CSC 648 Homework 4: MLP For Regression

Miguel Antonio Logarta

Due: Tuesday Oct 8, at 11:59 PM, 2024

1 Introduction

For this homework assignment, we are tasked with fitting a nonlinear regression curve using a Multilayer Perceptron. It requires prior knowledge from making a Linear Regression curve in a previous exercise and using an MLP to classify a XOR dataset in another exercise.

2 Loading the Data

For our data, we have to read in a csv file called *mlp_regression_data.csv*. In it contains two columns x, and y. Using pandas, I loaded the data into a dataframe. This will make it easier for me to normalize and load my data into the dataset later.

Since the assignment does not require us to split the data into training, validation, and, testing datasets, I just took the entire dataset and assigned it to *X_train* and *y_train*.

```
    # Load the data from csv file
    df = pd.read_csv('mlp_regression_data.csv')

    # Convert to numpy arrays
    X = df["x"].values # Features (Only 1 feature)
    y = df["y"].values # Labels (Only 1 class)

    # Set our training data
    X_train = X
    y_train = y
```

I also modified *MyDataset()* class to transform X and y from pandas dataframes into Torch Tensors. I learned that the code also transforms X and y from 1 dimensional array, into a 2 dimensional array with 1 column. This helped greatly when it came to training the model.

```
# Data loader
class MyDataset(Dataset):
    def __init__(self, X, y):
```

```python
        self.features = torch.tensor(X, dtype=torch.float32)
        self.features = self.features.view(-1,1) # Turn this
            into a 2D tensor with 1 column

        self.labels = torch.tensor(y, dtype=torch.float32)
        self.labels = self.labels.view(-1, 1)

    def __getitem__(self, index):
        x = self.features[index]
        y = self.labels[index]
        return x, y

    def __len__(self):
        return self.labels.shape[0]
```

```python
    # Load training data
    train_ds = MyDataset(X_train, y_train)

    # Load dataset into dataloader
    train_loader = DataLoader(
        dataset=train_ds,
        batch_size=32,
        shuffle=True,
    )
```

3 MLP Model

For my MLP Model, there were multiple things to configure. We only have 1 feature and 1 class. For my 3 hidden layers, I had 50 neurons for my first layer, then 100 for my second layer.

For my activation function, I learned that we had three activation functions from class. We have sigmoid, Linear Activation, and ReLU. Sigmoid is best for binary classification problems, but that isn't the probelm we're solving here. A linear activiation function seems to be the best sinec its goal is to predict a continous value (which is y in this case). ReLU is the same, however, it is for problems where the output is non-negative.

Since I know that our data is continous and our output data is non-negative, I chose ReLU as my activiation function.

```python
# Our MLP Model
class MLP(torch.nn.Module):
    def __init__(self, num_features, num_classes):
        super().__init__()

        self.all_layers = torch.nn.Sequential(
            # 1st hidden layer
            torch.nn.Linear(num_features, 50),
```

```
            # torch.nn.Sigmoid(),
            torch.nn.ReLU(),

            # 2nd hidden layer
            torch.nn.Linear(50, 100),
            # torch.nn.Sigmoid(),
            torch.nn.ReLU(),

            # output layer
            torch.nn.Linear(100, num_classes),
        )

    def forward(self, x):
        logits = self.all_layers(x)
        return logits
```

4 Training Loop

Here is my training loop. I am using the mse_loss() function. In the loop, I also computed the loss per batch and took the average of those losses into our epoch loss.

```
    # Begin training loop
    torch.manual_seed(123)
    model = MLP(num_features=1, num_classes=1) # 1 input
        neuron 1 output neuron
    optimizer = torch.optim.SGD(model.parameters(), lr=0.1)
        # Stochastic gradient descent

    num_epochs = 200
    losses = []

    for epoch in range(num_epochs):
        running_loss = 0.0
        model = model.train()
        for batch_idx, (features, labels) in enumerate(
            train_loader):

            # Forward pass
            logits = model(features)
            loss = F.mse_loss(logits, labels) # Loss
                function

            # Backwards pass
            optimizer.zero_grad()
            loss.backward()

            # Update model parameters
```

```
            optimizer.step()

            # Log the loss
            running_loss += loss.item()
            print(f'Epoch: {epoch+1:03d}/{num_epochs:03d}'
                f' | Batch {batch_idx+1:03d}/{len(
                    train_loader):03d}'
                f' | Loss: {loss:.2f}')

        # Divide the total running loss of the epoch by the
            size of the batch
        losses.append(running_loss / len(train_loader))
```

5 Normalizing the Data

Early on, I realized that not normalizing my data caused significant issues with my loss function. For example, when I was using ReLU, the loss would quickly spiral out of control and go to infinity. It caused the value of the loss to become NaN. So before, I plugged the data into the model, I normalized the data like so:

```
    # Normalize training data
    X_mean = X_train.mean(axis=0)
    X_std = X_train.std(axis=0)
    y_mean = y_train.mean(axis=0)
    y_std = y_train.std(axis=0)

    X_train = (X_train - X_mean) / X_std
    y_train = (y_train - y_mean) / y_std
```

This fixed my problems with the loss function

6 Plotting the data points

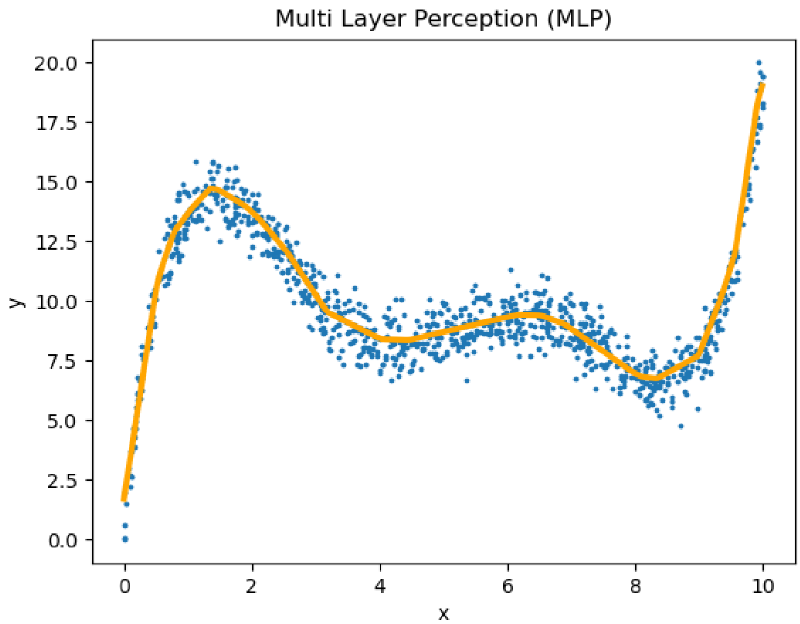
Using matplotlib, I had three separate functions to plot the original data points, the nonlinear regression curve, and the loss as a function of epoch

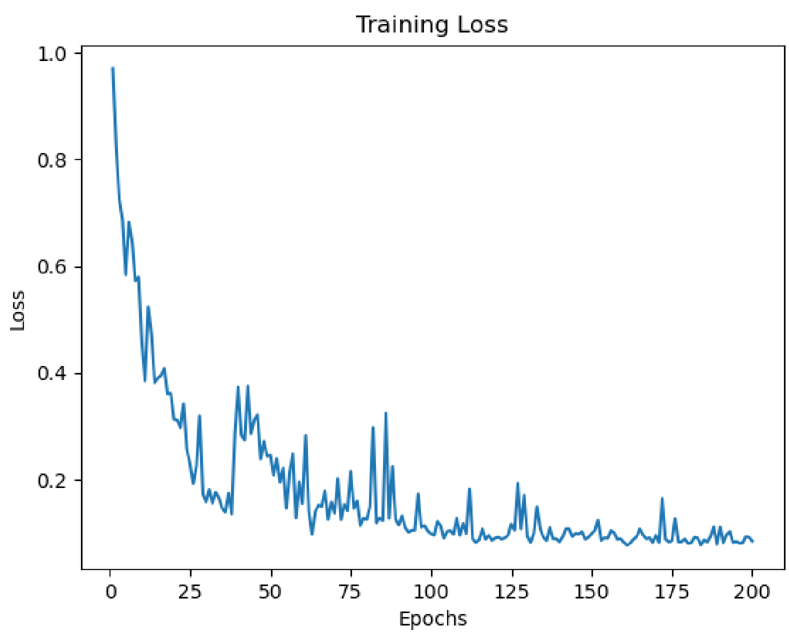
```

def plot_original_data(X, y):
    plt.title("Multi Layer Perception (MLP)")
    plt.scatter(X, y, s=3, label="Original Data")
    plt.xlabel("x")
    plt.ylabel("y")


def plot_decision_boundary(X, y, X_train, model, step=0.01):
    # Get data points
```

```python
    X_range = torch.arange(X_train.min(), X_train.max(), step).view(-1, 1)
    y_pred = model(torch.tensor(X_range, dtype=torch.float32)).detach().numpy()

    X_mean = X.mean(axis=0)
    X_std = X.std(axis=0)
    y_mean = y.mean(axis=0)
    y_std = y.std(axis=0)

    # Denormalize
    X_range = (X_range*X_std)+X_mean
    y_pred = (y_pred*y_std)+y_mean

    plt.plot(X_range, y_pred, color='orange', label='Regression Curve', linewidth=3)


def plot_training_loss(loss):
    plt.title("Training Loss")
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.plot(range(1, len(loss) + 1), loss, label="Training Loss")
    plt.show()
```

```python
    # Plot training results
    plot_original_data(X, y)
    plot_decision_boundary(X, y, X_train, model, step=0.01)
    plt.show()

    # Plot the training loss
    plot_training_loss(losses)
```

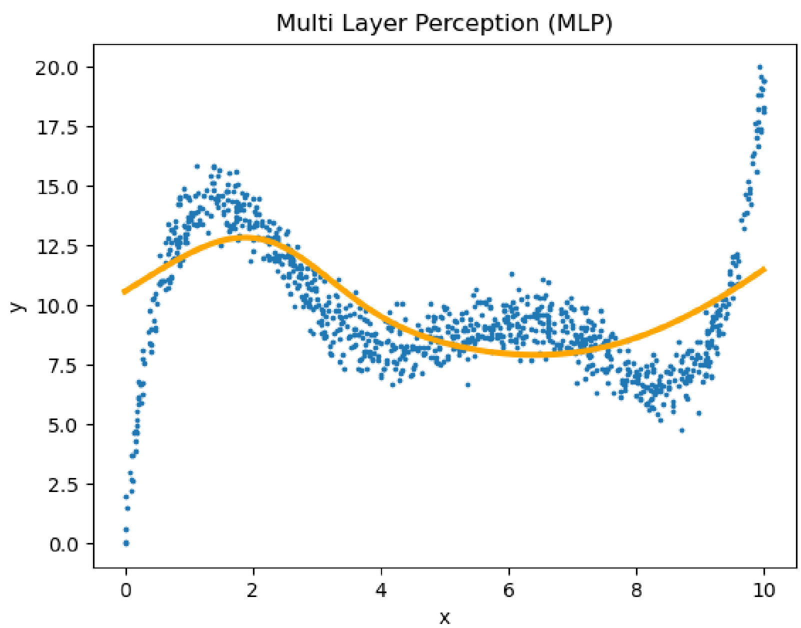
Here are my results:

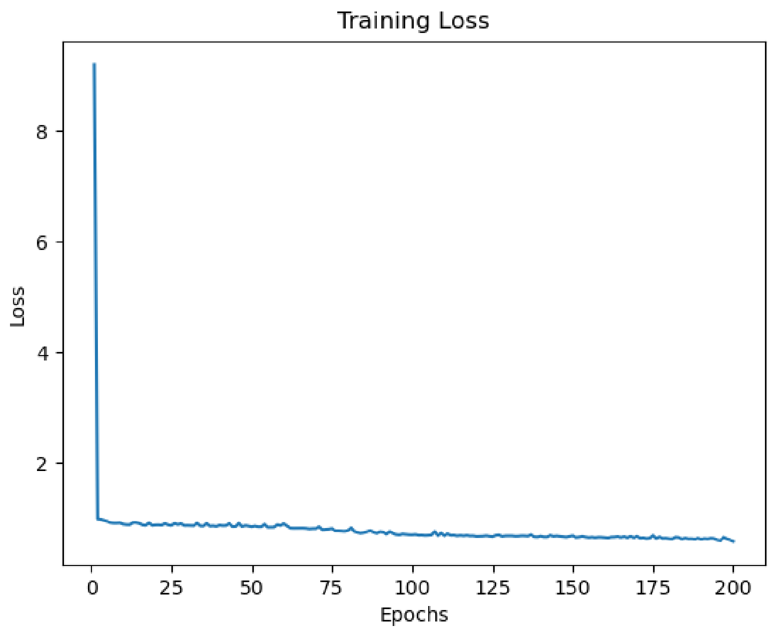

Multi Layer Perception (MLP)
y
x
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
0
2
4
6
8
10


Training Loss
Loss
Epochs
0.2
0.4
0.6
0.8
1.0
0
25
50
75
100
125
150
175
200

7 Hyper Parameters

I also played with some of the hyper parameters. The most interesting thing I found was how changing ReLU to Sigmoid made the regression line really inaccurate.




Training Loss
8
6
4
2
Loss
0
25
50
75
100
125
150
175
200
Epochs