coursework_02

February 18, 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.

[ ]:
```
# 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 copy
```

[ ]:
```
# 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):
    #### ADD YOUR CODE HERE ####
    indices = np.arange(lendata)
    idxs_train = np.random.choice(indices, int(lendata*percentage),␣
 ↪replace=False)
    idxs_test = np.setdiff1d(indices, idxs_train)

    return idxs_train, idxs_test

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

        all_images = []
        all_labels = []
        for label in Multiclass_labels_correspondances:
            for i in glob.glob(os.path.join(path_to_images,label,'*.png')):
                try:
                    all_images.append(copy.deepcopy(Image.open(i)))
                    all_labels.append(Multiclass_labels_correspondances[label])
                except:
                    print('Error opening image:',i)

        print('Number of images:', len(all_images))

        # Resize all images to a standard size

        self.transform = transforms.Compose([
            transform_extra,
            transforms.ToTensor()
        ]) if transform_extra is not None else transforms.ToTensor()

        for i in range(len(all_images)):
            all_images[i] = all_images[i].resize((img_size, img_size))

        # Extract the images and labels with the specified file indexes

        self.images = [(all_images[i], all_labels[i]) for i in idxs_train] if␣
↪train else [(all_images[i], all_labels[i]) for i in idxs_test]

    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
        sample, label = self.images[idx]
        sample = self.transform(sample)
        return sample, label
```

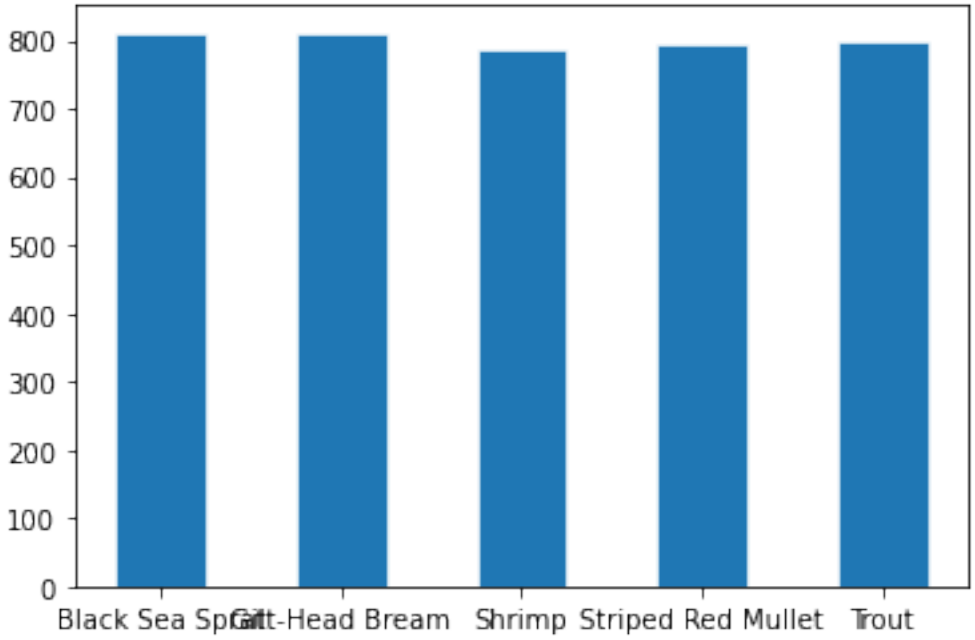
1.3 Step 2: Explore the data. (15 Points)

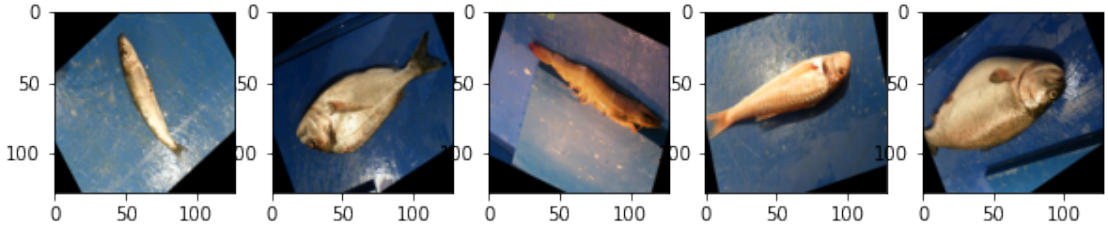
1.3.1 Step 2.1: Data visualisation. (5 points)

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

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

# Plot the number of samples per class

labels = []
count = []

for label in Multiclass_labels_correspondances:
    labels.append(label)
    count.append(len([i for i in dataset.images if i[1] ==␣
↪Multiclass_labels_correspondances[label]]))

_, ax = plt.subplots()
ax.bar(labels, count, width=0.5, edgecolor="white", linewidth=0.7)
# ax.set(xlim=(0, 5), xticks=np.arange(1, 5))

plt.show()

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

# Plot 1 sample from each of the five classes in the training set

for label in Multiclass_labels_correspondances:
    image = list(filter(lambda x: x[1] ==␣
↪Multiclass_labels_correspondances[label], dataset.images))[0][0]
    fig.add_subplot(1, 5, Multiclass_labels_correspondances[label] + 1)
    plt.imshow(image)

plt.show()
```

```
Number of images: 5000
```





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?

The dataset is fairly balanced, with around 800 training samples per class. If the dataset is not balanced, we have at least three options to choose:

1. Create duplicate samples by rotating, translating, or scaling the images.
2. Reduce the number of samples for classes that have more samples.
3. Record/obtain more samples for classes that have less samples.

The dataset is already pre-processed, and the images are already resized to 128x128 pixels.

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 insprired network.

[ ]:
```
class Net(nn.Module):
    def __init__(self, output_dims = 5):
        super(Net, self).__init__()
        # Define the network
        self.layer1 = nn.Sequential(
            nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=2),
            nn.BatchNorm2d(16),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))
        self.layer2 = nn.Sequential(
            nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=2),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))
        self.fc1 = nn.Linear(33*33*32, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, output_dims)


    def forward(self, x):
        # Forward propagation
        x = self.layer1(x)
        x = self.layer2(x)
        x = x.reshape(x.size(0), -1)
        x = self.fc1(x)
        x = self.fc2(x)
        x = self.fc3(x)
        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 = 'cpu'
```

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

[ ]:
```
# Network
model = Net().to(device)

# Loss function
criterion = nn.CrossEntropyLoss()

# Optimiser and learning rate
lr = 0.001
optimizer = torch.optim.Adam(model.parameters(), lr=lr)

# Number of iterations for training
epochs = 100

# Training batch size
train_batch_size = 32

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

```
Number of images: 5000
Number of images: 5000
```

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

Complete the training loop.

[ ]:
```
# Network
model = Net().to(device)

# Loss function
criterion = nn.CrossEntropyLoss().to(device)

# Optimiser and learning rate
lr = 0.001
optimizer = torch.optim.Adam(model.parameters(), lr=lr)
```

```python
# Number of iterations for training
epochs = 20

# Training batch size
train_batch_size = 64

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 = imgs.to(device)
        labs = labs.to(device)
        optimizer.zero_grad()
        outputs = model(imgs)
        loss = criterion(outputs, labs)
        loss.backward()
        optimizer.step()
        loss_curve.append(loss.item())

    print('--- Iteration {0}: training loss = {1:.4f} ---'.format(epoch + 1, np.
 ↪array(loss_curve).mean()))
```

```
  5%|          | 1/20 [00:34<11:00, 34.75s/it]
--- Iteration 1: training loss = 1.5529 ---
 10%|          | 2/20 [01:09<10:24, 34.70s/it]
--- Iteration 2: training loss = 0.2533 ---
 15%|          | 3/20 [01:44<09:49, 34.69s/it]
--- Iteration 3: training loss = 0.2247 ---
 20%|          | 4/20 [02:18<09:14, 34.64s/it]
--- Iteration 4: training loss = 0.1656 ---
 25%|         | 5/20 [02:53<08:38, 34.59s/it]
--- Iteration 5: training loss = 0.1284 ---
 30%|         | 6/20 [03:27<08:03, 34.56s/it]
--- Iteration 6: training loss = 0.0629 ---
 35%|        | 7/20 [04:02<07:28, 34.53s/it]
--- Iteration 7: training loss = 0.0801 ---
 40%|        | 8/20 [04:36<06:54, 34.50s/it]
```

```
--- Iteration 8: training loss = 0.0757 ---
 45%|        | 9/20 [05:11<06:19, 34.52s/it]
--- Iteration 9: training loss = 0.0683 ---
 50%|        | 10/20 [05:45<05:45, 34.55s/it]
--- Iteration 10: training loss = 0.0425 ---
 55%|        | 11/20 [06:20<05:11, 34.58s/it]
--- Iteration 11: training loss = 0.0395 ---
 60%|        | 12/20 [06:55<04:36, 34.60s/it]
--- Iteration 12: training loss = 0.0293 ---
 65%|       | 13/20 [07:29<04:02, 34.61s/it]
--- Iteration 13: training loss = 0.0357 ---
 70%|       | 14/20 [08:04<03:27, 34.63s/it]
--- Iteration 14: training loss = 0.0480 ---
 75%|      | 15/20 [08:39<02:53, 34.65s/it]
--- Iteration 15: training loss = 0.0751 ---
 80%|      | 16/20 [09:13<02:18, 34.65s/it]
--- Iteration 16: training loss = 0.0759 ---
 85%|      | 17/20 [09:48<01:43, 34.66s/it]
--- Iteration 17: training loss = 0.0536 ---
 90%|      | 18/20 [10:23<01:09, 34.65s/it]
--- Iteration 18: training loss = 0.0479 ---
 95%|     | 19/20 [10:57<00:34, 34.75s/it]
--- Iteration 19: training loss = 0.0450 ---
100%|     | 20/20 [11:32<00:00, 34.65s/it]
--- Iteration 20: training loss = 0.0267 ---
```

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

[ ]:
```python
# Deploy the model
model.eval()
ground_truth = []
predictions = []
with torch.no_grad():
    for imgs, labs in test_dataloader:
```

```
        imgs = imgs.to(device)
        labs = labs.to(device)
        outputs = model(imgs)
        _, predicted = torch.max(outputs.data, 1)
        ground_truth.extend(labs.cpu().numpy().tolist())
        predictions.extend(predicted.cpu().numpy().tolist())
```

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.

[ ]:
```
# Evaluate the performance of the model and visualize the confusion matrix
def confusion_matrix(ground_truth, predictions, dimension=5):
    cm = np.zeros((dimension, dimension))
    for i in range(len(ground_truth)):
        cm[ground_truth[i]][predictions[i]] += 1
    return cm

print('--- Performance of the model on the test set ---')
print(confusion_matrix(ground_truth, predictions))
# Calculate the accuracy, precision, recall, and F1 score of the model
accuracy = np.trace(confusion_matrix(ground_truth, predictions)) / np.
 ↪sum(confusion_matrix(ground_truth, predictions))
precision = np.zeros((5,))
recall = np.zeros((5,))
f1_score = np.zeros((5,))
for i in range(5):
    precision[i] = confusion_matrix(ground_truth, predictions)[i][i] / np.
 ↪sum(confusion_matrix(ground_truth, predictions)[i])
    recall[i] = confusion_matrix(ground_truth, predictions)[i][i] / np.
 ↪sum(confusion_matrix(ground_truth, predictions)[:, i])
    f1_score[i] = 2 * precision[i] * recall[i] / (precision[i] + recall[i])

# print accuracy, precision, recall, and F1 score for each class
print('Accuracy: {0:.4f}'.format(accuracy))
# print precision, recall, and F1 score for each class
for i in range(5):
    print('Class {0}: precision = {1:.4f}, recall = {2:.4f}, F1 score = {3:.
 ↪4f}'.format(i, precision[i], recall[i], f1_score[i]))
```

```
--- Performance of the model on the test set ---
[[154.  13.  14.  21.   0.]
 [ 10. 200.   2.   9.   0.]
 [  5.   2. 150.  18.   0.]
 [ 14.   8.   2. 178.   0.]
 [  0.   0.   0.   0.   0.]]
Accuracy: 0.8525
```

```
Class 0: precision = 0.7624, recall = 0.8415, F1 score = 0.8000
Class 1: precision = 0.9050, recall = 0.8969, F1 score = 0.9009
Class 2: precision = 0.8571, recall = 0.8929, F1 score = 0.8746
Class 3: precision = 0.8812, recall = 0.7876, F1 score = 0.8318
Class 4: precision = nan, recall = nan, F1 score = nan
```

```
/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-
packages/ipykernel_launcher.py:16: RuntimeWarning: invalid value encountered in
double_scalars
  app.launch_new_instance()
/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-
packages/ipykernel_launcher.py:17: RuntimeWarning: invalid value encountered in
double_scalars
```

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 include the new classes and regenerate the datasets and dataloaders.

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

[ ]:
```
LENDATA = 4000
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,␣
 ↪shuffle=True)
test_dataset = FishDataset(img_path, idxs_train, idxs_test, None, img_size=128,␣
 ↪train=False)
test_dataloader = DataLoader(test_dataset, batch_size=train_batch_size,␣
 ↪shuffle=False)
```

```
Number of images: 4000
Number of images: 4000
```

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`.

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

freeze_till_last(model)
# Modify the last layer. This layer is not freezed.
model.fc3.weight.requires_grad = True
model.fc3.bias.requires_grad = True
model.fc2.weight.requires_grad = True
model.fc2.bias.requires_grad = True
model.fc1.weight.requires_grad = True
model.fc1.bias.requires_grad = True

# Loss function
criterion = nn.CrossEntropyLoss()

# Optimiser and learning rate
lr = 0.001
optimizer = torch.optim.Adam(model.parameters(), lr=lr)

# Number of iterations for training
epochs = 20

# Training batch size
train_batch_size = 32
```

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

[ ]:
```
# 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 = imgs.to(device)
        labs = labs.to(device)
        optimizer.zero_grad()
        outputs = model(imgs)
        loss = criterion(outputs, labs)
        loss.backward()
        optimizer.step()
        loss_curve.append(loss.item())
```

```
    print('--- Iteration {0}: training loss = {1:.4f} ---'.format(epoch + 1, np.
 ↪array(loss_curve).mean()))

# Deploy the model on the test set
model.eval()
ground_truth = []
predictions = []
with torch.no_grad():
    for imgs, labs in test_dataloader:
        imgs = imgs.to(device)
        labs = labs.to(device)
        outputs = model(imgs)
        _, predicted = torch.max(outputs.data, 1)
        ground_truth.extend(labs.cpu().numpy().tolist())
        predictions.extend(predicted.cpu().numpy().tolist())
```

```
  5%|          | 1/20 [00:09<03:00,  9.48s/it]
--- Iteration 1: training loss = 3.8600 ---
 10%|          | 2/20 [00:18<02:49,  9.44s/it]
--- Iteration 2: training loss = 0.2005 ---
 15%|          | 3/20 [00:28<02:40,  9.42s/it]
--- Iteration 3: training loss = 0.0284 ---
 20%|          | 4/20 [00:37<02:31,  9.44s/it]
--- Iteration 4: training loss = 0.0155 ---
 25%|         | 5/20 [00:47<02:21,  9.45s/it]
--- Iteration 5: training loss = 0.0028 ---
 30%|         | 6/20 [00:56<02:12,  9.46s/it]
--- Iteration 6: training loss = 0.0855 ---
 35%|        | 7/20 [01:06<02:02,  9.44s/it]
--- Iteration 7: training loss = 0.0485 ---
 40%|        | 8/20 [01:15<01:53,  9.44s/it]
--- Iteration 8: training loss = 0.0068 ---
 45%|        | 9/20 [01:24<01:43,  9.44s/it]
--- Iteration 9: training loss = 0.0022 ---
 50%|        | 10/20 [01:34<01:34,  9.43s/it]
--- Iteration 10: training loss = 0.0005 ---
 55%|       | 11/20 [01:43<01:24,  9.43s/it]
```

```
--- Iteration 11: training loss = 0.0001 ---
 60%|       | 12/20 [01:53<01:15,  9.42s/it]
--- Iteration 12: training loss = 0.0001 ---
 65%|       | 13/20 [02:02<01:05,  9.42s/it]
--- Iteration 13: training loss = 0.0000 ---
 70%|       | 14/20 [02:12<00:56,  9.42s/it]
--- Iteration 14: training loss = 0.0000 ---
 75%|      | 15/20 [02:21<00:47,  9.41s/it]
--- Iteration 15: training loss = 0.0000 ---
 80%|      | 16/20 [02:30<00:37,  9.41s/it]
--- Iteration 16: training loss = 0.0000 ---
 85%|      | 17/20 [02:40<00:28,  9.41s/it]
--- Iteration 17: training loss = 0.0000 ---
 90%|      | 18/20 [02:49<00:18,  9.41s/it]
--- Iteration 18: training loss = 0.0000 ---
 95%|     | 19/20 [02:59<00:09,  9.41s/it]
--- Iteration 19: training loss = 0.0000 ---
100%|     | 20/20 [03:08<00:00,  9.42s/it]
--- Iteration 20: training loss = 0.0000 ---
```

[ ]:
```python
# Evaluate the performance
print('--- Performance of the model on the test set ---')
print(confusion_matrix(ground_truth, predictions, 4))
# Calculate the accuracy, precision, recall, and F1 score of the model
accuracy = np.trace(confusion_matrix(ground_truth, predictions, 4)) / np.
 ↪sum(confusion_matrix(ground_truth, predictions, 4))
precision = np.zeros((4,))
recall = np.zeros((4,))
f1_score = np.zeros((4,))
for i in range(4):
    precision[i] = confusion_matrix(ground_truth, predictions, 4)[i][i] / np.
 ↪sum(confusion_matrix(ground_truth, predictions, 4)[i])
    recall[i] = confusion_matrix(ground_truth, predictions, 4)[i][i] / np.
 ↪sum(confusion_matrix(ground_truth, predictions, 4)[:, i])
    f1_score[i] = 2 * precision[i] * recall[i] / (precision[i] + recall[i])

# print accuracy, precision, recall, and F1 score for each class
```

```
print('Accuracy: {0:.4f}'.format(accuracy))
```

```
--- Performance of the model on the test set ---
[[202.   0.   0.   0.]
 [  0. 221.   0.   0.]
 [  0.   0. 175.   0.]
 [  0.   0.   0. 202.]]
Accuracy: 1.0000
```

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

The finetuning has worked. The overall accuracy of the model is around 85.25%. We freezed the first few layers because the second task we are performing (e.g. classifying 4 new species of fish) is very similar to the first task (e.g. classifying 5 species of fish). The convolutional layers can be reused for similar tasks since their purpose is mainly to extract the features using kernel windows, not to really contribute to the classification. The only layer(s) need to be trained/learned is the fully connected layers that actually use the features extracted by the convolutional layers to classify the images. Furthermore, freezing the convolutional layers can speed up training time as we only need to calculate the loss and adjust the weights of the fully connected layers.