Pytorch Tutorial

Imports:

```
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader, TensorDataset, random_split
import torch.optim as optim
```

Set device (only for when you may use CUDA):

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

Seeding (reproducibility):

```
seed = 40               #any number
np.random.seed(seed)
torch.manual_seed(seed)
```

Data Reading/Preparation:

```
#data conversion
data = torch.from_numpy(dataset).to(device)  #to(device) possibility of GPU
data.size()

#Combine the X (feature/input) with Y (outputs/labels)
data = TensorDataset(x_tensor, y_tensor)     #data = torch.from_numpy(X,Y)

#spliting data
test_split = 0.8
valid_split = 0.2
samples = len(data)
samples_train     = int(test_split *samples)
samples_test      = samples - samples_train
samples_traindata = int(samples_train*(1-valid_split))
samples_valid     = samples_train - samples_traindata
train_dataset, test_data = random_split(data, [samples_train, samples_test])
train_data, valid_data   = random_split(train_dataset,
                                        [samples_traindata, samples_valid])

#spliting data
Y1hot = F.one_hot(Y, num_classes=8)

#automatic dataloading
train_loader = DataLoader(dataset=train_data, batch_size=16, shuffle=True)
valid_loader = DataLoader(dataset=valid_data, batch_size=16, shuffle=True)
test_loader = DataLoader(dataset=test_data, batch_size=16, shuffle=True)
```

Modeling the Network:

```
#template:
class network_name(nn.Module):
    def __init__(self, parameters):
        super().__init__()
        # define all the object functions from torch.nn needed for forward pass

    def forward(self, x):
        # Computes the outputs / predictions
        return outputs
```

Additional torch.NN functions that can be used in the initialization of the network:

```
nn.Linear()
nn.BatchNorm()
nn.Dropout()
```

Example: Define a network that has 2 inputs, 3 softmax outputs, 3 hidden layers with 4xrelu, 5xsigmoid, 6xtanh, :

```
class network233(nn.Module):
    def __init__(self):
        super().__init__()
        self.L1  = nn.Linear(in_features=2, out_features=4)
        self.L2  = nn.Linear(in_features=4, out_features=5)
        self.L3  = nn.Linear(in_features=5, out_features=6)
        self.out = nn.Linear(in_features=6, out_features=3)
        self.relu = nn.ReLU()
        self.sigm = nn.Sigmoid()
        self.smax = nn.Softmax()

    def forward(self, x):
        # Computes the outputs / predictions
        L1  = F.relu(self.L1(x))
        L2  = F.sigmoid(self.L2 (L1))
        L3  = F.tanh(self.L3(L2))
        out = F.softmax(self.out(L3))
        return out

model = network().to(device)
```

Example: Re-write and add Relu, sigmoid and softmax as object functions:

```
class network233(nn.Module):
    def __init__(self):
        super().__init__()
        self.L1   = nn.Linear(in_features=2, out_features=4)
        self.L2   = nn.Linear(in_features=4, out_features=5)
        self.L3   = nn.Linear(in_features=5, out_features=6)
        self.out  = nn.Linear(in_features=6, out_features=3)
        self.relu = nn.ReLU()
        self.sigm = nn.Sigmoid()
        self.tanh = nn.Tanh()
        self.smax = nn.Softmax()
```

```
    def forward(self, x):
        # Computes the outputs / predictions
        L1  = self.relu (self.L1(x))
        L2  = self.sigm (self.L2 (L1))
        L3  = self.tanh (self.L3(L2))
        out = self.smax (self.out(L3))
        return out

model = network().to(device)
```

Example: make the layer size dependent on function input

```
class network(nn.Module):
    def __init__(self, layer_sizes):
        super().__init__()
        self.L1  = nn.Linear(in_features=layer_sizes[0],out_features=layer_sizes[1])
        self.L2  = nn.Linear(in_features=layer_sizes[1],out_features=layer_sizes[2])
        self.L3  = nn.Linear(in_features=layer_sizes[2],out_features=layer_sizes[3])
        self.out = nn.Linear(in_features=layer_sizes[3],out_features=layer_sizes[4])

    def forward(self, x):
        # Computes the outputs / predictions
        L1  = F.relu(self.L1(x))
        L2  = F.sigmoid(self.L2 (L1))
        L3  = F.tanh(self.L3(L2))
        out = F.softmax(self.out(L3))
        return out


model = network([2,3,4,5,1).to(device)
print(model)
```

Define Learning Functions/Parameters:

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

optimzer = optim.Adam(model.parameters(), lr=0.0001)
```

Define Metric Functions to measure performance:

```
def accuracy_nonbinary(pred, label):
    return (pred.argmax(1) == label).type(torch.float).mean().item()

def accuracy_binary(pred, label):
    return (((pred>=0.5)*1.) == label).type(torch.float).mean().item()
```

Training Function for 1 epoch:

```
def train(dataloader, model, loss_fn, optimizer, metric):
    size = len(dataloader.dataset)
```

```
    loss_T   = 0
    model.train()
    for batch, (X, Y) in enumerate(dataloader):
        X = X.to(device)
        Y = Y.to(device)

        Yhat = model(X)
        loss = loss_fn(Yhat, Y)
        loss_T   += loss.item()
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        print(f"batch: {batch:>5d}, loss: {loss_T:>7f}, metric: {metric(Yhat,Y) :>5f}", end=' ')

    return loss_T
```

Testing Function:

```
def test(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss, metric_T  = 0, 0

    with torch.no_grad():
        for X, Y in dataloader:
            Yhat = model(X)
            test_loss += loss_fn(Yhat, Y).item()
            metric_T  += metric(Yhat, Y)* len(Y)

    test_loss /= num_batches
    metric_T  /= size
    print(f"\nloss: {test_loss:>7f}, metric: { metric_T:>5f}")
```

Training, validating, and testing for all epochs:

```
epochs = 100
for t in range(epochs):
    print(f"Epoch {t}\n")
    train(train_dataloader, model, loss_fn, optimizer, accuracy)
    test (valid_dataloader, model, loss_fn, accuracy)
test (test_dataloader, model, loss_fn, accuracy)
print("Done!")
```

Save/Load model:

```
torch.save(model, "model.pth")
model = torch.load("model.pth")
```