DATA COLLECTION PROCESS

- First randomly game playing and gather 10,000 rows.
- Equal wins on both sides left and right for game to fit the model in a fine way and gather 8,000 rows.
- 4 Columns, and 8,000 Rows
- Data type - Float

```
x1      float64
x2      float64
y1      float64
y2      float64
dtype: object
```

```
# importing game csv file through pandas as a game data frame
game_data_frame = pd.read_csv("ce889_dataCollection.csv")
game_data_frame.head(5)
```

	x1	x2	y1	y2
0	-209.204	394.690	0.722	-0.090
1	-209.114	393.968	0.636	-0.125
2	-208.989	393.332	0.554	-0.166
3	-208.823	392.778	0.474	-0.214
4	-208.608	392.304	0.397	-0.268

DATA PRE-PROCESSING (CONTINUE)

x1	x2	y1	y2
273.75	353.5	0.593	0
273.71	353.6	-0.1	0.04
273.71	353.8	-0.2	0
273.67	354.1	-0.3	0.04
273.663	354.5	-0.4	0.007
273.682	354.999	-0.499	-0.019
273.64	355.598	-0.598	0.042
273.61	356.294	-0.696	0.03
273.586	357.087	-0.793	0.024
273.56	357.976	-0.889	0.026
273.526	358.959	-0.983	0.034

- Removed noise in a way by checking far distribution of rows.

431.962	305.7	0.403	0.035
431.928	305.297	0.307	0.036
431.891	304.99	0.213	0.045
431.847	304.776	0.121	0.06
431.787	304.655	0.031	0.082
431.705	304.624	-0.057	0.11
431.595	304.681	-0.142	0.145
431.45	304.824	-0.228	0.18
431.27	305.052	-0.313	0.215
431.056	305.365	-0.399	0.25
430.806	305.764	-0.484	0.285
430.521	306.248	-0.57	0.32
430.202	306.818	-0.655	0.354
429.847	307.473	-0.741	0.389
429.458	308.213	-0.826	0.424
429.033	309.039	-0.911	0.459

```
0       False
1       False
2       False
3       False
4       False
       ...
5994     True
5995     True
5996     True
5997     True
5998    False
Length: 5999, dtype: bool
```

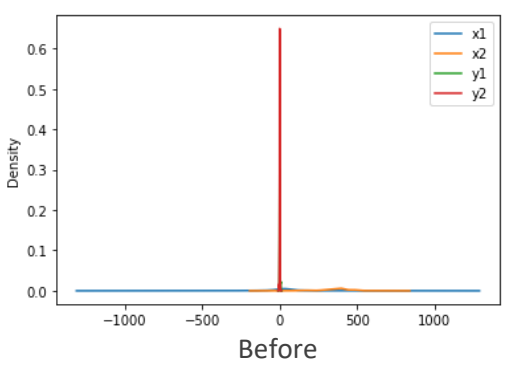
- Limit them to some decimal points for better numeric distribution
- Removed Duplication in input columns of the game for better model fit

DATA PRE-PROCESSING

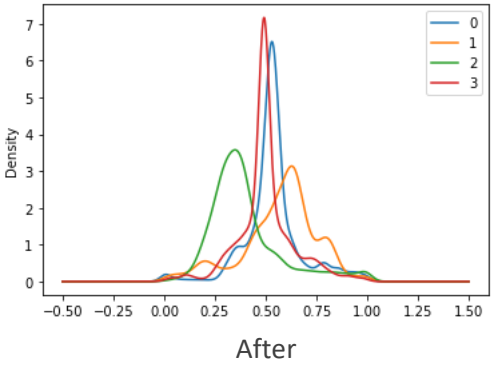
```
x1    9
x2    9
y1    9   <-- count
y2    9
dtype: int64
```

```
# replacing all the zero or empty values with mean of that column
game_data_frame['x1'].fillna((game_data_frame['x1'].mean()), inplace=True)
game_data_frame['x2'].fillna((game_data_frame['x2'].mean()), inplace=True)
game_data_frame['y1'].fillna((game_data_frame['y1'].mean()), inplace=True)
game_data_frame['y2'].fillna((game_data_frame['y2'].mean()), inplace=True)
```

- Replace NA(null) values with the mean of the column.



Before



After

- Data Distribution – Line Graph
- Min-Max Scaler

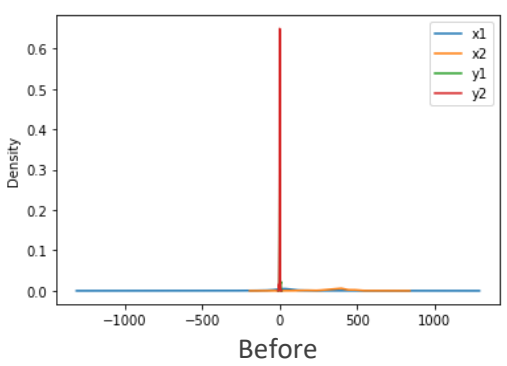
70-30 split between training and validation

DATA PRE-PROCESSING

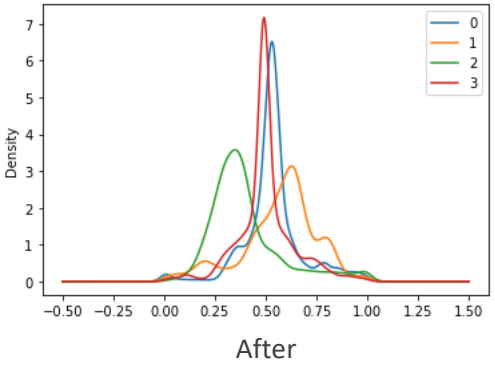
```
x1    9
x2    9
y1    9   <-- count
y2    9
dtype: int64
```

```
# replacing all the zero or empty values with mean of that column
game_data_frame['x1'].fillna((game_data_frame['x1'].mean()), inplace=True)
game_data_frame['x2'].fillna((game_data_frame['x2'].mean()), inplace=True)
game_data_frame['y1'].fillna((game_data_frame['y1'].mean()), inplace=True)
game_data_frame['y2'].fillna((game_data_frame['y2'].mean()), inplace=True)
```

- Replace NA(null) values with the mean of the column.



Before



After

- Data Distribution – Line Graph
- Min-Max Scaler

70-30 split between training and validation

NEURON CLASS

```python
# A neural network class
class Neuron:

    # init method or constructor
    def __init__(self, AV, number_of_weights, index):
        self.AV = AV # setting neuron Activation Value
        # weights list

        # defining 2 more variables for the back propogation
        self.delta_weights = []
        self.grad_val = 0

        self.weights_list = [] # randomizing weights of the neuron according to the no. of weights
        for i in range(0, number_of_weights):
            self.weights_list.append(random.random())
            self.delta_weights.append(0)

        self.index = index # setting neuron index

    def sigmoid(self, x):
        self.AV = 1 / (1 + math.exp(-(our_lambda * x))) #applying the sigmoid function

    def mult_weights(self, prevLayer):
        total_sum = 0
        for i in range(0, len(prevLayer)): # calculating weights by multiplying AV of each neuron with weights and adding
            total_sum = total_sum + (float(prevLayer[i].AV) * float(prevLayer[i].weights_list[self.index]))

        self.sigmoid(total_sum)
```

FEEDFORWARD & BACKWARD

```python
# we sent our actual outputs that we expect our feed forward network to predict
def back_propogation_process(outputs):
    error = []

    # first calculating errors where we subtract the expected output which we pass as parameter to the
    # back propogation and then subtract it with the actual result we get for our output which is stored in Activation value of our layer
    for i in range(0, len(output_layer)):
        error.append(float(outputs[i]) - float(output_layer[i].AV))

    # now calculating gradiant decent for output layer which uses error which we calculated
    for i in range(0, len(output_layer)):
        output_layer[i].grad_val = our_lambda * output_layer[i].AV * (1 - output_layer[i].AV) * error[i]

    # now calculating gradiant decent for out hidden layer where we don't use any error
    for i in range(0, len(hidden_layer)):
        # now in this loop we first multiply the calculated gradiant decent with the weights of the hidden layer first for which we needed the addition result which we add
        result = 0
        for j in range(0, len(output_layer)):
            result = result + ( float(output_layer[j].grad_val) * float(hidden_layer[i].weights_list[j]) )

        hidden_layer[i].grad_val = our_lambda * hidden_layer[i].AV * (1 - hidden_layer[i].AV) * result


    # for now as we have calculated gradiatn decent for both the hidden and output layer so now it's time to calculate weight updations for neurons
    for i in range(0, len(hidden_layer)):
        for j in range(0, len(output_layer)): # We have momentum same as like lambda where we tell it how fast it should move?
            hidden_layer[i].delta_weights[j] = learning_rate * float(output_layer[j].grad_val) * float(hidden_layer[i].AV) + momentum_mt * float(hidden_layer[i].delta_weights[j])

    #  time for us to calculate updated weights for the input layer
    for i in range(0, len(input_layer)):
        for j in range(0, (len(hidden_layer) - 1)): # We have momentum same as like lambda where we tell it how fast it should move?
            input_layer[i].delta_weights[j] = learning_rate * float(hidden_layer[j].grad_val) * float(input_layer[i].AV) + momentum_mt * float(input_layer[i].delta_weights[j])

    # now finally updating weights after calculating new delta weights on the basis of gradiant
    for i in range(0, len(hidden_layer)):
        for j in range(0, len(hidden_layer[i].weights_list)): # We have momentum same as like lambda where we tell it how fast it should move?
            hidden_layer[i].weights_list[j] = float(hidden_layer[i].weights_list[j]) + float(hidden_layer[i].delta_weights[j])

    for i in range(0, len(input_layer)):
        for j in range(0, len(input_layer[i].weights_list)): # We have momentum same as like lambda where we tell it how fast it should move?
            input_layer[i].weights_list[j] = float(input_layer[i].weights_list[j]) + float(input_layer[i].delta_weights[j])

    # Note: we don't calculate any gradiant decent for our input layer

    return
```

```python
def feed_forward_process(inputs):
    # Setting input_layer activation values
    for i in range(0, len(input_layer)):
        input_layer[i].AV = inputs[i]

    # Calculating hidden_layer activation values on the basis of input_layer
    for i in range(0, (len(hidden_layer) - 1)):
        hidden_layer[i].mult_weights(input_layer)

    hidden_layer[-1].AV = 1 #It's our bios which is for 3rd neuron we are setting

    # Calculating output_layer activation values on the basis of hidden_layer
    for i in range(0, len(output_layer)):
        output_layer[i].mult_weights(hidden_layer)

    return
```

NEURAL NETWORK IN GAME

```python
def predict(self, input_row):
    input_row_list = input_row.split(",") # Getting input_row from game as string and convering it to list for prediction
    output_row_list = prediction([float(input_row_list[0]), float(input_row_list[1])]) # predicting it using our model
    return output_row_list # return ing it back to game so that game could make move accordingly.
```

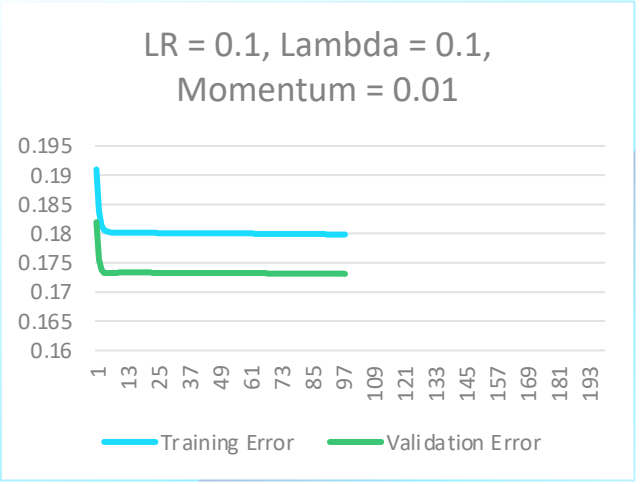
game function

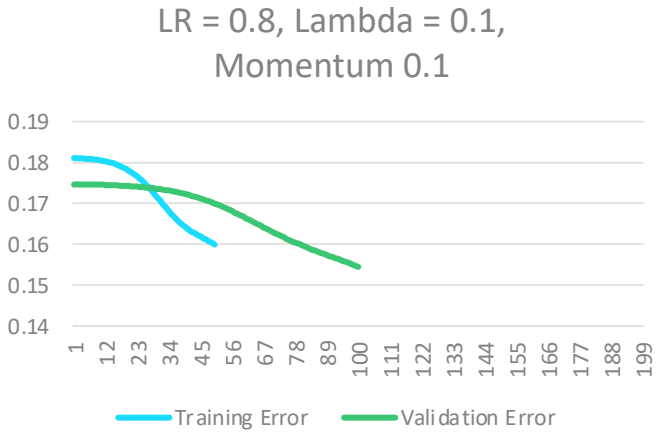
Send input

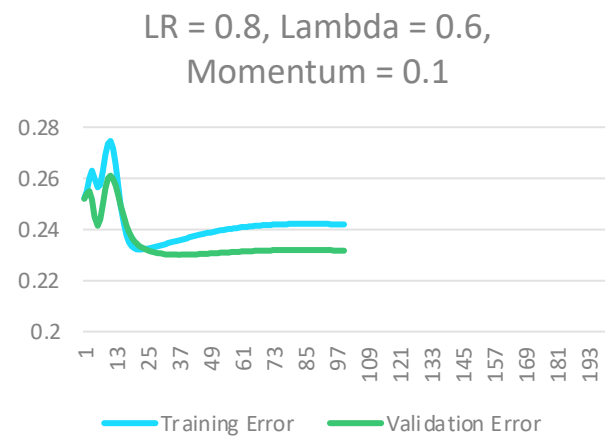
```python
# specifying min and maximum values according to model
x1_max = 477.731
x2_max = 567.535
y1_max = 7.995
y2_max = 4.339

x1_min = -542.953
x2_min = 65.460
y1_min = -3.866
y2_min = -5.147

def normalization(inputs):
    input1 = (inputs[0] - (x1_min)) / ((x1_max) - (x1_min))  # Normalizing first data point from input row
    input2 = (inputs[1] - (x2_min)) / ((x2_max) - (x2_min))  # Normalizing second data point from input row
    return [input1, input2]

def de_normalization(outputs):
    # As we have normalized now need to convert it back to the original state using de_normalization
    return [(outputs[0] * ((y1_max) - (y1_min)) + (y1_min)),(outputs[1] * ((y2_max) - (y2_min)) + (y2_min))]

def prediction(input_row):
    # we predict in this function that what would be the result against one game input
    normalized_result = normalization(input_row) # Normalizing the input row before passing to feed_forward
    normalized_result.append(1) # Appending bias into normalized list of inputs
    print(normalized_result)

    load_weights() # Loading weights

    feed_forward_process(normalized_result) # Checking feed_forward_ with the input row

    print([output_layer[0].AV,output_layer[1].AV])
    return de_normalization([output_layer[0].AV,output_layer[1].AV]) # returning denormalized predicted result
```

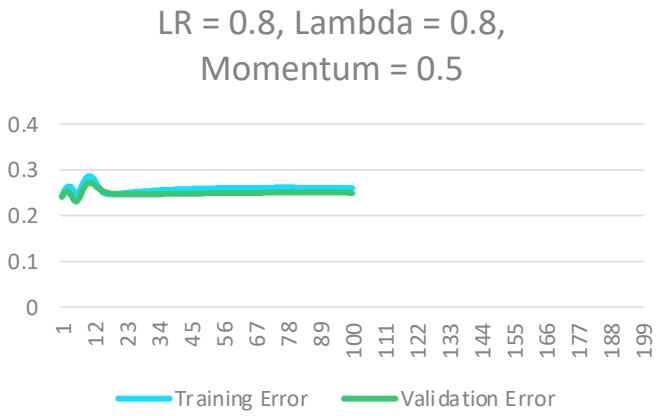
return prediction

HYPERPARAMETERS









STOPPING CRITERIA

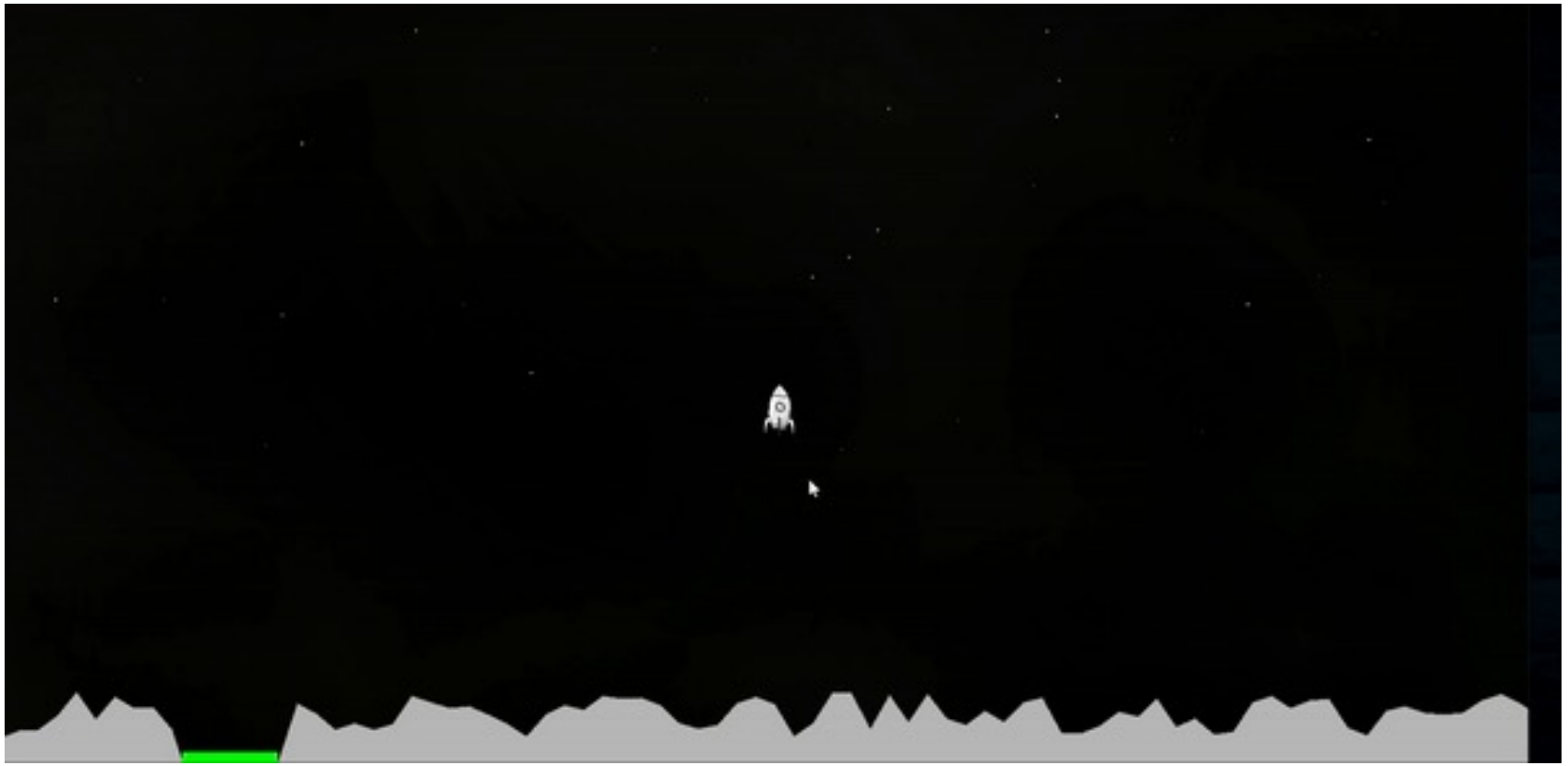
1. Check for every previous and current error up to 5 decimal points for stopping.

```
# Checking if upto 5 decimals we don't see any siginifant change then stopping
if float("{:.5f}".format(error)) == float("{:.5f}".format(stopping_error_check)):
    print("Stopping")
    break
else:
    stopping_error_check = error
```

2. Used number of epochs if there’s no significant change in error found out.

```
for epoch_count in range(2000): # Controlling no. of epochs in case if the model will not in any case
    print('Epoch no. ', epoch_count + 1)
```

NETWORKING PERFORMANCE IN GAME



It follows target no matter where it is but need more dataset for landing.