Imports

Regarding data pre-processing

```
import pandas as pd
import numpy as np
from torchvision import transforms
from functools import reduce
```

To create our dataset

```
from torch.utils.data import Dataset, DataLoader, ConcatDataset
from PIL import Image
```

Related to model designing, training...

```
import torch
from torch import nn
from torchvision.models import resnet18
import matplotlib.pyplot as plt
```

Regarding data visualization

Loading the Data

We will load the data by only having the csv. In the csv, we will find the name and the amount of water in the image, called `score`.

```
train_df, test_df = [pd.read_csv(path) for path in ['train.csv', 'test.csv']]
print(len(train_df), len(test_df), len(test_df) / len(train_df) * 100, "%")
```

```
2130 711 33.38028169014085 %
```

Data Augmentation

The data augmentation is a technique used for augment the training dataset by using the current data collected.

How to create transformation functions with classes?

```
class Resize:

```

```
    def __init__(self, size):
        self.size = size

    def __call__(self, img):
        return img.resize((self.size))


class Rotate:
    """Rotate by one of the given angles."""

    def __init__(self, angle):
        self.angle = angle

    def __call__(self, x):
        return transforms.functional.rotate(x, self.angle)
```

We compose different transformations to augment the data by a factor of 11:

- 3 augmentations by rotating the Image
- 8 augmentations by flipping the image and rotating (or not)

```


first_processing = transforms.Compose([
    transforms.Resize((224,224)),
    transforms.ToTensor()
    ])

rotations = [transforms.Compose([
    first_processing,
    Rotate(angle)]) for angle in [90, 180, 270]]

flips = list(reduce(lambda x, y: list(x) + list(y), [
    [transforms.Compose([
        first_processing,
        flip,
        Rotate(angle)
    ]) for angle in [0, 90, 180, 270]]
            for flip in [transforms.functional.vflip, transforms.
               functional.hflip]]))
```

Finally, we can join all the transformation compositions and the original processing into a list.

```
PROCESSINGS = [first_processing] + rotations + flips
```

Creating the Dataset class

Code for processing data samples can get messy and hard to maintain; we ideally want our dataset code to be decoupled from our model training code for better readability and modularity.

PyTorch provides two data primitives: `torch.utils.data.DataLoader` and `torch.utils.data.Dataset` that allow you to use pre-loaded datasets as well as your own data. Dataset stores the samples and their corresponding labels, and DataLoader wraps an iterable around the Dataset to enable easy access to the samples.

The `Dataset` class needs the implementation of the methods `__len__` and `__getitem__`.

```
class WaterDataset(Dataset):

    def __init__(self, df, processing):
        self.df = df
        self.processing = processing

    def __len__(self):
        return len(self.df)

    def __getitem__(self, idx):
        tup = self.df.iloc[idx]
        img = Image.open(tup['name'])
        img = self.processing(img)
        return img, torch.Tensor([tup['score']])

train_dataset = ConcatDataset(
    [WaterDataset(train_df, processing)
    for processing in PROCESSINGS])
test_dataset = WaterDataset(test_df, first_processing)
print(len(train_dataset), len(test_dataset), len(test_dataset) / len(
    train_dataset) * 100, "%")
```

```
25560 711 2.7816901408450705 %
```

Fitting the datasets to dataloaders

The `Dataset` retrieves our dataset's features and labels one sample at a time. While training a model, we typically want to pass samples in "minibatches", reshuffle the data at every epoch to reduce model overfitting, and use Python's multiprocessing to speed up data retrieval.

`DataLoader` is an iterable that abstracts this complexity for us in an easy API.

```
trainloader = DataLoader(train_dataset, batch_size=64, shuffle=True,
    num_workers=0)
```

```
testloader = DataLoader(test_dataset, batch_size=64, shuffle=True,
    num_workers=0)
```

Designing the model

```
class DeepModel(nn.Module):
    def __init__(self, output):
        super(DeepModel, self).__init__()
        self.model = resnet18(pretrained=True)
        self.output = nn.Sequential(
            nn.Linear(1000, 1),
            nn.Sigmoid(),
        )

    def forward(self, x):
        x = self.model(x)
        return self.output(x)

model = DeepModel(1)
```

Many layers inside a neural network are parameterized, i.e. have associated weights and biases that are optimized during training.

Subclassing `nn.Module` automatically tracks all fields defined inside your model object, and makes all parameters accessible using your model's `parameters()` or `named_parameters()` methods.

```
model
```

We must ensure that the model can inference the instances of the dataset.

Since the shape is the same for all of the images, by proving that one image can be inferenced, it can be inferenced by all of them.

```
model(test_dataset[0][0].reshape(1, 3, 224, 224))
```

```
tensor([[0.3458]], grad_fn=<SigmoidBackward0>)
```

Training

Hyper-parameters

Now that we have a model and data it's time to train, validate and test our model by optimizing its parameters on our data. Training a model is an iterative process.

In each iteration the model makes a guess about the output, calculates the error in its guess (loss), collects the derivatives of the error with respect to its parameters (as we saw in the previous section), and optimizes these parameters using gradient descent.

```
learning_rate = 1e-3
batch_size = 64
epochs = 20
```

```
loss_fn = nn.MSELoss()
```

```
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

The device!

We want to be able to train our model on a hardware accelerator like the GPU, if it is available. Let's check to see if torch.cuda is available, else we continue to use the CPU.

```
DEVICE = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

model = model.to(DEVICE)
```

```
def train_loop(dataloader, model, loss_fn, optimizer, device):
    size = len(dataloader.dataset)
    model.train()
    train_loss = 0
    for batch, (X, y) in enumerate(dataloader):
        X = X.to(device)
        y = y.to(device)
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

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

        # Stats
        train_loss += loss.item()
        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
    return train_loss
```

```


def test_loop(dataloader, model, loss_fn, device):
```

```
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss, correct = 0, 0

    model.eval()
    for X, y in dataloader:
        X = X.to(device)
        y = y.to(device)
        pred = model(X)
        test_loss += loss_fn(pred, y).item()
        correct += (pred.argmax(1) == y).type(torch.float).sum().item()

    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss:
       {test_loss:>8f} \n")
    return test_loss.detach().cpu()
```

```
train_losses = []
test_losses = []
min_loss = np.inf

for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")

    current_train_loss = train_loop(trainloader, model, loss_fn,
       optimizer, DEVICE)
    train_losses.append(current_test_loss)

    current_test_loss = test_loop(testloader, model, loss_fn, DEVICE)
    test_losses.append(current_test_loss)

    print("[{},{}] TRAIN: {:.5f} \t TEST: {:.5f}".format(t, epochs,
       current_train_loss, current_test_loss))

    if min_loss > current_test_loss:
        torch.save(model.state_dict(), 'model_weights.pth')
        min_loss = current_test_loss

print("Done!")
```

Data visualization

```
train_losses = list(range(30))
test_losses = [30 - 1 for i in range(30)]
plt.plot(train_losses, label="Train loss")
plt.plot(test_losses, label="Test loss")
plt.xlabel("Epochs")
```

```
plt.ylabel("MSE")
plt.grid(True)
plt.legend()
```