coursework_02

March 1, 2022

1 Coursework 2: Fish Classification

Created by Athanasios Vlontzos and Wenjia Bai

In this coursework, you will be exploring the application of convolutional neural networks for image classification tasks. As opposed to standard applications such as object or face classification, we will be dealing with a slightly different domain, fish classification for precision fishing.

In precision fishing, engineers and fishmen collaborate to extract a wide variety of information about the fish, their species and wellbeing etc. using data from satellite images to drones surveying the fisheries. The goal of precision fishing is to provide the marine industry with information to support their decision making processes.

Here your will develop an image classification model that can classify fish species given input images. It consists of two tasks. The first task is to train a model for the following species: - Black Sea Sprat - Gilt-Head Bream - Shrimp - Striped Red Mullet - Trout

The second task is to finetune the last layer of the trained model to adapt to some new species, including: - Hourse Mackerel - Red Mullet - Red Sea Bream - Sea Bass

You will be working using a large-scale fish dataset [1].

[1] O. Ulucan, D. Karakaya and M. Turkan. A large-scale dataset for fish segmentation and classification. Innovations in Intelligent Systems and Applications Conference (ASYU). 2020.

1.1 Step 0: Download data.

Download the Data from here – make sure you access it with your Imperial account.

It is a ~2.5GB file. You can save the images and annotations directories in the same directory as this notebook or somewhere else.

The fish dataset contains 9 species of fishes. There are 1,000 images for each fish species, named as %05d.png in each subdirectory.

1.2 Step 1: Load the data. (15 Points)

- Complete the dataset class with the skeleton below.
- Add any transforms you feel are necessary.

Your class should have at least 3 elements - An `__init__` function that sets up your class and all the necessary parameters. - An `__len__` function that returns the size of your dataset. - An __getitem__ function that given an index within the limits of the size of the dataset returns the associated image and label in tensor form.

You may add more helper functions if you want.

In this section we are following the Pytorch dataset class structure. You can take inspiration from their documentation.

[1]:
```
# Dependencies
import pandas as pd
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
import os
from PIL import Image
import numpy as np
from tqdm import tqdm
import torch
import torch.nn as nn
import torch.nn.functional as F
import matplotlib.pyplot as plt
import glob
import sklearn.metrics as skm
```

[2]:
```
# We will start by building a dataset class using the following 5 species of␣
↪fishes
Multiclass_labels_correspondances = {
    'Black Sea Sprat': 0,
    'Gilt-Head Bream': 1,
    'Shrimp': 2,
    'Striped Red Mullet': 3,
    'Trout': 4
}

# The 5 species will contain 5,000 images in total.
# Let us split the 5,000 images into training (80%) and test (20%) sets
def split_train_test(lendata, percentage=0.8):
    return np.array_split(np.random.permutation(lendata), [round(percentage *␣
↪lendata)])

LENDATA = 5000
np.random.seed(42)
idxs_train, idxs_test = split_train_test(LENDATA, 0.8)

# Implement the dataset class
class FishDataset(Dataset):
    def __init__(self,
                 path_to_images,
                 idxs_train,
                 idxs_test,
                 transform_extra=None,
                 img_size=128,
                 train=True):
        # path_to_images: where you put the fish dataset
        # idxs_train: training set indexes
        # idxs_test: test set indexes
        # transform_extra: extra data transform
        # img_size: resize all images to a standard size
        # train: return training set or test set

        # Load all the images and their labels

        # Resize all images to a standard size and
        tfs = [transforms.Resize((img_size, img_size)), transforms.ToTensor()]

        if transform_extra is not None:
            tfs.insert(0, transform_extra)

        transform = transforms.Compose(tfs)

        images = []
        labels = []

        for class_name, label in Multiclass_labels_correspondances.items():
            files = glob.glob(f"{path_to_images}/{class_name}/*.png")
            print("Class name: " + class_name) # Print class name to track␣
↪progress.
            for file in files:
                image = Image.open(file)
                tensor = transform(image)
                images.append(tensor)
                labels.append(label)
                image.close()

        # Extract the images and labels with the specified file indexes
        indicies = idxs_train if train else idxs_test
        self.images = [images[i] for i in indicies]
        self.labels = [labels[i] for i in indicies]

    def __len__(self):
        # Return the number of samples
        return len(self.images)

    def __getitem__(self, idx):
        # Get an item using its index
        # Return the image and its label
        return self.images[idx], self.labels[idx]
```

1.3 Step 2: Explore the data. (15 Points)

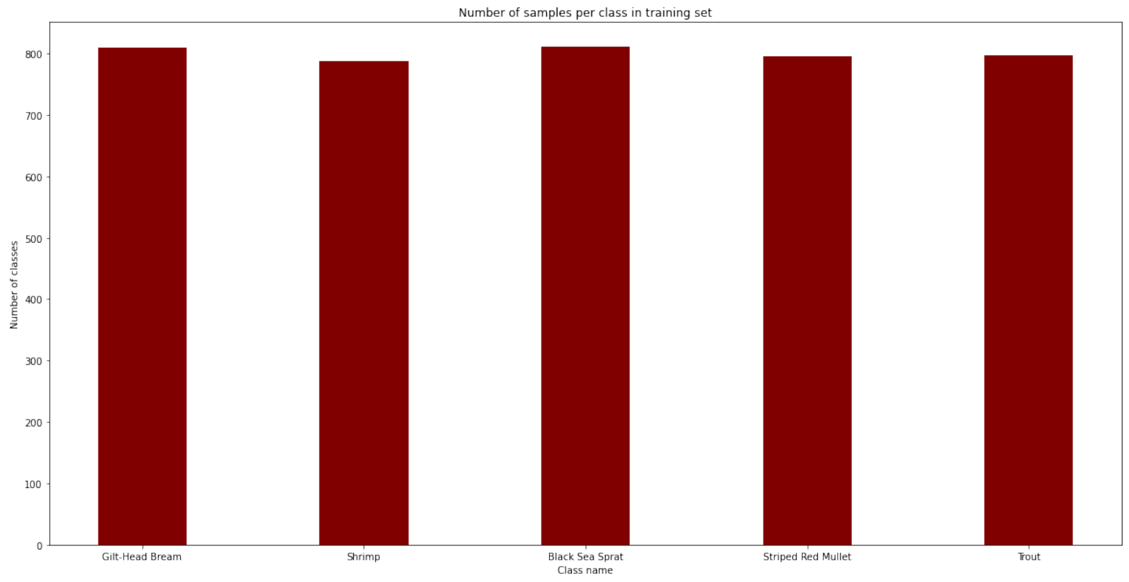
1.3.1 Step 2.1: Data visualisation. (5 points)

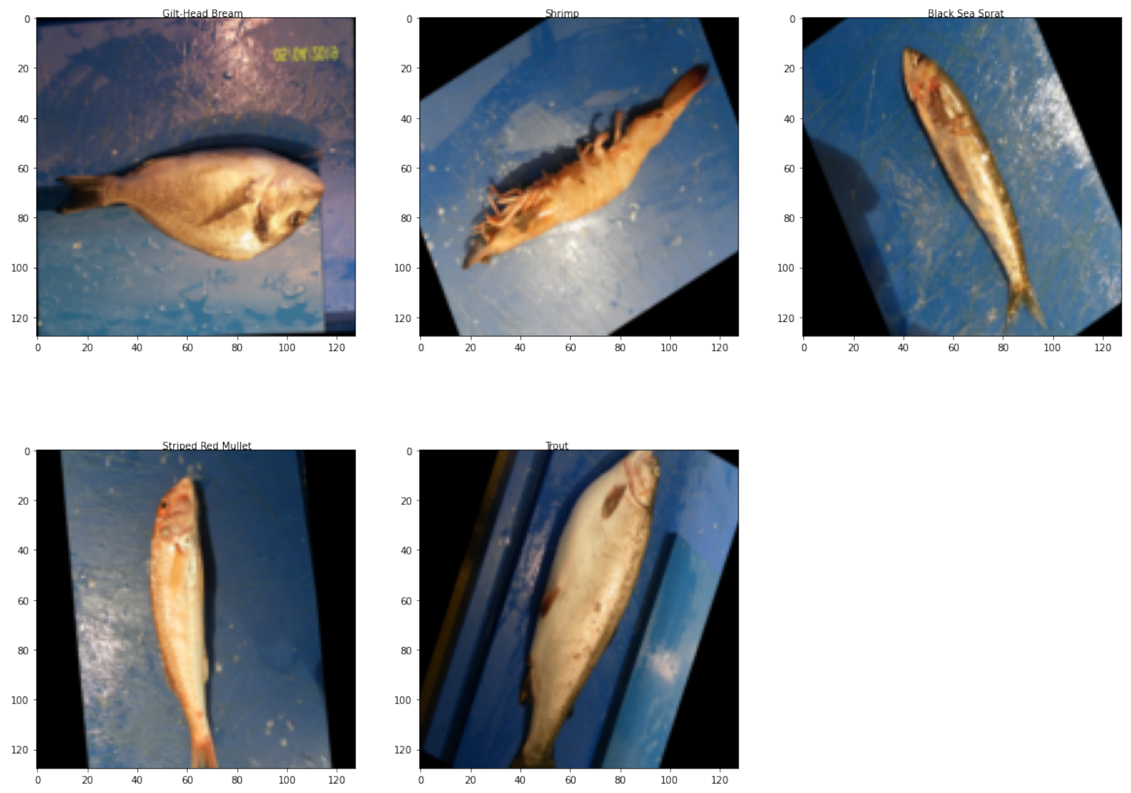
- Plot data distribution, i.e. the number of samples per class.
- Plot 1 sample from each of the five classes in the training set.

[3]:
```
# Training set
img_path = './fish-dataset'
dataset  = FishDataset(img_path, idxs_train, idxs_test, None, img_size=128,␣
 ↪train=True)

inverted_labels_correspondances = dict(zip(
    Multiclass_labels_correspondances.values(),␣
 ↪Multiclass_labels_correspondances.keys()))

# Plot the number of samples per class
sample_map = {}
first_of_each = []
for idx, label in enumerate(dataset.labels):
    if label in sample_map:
        sample_map[label] += 1 # Add to count
    else:
        first_of_each.append((idx, inverted_labels_correspondances[label])) #␣
 ↪New class
        sample_map[label] = 1

print(sorted(sample_map.items())) # Sanity check
classes = list([inverted_labels_correspondances[i] for i in sample_map.keys()])␣
 ↪# list of class names
count = list(sample_map.values()) # list of count of each class

fig = plt.figure(figsize = (20, 10))

plt.bar(classes, count, color ='maroon', width = 0.4)

plt.xlabel("Class name")
plt.ylabel("Number of classes")
plt.title("Number of samples per class in training set")
plt.show()

# Plot 1 sample from each of the five classes in the training set
fig = plt.figure(figsize=(20, 15))
rows = 2
columns = 3
```

```
for i, (idx, label) in enumerate(first_of_each):
    ax = fig.add_subplot(rows, columns, i + 1)
    ax.annotate(label, xy=(50, -0.694)) # Label each image
    plt.imshow(dataset.images[idx].permute(1, 2, 0).numpy())
plt.show()
```

```
Class name: Black Sea Sprat
Class name: Gilt-Head Bream
Class name: Shrimp
Class name: Striped Red Mullet
Class name: Trout
[(0, 811), (1, 809), (2, 787), (3, 796), (4, 797)]
```





1.3.2 Step 2.2: Discussion. (10 points)

- Is the dataset balanced?
- Can you think of 3 ways to make the dataset balanced if it is not?
- Is the dataset already pre-processed? If yes, how?

Yes, we can see that the training dataset is roughly balanced. This is because the original dataset we start off with is perfectly balanced (it contains exactly 1000 of each type of fish), but when we randomly sample our training and test datasets, we may get a slightly higher or lower portion of the different types of fish in each set. However, since the psuedorandom generator used is good, the distribution of different classes in the training and test sets is almost balanced. For example, on one occasion the distribution looked like this: {0: 811, 1: 809, 2: 787, 3: 796, 4: 797}.

If the dataset were not balanced, we could (1) upsample the minority class(es), (2) downsample the majority class(es), or (3) normalise our results (or possibly different combinations of these three). Upsampling the minority class would consist of duplicating images from the classes that we have fewer entries in, perhaps also slightly processing these images (for example, by rotating or adding noise). Downsampling involves removing images from classes where you have more datapoints. When normalising results on the other hand, you train your model using the imbalanced dataset, but then normalise your per class metrics (accuracy, recall, precision, f1, etc.), based on the on the number of datapoints in that class.

Yes, excluding the pre-processing we have done above (making all images 128x128 pixels), we can see the dataset has already been pre-processed. This is obvious if we just look at the images. It is clear that many of the images are simply the same image, but rotated. This shows clearly that some pre-processing was done on this data, perhaps to increase the number of samples or to balance out the dataset. We may also notice that all the images are the same size despite being orientated differently, and that they have a black area surrouding them, which changes with the orientation. This tells us that the images have already been cropped to the same size.

1.4 Step 3: Multiclass classification. (55 points)

In this section we will try to make a multiclass classifier to determine the species of the fish.

1.4.1 Step 3.1: Define the model. (15 points)

Design a neural network which consists of a number of convolutional layers and a few fully connected ones at the end.

The exact architecture is up to you but you do NOT need to create something complicated. For example, you could design a LeNet inspried network.

[4]:
```python
class Net(nn.Module):
    def __init__(self, output_dims = 1):
        super(Net, self).__init__()
        self.cnn_layers = nn.Sequential(
            # Conv layer 1
            nn.Conv2d(3, 30, kernel_size=(5, 5), stride=1, padding=1),
            nn.BatchNorm2d(30),                         # Batch Normalisation
            nn.ReLU(inplace=True),                      # ReLU
            nn.MaxPool2d(kernel_size=(2, 2), stride=2), # MaxPool
            # Conv layer 2
            nn.Conv2d(30, 40, kernel_size=(5, 5), stride=1, padding=1),
            nn.BatchNorm2d(40),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=(2, 2), stride=2),
            # Conv layer 3
            nn.Conv2d(40, 50, kernel_size=(5, 5), stride=1, padding=1),
            nn.BatchNorm2d(50),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=(2, 2), stride=2),
            # Conv layer 4
            nn.Conv2d(50, 60, kernel_size=(5, 5), stride=1, padding=1),
            nn.BatchNorm2d(60),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=(2, 2), stride=2),
        )

        self.linear_layers = nn.Sequential(
            # Fully connected layer 1
            nn.Linear(2160, 800),
            nn.ReLU(),
            # FCL 2
            nn.Linear(800, 200),
            nn.ReLU(),
            # FCL 3
            nn.Linear(200, 80),
            nn.ReLU(),
            # FCL 4 (output layer)
            nn.Linear(80, len(Multiclass_labels_correspondances)),
            # No need to apply Softmax - this is handled by nn.CrossEntropyLoss.
        )

    def forward(self, x):
        # Forward propagation
        x = self.cnn_layers(x)    # Conv layers
        x = x.view(x.size(0), -1) # Flatten
        x = self.linear_layers(x) # Fully connected layers
        return x

# Since most of you use laptops, you may use CPU for training.
# If you have a good GPU, you can set this to 'gpu'.
device = 'cuda'
```

1.4.2 Step 3.2: Define the training parameters. (10 points)

- Loss function
- Optimizer
- Learning Rate
- Number of iterations
- Batch Size
- Other relevant hyperparameters

[5]:
```
# Network - send to GPU
model = Net().to(device)

# Loss function
criterion = nn.CrossEntropyLoss()

# Optimiser and learning rate
# These are the optimal values for learning rate and weight decay based on a␣
 ↪simple
# hyper parameter grid search.
lr = 0.003
wd = 0.001 # Weight decay to implement L2 regularisation
optimiser = torch.optim.Adam(model.parameters(), lr=lr, weight_decay=wd)

# Number of iterations for training
epochs = 20 # Beyond 20 epochs accuracy stagnated or even decreased
```

```
# Training batch size
train_batch_size = 200

# Based on the FishDataset, use the PyTorch DataLoader to load the data during␣
↪model training
train_dataset = FishDataset(img_path, idxs_train, idxs_test, None,␣
↪img_size=128, train=True)
train_dataloader = DataLoader(train_dataset, batch_size=train_batch_size)
test_dataset = FishDataset(img_path, idxs_train, idxs_test, None, img_size=128,␣
↪train=False)
test_dataloader = DataLoader(test_dataset)
```

```
Class name: Black Sea Sprat
Class name: Gilt-Head Bream
Class name: Shrimp
Class name: Striped Red Mullet
Class name: Trout
Class name: Black Sea Sprat
Class name: Gilt-Head Bream
Class name: Shrimp
Class name: Striped Red Mullet
Class name: Trout
```

1.4.3 Step 3.3: Train the model. (15 points)

Complete the training loop.

[6]:
```
# Training
for epoch in tqdm(range(epochs)):
    model.train()
    loss_curve = []

    for imgs, labs in train_dataloader:
        # Get a batch of training data and train the model
        imgs, labs = imgs.to(device), labs.to(device)
        optimiser.zero_grad()
        prediction = model(imgs)
        loss = criterion(prediction, labs)
        loss.retain_grad()
        loss.backward()
        optimiser.step()
        loss_curve.append(loss.item())
    print(f'--- Iteration {epoch + 1}: training loss = {np.array(loss_curve).
↪mean()} ---')
```

```
  5%|
| 1/20 [00:05<01:37,  5.13s/it]
```

```
--- Iteration 1: training loss = 1.2108610779047013 ---

 10%|
| 2/20 [00:10<01:31,  5.06s/it]

--- Iteration 2: training loss = 0.4649840034544468 ---

 15%|
| 3/20 [00:15<01:25,  5.02s/it]

--- Iteration 3: training loss = 0.18856277763843537 ---

 20%|
| 4/20 [00:20<01:21,  5.09s/it]

--- Iteration 4: training loss = 0.09349428359419107 ---

 25%|
| 5/20 [00:25<01:17,  5.14s/it]

--- Iteration 5: training loss = 0.06929997550323605 ---

 30%|
| 6/20 [00:30<01:12,  5.17s/it]

--- Iteration 6: training loss = 0.04331161612644792 ---

 35%|
| 7/20 [00:35<01:05,  5.08s/it]

--- Iteration 7: training loss = 0.10006201560609043 ---

 40%|
| 8/20 [00:40<00:59,  4.96s/it]

--- Iteration 8: training loss = 0.04051881120540202 ---

 45%|
| 9/20 [00:45<00:53,  4.88s/it]

--- Iteration 9: training loss = 0.012873238429892808 ---

 50%|
| 10/20 [00:49<00:48,  4.84s/it]

--- Iteration 10: training loss = 0.012637658277526498 ---

 55%|
| 11/20 [00:54<00:43,  4.87s/it]

--- Iteration 11: training loss = 0.013419696263736115 ---

 60%|
| 12/20 [00:59<00:39,  4.91s/it]

--- Iteration 12: training loss = 0.015069153183139861 ---

 65%|
| 13/20 [01:04<00:33,  4.86s/it]
```

```
--- Iteration 13: training loss = 0.022965467078029177 ---

 70%|
                                | 14/20 [01:09<00:28,  4.80s/it]

--- Iteration 14: training loss = 0.3309222874697298 ---

 75%|
                              | 15/20 [01:13<00:23,  4.76s/it]

--- Iteration 15: training loss = 0.04578482094220817 ---

 80%|
                           | 16/20 [01:18<00:18,  4.73s/it]

--- Iteration 16: training loss = 0.01987174842506647 ---

 85%|
                         | 17/20 [01:23<00:14,  4.73s/it]

--- Iteration 17: training loss = 0.019269794062711298 ---

 90%|
                       | 18/20 [01:27<00:09,  4.72s/it]

--- Iteration 18: training loss = 0.008839921175967902 ---

 95%|
                    | 19/20 [01:32<00:04,  4.71s/it]

--- Iteration 19: training loss = 0.0031805290636839345 ---

100%|
                  | 20/20 [01:37<00:00,  4.86s/it]

--- Iteration 20: training loss = 0.0013939979748101906 ---
```

1.4.4 Step 3.4: Deploy the trained model onto the test set. (10 points)

[7]:
```python
# Deploy the model
with torch.no_grad():
    model.eval()
    predicted_labels, true_labels = zip(*[(model.forward(x.to(device)), y.
 ↪to(device).item()) for x, y in test_dataloader])
```

1.4.5 Step 3.5: Evaluate the performance of the model and visualize the confusion matrix. (5 points)

You can use sklearns related function.

[8]:
```python
def acc_and_conf(predicted_labels, true_labels):
    correct = 0
    y_pred = []
    y_gold = []
    for pred, gold in zip(predicted_labels, true_labels):
        y_pred.append(pred.argmax().item())
        y_gold.append(gold)
        if pred.argmax() == gold:
            correct += 1

    return (correct / len(true_labels), skm.multilabel_confusion_matrix(y_gold,␣
↪y_pred),
            skm.classification_report(y_gold, y_pred))

acc, conf, report = acc_and_conf(predicted_labels, true_labels)
print(f"Accuracy: {acc}")
print(conf)
print(report)
# Cleary this is slightly overfitted:
```

```
Accuracy: 1.0
[[[811   0]
  [  0 189]]

 [[809   0]
  [  0 191]]

 [[787   0]
  [  0 213]]

 [[796   0]
  [  0 204]]

 [[797   0]
  [  0 203]]]
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       189
           1       1.00      1.00      1.00       191
           2       1.00      1.00      1.00       213
           3       1.00      1.00      1.00       204
           4       1.00      1.00      1.00       203

    accuracy                           1.00      1000
   macro avg       1.00      1.00      1.00      1000
weighted avg       1.00      1.00      1.00      1000
```

1.5 Step 4: Finetune your classifier. (15 points)

In the previous section, you have built a pretty good classifier for certain species of fish. Now we are going to use this trained classifier and adapt it to classify a new set of species:

```
'Hourse Mackerel
'Red Mullet',
'Red Sea Bream'
'Sea Bass'
```

1.5.1 Step 4.1: Set up the data for new species. (2 points)

Overwrite the labels correspondances so they only incude the new classes and regenerate the datasets and dataloaders.

[9]:
```python
Multiclass_labels_correspondances = {
    'Hourse Mackerel': 0,
    'Red Mullet': 1,
    'Red Sea Bream': 2,
    'Sea Bass': 3}

LENDATA = 3999 # Removed corrupted image
idxs_train,idxs_test = split_train_test(LENDATA, 0.8)

# Dataloaders
train_dataset = FishDataset(img_path, idxs_train, idxs_test, None,␣
 ↪img_size=128, train=True)
train_dataloader = DataLoader(train_dataset, batch_size=train_batch_size)
test_dataset = FishDataset(img_path, idxs_train, idxs_test, None, img_size=128,␣
 ↪train=False)
test_dataloader = DataLoader(test_dataset)
```

```
Class name: Hourse Mackerel
Class name: Red Mullet
Class name: Red Sea Bream
Class name: Sea Bass
Class name: Hourse Mackerel
Class name: Red Mullet
Class name: Red Sea Bream
Class name: Sea Bass
```

1.5.2 Step 4.2: Freeze the weights of all previous layers of the network except the last layer. (5 points)

You can freeze them by setting the gradient requirements to `False`.

[12]:
```python
def freeze_till_last(model):
    for param in model.parameters():
        param.requires_grad = False
```

```python
freeze_till_last(model)
# Modify the last layer. This layer is not freezed.
model.linear_layers = nn.Sequential(
            # Fully connected layer 1
            nn.Linear(2160, 800),
            nn.ReLU(),
            # FCL 2
            nn.Linear(800, 200),
            nn.ReLU(),
            # FCL 3
            nn.Linear(200, 80),
            nn.ReLU(),
            # FCL 4 (output layer)
            nn.Linear(80, 4)).to(device)

for name, p in model.named_parameters():
    print(name, p.requires_grad)

# Loss function
criterion = nn.CrossEntropyLoss()

# Optimiser and learning rate
lr = 0.01
wd = 0.001 # Weight decay to implement L2 regularisation
optimiser = torch.optim.Adam(model.parameters(), lr=lr, weight_decay=wd)

# Number of iterations for training
epochs = 20

# Training batch size
train_batch_size = 200
```

```
cnn_layers.0.weight False
cnn_layers.0.bias False
cnn_layers.1.weight False
cnn_layers.1.bias False
cnn_layers.4.weight False
cnn_layers.4.bias False
cnn_layers.5.weight False
cnn_layers.5.bias False
cnn_layers.8.weight False
cnn_layers.8.bias False
cnn_layers.9.weight False
cnn_layers.9.bias False
cnn_layers.12.weight False
cnn_layers.12.bias False
cnn_layers.13.weight False
cnn_layers.13.bias False
linear_layers.0.weight True
linear_layers.0.bias True
linear_layers.2.weight True
linear_layers.2.bias True
linear_layers.4.weight True
linear_layers.4.bias True
linear_layers.6.weight True
linear_layers.6.bias True
```

1.5.3 Step 4.3: Train and test your finetuned model. (5 points)

[11]:
```
# Finetune the model
for epoch in tqdm(range(epochs)):
    model.train()
    loss_curve = []

    for imgs, labs in train_dataloader:
        # Get a batch of training data and train the model
        imgs, labs = imgs.to(device), labs.to(device)
        optimiser.zero_grad()
        prediction = model(imgs)
        loss = criterion(prediction, labs)
        loss.retain_grad()
        loss.backward()
        optimiser.step()
        loss_curve.append(loss.item())
    print(f'--- Iteration {epoch + 1}: training loss = {np.array(loss_curve).
 ↪mean()} ---')

# Deploy the model on the test set
with torch.no_grad():
    model.eval()
    predicted_labels, true_labels = zip(*[(model.forward(x.to(device)), y.
 ↪to(device).item()) for x, y in test_dataloader])

# Evaluate the performance
acc, conf, report = acc_and_conf(predicted_labels, true_labels)
print(f"Accuracy: {acc}")
print(conf)
print(report)
```

```
  5%|
| 1/20 [00:03<01:10,  3.72s/it]

--- Iteration 1: training loss = 2.2389076575636864 ---

 10%|
| 2/20 [00:07<01:06,  3.72s/it]

--- Iteration 2: training loss = 1.0184555687010288 ---

 15%|
| 3/20 [00:11<01:03,  3.72s/it]

--- Iteration 3: training loss = 0.5933783017098904 ---

 20%|
| 4/20 [00:14<00:59,  3.72s/it]

--- Iteration 4: training loss = 0.28016181755810976 ---

 25%|
| 5/20 [00:18<00:55,  3.73s/it]

--- Iteration 5: training loss = 0.13371510850265622 ---

 30%|
| 6/20 [00:22<00:52,  3.74s/it]

--- Iteration 6: training loss = 0.0697159762494266 ---

 35%|
| 7/20 [00:26<00:48,  3.74s/it]

--- Iteration 7: training loss = 0.03880029375432059 ---

 40%|
| 8/20 [00:29<00:45,  3.75s/it]

--- Iteration 8: training loss = 0.06380603602156043 ---

 45%|
| 9/20 [00:33<00:41,  3.75s/it]

--- Iteration 9: training loss = 0.17864900454878807 ---

 50%|
| 10/20 [00:37<00:37,  3.74s/it]

--- Iteration 10: training loss = 0.0896283492911607 ---

 55%|
| 11/20 [00:41<00:33,  3.74s/it]

--- Iteration 11: training loss = 0.03632982727140188 ---

 60%|
| 12/20 [00:44<00:30,  3.77s/it]

--- Iteration 12: training loss = 0.02487476178794168 ---

 65%|
| 13/20 [00:48<00:26,  3.77s/it]

--- Iteration 13: training loss = 0.026129662001039833 ---
```

```
 70%|
                                | 14/20 [00:52<00:22,  3.77s/it]

--- Iteration 14: training loss = 0.03150241306866519 ---

 75%|
                              | 15/20 [00:56<00:18,  3.76s/it]

--- Iteration 15: training loss = 0.031803165446035564 ---

 80%|
                            | 16/20 [00:59<00:14,  3.75s/it]

--- Iteration 16: training loss = 0.013869807065930218 ---

 85%|
                          | 17/20 [01:03<00:11,  3.75s/it]

--- Iteration 17: training loss = 0.0033477010620117653 ---

 90%|
                        | 18/20 [01:07<00:07,  3.77s/it]

--- Iteration 18: training loss = 0.015797618751093978 ---

 95%|
                      | 19/20 [01:11<00:03,  3.75s/it]

--- Iteration 19: training loss = 0.04214506002608687 ---

100%|
                    | 20/20 [01:14<00:00,  3.75s/it]

--- Iteration 20: training loss = 0.09482445986941457 ---


Accuracy: 0.84375
[[[533  44]
  [ 19 204]]

 [[600   3]
  [  8 189]]

 [[608   1]
  [ 76 115]]

 [[534  77]
  [ 22 167]]]
              precision    recall  f1-score   support

           0       0.82      0.91      0.87       223
           1       0.98      0.96      0.97       197
           2       0.99      0.60      0.75       191
           3       0.68      0.88      0.77       189
```

```
    accuracy                           0.84       800
   macro avg       0.87      0.84      0.84       800
weighted avg       0.87      0.84      0.84       800
```

1.5.4 Step 4.4: Did finetuning work? Why did we freeze the first few layers? (3 points)

Whether finetuning works or not depends on whether we unfreeze only the output layer, or unfreeze all of the fully connected layers.

When I tried freezing all layers except the output layer, I got the following accuracy and confusion matrix:

Accuracy: 0.23625

[[[577 0]

[223 0]]

[[603 0]

[197 0]]

[[609 0]

[191 0]]

[[0 611]

[0 189]]]

As we can see, the accuracy is hardly better than random. This is due to the fact that we are classifying totally different classes of fish, and neither our conv layers nor our fully-connected layers have been trained for this. Therefore, since we can only train the final layer, we are essentially only training a linear function - in order to be able to fit a non-linear function, we need to be able to train multiple layers separated by a non-linear activation function. Therefore, this model performs poorly, as expected. This makes sense since the majority of our fully-connected layers are trained for a different dataset consisting of different classes.

However, if we instead unfreeze all of our fully-connected layers, so that only the convolutional layers are frozen, finetuning works very well. This is because the conv layers are responsible for generating a feature map from an image - it is reasonable to assume that the conv layers trained on the first dataset should also work on the new one, since both datasets consist of similar images of fish (even if they are different breeds).

Therefore, the frozen conv layers produce a feature map that is still highly descriptive of the new images, and then the unfrozen linear layers can be trained for the new dataset. We can see that this produces a good result:

Accuracy: 0.84375

[[[533 44]

[19 204]]

```
[[600 3]
[ 8 189]]
[[608 1]
[ 76 115]]
[[534 77]
[ 22 167]]]
```

When comparing this with the error from training the whole (slightly overfitted) model, we realise the difference is relatively small:

```
Accuracy: 1.0
[[[811 0]
[ 0 189]]
[[809 0]
[ 0 191]]
[[787 0]
[ 0 213]]
[[796 0]
[ 0 204]]
[[797 0]
[ 0 203]]]
```

The advantage of this approach is that it is much faster to train. Training the full model took roughly 5 seconds per iteration, or 100 seconds in total, whereas training only the last layers took roughly 3.7 seconds per iteration, or 74 seconds in total. The larger the dataset, and the more complex the model architecture, the greater this difference will be. Thus in some situations it may be highly advantages to take a pre-trained model and finetune it for your dataset rather than train an entirely new model.