Kaleb Hannan

HW5

12/3/24

When I first started working, I used pandas to load the data from the CSV into a data frame so that I could work with it. After loading the data into the program using pandas, I split the number of games won from the rest of the data. Then, after getting the main data separate from the number of games won, I normalized the data so that all of the values would be between 0 and 1 so that no one field had more influence than the others.

To split the data by percentage, I used the sklean library and split the data into 85% training and 15% test.

I used all of the data to make a prediction except the 'Wins,' which I used as a target, the 'year,' and the 'team name,' which will not hold any value when making a prediction.

After normalizing the data, I started by creating my neural network with four linear layers using the nn.Sequential() function to create a liner relu stack. The first layer takes in all of the inputs and outputs 64 neurons. The second layer takes 64 neurons and outputs 32 neurons. The third layer takes 32 neurons and returns 16 neurons. The fourth layer takes the 16 neurons and then returns one value representing the number of wins the Model thinks the team got based on the stats given. I used relu as all of the values I pass into the model are positive, and from what I have read, this activation function is a good general activation function, so this is the one that I tried.

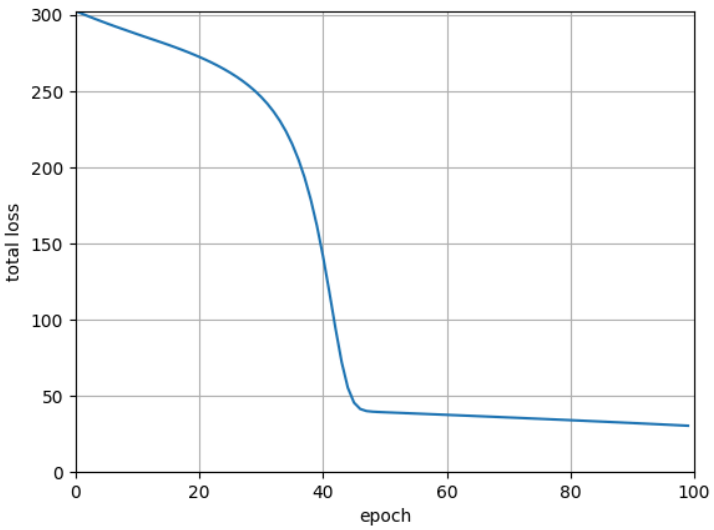
The loss function that I used to train the model was the Mean Squared Error. I used it to see how far away my prediction was from the actual number of wins.

I also used the SGD optimization function when setting up my model.

```
class BBModel(nn.Module):
    def __init__(self,input_size):
        super().__init__()
        self.linear_relu_stack = nn.Sequential(
        nn.Linear(input_size, 64),
        nn.ReLU(),
        nn.Linear(64,32),
        nn.ReLU(),
        nn.Linear(32,16),
        nn.ReLU(),
        nn.Linear(16,1)
        )
    def forward(self, x):
        x = self.linear_relu_stack(x)
        return x
```

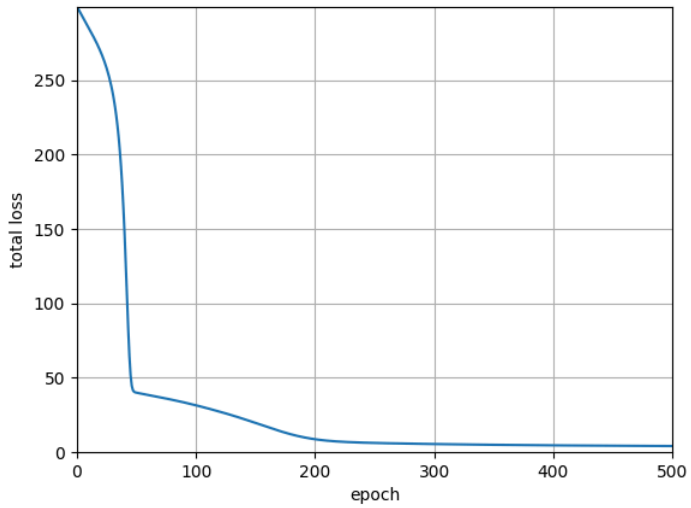
I first trained with 100 epochs using this model at a learning rate of 0.001, and I ended up with a loss of 32.96 when testing and running the testing data against the model.

Fig 1: 100 epochs, 0.001 lr, first model with four layers



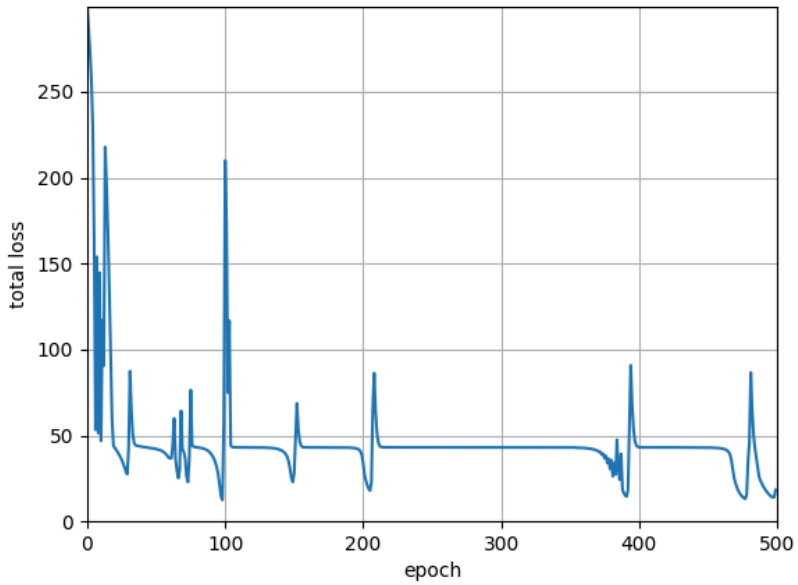
Next, I tried testing using 500 epochs at a learning rate of 0.001. When testing with the model trained with 500 epochs, I got a test loss of 4.5929

Fig 2: 500 epochs, 0.001 lr, first model with four layers



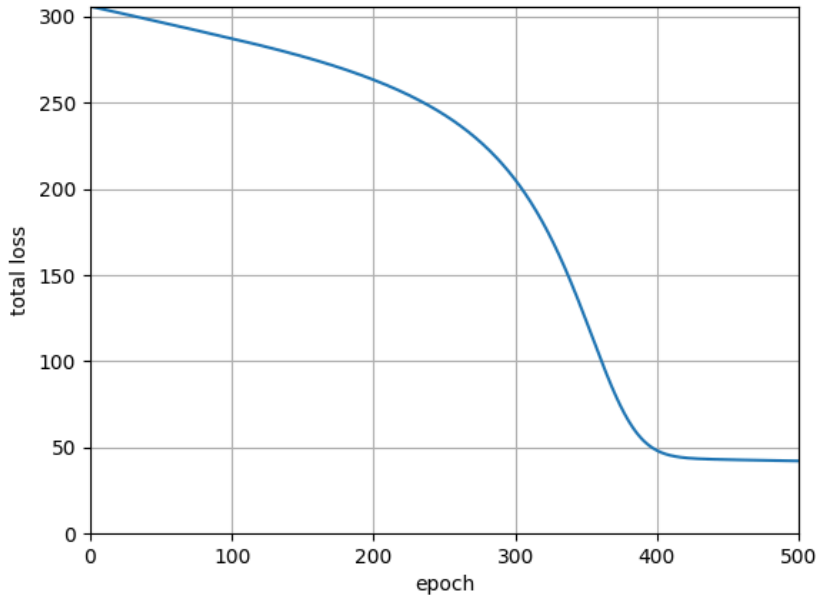
After this, I wanted to test and see what would happen if I changed the learning rate to 0.01. By doing that, my testing loss was changed from 4.3893 to 31.2268. So, I believe that the model was learning too fast.

Fig 3: 500 epochs, 0.01 lr, first model with four layers



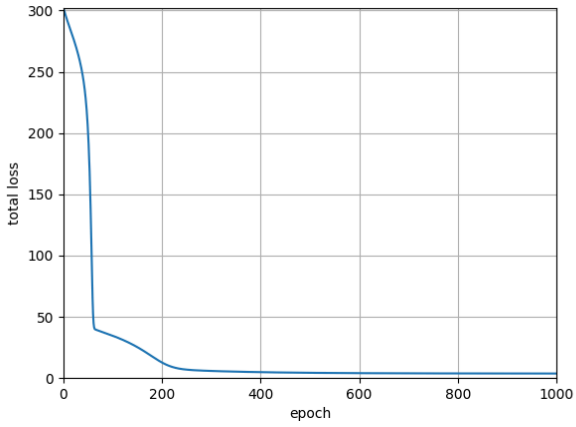
Next, I changed the learning rate to 0.0001 to see if it decreased my loss. When doing that, I got a testing loss of 63.1391. In this case, I did not give it enough epochs model, so the model did not have enough time to learn properly.

Fig 4: 500 epochs, 0.0001 lr, the first model with four layers



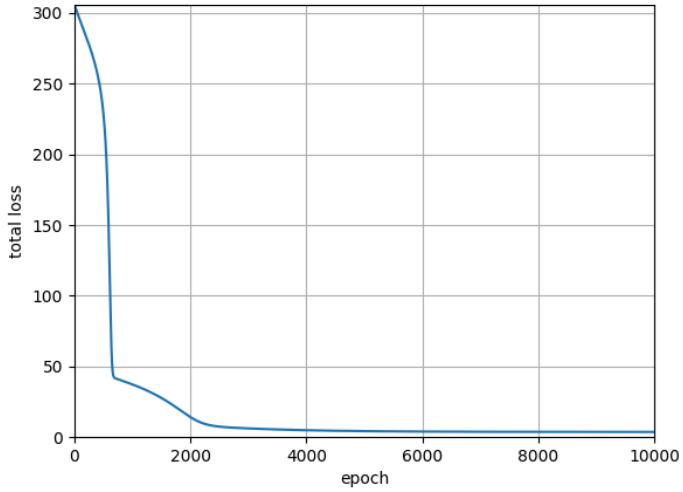
Then, I tried training the model with 1000 epochs to see if I could get the model to be more accurate. Training the model with 1000 epochs, my loss was 3.8071, which was the best I had gotten to this point.

Fig 5: 1000 epochs, 0.001 lr, the first model with four layers



Then I went back and tried the slower learning rate, but I used 10,000 epochs at a learning rate of 0.0001, and I was able to get a test loss of 3.7452, which is slightly better than before.

Fig 6: 10,000 epochs, 0.0001 lr, the first model with four layers

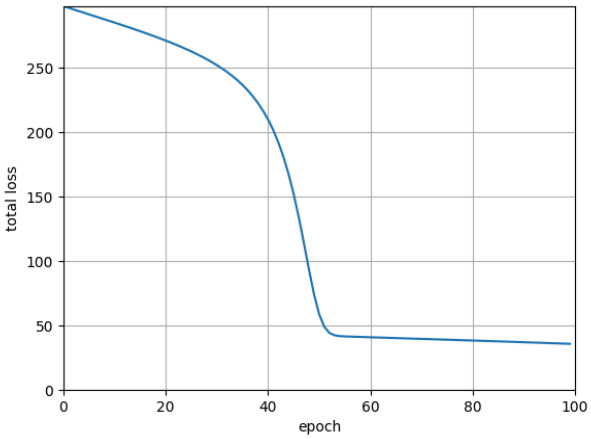


These two graphs are very interesting as they look almost identical, but Figure 6 is at a 100-times scale.

After this, I wanted to test and see how my results would differ when using data that is split by year. To do this, I took the stats from 2010 and 2015, used that test data, and used the rest as training data.

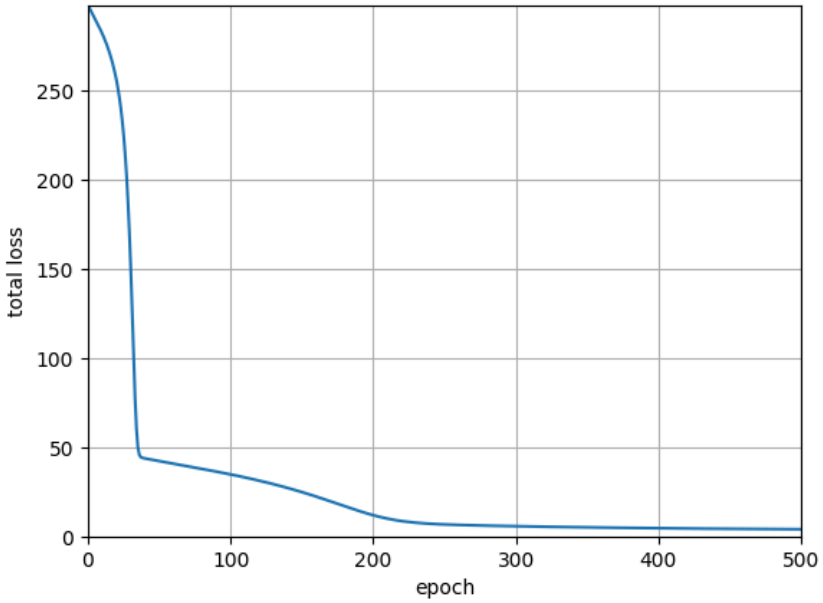
So, I wanted to start back at the beginning to see the difference between the two data sets. With 100 epochs and an LR of 0.001, I got a test loss of 39.6628, which is slightly worse than before.

Fig 7: 100 epochs, 0.001 lr, the first model with four layers using data split by year



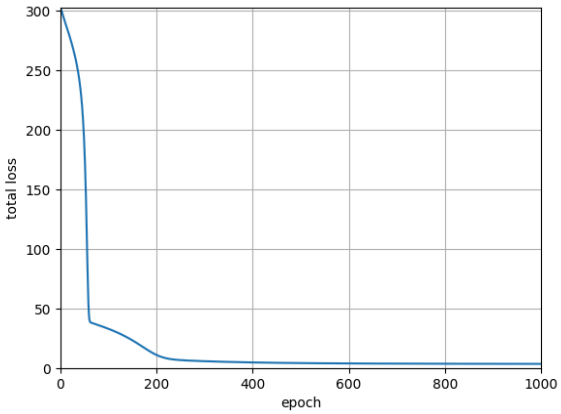
Then, I tried doing 500 epochs with a learning rate of 0.001, and I got a testing loss of 4.2175, which was slightly better than the first data set.

Fig 8: 500 epochs, 0.001 lr, the first model with four layers using data split by year



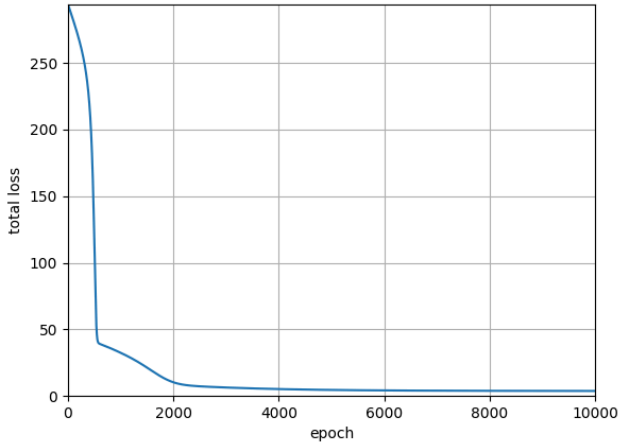
Then, I wanted to try with 1000 epochs to see if that one was better than the first data set. When doing it with 1000 epochs, I got a testing loss of 3.5311, better than the first data set. However, when training the model, its last epoch loss value was 3.8126, which is larger than the testing loss, which I thought was weird.

Fig 9: 1000 epochs, 0.001 lr, the first model with four layers using data split by year



Now, for compassion, I wanted to try 10,000 epochs at a learning rate of 0.0001 to compare to the first data set. This was similar to the others, where the testing loss was slightly better at 3.5169 but was smaller than the last epochs at 3.7823.

Fig 10: 10,000 epochs, 0.0001 lr, the first model with four layers using data split by year



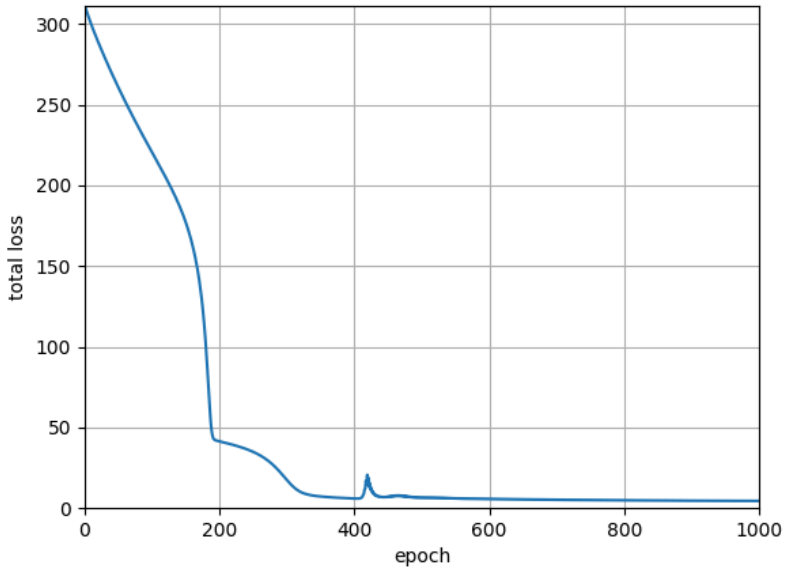
So, the next thing I did was add another layer to my model to see if I could make it more accurate. When testing this, I went back to the data set split by percent 85% training and 15% testing as I did not know if the testing loss being smaller than the training loss with the data split by year was a result of over-training or something else that I just do not understand.

I added one more layer to the network, so it now has five layers: the first layer has 64 neurons, the third layer has 32 neurons, the fourth layer has 16 neurons, and the fourth layer has eight neurons. The last input gives you the output.

```
class BBModel(nn.Module):
    def __init__(self,input_size):
        super().__init__()
        self.linear_relu_stack = nn.Sequential(
        nn.Linear(input_size, 64),
        nn.ReLU(),
        nn.Linear(64,32),
        nn.ReLU(),
        nn.Linear(32,16),
        nn.ReLU(),
        nn.Linear(16,8),
        nn.ReLU(),
        nn.Linear(8,1)
        )
    def forward(self, x):
        x = self.linear_relu_stack(x)
        return x
```

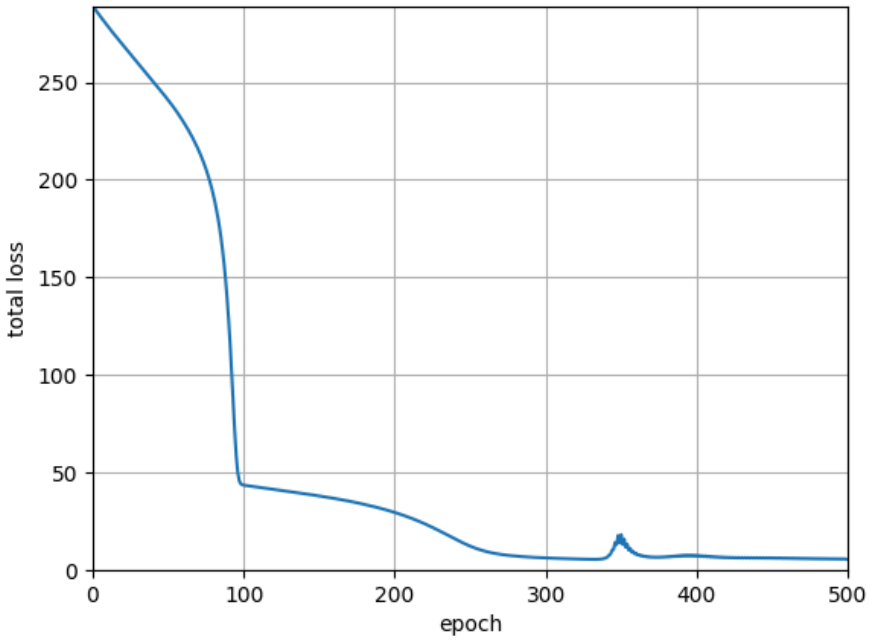
With the 5-layer model with 1000 epochs and lr of 0.001, I got a testing loss of 4.3798, which was not too bad. However, it looks like the training has a slight inconsistency around 400 epochs, and the testing loss was higher with this than with the last model with only 4 layers.

Fig 11: 1000 epochs, 0.001 lr, the second model with five layers using data split by percent



Next, I wanted to test this with 500 epochs, but I did not think it would improve it. And got a test loss rate of 6.2056, which is worse than with only four layers.

Fig 12: 500 epochs, 0.001 lr, the second model with five layers using data split by percent

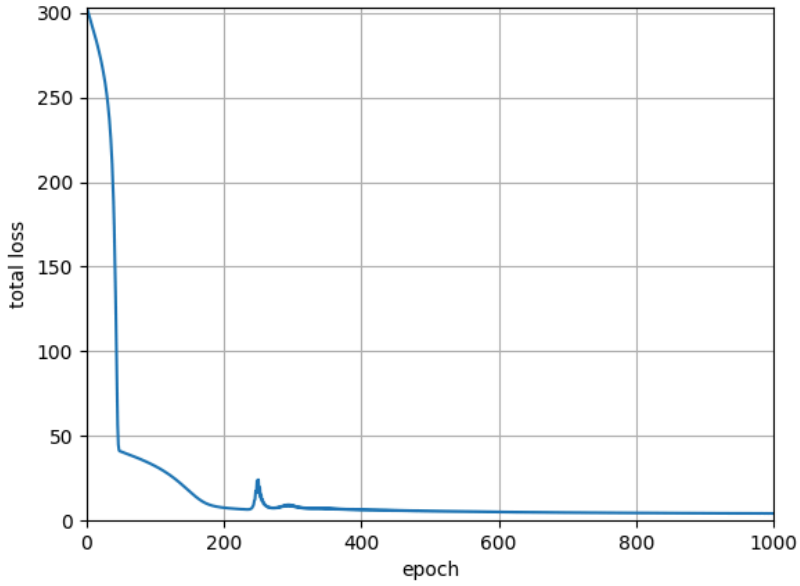


I wanted to change the model and add a layer to the top so the model would be the first layer neurons would be 128, the second layer 64, the third layer would be 3,2, the fourth layer would be 16 neurons, and the 5th layer will be one and the final output.

```
class BBModel(nn.Module):
    def __init__(self,input_size):
        super().__init__()
        self.linear_relu_stack = nn.Sequential(
        nn.Linear(input_size, 128),
        nn.ReLU(),
        nn.Linear(128,64),
        nn.ReLU(),
        nn.Linear(64,32),
        nn.ReLU(),
        nn.Linear(32,16),
        nn.ReLU(),
        nn.Linear(16,1)
        )
    def forward(self, x):
        x = self.linear_relu_stack(x)
        return x
```

Testing with 1000 epochs and a lr of 0.001 resulted in a testing loss of 4.4642, which is worse than the other models I have tried with the same epochs and lr. This model got even worse results than the second iteration of my model.

Fig 13: 1000 epochs, 0.001 lr, the third model with five layers using data split by percent

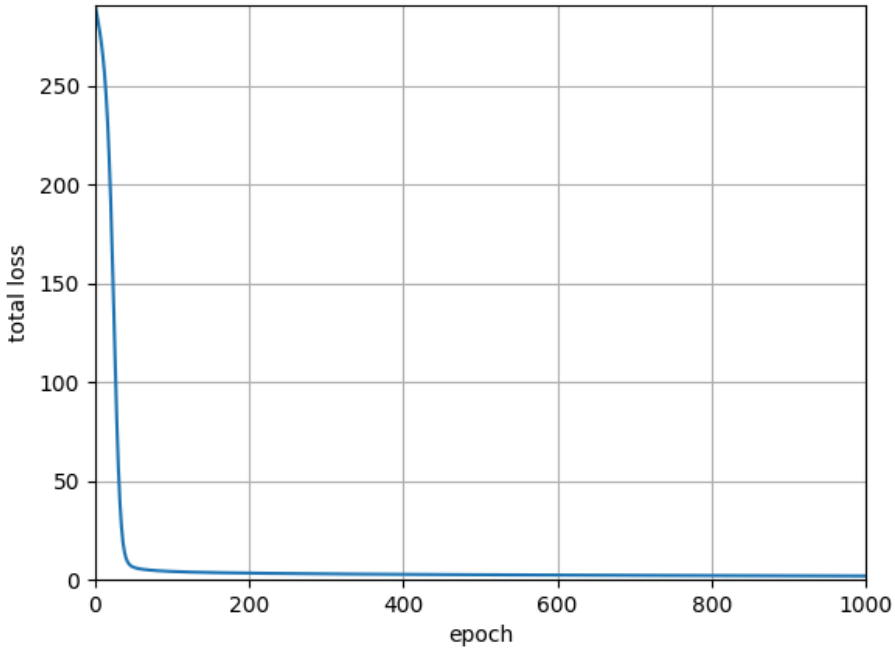


I believe that with this model, they are getting overfit to the data. After this, I tried adding a batch normalization to see if I could improve the third model. I did this to try to help eliminate some of the outlier values that might be in the data set.

```
class BBModel(nn.Module):
    def __init__(self,input_size):
        super().__init__()
        self.linear_relu_stack = nn.Sequential(
        nn.Linear(input_size, 128),
        nn.BatchNorm1d(128),
        nn.ReLU(),
        nn.Linear(128,64),
        nn.BatchNorm1d(64),
        nn.ReLU(),
        nn.Linear(64,32),
        nn.BatchNorm1d(32),
        nn.ReLU(),
        nn.Linear(32,16),
        nn.ReLU(),
        nn.Linear(16,1)
        )
    def forward(self, x):
        x = self.linear_relu_stack(x)
        return x
```

When I tested using this model with 1000 epochs and a lr of 0.001, I got a training loss of 1.9530, but my test loss was 5.1225, which means that my model was probably overfitting and that it is not a good solution for this problem.

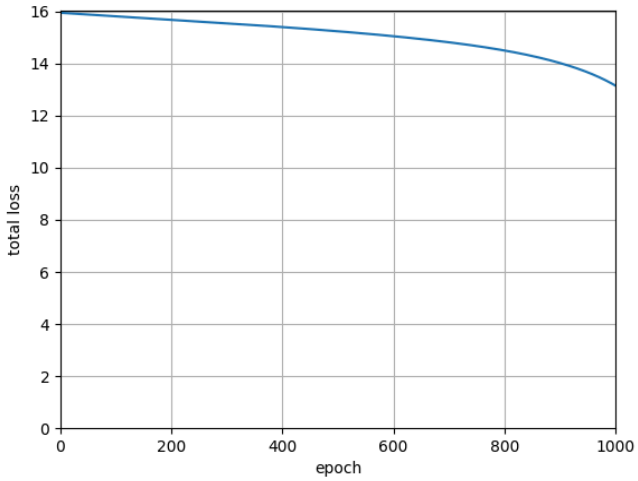
Fig 14: 1000 epochs, 0.001 lr, the fourth model with five layers plus bash normalization using data split by percent.



From this point on, I am going to be testing with my first model with epochs = 1000, and Learning Rate = 0.001 as this is were I have got my best results so far.

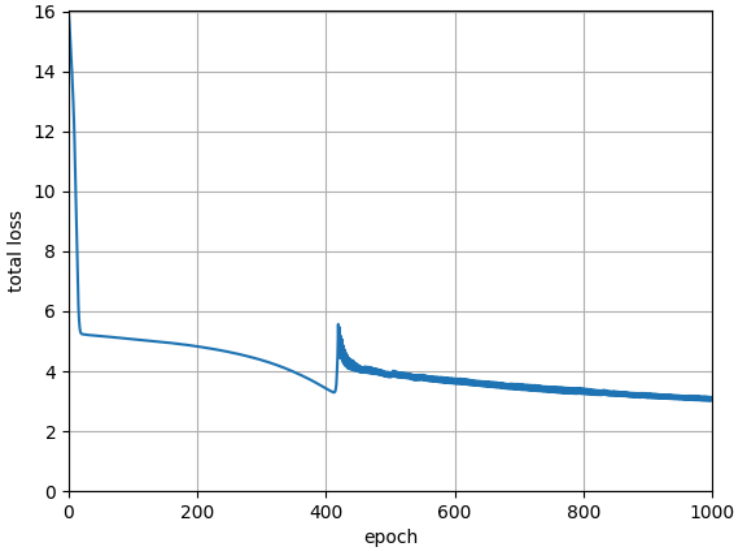
After this, I wanted to compare the Mean Squared Error loss function to the Mean Absolute Error to see how my results would differ from the normalized data between 0 and 1. When doing this, I got a test loss of 13.4731, whereas with the Mean Squared Error, I got a loss of 3.8071. So, the Mean Squared Error is significantly better for how I have my data set up.

Fig 15: Mean Absolute Error 1000 epochs, 0.001 lr, the first model with four layers using data split by percent.



After this, I tried this same test but without normalizing the data between 0 and 1. When doing this with 1000 epochs and an lr of 0.001, I got a training loss of 3.0004, but the model is extremely overfit. But at about 400 epochs, the training loss went from 3.3200 up to 5.5683 and started to come back down slowly.

Fig 16: Mean Absolute Error 1000 epochs, 0.001 lr, the first model with four layers using data split by percent Non-normalized data between 1 and 0.



In conclusion, the final model that worked the best for this problem for me happened to be the first model in which I created using the nn.Sequential() function to create a liner relu stack with four layers. The first layer takes in all of the inputs and outputs 64 neurons. The second layer takes 64 neurons and outputs 32 neurons. The third layer takes 32 neurons and returns 16 neurons. The fourth layer takes the 16 neurons and then returns one value representing the number of wins the Model thinks the team will get. I used the Mean Squared Error loss function, the SGD optimization function with the epochs = 1000, and the learning rate = 0.001. I also thought that training it with the data that was split by percentage, where I had 85% training data and 15% testing data, was slightly more consistent when training this model with the data that I split by year. Also, even though when training this model using. 10,000 epochs and a learning rate = 0.0001, where my best test loss was 3.7452, was less consistent running it more than once than training the model with epochs = 1000, and the learning rate = 0.001, where I got a testing loss of 3.8071 as well as not know if the extra time training would be worth using a larger data set, and that is why I choose epochs = 1000, and the learning rate = 0.001 over 10,000 epochs and a learning rate = 0.0001.