Using Genetic algorithms to generate weights for Neural Networks

Evans Wahome Gichuki
6677196, eg00850@surrey.ac.uk

Abstract

A sort of algorithm that aims to maximise/minimize a function is known as a genetic algorithm. We explain, show, and explore genetic algorithms and how they can be applied with a Multilayer Perceptron in this study. We'll go over what makes up genetic algorithms and how they're created to address a specific minimization problem. We want to construct weights that will be input to a self-declared neural network and evaluate how each combination of weights affects the model's output using Python's DEAP package.

Introduction

A Feedforward neural network, also known as Multilayer perceptrons, comprises weights, perceptrons and biases. And is one of the most common among deep learning models. This type of model is called Feedforward because it consists of several simple perceptron-like processing units (called neurons), organized in layers. Every neuron in a layer is connected with all the neurons in the next layer. This continues until we reach the output layer which then returns the model output.

A feedforward network aims to approximate some function f(x). In this case the function to be approximated is as below

$$y = \sin(3.5\, x_1 + 1.0)\cos(5.5\, x_2),$$

$$x_1, x_2 \in [-1, 1]$$

Where the values of x1 and x2 were to be between the ranges of -1 to 1 as shown above During training, the classifier f(x1,x2) maps the inputs x1 and x2 to the y class. This is done by modifying some parameters known as weights. The best weights would be those that would produce the most accurate results which would mean we have the best function approximation.

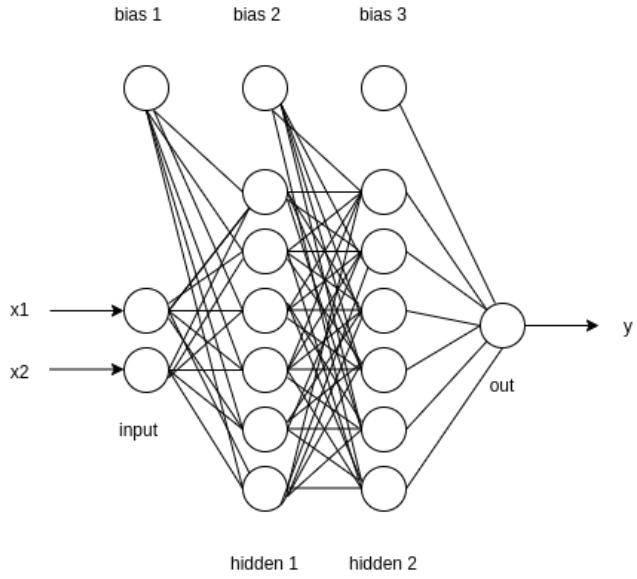

Fig 1.1: neural network architecture

The leftmost layer is called the input layer. Neurons located in this layer are known as the input neurons. These input neurons would be the same number as the number columns to be input as training data i.e x1 and x2. The rightmost layer is also known as the output layer. Neurons on this layer are known as output neurons. The number of these neurons depends on the number of classes to be predicted (incase of classification). In this case, we had one output neuron as we were predicting one continuous float value.

The input variables x1 and x2 are multiplied by weights and later summed together as shown below. The bias is then added to the result. The bias is used to increase the flexibility of the model.

$$v_k = \sum_{j=1}^{n} W_{kj} X_j$$

After the summation is done the result is then passed to an activation function that squashes the value to a fixed range output. In this case we used a sigmoid activation for the hidden layers. SImilar to the one shown below.

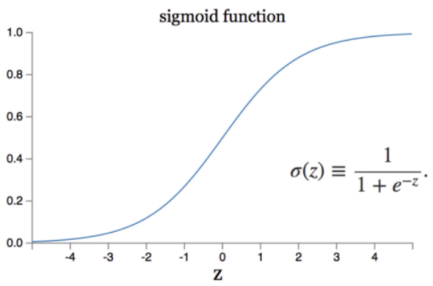

Fig 1.2 SIgmoid activation function

As seen above, the weights are the most important part of the neural network.

Various techniques have been developed to ensure neural nets produce the best outputs some of which include adding a dropout, using less complex networks etc. In this paper we show how genetic algorithms can be used to generate and optimize the weight generation process.

1.1

The values of x1 and x2 and generated randomly from a range between -1 and 1. To get the respective values of y, we pass the two variables x1 and x2 through the function shown in fig 1. This would return y values which would then be appended to a list which would be merged with the input values to create the dataset. This was done for 1100 values as instructed.

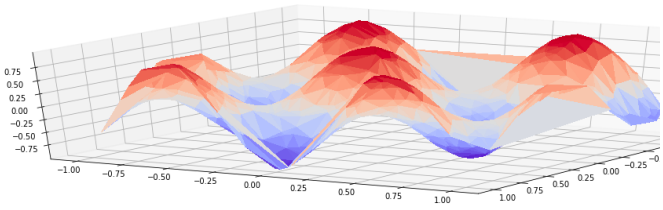

Fig 1.1.1 plot of y=f(x1,x2) dataset

1.2

After generating the dataset with 1100 values we split the data into training and testing dataset. The train and test sizes are 1000 and 100 respectively

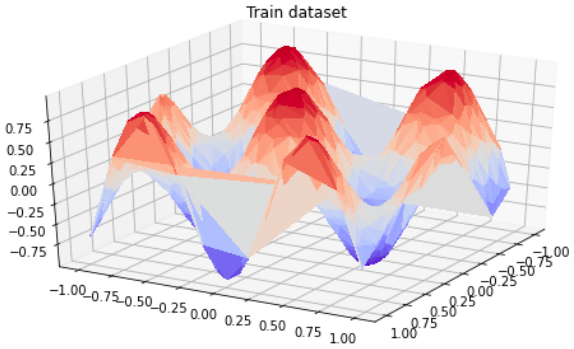

Fig 1.2.1: Training dataset split of 1000

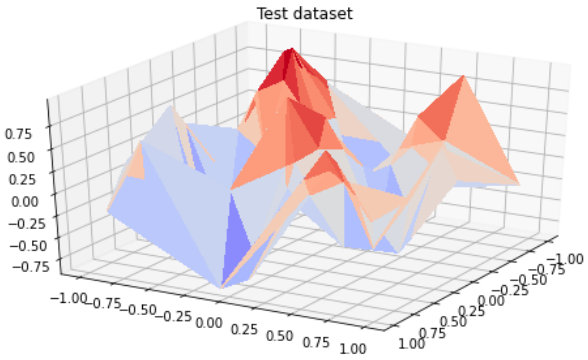

Fig 1.2.2 : Testing dataset split 100

1.3

A class data structure was used to create the neural network. The class Net contains 3 linear layers that are initialized by setting the size of the neurons on each layer.

The neural network comprises 2 input nodes, 2 six node hidden layers, and one output layer.

This was done so as to emulate fig 3. The class output is as shown below

```
Net(
  (hidden): Linear(in_features=2, out_features=6, bias=True)
  (hidden2): Linear(in_features=6, out_features=6, bias=True)
  (out): Linear(in_features=6, out_features=1, bias=True)
)
```

Fig 1.3.1 Neural network initialization

The class also has a forward propagation method that is responsible for moving the data from the input to output layers through the hidden layers by applying the sigmoid activation function.

1.4

To add weights into the network, we use the weightsIntoNetwork function which takes in a list of 67 weights. The weights are then sliced based on the network structure. The first 12 weights go to the layer between the input and the hidden layer, The next 36 weights go to the layer between the 1st hidden layer and the 2nd hidden layer, the next 6 weights are assigned between the 2nd hidden layer and the output. The remaining 13 weights would go to the biases of each layer. We then save the neural net to disk.

The weightsOutofNetwork takes in the neural network object and reads the weight values of each layer, reshapes it and returns a list of weights

To change the 1st layer weights, the modifyNetworkLayer takes in the mynet object, extracts and prints the first layer weights, the function then adds 0.1 to the first three weights. The old weights and the new weights are then returned and output as shown in figure 1.4.1 with the first layer consisting of the first 12 rows. The part highlighted in yellow shows the 3 weights that were modified by adding 0.1

	old weights	new weights
0	0.000000	0.100000
1	0.015152	0.115152
2	0.030303	0.130303
3	0.045455	0.045455
4	0.060606	0.060606
5	0.075758	0.075758
6	0.090909	0.090909
7	0.106061	0.106061
8	0.121212	0.121212
9	0.136364	0.136364
10	0.151515	0.151515
11	0.166667	0.166667
12	0.181818	0.181818
13	0.196970	0.196970
14	0.212121	0.212121
15	0.227273	0.227273

Fig 1.4.1: Table showing modified weights

1.5

To generate the weights, a binary coded genetic algorithm using grey coding is used. The algorithm starts by creating a population of size 50 and evaluating the individuals which is done by passing them to the fitness function.

The fitness function eval_sphere takes in an individual as a parameter. The individual is then reshaped to form 67, 30 bit chromosomes. Passing each of these chromosomes through the decoding function chrom2real, returned a weight value ranging from -20 to 20. This value is then divided by 20 so as to scale it between the range -1 and 1. These scaled weights are then inserted into the neural net using the weightsIntoNetwork function.

Using the neural network object mynet, we pass the dataset and get the predicted output based on the previously initialised weights. These predictions will then be compared to the original values i.e y_train using the mean squared error loss as shown below.

$$L(y, \hat{y}) = \frac{1}{N}\sum_{i=0}^{N}(y - \hat{y}_i)^2$$

The y represents the predicted and the actual values. Being a minimization problem we want the weights with the smallest loss/error. We return the inverse of the mean squared error as the DEAP library does not directly allow minimization while using the roulette selection.

Within the generations another neural net, that has been pre loaded with the best individual's decoded weights, is used for testing. And the results are as below. This went on for a 1000 generations.

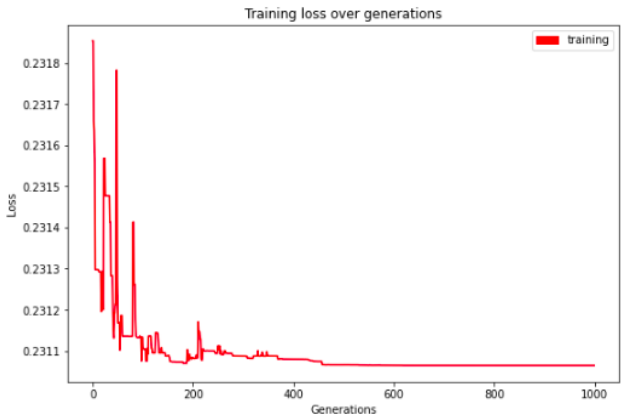

Fig 1.5.1 Training loss over generations

The training loss is seen to decrease with increase in generations.This shows the model is learning as the mean squared error loss is decreasing. The graph is then seen to flatten after some point. This might be the model has reached the optimal solution or might be stuck in a local minima.

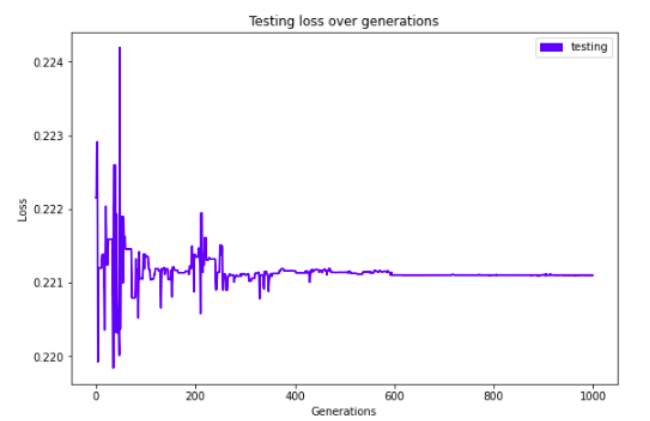

Fig 1.5.2 Testing loss over generations

The testing dataset is seen to start by having high, rapid spikes in loss, and later flattening with increase in generations. This spikes earlier on are due to the model having not fully trained, and gradually flattening as an optimal solution is reached.

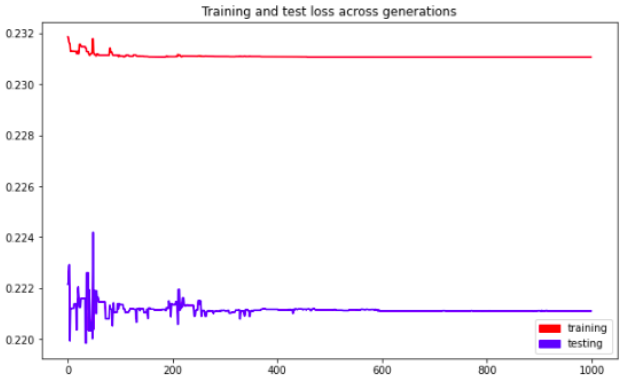

Fig 1.5.3 combined training and testing loss over generations

When the two graphs are merged, we can see the training loss being larger than the testing loss, this is because the model is learning and is still making plenty of mistakes(errors) which result in a higher loss. As the loss flattens at around 500 generations, this would be the optimal time to stop the training.

1.6

After passing 1100 values ranging from -1 to 1, as 2 columns(x1,x2) into the neural network, the model predicted values which when plotted together with the x1 and x2 produced the smooth graph below in Fig 1.6.1

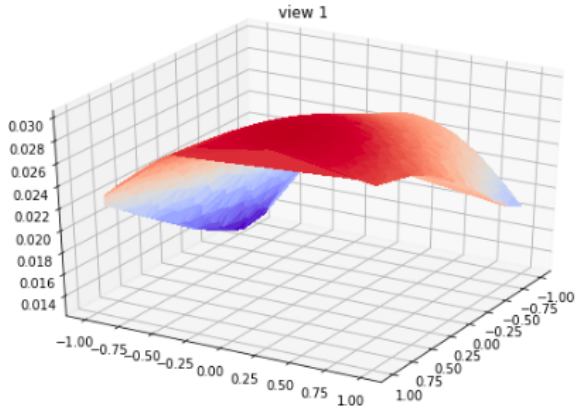

Fig 1.6.1 neural network predictions view 1

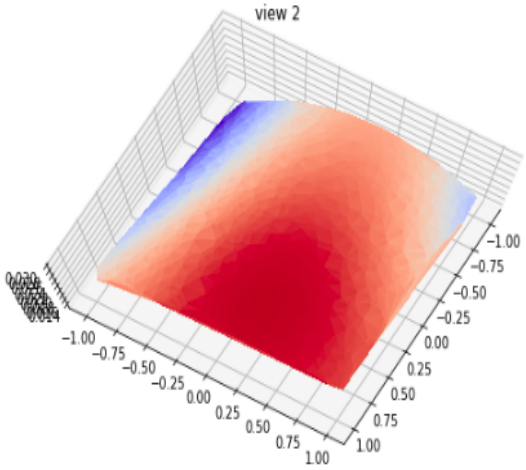

Fig 1.6.2 neural network predictions view 2

1.7

The function real2chrome takes in a weight (decoded value) and returns the chromosome (encoded value). The figure 1.7.1 shows the sample weights generated randomly within the specified range. The weights are then passed to the real2chrome function to generate the genotypes which are seen in the 2nd column. These genotypes are then passed to the chrom2real function which returns them to the original values as seen under phenotypes.

	sample weights	genotypes	phenotypes
0	-1.000000	01000101010101010101010101010101	-1.000000
1	-0.969697	01000101001011101010000010100	-0.969697
2	-0.939394	01000101000000101001011101010 0	-0.939394
3	-0.909091	01000111001110011100111001110 0	-0.909091
4	-0.878788	01000111010100000010100101110 1	-0.878788
...	...	...	...
62	0.878788	11000111010100000010100101110 1	0.878788
63	0.909091	11000111001110011100111001110 0	0.909091
64	0.939394	11000101000000101001011101010 0	0.939394
65	0.969697	11000101001011101010000010100	0.969697
66	1.000000	11000101010101010101010101010101	1.000000

67 rows × 3 columns

Fig 1.7.1 table showing encoded and decoded weights

1.8

The Lamarckian approach explains how offspring can inherit learnt traits from the parents, similar to giraffes having a long neck when they initially had shorter necks but had to evolve so as to survive (according to the theory) . This can also be seen in our genetic algorithm.

The algorithm starts by generating a population of size 50, the 50 individuals are then sent to the fitness function eval_sphere which was re-initialised. When the individual is received by the fitness function, the weights are extracted as explained earlier, and applied to the network.

This approach uses a local search optimizer known as 'rprop' which implements the resilient backpropagation algorithm. The optimizer was initialized with a learning rate of 0.02 which would allow us to get to the optimal solution more efficiently.

A loop is then run for 30 iterations inside the fitness function which is then used to simulate the local search. After the back propagation, the new learnt weights are extracted and encoded to a chromosome which will then be returned back to the population together with its fitness. The generations were set to 100 as the learning took a lot of time.

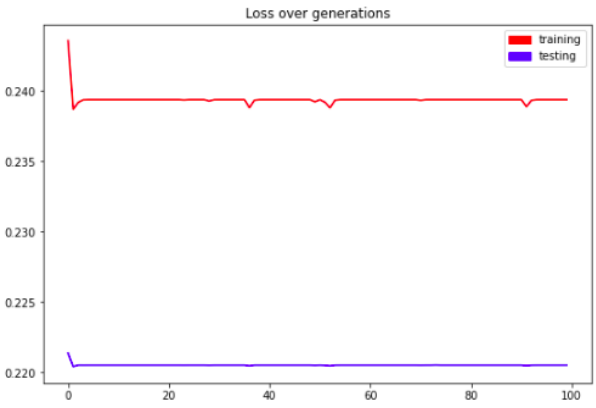

Fig 1.8.1 joint Lamarckian training and testing graph

The figure 1.8.1 above shows the general loss values being smaller than the ones without learning as seen in fig 1.5.3 this is because the model performs better because of the learnt weights. The loss also seems to flatten quite earlier in the evolution; this might be because an optimal solution was reached.

1.9

In the Baldwinian approach, all the traits acquired by the individual will only help increase the individual's fitness. The acquired traits will however not be passed down to the offspring. Such traits in real life are such as an injury.

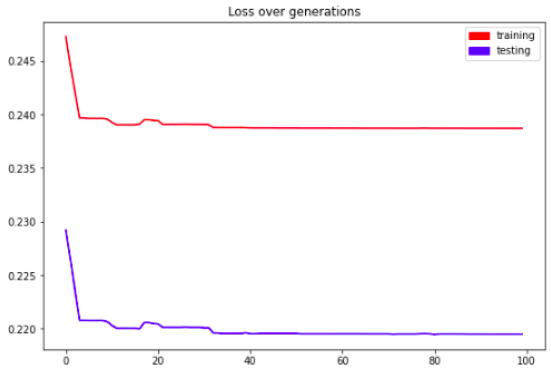

Fig 1.9.1 joint Baldwinian training and testing graph

The Baldwinian approach takes on a similar approach as that of Lamarckian, the only difference is in Baldwinian we do not replace the population with the learnt chromosomes.

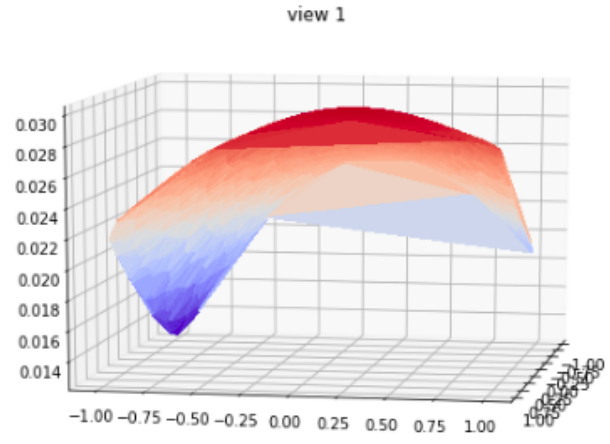

Fig 1.9.2 Neural net output surface plot view 1

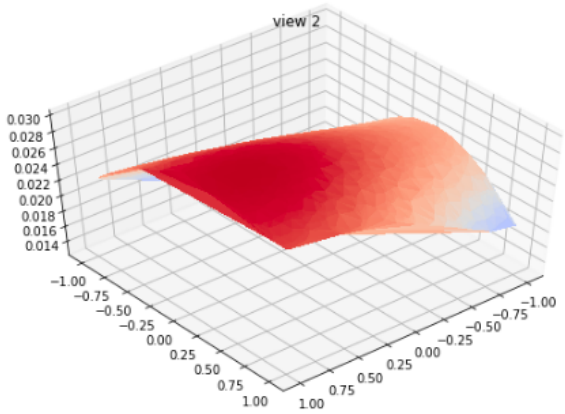

Fig 1.9.3 Neural net output surface plot view 2

Comparison between Lamarkian and Baldwinian

The two approaches are quite similar but have the following differences according to my observation

Lamarckian	**Baldwinian**
Has generally lower loss values which result in better fitnesses	Has higher losses as compared to Lamarckian
Converges earlier in the generations	Converges later as compared to Lamarckian
Trains slower for the same number of generations	Trains faster as compared to Lamarckian

Fig 1.9.4 table showing differences between Lamarkian and Baldwinian approaches of evolution

References

[1] Kinnear, K. E. (1994). A Perspective on the Work in this Book. In K. E. Kinnear (Ed.), Advances in Genetic Programming (pp. 3-17). Cambridge: MIT Press.

[2]Choi S-S, Moon B-R (2005) A graph-based Lamarckian–Baldwinian hybrid for the sorting network problem. IEEE Trans Evol Comput 9:105–114.

Appendix

```
# -*- coding: utf-8 -*-
"""ci 2.ipynb

Automatically generated by Colaboratory.

Original file is located at

https://colab.research.google.com/drive/1FfFkfP5Fk05Rqk4H142FT7McsF2eFdEZ
"""

!pip install deap

import pandas as pd
import torch
import random
import numpy as np
from sympy.combinatorics.graycode import GrayCode
from sympy.combinatorics.graycode import gray_to_bin
from deap import creator, base, tools, algorithms
from numpy import genfromtxt
import torch.nn.functional as F
from numpy import genfromtxt
import math,torch
from sympy.combinatorics.graycode import bin_to_gray
import matplotlib.pyplot as plt
import matplotlib
import matplotlib.patches as mpatches

redpatch = mpatches.Patch(color='red', label='training')
bluepatch = mpatches.Patch(color='blue', label='testing')

# formula we are trying to predict, and generate op
def theformula(x1,x2):
    return math.sin(3.5*x1 + 1.0)*math.cos(5.5*x2)

# function for generating the dataset
def generateData(size):
    x1 = np.random.uniform(low=-1, high=1, size=(size,))
    print("-> ",type(x1))
    x2 = np.random.uniform(low=-1, high=1, size=(size,))
    y = [theformula(i,j) for i,j in zip(x1,x2)]
    data = pd.DataFrame([x1,x2,y]).transpose()
    data.columns=["x1","x2","y"]
    return data

data = generateData(1100)
data

"""1.1"""

fig = plt.figure(figsize=(20,5))
# surface_plot with color grading and color bar
ax = plt.axes(projection ="3d")
ax.plot_trisurf(data['x1'], data['x2'], data['y'],cmap=matplotlib.cm.coolwarm, linewidth=0,zorder=0, antialiased=False)
ax.view_init(30,30)

"""1.2"""

# getting train and test dataset
train = data.head(1000)
test = data.tail(100)
train.shape,test.shape

fig = plt.figure(figsize = (20,5))
ax = fig.add_subplot(1, 2, 1, projection='3d')
ax.plot_trisurf(train['x1'], train['x2'], train['y'],cmap=matplotlib.cm.coolwarm, linewidth=0,zorder=0, antialiased=False)
ax.view_init(30,30)
plt.title("Train dataset")

ax = fig.add_subplot(1, 2, 2, projection='3d')
ax.plot_trisurf(test['x1'], test['x2'], test['y'],cmap=matplotlib.cm.coolwarm, linewidth=0,zorder=0, antialiased=False)
ax.view_init(30,30)
plt.title("Test dataset")
```

```
"""1.3"""

class Net(torch.nn.Module):
    # initialise two hidden layers and one output layer
    def __init__(self, n_feature, n_hidden1,n_hidden2, n_output):
        super(Net, self).__init__()
        self.hidden = torch.nn.Linear(n_feature, n_hidden1)  # 1st hidden layer
        self.hidden2 = torch.nn.Linear(n_hidden1, n_hidden2)  # 2nd hidden layer
        self.out = torch.nn.Linear(n_hidden2, n_output)  # output layer


    # foward propagation, input -> hidden -> out
    def forward(self, x):
        x = torch.sigmoid(self.hidden(x))  # activation function(sigmoid)
        x = torch.sigmoid(self.hidden2(x))
        x = self.out(x) # (linear layer)
        return x

mynet = Net(n_feature=2, n_hidden1=6,n_hidden2=6, n_output=1)
mynet

"""1.4"""

weights = np.linspace(0, 1, 67)
weights

def weightsIntoNetwork(weights):
    weights = np.array(weights) #make sure the weights are np.arrays

    # reshaping the weights to match the layers
    hidden_layer = torch.FloatTensor(weights[:12].reshape(6,2)) # 12
    hidden_layer_2 = torch.FloatTensor(weights[12:48].reshape(6,6)) # 36
    output_layer = torch.FloatTensor(weights[48:54].reshape(1,6)) # 6

    # reshaping weights to match biases
    hidden_bias = torch.FloatTensor(weights[54:60])
    hidden_bias_2 = torch.FloatTensor(weights[60:66])
    out_p_bias = torch.FloatTensor(weights[66:67])

    # inserting weights into to the nn
    mynet.hidden.weight = torch.nn.parameter.Parameter(hidden_layer)
    mynet.hidden2.weight = torch.nn.parameter.Parameter(hidden_layer_2)
    mynet.out.weight = torch.nn.parameter.Parameter(output_layer)
    mynet.hidden.bias = torch.nn.parameter.Parameter(hidden_bias)
    mynet.hidden2.bias = torch.nn.parameter.Parameter(hidden_bias_2)
    mynet.out.bias = torch.nn.parameter.Parameter(out_p_bias)
    # save the nn
    torch.save(mynet.state_dict(), 'net_params.pkl')
    return f"inserted: {weights}"

def weightsOutofNetwork(mynet):
  # function for getting the weights out of the network.
  # uses the mynet.hidden.weight to extract the weights, then converts to list and finally reshaped
    return list(np.array(mynet.hidden.weight.tolist()).reshape(12,)) + list(np.array(mynet.hidden2.weight.tolist()).reshape(36,)) + list(np.array(mynet.out.weight.tolist()).reshape(6,)) + list(np.array(mynet.hidden.bias.tolist())) + list(np.array(mynet.hidden2.bias.tolist())) + list(np.array(mynet.out.bias.tolist()))

weightsIntoNetwork(weights)
```

```
weightsOutofNetwork(mynet)

def changeFirstLayerWeights(weights):
    # print("Weights to be modified: ",weights[0:3])
# selects the first 3 weights
    for index,item in enumerate(weights[0:3]):
        weights[index] = weights[index] + 0.1 #
modifying by adding 0.1



    # torch.save(mynet.state_dict(),
'net_params.pkl')
    return weights

def modifyNetworkLayer(mynet):
     # this function calls the
changefirstlayerweights function that changes the
weights of the first layer
    #  and returns the new weights
    print(f"first layer weights:
{np.array(mynet.hidden.weight.tolist()).reshape(1
2,)}" )
    print("")
     # we extract the hidden layer 1 weights
    modified_new_layer =
list(changeFirstLayerWeights(np.array(mynet.hidd
en.weight.tolist()).reshape(12,)))
    print("modified first layer weights:
",modified_new_layer)
    old_weights =
list(np.array(mynet.hidden.weight.tolist()).reshape
(12,)) +
list(np.array(mynet.hidden2.weight.tolist()).reshap
e(36,)) +
list(np.array(mynet.out.weight.tolist()).reshape(6,)
) + list(np.array(mynet.hidden.bias.tolist())) +
list(np.array(mynet.hidden2.bias.tolist())) +
list(np.array(mynet.out.bias.tolist()))
    print("old weights: ",old_weights)
    new_weights = modified_new_layer +
list(np.array(mynet.hidden2.weight.tolist()).reshap
e(36,)) +
list(np.array(mynet.out.weight.tolist()).reshape(6,)
) + list(np.array(mynet.hidden.bias.tolist())) +
list(np.array(mynet.hidden2.bias.tolist())) +
list(np.array(mynet.out.bias.tolist()))
    # print("Modified ")
    weightsIntoNetwork(new_weights)
    return old_weights,new_weights

old_weights,new_weights =
modifyNetworkLayer(mynet)
weightsIntoNetwork(new_weights)
print("new weights:
",weightsOutofNetwork(mynet))

def custom_style(row):
    # print(row["old weights"])
    color = 'white'
    if row["old weights"] != row["new weights"]:
        color = 'yellow'

    return ['background-color: %s' %
color]*len(row.values)

res=pd.DataFrame(list(zip(old_weights,
new_weights)),columns=['old weights','new
weights'])
res.style.apply(custom_style, axis=1)



"""1.5"""

# formating dataset
x_train=train[train.columns.drop('y')]
x_train=x_train.values.tolist()
x_test=test[test.columns.drop('y')]
x_test=x_test.values.tolist()

y_train = train['y'].tolist()
y_test = test['y'].tolist()

# converting to tensors for the neural net
x_train = torch.as_tensor(x_train,
dtype=torch.float32)
y_train = torch.as_tensor(y_train,
dtype=torch.float32)
```

```
x_test = torch.as_tensor(x_test,
dtype=torch.float32)
y_test = torch.as_tensor(y_test,
dtype=torch.float32)

creator.create("FitnessMax", base.Fitness,
weights=(1.0,))
creator.create("Individual", list,
fitness=creator.FitnessMax)

popSize     = 50 #Population size
dimension   = 67 #Numer of decision variable x
numOfBits   = 30 #Number of bits in the
chromosomes
iterations  = 1000 #Number of generations to be
run
dspInterval = 10
nElitists   = 1 #number of elite individuals selected
omega       = 5
crossPoints = 2 #variable not used. instead
tools.cxTwoPoint
crossProb   = 0.6
flipProb    = 1. / (dimension * numOfBits) #bit
mutate prob
mutateprob  = .1 #mutation prob
maxnum      = 2**numOfBits #absolute max size
of number coded by binary list 1,0,0,1,1,....

loss_func = torch.nn.MSELoss()
toolbox = base.Toolbox()

# Attribute generator
#                 define 'attr_bool' to be an attribute
('gene')
#                 which corresponds to integers
sampled uniformly
#                 from the range [0,1] (i.e. 0 or 1
with equal
#                 probability)
toolbox.register("attr_bool", random.randint, 0, 1)

# Structure initializers
#define 'individual' to be an individual
#consisting of numOfBits*dimension 'attr_bool'
elements ('genes')
toolbox.register("individual", tools.initRepeat,
creator.Individual,
    toolbox.attr_bool, numOfBits*dimension)

# define the population to be a list of individuals
toolbox.register("population", tools.initRepeat, list,
toolbox.individual)



def eval_sphere(ind):

    ind = np.asarray(ind)
    # get the decoded values of the chromosome
    weights =np.asarray(separatevariables(ind))
    # insert weights into network
    weightsIntoNetwork(weights)

    # get the y_pred
    out =  mynet(x_train)
    # get the mle loss
    loss = loss_func(out,y_train)



    #
mynet.load_state_dict(torch.load('net_params.pkl'
))
    return (1/(0.01+loss.item()),)



#----------
# Operator registration
#----------
# register the goal / fitness function
toolbox.register("evaluate", eval_sphere)

# register the crossover operator
toolbox.register("mate", tools.cxTwoPoint)

# register a mutation operator with a probability to
toolbox.register("mutate", tools.mutFlipBit,
indpb=flipProb)

# roulette selection declaration
```

```
toolbox.register("select", tools.selRoulette,
fit_attr='fitness')

#----------

# Convert chromosome to real number
# input: list binary 1,0 of length numOfBits
representing number using gray coding
# output: real value
def chrom2real(c):

    indasstring=''.join(map(str, c))
    degray=gray_to_bin(indasstring)
    numasint=int(degray, 2) # convert to int from
base 2 list
    numinrange= -20+40*numasint/maxnum

    # divide by 20 to fit it into the required range -1
to 1
    return numinrange/20



def separatevariables(v):
    v = v.reshape((67, 30))
    # v is 67, 30 bit chromosomes
    # the function then returns 67 decoded values
    return [chrom2real(i) for i in v]


def main():
    train_losses = []
    test_losses = []
    # create an initial population of individuals
(where
    # each individual is a list of integers)
    pop = toolbox.population(n=popSize)



    # Evaluate the entire population
    fitnesses = list(map(toolbox.evaluate, pop))

    # assign fitnesses to individuals
    for ind, fit in zip(pop, fitnesses):
        ind.fitness.values = fit

    g = 0
    while g < iterations:
        g = g + 1
        print("-- Generation %i --" % g)



        # Select the next generation individuals
        offspring = tools.selBest(pop, nElitists) +
toolbox.select(pop,len(pop)-nElitists)
        # Clone the selected individuals
        offspring = list(map(toolbox.clone, offspring))

        # Apply crossover and mutation on the
offspring
        for child1, child2 in zip(offspring[::2],
offspring[1::2]):

            # cross two individuals with probability
CXPB
            if random.random() < crossProb:
                toolbox.mate(child1, child2)
                del child1.fitness.values
                del child2.fitness.values

        for mutant in offspring:

            # mutate an individual with probability
mutateprob
            if random.random() < mutateprob:
                toolbox.mutate(mutant)
                del mutant.fitness.values

        # Evaluate the individuals with an invalid
fitness
        invalid_ind = [ind for ind in offspring if not
ind.fitness.valid]
        fitnesses = map(toolbox.evaluate,
invalid_ind)
        for ind, fit in zip(invalid_ind, fitnesses):
            ind.fitness.values = fit

        # The population is entirely replaced by the
offspring
        best_ind = tools.selBest(pop, 1)[0]
```

```
train_losses.append(1/best_ind.fitness.values[0])
        pop[:] = offspring

        eval_sphere(best_ind)
        out =  mynet(x_test)  # input x and predict
based on x
        loss = loss_func(out,y_test)
        test_losses.append(loss.item())


    print("-- End of (successful) evolution --")
    return train_losses,test_losses


if __name__ == "__main__":
    train_losses,test_losses= main()

plt.figure(figsize=(10,6))
plt.plot(train_losses, 'r')
plt.plot(test_losses, 'b')
plt.title("Training and test loss across
generations")
plt.legend(handles=[redpatch,bluepatch])
plt.show()

fig = plt.figure(figsize = (20,6))
ax = fig.add_subplot(1, 2, 1)
plt.plot(train_losses, 'r')
plt.title("Training loss over generations")
plt.legend(handles=[redpatch])
plt.xlabel("Generations")
plt.ylabel("Loss")

ax = fig.add_subplot(1, 2, 2)
plt.plot(test_losses, 'b')
plt.title("Testing loss over generations")
plt.legend(handles=[bluepatch])
plt.xlabel("Generations")
plt.ylabel("Loss")
plt.show()

"""1.6"""

x_values=data[data.columns.drop('y')]
x_values=x_values.values.tolist()
x_values = torch.as_tensor(x_values,
dtype=torch.float32)
# get the predicted values
out =  mynet(x_values)
y=pd.DataFrame(out.tolist(),columns=['y'])

# surface_plot with color grading and color bar
fig = plt.figure(figsize = (20,6))
ax = fig.add_subplot(1, 2, 1, projection='3d')
ax.plot_trisurf(data['x1'],
data['x2'],y['y'],cmap=matplotlib.cm.coolwarm,
linewidth=0,zorder=0, antialiased=False)
ax.view_init(30,30)
plt.title("view 1")

ax = fig.add_subplot(1, 2, 2, projection='3d')
ax.plot_trisurf(data['x1'],
data['x2'],y['y'],cmap=matplotlib.cm.coolwarm,
linewidth=0,zorder=0, antialiased=False)
plt.title("view 2")
ax.view_init(80,30)

"""1.7"""

popSize     = 50 #Population size
dimension   = 67 #Numer of decision variable x
numOfBits   = 30 #Number of bits in the
chromosomes
iterations  = 100 #Number of generations to be
run
dspInterval = 10
nElitists   = 1 #number of elite individuals selected
omega       = 5
crossPoints = 2 #variable not used. instead
tools.cxTwoPoint
crossProb   = 0.6
flipProb    = 1. / (dimension * numOfBits) #bit
mutate prob
mutateprob  = .1 #mutation prob
maxnum      = 2**numOfBits

# commonly reused
def chrom2real(c):
    indasstring=''.join(map(str, c))
```

```
    degray=gray_to_bin(indasstring)
    numasint=int(degray, 2) # convert to int from
base 2 list
    numinrange= -20+40*numasint/maxnum

    # devide by 20 to fit it into the required range -1
to 1
    return numinrange

def separatevariables(v):

    v = v.reshape((67, 30))
    # v is 67, 30 bit chromosomes
    # the function then returns 67 decoded values
    return [chrom2real(i) for i in v]

def real2chrome(weight):
  # setting required string lenth t to 30
    t=30
  # check if weighteight passed the limits
    if weight > 20:
        weight = 20
    elif weight < -20:
        weight = -20
    numinrange= (weight+20)*maxnum/40
    binary_rep=bin(int(numinrange))[2:]
    grayValues=bin_to_gray(binary_rep)
    lengrayval=len(grayValues)
    # chromosome length is not always 30 we add
0s to the beginning if less than 30
    if lengrayval<t:
        dif =t-lengrayval
        return ('0'*dif)+grayValues
    elif lengrayval==t:
        return grayValues

real2chrome(-0.96969697)

chrom2real("01000101001011101010000001010
0")

# generating random weights
sample_weights = np.linspace(-1, 1, 67)
# sample_weights

# chromosome values of the sample weights
genotypes = [real2chrome(i) for i in
sample_weights]
# genotypes

# the reversed chromosome/ decoded values
phenotypes = [chrom2real(i) for i in genotypes]
# phenotypes

res2=pd.DataFrame(list(zip(sample_weights,geno
types, phenotypes)),columns=["sample
weights",'genotypes','phenotypes'])
res2

"""# 1.8"""

creator.create("FitnessMax", base.Fitness,
weights=(1.0,))
creator.create("Individual", list,
fitness=creator.FitnessMax)

popSize     = 50 #Population size
dimension   = 67 #Numer of decision variable x
numOfBits   = 30 #Number of bits in the
chromosomes
iterations  = 100 #Number of generations to be
run
dspInterval = 10
nElitists   = 1 #number of elite individuals selected
omega       = 5
crossPoints = 2 #variable not used. instead
tools.cxTwoPoint
crossProb   = 0.6
flipProb    = 1. / (dimension * numOfBits) #bit
mutate prob
mutateprob  = .1 #mutation prob
maxnum      = 2**numOfBits #absolute max size
of number coded by binary list 1,0,0,1,1,....

optimizer =
torch.optim.Rprop(mynet.parameters(), lr=0.02)
loss_func = torch.nn.MSELoss()

toolbox = base.Toolbox()
```

```
# Attribute generator
#                 define 'attr_bool' to be an attribute
('gene')
#                 which corresultponds to integers
sampled uniformly
#                 from the range [0,1] (i.e. 0 or 1
with equal
#                 probability)
toolbox.register("attr_bool", random.randint, 0, 1)

# Structure initializers
#                   define 'individual' to be an
individual
#                   consisting of
numOfBits*dimension 'attr_bool' elements
('genes')
toolbox.register("individual", tools.initRepeat,
creator.Individual,
    toolbox.attr_bool, numOfBits*dimension)

# define the population to be a list of individuals
toolbox.register("population", tools.initRepeat, list,
toolbox.individual)

# register the crossover operator
toolbox.register("mate", tools.cxTwoPoint)

# register a mutation operator with a probability to
# flip each attribute/gene of 0.05
toolbox.register("mutate", tools.mutFlipBit,
indpb=flipProb)

# operator for selecting individuals for breeding
the next
# generation
toolbox.register("select", tools.selRoulette,
fit_attr='fitness')

def chrom2real(c):
    indasstring=''.join(map(str, c))
    degray=gray_to_bin(indasstring)
    numasint=int(degray, 2) # convert to int from
base 2 list
    numinrange= -20+40*numasint/maxnum
    # devide by 20 to fit it into the required range -1
to 1
    return numinrange/20

# out is supposed to be the individual
def eval_sphere(ind):
    result2 = []
    ind = np.asarray(ind)
#   get weights
    weights =np.asarray(separatevariables(ind))

    weightsIntoNetwork(weights)
    result = ''
    #
    for t in range(30):
        result2 =[]
        out =  mynet(x_train)
        # compute loss
        loss = loss_func(out,y_train)
        # initialize gradient
        optimizer.zero_grad()
        loss.backward()  # backpropagation and
computing of gradients
        optimizer.step()  # apply gradients, go down
gradient
        learnt_weights =
weightsOutofNetwork(mynet)

        # decoding the chromosomes
        for i in learnt_weights:
            result = result + real2chrome(i)

        for i in list(result.strip(" ")):
            result2.append(int(i))

        # re init the result
        result = ''
    return
(1/(0.01+loss.item()),creator.Individual(result2))


# register the goal / fitness function
toolbox.register("evaluate", eval_sphere)
```

```
def initializePopulation(l,t,individuals):
    return individuals


# # this will help update the population

def main():
    # create an initial population of individuals
(where
    # each individual is a list of integers)
    pop = toolbox.population(n=popSize)



    # Evaluate the entire population
    fitnesses = list(map(toolbox.evaluate, pop))

    # unpacking the fitnesses to assign to the
individuals
    fitnesses = [(i[0],) for i in fitnesses]
    for ind, fit in zip(pop, fitnesses):
        ind.fitness.values = fit



    print("  Evaluated %i individuals" % len(pop))



    # Variable keeping track of the number of
generations
    g = 0
    train_losses = []
    test_losses = []
    # Begin the evolution

    while g < iterations:
        # A new generation
        g = g + 1
        print("-- Generation %i --" % g)



        ind_fitnesses = list(map(toolbox.evaluate,
pop))

        fitnesses = [(i[0],) for i in ind_fitnesses]
        individuals = [i[1] for i in ind_fitnesses]


        for ind, fit in zip(pop, fitnesses):
            ind.fitness.values = fit

        # this will update the population

toolbox.register("updatePopulation",initializePopul
ation,list,creator.Individual,individuals)

        # update population
        offspring = toolbox.updatePopulation()

        # Clone the selected individuals
        offspring = list(map(toolbox.clone, offspring))

        # Apply crossover and mutation on the
offspring
        for child1, child2 in zip(offspring[::2],
offspring[1::2]):

            # cross two individuals with probability
CXPB
            if random.random() < crossProb:
                toolbox.mate(child1, child2)
                del child1.fitness.values
                del child2.fitness.values

        for mutant in offspring:

            # mutate an individual with probability
mutateprob
            if random.random() < mutateprob:
                toolbox.mutate(mutant)
                del mutant.fitness.values

#         # Evaluate the individuals with an invalid
fitness
        invalid_ind = [ind for ind in offspring if not
ind.fitness.valid]
        print("Invalid Ind: ",len(invalid_ind))
        fitnesses = map(toolbox.evaluate,
invalid_ind)

        # have to unpack the fitnesses and
individuals
        fitnesses = [(i[0],) for i in ind_fitnesses]
```

```
        individuals = [i[1] for i in ind_fitnesses]

        print("fitnesses after: ",len(fitnesses))

        for ind, fit in zip(invalid_ind, fitnesses):
            ind.fitness.values = fit

        pop[:] = offspring  # The population is
entirely replaced by the offspring

        best_ind = tools.selBest(pop, 1)[0]
        # get the inverse for roullete selection

train_losses.append(1/best_ind.fitness.values[0])



        eval_sphere(best_ind) # input the best
weights into the network
        out =  mynet(x_test)  # input x and predict
based on x
        loss = loss_func(out,y_test)
        test_losses.append(loss.item())


    print("------Successfull evolution-----")
    return train_losses,test_losses


if __name__ == "__main__":

    train_losses,test_losses = main()

plt.figure(figsize=(9,6))
plt.plot(train_losses, 'r')
plt.plot(test_losses, 'b')
plt.title("Loss over generations")
plt.legend(handles=[redpatch,bluepatch])
plt.show()

fig = plt.figure(figsize = (20,6))
ax = fig.add_subplot(1, 2, 1)
plt.plot(train_losses, 'r')
plt.title("Training loss over generations")
plt.legend(handles=[redpatch])
plt.xlabel("Generations")
plt.ylabel("Loss")

ax = fig.add_subplot(1, 2, 2)
plt.plot(test_losses, 'b')
plt.title("Testing loss over generations")
plt.legend(handles=[bluepatch])
plt.xlabel("Generations")
plt.ylabel("Loss")
plt.show()

"""# 1.9"""

creator.create("FitnessMax", base.Fitness,
weights=(1.0,))
creator.create("Individual", list,
fitness=creator.FitnessMax)

popSize     = 50 #Population size
dimension   = 67 #Numer of decision variable x
numOfBits   = 30 #Number of bits in the
chromosomes
iterations  = 100 #Number of generations to be
run
dspInterval = 10
nElitists   = 1 #number of elite individuals selected
omega       = 5
crossPoints = 2 #variable not used. instead
tools.cxTwoPoint
crossProb   = 0.6
flipProb    = 1. / (dimension * numOfBits) #bit
mutate prob
mutateprob  = .1 #mutation prob
maxnum      = 2**numOfBits #absolute max size
of number coded by binary list 1,0,0,1,1,....


optimizer =
torch.optim.Rprop(mynet.parameters(), lr=0.02)
loss_func = torch.nn.MSELoss()
toolbox = base.Toolbox()

# Attribute generator
#                 define 'attr_bool' to be an attribute
('gene')
```

```
#                   which corresponds to integers
sampled uniformly
#                   from the range [0,1] (i.e. 0 or 1
with equal
#                   probability)
toolbox.register("attr_bool", random.randint, 0, 1)

# Structure initializers
#                    define 'individual' to be an
individual
#                    consisting of
numOfBits*dimension 'attr_bool' elements
('genes')
toolbox.register("individual", tools.initRepeat,
creator.Individual,
    toolbox.attr_bool, numOfBits*dimension)

# define the population to be a list of individuals
toolbox.register("population", tools.initRepeat, list,
toolbox.individual)



def eval_sphere(ind):
    weights
=np.asarray(separatevariables(np.asarray(ind)))

    weightsIntoNetwork(weights)

    for t in range(30):
        out =  mynet(x_train)
        loss = loss_func(out,y_train)
        # initialize gradient
        optimizer.zero_grad()
        loss.backward()  # backpropagation,
compute gradients
        optimizer.step()  # apply gradients
        # print("what is: ",t)
    return (1/(0.01+loss.item()),)

#----------
# Operator registration
#----------
# register the goal / fitness function
toolbox.register("evaluate", eval_sphere)

# register the crossover operator
toolbox.register("mate", tools.cxTwoPoint)

# register a mutation operator with a probability to
# flip each attribute/gene of 0.05
toolbox.register("mutate", tools.mutFlipBit,
indpb=flipProb)

# operator for selecting individuals for breeding
the next
# generation: each individual of the current
generation
# is replaced by the 'fittest' (best) of three
individuals
# drawn randomly from the current generation.
toolbox.register("select", tools.selRoulette,
fit_attr='fitness')

#----------

# Convert chromosome to real number
# input: list binary 1,0 of length numOfBits
representing number using gray coding
# output: real value
def chrom2real(c):

    indasstring=''.join(map(str, c))
    degray=gray_to_bin(indasstring)
    numasint=int(degray, 2) # convert to int from
base 2 list
    numinrange= -20+40*numasint/maxnum
    return numinrange/20

# input: concatenated list of binary variables
# output: tuple of real numbers representing
those variables
# requires a loop or other method

def separatevariables(v):


    v = v.reshape((67, 30))

    weights = [chrom2real(i) for i in v]
    return weights
```

```
def main():
    #random.seed(64)

    # create an initial population of individuals
(where
    # each individual is a list of integers)
    pop = toolbox.population(n=popSize)

    # Evaluate the entire population
    fitnesses = list(map(toolbox.evaluate, pop))

    for ind, fit in zip(pop, fitnesses):
        ind.fitness.values = fit

    print("  Evaluated %i individuals" % len(pop))



    # Variable keeping track of the number of
generations
    g = 0
    train_losses = []
    test_losses = []

    # Begin the evolution
    while g < iterations:
        # A new generation
        g = g + 1
        print("-- Generation %i --" % g)

        # Select the next generation individuals
        offspring = tools.selBest(pop, nElitists) +
toolbox.select(pop,len(pop)-nElitists)
        # Clone the selected individuals
        offspring = list(map(toolbox.clone, offspring))



        # Apply crossover and mutation on the
offspring
        for child1, child2 in zip(offspring[::2],
offspring[1::2]):
            # cross two individuals with probability
CXPB
            if random.random() < crossProb:
                toolbox.mate(child1, child2)
                del child1.fitness.values
                del child2.fitness.values

        for mutant in offspring:

            # mutate an individual with probability
mutateprob
            if random.random() < mutateprob:
                toolbox.mutate(mutant)
                del mutant.fitness.values

        # Evaluate the individuals with an invalid
fitness
        invalid_ind = [ind for ind in offspring if not
ind.fitness.valid]
        fitnesses = map(toolbox.evaluate,
invalid_ind)
        for ind, fit in zip(invalid_ind, fitnesses):
            ind.fitness.values = fit

        # The population is entirely replaced by the
offspring
        best_ind = tools.selBest(pop, 1)[0]

train_losses.append(1/best_ind.fitness.values[0])
        pop[:] = offspring

        eval_sphere(best_ind)
        out =  mynet(x_test)  # input x and predict
based on x
        loss = loss_func(out,y_test)
        test_losses.append(loss.item())

    best_ind = tools.selBest(pop, 1)[0]
    print("-- End of (successful) evolution --")

    return train_losses,best_ind,test_losses


if __name__ == "__main__":
    train_losses,best_ind,test_losses= main()
```

```
# train_losses

plt.figure(figsize=(9,6))
plt.plot(train_losses, 'r')
plt.plot(test_losses, 'b')
plt.title("Loss over generations")
plt.legend(handles=[redpatch,bluepatch])
plt.show()

fig = plt.figure(figsize = (20,6))
ax = fig.add_subplot(1, 2, 1)
plt.plot(train_losses, 'r')
plt.title("Training loss over generations")
plt.legend(handles=[redpatch])
plt.xlabel("Generations")
plt.ylabel("Loss")

ax = fig.add_subplot(1, 2, 2)
plt.plot(test_losses, 'b')
plt.title("Testing loss over generations")
plt.legend(handles=[bluepatch])
plt.xlabel("Generations")
plt.ylabel("Loss")
plt.show()

eval_sphere(best_ind)
out =  mynet(x_values)
y_values=pd.DataFrame(out.tolist(),columns=['y'])



# surface_plot with color grading and color bar
fig = plt.figure(figsize = (20,6))
ax = fig.add_subplot(1, 2, 1, projection='3d')
ax.plot_trisurf(data['x1'],
data['x2'],y_values['y'],cmap=matplotlib.cm.coolw
arm, linewidth=0,zorder=0, antialiased=False)
ax.view_init(30,30)
plt.title("view 1")

ax = fig.add_subplot(1, 2, 2, projection='3d')
ax.plot_trisurf(data['x1'],
data['x2'],y_values['y'],cmap=matplotlib.cm.coolw
arm, linewidth=0,zorder=0, antialiased=False)
plt.title("view 2")
ax.view_init(80,30)
plt.show()
```