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 [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
```

In [12]:
```
# 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):
    idxs_train = int(lendata * percentage)
    idxs_test = idxs_train
    return idxs_train, idxs_test

LENDATA = 5000
np.random.seed(42)
# idxs_train: 4000
# idxs_test: 1000 if the sample size is 5000 in total
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
        dataset = []
        labels = []
        # we are only loading 5 species of fishes
        for dirct in os.listdir(path_to_images):
            if dirct in Multiclass_labels_correspondances:
                for file in os.listdir(os.path.join(path_to_images, dirct)):
                    image_path = os.path.join(path_to_images, dirct, file)
                    image = Image.open(image_path).convert('RGB')
                    # Resize all images to a standard size
                    image = image.resize((img_size, img_size))
                    # convert image to NumPy array, use Image.fromarray() to convert back
                    dataset.append(np.asarray(image))
                    labels.append(Multiclass_labels_correspondances[dirct])


        # Extract the images and labels with the specified file indexes
        dataset = np.array(dataset)
        labels = np.array(labels)

        shuffled_indices = np.random.permutation(len(dataset))
        train_set = dataset[shuffled_indices[:idxs_train]]
        test_set = dataset[shuffled_indices[idxs_test:]]
        train_labels = labels[shuffled_indices[:idxs_train]]
        test_labels = labels[shuffled_indices[idxs_test:]]

        # store train and test set in a dictionary also labels
        self.dataset = train_set if train else test_set
        self.labels = train_labels if train else test_labels


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

    def __getitem__(self, idx):
        # Get an item using its index
        # Return the image and its label
        if idx < self.__len__():
            trans = transforms.ToTensor()
            image = trans(Image.fromarray(self.dataset[idx]))
            label = torch.tensor(self.labels[idx])
            return image, label
```

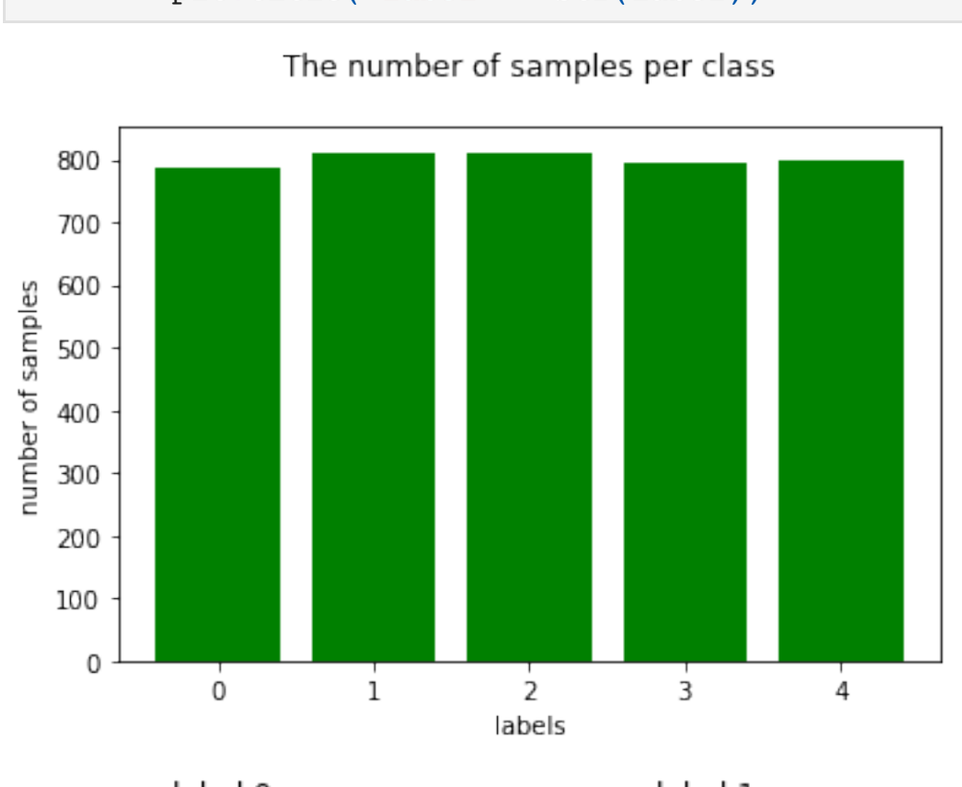
Step 2: Explore the data. (15 Points)

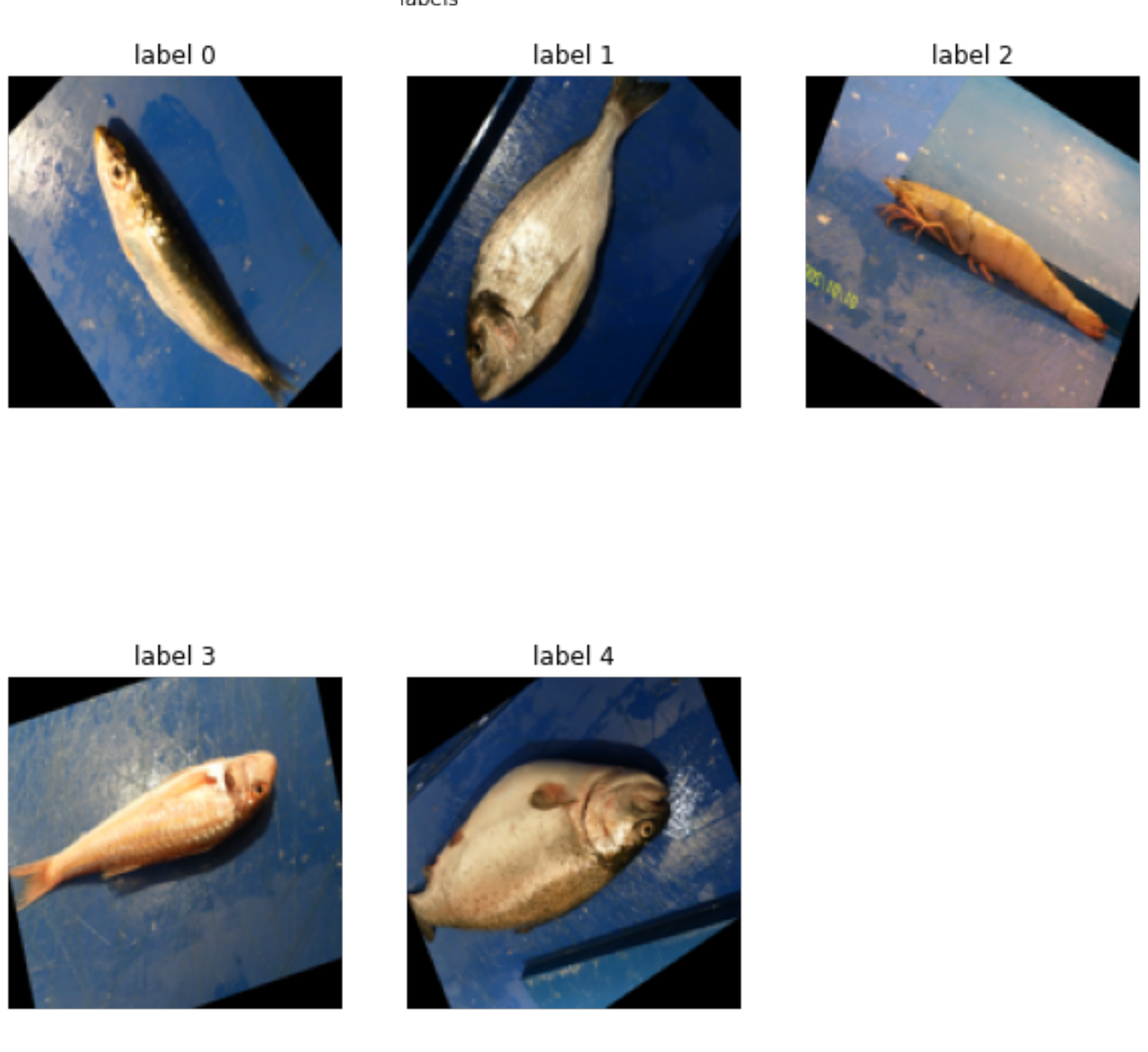
Step 2.1: Data visualisation. (5 points)

- 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 [6]:
```
# Training set
img_path = 'Fish_Dataset'
dataset  = FishDataset(img_path, idxs_train, idxs_test, None, img_size=128, train=True)
```

In [96]:
```
# Plot the number of samples per class
num_bins = len(Multiclass_labels_correspondances)
plt.title('The number of samples per class\n')
plt.xlabel('labels')
plt.ylabel('number of samples')
arr = plt.hist(dataset.labels, bins = np.arange(0, num_bins+1), rwidth = 0.8, align = 'left', facecolor='g')
plt.show()

# Plot 1 sample from each of the five classes in the training set
rows = 2
columns = 3
fig = plt.figure(figsize=(10, 10))
num_of_species = len(Multiclass_labels_correspondances)
species = list(range(num_of_species))
for i in range(len(dataset)):
    if not species:
        break
    image, label = dataset.__getitem__(i)
    label = torch.IntTensor.item(label)
    if label in species:
        convert_to_image = transforms.ToPILImage()
        species.remove(label)
        fig.add_subplot(rows, columns, label+1)
        plt.axis('off')
        plt.imshow(convert_to_image(image))
        plt.title('label '+ str(label))
```





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?

1. Yes the dataset is balanced, with roughly equal number of samples per class labels.
2. (a): We can downsample the majority class - by selecting randomly the same number of samples as the minority class (b): We can also upsample the minority class, find or create more samples, we can use "data augmentation" we talked about during the lecture. (c): If we care about the performance metrics/confsion matrix, it is ok for the dataset to be imbalanced. What we can do is that normalise the result, by dividing by the total number of samples per class labels. This way, the result will no longer be misleading by the majority class.
3. Yes. We have resized all the images to a standard size(128 * 128). The dataset of all images come with labels already given, in a way it's also pre-processed.

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 [20]:
```
# reference implementation to the Pytorch documentation:
# https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html
class Net(nn.Module):
    def __init__(self, output_dims = 1):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 29 * 29, 128)
        self.fc2 = nn.Linear(128, 84)
        self.fc3 = nn.Linear(84, 5)

    def forward(self, x):
        # Forward propagation
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(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 = 'gpu'
```

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

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

In [21]:
```
# Network
model = Net()

# Loss function
criterion = nn.CrossEntropyLoss()

# Optimiser and learning rate
lr = 0.005
optimizer = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9)

# Number of iterations for training
epochs = 20

# Training batch size
train_batch_size = 20
```

In [7]:
```
# 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=True)
```

Step 3.3: Train the model. (15 points)

Complete the training loop.

In [22]:
```
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
        optimizer.zero_grad()
        outputs = model(imgs)
        loss = criterion(outputs, labs)
        loss_curve += [loss.item()]
        loss.backward()
        optimizer.step()

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

```
  5%|██                                      | 1/20 [00:11<03:37, 11.47s/it]
--- Iteration 1: training loss = 1.3786 ---
 10%|████                                    | 2/20 [00:23<03:30, 11.69s/it]
--- Iteration 2: training loss = 0.9069 ---
 15%|██████                                  | 3/20 [00:35<03:21, 11.85s/it]
--- Iteration 3: training loss = 0.6409 ---
 20%|████████                                | 4/20 [00:47<03:10, 11.92s/it]
--- Iteration 4: training loss = 0.4362 ---
 25%|██████████                              | 5/20 [00:59<03:01, 12.08s/it]
--- Iteration 5: training loss = 0.3111 ---
 30%|████████████                            | 6/20 [01:12<02:54, 12.44s/it]
--- Iteration 6: training loss = 0.2056 ---
 35%|██████████████                          | 7/20 [01:25<02:43, 12.61s/it]
--- Iteration 7: training loss = 0.1685 ---
 40%|████████████████                        | 8/20 [01:38<02:32, 12.70s/it]
--- Iteration 8: training loss = 0.1153 ---
 45%|██████████████████                      | 9/20 [01:51<02:20, 12.74s/it]
--- Iteration 9: training loss = 0.0814 ---
 50%|████████████████████                   | 10/20 [02:04<02:07, 12.73s/it]
--- Iteration 10: training loss = 0.0343 ---
 55%|██████████████████████                 | 11/20 [02:17<01:54, 12.75s/it]
--- Iteration 11: training loss = 0.1008 ---
 60%|████████████████████████               | 12/20 [02:29<01:41, 12.74s/it]
--- Iteration 12: training loss = 0.0292 ---
 65%|██████████████████████████             | 13/20 [02:42<01:28, 12.71s/it]
--- Iteration 13: training loss = 0.0294 ---
 70%|████████████████████████████           | 14/20 [02:55<01:16, 12.72s/it]
--- Iteration 14: training loss = 0.0327 ---
 75%|██████████████████████████████         | 15/20 [03:07<01:03, 12.73s/it]
--- Iteration 15: training loss = 0.0059 ---
 80%|████████████████████████████████       | 16/20 [03:20<00:50, 12.72s/it]
--- Iteration 16: training loss = 0.0003 ---
 85%|██████████████████████████████████     | 17/20 [03:33<00:38, 12.72s/it]
--- Iteration 17: training loss = 0.0002 ---
 90%|████████████████████████████████████   | 18/20 [03:45<00:25, 12.68s/it]
--- Iteration 18: training loss = 0.0001 ---
 95%|██████████████████████████████████████ | 19/20 [03:58<00:12, 12.68s/it]
--- Iteration 19: training loss = 0.0001 ---
100%|███████████████████████████████████████| 20/20 [04:11<00:00, 12.57s/it]
--- Iteration 20: training loss = 0.0001 ---
```

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

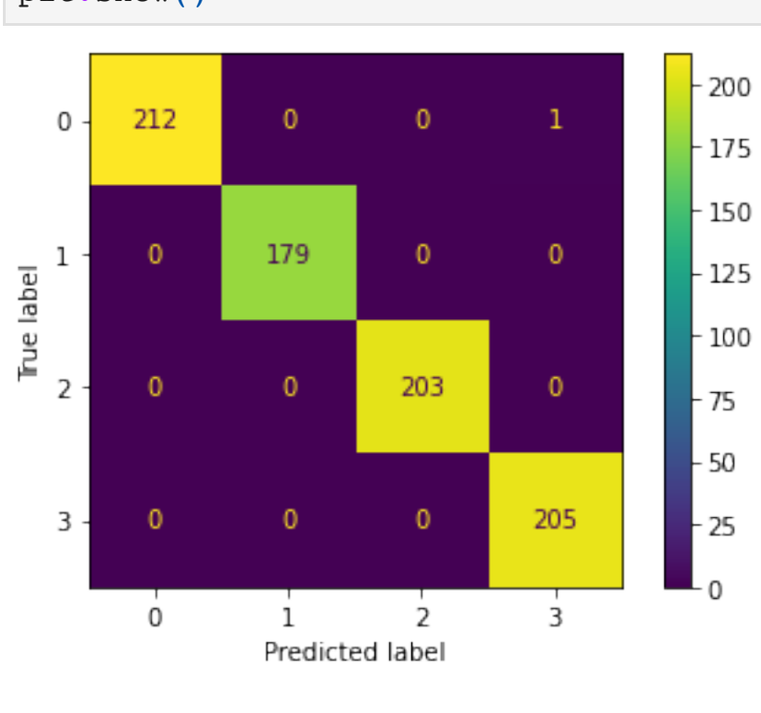
In [23]:
```
# Deploy the model
test_labels = []
pred_labels = []
correct = 0
total = 0
for imgs, labs in test_dataloader:
    # calculate outputs by running images through the network
    outputs = model(imgs)
    # the class with the highest energy is what we choose as prediction
    _, predicted = torch.max(outputs.data, 1)
    test_labels.extend(labs.tolist())
    pred_labels.extend(predicted.tolist())
    total += labs.size(0)
    correct += (predicted == labs).sum().item()
print(f'Accuracy of the model on the {len(test_dataset)} test images: {100 * correct // total} %')
```

```
Accuracy of the model on the 800 test images: 99 %
```

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

You can use sklearns related function.

In [24]:
```
# Evaluate the performance of the model
from sklearn.metrics import ConfusionMatrixDisplay
ConfusionMatrixDisplay.from_predictions(test_labels, pred_labels)
plt.show()
```



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 [13]:
```
# there is an image corrupted in the 'Red Mullet', 00081.png, which cannot be loaded into
# the dataset. A random image in the 'Red Mullet' is chosen and is rotated in random direction
# and renamed as a replace for 00081.png
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=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 [25]:
```
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 = nn.Linear(84, 4)

# Loss function
criterion =nn.CrossEntropyLoss()

# Optimiser and learning rate
lr = 0.05
optimizer = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9)

# Number of iterations for training
epochs = 20

# Training batch size
train_batch_size = 20
```

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

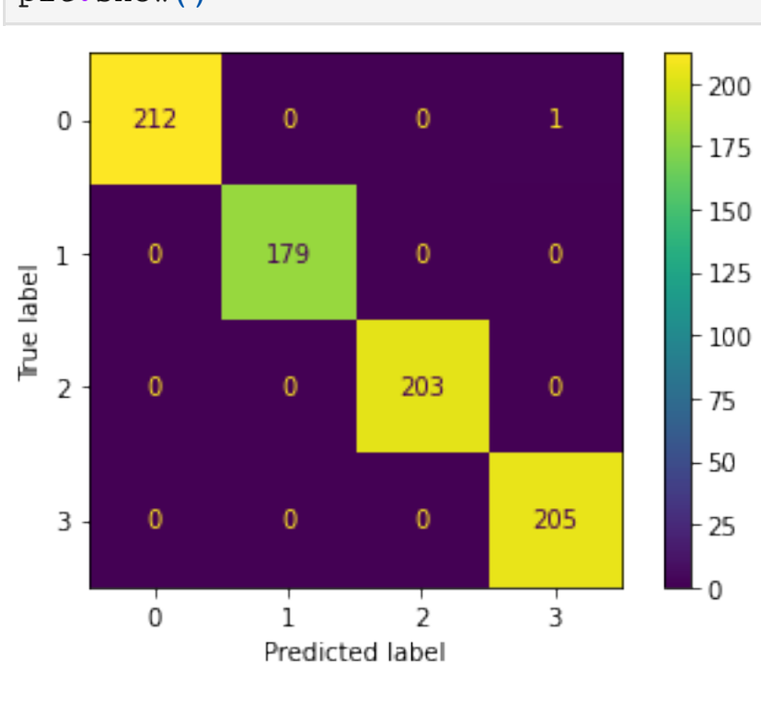
In [26]:
```
# Finetune the model
for epoch in tqdm(range(epochs)):
    model.train()
    loss_curve = []

    for imgs, labs in train_dataloader:
        # Get a batch of training data and train the model
        optimizer.zero_grad()
        outputs = model(imgs)
        loss = criterion(outputs, labs)
        loss_curve += [loss.item()]
        loss.backward()
        optimizer.step()

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

```
  5%|██                                      | 1/20 [00:07<02:25,  7.66s/it]
--- Iteration 1: training loss = 0.0216 ---
 10%|████                                    | 2/20 [00:15<02:17,  7.65s/it]
--- Iteration 2: training loss = 0.0000 ---
 15%|██████                                  | 3/20 [00:23<02:10,  7.68s/it]
--- Iteration 3: training loss = 0.0000 ---
 20%|████████                                | 4/20 [00:30<02:03,  7.72s/it]
--- Iteration 4: training loss = 0.0000 ---
 25%|██████████                              | 5/20 [00:38<01:56,  7.74s/it]
--- Iteration 5: training loss = 0.0000 ---
 30%|████████████                            | 6/20 [00:46<01:48,  7.76s/it]
--- Iteration 6: training loss = 0.0000 ---
 35%|██████████████                          | 7/20 [00:54<01:41,  7.79s/it]
--- Iteration 7: training loss = 0.0000 ---
 40%|████████████████                        | 8/20 [01:02<01:33,  7.80s/it]
--- Iteration 8: training loss = 0.0000 ---
 45%|██████████████████                      | 9/20 [01:09<01:25,  7.82s/it]
--- Iteration 9: training loss = 0.0000 ---
 50%|████████████████████                   | 10/20 [01:17<01:18,  7.82s/it]
--- Iteration 10: training loss = 0.0000 ---
 55%|██████████████████████                 | 11/20 [01:25<01:10,  7.83s/it]
--- Iteration 11: training loss = 0.0000 ---
 60%|████████████████████████               | 12/20 [01:33<01:02,  7.83s/it]
--- Iteration 12: training loss = 0.0000 ---
 65%|██████████████████████████             | 13/20 [01:41<00:54,  7.83s/it]
--- Iteration 13: training loss = 0.0000 ---
 70%|████████████████████████████           | 14/20 [01:48<00:46,  7.79s/it]
--- Iteration 14: training loss = 0.0000 ---
 75%|██████████████████████████████         | 15/20 [01:56<00:38,  7.80s/it]
--- Iteration 15: training loss = 0.0000 ---
 80%|████████████████████████████████       | 16/20 [02:04<00:31,  7.79s/it]
--- Iteration 16: training loss = 0.0000 ---
 85%|██████████████████████████████████     | 17/20 [02:12<00:23,  7.77s/it]
--- Iteration 17: training loss = 0.0000 ---
 90%|████████████████████████████████████   | 18/20 [02:20<00:15,  7.80s/it]
--- Iteration 18: training loss = 0.0000 ---
 95%|██████████████████████████████████████ | 19/20 [02:27<00:07,  7.80s/it]
--- Iteration 19: training loss = 0.0000 ---
100%|███████████████████████████████████████| 20/20 [02:35<00:00,  7.79s/it]
--- Iteration 20: training loss = 0.0000 ---
```

In [27]:
```
# Deploy the model on the test set
test_labels = []
pred_labels = []
correct = 0
total = 0
for imgs, labs in test_dataloader:
    # calculate outputs by running images through the network
    outputs = model(imgs)
    # the class with the highest energy is what we choose as prediction
    _, predicted = torch.max(outputs.data, 1)
    test_labels.extend(labs.tolist())
    pred_labels.extend(predicted.tolist())
    total += labs.size(0)
    correct += (predicted == labs).sum().item()
print(f'Accuracy of the model on the {len(test_dataset)} test images: {100 * correct // total} %')
```

```
Accuracy of the model on the 800 test images: 99 %
```

In [28]:
```
# Evaluate the performance
ConfusionMatrixDisplay.from_predictions(test_labels, pred_labels)
plt.show()
```



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

ADD YOUR RESPONSE HERE

Yes. It can be seen that during training, the model trainning loss reaches 0, and the accuracy on the test set is 99%, with only one sample being falsely classified (same accuracy as the first model).

The freezing is to make the training faster. We can see on the logs on both training process, the first one shows about 12.5 seconds per iteration on average, but the second one had a great increase in training speed, with only 7.7 seconds per iteration, which is 38% percent speed increase.