Final Report Graduate Project

Christoph Hagenauer*

May 30, 2025

Contents

*University of South Carolina Beaufort

1 Introduction

The focus of this project is on building a dataset with high-quality photos from my personal library. I achieved 16 categories with a minimum of 100 images per category. These categories mainly are images of animals, but also include images of different flowers. The goal of this dataset was to be a real-world example of the type of images people take on a regular basis. Furthermore, I provide a dataloader for Pytorch, and I evaluate the dataset on pretrained models such as GoogLeNet.

1.1 Timeline and Milestones

I divided the project into three main Sprints:

- Sprint 1 (March 13 – April 13): Take photos, gather existing photos, as well as sorting them in their respective categories.
- Sprint 2 (April 14 – April 20): Export the images, build the dataloader, and perform an evaluation with existing models within PyTroch.
- Sprint 3 (April 20 - May 2): Write a report and evaluate the findings.

1.2 Technical Approach

To organize the images I used Excire Foto, which has some built in tools such as similarity search and AI based image recognition, however I had to fine-tune the images within the categories. I used python as my programming language, Visual Studio Code as my IDE, and Anaconda and pip as my package managers. With python I utilized libraries such as Pytorch, Numpy, Pandas, Matplotlib, Seaborn, Scikit-learn, and PIL. Pytorch was used to build the dataloader and use existing CNN models. Numpy and Pandas were used to store the data and perform matrix operations. Matplotlib and Seaborn were utilized for visualizations and Scikit-learn for preprocessing and metrics. The PIL library was used to load the images.

2 The Dataset

The dataset consist of 16 categories, which are: Bee, Blackbird, Butterfly, Chamois, Deer, Duck, Flower, Frog, Highland Cow, Kingfisher, Rose, Seagull, Squirrel, Tit, Tulip, and Woodpecker. Each category has at least 100 images; some exceed 200. In total, the dataset has $2,516$ images. A sample of images is shown in Figure 1.

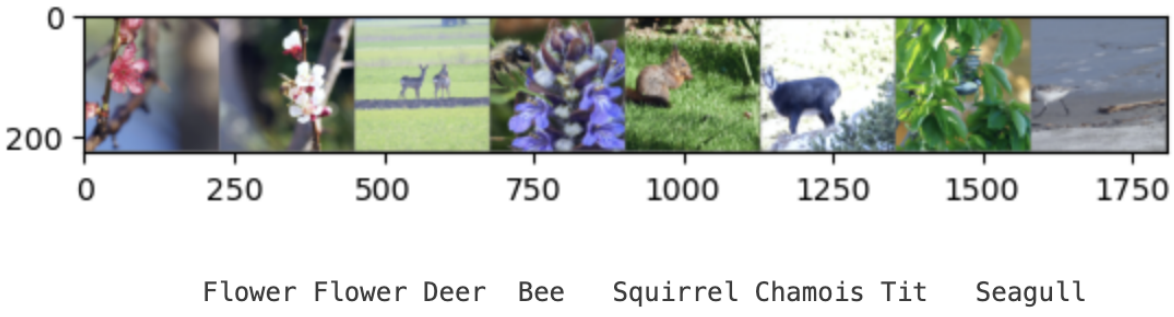

Figure 1: Sample of images

2.1 Data Wrangling

Since I collected the photos, I did not have any missing values. Therefore, no imputation had to be performed. The only preprocessing steps required were to transform and resize the photos. Each image has a long edge of $2,048$ pixels. The second edge was automatically set, since I only specified the long edge when exporting the photos. Therefore, the images have to be resized so that they all have the same dimensions, as is required by CNN models. Furthermore, resizing was performed to use less system resources, since the photos are of relatively high quality.

3 DataLoader class

In this section, I share the options and details about the dataloader class. For full code details, please see Appendix A. The requirement for this dataset is to be easily usable within the Pytroch environment. Therefore, after reading the image with the PIL library, I transform it using transformers available in Pytorch. For the transformation the user can set the image size, the crop, as well as mean and standard deviation. However, I also set default values in accordance with the requirements for Pytorch's built-in CNNs, which can be found here [1]. The class also implements default methods such as returning the number of samples *(__len__)* and returning the the images *(__getitem__)*. Furthermore, I implemented a method to plot a few samples, where the number of sample to be plot can be specified. I also included a method to return the class names available in the dataset. These functionalities are demonstrated in Appendix B

3.1 Evaluation of dataset using CNNs

After finishing the dataloader class, I evaluated the dataset on pretrained models available in Pytorch [1]. The code for this evaluation is available in Appendix C. I evaluated four models: GoogleLeNet, ResNet18, MobileNet V2, and EfficientNet B0. For all of these models I used starting weights available in Pytorch. For each model I share the training and testing accuracy and loss, which are visualized in the following figures. Especially, the MobileNet V2 (Figure 4) and EfficientNet B0 (Figure 5) show very promising results, while ResNet18 (Figure 3) and GoogLeNet (Figure 2) are performing a little worse and a lot worse, respectively. Upon examination of the figures, I was able to observe a high accuracy, which I am very happy with. This means the models are able to fine-tune their weights and adapt to my dataset. However, I also saw some overfitting which means that there might be optimization steps to take within the models, or by even designing my own. However, the design of my own CNN is left for future work.

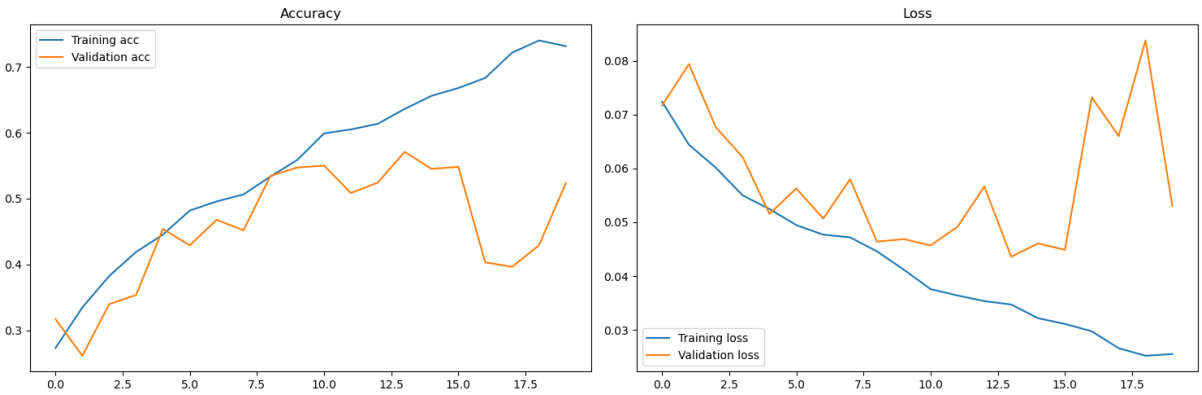

Figure 2: GoogLe Net

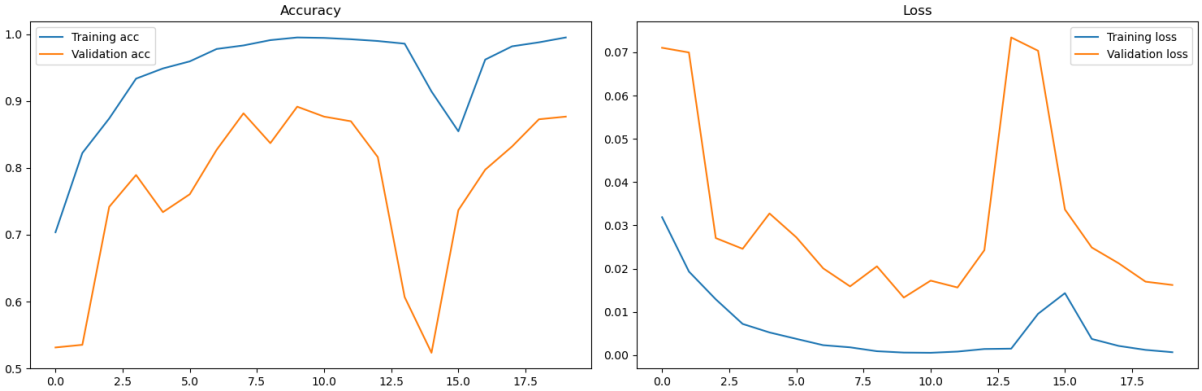

Figure 3: ResNet 18

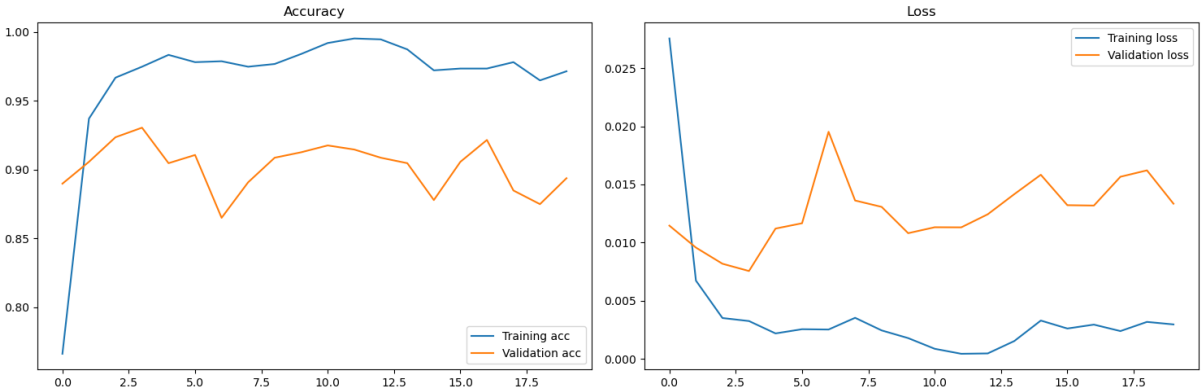

Figure 4: MobileNet V2

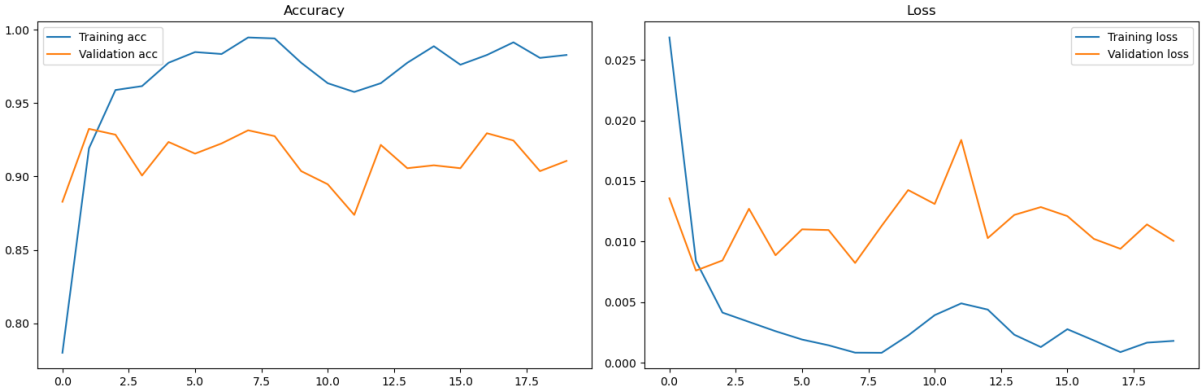

Figure 5: EfficientNet B0

4 Lessons learned

It is quite difficult to collect such a large sample of images. Nonetheless, this dataset is still very small compared to CIFAT-10, which, for example, has $60,000$ images. The difficulty lies in sorting and then labeling the images. Although Excire Photo was of great help, it still required several hours to sort and label all the images.

That being said, I had a lot of fun taking the pictures and building the dataloader class! Furthermore, it was a great exercise for me to familiarize myself with the Pytorch API, which I believe I will be using more in the future.

References

[1] PyTorch Team. Torchvision models. `https://pytorch.org/vision/0.12/models.html`, 2022. Accessed: 2025-03-24.

A DataLoader Class

```
"""
    Course: CSCI B522
    Assignment: Graduate Project - image Dataloader
    Author: Christoph Hagenauer
"""

import os
from PIL import Image
import random
from matplotlib import image
import matplotlib.pyplot as plt
import numpy as np
from sklearn.preprocessing import LabelEncoder
import torchvision.transforms as T
import torch
from multiprocessing import Pool, cpu_count
from itertools import repeat

# mean = np.array([0.485, 0.456, 0.406])
# std = np.array([0.229, 0.224, 0.225])

def process_image(file, img_size, img_crop, mean, std):
    """
        Normalize and Resize Images
    """
    normalize = T.Normalize(mean=mean, std=std)
    transform = T.Compose([T.Resize(img_size), T.CenterCrop(img_crop), T.ToTensor(), normalize])
    photo = Image.open(file).convert("RGB")
    img_tensor = transform(photo)
    return img_tensor, file

class Load_Images():
    """
    Load all the images and return Dataset
    Specify the number of sample per class with num_samples_per_class
    The image size with img_size
    The crop into the image with img_crop (You have to calculate the
    crop as the image size -> img_size - img_crop = crop)
    Furthermore mean and standard deviation can be provided for
    the image transformation.
    """
    def __init__(self, num_samples_per_class=512, img_size=256, img_crop=224,
                 mean = np.array([0.485, 0.456, 0.406]),
                 std = np.array([0.229, 0.224, 0.225])):

        self.mean = mean
        self.std = std

        if img_size != 256 and img_crop == 224:
            img_crop = img_size * 0.9

        folder_path = 'Photos_Model'
        photos = os.listdir(folder_path)
        files = [os.path.join(folder_path, photo) for photo in photos]

        processes = max(cpu_count(), 1)

        with Pool(processes=processes) as pool:
            results = pool.starmap(process_image, zip(files, repeat(img_size),
                                    repeat(img_crop), repeat(mean), repeat(std)))

        features = []
        targets = []
        classes = set()

        count_per_class = {}

        for img_tensor, file in results:
            category = os.path.basename(file).split('-')[0]
            if category not in count_per_class:
                count_per_class[category] = 1
            else:
```

```
            count_per_class[category] += 1
        if count_per_class[category] <= num_samples_per_class:
            features.append(img_tensor)
            targets.append(category)
            classes.add(category)
        else:
            break

    le = LabelEncoder()
    self.classes = np.array(sorted(classes)) # Sorting is necessary to preserve order and be able to
 map encoded targets
    self.encoded_classes = le.fit_transform(self.classes)
    self.encoded_targets = le.transform(targets)
    self.features = features

# Displaying sample Photos
def plot_sample(self, num=5):
    """
    Plot sample images from dataset
    specify how many with num
    """
    for i in range(num):
        idx = random.randint(0, len(self.features)-1)
        # print(self.features[idx].shape)
        img = self.features[idx] * torch.tensor(self.std).view(3, 1, 1) + torch.tensor(self.mean).view
 (3, 1, 1)
        npimg = img.numpy()
        npimg = np.clip(npimg, 0, 1)
        print(self.classes[self.encoded_targets[idx]])
        plt.imshow(np.transpose(npimg, (1, 2, 0)))
        plt.show()


def get_classes(self):
    """
    Returns the class names in the dataset
    """
    return self.classes, self.encoded_classes

def __len__(self):
    """
    Returns the number of samples in the dataset
    """
    return len(self.encoded_targets)

def __getitem__(self, idx):
    """
    Returns features and labels for given index
    """
    return self.features[idx], self.encoded_targets[idx]
```

B Testing Dataloader

```
import random
random.seed(42)

import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import train_test_split
from torch.utils.data import DataLoader
from torchvision.utils import make_grid

from load_images import Load_Images # My dataloader class

img_data = Load_Images(num_samples_per_class=1000)

classes, encoded_classes = img_data.get_classes()
print(len(classes), classes, encoded_classes)

print(img_data.encoded_targets)
print(len(img_data))
print(img_data.features[1].shape)

batch_size=8

trainset, valset = train_test_split(img_data, train_size=0.7, random_state=42, stratify=img_data.
    encoded_targets)

print(len(trainset))
print(len(valset))

trainloader = DataLoader(trainset, batch_size=batch_size)
valloader = DataLoader(valset, batch_size=batch_size)

print(f"Distribution of data between classes: {np.bincount(img_data.encoded_targets)/len(img_data.
    encoded_targets)*100}.")
img_data.plot_sample(10)

# This is from Pytorch CIFAR10 notebook to check if my dataloader works
def imshow(img):
    img = img * 0.229 + 0.485     # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()

# get some random training images
dataiter = iter(valloader)
images, labels = next(dataiter)

print(images.shape)
print(labels)

# show images
imshow(make_grid(images))
# print labels
print(' '.join(f'{classes[labels[j]]:5s}' for j in range(batch_size)))
```

C CNN Testing

```
import os
import random
random.seed(42)
import time
import torch
import torch.nn as nn
from torchinfo import summary
from torch.utils.data import DataLoader
from torchvision.models import resnet18, ResNet18_Weights
from torchvision.models import efficientnet_b0, EfficientNet_B0_Weights
from torchvision.models import mobilenet_v2, MobileNet_V2_Weights
from torchvision import models
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import train_test_split
from load_images import Load_Images

img_data = Load_Images(num_samples_per_class=1000)
classes, encoded_classes = img_data.get_classes()
batch_size=8
trainset, valset = train_test_split(img_data, train_size=0.7, random_state=42, stratify=img_data.
    encoded_targets)
print(len(trainset))
print(len(valset))
trainloader = DataLoader(trainset, batch_size=batch_size)
valloader = DataLoader(valset, batch_size=batch_size)

def validate(device, net, dataloader,loss_fn=nn.NLLLoss()):
    net.eval()
    count,acc,loss = 0,0,0
    with torch.no_grad():
        for features,labels in dataloader:
            features, labels = features.to(device), labels.to(device)
            out = net(features)
            loss += loss_fn(out,labels)
            pred = torch.max(out,1)[1]
            acc += (pred==labels).sum()
            count += len(labels)
    return loss.item()/count, acc.item()/count

def train_epoch(device, net, dataloader, lr=0.01, optimizer=None, loss_fn=nn.CrossEntropyLoss()):
    optimizer = optimizer or torch.optim.Adam(net.parameters(), lr=lr)
    net.train()
    total_loss, acc, count = 0, 0, 0
    start_epoch_time = time.time()  # Start timer for the epoch

    for batch_idx, (features, labels) in enumerate(dataloader):
        batch_start = time.time()  # Start timer for batch
        features, labels = features.to(device), labels.to(device)
        net.to(device)
        optimizer.zero_grad()
        out = net(features)
        loss = loss_fn(out, labels)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
        _, predicted = torch.max(out, 1)
        acc += (predicted == labels).sum().item()
        count += len(labels)
        batch_time = time.time() - batch_start  # Time taken for this batch
        if batch_idx % 10 == 0: print(f"\nBatch {batch_idx+1}/{len(dataloader)} | {batch_time:.4f}s per
    batch", end="\r")
    epoch_time = time.time() - start_epoch_time  # Total time for the epoch
    samples_per_second = count / epoch_time  # Training speed
    return total_loss / count, acc / count, epoch_time, samples_per_second
```

```
def train(device, net, train_loader, test_loader, optimizer=None, lr=0.0001, epochs=10, loss_fn=nn.
    CrossEntropyLoss(), save_path="model_checkpoints"):
    os.makedirs(save_path, exist_ok=True)
    start_epoch = 0
    optimizer = optimizer or torch.optim.Adam(net.parameters(), lr=lr, weight_decay=1e-4)
    res = {'train_loss': [], 'train_acc': [], 'val_loss': [], 'val_acc': [], 'epoch_times': [], '
    samples_per_sec': []}
    start_time = time.time()
    for ep in range(start_epoch, start_epoch+epochs):
        tl, ta, epoch_time, speed = train_epoch(device, net, train_loader, optimizer=optimizer, lr=lr,
    loss_fn=loss_fn)
        vl, va = validate(device, net, test_loader, loss_fn=loss_fn)
        elapsed_time = time.time() - start_time
        remaining_time = (epochs+start_epoch - (ep + 1)) * epoch_time
        print(f"\nEpoch {ep+1}/{epochs+start_epoch} | {epoch_time:.2f}s | {speed:.2f} samples/sec")
        print(f"Train acc={ta:.3f}, Val acc={va:.3f}, Train loss={tl:.3f}, Val loss={vl:.3f}")
        print(f"Total elapsed: {elapsed_time/60:.2f} min | Estimated remaining: {remaining_time/60:.2f} min
    ")
        res['train_loss'].append(tl)
        res['train_acc'].append(ta)
        res['val_loss'].append(vl)
        res['val_acc'].append(va)
        res['epoch_times'].append(epoch_time)
        res['samples_per_sec'].append(speed)
        model_path = os.path.join(save_path, f"_{device}_epoch_{ep+1}.pth")
        torch.save(net.state_dict(), model_path)
        print(f"Model saved at {model_path}\n")
        if device == "cuda":
            torch.cuda.empty_cache()
    total_time = time.time() - start_time
    print(f"Training complete in {total_time/60:.2f} minutes")
    return res

def train_model(model=models.GoogLeNet(num_classes=len(classes), init_weights=True, aux_logits=False),
                batch_size=16, epochs=5, learning_rate=0.001):
    script_dir = os.path.dirname(os.path.abspath("Preping_Images_for_Pytorch.ipynb"))
    print("Script directory:", script_dir)
    # setup device
    device = torch.accelerator.current_accelerator().type if torch.accelerator.is_available() else "cpu"
    print(f"Using {device} device")
    trainset, valset = train_test_split(img_data, train_size=0.6, random_state=42, shuffle=True, stratify=
    img_data.encoded_targets)
    print(len(trainset))
    print(len(valset))
    trainloader = DataLoader(trainset, batch_size=batch_size)
    valloader = DataLoader(valset, batch_size=batch_size)
    net = model.to(device)
    hist = train(device, net, trainloader, valloader, epochs=epochs, save_path=script_dir, lr=learning_rate
    )
    # Plot training and validation accuracy/loss
    fig1, ax1 = plt.subplots(1, 2, figsize=(15, 5))
    # Accuracy
    ax1[0].plot(hist['train_acc'], label='Training acc')
    ax1[0].plot(hist['val_acc'], label='Validation acc')
    ax1[0].set_title('Accuracy')
    ax1[0].legend()
    # Loss
    ax1[1].plot(hist['train_loss'], label='Training loss')
    ax1[1].plot(hist['val_loss'], label='Validation loss')
    ax1[1].set_title('Loss')
    ax1[1].legend()
    plt.tight_layout()
    plt.show()
    fig, ax = plt.subplots(1, 10, figsize=(15, 4))
    # Ensure weight_tensor has the correct number of filters
    weight_tensor = next(net.parameters()).detach().cpu()  # Move to CPU for visualization
```

```
    # Select first 10 filters (assuming first convolutional layer has at least 10 filters)
    for i in range(10):
        if i >= weight_tensor.shape[0]:  # In case there are fewer than 10 filters
            break
        img = weight_tensor[i, 0].numpy()  # Take the first channel of the filter and convert to NumPy
        ax[i].imshow(img, cmap="gray")  # Display as grayscale image
        ax[i].axis("off")  # Remove axes for better visualization
    plt.tight_layout()
    plt.show()
    dataiter = iter(valloader)
    images, labels = next(dataiter)
    print(summary(net,input_size=(images.shape)))

train_model(epochs=20, batch_size=32) # training default model, GoogLeNet

model = resnet18(weights=ResNet18_Weights.DEFAULT)
model.fc = nn.Linear(model.fc.in_features, len(classes))
train_model(model=model, epochs=20, batch_size=32)

model = efficientnet_b0(weights=EfficientNet_B0_Weights.DEFAULT)
model.classifier[1] = nn.Linear(model.classifier[1].in_features, len(classes))
train_model(model=model, epochs=20, batch_size=32)

model = mobilenet_v2(weights=MobileNet_V2_Weights.DEFAULT)
model.classifier[1] = nn.Linear(model.classifier[1].in_features, len(classes))
train_model(model=model, epochs=20, batch_size=32)
```