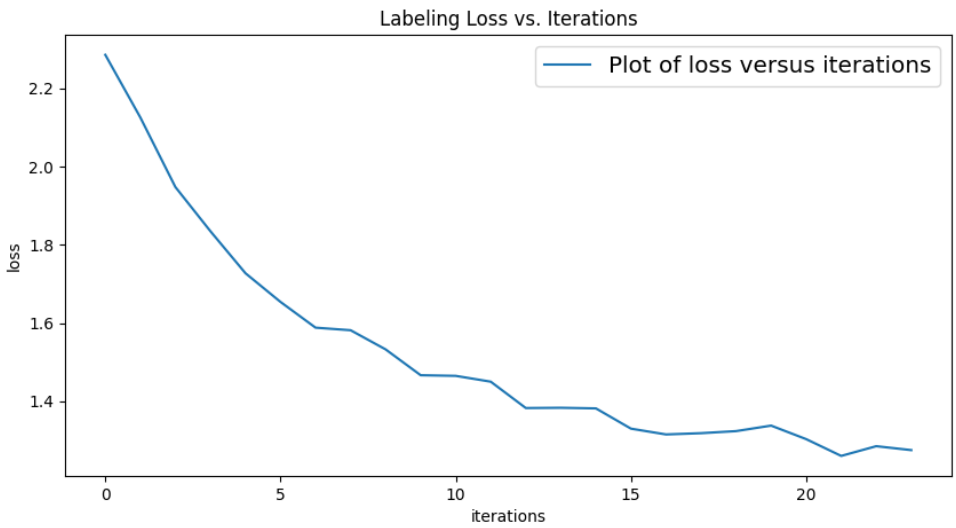

Figure 1: CIFAR10 loss graph at the end of the training.

CIFAR10 Example - Accuracy with Net()

```
Prediction accuracy for plane : 53 %
Prediction accuracy for   car : 72 %
Prediction accuracy for  bird : 50 %
Prediction accuracy for   cat : 33 %
Prediction accuracy for  deer : 32 %
Prediction accuracy for   dog : 55 %
Prediction accuracy for  frog : 67 %
Prediction accuracy for horse : 46 %
Prediction accuracy for  ship : 77 %
Prediction accuracy for truck : 33 %

Overall accuracy of the network on the 10000 test images: 52 %
```

CIFAR10 Example - Confusion Matrix with Net()

Notice how the diagonal entries are higher which suggests that the network predicts the correct label with higher accuracy.

```
Displaying the confusion matrix:

            plane     car    bird     cat    deer     dog    frog   horse    ship   truck
  plane:    53.90    3.30    7.70    2.00    1.80    0.70    2.80    0.30   27.10    0.40
    car:     3.00   72.90    1.10    1.30    0.50    0.40    1.40    0.00   14.70    4.70
   bird:     6.80    1.40   50.70    7.90    7.80    9.20    8.80    1.00    5.60    0.80
    cat:     2.70    2.50   12.30   33.40    4.70   25.90    8.90    2.00    5.60    2.00
   deer:     3.90    1.80   24.90    7.00   32.70    7.20   11.80    6.60    3.70    0.40
    dog:     1.60    1.50   12.00   14.40    3.80   55.60    4.50    3.50    2.90    0.20
   frog:     0.90    1.60    8.80    7.20    3.60    6.70   67.20    0.30    2.90    0.80
  horse:     3.40    0.80   11.90    8.80    8.00   13.90    2.00   46.10    2.80    2.30
   ship:    11.40    3.20    2.40    1.70    1.20    1.20    0.50    0.20   77.90    0.30
  truck:     5.00   27.70    2.80    4.20    0.40    1.10    2.30    0.70   22.60   33.20
```

CIFAR10 Example - Loss with Net2()

The loss as a function of iterations for the example code for training CIFAR10 and with the network **Net2()**. The loss appears to be lower than the first trial above in Fig. 1 which used a simpler network.

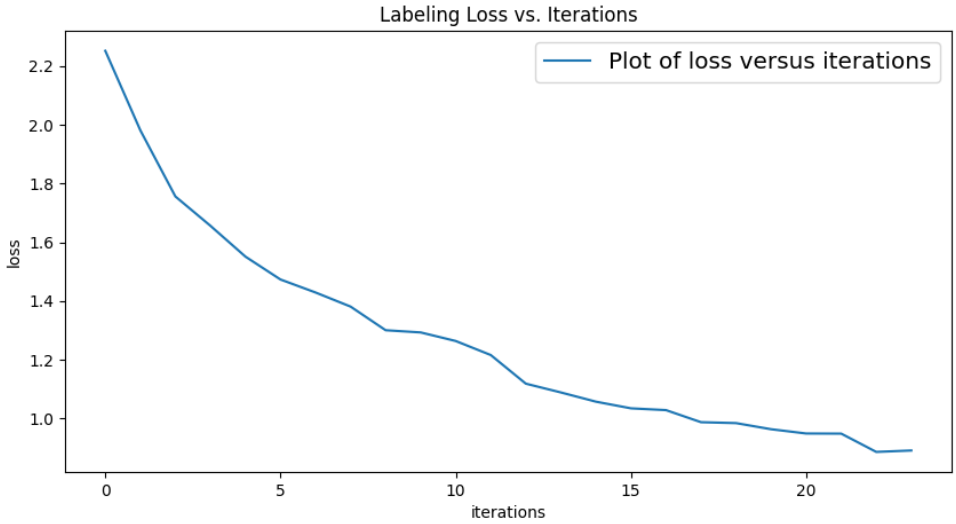

Figure 2: CIFAR10 loss graph at the end of the training with the more complex network Net2()

CIFAR10 Example - Accuracy with Net2()

Notice how the overall accuracy is **69%** while the previous network only achieved **52%**. A substantial improvement.

```
Prediction accuracy for plane : 69 %
Prediction accuracy for   car : 81 %
Prediction accuracy for  bird : 57 %
Prediction accuracy for   cat : 54 %
Prediction accuracy for  deer : 61 %
Prediction accuracy for   dog : 33 %
Prediction accuracy for  frog : 89 %
Prediction accuracy for horse : 68 %
Prediction accuracy for  ship : 75 %
Prediction accuracy for truck : 80 %
```

```
Overall accuracy of the network on the 10000 test images: 67 %
```

CIFAR10 Example - Confusion Matrix with Net2()

Notice how the diagonal entries are higher which suggests that the network predicts the correct label with higher accuracy.

But also compared with the previous experiment, the diagonal entries which correspond to accuracies are much higher.

```
Displaying the confusion matrix:

            plane    car   bird    cat   deer    dog   frog  horse   ship  truck

 plane:     69.40   2.20  12.00   2.50   2.30   0.10   2.10   1.30   4.70   3.40
   car:      1.50  81.10   1.20   0.10   0.40   0.10   2.20   0.10   1.00  12.30
  bird:      4.10   0.50  57.10   7.10  10.00   1.80  14.80   2.40   0.60   1.60
   cat:      0.90   0.90   7.30  54.10   5.90   4.10  20.60   2.90   1.10   2.20
  deer:      1.30   0.20   5.70   7.20  61.80   0.70  13.80   8.00   0.90   0.40
   dog:      0.90   0.70   7.40  33.20   6.10  33.20  13.00   3.30   1.10   1.10
  frog:      0.30   0.40   3.90   2.60   2.70   0.00  89.20   0.30   0.20   0.40
 horse:      0.70   0.60   4.40   7.30  10.90   1.60   3.00  68.80   0.40   2.30
  ship:      7.60   5.10   3.10   1.30   0.90   0.10   1.30   1.00  75.90   3.70
 truck:      3.50   8.30   0.60   1.20   0.70   0.00   1.90   1.10   1.80  80.90
```

1.4 Understanding Network Architecture

As stated in the instructions, we should expect the network architecture to have a final layer with a dimension of 10 to match the 10 classes we are predicting. Upon inspecting the **Net()** and **Net2()** architecture, the final layer indeed outputs 10 as seen below.

Net() final linear layer outputs 10 items.

```
...
...
self.fc3 = nn.Linear(84, 10)
```

Net2() final linear layer outputs 10 items.

```
...
...
self.fc3 = nn.Linear(100, 10)
```

1.5 Analyze Resolution Changes

The input images are of size 32×32.

Net(): Quickly reduces resolution due to lack of padding, which limits its ability to retain spatial detail.

Net2(): Uses padding and pooling effectively to control resolution changes, gradually reducing spatial dimensions while preserving spatial features for higher accuracy on complex tasks.

1.6 Kernel Size and Padding

- **Larger Kernels (5×5):**
  - **Net() architecture**
    - Both convolutional layers use 5×5 kernels, capturing larger spatial patterns but requiring more parameters.
    - Rapid spatial resolution reduction can hinder the preservation of fine-grained spatial information.
  - **Net2() architecture**
    - Conv1 uses a 5×5 kernel with padding to preserve spatial dimensions.
    - Subsequent layers use smaller kernels (3×3, 2×2) to refine features and reduce computational complexity.

- **Smaller Kernels (3×3, 2×2):**
  - They focus on smaller, local patterns, making them computationally efficient.
  - **Net2() architecture** Uses these kernels in deeper layers to progressively refine features that have been extracted in earlier layers, which improves performance.

Padding

- **Without Padding:**
  - **Net() architecture**
    - The lack of padding reduces spatial resolution after each convolution.
    - Example: Input (32×32) $\rightarrow$ After Conv1 (28×28) $\rightarrow$ After Conv2 (24×24).
    - **Consequences?**
      - Loss of valuable edge or boundary information as they are lost when convoloving.
- **With Padding:**
  - **Net2() architecture**
    - Padding make sures the spatial dimensions are preserved after convolutions.
    - Example: Input (32×32) $\rightarrow$ After Conv1 (32×32).
    - **Consequences?**
      - Preserve spatial information for deeper layers.

Performance Implications

- **Feature Extraction:**
  - **Net() architecture** Limited filters and no padding lead to poor feature extraction.
  - **Net2() architecture** Using padding and progressive refinement improve feature extraction.

2 Programming Tasks

2.1 Dataset Creation

NOTE: Some parts of the code below may have been adopted from the code snippet provided in the homework. [1]

The code below specify the paths for the dataset and annotations, specify the augmentation transformation and define a function to create the 3 datasets:

- Single instance
- Multiple instances, same
- Multiple instances, different

If the data were not sufficient to extract the required dataset (500 images in total per category for each dataset, 400 of which are for training, 100 for validation), then the code will use augmentation to transform some images to meet the required threshold.

The categories are as follows:

- airplane
- train
- bird
- giraffe
- clock

```
import os
import shutil
from pycocotools.coco import COCO
from PIL import Image
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
import torchvision.transforms as transforms
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns
import random
from tabulate import tabulate

# Set paths
data_dir = "data"
ann_file = os.path.join(data_dir, "annotations", "instances_train2014.json")
image_dir = os.path.join(data_dir, "train2014")
output_dir = "output_datasets"
os.makedirs(output_dir, exist_ok=True)

# Selected classes (5 from the single-instance categories)
classes = ['airplane', 'train', 'bird', 'giraffe', 'clock']
datasets = ['single_instance', 'multi_instance_same', 'multi_instance_diff']

#Initialize COCO API
coco = COCO(ann_file)

# Augmentation transformation
augmentation_transform = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(degrees=30),
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),
    transforms.RandomResizedCrop((64, 64), scale=(0.8, 1.0)),
    transforms.ToTensor(),
    transforms.ToPILImage(),
])

```

```
41
42
43 #When the images are deficient (did not meet the training and validation threshold of
       500 images, augment some images)
44 def augment_images(input_dir, output_dir, required_count):
45     """
46     Augments images in the input directory until the required count is met.
47     """
48     existing_images = os.listdir(input_dir)
49     augmented_count = len(existing_images)
50
51     while augmented_count < required_count:
52         # randomly select an image for augmentation
53         image_name = random.choice(existing_images)
54         img_path = os.path.join(input_dir, image_name)
55         img = Image.open(img_path)
56
57         #Apply the augmentation transform
58         augmented_img = augmentation_transform(img)
59
60         #save the augmented image
61         augmented_img.save(os.path.join(output_dir, f"aug_{augmented_count}.jpg"))
62         augmented_count += 1
63
64 #A function to extract the images
65 def extract_images_for_class(class_name, dataset_type, min_instances, max_instances,
       multiple_categories=False):
66     cat_ids = coco.getCatIds(catNms=[class_name])
67     img_ids = coco.getImgIds(catIds=cat_ids)
68
69     #To remove the duplicates
70     img_ids = list(set(img_ids))
71
72     #Create the directories for training and validation
73     train_dir = os.path.join(output_dir, dataset_type, 'train', class_name)
74     val_dir = os.path.join(output_dir, dataset_type, 'val', class_name)
75     os.makedirs(train_dir, exist_ok=True)
76     os.makedirs(val_dir, exist_ok=True)
77
78     extracted = 0
79     for img_id in img_ids:
80         if extracted >= 500:  # because the total images are 500. 400 for training and
               100 for validation
81             break
82         img_info = coco.loadImgs(img_id)[0]
83         ann_ids = coco.getAnnIds(imgIds=img_id, iscrowd=False)
84         anns = coco.loadAnns(ann_ids)
85
86
87         obj_counts = {}
88         for ann in anns:
89             cat_name = coco.loadCats(ann['category_id'])[0]['name']
90             obj_counts[cat_name] = obj_counts.get(cat_name, 0) + 1
91
92         if multiple_categories:
93             valid = (obj_counts.get(class_name, 0) >= min_instances) and (len(obj_counts
                   ) >= 2)
94         else:
95             valid = (obj_counts.get(class_name, 0) >= min_instances)
96             if max_instances is not None:
97                 valid = valid and (obj_counts.get(class_name, 0) <= max_instances)
98
99         if valid:
100            # Saving
101            save_dir = train_dir if extracted < 400 else val_dir
102            img_path = os.path.join(image_dir, img_info['file_name'])
103            img = Image.open(img_path).resize((64, 64))
104            img.save(os.path.join(save_dir, img_info['file_name']))
105            extracted += 1
106
107    # If the threshold is not met, then augment images.
108    if extracted < 400:
109        augment_images(train_dir, train_dir, 400)
```

```
    if extracted < 500:
        augment_images(val_dir, val_dir, 100)
```

Extracting the images

```
# Create datasets
for object_class in classes:
    # Dataset 1: Single instance
    extract_images_for_class(object_class, 'single_instance', 1, 1)
    # Dataset 2: Multi instance of same object
    extract_images_for_class(object_class, 'multi_instance_same', 2, None)
    # Dataset 3: Multi instance of different objects
    extract_images_for_class(object_class, 'multi_instance_diff', 1, None,
        multiple_categories=True)
print ("Successfully extracted the images")
```

Creting the Dataset and Dataloader

The following creates a dataset to be used by the dataloader.

```
# Dataset creation
class COCODataset(Dataset):
    def __init__(self, parent_dir, split, transform=None):
        self.transform = transform
        self.image_paths = []
        self.labels = []
        class_to_idx = {object_class: i for i, object_class in enumerate(classes)}

        for object_class in classes:
            cls_dir = os.path.join(parent_dir, split, object_class)
            for img_file in os.listdir(cls_dir):
                self.image_paths.append(os.path.join(cls_dir, img_file))
                self.labels.append(class_to_idx[object_class])

    def __len__(self):
        return len(self.image_paths)

    def __getitem__(self, idx):
        img = Image.open(self.image_paths[idx]).convert('RGB')
        if self.transform:
            img = self.transform(img)
        return img, self.labels[idx]
```

2.2 Verifying that each dataset has 400 training and 100 testing images

Training and validation count

To ensure that the training and validation numbers are correct (400 images per class for training, 100 images per class for validation), we add a statement that perform augmentation whenever there is deficiency in those thresholds. The augmentation basically make new versions of the same images but with some random rotation, etc. The augmentation code has been included in the **Dataset Creation** subsection previously.
Table. 2 and Table. 3 show that we have successfully extracted the correct count for each class.

We use the following ode to calculate the dataset:

```
#Print the classes counts
def create_count_tables(datasets):
    train_table = []
    val_table = []

    for dataset in datasets:
        train_row = [dataset]
        val_row = [dataset]

        for cls in classes:
```

```
            # Count training images
            train_path = os.path.join(output_dir, dataset, 'train', cls)
            train_count = len(os.listdir(train_path))
            train_row.append(train_count)

            # Count validation images
            val_path = os.path.join(output_dir, dataset, 'val', cls)
            val_count = len(os.listdir(val_path))
            val_row.append(val_count)

        train_table.append(train_row)
        val_table.append(val_row)

    # Create headers
    headers = ["Dataset Type"] + classes

    print("\nTraining Data Counts:")
    print(tabulate(train_table, headers=headers, tablefmt="grid"))

    print("\nValidation Data Counts:")
    print(tabulate(val_table, headers=headers, tablefmt="grid"))

# Generate tables
create_count_tables(datasets)
```

Dataset Type	airplane	train	bird	giraffe	clock
single_instance	400	400	400	400	400
multi_instance_same	400	400	400	400	400
multi_instance_diff	400	400	400	400	400

Table 2: Training Data Counts

Dataset Type	airplane	train	bird	giraffe	clock
single_instance	100	100	100	100	100
multi_instance_same	100	100	100	100	100
multi_instance_diff	100	100	100	100	100

Table 3: Validation Data Counts

2.3 Data visualization

to visualize the datasets, we plot 3 images from all the five categories for all the 3 datasets.

```
def plot_dataset_samples(datasets):
    # Create one plot per dataset type
    for dataset_idx, dataset in enumerate(datasets):
        fig, axes = plt.subplots(len(classes), 3, figsize=(15, 20))

        # Main title for each plot
        dataset_names = {
            'single_instance': 'Single Instance Dataset',
            'multi_instance_same': 'Multi-Instance (Same) Dataset',
            'multi_instance_diff': 'Multi-Instance (Different) Dataset'
        }
        fig.suptitle(f"{dataset_names[dataset]} Samples\n"
                     "(Rows: Object Categories, Columns: Random Samples)",
                     fontsize=14,
                     fontweight='bold',
                     y=1.02)

        # Configure grid layout
        plt.subplots_adjust(hspace=0.3, wspace=0.1)

        # Remove empty axes
        for ax in axes.flat:
            ax.axis('off')
            ax.set_xticks([])
            ax.set_yticks([])

        # Plot images for each category
        for row_idx, class_name in enumerate(classes):
            img_dir = os.path.join(output_dir, dataset, 'train', class_name)
            all_images = [f for f in os.listdir(img_dir) if f.endswith('.jpg')]
            random.shuffle(all_images)

            # Plot 3 samples across columns
            for col_idx in range(3):
                ax = axes[row_idx, col_idx]
                if col_idx < len(all_images):
                    img = Image.open(os.path.join(img_dir, all_images[col_idx]))
                    ax.imshow(img)
                    ax.set_title(f"Sample {col_idx+1}", fontsize=9, pad=4)

                # Add row labels only to first column
                if col_idx == 0:
                    ax.text(-0.3, 0.5, class_name,
                            rotation=0,
                            fontsize=12,
                            va='center',
                            ha='right',
                            transform=ax.transAxes)

        plt.tight_layout()
        plt.show()

datasets = ['single_instance', 'multi_instance_same', 'multi_instance_diff']
plot_dataset_samples(datasets)
```

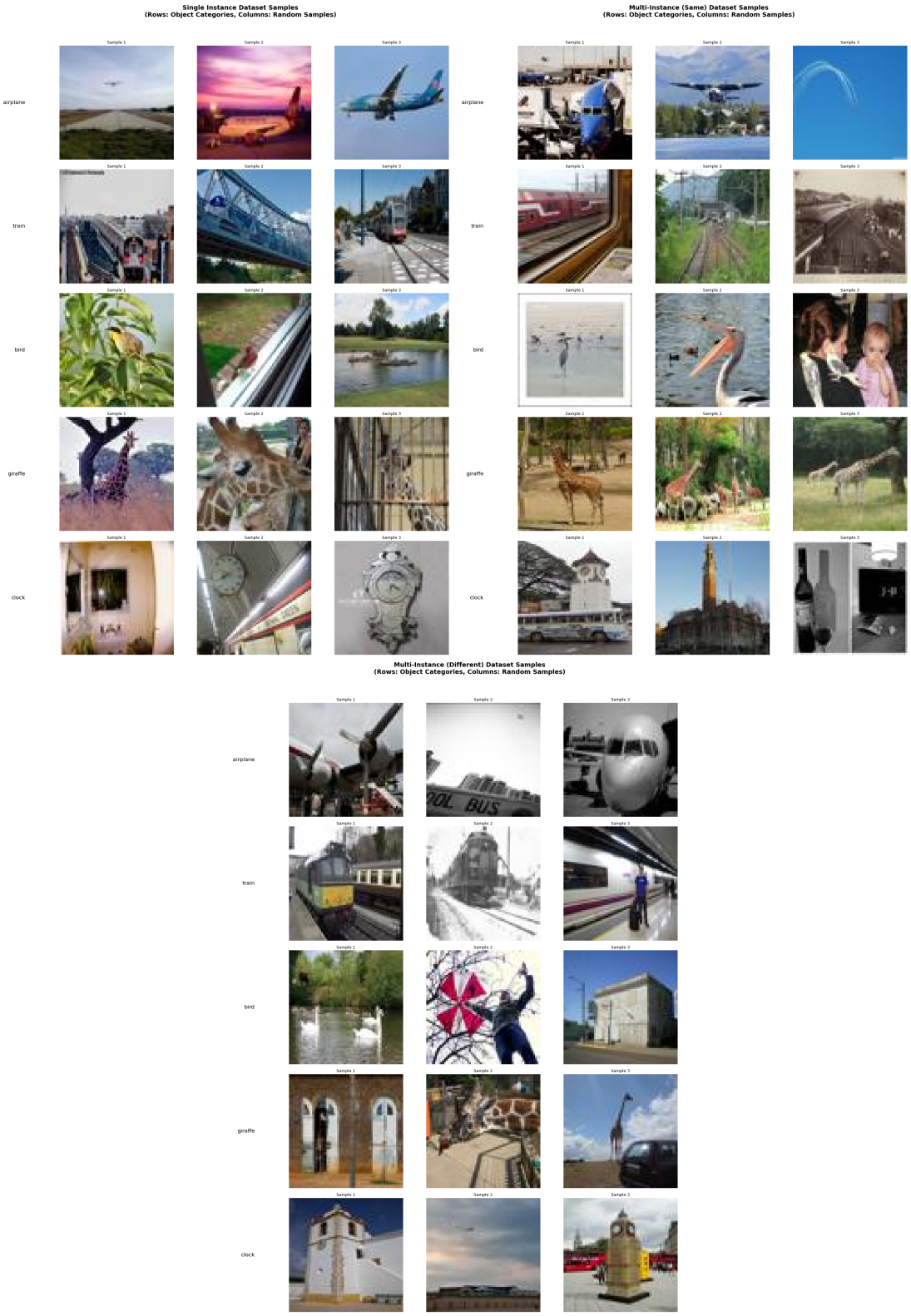

Figure 3: From left to right, examples of the **Single instance**, followed by **Multiple instances, same** and **Multiple instances, differnet** images.

2.4 Tables for Train and Validation Count

The count tables for training and validation have been included in Table. 2 and Table. 3 respectively.

2.5 Network implementation

The following is the network implementation as instructed.

```
# Model Definition
class HW4Net(nn.Module):
    def __init__(self):
        super().__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(nn.functional.relu(self.conv1(x)))
        x = self.pool(nn.functional.relu(self.conv2(x)))
        x = x.view(x.size(0), -1)
        x = nn.functional.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

Model Parameters

To calculate the model parameters, we use the following code:

```
def get_num_parameters(model):
    total_params = 0
    trainable_params = 0
    for param in model.parameters():
        num_params = param.numel()
        total_params += num_params
        #Check if the parameters is trainable.
        #Our model do not have non-trainable paremeters, so we expect it to be the same
            as total
        if param.requires_grad:
            trainable_params += num_params
    return total_params, trainable_params
```

	Number of Conv. Layers	Number of parameters
HW4Net (baseline)	2	406885
HWNet1 (bonus)	5	1119877
HWNet2 (bonus)	10	1706133

Table 4: The number of parameters (trinable) for each of the networks we will be using. The baseline network is the first network **HW4Net**. The other two are the network used for the bonus part.

Dataset and Dataloader

Using the Dataset and Dataloader classes from *torch.utils.data.Dataset* and *torch.utils.data.Dataloader*, I wrote the following classes to create the dataset and load it.

```
# Dataset creation
class COCODataset(Dataset):
    def __init__(self, parent_dir, split, transform=None):
        self.transform = transform
        self.image_paths = []
        self.labels = []
        class_to_idx = {object_class: i for i, object_class in enumerate(classes)}

        for object_class in classes:
            cls_dir = os.path.join(parent_dir, split, object_class)
            for img_file in os.listdir(cls_dir):
                self.image_paths.append(os.path.join(cls_dir, img_file))
                self.labels.append(class_to_idx[object_class])

    def __len__(self):
```

```
        return len(self.image_paths)

    def __getitem__(self, idx):
        img = Image.open(self.image_paths[idx]).convert('RGB')
        if self.transform:
            img = self.transform(img)
        return img, self.labels[idx]

# Transforms
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

model_names = ["HW1", "HW2", "HW4"]
results = {} #To hold the accuracies and taining loss
epochs = 100

for ds in datasets:
    train_dataset = COCODataset(os.path.join(output_dir, ds), 'train', transform)
    val_dataset = COCODataset(os.path.join(output_dir, ds), 'val', transform)
    train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
    val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)

    results[ds]={}
    for model_name in model_names:

    ....
    ....
    ....
```

Training routine

```


# Training Function
def train_model(model, train_loader, val_loader, dataset_name, epochs, model_name):
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    model.to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=1e-3, betas=(0.9, 0.99))

    train_losses = []
    val_accuracies = []

    for epoch in range(epochs):
        model.train()
        running_loss = 0.0
        num_batches = len(train_loader)
        for i, (inputs, labels) in enumerate(train_loader):
            inputs, labels = inputs.to(device), labels.to(device)

            #Clearing the gradients
            optimizer.zero_grad()

            #Prediction
            outputs = model(inputs)
            loss = criterion(outputs, labels)

            #Propagating the loss
            loss.backward()
            optimizer.step()
            running_loss += loss.item()

            #print("i = ", i)
            #train_losses.append(loss)

            #if (i+1) % 100 == 0:
            #    avg_loss = running_loss / 100
            #    train_losses.append(avg_loss)
            #    running_loss = 0.0
```

```
        avg_loss = running_loss / num_batches
        train_losses.append(avg_loss)



        # Validation
        correct = 0
        total = 0
        model.eval()
        with torch.no_grad():
            for inputs, labels in val_loader:
                inputs, labels = inputs.to(device), labels.to(device)
                outputs = model(inputs)
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()
        accuracy = 100 * correct / total
        val_accuracies.append(accuracy)

        if (epoch%10==0):

            print(f"Model: {model_name}. Dataset: {dataset_name} - Epoch {epoch+1},
                Training loss: {avg_loss:.4f} | Val Acc: {accuracy:.2f}%")

    return train_losses, val_accuracies

#Plot Confusion Matrix
def evaluate(model, val_loader, dataset_name, model_name):
    device = next(model.parameters()).device
    all_labels = []
    all_preds = []
    model.eval()
    with torch.no_grad():
        for inputs, labels in val_loader:
            inputs = inputs.to(device)
            outputs = model(inputs)
            _, preds = torch.max(outputs, 1)
            all_labels.extend(labels.cpu().numpy())
            all_preds.extend(preds.cpu().numpy())

    cm = confusion_matrix(all_labels, all_preds)
    plt.figure(figsize=(10,7))
    sns.heatmap(cm, annot=True, fmt='d', xticklabels=classes, yticklabels=classes)
    plt.title('Model: ' + model_name + ' - Dataset: ' + dataset_name + ' - Confusion
        Matrix')
    plt.show()

model_names = ["HW1", "HW2", "HW4"]
results = {} #To hold the accuracies and taining loss
epochs = 100

for ds in datasets:
    train_dataset = COCODataset(os.path.join(output_dir, ds), 'train', transform)
    val_dataset = COCODataset(os.path.join(output_dir, ds), 'val', transform)
    train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
    val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)

    results[ds]={}
    for model_name in model_names:

        if model_name == "HW1":
            model = HWNet1()
        elif model_name == "HW2":
            model = HWNet2()
        elif model_name == "HW4":
            model = HW4Net()

        total_params, trainable_params = get_num_parameters(model)
        print ("Current model: ", model_name, ". Total params: ", str(total_params))
        print ("Current model: ", model_name, ". Trainable params: ", str(
            trainable_params))


```

```
        print("Model: ", model_name, ". Training on ", ds, " dataset.")

        #Training
        train_loss, val_acc = train_model(model, train_loader, val_loader, ds, epochs,
            model_name)

        #Appending the results
        #results[ds] = {'train_loss': train_loss, 'val_acc': val_acc}
        results[ds][model_name] = {'train_loss': train_loss, 'val_acc': val_acc}

        #Evaluating (confusion matrix)
        evaluate(model, val_loader, ds, model_name)

        # Plotting the loss
        plt.figure(figsize=(10, 5))
        #plt.plot(range(1, epochs + 1), train_loss, label='Train Loss')  # x-axis from 1
             to epochs
        plt.plot(train_loss, label='Train Loss')  # x-axis from 1 to epochs
        #plt.plot(range(1, epochs + 1), val_accuracies, label='Validation Accuracy') #If
             you also want to plot validation accuracy
        plt.xlabel('Epoch')
        plt.ylabel('Loss')
        plt.title('Model: ' + model_name + ' - Dataset: ' + ds + ' - Training Loss')
        plt.legend()
        plt.grid(True)
        plt.xticks(range(1, epochs + 1)) # Set x-axis ticks to be integers
        plt.show()

        #If you also want to plot the validation accuracy:
        plt.figure(figsize=(10, 5))
        #plt.plot(range(1, epochs + 1), val_acc, label='Validation Accuracy', color='red
            ')
        plt.plot(val_acc, label='Validation Accuracy', color='red')
        plt.xlabel('Epoch')
        plt.ylabel('Validation Accuracy')
        plt.title('Model: ' + model_name + ' - Dataset: ' + ds + ' - Validaation
            accuracy')
        #plt.title(ds+' with model: ' + model_name + ' - Validation Accuracy vs. Epoch')
        plt.legend()
        plt.grid(True)
        plt.xticks(range(1, epochs + 1)) # Set x-axis ticks to be integers
        plt.show()
```

3 Baseline HW4Net() Performance

The baseline netwwork only contained 2 convolutional layers with limited number of filters (16, 32). The performance on such a CNN is limited and usually fails to learn the finer details that allows is to descrimate between classes of objects. We anticipate that the other two (included in the bonus) network to perform much better.

In my case, the network performed best when it was presented with multiple instances of the same object, followed closely by single instance, and performed noticably worse on the multi instances with different objects, which suggests that the more comlicated scenario where multiple different objects are present may need bigger network or more confolutional layers or differnet approach. And having multiple instances of the same object may assist the network to generalize as it see multiple instances with different conditions of the same class.

3.1 Baseline HW4Net() Single Instance Loss

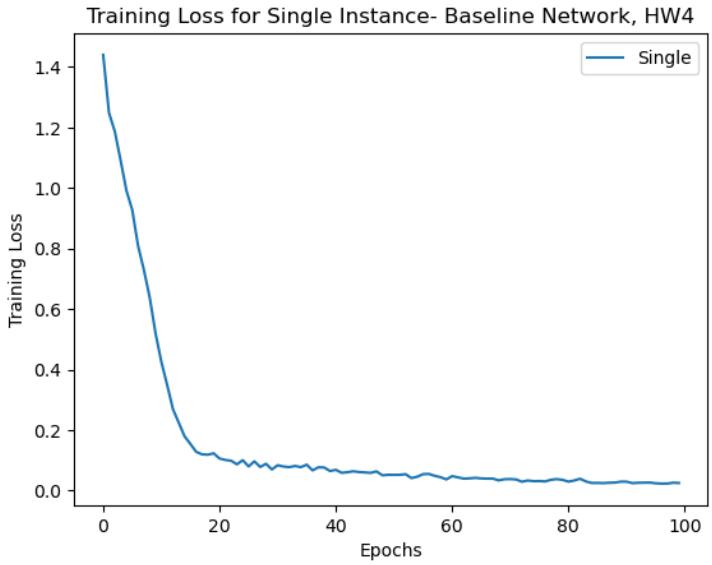

Figure 4: The baseline **HW4Net()** architecture with **Single Instance**

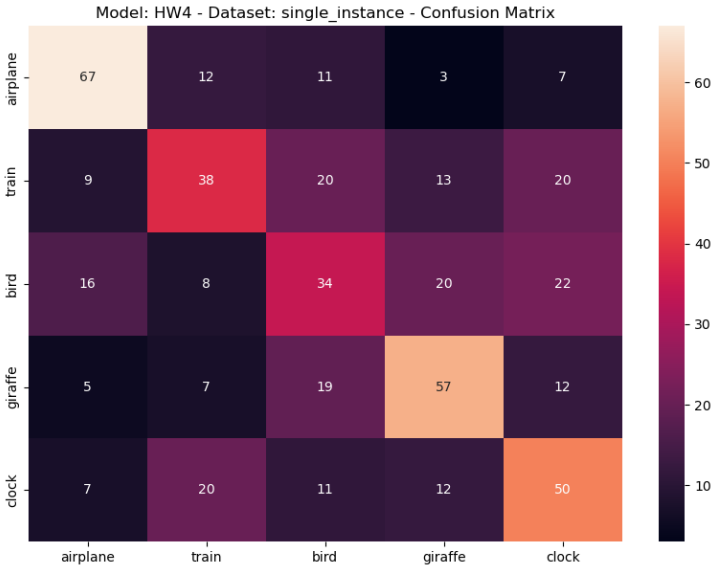

Figure 5: Confusion matrix for the single instance dataset on baseline HW4 network

3.2 Baseline HW4Net() Multi Instance Same Loss

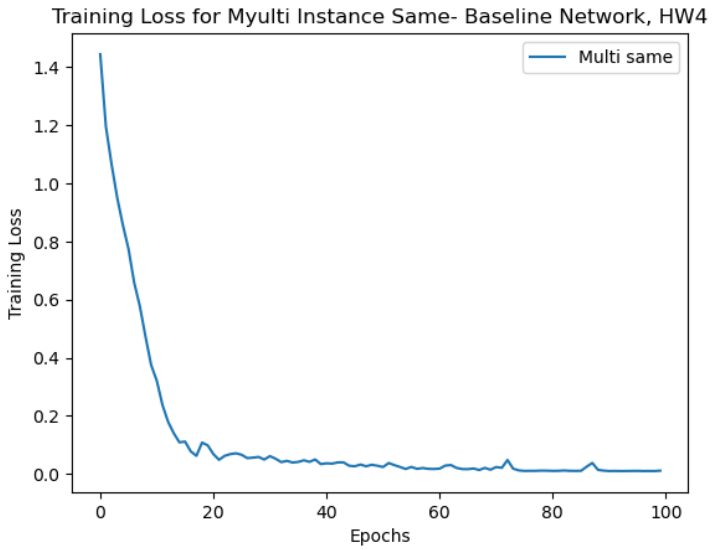

Figure 6: The baseline **HW4Net()** architecture with **Multi Instance Same**

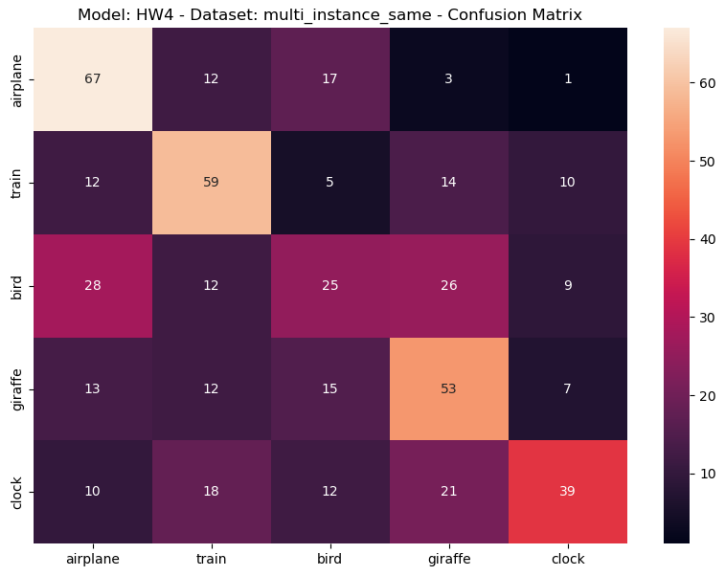

Figure 7: The baseline Confusion matrix for the multi instance same object dataset on baseline HW4 network

3.3 Baseline HW4Net() Multi Instance Different Loss

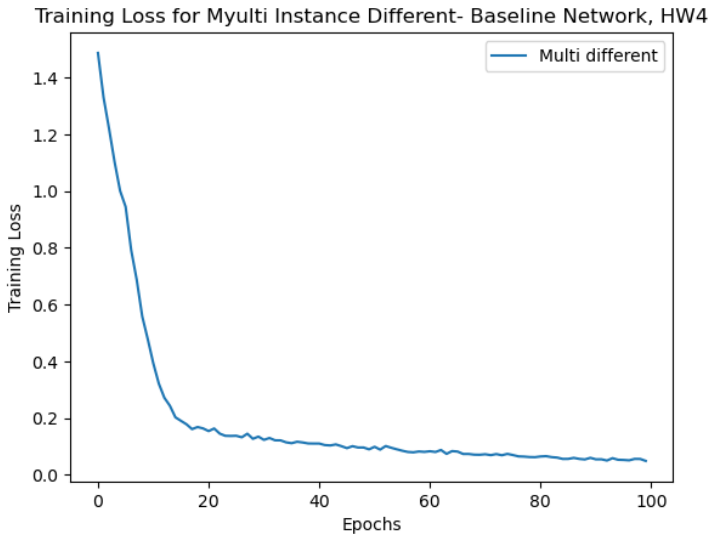

Figure 8: The baseline **HW4Net()** architecture with **Multi Instance Different**

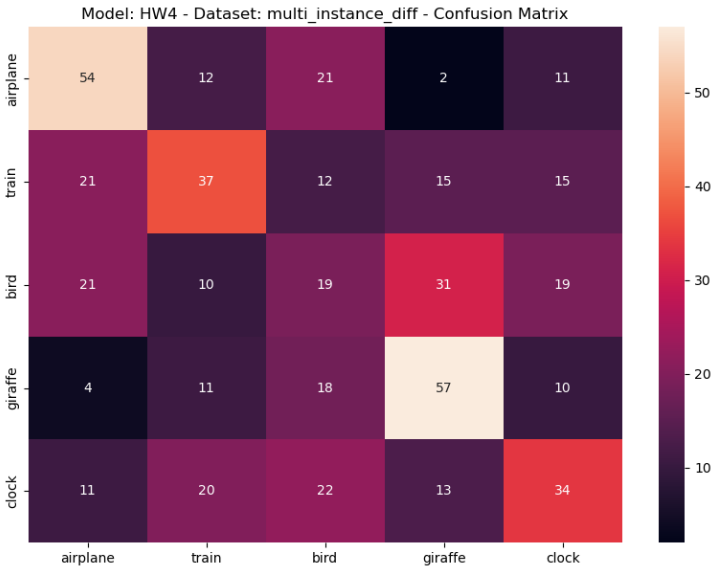

Figure 9: The baseline **HW4Net()** architecture confusion matrix for Multi Instance Same

3.4 Baseline HW4Net() Comparison for all Datasets.

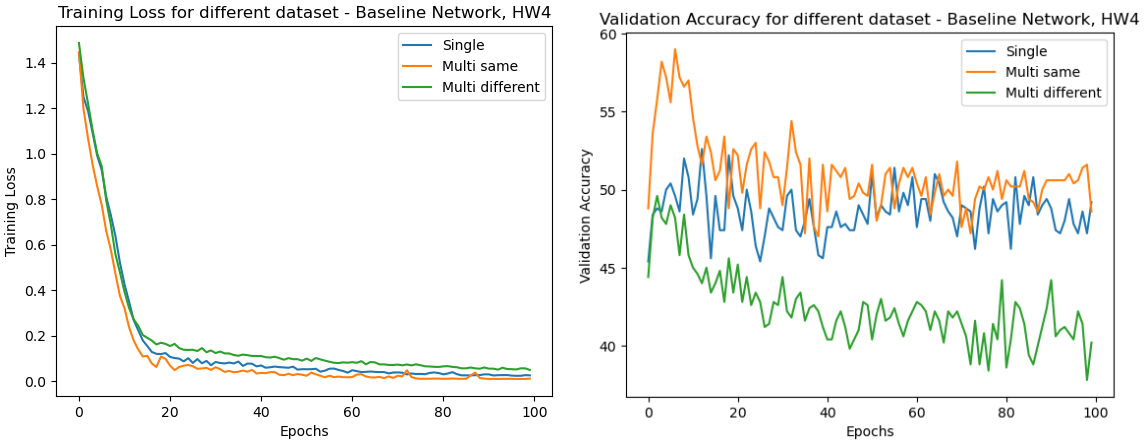

Figure 10: The baseline **HW4Net()** architecture. **Left**: loss value for the 3 different datasets as a function of epochs. **Right**: validation accuracy.

	Number of parameters	Accuracy at the end	Maximum Accuracy
Single Instance	406885	**49.200%**	52.600%
Multi Same	406885	48.600 %	**59.000%**
Multi Different	406885	40.200%	49.600%

Table 5: Baseline network **HW4Net**) number of parameters, and validation accuracy at the end final epoch and the maximum validation accuracy acheived for each dataset. The **boldface** value is the highest in each column.

Misclassified Images

The following are examples of images that have been misclassified for each dataset. The images are normalized due to the transformation so they may appear darker than the original images. We'll try to hypothesize why might those images been misclassified.

1. Single Instance The images are of planes but have been categorized as: cloc, train, and train respectively. We can see the resemblence of a train car in the later two images (middle, and right image) which may look vaguely similar to the model. The clock however may be harder to interpret. However, we should bear in mind that the dataset include huge clocks that are installed in public spaces and plazas (think Big Ben in London). So those may share some edge features with the plane image on the left.

Figure 11: Misclassified images for the **Single instance** dataset

2. Multiple Instance, same The random selected images were again of planes, but this time they were categorized as train, giraffe, and train respectively. The image on the left have some repeated pattern which is often seen and associated with the repeated pattern of the train cars and rails, so the model might have mistaken it for a train accordingly. The middle image is harder to interpret, but if you look closley, there is a bulge (part of the plane) that is visible with a sky background. Usually giraffe have similar looking bulge (their faces) with the sky as a background, which may explain why the model thought it's a giraffe. The third image may again resemble some overall shape and color of some trains.

Figure 12: Misclassified images for the **Multi instance, same** dataset

3. Multiple Instance, different Those images might be the best to interpret compared with the prevoius datasets. The images again are for planes but have been misclassified as a bird, a clock, and a clock respectively. It is easy to see why the network could mistake the airplane for a bird, both of them are in the sky, and frankly look similar when they only occupy a small portion of the image. The middle image is more interesting, and even for humans, we could easily interpret this as a big clock (like those installed in big plazas), but with careful inspection, the propellers on the side tell us that it's a plane, something the model did not pick up during training. The thir image is harder to interpret but it may have shared some features or edges or dials with a clock and hence confused the network.

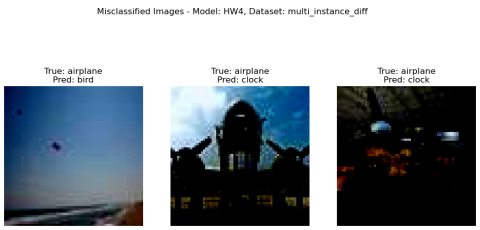

Figure 13: Misclassified images for the **Multi instance, different** dataset

3.4.1 By observing your classification accuracies, which dataset think are more difficult to correctly differentiate and why?

The **multi instances with different objects** was the most difficult to correctly classify. I would hypothesize that those objects are often appear cropped or at the corner of the image or only part of them shows up in the image but they are still counted as present. For example, often only part of the plane is showing in an image and it is still considered present, which presents a challenege for a small network like the baseline network we used with only 2 convolutional layers.

To account for those, we may need larger network, or convolutional layers with deeper filters (more number of filters) to extract finer descrimations between classes.

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

The **bird** class consistantly performed worst by a big margin compared to the other classes.

By examining the bird images, birds come in many different shapes, colors, and sizes. They are also not as deterministic in terms of pose, and location as images of planes or cars. They also appear in a wide variety of surrounding, from trees, to beaches, etc., all of which make them harder to classify.

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

Make a bigger network with more convolutional layers and larger number of filters, and expand the augmentations to include random rotations, cropping, etc.

4 Bonus: Networks with more convolutional layers

As one may anticipate, using a CNN with more convolutional layers may hypothetically improve the performance. Here we implement two networks to test such hypothesis and provide the training loss, validation accuracy and some analysis. The two networks we implement are:

- **HWNet1()**: with 5 convolutional layers
- **HWNet2()**: with 10 convolutional layers

```
# Models for the bonus extra credit
class HWNet1(nn.Module):
    def __init__(self):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 16, 3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),

            nn.Conv2d(16, 32, 3, padding=1),
            nn.ReLU(),
            nn.Conv2d(32, 32, 3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),

            nn.Conv2d(32, 64, 3, padding=1),
            nn.ReLU(),
            nn.Conv2d(64, 64, 3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),
        )
        self.classifier = nn.Sequential(
            nn.Linear(64 * 8 * 8, 256),
            nn.ReLU(),
            nn.Linear(256, 5)
        )

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)
        x = self.classifier(x)
        return x

class HWNet2(nn.Module):
    def __init__(self):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 16, 3, padding=1),
            nn.ReLU(),
            nn.Conv2d(16, 16, 3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),

            nn.Conv2d(16, 32, 3, padding=1),
            nn.ReLU(),
            nn.Conv2d(32, 32, 3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),

            nn.Conv2d(32, 64, 3, padding=1),
            nn.ReLU(),
            nn.Conv2d(64, 64, 3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),

            nn.Conv2d(64, 128, 3, padding=1),
            nn.ReLU(),
            nn.Conv2d(128, 128, 3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),

            nn.Conv2d(128, 256, 3, padding=1),
```

```
            nn.ReLU(),
            nn.Conv2d(256, 256, 3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),
        )
        self.classifier = nn.Sequential(
            nn.Linear(256 * 2 * 2, 512),
            nn.ReLU(),
            nn.Linear(512, 5)
        )

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)
        x = self.classifier(x)
        return x
```

Bonus: HWNet1() with 5 convolutional layers - Performance

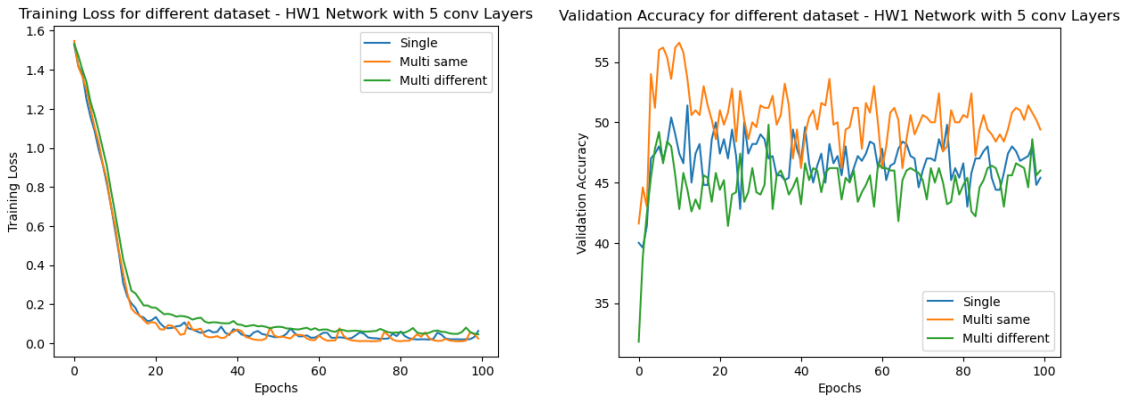

Figure 14: The bonus **HWNet1()** architecture with 5 convolutional layers. **Left**: loss value for the 3 different datasets as a function of epochs. **Right**: validation accuracy.

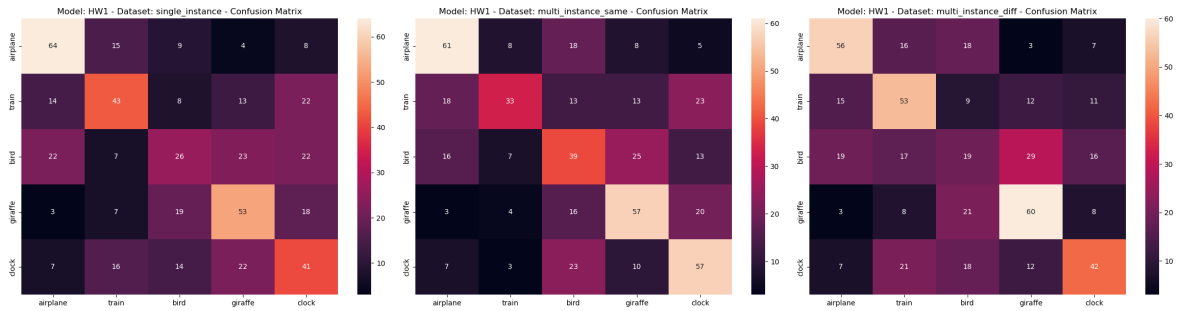

Figure 15: The bonus **HWNet1()** architecture with 5 convolutional layers. The three matrices from left to right are the confusion matrices for the single instance, multi instance same object, and multi instance different object.

Bonus: HWNet2() with 10 convolutional layers - Performance

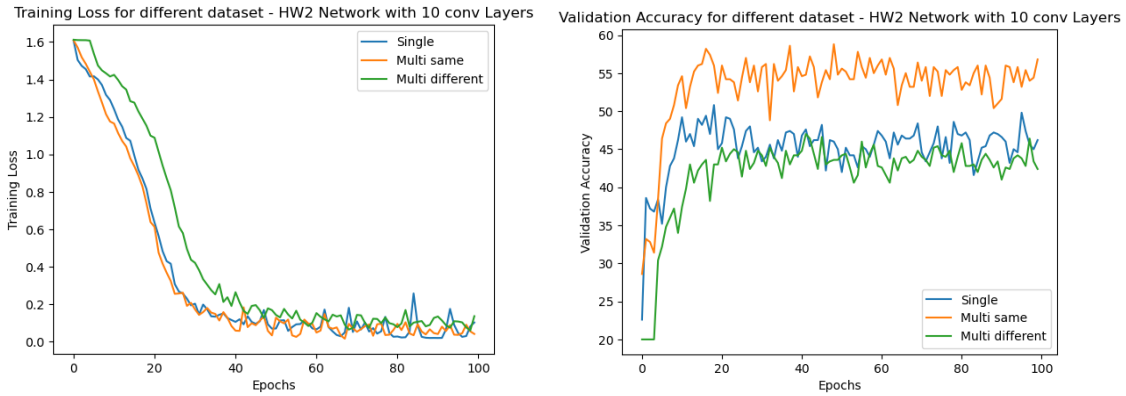

Figure 16: The bonus **HWNet2()** architecture with 10 convolutional layers. **Left**: loss value for the 3 different datasets as a function of epochs. **Right**: validation accuracy.

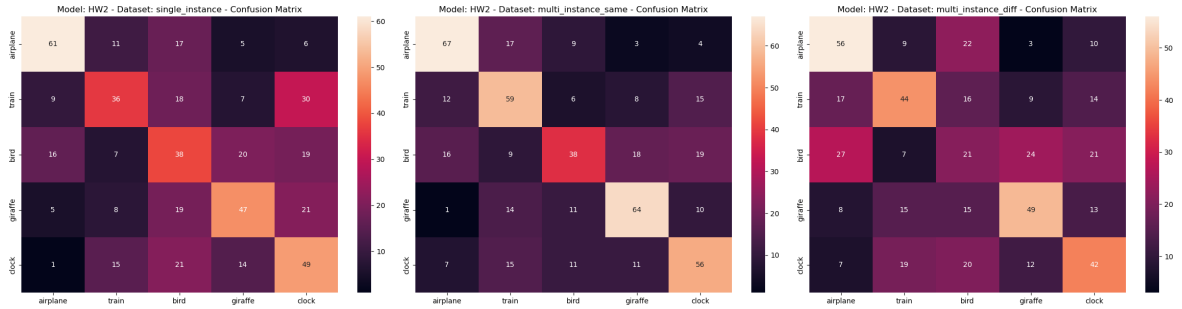

Figure 17: The bonus **HWNet2()** architecture with 10 convolutional layers. he three matrices from left to right are the confusion matrices for the single instance, multi instance same object, and multi instance different object.

Anslysis of the bonus models compared to the baseline

Network	Number of Parameters	Accuracy at the end
H1 (5 conv)	1119877	45.400%
H2 (10 conv)	1706133	46.200%
H4 Baseline	406885	**49.200%**

Table 6: Validation Accuracy for H1, H2, and H4 with for the **Single Instance** case. **Boldface** values are highest in each column.

	Number of Parameters	Accuracy at the end
H1 (5 conv)	1119877	49.400%
H2 (10 conv)	1706133	**56.800%**
H4 Baseline	406885	48.600%

Table 7: Validation Accuracy for H1, H2, and H4 with for the **Multi Instance Same** case. **Boldface** values are highest in each column.

	Number of Parameters	Accuracy at the end
H1 (5 conv)	1119877	**46.000%**
H2 (10 conv)	1706133	42.400%
H4 Baseline	406885	40.200%

Table 8: Validation Accuracy for H1, H2, and H4 with for the **Multi Instance Different** case. **Boldface** values are highest in each column.

4.1 Bonus: Analysis of the bonus networks

Question: Does a deeper network always result in better performance?
Based on my experiment above with the architecture I used, the accuracy for the multi instance (same and different) both improved when using a larger network with more convolutional layers which suggests that if the task is more complicated, a more complex network might be needed to capture the underlying complexity of the data.

However, based on my limited experiment here, in the single instance case, the baseline network was sufficient and perofrmed better than the other two. This could have been just a random coincedence, or an indication that larger network may not always be better, or they may need to train for longer to achieve comparable results on less complicated tasks.

One improvement that I could also suggest is fiddling with the number of filters, as this may result in learning finer features and improving the network classifcation accuracy.

References

[1] Avinash Kak. Bme 646 and ece 60146. deep learning. `https://engineering.purdue.edu/DeepLearn/`, 2025. Accessed: February 2025.

[2] Avinash Kak. Dlstudio-2.5.1. `https://engineering.purdue.edu/kak/distDLS/`, 2025. Accessed: February 2025.

[3] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13*, pages 740–755. Springer, 2014.