

Multilayer Perceptron Networks

The main objective of this task is to train feed-forward multilayer perceptron networks with one hidden layer to approximate the following function:

$$y = \sin(2x_1 + 2.0) \cos(0.5x_2) + 0.5$$

$$x_1, x_2 \in [0, 2\pi]$$

The diagram 5 is the structure of the feed-forward neural network to approximate the above function. There are two inputs x_1 and x_2 . These two input nodes are connected to six neurons in the hidden layer. w_{ji} ($i = 1, 2, j =$

$1, 2, 3, 4, 5, 6$) are weights for the connections between input nodes and hidden nodes. A bias is connected to all hidden nodes with weights w_j^0 . $net_j = \sum(w_{ji} x_i)$ is the inputs of the hidden nodes. $f()$ is a sigmoid function used as an activation function in the hidden neurons and $o_j = f(net_j)$ is the outputs of the hidden nodes after applying the activation function. Hidden nodes are connected to one output node with weights v_j and a bias is connected to the output node with weight v^0 . $y = \sum(v_j o_j)$ is the output of the neural network.

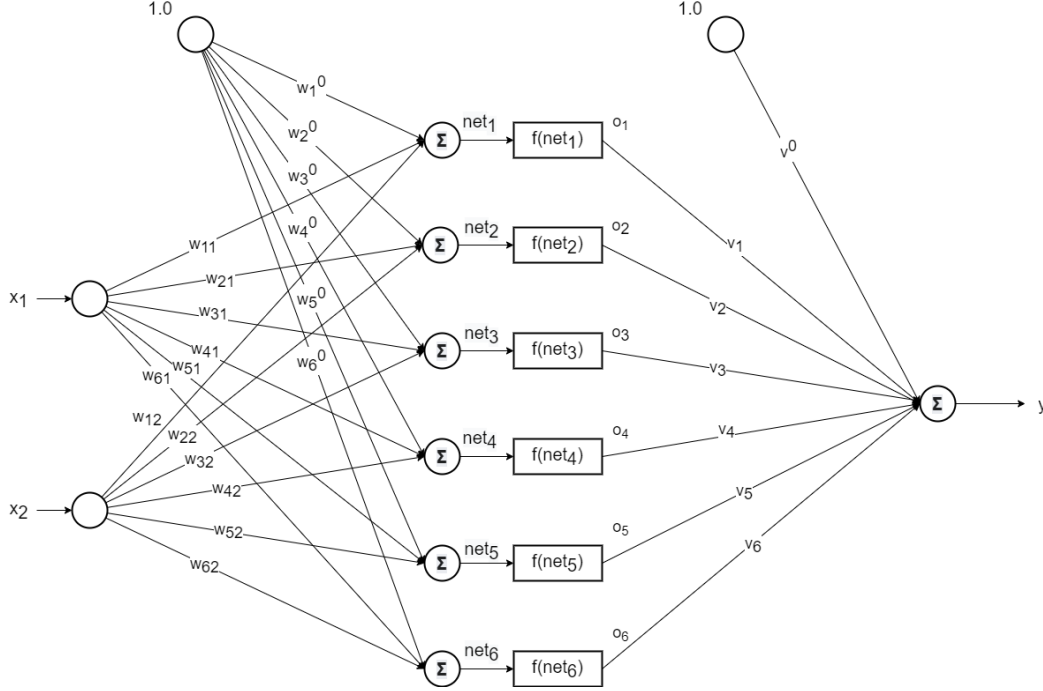


Diagram 5: The structure of the Neural Network

21×2 values are generated for x_1 and x_2 in range between 0 and 2π and corresponding y values are calculated according to the function above.

Random 11 values are selected as the training dataset and the rest as the test dataset. Training dataset is saved in a file named train.dat and test dataset is saved in a file named test.dat. Table 4 and 5 show the datasets saved in the files in train.dat and test.dat, respectively, when the samples are generated randomly with random seed set to 1004.

i	$x_1(i)$	$x_2(i)$	$y(i)$
1	3.69841282664304E-01	5.71321051051943E+00	1.24601106645216E-01
2	3.86730063755006E+00	3.81457893231066E+00	6.00668378233898E-01
3	3.11862368295828E+00	8.01434507917177E-01	1.35397724022026E+00
4	4.86623069123931E+00	5.19359473007613E+00	1.13335504265003E+00
5	6.54594994726704E-01	2.67568195526164E+00	4.61490351206805E-01
6	1.98476639168532E+00	4.16199891405038E+00	6.50674468348775E-01
7	3.27867235391040E+00	4.57057432478613E+00	2.72689490452282E-04

8	2.44740362626799E-01	5.38152220376657E+00	-4.62313352359181E-02
9	1.54985132748291E+00	2.48582985547530E+00	2.01816295480014E-01
10	4.11500155234035E+00	5.47510133045230E+00	1.16293243085397E+00
11	2.63586031826309E+00	3.79755218856439E+00	2.30949165444585E-01

Table 4: Dataset saved in train.dat

i	$x_1(i)$	$x_2(i)$	$y(i)$
1	4.69039976676083E+00	5.42821904529900E+00	1.34332076614932E+00
2	3.51848578989158E+00	5.05690065232357E-01	8.66134910830460E-01
3	1.45541378730716E+00	1.52218973163965E+00	-2.09871629040892E-01
4	2.75510787637882E+00	4.48488315830224E+00	-8.58667494080190E-02
5	1.92027209046635E+00	1.39419203918687E+00	1.71597096604271E-01
6	1.43615220569840E+00	5.29563994862735E+00	1.36931589481192E+00

7	4.243617373712 41E+00	2.1590513610401 8E+00	8.7903880673268 1E-02
8	4.999533346377 50E+00	6.1574905996064 2E+00	1.0362993784059 3E+00
9	3.570007206797 07E+00	1.2929546308511 5E+00	7.2424176397167 0E-01
10	3.546677796639 38E+00	3.7073735566738 7E+00	4.0917358561808 5E-01

Table 5: Dataset saved in test.dat

Diagram 6 and 7 show these datasets in three-dimensional graphics

Diagram 6: Training Data in 3D

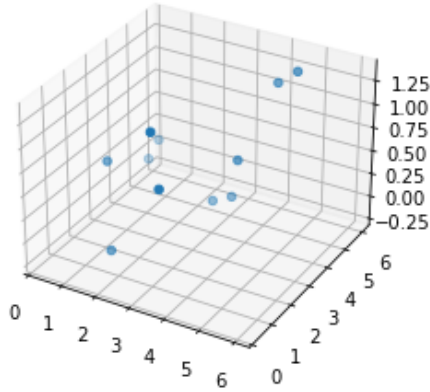
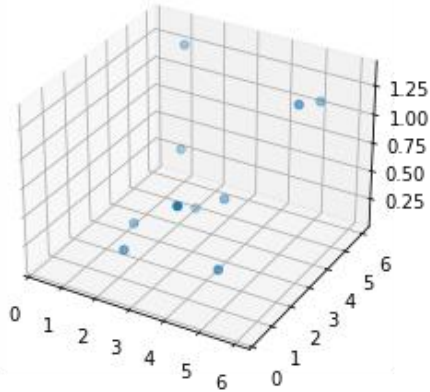


Diagram 7: Test Data in 3D



The algorithm is programmed using Python with PyTorch, Numpy, Sympy and DEAP. The code is in the Appendix A.

There are two inputs and one bias connecting to 6 hidden nodes so the number of connections between the input layer and the hidden layer is $(2 + 1) \times 6 = 18$. There are 6 hidden nodes and a bias connecting to 1 output node so the number of connections between the hidden layer and the output layer is $(6 + 1) \times 1 = 7$. Therefore, the number of weights of the neural network is $18 + 7 = 25$. Each weight uses 15 bits so the total number of bits of each individual is $25 \times 15 = 375$ bits.

N-point crossover and uniform crossover can be used as a crossover operator and flip bit mutation can be used as a mutation operator for binary coding.

According to DeJong (1975), one-point crossover with probability of 0.6 and mutation probability of $1/n$ where n is population size generate best performance when population size is between 50 and 100.

I have tested different crossover operations. I run the code 6 times for each operation with the same random seeds. Crossover probability and mutation probability were set to 0.6 and $1/\text{popSize}$ ($\text{popSize} = 100$), respectively. The averages of the loss with the test data using the best individual after 100 generations are used as the results.

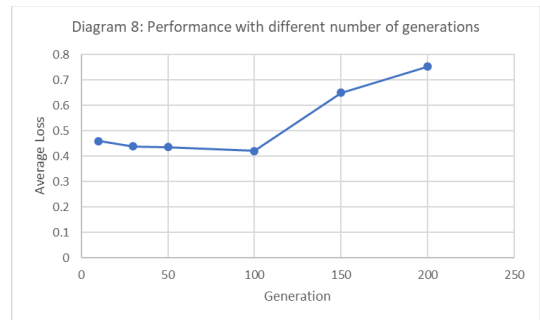
Table 5 shows the result of this test.

Crossover type	Average loss
One-point crossover	0.386
Two-point crossover	0.881
Uniform crossover with flipprob=0.3	1.084
Uniform crossover with flipprob=0.6	0.504
Uniform crossover with flipprob=0.9	0.832

Table 5: Average MSE after 100 generations using different crossover operations

This test also shows that using one-point crossover performs the best.

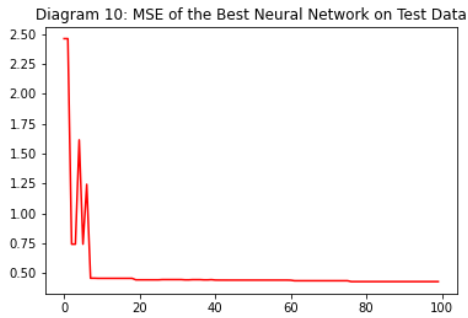
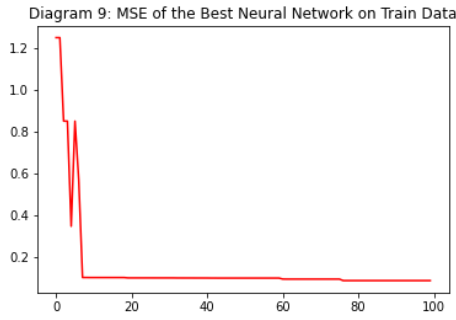
I also tested the performance of the genetic algorithm with different number of generations. I run the code 6 times for each number of generations with the same random seeds and took averages of the loss. Diagram 8 is the plot of the test results.



It shows that the result gets better until 100 and it gets worse after 100 generations.

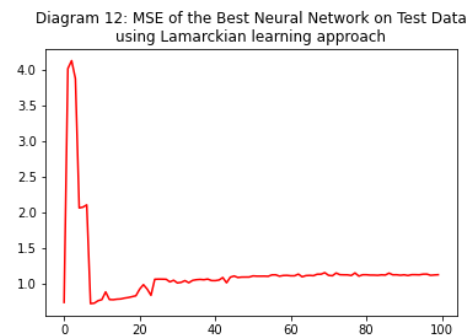
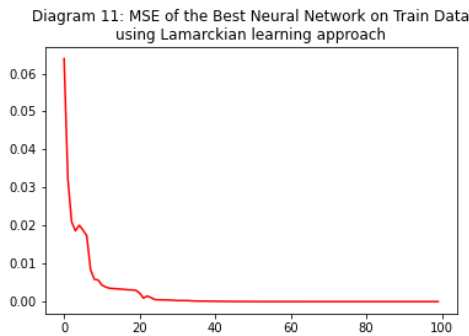
Owing to the above tests, I decided to use one-point crossover with the probability of 0.6 and flip bit mutation with the probability of $1/100$ and set the number of generations 100.

Diagram 9 and 10 show the change of loss of the network using the best individual on training data and test data, respectively.



Lines are added to the code for lifetime learning. The code is in the Appendix B. After the crossover and mutation, a lifetime learning is applied to each individual. The weights of the network are set with the new individual. Gradient calculation is disabled when the weights are set. As a lifetime learning, Rprop algorithm is used to train the network for 30 times. The weights of the network after the lifetime learning is converted to Gray code using “real2chrom” function and replaced with the individual using “get_weight” function. The MSE of the network is set to individual as a fitness value. The same lifetime learning is applied to the whole population.

Diagram 11 and 12 show the change of loss of the network using the best individuals on training data and test data, respectively, using Lamarckian learning approach.



The line where individuals are replaced with new individuals after lifetime learning is removed to change the learning approach to Baldwinian approach.

The diagrams show the loss using pre-local and post-local learning value of best individual on training set and test set over the generations.

Diagram 13: MSE of the Pre-local Best Neural Network on Train Data using Baldwinian learning approach

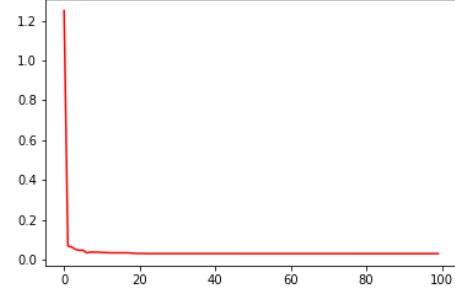


Diagram 14: MSE of the Pre-local Best Neural Network on Test Data using Baldwinian learning approach

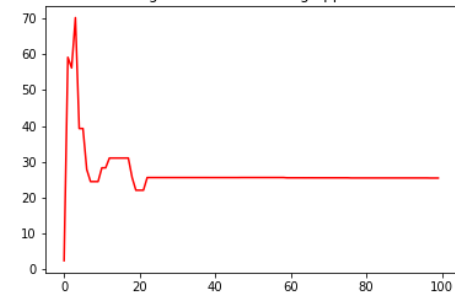


Diagram 15: MSE of the Post-local Best Neural Network on Train Data using Baldwinian learning approach

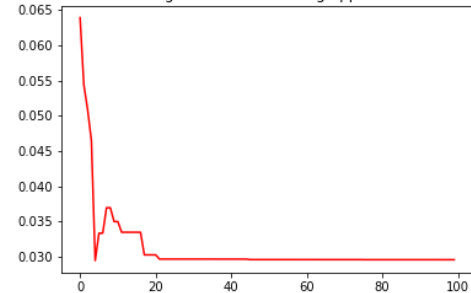
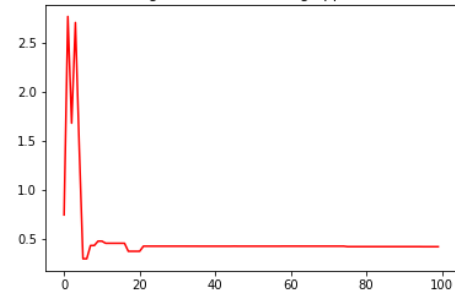


Diagram 16: MSE of the Post-local Best Neural Network on Test Data using Baldwinian learning approach



To generate the diagrams, some lines are added to the code. The code is in the Appendix C. The individuals with invalid fitness values are evaluated after the crossover and mutation and the weights are set to the network using the best individual to test the network before the lifetime learning. The weights after the lifetime learning are assigned to each individual as an attribute "newind" instead of replacing the original individual. The best individual in the population is selected and weights of the network is set with

the "newind" attribute of the individual to test the network after the lifetime learning.

When comparing the results, Baldwinian learning approach is slower than Lamarckian learning approach. Baldwinian learning approach updates the fitness values of individuals but does not update the individuals after the lifetime learning, so the lifetime learning has influence only on the selection of the individuals at the beginning of each generation. On the other hand, Lamarckian learning approach updates individuals after the lifetime learning so the individuals that are selected at the beginning of each generation are better than the previous generation. Therefore, Lamarckian learning approach is faster than Baldwinian learning approach. Using Baldwinian learning approach can be better than algorithms without lifetime

learning because it can select better individuals at the beginning of each generation. However, according to the diagram 13, it seems like the evolution is slowed down after 20 generations. It can be because of the hidden effect where the differences between the fitness values of individuals become very small and the selection pressure decreases. It cannot select better individuals according to the fitness values calculated in the previous generation after the lifetime learning, so the result does not improve very well.

References

DeJong KA (1975). Analysis of the behavior of a class of genetic adaptive. Ph.D. thesis, Univ. of Michigan.

Appendix A

Python code

```
import random
from numpy import random as nprand
import math
import torch
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sympy.combinatorics.graycode import GrayCode
from sympy.combinatorics.graycode import gray_to_bin

from deap import creator, base, tools, algorithms

random.seed(1004)
nprand.seed(1004)

# generate 21 samples for x1 and x2 in range between 0 and 2π
x = nprand.rand(21,2)*2*math.pi
# calculate corresponding y values
y = np.array([math.sin(2*x[i,0]+2.0)*math.cos(0.5*x[i,1])+0.5 for i in range(21)])
# indices to choose random 11 values as the training dataset
index_train = np.random.choice(21, 11, replace=False)
# indices to choose the rest of values as the testing dataset
index_test = np.setdiff1d(np.arange(21), index_train)

# select 11 samples randomly as the training dataset and the rest as the testing dataset
train_x = x[index_train]
test_x = x[index_test]
train_y = y[index_train]
test_y = y[index_test]

# organise the datasets
train_data = np.stack((train_x[:,0],train_x[:,1],train_y), axis=1)
test_data = np.stack((test_x[:,0],test_x[:,1],test_y), axis=1)

# save datasets in data files
np.savetxt('train.dat',train_data)
np.savetxt('test.dat',test_data)

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.text2D(0.05, 0.95, "Diagram 5: Training Data in 3D", transform=ax.transAxes)
ax.set_xlim3d(0, 2*math.pi)
ax.set_ylim3d(0,2*math.pi)
ax.scatter(x[:11,0],x[:11,1],y[:11])
plt.savefig('d5.png')

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.text2D(0.05, 0.95, "Diagram 6: Test Data in 3D", transform=ax.transAxes)
ax.set_xlim3d(0, 2*math.pi)
ax.set_ylim3d(0,2*math.pi)
ax.scatter(x[11:,0],x[11:,1],y[11:])
plt.savefig('d6.png')
plt.show()

train_x = torch.as_tensor(train_x, dtype=torch.float32)
train_y = torch.as_tensor(train_y, dtype=torch.float32).reshape(11,1)
test_x = torch.as_tensor(test_x, dtype=torch.float32)
test_y = torch.as_tensor(test_y, dtype=torch.float32).reshape(10,1)
```

```

# Create single objective minimizing fitness class called "FitnessMin"
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
# Create an Individual class inherit from list with fitness attribute
creator.create("Individual", list, fitness=creator.FitnessMin)

n_feature=2
n_hidden=6
n_output=1
dimension = (n_feature+1)*n_hidden+(n_hidden+1)*n_output #Number of weights
numOfBits = 15 #Number of bits of each weight
maxnum = 2**numOfBits #absolute max size of number coded by binary list 1,0,0,1,1,...
popSize = 100 #Population size
iterations = 100 #Number of generations to be run
nElitists = 1 #number of elite individuals selected
crossProb = 0.6 # the probability for performing crossover
flipProb = 1. / (dimension * numOfBits) #Independent probability for each attribute to be exchanged
mutateprob = 1. / popSize # the probability for performing mutation

print("Total number of bits required for encoding the weights: ", numOfBits*dimension, " bits")

class Net(torch.nn.Module):
    # initialise one hidden layer and one output layer
    def __init__(self, n_feature, n_hidden, n_output):
        super(Net, self).__init__()
        self.hidden = torch.nn.Linear(n_feature, n_hidden) # hidden layer
        self.out = torch.nn.Linear(n_hidden, n_output) # output layer

    # connect up the layers: the input passes through the hidden, then the sigmoid, then the output layer
    def forward(self, x):
        x = torch.sigmoid(self.hidden(x)) # activation function for hidden layer
        x = self.out(x)
        return x

net = Net(n_feature=n_feature, n_hidden=n_hidden, n_output=n_output) # define the network
loss_func = torch.nn.MSELoss() #define loss function to calculate mean squared error

# define function to calculate a loss
def calcFitness(individual):
    set_weight(individual)
    out = net(train_x) # input x and predict based on x
    loss = loss_func(out, train_y)
    return loss.item(),

# define function to convert gray code to real number
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=-10+20*numasint/(maxnum-1)
    return numinrange

# define function to separate an individual into weights in real number
def separatevariables(v):
    sep = []
    for i in range (0,numOfBits*dimension,numOfBits):
        sep.append(chrom2real(v[i:i+numOfBits]))
    return sep

# define function to set weights to the neural network
def set_weight(ind):
    sep=separatevariables(ind)
    num = 0

```

```

for i in range (n_hidden):
    for j in range (n_feature):
        net.hidden.weight[i][j] = sep[num]
        num += 1
for i in range (n_output):
    for j in range (n_hidden):
        net.out.weight[i][j] = sep[num]
        num += 1
for i in range (n_hidden):
    net.hidden.bias[i] = sep[num]
    num += 1
for i in range (n_output):
    net.out.bias[i] = sep[num]
    num += 1

# register functions to the toolbox
toolbox = base.Toolbox()
# attr_bool function which returns 0 or 1 with equal probability
toolbox.register("attr_bool", random.randint, 0, 1)
# individual function to generate an individual consisting of numOfBits*dimension attr_bool elements
toolbox.register("individual", tools.initRepeat, creator.Individual,
    toolbox.attr_bool, numOfBits*dimension)
# population function to generate a list of individuals
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
# evaluate function which calls calcFitness
toolbox.register("evaluate", calcFitness)
# mate function using two-point crossover
toolbox.register("mate", tools.cxOnePoint)
# mutate function using flip bit mutation
toolbox.register("mutate", tools.mutFlipBit, indpb=flipProb)
# select function using tournament selection
toolbox.register("select", tools.selTournament, fit_attr='fitness')

# create an initial population of individuals
pop = toolbox.population(n=popSize)

# Evaluate the initial population
fitnesses = list(map(toolbox.evaluate, pop))
for ind, fit in zip(pop, fitnesses):
    ind.fitness.values = fit

# Variable keeping track of the number of generations
g = 0
# Initialise arrays to store mean squared errors of the training data and test data over the generations
loss_values_train = []
loss_values_test = []

# Begin the evolution
while g < iterations:
    # A new generation
    g = g + 1

    # Select the next generation individuals
    offspring = tools.selBest(pop, nElitists) + toolbox.select(pop, len(pop)-nElitists, 2)
    # Clone the selected individuals
    offspring = list(map(toolbox.clone, offspring))

    for child1, child2 in zip(offspring[::2], offspring[1::2]):

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

```

```
# fitness values of the children must be recalculated later
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)
        # fitness values of the children must be recalculated later
        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
pop[:] = offspring
```

```
# Select the best individual in the population.
best_ind = tools.selBest(pop, 1)[0]
m = best_ind.fitness.values[0]
loss_values_train.append(m)
```

```
# Set weights using the best individual
set_weight(best_ind)
```

```
# Test the best individual with the test data
out = net(test_x)
loss = loss_func(out, test_y)
loss_values_test.append(loss.item())
```

```
plt.title('Diagram 8: MSE of the Best Neural Network on Train Data')
plt.plot(np.array(loss_values_train), 'r')
plt.savefig('d8.png')
plt.show()
```

```
plt.title('Diagram 9: MSE of the Best Neural Network on Test Data')
plt.plot(np.array(loss_values_test), 'r')
plt.savefig('d9.png')
plt.show()
```


Appendix B
Modified Python code for lifetime learning
(Modified part is highlighted in yellow)

```
from sympy.combinatorics.graycode import gray_to_bin, bin_to_gray
```

```
...
```

```
def set_weight(ind):  
    with torch.no_grad():  
        sep=separatevariables(ind)  
        num = 0  
        for i in range (n_hidden):  
            for j in range (n_feature):  
                net.hidden.weight[i][j] = sep[num]  
                num += 1  
        for i in range (n_output):  
            for j in range (n_hidden):  
                net.out.weight[i][j] = sep[num]  
                num += 1  
        for i in range (n_hidden):  
            net.hidden.bias[i] = sep[num]  
            num += 1  
        for i in range (n_output):  
            net.out.bias[i] = sep[num]  
            num += 1
```

```
# define function to convert real number to gray code
```

```
def real2chrom(n):  
    if n > 10:  
        n = 10  
    if n < -10:  
        n = -10  
    numasint = int((n+10)*(maxnum-1)/20)  
    bin = format(numasint, 'b')  
    gray = bin_to_gray(bin)  
    gray = [int(i) for i in gray]  
    for i in range(numOfBits-len(gray)):  
        gray.insert(0,0)  
    return gray
```

```
# define function to get weights from the neural network
```

```
def get_weight():  
    sep = []  
    for i in range (n_hidden):  
        for j in range (n_feature):  
            sep.append(net.hidden.weight[i][j].item())  
    for i in range (n_output):  
        for j in range (n_hidden):  
            sep.append(net.out.weight[i][j].item())  
    for i in range (n_hidden):  
        sep.append(net.hidden.bias[i].item())  
    for i in range (n_output):  
        sep.append(net.out.bias[i].item())
```

```
ind = []  
for n in sep:  
    ind += real2chrom(n)  
return ind
```

```
...
```

```

# Lifetime
for i, ind in enumerate(offspring):
    set_weight(ind)
    optimizer = torch.optim.Rprop(net.parameters(), lr=0.02)
    for t in range(30):
        out = net(train_x) # input x and predict based on x
        loss = loss_func(out, train_y)
        optimizer.zero_grad() # clear gradients for next train
        loss.backward() # backpropagation, compute gradients
        optimizer.step() # apply gradients
    out = net(train_x) # input x and predict based on x
    loss = loss_func(out, train_y)
    offspring[i] = creator.Individual(get_weight())
    offspring[i].fitness.values = (loss.item(),)

```

...

```

# plt.title('Diagram 8: MSE of the Best Neural Network on Train Data')
# plt.plot(np.array(loss_values_train), 'r')
# plt.savefig('d8.png')
# plt.show()

```

```

# plt.title('Diagram 9: MSE of the Best Neural Network on Test Data')
# plt.plot(np.array(loss_values_test), 'r')
# plt.savefig('d9.png')
# plt.show()

```

```

plt.title('Diagram 10: MSE of the Best Neural Network on Train Data\nusing Lamarckian learning approach')
plt.plot(np.array(loss_values_train), 'r')
plt.savefig('d10.png')
plt.show()

```

```

plt.title('Diagram 11: MSE of the Best Neural Network on Test Data\nusing Lamarckian learning approach')
plt.plot(np.array(loss_values_test), 'r')
plt.savefig('d11.png')
plt.show()

```

Appendix C
Modified Python code for Baldwinian learning approach
(Modified part is highlighted in yellow)

```
# loss_values_train = []
# loss_values_test = []
loss_values_train_before = []
loss_values_test_before = []
loss_values_train_after = []
loss_values_test_after = []

...

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

# Select the best individual in the population.
best_ind = tools.selBest(pop, 1)[0]
m = best_ind.fitness.values[0]
loss_values_train_before.append(m)

# Set weights using the best individual
set_weight(best_ind)

# Test the best individual with the test data
out = net(test_x)
loss = loss_func(out, test_y)
loss_values_test_before.append(loss.item())

# Lifetime
for i, ind in enumerate(offspring):
    set_weight(ind)
    optimizer = torch.optim.Rprop(net.parameters(), lr=0.02)
    for t in range(30):
        out = net(train_x) # input x and predict based on x
        loss = loss_func(out, train_y)
        optimizer.zero_grad() # clear gradients for next train
        loss.backward() # backpropagation, compute gradients
        optimizer.step() # apply gradients
        out = net(train_x) # input x and predict based on x
        loss = loss_func(out, train_y)
#     offspring[i] = creator.Individual(get_weight())
#     offspring[i].fitness.values = (loss.item(),)
#     offspring[i].fitness.newind = get_weight()

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

# Select the best individual in the population.
best_ind = tools.selBest(pop, 1)[0]
m = best_ind.fitness.values[0]
loss_values_train_after.append(m)

# Set weights using the best individual
set_weight(best_ind.fitness.newind)

# Test the best individual with the test data
out = net(test_x)
loss = loss_func(out, test_y)
loss_values_test_after.append(loss.item())

...
```

```
# plt.title('Diagram 10: MSE of the Best Neural Network on Train Data\nusing Lamarckian learning approach')
# plt.plot(np.array(loss_values_train), 'r')
# plt.savefig('d10.png')
# plt.show()
```

```
# plt.title('Diagram 11: MSE of the Best Neural Network on Test Data\nusing Lamarckian learning approach')
# plt.plot(np.array(loss_values_test), 'r')
# plt.savefig('d11.png')
# plt.show()
```

```
plt.title('Diagram 12: MSE of the Best Neural Network on Train Data before lifetime learning')
plt.plot(np.array(loss_values_train_before), 'r')
plt.savefig('d12.png')
plt.show()
```

```
plt.title('Diagram 13: MSE of the Best Neural Network on Test Data before lifetime learning')
plt.plot(np.array(loss_values_test_before), 'r')
plt.savefig('d13.png')
plt.show()
```

```
plt.title('Diagram 14: MSE of the Best Neural Network on Train Data using Baldwinian learning approach')
plt.plot(np.array(loss_values_train_after), 'r')
plt.savefig('d14.png')
plt.show()
```

```
plt.title('Diagram 15: MSE of the Best Neural Network on Test Data using Baldwinian learning approach')
plt.plot(np.array(loss_values_test_after), 'r')
plt.savefig('d15.png')
plt.show()
```