Linear Regression pytorch

```
import torch
import torch.nn as nn
import numpy as np

# Construct Model
class LinearRegression(nn.Module):
    def __init__(self, in_dim, out_dim):
        super(LinearRegression, self).__init__()
        self.lin = nn.Linear(in_features=in_dim, out_features=out_dim)

    def forward(self, x):
        return self.lin(x)

if __name__=='__main__':
    # Define dataset
    X_train = torch.tensor([[1], [2], [3], [4], [5]], dtype=torch.float32)
    y_train = torch.tensor([[2], [4], [6], [8], [10]], dtype=torch.float32)
    y_test = torch.tensor([12.0], dtype=torch.float32)

    n_samples, n_features = X_train.shape
    n_outputs = y_train.shape[1]

    # Define Model
    model = LinearRegression(in_dim=n_features, out_dim=n_outputs)

    # Define Loss Function
    criterion = nn.MSELoss()

    # Hyperparameters
    learning_rate = 0.025
    epochs = 600

    # Define optimizer
    optimizer = torch.optim.SGD(params=model.parameters(), lr=learning_rate)


    # Forward Pass
    for epoch in range(epochs):
        # Training Loop
        # Forward pass
        y_pred =  model(X_train)
        loss = criterion(y_train, y_pred)

        # Backward pass (compute gradients)
        loss.backward()
        # update weights
        optimizer.step()

        # Zero gradients
        optimizer.zero_grad()

        if epoch%10==0:
            print(f"Epoch: {epoch+1}, Loss: {loss.item()}")

    # preditions with test set
    pred = model(y_test)
    print(f"prediction for 12: {pred.item()}")
```

Logistic Regresssion:

```
import torch
import torch.nn as nn
import numpy as np
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

# Construct Model
class LogisticRegression(nn.Module):
    def __init__(self, in_dim, out_dim):
        super(LogisticRegression, self).__init__()
        self.lin = nn.Linear(in_features=in_dim, out_features=out_dim)

    def forward(self, x):
        return torch.sigmoid(self.lin(x))

if __name__=='__main__':
    ## Define dataset
    bc = datasets.load_breast_cancer()
    X = bc['data']
    y = bc['target']
    n_samples, n_features = X.shape

    # Split test and train dataset
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
                                                        shuffle=True, random_state=42)

    # Scale
    sc = StandardScaler()
    X_train = sc.fit_transform(X_train)
    X_test = sc.transform(X_test)

    # Convert dataset into Tensors
    X_train = torch.from_numpy(X_train.astype(np.float32))
    X_test = torch.from_numpy(X_test.astype(np.float32))
    y_train = torch.from_numpy(y_train.astype(np.float32)).view(-1,1)
    y_test = torch.from_numpy(y_test.astype(np.float32)).view(-1,1)

    # Define Model
    model = LogisticRegression(in_dim=n_features, out_dim=1)

    # Define Loss Function
    # criterion = nn.BCELoss()
    criterion = nn.BCEWithLogitsLoss()

    # Hyperparameters
    learning_rate = 0.025
    epochs = 600

    # Define optimizer
    optimizer = torch.optim.SGD(params=model.parameters(), lr=learning_rate)


    # Forward Pass
    for epoch in range(epochs):
        # Training Loop
        # Forward pass
        y_pred =  model(X_train)
        loss = criterion(y_train, y_pred)

        # Backward pass (compute gradients)
        loss.backward()
        # update weights
        optimizer.step()

        # Zero gradients
        optimizer.zero_grad()

        if epoch%10==0:
            print(f"Epoch: {epoch+1}, Loss: {loss.item()}")

    # preditions with test set
    with torch.no_grad():
        pred = model(X_test)
    predicted_classes = (pred > 0.5).float()
    accuracy = (predicted_classes == y_test).sum() / y_test.shape[0]
    print(f"Test Accuracy: {accuracy.item()*100:.2f}%")
```

Dataset and DataLoader

```
import torch
import torch.nn
from torch.utils.data import Dataset, DataLoader
import numpy as np
import math

class wineDataset(Dataset):
    def __init__(self, dataPath):
        data = np.loadtxt(dataPath, delimiter=',', skiprows=1)
        self.X = torch.from_numpy(data[:, 1:])
        self.y = torch.from_numpy(data[:, 0])
        self.n_samples = self.X.shape[0]

    def __getitem__(self, index):
        return self.X[index], self.y[index]

    def __len__(self):
        return self.n_samples



if __name__=='__main__':
    data_path = 'wine.csv'
    dataset = wineDataset(data_path)

    # Verify the wineData is correctely loaded
    first_data = dataset[0]
    features, label = first_data
    print(f'FEATURES: \n {features} \nLABEL: {label}')

    # DataLoader
    dataloader = DataLoader(dataset=dataset, batch_size=4, shuffle=True, num_workers=2)

    # Veriry DataLoader
    dataiter = iter(dataloader)
    data = next(dataiter)
    features, labels = data
    print(f'FEATURES: \n {features} \nLABEL: {labels}') # expect to print data with batch size specified

    # Training loop
    num_epochs = 2
    total_samples = len(dataset)
    n_iterations = math.ceil(total_samples/4) # batchsize = 4

    for epoch in range(num_epochs):
        for i, (inputs, labels) in enumerate(dataloader):
            # Forward pass
            # Backward pass

            if(i+1)%5==0:
                print(f'Epoch: {epoch+1}/{num_epochs}, Step: {i+1}/{n_iterations}, inputs: {inputs.shape}')

    print("Training completed, predict with test dataset")
```

Dataset MNIST

```
import torch
import torchvision
from torch.utils.data import Dataset, DataLoader

dataset = torchvision.datasets.MNIST(root='./data', train=True, transform=torchvision.transforms.ToTensor(), download=True)
dataloader = DataLoader(dataset=dataset, batch_size=4, shuffle=True)

## Training Loop
```