PREDICTION OF MATERIAL REMOVAL RATE (MRR) IN CHEMICAL MECHANICAL POLISHING PROCESS

Abstract

Developing ML models to predict material removal rate in a chemical polishing process using Neural Networks Approach

David Adeniji

doad224@uky.edu

1.0 Introduction/Project Background

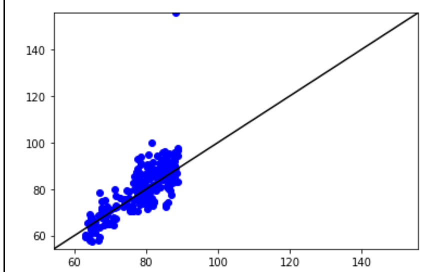
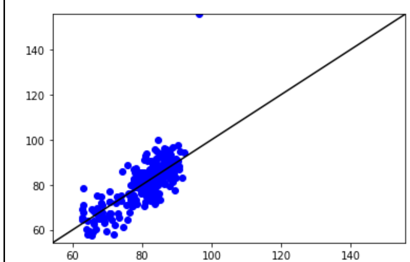
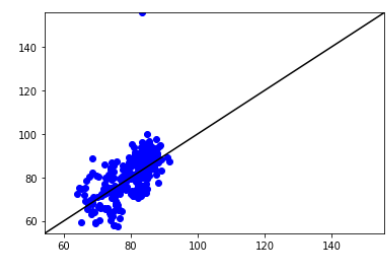
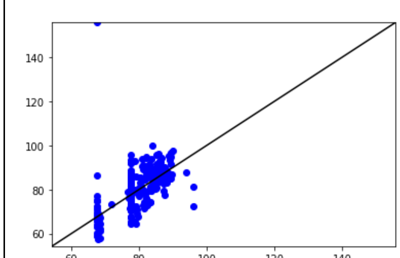
Given two sets of data (Data1 & Data2) with 25 independent variables each, we are to predict the polishing removal rate of material from a wafer for each dataset using neural network machine learning approach. Table 1 below shows the data column structure and description of the variables.

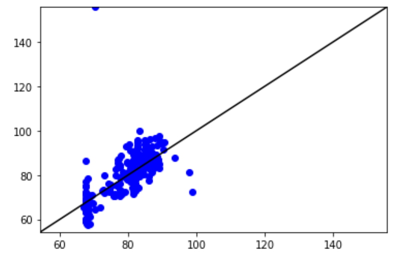
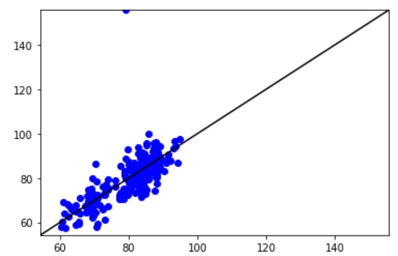
Table 1. Description of data structure and variables

Column Symbol	Column Name	Description
X1	Machine ID	Numeric ID of machine
X2	Machine Data	Numeric ID of wafer ring location in machine
X3	Timestamp	Seconds
X4	Wafer ID	Number representing ID of wafer
X5	Stage	A or B representing a different type of processing stage
X6	Chamber	Chamber in machine for wafer processing
X7	Usage of Backing Film	A usage measure of polish-pad backing film
X8	Usage of Dresser	A usage measure of dresser
X9	Usage of Polishing Table	A usage measure of polishing table
X10	Usage of Dresser Table	A usage measure of dresser table
X11	Pressurized Chamber Pressure	Chamber pressure
X12	Main Outer Air Bag Pressure	Pressure related to wafer placement
X13	Center Air Bag Pressure	Pressure related to wafer placement
X14	Retainer Ring Pressure	Pressure related to wafer placement
X15	Ripple Air Bag Pressure	Pressure related to wafer placement
X16	Usage of Membrane	A usage measure of polishing membrane
X17	Usage of Pressurized Sheet	A usage measure of wafer carrier flexible sheet
X18	Slurry Flow Line A	Flow rate of slurry type A
X19	Slurry Flow Line B	Flow rate of slurry type B
X20	Slurry Flow Line C	Flow rate of slurry type C
X21	Wafer Rotation	Rotation rate of wafer
X22	Stage Rotation	Rotation rate of stage
X23	Head Rotation	Rotation rate of head
X24	Dressing Water Status	Status of dressing water
X25	Edge Air Bag Pressure	Pressure of bag on edge of wafer
Y	***Material Removal Rate***	***Material Removal Rate***

Variables 1,2,3,4,5, are unique machine data and would not affect the material removal rate (MRR). Therefore Columns 1 to 5 are dropped, and only Columns 6 to 25 are used.

2.0 Data Analysis

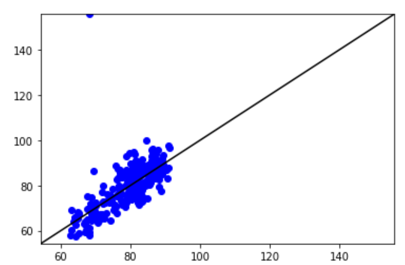
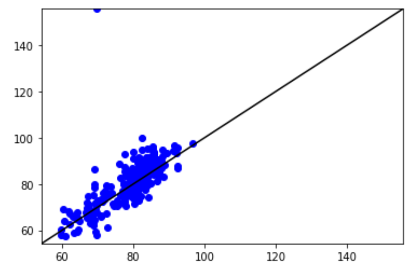
<table>
<tr><th colspan="6">SELF-FORWARD & BACKWARD PROPAGATION</th></tr>
<tr><td colspan="6">Part A (i): Different Input Variables</td></tr>
<tr><td>Variables</td><td>Layers</td><td>[Learning Rate, Epochs]</td><td>R-Squared Value</td><td>RMSE</td><td>Plots</td></tr>
<tr><td>X_6, X_7, X_8, X_9</td><td>(4,9,1)</td><td>$[60000, 0.05]$</td><td>0.559</td><td>7.50</td><td>
</td></tr>
<tr><td>$X_5, X_6, X_7, X_8, X_9, X_{10}$ X_{11}</td><td>(7,10,1)</td><td>$[20000, 0.05]$</td><td>0.5863</td><td>7.43</td><td>
</td></tr>
<tr><td>X_8, X_9, X_{10}, X_{11}</td><td>[4,9,1]</td><td>$[60000, 0.05]$</td><td>0.378</td><td>8.34</td><td>
</td></tr>
<tr><td colspan="6">Findings:
Results shown above shows similar trend to results from project 1, using only variables X_6, X_7, X_8, X_9 gave the best R2 and RMSE performance. Adding more variable to this four reduces the performance, since other variables are not quite relevant as explained in project 1.</td></tr>
<tr><td colspan="6">Part A (ii): Different Number of Hidden Layers</td></tr>
<tr><td>X_6, X_7, X_8, X_9</td><td>(4,2,1)</td><td>$[60000, 0.05]$</td><td>0.342</td><td>8.3</td><td>
</td></tr>
</table>

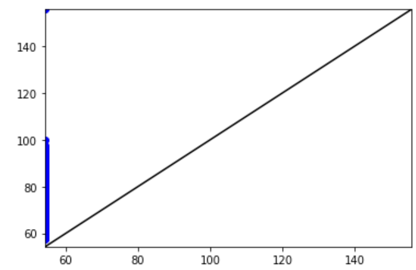
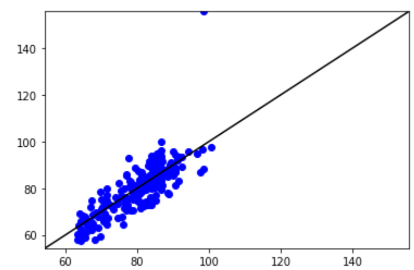
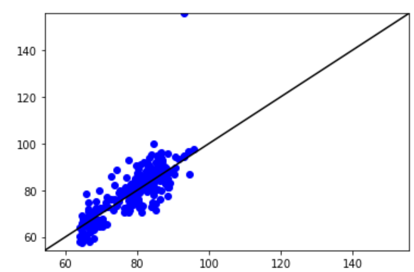
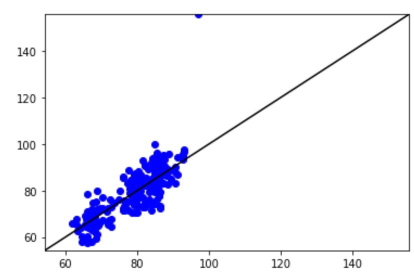
X_6, X_7, X_8, X_9	(4,3,1)	$[60000, 0.05]$	0.428	8.15	
X_6, X_7, X_8, X_9	(4,13,1)	$[60000, 0.05]$	0.536	7.31	

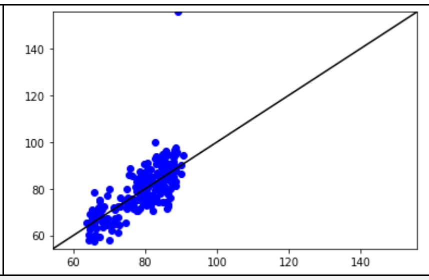
Findings:

As shown in Part A(ii) above, increasing the hidden neuron layer and maintaining the selected variables, epochs and learning rates, improves the R2 and RMSE performance significantly.

Part A (iii): Different Learning Rates

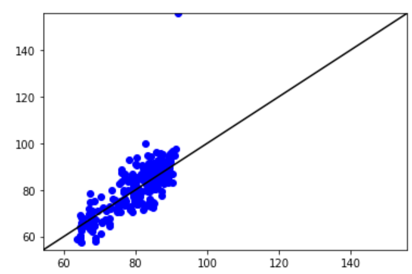
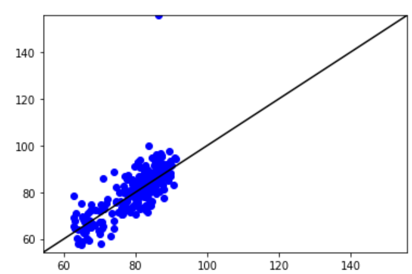
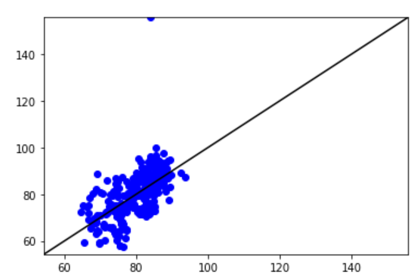
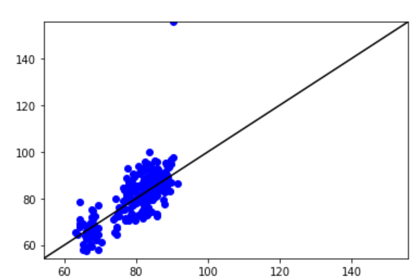
Variables	*Layers*	*[Learning Rate, Epochs]*	*R-Squared Value*	**RMSE**	**Plots**
X_6, X_7, X_8, X_9	(4,9,1)	$[60000, 0.01]$	0.433	7.96	
X_6, X_7, X_8, X_9	(4,9,1)	$[60000, 0.1]$	0.475	7.65	

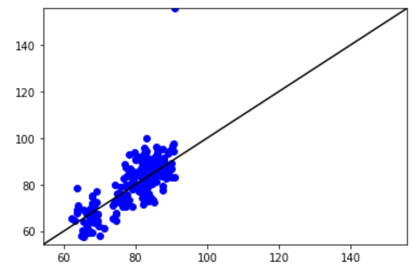
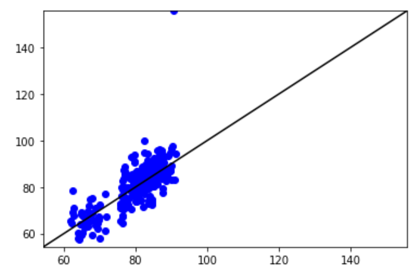
X_6, X_7, X_8, X_9	(4,9,1)	$[60000, 0.5]$	-6.07	28.11	
Findings: As shown in Part A(iii) above, increasing the learning rates, while maintaining the selected variables, number of epochs, hidden neuron layer and learning rates, improves the R2 and RMSE performance significantly. *However, it was found that at learning rate of 0.5 and above, the prediction and neural network performance is very poor.*					
Part D: Extra Hidden Layer					
$X_6, X_7, X_8, X_9, X_{10}, X_{11}$	(6,8,8,1)	$[100000, 0.03]$	0.662	6.14	
$X_6, X_7, X_8, X_9, X_{10}, X_{11}$	(6,8,8,1)	$[60000, 0.03]$	0.606	6.63	
$X_6, X_7, X_8, X_9, X_{10}, X_{11}$	(6,8,8,1)	$[100000, 0.01]$	0.629	6.43	

$X_6, X_7, X_8, X_9, X_{10}, X_{11}$	(6,8,8,1)	[60000, 0.01]	0.56	7.05	

Findings:

Similar trend in changes were found after adding and extra hidden neuron layer. Also, the performance also improved compared to not having the hidden layer.

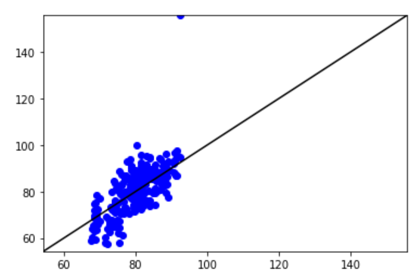
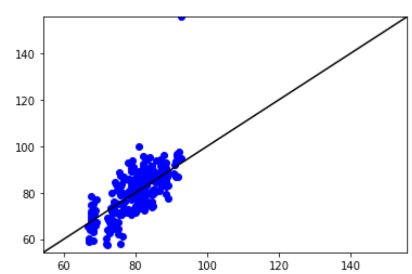
TORCH.nn					
Variables	***Layers***	***[Epochs, Learning Rate]***	***R-Squared Value***	**RMSE**	**Plots**
Part B (i): Different Input Variables					
X_6, X_7, X_8, X_9	(4,9,1)	[60000, 5]	0.596	6.72	
$X_5, X_6, X_7, X_8, X_9, X_{10}$ X_{11}	(7,10,1)	[60000, 5]	0.560	7.01	
X_8, X_9, X_{10}, X_{11}	[4,9,1]	[60000, 5]	0.384	8.31	
Findings: Results in Part B (i) above shows similar trend to results from Part A (i), using only variables X_6, X_7, X_8, X_9 gave the best R2 and RMSE performance. Adding more variable to this four reduces the performance, since other variables are not quite relevant as explained above. Using only two out of these four variables gave the worst performance in combinations tried above.					
Part B (ii): Different Hidden Layer Neurons					
X_5, X_6, X_7, X_8	(4,2,1)	[60000, 5]	0.547	7.12	

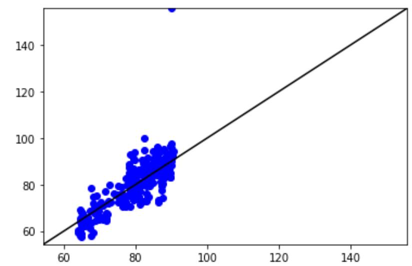
Variables	***Layers***	***[Epochs, Learning Rate]***	***R-Squared Value***	**RMSE**	**Plots**
X_5, X_6, X_7, X_8	(4,3,1)	$[60000, 5]$	0.548	7.12	
X_6, X_7, X_8, X_9	(4,13,1)	$[60000, 5]$	0.562	6.99	

Findings:

As shown in Part B(ii) above, increasing the hidden neuron layer and maintaining the selected variables, epochs and learning rates, improves the R2 and RMSE performance significantly.

Part B (iii): Different Learning Rates

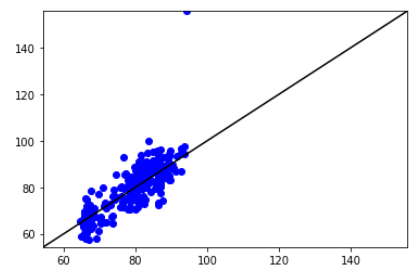
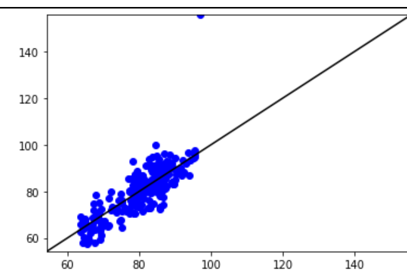
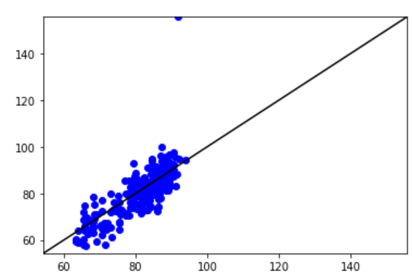
Variables	***Layers***	***[Learning Rate, Epochs]***	***R-Squared Value***	**RMSE**	**Plots**
X_6, X_7, X_8, X_9	(4,9,1)	$[60000, 0.5]$	0.494	7.52	
X_6, X_7, X_8, X_9	(4,9,1)	$[60000, 1]$	0.502	7.46	

Variables	***Layers***	***[Epochs, Learning Rate]***	***R-Squared Value***	**RMSE**	**Plots**
X_6, X_7, X_8, X_9	(4,9,1)	$[60000, 10]$	0.592	6.75	

Findings:

As shown in Part B(iii) above, increasing the learning rates, while maintaining the selected variables, number of epochs, hidden neuron layer and learning rates, improves the R2 and RMSE performance significantly.

Part C: Different Torch Network Structure

X_6, X_7, X_8, X_9	(4,9,4,1)	$[60000, 10]$	0.61	6.60	
X_6, X_7, X_8, X_9	(4,9,9,1)	$[60000, 10]$	0.645	6.30	
X_6, X_7, X_8, X_9	(4,9,9,1,1)	$[60000, 10]$	0.611	6.59	

Findings:

As shown in Part C above, increasing the number of layers, while maintaining other parameters, improves the R2 and RMSE performance significantly.

3.0 Conclusion

Overall, the analysis/findings above show that neural network parameters such as input variables, learning rates, number of epochs, hidden neuron layers significantly influence the network performance and it would take a proper combination of these parameter to give the best performance.

4.0 Appendix

SELF-CODED FORWARD AND BACKPROPAGATION (3-LAYERS)

In [1]:
```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
np.random.seed(1)
```

In [2]:
```
Data = pd.read_excel('data2.xlsx',header=None)
Data = Data.dropna(axis='columns')
X = Data.iloc[1:,8:12]
Y = Data.iloc[1:,[25]]
```

In [3]:
```
X = X.to_numpy() #convert data frame to numpy array
Y = Y.to_numpy()
```

In [4]:
```
# data normalization, normalization to [0 1] range
X_Norm = np.empty_like(X)
for i in range(X.shape[1]):
    data_ = X[:,i]
    X_Norm[:,i] = (data_-np.amin(data_))/(np.amax(data_)-np.amin(data_))

# normalize Y data
Y_Min = np.amin(Y)
Y_Max = np.amax(Y)
Y_Norm = (Y-Y_Min)/(Y_Max-Y_Min)
```

In [5]:
```
# prepare variables and target

index = np.arange(len(Y))
np.random.shuffle(index) #disorder the original data

m = np.ceil(0.7*len(Y)) # 70% for training and 30% for testing
m = int(m) #covert float type to int type
X_Train = X_Norm[index[:m]]
Y_Train = Y_Norm[index[:m]].squeeze()

X_Test = X_Norm[index[m:]]
Y_Test = Y_Norm[index[m:]].squeeze()

#print(X_Train.shape, Y_Train.shape)
#print((Y_Test* (Y_Max - Y_Min) + Y_Min))
```

In [6]:
```
print(Y_Test.shape)
```

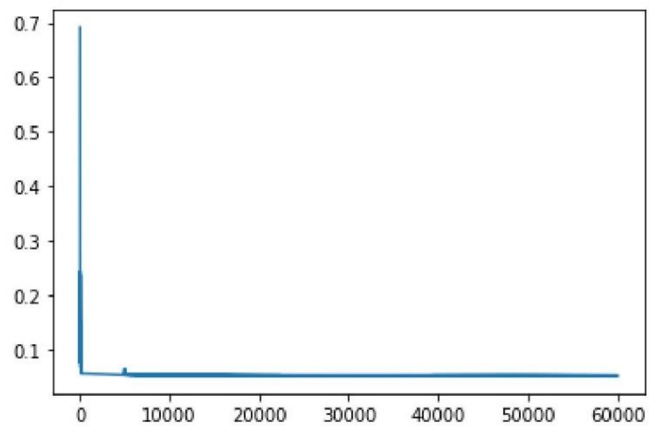
```
(243,)
```

In [7]:
```
# define sigmoid function and sigmoid derivative
def sigmoid(x):
    y = 1/(1+np.exp(-x))
    return y
def sigmoid_derivative(y):
    return y*(1-y)
```

In [8]:
```
class NeuralNetwork():
    def __init__(self, x, y, layer_numbers, learning_rate, epochs): #layer_num
bers = [4, 2*4+1=9,1]
        self.input = x
        self.y = y
        self.layer_numbers = layer_numbers
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.Weights0 = np.random.rand(self.layer_numbers[0],self.layer_number
s[1]) #W0: 4*9
        self.Weights1 = np.random.rand(self.layer_numbers[1],self.layer_number
s[2]) #W1: 9*1
        self.epoch = []
        self.error_history = []

    def forward(self):
        self.hidden_output = sigmoid(np.dot(self.input,self.Weights0))   #calc
ulate the hidden neuron values, np.dot vector based multiplication
        self.output = sigmoid(np.dot(self.hidden_output,self.Weights1))  #calc
ulate the output neuron values

    def backpropagation(self):
        self.error = np.average(np.abs(self.y-self.output)) #sum(|Yactual-Y|)/
No.(Y) 100*1
        d_Weights1 = np.dot(self.hidden_output.T,(self.y-self.output)*sigmoid_
derivative(self.output))             #gradient for W1: H'*(Yactual-Y)*sigmoi
d'(Y)
        layer_error1 = np.dot((self.y-self.output)*sigmoid_derivative(self.out
put), self.Weights1.T) #partile derivative w.r.t H (Yactual)*sigmoid'(Y)*W1'
        d_Weights0 = np.dot(self.input.T,layer_error1*sigmoid_derivative(self.
hidden_output)) #gradient for W0  dJ/dW0 = (dJ/dY)(dY/dH)(dH/dW0) = X'*layer_e
rror1*sigmoid'(H)

        self.Weights0 = self.Weights0 + self.learning_rate*d_Weights0 #update
 W0
        self.Weights1 = self.Weights1 + self.learning_rate*d_Weights1 #update
 W1

    def train(self):
        for epoch in range(self.epochs):
            self.forward()
            self.backpropagation()
            self.epoch.append(epoch) #np.arrange(epochs)
            self.error_history.append(self.error)

    def predict(self,new_data):
        hidden_output = sigmoid(np.dot(new_data,self.Weights0))   #calculate t
he hidden neuron values, np.dot vector based multiplication
        output = sigmoid(np.dot(hidden_output,self.Weights1))
        return output
```

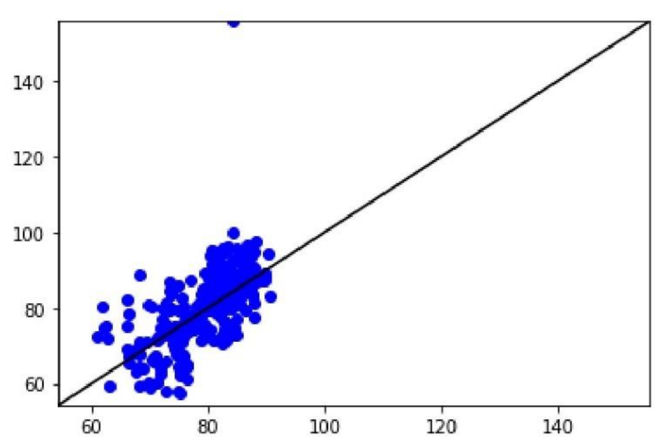
In [9]:
```
layer_numbers = [4,9,1]
learning_rate =0.05
epochs = 60000
Y_Train = np.reshape(Y_Train,(len(Y_Train),1))
Net = NeuralNetwork(X_Train, Y_Train, layer_numbers, learning_rate, epochs) #d
efine an object belonging to the class
Net.train()
plt.figure()
plt.plot(Net.epoch, Net.error_history)
```

Out[9]: [<matplotlib.lines.Line2D at 0x2cf37686cf8>]



In [10]:
```
# testing
y_predict = Net.predict(X_Test)
y_predicted = y_predict * (Y_Max - Y_Min) + Y_Min
Y_Test = Y_Test * (Y_Max - Y_Min) + Y_Min
Y_Test = Y_Test.reshape(len(Y_Test),1)
plt.scatter(y_predicted, Y_Test, c = 'b',marker = 'o')
plt.xlim(Y_Min, Y_Max)
plt.ylim(Y_Min, Y_Max)
plt.plot([Y_Min, Y_Max],[Y_Min, Y_Max],'k-')
```

Out[10]: [<matplotlib.lines.Line2D at 0x2cf376f0a58>]



In [12]:
```
def rmse(y_predicted,y):
    ssr = np.sum((y_predicted-y)**2)
    rmse= (ssr/len(y))**0.5
    return(rmse)
print(rmse(y_predicted, Y_Test),r2(y_predicted, Y_Test))
```

```
8.34031316808369 0.37786233357025856
```

In [11]:
```
def r2(y_predicted,y):
    sst = np.sum((y-y.mean())**2)
    ssr = np.sum((y_predicted-y)**2)
    r2 = 1-(ssr/sst)
    return(r2)
r2(y_predicted, Y_Test)
```

Name: David Adeniji *Project 3* *ME 699, Spring 2020*

SELF-CODED FORWARD AND BACKPROPAGATION (4-LAYERS)

In [1]:
```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
np.random.seed(1)
```

In [2]:
```
Data = pd.read_excel('data2.xlsx',header=None)
Data = Data.dropna(axis='columns')
X = Data.iloc[1:,6:12]
Y = Data.iloc[1:,[25]]
```

In [3]:
```
X = X.to_numpy() #convert data frame to numpy array
Y = Y.to_numpy()
```

In [4]:
```
# data normalization, normalization to [0 1] range
X_Norm = np.empty_like(X)
for i in range(X.shape[1]):
    data_ = X[:,i]
    X_Norm[:,i] = (data_-np.amin(data_))/(np.amax(data_)-np.amin(data_))

# normalize Y data
Y_Min = np.amin(Y)
Y_Max = np.amax(Y)
Y_Norm = (Y-Y_Min)/(Y_Max-Y_Min)
```

In [13]:
```
# prepare variables and target

index = np.arange(len(Y))
np.random.shuffle(index) #disorder the original data

m = np.ceil(0.7*len(Y)) # 70% for training and 30% for testing
m = int(m) #covert float type to int type
X_Train = X_Norm[index[:m]]
Y_Train = Y_Norm[index[:m]].squeeze()

X_Test = X_Norm[index[m:]]
Y_Test = Y_Norm[index[m:]].squeeze()

#print(X_Train.shape, Y_Train.shape)
#print((Y_Test* (Y_Max - Y_Min) + Y_Min))
```

In [6]:
```
print(Y_Test.shape)
```

```
(243,)
```

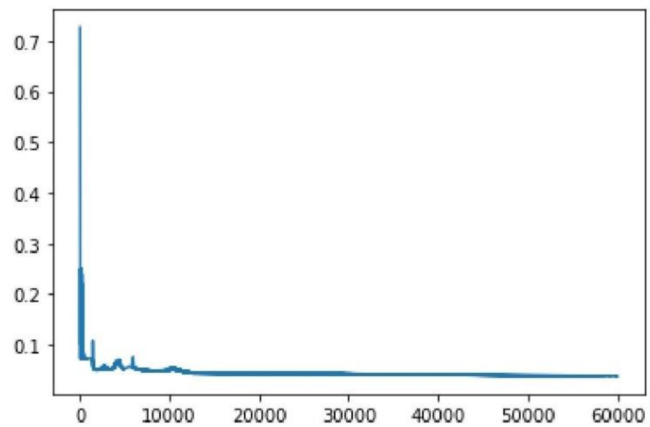
In [7]:
```
# define sigmoid function and sigmoid derivative
def sigmoid(x):
    y = 1/(1+np.exp(-x))
    return y
def sigmoid_derivative(y):
    return y*(1-y)
```

In [8]:
```
class NeuralNetwork():
    def __init__(self, x, y, layer_numbers, learning_rate, epochs): #layer_num
bers = [4, 2*4+1=9,1]
        self.input = x
        self.y = y
        self.layer_numbers = layer_numbers
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.Weights0 = np.random.rand(self.layer_numbers[0],self.layer_number
s[1]) #W0: 4*9
        self.Weights1 = np.random.rand(self.layer_numbers[1],self.layer_number
s[2]) #W1: 9*1
        self.Weights2 = np.random.rand(self.layer_numbers[2],self.layer_number
s[3])
        self.epoch = []
        self.error_history = []

    def forward(self):
        self.hidden_output = sigmoid(np.dot(self.input,self.Weights0)) #calcul
ate the hidden neuron values, np.dot vector based multiplication
        self.hidden_output2 = sigmoid(np.dot(self.hidden_output,self.Weights1
))
        self.output = sigmoid(np.dot(self.hidden_output2,self.Weights2)) #calc
ulate the output neuron values

    def backpropagation(self):
        self.error = np.average(np.abs(self.y-self.output)) #sum(|Yactual-Y|)/
No.(Y) 100*1
        d_Weights2 = np.dot(self.hidden_output2.T,(self.y-self.output)*sigmoid
_derivative(self.output)) #gradient for W1: H'*(Yactual-Y)*sigmoid'(Y)
        layer_error2 = np.dot((self.y-self.output)*sigmoid_derivative(self.out
put), self.Weights2.T) #partile derivative w.r.t H (Yactual)*sigmoid'(Y)*W1'
        d_Weights1 = np.dot(self.hidden_output.T,layer_error2*sigmoid_derivati
ve(self.hidden_output2)) #gradient for W0 dJ/dW0 = (dJ/dY)(dY/dH)(dH/dW0) =
 X'*layer_error1*sigmoid'(H)
        layer_error1 = np.dot((self.y-self.output)*sigmoid_derivative(self.hid
den_output2), self.Weights1.T) #partile derivative w.r.t H (Yactual)*sigmoi
d'(Y)*W1'
        d_Weights0 = np.dot(self.input.T,layer_error1*sigmoid_derivative(self.
hidden_output))

        self.Weights0 = self.Weights0 + self.learning_rate*d_Weights0 #update
 W0
        self.Weights1 = self.Weights1 + self.learning_rate*d_Weights1 #update
 W1
        self.Weights2 = self.Weights2 + self.learning_rate*d_Weights2

    def train(self):
        for epoch in range(self.epochs):
            self.forward()
            self.backpropagation()
            self.epoch.append(epoch) #np.arrange(epochs)
            self.error_history.append(self.error)

    def predict(self,new_data):
        hidden_output = sigmoid(np.dot(new_data,self.Weights0)) #calculate the
```

```
hidden neuron values, np.dot vector based multiplication
        hidden_output2 = sigmoid(np.dot(hidden_output,self.Weights1))
        output = sigmoid(np.dot(hidden_output2,self.Weights2))
        return output
```

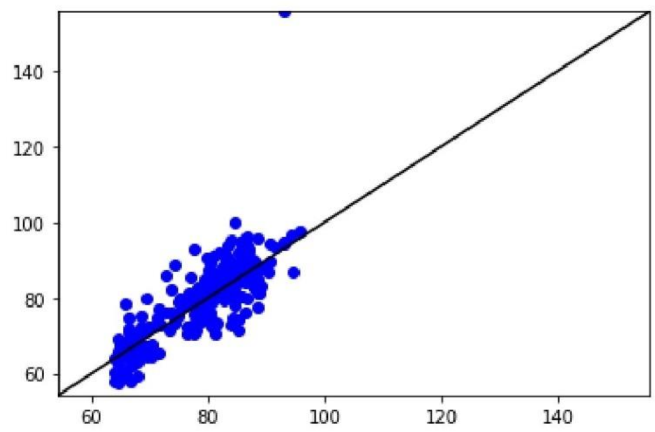
In [9]:
```
layer_numbers = [6,8,8,1]
learning_rate =0.03
epochs = 60000
Y_Train = np.reshape(Y_Train,(len(Y_Train),1))
Net = NeuralNetwork(X_Train, Y_Train, layer_numbers, learning_rate, epochs) #d
efine an object belonging to the class
Net.train()
plt.figure()
plt.plot(Net.epoch, Net.error_history)
```

Out[9]:
```
[<matplotlib.lines.Line2D at 0x1f6b7dcda20>]
```



In [10]:
```
# testing
y_predict = Net.predict(X_Test)
y_predicted = y_predict * (Y_Max - Y_Min) + Y_Min
Y_Test = Y_Test * (Y_Max - Y_Min) + Y_Min
Y_Test = Y_Test.reshape(len(Y_Test),1)
plt.scatter(y_predicted, Y_Test, c = 'b',marker = 'o')
plt.xlim(Y_Min, Y_Max)
plt.ylim(Y_Min, Y_Max)
plt.plot([Y_Min, Y_Max],[Y_Min, Y_Max],'k-')
```

Out[10]:
```
[<matplotlib.lines.Line2D at 0x1f6b76f1b00>]
```



In [12]:
```
def rmse(y_predicted,y):
    ssr = np.sum((y_predicted-y)**2)
    rmse= (ssr/len(y))**0.5
    return(rmse)
print(rmse(y_predicted, Y_Test),r2(y_predicted, Y_Test))
```

```
6.635551172273885 0.6061999335904353
```

In [11]:
```
def r2(y_predicted,y):
    sst = np.sum((y-y.mean())**2)
    ssr = np.sum((y_predicted-y)**2)
    r2 = 1-(ssr/sst)
    return(r2)
r2(y_predicted, Y_Test)
```

Name: David Adeniji *Project 3* *ME 699, Spring 2020*

TORCH.NN (3-LAYERS)

In [1]:
```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
import torch
np.random.seed(1)
torch.random.manual_seed(1)
```

Out[1]: `<torch._C.Generator at 0x13df1735bf0>`

In [2]:
```
Data = pd.read_excel('data2.xlsx',header=None)
Data = Data.dropna(axis='columns')
X = Data.iloc[1:,6:10]
Y = Data.iloc[1:,[25]]
```

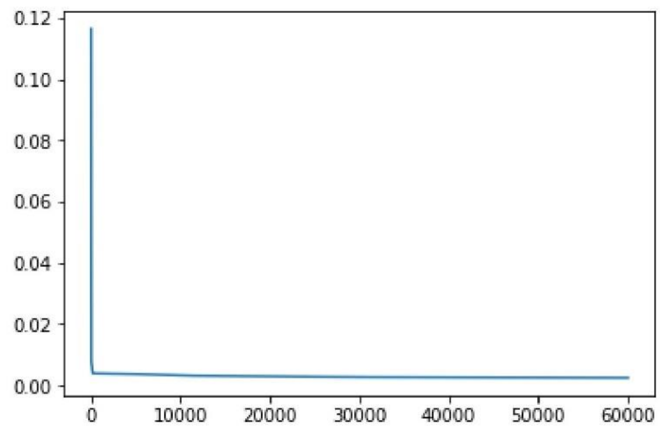
In [3]:
```
X = X.to_numpy() #convert data frame to numpy array
Y = Y.to_numpy()
```

In [4]:
```
# data normalization, normalization to [0 1] range
X_Norm = np.empty_like(X)
for i in range(X.shape[1]):
    data_ = X[:,i]
    X_Norm[:,i] = (data_-np.amin(data_))/(np.amax(data_)-np.amin(data_))

# normalize Y data
Y_Min = np.amin(Y)
Y_Max = np.amax(Y)
Y_Norm = (Y-Y_Min)/(Y_Max-Y_Min)
Y_Norm = Y_Norm.reshape(len(Y_Norm),1)
```

In [5]:
```
# prepare variables and target

index = np.arange(len(Y))
np.random.shuffle(index) #disorder the original data

m = np.ceil(0.7*len(Y)) # 70% for training and 30% for testing
m = int(m) #covert float type to int type
X_Train = X_Norm[index[:m]]
Y_Train = Y_Norm[index[:m]]

X_Test = X_Norm[index[m:]]
Y_Test = Y_Norm[index[m:]]
```

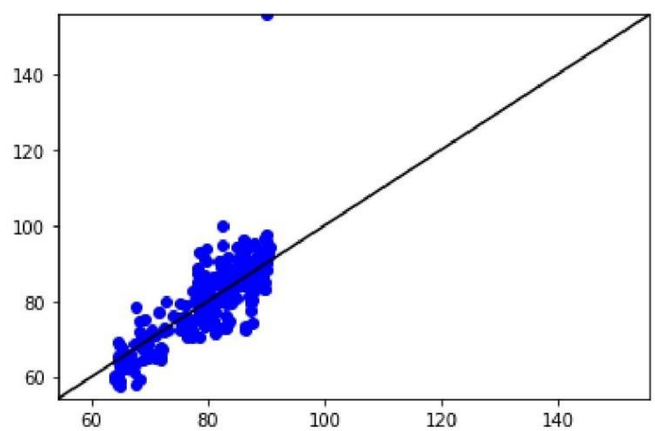
In [6]:
```
# convert numpy array to torch tensor
X_Train_Tensor = torch.tensor(X_Train).float()
X_Test_Tensor = torch.tensor(X_Test).float()
Y_Train_Tensor = torch.tensor(Y_Train).float()
Y_Test_Tensor = torch.tensor(Y_Test).float()
```

Out[7]: [<matplotlib.lines.Line2D at 0x13dfa129080>]



In [8]:
```
# testing
y_predict = net(X_Test_Tensor)
y_predicted = y_predict.detach() * (Y_Max - Y_Min) + Y_Min
Y_Test = Y_Test_Tensor * (Y_Max - Y_Min) + Y_Min
plt.scatter(y_predicted, Y_Test, c = 'b',marker = 'o')
plt.xlim(Y_Min, Y_Max)
plt.ylim(Y_Min, Y_Max)
plt.plot([Y_Min, Y_Max],[Y_Min, Y_Max],'k-')
```

Out[8]: [<matplotlib.lines.Line2D at 0x13df24b5278>]



In [10]:
```
def rmse(y_predicted,y):
    ssr = np.sum((y_predicted-y)**2)
    rmse= (ssr/len(y))**0.5
    return(rmse)
print(rmse(y_predicted.numpy(), Y_Test.numpy()),r2(y_predicted.numpy(), Y_Test
.numpy()))
```

6.752090630364675 0.5922459959983826

In [7]:
```
class Net(torch.nn.Module):
    def __init__(self, layer_numbers):
        super().__init__()
        self.hidden = torch.nn.Linear(layer_numbers[0],layer_numbers[1],bias =
False)
        self.output = torch.nn.Linear(layer_numbers[1],layer_numbers[2],bias =
False)
        self.sigmoid = torch.nn.Sigmoid()

    def forward(self, x):
        x = self.hidden(x)
        x = self.sigmoid(x)
        x = self.output(x)
        x = self.sigmoid(x)
        return x

layer_numbers = [4,9,1]
epochs = 60000
net = Net(layer_numbers)

criterion = torch.nn.MSELoss()
optimizer = torch.optim.SGD(net.parameters(), lr = 10)
loss_history = np.zeros(epochs)

for epoch in range(epochs):
    #forward process
    Y_pred = net(X_Train_Tensor)

    #loss
    loss = criterion(Y_pred,Y_Train_Tensor)
    #calculate gradients in backpropagation
    optimizer.zero_grad()
    loss.backward()
    #update weights
    optimizer.step()

    loss_history[epoch] = loss

plt.plot(np.arange(epochs),loss_history)
```

In [11]:
```
def r2(y_predicted,y):
    sst = np.sum((y-y.mean())**2)
    ssr = np.sum((y_predicted-y)**2)
    r2 = 1-(ssr/sst)
    return(r2)
r2(y_predicted, Y_Test)
```

Name: David Adeniji *Project 3* *ME 699, Spring 2020*

TORCH.NN (4-LAYERS)

In [1]:
```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
import torch
np.random.seed(1)
torch.random.manual_seed(1)
```

Out[1]: `<torch._C.Generator at 0x158dd026bf0>`

In [2]:
```
Data = pd.read_excel('data2.xlsx',header=None)
Data = Data.dropna(axis='columns')
X = Data.iloc[1:,6:10]
Y = Data.iloc[1:,[25]]
```

In [3]:
```
X = X.to_numpy() #convert data frame to numpy array
Y = Y.to_numpy()
```

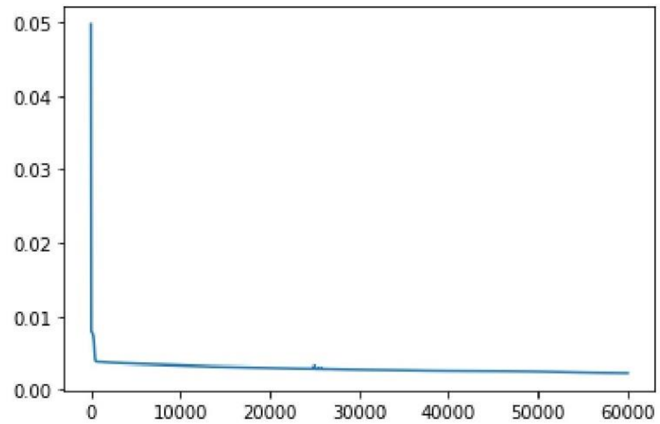
In [4]:
```
# data normalization, normalization to [0 1] range
X_Norm = np.empty_like(X)
for i in range(X.shape[1]):
    data_ = X[:,i]
    X_Norm[:,i] = (data_-np.amin(data_))/(np.amax(data_)-np.amin(data_))

# normalize Y data
Y_Min = np.amin(Y)
Y_Max = np.amax(Y)
Y_Norm = (Y-Y_Min)/(Y_Max-Y_Min)
Y_Norm = Y_Norm.reshape(len(Y_Norm),1)
```

In [5]:
```
# prepare variables and target

index = np.arange(len(Y))
np.random.shuffle(index) #disorder the original data

m = np.ceil(0.7*len(Y)) # 70% for training and 30% for testing
m = int(m) #covert float type to int type
X_Train = X_Norm[index[:m]]
Y_Train = Y_Norm[index[:m]]

X_Test = X_Norm[index[m:]]
Y_Test = Y_Norm[index[m:]]
```

In [6]:
```
# convert numpy array to torch tensor
X_Train_Tensor = torch.tensor(X_Train).float()
X_Test_Tensor = torch.tensor(X_Test).float()
Y_Train_Tensor = torch.tensor(Y_Train).float()
Y_Test_Tensor = torch.tensor(Y_Test).float()
```

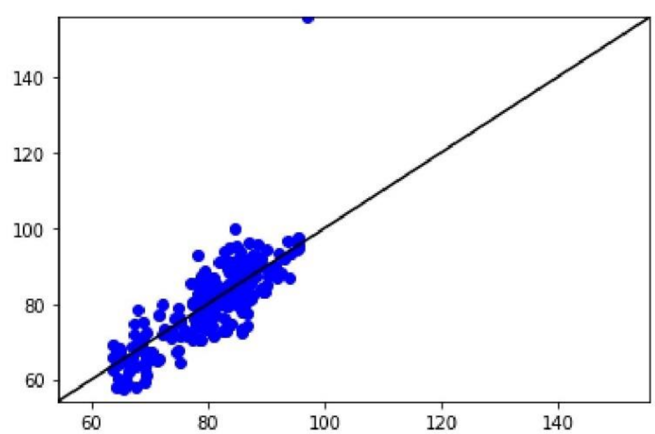
In [7]:
```
class Net(torch.nn.Module):
    def __init__(self, layer_numbers):
        super().__init__()
        self.hidden = torch.nn.Linear(layer_numbers[0],layer_numbers[1],bias =
False)
        self.hidden2 = torch.nn.Linear(layer_numbers[1],layer_numbers[2],bias
= False)
        self.output = torch.nn.Linear(layer_numbers[2],layer_numbers[3],bias =
False)
        self.sigmoid = torch.nn.Sigmoid()

    def forward(self, x):
        x = self.hidden(x)
        x = self.sigmoid(x)
        x = self.hidden2(x)
        x = self.sigmoid(x)
        x = self.output(x)
        x = self.sigmoid(x)
        return x

layer_numbers = [4,9,9,1]
epochs = 60000
net = Net(layer_numbers)

criterion = torch.nn.MSELoss()
optimizer = torch.optim.SGD(net.parameters(), lr = 10)
loss_history = np.zeros(epochs)

for epoch in range(epochs):
    #forward process
    Y_pred = net(X_Train_Tensor)

    #loss
    loss = criterion(Y_pred,Y_Train_Tensor)
    #calculate gradients in backpropagation
    optimizer.zero_grad()
    loss.backward()
    #update weights
    optimizer.step()

    loss_history[epoch] = loss

plt.plot(np.arange(epochs),loss_history)
```

Out[7]: [<matplotlib.lines.Line2D at 0x158df8208d0>]



In [8]:
```
# testing
y_predict = net(X_Test_Tensor)
y_predicted = y_predict.detach() * (Y_Max - Y_Min) + Y_Min
Y_Test = Y_Test_Tensor * (Y_Max - Y_Min) + Y_Min
plt.scatter(y_predicted, Y_Test, c = 'b',marker = 'o')
plt.xlim(Y_Min, Y_Max)
plt.ylim(Y_Min, Y_Max)
plt.plot([Y_Min, Y_Max],[Y_Min, Y_Max],'k-')
```

Out[8]: [<matplotlib.lines.Line2D at 0x158df6a6da0>]



In [9]:
```
def r2(y_predicted,y):
    sst = np.sum((y-y.mean())**2)
    ssr = np.sum((y_predicted-y)**2)
    r2 = 1-(ssr/sst)
    return(r2)
r2(y_predicted.numpy(), Y_Test.numpy())
```

Out[9]: 0.6447165012359619

In [10]:
```
def rmse(y_predicted,y):
    ssr = np.sum((y_predicted-y)**2)
    rmse= (ssr/len(y))**0.5
    return(rmse)
print(rmse(y_predicted.numpy(), Y_Test.numpy()),r2(y_predicted.numpy(), Y_Test
.numpy()))
```

```
6.302700268241883 0.6447165012359619
```