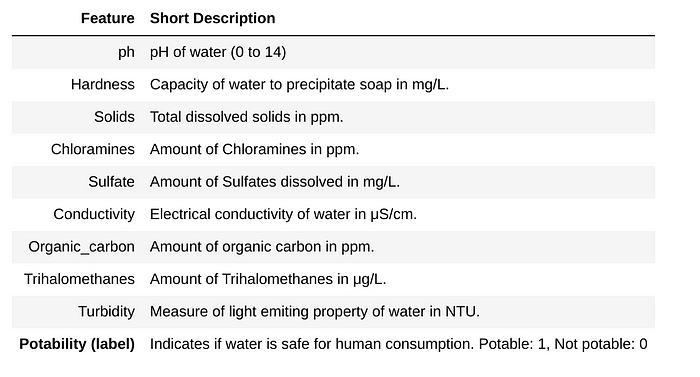
**Introduction to PyTorch**

**Going through the Workflow of a PyTorch Project**

In this article, we will go through the lifecycle of a Deep Learning project using PyTorch. We assume that you are already familiar with Neural Networks and will not explain them in detail, but only consider the PyTorch-specific aspects. We will mainly follow the steps shown in the [official documentation](https://pytorch.org/tutorials/beginner/basics/intro.html) of PyTorch, but consider a different example. In the documentation an example of image classification is presented, here we will consider tabular data stored in a .csv file. This means that some changes will be necessary, especially in preparing the dataset. Having two different examples should help to understand the general workflow of a PyTorch project better. Additionally to this post, you can follow the [colab notebook](https://drive.google.com/file/d/1p31TH09BExMYyo-cm2DfcacoTdxONgwe/view?usp=sharing) including the entire code and workflow structure, or find the notebook on [GitHub](https://github.com/froukje/articles/blob/main/06_pytorch_introduction.ipynb).

**Data**

The data used is downloaded from [kaggle](https://www.kaggle.com/datasets/adityakadiwal/water-potability) [1] and is freely available. The data describes different features that are needed to determine the water quality in an urban environment. The objective is to predict whether the water quality is good or bad. That is we are considering a binary classification problem. In total 9 features, which are all numerical, and the label are given.



The features and the label of the given dataset.

**Preprocessing**

As in every Data Science project, first, some preprocessing needs to be done. This is independent of the deep learning framework we use. Therefore we won´t go into detail here. For further information, please refer to the [colab notebook](https://colab.research.google.com/drive/1p31TH09BExMYyo-cm2DfcacoTdxONgwe) or [GitHub](https://github.com/froukje/articles/blob/main/06_pytorch_introduction.ipynb).

We are lucky and the data does not need very much preparation. All features are numerical and float type. There are however some missing values, which we imputed with the mean of the corresponding feature. Also to consider is that the target variable is not equally distributed, but there are much more 0s (bad water quality) than 1s (good water quality). For the sake of simplicity, we upsampled the data, such that the number of 0s is equal to the number of 1s by randomly drawing samples from the subset of 1s until the same number of samples is reached. Finally, we divided the dataset into a training (80%), validation (10%), and test (10%) set and scaled the data.

**Create a PyTorch Dataset**

In order to use our data in a PyTorch model, we need to bring it into a specific form: a *PyTorch Dataset.* The construction of this dataset is decoupled from the model. The dataset object stores the samples and their corresponding labels. At this point, this example deviates slightly from the PyTorch documentation page. The example data used in the documentation is [FashionMNIST](https://www.kaggle.com/datasets/zalando-research/fashionmnist). For this (and several other datasets) PyTorch offers pre-loaded datasets. To see how to load these datasets, you can check their [PyTorch tutorial](https://pytorch.org/tutorials/beginner/basics/data_tutorial.html). However, if you want to use PyTorch for your own data you most likely have to write your own customized Dataset class.

To create a customized Dataset class, we can inherit from the Dataset class provided by PyTorch. We have to adjust the three following main methods:

The **\_\_init\_\_** method is run once when instantiating the dataset object. In this simple example, only the input and labels are stored as tensors

The **\_\_len\_\_** method returns the number of samples in our dataset.

The **\_\_getitem\_\_** method loads and returns a sample from the dataset at the given index.

The dataset is also the place for transformations when working e.g. with image data. In our tabular data, this is not relevant and therefore not covered here. For the water quality problem considered here, the customized Dataset class looks as follows:

class WaterDataset(Dataset):  
 def \_\_init\_\_(self, X, y):  
   
 # The \_\_init\_\_ method is run once when instantiating the Dataset object  
 self.X = torch.tensor(X)  
 self.y = torch.tensor(np.array(y).astype(float))  
   
 def \_\_len\_\_(self):  
   
 # The \_\_len\_\_ method returns the number of samples in our dataset.  
 return len(self.y)  
   
 def \_\_getitem\_\_(self, idx):  
   
 # The \_\_getitem\_\_ method loads and returns a sample from the dataset  
 # at the given index idx.  
 return self.X[idx], self.y[idx]

Using this class, we define the datasets for the training, validation, and test data.

train\_dataset = WaterDataset(X\_train, y\_train)  
val\_dataset = WaterDataset(X\_val, y\_val)  
test\_dataset = WaterDataset(X\_test, y\_test)

**Define the DataLoader**

After creating a Dataset, we use the PyTorch DataLoader to wrap an iterable around it that permits to easy access the data during training and validation. The Dataset retrieves our dataset’s features and labels one sample at a time. When training a model, we usually want to pass samples of batches and reshuffle the data at every epoch. In this example, when iterating through the DataLoader, each iteration returns a minibatch of 32 samples. It is possible to further configure the DataLoader. For all possible configurations please refer to the [documentation](https://pytorch.org/docs/stable/data.html).

train\_dataloader = DataLoader(train\_dataset, batch\_size=32, shuffle=True)  
val\_dataloader = DataLoader(val\_dataset, batch\_size=32)  
test\_dataloader = DataLoader(test\_dataset, batch\_size=32)

**Define a Model**

Now, the data is prepared and we are ready to define a model. We assume that you are familiar with the general structure of a Neural Net. In PyTorch the **torch.nn** namespace provides all the building blocks to create a Neural Net. The model we use in this example is very simple and only consists of [linear layers](https://pytorch.org/docs/stable/generated/torch.nn.Linear.html#torch.nn.Linear), the [ReLu activation function](https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html#torch.nn.ReLU), and a [Dropout layer](https://pytorch.org/docs/stable/generated/torch.nn.Dropout.html). For an overview of all pre-defined layers in PyTorch, please refer to the [documentation](https://pytorch.org/docs/stable/nn.html).

We can build our own model by inheriting from the **nn.Module***.* A PyTorch model contains at least two methods. The **\_\_init\_\_** method, where all needed layers are instantiated, and the **forward** method, where the final model is defined. Here is an example model, that gives good enough results for our example data.

class WaterNet(nn.Module):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
   
 # define 4 linear layers  
 # the first input dimension is the number of features  
 # the output layer is 1   
 # the other in and output parameters are hyperparamters and can be changed   
 self.fc1 = nn.Linear(in\_features=9, out\_features=64)  
 self.fc2 = nn.Linear(in\_features=64, out\_features=32)  
 self.fc3 = nn.Linear(in\_features=32, out\_features=16)  
 self.fc4 = nn.Linear(in\_features=16, out\_features=1)  
 self.relu = nn.ReLU()  
 self.dropout = nn.Dropout()  
   
 def forward(self, x):  
  
 # apply the linear layers with a relu activation  
 x = self.fc1(x)  
 x = self.relu(x)  
 x = self.dropout(x)  
 x = self.fc2(x)  
 x = self.relu(x)  
 x = self.fc3(x)  
 x = self.relu(x)  
 x = self.fc4(x)  
   
 return x.squeeze()

This model consists of four linear layers. The number of input and output features are defined, which set the size of the input and output sample. Our data consists of 9 features, so the number of input features in the first layer is 9. The output feature size can be changed but must fit the next input feature size. Since we finally want a 1-dimensional output (0 or 1) the last output feature size is equal to 1. Note, that the final sigmoid-layer is not applied here. We will explain this in the next section.

**Train the Model**

Next, we need to train the model. Training a model in PyTorch consists of four main steps:

1. Apply the model
2. Calculate the loss
3. Backpropagation
4. Update the weights

To train one epoch, these steps need to be done for all batches in the *train\_dataloader*. Another loop then needs to go over the desired number of epochs. In pseudocode the training of one epoch looks as follows:

for batch in train\_dataloader:  
 # apply model  
 y\_hat = model(x)  
 # calculate loss  
 loss = loss\_function(y\_hat, y)  
 # backpropagation  
 loss.backward  
 # update weights  
 optimizer.step()

The optimizer and the loss function still need to be defined. We will do this in the next section. Below is a function that includes this training loop. Additionally, some metrics (accuracy, recall, and precision) are calculated. Note, that we set the model to training mode (model.train()) in contrast to the evaluation mode (model.eval()). This affects dropout or batch normalization layers, which are treated differently for training and validation. The inputs of this function are

1. The **model**. This is the above-defined model.

2. The **device**. This can be either GPU or CPU and will be set in the next section.

3. The **train\_dataloader**. The above-defined dataloader for the training data.

4. The **optimizer**. The optimizer is used to minimize the error. We will also specify this in the next section.

5. The **loss function (criterion)**. We will specify this in the next section.

6. The **epoch**. The current epoch.

def train(model, device, train\_dataloader, optimizer, criterion, epoch, print\_every):  
 '''  
 parameters:  
 model - the model used for training  
 device - the device we work on (cpu or gpu)  
 train\_dataloader - the training data wrapped in a dataloader  
 optimizer - the optimizer used for optimizing the parameters  
 criterion - loss function  
 epoch - current epoch  
 print\_every - integer after how many batches results should be printed  
 '''  
   
 # create empty list to store the train losses  
 train\_loss = []  
 # variable to count correct classified samples  
 correct = 0  
 # variable to count true positive, false positive and false negative samples  
 TP = 0  
 FP = 0  
 FN = 0  
 # create an empty list to store the predictions  
 predictions = []  
  
 # set model to training mode, i.e. the model is used for training  
 # this effects layers like BatchNorm() and Dropout  
 # in our simple example we don't use these layers  
 # for the sake of completeness 'model.train()' is included   
 model.train()  
  
 # loop over batches  
 for batch\_idx, (x, y) in enumerate(train\_dataloader):  
   
 # set data to device  
 x, y = x.to(device), y.to(device)  
  
 # set optimizer to zero  
 optimizer.zero\_grad()  
  
 # apply model   
 y\_hat = model(x.float())  
   
 # calculate loss  
 loss = criterion(y\_hat, y.float())  
 train\_loss.append(loss.item())  
  
 # backpropagation  
 loss.backward()  
   
 # update the weights  
 optimizer.step()  
   
 # print the loss every x batches  
 if batch\_idx % print\_every == 0:  
 percent = 100. \* batch\_idx / len(train\_dataloader)  
 print(f'Train Epoch {epoch} \  
 [{batch\_idx \* len(train\_dataloader)}/{len(train\_dataloader.dataset)} \  
 ({percent:.0f}%)] \tLoss: {loss.item():.6f}')  
  
 # calculate some metrics  
   
 # to get the predictions, we need to apply the sigmoid layer  
 # this layer maps the data to the range [0,1]  
 # we set all predictions > 0.5 to 1 and the rest to 0  
 y\_pred = torch.sigmoid(y\_hat) > 0.5  
 predictions.append(y\_pred)  
 correct += (y\_pred == y).sum().item()  
 TP += torch.logical\_and(y\_pred == 1, y == 1).sum()  
 FP += torch.logical\_and(y\_pred == 1, y == 0).sum()  
 FN += torch.logical\_and(y\_pred == 0, y == 1).sum()  
  
 # total training loss over all batches  
 train\_loss = torch.mean(torch.tensor(train\_loss))  
 epoch\_accuracy = correct/len(train\_dataloader.dataset)  
 # recall = TP/(TP+FN)  
 epoch\_recall = TP/(TP+FN)  
 # precision = TP/(TP+FP)  
 epoch\_precision = TP/(TP+FP)  
  
 return epoch\_accuracy, train\_loss, epoch\_recall, epoch\_precision

The validation of the model is similar, but without the backpropagation and without updating the weights. In pseudocode, the validation for one epoch looks like this.

for batch in val\_dataloader:  
 # apply model  
 y\_hat = model(x)  
 # calculate loss  
 loss = loss\_function(y\_hat, y)

A function for validating one epoch is given below. It is very similar to the previous function for training. Note, that the model is set to evaluation mode (model.eval()) and since no gradients need to be calculated during the validation we set with torch.no\_grad(). This will reduce memory consumption for computations.

def valid(model, device, val\_dataloader, criterion):  
 '''  
 parameters:  
 model - the model used for training  
 device - the device we work on (cpu or gpu)  
 val\_dataloader - the validation data wrapped in a dataloader  
 criterion - loss function  
 '''  
   
 # create an empty list to store the loss  
 val\_loss = []  
 # variable to count correct classified samples  
 correct = 0  
 # variable to count true positive, false positive and false negative samples  
 TP = 0  
 FP = 0  
 FN = 0  
 # create an empty list to store the predictions  
 predictions = []  
  
 # set model to evaluation mode, i.e.   
 # the model is only used for inference, this has effects on  
 # dropout-layers, which are ignored in this mode and batchnorm-layers, which use running statistics  
 model.eval()  
  
 # disable gradient calculation   
 # this is useful for inference, when we are sure that we will not call Tensor.backward().   
 # It will reduce memory consumption for computations that would otherwise have requires\_grad=True.  
 with torch.no\_grad():  
 # loop over batches  
 for x, y in val\_dataloader:  
   
 # set data to device  
 x, y = x.to(device), y.to(device)  
  
 # apply model  
 y\_hat = model(x.float())  
   
 # append current loss  
 loss = criterion(y\_hat, y.float())  
 val\_loss.append(loss.item())  
  
 # calculate some metrics  
   
 # to get the predictions, we need to apply the sigmoid layer  
 # this layer maps the data to the range [0,1]  
 # we set all predictions > 0.5 to 1 and the rest to 0  
 y\_pred = torch.sigmoid(y\_hat) > 0.5   
 predictions.append(y\_pred)  
 correct += (y\_pred == y).sum().item()#y\_pred.eq(y.view\_as(y\_pred)).sum().item()  
 TP += torch.logical\_and(y\_pred == 1, y == 1).sum()  
 FP += torch.logical\_and(y\_pred == 1, y == 0).sum()  
 FN += torch.logical\_and(y\_pred == 0, y == 1).sum()  
   
 # total validation loss over all batches  
 val\_loss = torch.mean(torch.tensor(val\_loss))  
 epoch\_accuracy = correct/len(val\_dataloader.dataset)  
 # recall = TP/(TP+FN)  
 epoch\_recall = TP/(TP+FN)  
 # precision = TP/(TP+FP)  
 epoch\_precision = TP/(TP+FP)  
   
 print(f'Validation: Average loss: {val\_loss.item():.4f}, \  
 Accuracy: {epoch\_accuracy:.4f} \  
 ({100. \* correct/len(val\_dataloader.dataset):.0f}%)')  
   
 return predictions, epoch\_accuracy, val\_loss, epoch\_recall, epoch\_precision

**Putting it all Together**

To finally train a neural network using PyTorch, we need to do the following:

1. **Generate Datasets** from the data and wrap them into a **Dataloader**

* We already did this in the previous sections, however, for the sake of completeness we will generate them again.

2. **Define the model**

We use our above-defined model WaterNet.

3. **Define the optimizer**

* There are different [optimizers available in PyTorch](https://pytorch.org/docs/stable/optim.html). We use the Adam optimizer, which is a very common one. You can however try different ones. The Adam optimizer is an extension of the Stochastic Gradient Descent. Said in a simplified way the difference is that Stochastic Gradient Descent keeps the learning rate constant during training, while in Adam it is adapted. An introduction to the Adam optimizer can be found [here](https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/).

4. **Define the loss function**

* We are considering a binary classification problem with a 1-dimensional output, the default choice for this type of problem is the [binary cross entropy](https://towardsdatascience.com/understanding-binary-cross-entropy-log-loss-a-visual-explanation-a3ac6025181a), which we will use.
* Note that we didn’t apply the final [sigmoid-layer](https://pytorch.org/docs/stable/generated/torch.nn.Sigmoid.html) in our model. This was done on purpose since PyTorch offers the [nn.BCEWithLogitsLoss()](https://pytorch.org/docs/stable/generated/torch.nn.BCEWithLogitsLoss.html) method, which combines the final sigmoid layer and the binary cross entropy. We could also apply these two methods separately, i.e. the [nn.Sigmoid](https://pytorch.org/docs/stable/generated/torch.nn.Sigmoid.html) layer as a final step of the model and then the [nn.BCE()](https://pytorch.org/docs/stable/generated/torch.nn.BCELoss.html) loss. However, using nn.BCEWithLogitsLoss() is the recommended way of dealing with binary classification problems, as it is numerically more stable.

Before we start to train the model, we set the hyperparameters. Hyperparameters are adjustable parameters that let you control the model optimization process. Different hyperparameter values can impact model training and convergence rates. In our case, we have three hyperparameters, that we have to set. Note, that also the in- and output features of the model layers are also hyperparameters. We set them to fix values in the model, you can however try different values.

* batch\_size: Training and validation batch size
* epochs: Number of epochs to train
* learning\_rate: The learning rate

We also set the variable print\_every. This is not a hyperparameter but just determines how often the loss is printed during training and validation. Note, that we further have to manually set the device to “cuda” if a GPU is available.

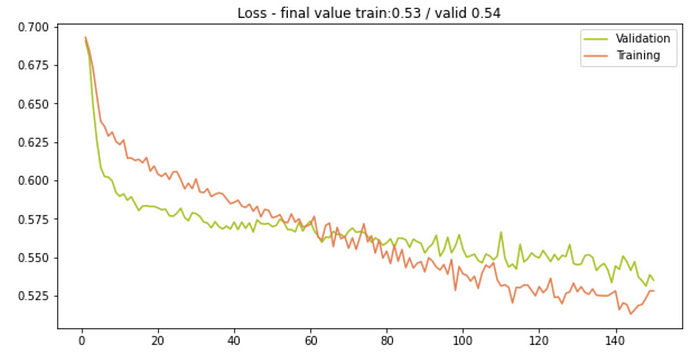
# hyperparamters   
batch\_size = 32  
epochs = 150  
learning\_rate = 1e-3  
print\_every = 200  
  
# set device to GPU, if available  
device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")  
  
# create datasets  
train\_dataset = WaterDataset(X\_train, y\_train)  
valid\_dataset = WaterDataset(X\_val, y\_val)  
  
# wrap into dataloader  
train\_dataloader = DataLoader(train\_dataset, batch\_size=batch\_size, shuffle=True, drop\_last=True)  
val\_dataloader = DataLoader(valid\_dataset, batch\_size=batch\_size)  
  
# define the model and move it to the available device  
model = WaterNet().to(device)  
  
# define the optimizer  
optimizer = torch.optim.Adam(model.parameters(), lr=learning\_rate)#, weight\_decay=1e-4)  
  
# define loss  
criterion = nn.BCEWithLogitsLoss()

Now, we are ready for training. Below is the final training loop. Additionally, the calculated metrics are saved for each epoch.

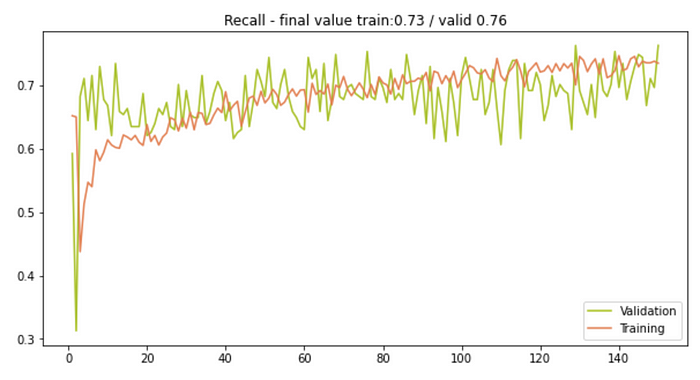
# create empty lists to store the accuracy and loss per validation epoch   
train\_epoch\_accuracy = []  
train\_epoch\_loss = []  
train\_epoch\_recall = []  
train\_epoch\_precision = []  
val\_epoch\_accuracy = []  
val\_epoch\_loss = []  
val\_epoch\_recall = []  
val\_epoch\_precision = []  
  
# loop over number of epochs  
for epoch in range(epochs):  
   
 # train model for each epoch  
 train\_accuracy, train\_loss, train\_recall, train\_precision = \  
 train(model, device, train\_dataloader, optimizer, criterion, epoch, print\_every)  
 # save training loss and accuracy for each epoch  
 train\_epoch\_accuracy.append(train\_accuracy)  
 train\_epoch\_loss.append(train\_loss)  
 train\_epoch\_recall.append(train\_recall)  
 train\_epoch\_precision.append(train\_precision)  
   
 # validate model for each epoch  
 predictions, val\_accuracy, val\_loss, val\_recall, val\_precision = \  
 valid(model, device, val\_dataloader, criterion)  
 # save validation loss and accuracy for each epoch  
 val\_epoch\_accuracy.append(val\_accuracy)

**Evaluate the Results**

To evaluate the results we can have a look at the loss and the metrics during training and evaluation.



The loss for training and validation for 150 epochs.



The recall for training and validation for 150 epochs.

You can find the plots for the other calculated metrics in the [notebook](https://colab.research.google.com/drive/1p31TH09BExMYyo-cm2DfcacoTdxONgwe).

**Save the Model**

If we later want to use our model we need to save it. We can do that by saving it’s state\_dict().

torch.save(model.state\_dict(), 'water\_model\_weights.pth')

We can then load it with

model = WaterNet()  
model.load\_state\_dict(torch.load('water\_model\_weights.pth'))  
model.eval()

Don’t forget to set the model to evaluation mode using model.eval() to put the dropout and batch normalization layers in evaluation mode.

**Apply the Model to the Test Set**

We now apply our trained model to the test data.

# create a dataset  
test\_dataset = WaterDataset(X\_test, y\_test)  
  
# wrap into dataloader  
test\_dataloader = DataLoader(test\_dataset, batch\_size=batch\_size)  
  
predictions, test\_accuracy, test\_loss, test\_recall, test\_precision = \  
 valid(model, device, val\_dataloader, criterion)

**Conclusion**

In this article, a detailed example of how to use PyTorch for a Deep Learning project was shown. The individual steps in the Deep Learning workflow were discussed and applied to a concrete dataset. An essential step before training the model is to bring the data into the correct form and define a customized dataset for the specific application. When training the model the four main steps are (1) apply the model, (2) calculate the loss, (3) perform backpropagation and, (4) update the weights. An example training function, where all these steps are performed is defined and applied. Finally, to use a model it is important to know how to store and reload it.