

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*.

```
1      # Load the data from csv file
2      df = pd.read_csv('mlp_regression_data.csv')
3
4      # Convert to numpy arrays
5      X = df["x"].values # Features (Only 1 feature)
6      y = df["y"].values # Labels (Only 1 class)
7
8      # Set our training data
9      X_train = X
10     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.

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

```

4         self.features = torch.tensor(X, dtype=torch.float32)
5         self.features = self.features.view(-1,1) # Turn this
           into a 2D tensor with 1 column
6
7         self.labels = torch.tensor(y, dtype=torch.float32)
8         self.labels = self.labels.view(-1, 1)
9
10        def __getitem__(self, index):
11            x = self.features[index]
12            y = self.labels[index]
13            return x, y
14
15        def __len__(self):
16            return self.labels.shape[0]

```

```

1    # Load training data
2    train_ds = MyDataset(X_train, y_train)
3
4    # Load dataset into dataloader
5    train_loader = DataLoader(
6        dataset=train_ds,
7        batch_size=32,
8        shuffle=True,
9    )

```

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 problem we're solving here. A linear activation function seems to be the best since its goal is to predict a continuous 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 continuous and our output data is non-negative, I chose ReLU as my activation function.

```

1    # Our MLP Model
2    class MLP(torch.nn.Module):
3        def __init__(self, num_features, num_classes):
4            super().__init__()
5
6            self.all_layers = torch.nn.Sequential(
7                # 1st hidden layer
8                torch.nn.Linear(num_features, 50),

```

```
9         # torch.nn.Sigmoid(),
10        torch.nn.ReLU(),
11
12        # 2nd hidden layer
13        torch.nn.Linear(50, 100),
14        # torch.nn.Sigmoid(),
15        torch.nn.ReLU(),
16
17        # output layer
18        torch.nn.Linear(100, num_classes),
19    )
20
21    def forward(self, x):
22        logits = self.all_layers(x)
23        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.

```
1    # Begin training loop
2    torch.manual_seed(123)
3    model = MLP(num_features=1, num_classes=1) # 1 input
4           neuron 1 output neuron
5
6    optimizer = torch.optim.SGD(model.parameters(), lr=0.1)
7           # Stochastic gradient descent
8
9    num_epochs = 200
10   losses = []
11
12   for epoch in range(num_epochs):
13       running_loss = 0.0
14       model = model.train()
15       for batch_idx, (features, labels) in enumerate(
16           train_loader):
17
18           # Forward pass
19           logits = model(features)
20           loss = F.mse_loss(logits, labels) # Loss
21           function
22
23           # Backwards pass
24           optimizer.zero_grad()
25           loss.backward()
26
27           # Update model parameters
```

```

23         optimizer.step()
24
25         # Log the loss
26         running_loss += loss.item()
27         print(f'Epoch: {epoch+1:03d}/{num_epochs:03d}',
28               f' | Batch {batch_idx+1:03d}/{len(
29                   train_loader):03d}',
30               f' | Loss: {loss:.2f}')
```

31 # Divide the total running loss of the epoch by the
32 size of the batch
33 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:

```

1     # Normalize training data
2     X_mean = X_train.mean(axis=0)
3     X_std = X_train.std(axis=0)
4     y_mean = y_train.mean(axis=0)
5     y_std = y_train.std(axis=0)
6
7     X_train = (X_train - X_mean) / X_std
8     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

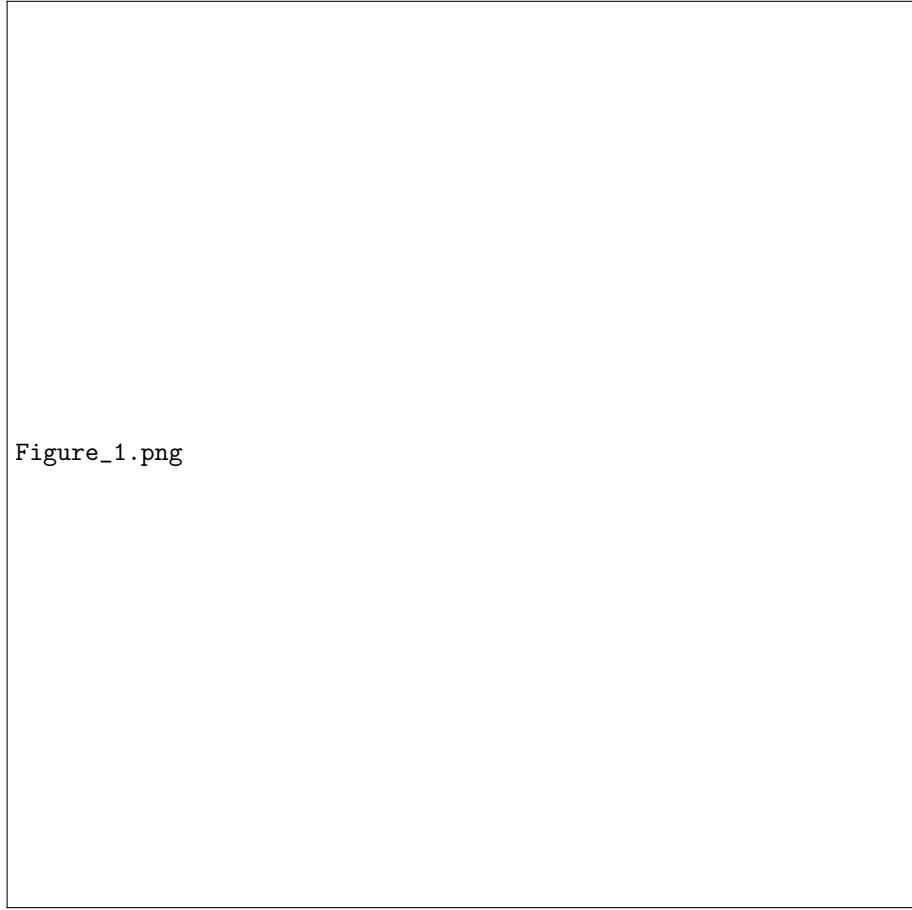
```

1
2     def plot_original_data(X, y):
3         plt.title("Multi Layer Perception (MLP)")
4         plt.scatter(X, y, s=3, label="Original Data")
5         plt.xlabel("x")
6         plt.ylabel("y")
7
8
9     def plot_decision_boundary(X, y, X_train, model, step=0.01):
10        # Get data points

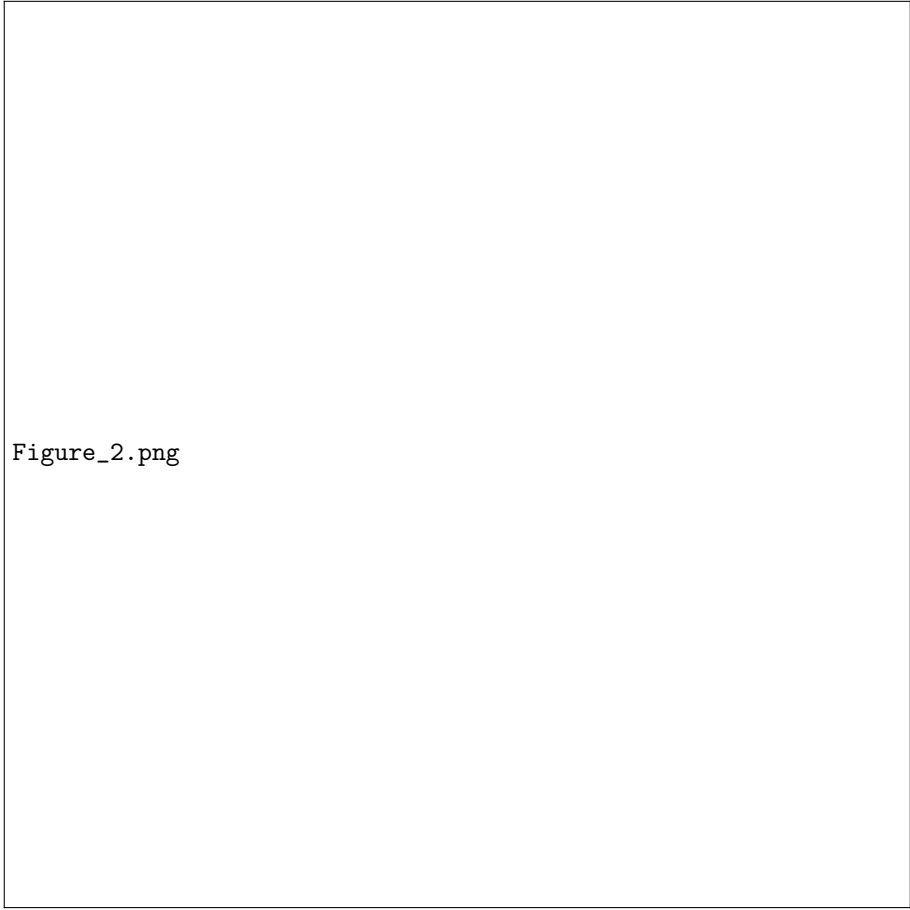
```

```
11 X_range = torch.arange(X_train.min(), X_train.max(),
12                        step).view(-1, 1)
13 y_pred = model(torch.tensor(X_range, dtype=torch.float32
14                             15)).detach().numpy()
16
17 X_mean = X.mean(axis=0)
18 X_std = X.std(axis=0)
19 y_mean = y.mean(axis=0)
20 y_std = y.std(axis=0)
21
22 # Denormalize
23 X_range = (X_range*X_std)+X_mean
24 y_pred = (y_pred*y_std)+y_mean
25
26 plt.plot(X_range, y_pred, color='orange', label='
27         Regression Curve', linewidth=3)
28
29 def plot_training_loss(loss):
30     plt.title("Training Loss")
31     plt.xlabel("Epochs")
32     plt.ylabel("Loss")
33     plt.plot(range(1, len(loss) + 1), loss, label="Training
34             Loss")
35     plt.show()
36
37 # Plot training results
38 plot_original_data(X, y)
39 plot_decision_boundary(X, y, X_train, model, step=0.01)
40 plt.show()
41
42 # Plot the training loss
43 plot_training_loss(losses)
```

Here are my results:



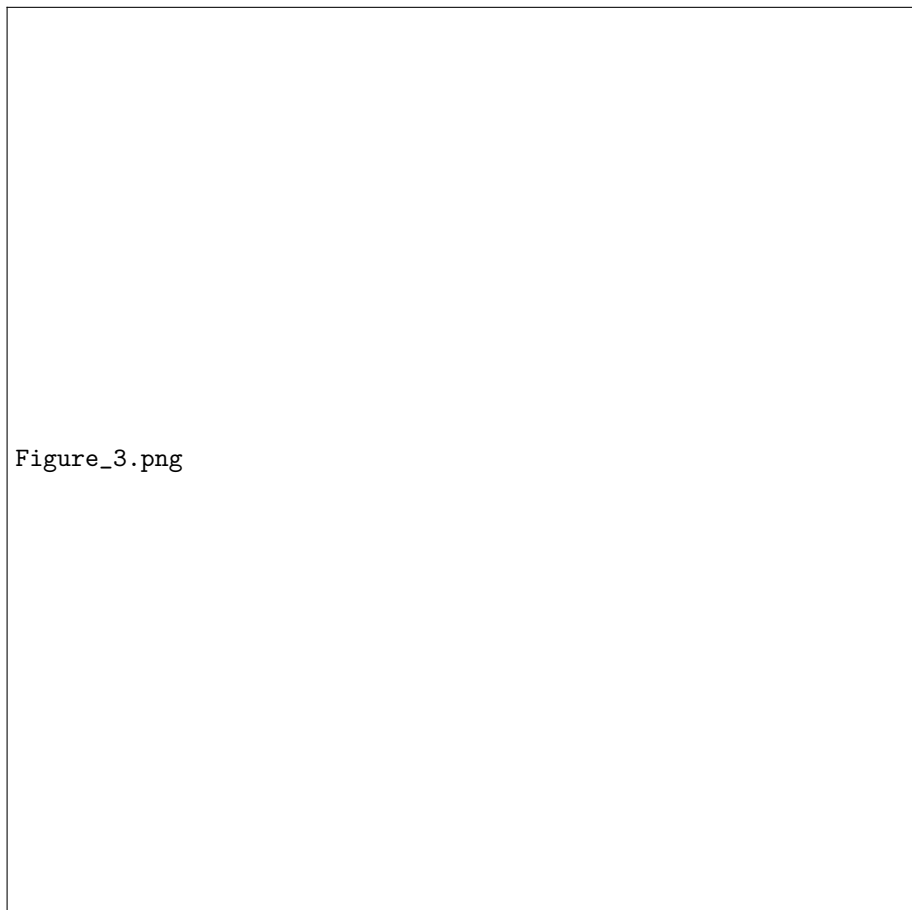
Figure_1.png



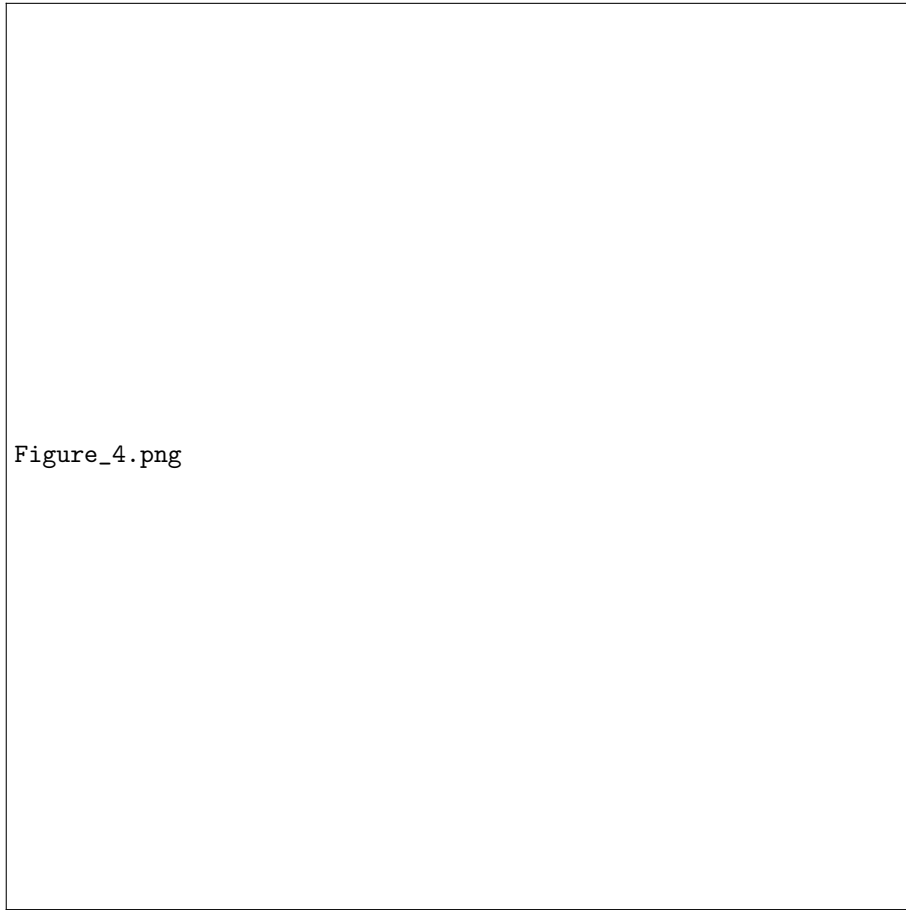
Figure_2.png

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.



Figure_3.png



Figure_4.png