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.

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.

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.

In [237…

```
# Dependencies
import random
from itertools import groupby

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
```

In [238…

```
# 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):
    split_idx = int(lendata * percentage)
    idxs_list = list(range(lendata))
    random.shuffle(idxs_list)
    idxs_train = set(idxs_list[:split_idx])
    idxs_test = set(idxs_list[split_idx:])
    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

        indexes = idxs_train if train else idxs_test
        paths = []

        samples = []

        for class_name in Multiclass_labels_correspondances:
            label = Multiclass_labels_correspondances[class_name]
            for path in glob.glob(path_to_images + '/' + class_name + '/*'):
                paths.append((path, label))

        paths = np.array(paths)

        for i in range(len(paths)):
            if i in indexes:
                image = Image.open(paths[i][0]).resize((img_size, img_size))
                tensor = transforms.ToTensor()(image)
                if transform_extra:
                    tensor = transform_extra(tensor)
                image.close()
                samples.append((tensor, int(paths[i][1])))

        self.samples = samples


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


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

Step 2: Explore the data. (15 Points)

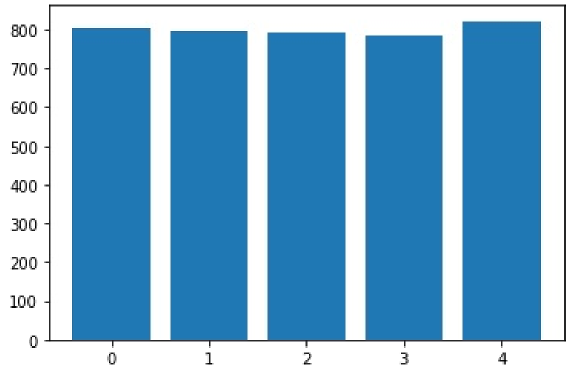
Step 2.1: Data visualisation. (5 points)

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

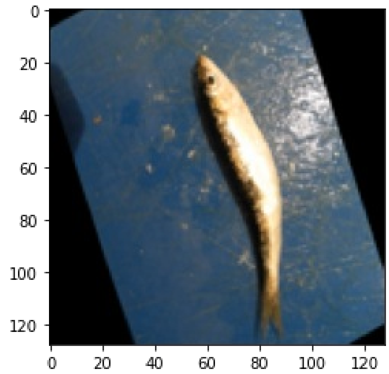
In [239…

```
# Training set
```

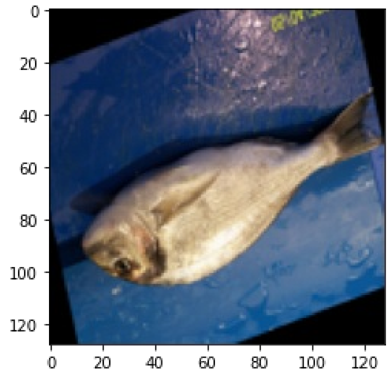
```python
img_path = './dataset'
dataset = FishDataset(img_path, idxs_train, idxs_test, None, img_size=128, train=True)

# Plot the number of samples per class
groups = {k: list(v) for k, v in groupby(dataset, key=lambda x: x[1])}
counts = dict(map(lambda e: (e[0], len(e[1])), groups.items()))

plt.bar(range(len(counts)), list(counts.values()), align='center')
plt.show()

# Plot 1 sample from each of the five classes in the training set
for label, group in groups.items():
    print("Sample from group {0}: ".format(label))
    plt.imshow(transforms.ToPILImage()(group[0][0]))
    plt.show()
```



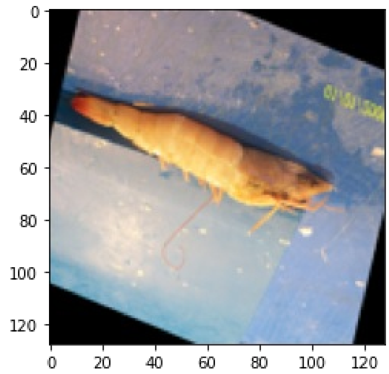
Sample from group 0:



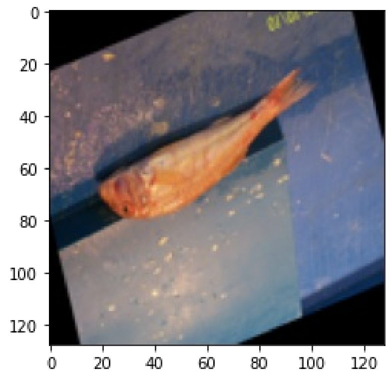
Sample from group 1:



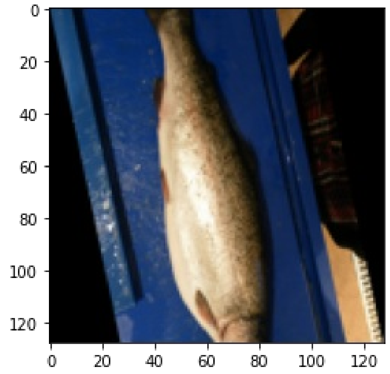
Sample from group 2:



Sample from group 3:



Sample from group 4:



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?

ADD YOUR RESPONSE HERE

1. Looking at the plot, we can say that the dataset is pretty balanced, as all classes have roughly the same number of samples.
2. To balance the dataset, we need to either increase the number of entries in the under-represented categories or decrease it in the others (usually the former). To this end, we have a few options: i. Upsampling: Introduce duplicates of entries in under-represented categories, so the model will train more on them. Similarily, we may use downsampling to reduce the presence of the majority classes. ii. Data augmentation: Introduce modified copies of existing samples. This is different from upsampling as we are creating new data for our model to train on instead of repeating old data. iii. Re-shuffling the data: This is a naive and uncertain solution, but, with a bit of luck, may result in a more balanced train - test split.
3. The dataset is pre-processed in the **init** step, as we bring all images to the same size (in our case, 128x128). Furthermore, it appears that the dataset was subjected to data augmentation, as there are multiple samples of the same rotated image.

Step 3: Multiclass classification. (55 points)

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

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.

In [240_

```
class Net(nn.Module):
    def __init__(self, output_dims=1):
        super(Net, self).__init__()
        self.pool = nn.AvgPool2d(2, stride=2)
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=6, kernel_size=(5, 5))
        self.conv2 = nn.Conv2d(in_channels=6, out_channels=16, kernel_size=(5, 5))
        self.conv3 = nn.Conv2d(in_channels=16, out_channels=120, kernel_size=(5, 5))
        self.linear1 = nn.Linear(75000, 120)
        self.linear2 = nn.Linear(120, 84)
```

```python
        self.linear3 = nn.Linear(84, output_dims)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool(x)
        x = F.relu(self.conv2(x))
        x = self.pool(x)
        x = F.relu(self.conv3(x))
        x = x.view(x.shape[0], -1)
        x = self.linear1(x)
        x = F.relu(self.linear2(x))
        x = self.linear3(x)
        return x


# Pick GPU as device if available
if torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')

print("Training on: {0}".format(device))
```

```
Training on: cuda
```

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

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

In [241…

```python
# Initialize Network and move it to device
model = Net(5)
model.to(device)

# Loss function
criterion = nn.CrossEntropyLoss()

# Optimiser and learning rate
lr = 0.01
optimizer = torch.optim.SGD(model.parameters(), lr=lr)

# Number of iterations for training
epochs = 20

# Training batch size
train_batch_size = 16

# Based on the FishDataset, use the PyTorch DataLoader to load the data during model training
train_dataset = FishDataset(img_path, idxs_train, idxs_test, train=True)
train_dataloader = DataLoader(train_dataset, batch_size=train_batch_size, shuffle=True)
test_dataset = FishDataset(img_path, idxs_train, idxs_test, train=False)
test_dataloader = DataLoader(test_dataset, shuffle=True)
```

Step 3.3: Train the model. (15 points)

Complete the training loop.

In [242…

```python
def train(model, dataloader, epochs, optimizer, criterion, device):
    for epoch in tqdm(range(epochs)):
        model.train()
        loss_curve = []

        for imgs, labs in dataloader:
            # Move data to device
            imgs = imgs.to(device)
            labs = labs.to(device)

            output = model(imgs)
            loss = criterion(output, labs)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            loss_curve += [loss.item()]

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

In [243…

```python
train(model, train_dataloader, epochs, optimizer, criterion, device)
```

```
  5%|▌          | 1/20 [00:02<00:45,  2.38s/it]
--- Iteration 1: training loss = 1.5850 ---
 10%|█          | 2/20 [00:04<00:44,  2.47s/it]
--- Iteration 2: training loss = 1.2218 ---
 15%|█▌         | 3/20 [00:07<00:42,  2.47s/it]
--- Iteration 3: training loss = 1.0049 ---
 20%|██         | 4/20 [00:10<00:40,  2.56s/it]
--- Iteration 4: training loss = 0.8093 ---
 25%|██▌        | 5/20 [00:12<00:38,  2.56s/it]
--- Iteration 5: training loss = 0.6044 ---
 30%|███        | 6/20 [00:15<00:36,  2.60s/it]
--- Iteration 6: training loss = 0.4183 ---
 35%|███▌       | 7/20 [00:17<00:33,  2.61s/it]
--- Iteration 7: training loss = 0.2923 ---
 40%|████       | 8/20 [00:20<00:30,  2.58s/it]
--- Iteration 8: training loss = 0.2511 ---
 45%|████▌      | 9/20 [00:22<00:27,  2.50s/it]
--- Iteration 9: training loss = 0.1500 ---
 50%|█████      | 10/20 [00:25<00:24,  2.49s/it]
--- Iteration 10: training loss = 0.1348 ---
 55%|█████▌     | 11/20 [00:27<00:22,  2.49s/it]
--- Iteration 11: training loss = 0.2688 ---
 60%|██████     | 12/20 [00:30<00:19,  2.47s/it]
--- Iteration 12: training loss = 0.0814 ---
 65%|██████▌    | 13/20 [00:32<00:17,  2.47s/it]
--- Iteration 13: training loss = 0.0333 ---
 70%|███████    | 14/20 [00:35<00:15,  2.51s/it]
--- Iteration 14: training loss = 0.1062 ---
 75%|███████▌   | 15/20 [00:37<00:12,  2.52s/it]
--- Iteration 15: training loss = 0.0220 ---
 80%|████████   | 16/20 [00:40<00:09,  2.49s/it]
--- Iteration 16: training loss = 0.0059 ---
 85%|████████▌  | 17/20 [00:42<00:07,  2.47s/it]
--- Iteration 17: training loss = 0.0024 ---
 90%|█████████  | 18/20 [00:45<00:05,  2.51s/it]
--- Iteration 18: training loss = 0.0012 ---
 95%|█████████▌| 19/20 [00:47<00:02,  2.52s/it]
--- Iteration 19: training loss = 0.0007 ---
100%|██████████| 20/20 [00:50<00:00,  2.52s/it]
--- Iteration 20: training loss = 0.0005 ---
```

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

In [244_

```python
# Return predicted and actual results of testing model on data from dataloader
def test(model, dataloader, device):
    pred = []
    act = []

    for imgs, labs in dataloader:
        # Move data to device
        imgs = imgs.to(device)
        labs = labs.to(device)

        # Update predicted and actual lists
        pred.append(torch.argmax(model(imgs)).item())
        act.append(labs.item())

    return pred, act
```

In [245_

```python
predicted, actual = test(model, test_dataloader, device)
```

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

You can use sklearns related function.

In [246_

```python
from sklearn.metrics import accuracy_score, confusion_matrix, precision_score, recall_score
```

In [247_

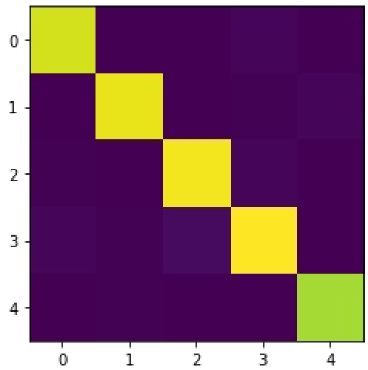
```python
# Print evaluation metrics
def evaluate(pred, act):
    print("Accuracy: {0}\n".format(accuracy_score(act, pred)))
    print("Precisions: {0}\n".format(precision_score(act, pred, average=None)))
    print("Recalls: {0}\n".format(recall_score(act, pred, average=None)))

    conf = confusion_matrix(act, pred)
    print("Confusion Matrix:")
    print(conf)
```

```
    print("\nVisualized:")
    plt.imshow(conf)
    plt.show()
```

In [248…

```
evaluate(predicted, actual)
```

```
Accuracy: 0.977

Precisions: [0.97959184 0.99004975 0.97115385 0.96261682 0.98342541]

Recalls: [0.98461538 0.98029557 0.97584541 0.9537037  0.99441341]

Confusion Matrix:
[[192   0   0   3   0]
 [  0 199   0   1   3]
 [  1   0 202   4   0]
 [  3   1   6 206   0]
 [  0   1   0   0 178]]

Visualized:
```



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'
```

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.

In [249…

```
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, train=True)
train_dataloader = DataLoader(train_dataset, batch_size=train_batch_size, shuffle=True)
test_dataset = FishDataset(img_path, idxs_train, idxs_test, train=False)
test_dataloader = DataLoader(test_dataset, shuffle=True)
```

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`.

In [250…

```
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 frozen.
model.linear3 = torch.nn.Linear(84, len(Multiclass_labels_correspondances))
model.to(device)
```

```python
# Loss function
criterion = nn.CrossEntropyLoss()

# Optimiser and learning rate
lr = 0.01
optimizer = torch.optim.SGD(params=model.parameters(), lr=lr)

# Number of iterations for training
epochs = 30

# Training batch size
train_batch_size = 16
```

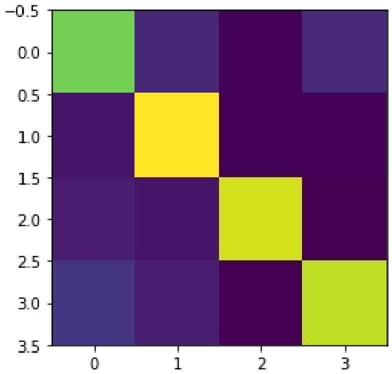
Step 4.3: Train and test your finetuned model. (5 points)

In [251…

```python
# Finetune the model
train(model, train_dataloader, epochs, optimizer, criterion, device)

# Deploy the model on the test set
predicted, actual = test(model, test_dataloader, device)

# Evaluate the performance
evaluate(predicted, actual)
```

```
  3%|▎         | 1/30 [00:00<00:19,  1.52it/s]
--- Iteration 1: training loss = 0.9072 ---
  7%|▋         | 2/30 [00:01<00:19,  1.45it/s]
--- Iteration 2: training loss = 0.6358 ---
 10%|█         | 3/30 [00:02<00:18,  1.44it/s]
--- Iteration 3: training loss = 0.5763 ---
 13%|█▎        | 4/30 [00:02<00:17,  1.46it/s]
--- Iteration 4: training loss = 0.5521 ---
 17%|█▋        | 5/30 [00:03<00:16,  1.48it/s]
--- Iteration 5: training loss = 0.5412 ---
 20%|██        | 6/30 [00:04<00:16,  1.44it/s]
--- Iteration 6: training loss = 0.5156 ---
 23%|██▎       | 7/30 [00:04<00:17,  1.35it/s]
--- Iteration 7: training loss = 0.5156 ---
 27%|██▋       | 8/30 [00:05<00:15,  1.38it/s]
--- Iteration 8: training loss = 0.5103 ---
 30%|███       | 9/30 [00:06<00:15,  1.39it/s]
--- Iteration 9: training loss = 0.5027 ---
 33%|███▎      | 10/30 [00:07<00:14,  1.37it/s]
--- Iteration 10: training loss = 0.5031 ---
 37%|███▋      | 11/30 [00:07<00:13,  1.37it/s]
--- Iteration 11: training loss = 0.4714 ---
 40%|████      | 12/30 [00:08<00:12,  1.40it/s]
--- Iteration 12: training loss = 0.4646 ---
 43%|████▎     | 13/30 [00:09<00:12,  1.41it/s]
--- Iteration 13: training loss = 0.4945 ---
 47%|████▋     | 14/30 [00:09<00:11,  1.41it/s]
--- Iteration 14: training loss = 0.4767 ---
 50%|█████     | 15/30 [00:10<00:10,  1.40it/s]
--- Iteration 15: training loss = 0.4841 ---
 53%|█████▎    | 16/30 [00:11<00:10,  1.40it/s]
--- Iteration 16: training loss = 0.4865 ---
 57%|█████▋    | 17/30 [00:12<00:09,  1.40it/s]
--- Iteration 17: training loss = 0.4729 ---
 60%|██████    | 18/30 [00:12<00:08,  1.39it/s]
--- Iteration 18: training loss = 0.4815 ---
 63%|██████▎   | 19/30 [00:13<00:07,  1.41it/s]
--- Iteration 19: training loss = 0.4524 ---
 67%|██████▋   | 20/30 [00:14<00:07,  1.41it/s]
--- Iteration 20: training loss = 0.4612 ---
 70%|███████   | 21/30 [00:15<00:06,  1.36it/s]
--- Iteration 21: training loss = 0.4575 ---
 73%|███████▎  | 22/30 [00:15<00:06,  1.33it/s]
--- Iteration 22: training loss = 0.4764 ---
 77%|███████▋  | 23/30 [00:16<00:05,  1.34it/s]
--- Iteration 23: training loss = 0.4612 ---
 80%|████████  | 24/30 [00:17<00:04,  1.35it/s]
--- Iteration 24: training loss = 0.4434 ---
 83%|████████▎ | 25/30 [00:18<00:03,  1.34it/s]
--- Iteration 25: training loss = 0.4686 ---
 87%|████████▋ | 26/30 [00:18<00:02,  1.34it/s]
--- Iteration 26: training loss = 0.4682 ---
 90%|█████████ | 27/30 [00:19<00:02,  1.33it/s]
--- Iteration 27: training loss = 0.4587 ---
 93%|█████████▎| 28/30 [00:20<00:01,  1.33it/s]
```

```
--- Iteration 28: training loss = 0.4593 ---
 97%|█████████▋| 29/30 [00:21<00:00,  1.34it/s]
--- Iteration 29: training loss = 0.4477 ---
100%|██████████| 30/30 [00:21<00:00,  1.38it/s]
--- Iteration 30: training loss = 0.4622 ---

Accuracy: 0.8135168961201502

Precisions: [0.71       0.77826087 0.93820225 0.84816754]

Recalls: [0.74736842 0.89949749 0.84343434 0.76415094]

Confusion Matrix:
[[142  21   4  23]
 [ 12 179   4   4]
 [ 16  13 167   2]
 [ 30  17   3 162]]

Visualized:
```



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

It seems that fine-tuning did a relatively good job. Just training the last layer (changed from 5 output neurons to 4), we get a decent accuracy (highest recorded: 0.8625; mean: ~0.8). We do not reach the performance of the original model, but that is to be expected.

Freezing the first layers has two effects: First of all, our model previously learned to classify some types of fish, so it learned to extract certain features from the input and make deductions based on those. By freezing some layers, we see how well our network's 'experience' applies to new types of fish that it had never seen, provided we allow only a part of it to train. Essentially, we are checking how much the features it had already learnt to detect can be used when dealing with new, similar data.

Secondly, freezing speeds up the training process for the network, as the frozen layers are not trained. If this were a very complex network, and we wished to adapt it to some new requirement, as we are doing now, we may wish to only train a part of it, thereby sacrificing some performance in favour of faster training.