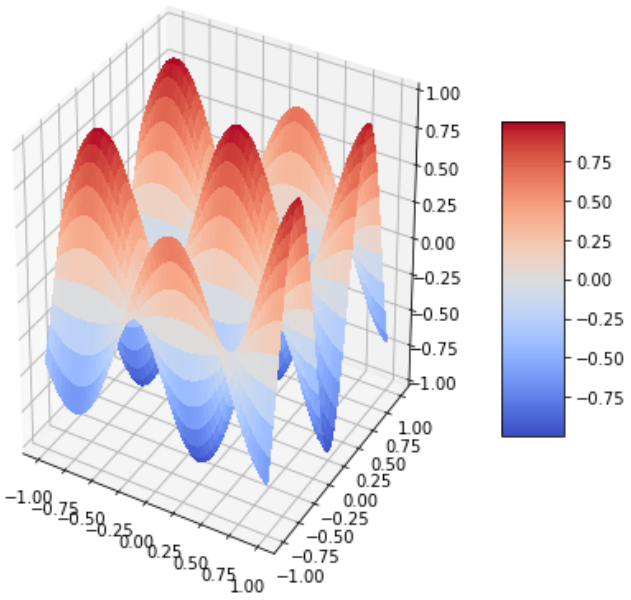
COM3013 Coursework

Arthur Butcher
Computer Science Student
University of Surrey
ab02038@surrey.ac.uk

Introduction

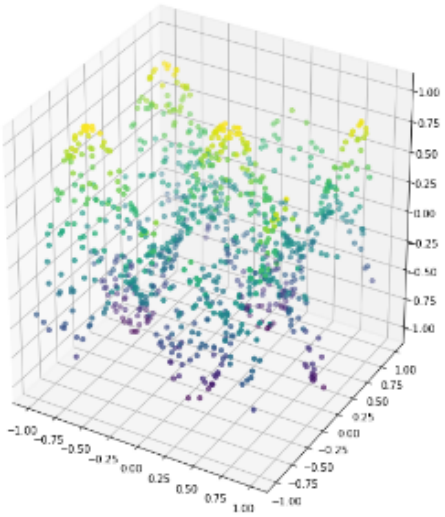
In this document I will be answering the questions detailed in the 'CW2_COM3013_2021' pdf file. For each question, I will be giving a brief explanation of what code I used as a base, what I modified to fit the question, and how I created the accompanying graphs.

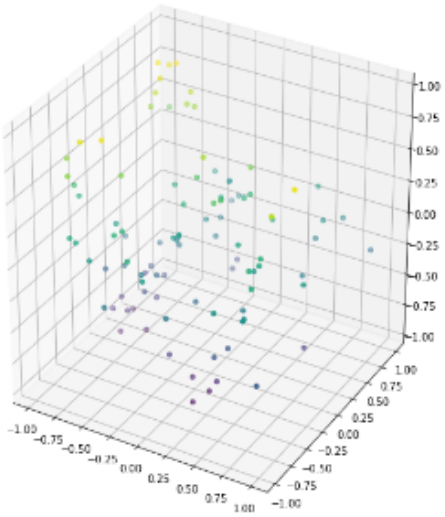
Question 1.1



To create this graph, I did a simple surface plot of x and y values between -1 and 1, and z values generated by feeding the x and y values into the given evaluation function.

Question 1.2





To create these graphs, I generated 1100 x1 and x2 values in the range of -1 and 1 and used the evaluation function to generate y values. I then assigned the first 1000 values of each list to training data, and the last 100 values to test data, ensuring that all data was stored in tensor form.

I then plotted both sets of data in a 3D space using scatter3D.

Question 1.3

```
class Net(torch.nn.Module):
    def __init__(self, n_feature, n_hidden, n_hidden2, n_output):
        super(Net, self).__init__()
        self.hidden = torch.nn.Linear(n_feature, n_hidden)
        self.hidden2 = torch.nn.Linear(n_hidden, n_hidden2)
        self.out = torch.nn.Linear(n_hidden2, n_output)

    def forward(self, x):
        x = torch.sigmoid(self.hidden(x))
        x = torch.sigmoid(self.hidden2(x))
        x = self.out(x)
        return x
```

```
feature=2
hidden=6
output=1
net = Net(n_feature = feature, n_hidden = hidden, n_hidden2 = hidden, n_output = output)
```

```
print(net)
```

```
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)
)
```

I created a neural network to the required specification and printed out its structure (see above).

Question 1.4

To create the 'weightsOutofNetwork' function, I first stored the weights of the hidden and out layers in list form. I then repeated this process with the bias data for each layer.

I then flattened the weights lists for each layer, enabling me to return the weights and bias in one cohesive output list.

To create the 'weightsIntoNetwork' function, I had to first divide up the weights into their respective sections according to how I structured the weights list in the output of 'weightsOutofNetwork'.

Once I had divided up the weight and bias for each layer, I parsed each list to a tensor, before loading them into the NN as parameters.

To test these functions, I pulled a list of weights from the network using the 'weightsOutofNetwork' function and changed the first three weights to the value of 1.0.

I then used the 'weightsIntoNetwork' to reinsert the weights, and then printed out the weights of the first hidden layer.

Results are below (they should be in bold, though it seems a little vague on Colab).

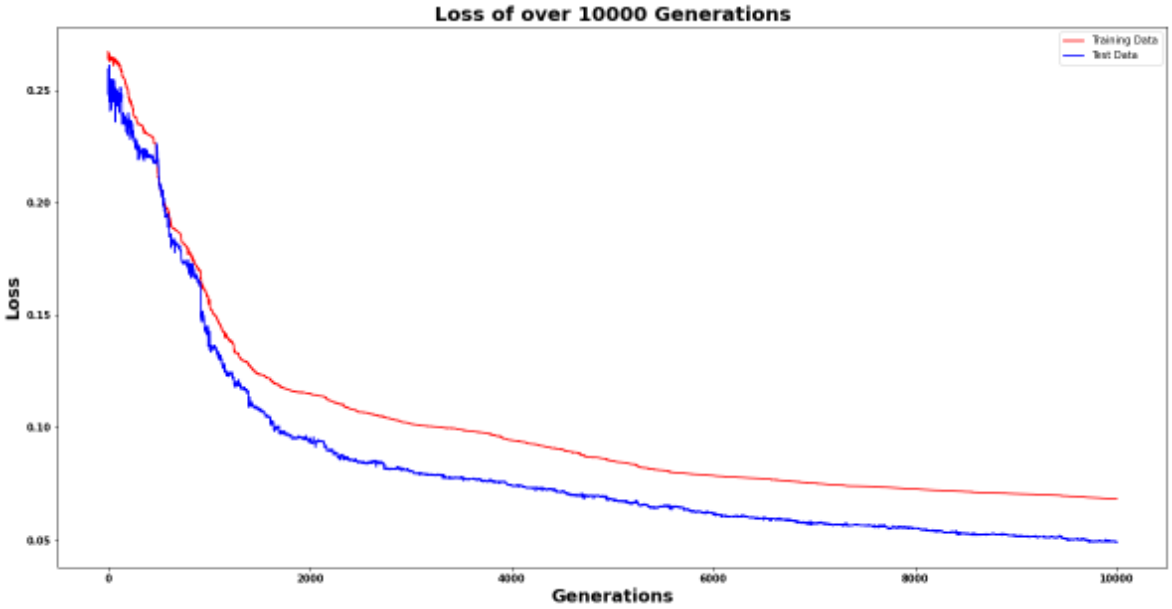
```
1.0
1.0
1.0
0.024339383468031883
0.15492574870586395
-0.5798636078834534
-0.60334712266922
-0.41088345646858215
0.18886053562164307
0.11381132155656815
-0.1431533694267273
0.2056599259376526
```

Question 1.5

For my hyperparameters, I initially started with a population of 200, however I quickly found that such a population was computationally demanding and, due to the low cross and mutate probability settings, quickly stagnated.

After changing my population to a more conservative 50, decided that due to the inaccurate nature of using genetic algorithms in this nature, having a high mutate and cross probability would be beneficial. Since this resulted in very large changes, I increased the number of elite individuals to 10% of the total population, in the hope that they would act as a mild anchor.

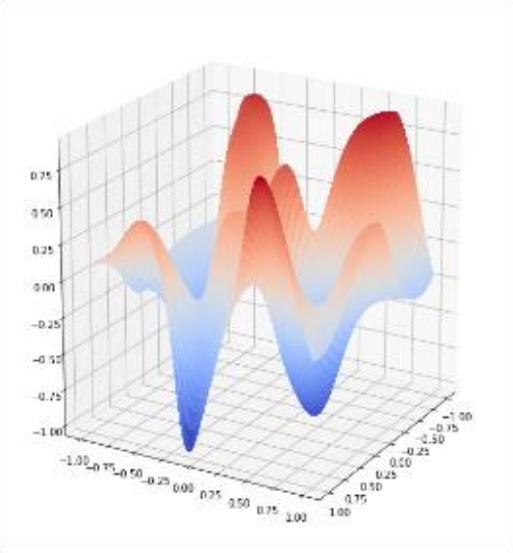
Using this approach, I set the probabilities to 1, and ran for 10,000 iterations, resulting in a loss value of 0.06.

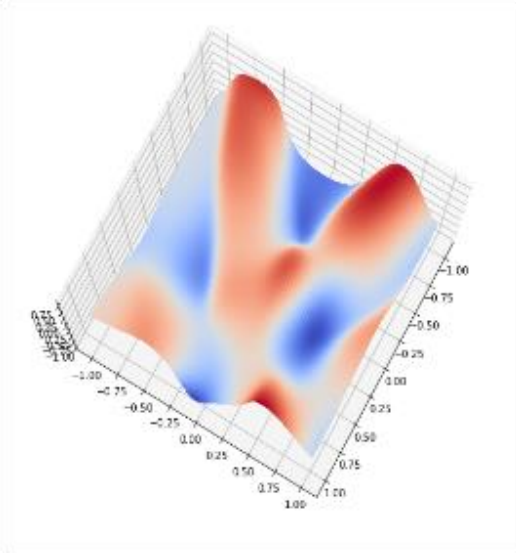


Question 1.6

During the 1.5 code, I calculated and appended the loss values to a list after every generation for the best individual, before loading the weights from that individual into the NN and doing the same for the test data.

I then created a 2D grid of values between -1 and 1 using a combination of .ravel() and np.c_[], feeding that grid into the NN from 1.5, and using the output of the net as the z values for my 3D surface plot.





Question 1.7

To create this tool, I defined a function called 'weights_to_chromosome_gray' which takes a simple list of weights. I then used a for loop to cycle through each number in the list, using np.clip() to ensure that all values were between -20 and 20, applying an inverse of the 'numinrange' function to

ensure that all values were scaled between 0 and maxnum, before formatting it as a binary number. I then using sympy's 'bin_to_gray' to convert the binary number to gray, before using another for loop to cycle through the gray coded number, adding each element to a list, and returning that list as the new chromosome.

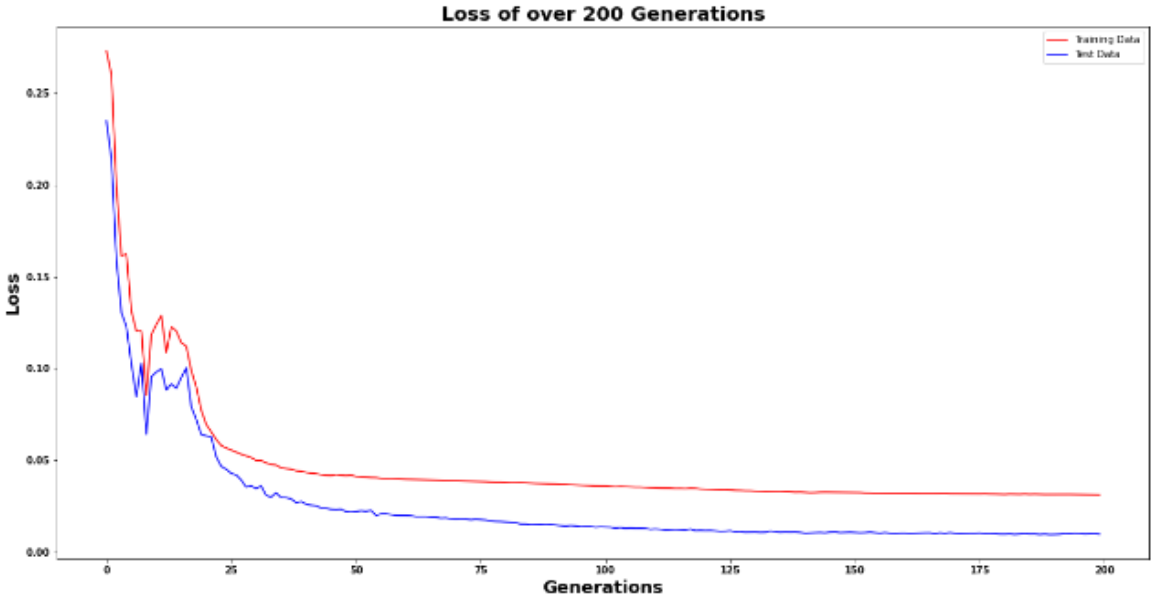
```
maxnum = 2**30
def weights_to_chromosome_gray(list_of_weights):
    chrom = []

    for number in list_of_weights:
      number_in_boundaries = np.clip(number, -20, 20)
      number_in_range = int(((number_in_boundaries + 20) * (maxnum-1)) / 40)
      binary_number = format(number_in_range, '030b')
      gray_coded_number = bin_to_gray(binary_number)
      for y in range(len(gray_coded_number)):
        chrom.append(int(gray_coded_number[y]))

    return chrom
```

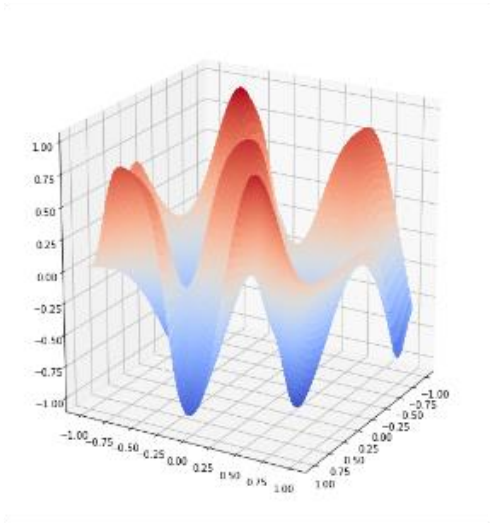
```
def test():
  n = weightsOutofNetwork()

  print(n[:5])
  print("\n")

  gray_bits = []
  gray_bits = weights_to_chromosome_gray(n)
  print(gray_bits[:5])
  print("\n")

  test_chrom_weights = seperatevariables(gray_bits)
  print(test_chrom_weights[:5])
  print("\n")

test()
```

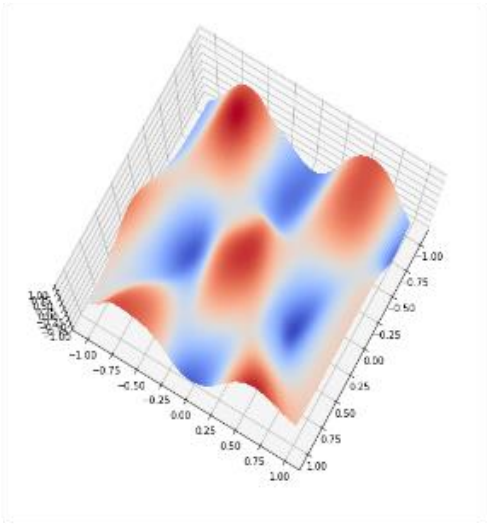
```
[-4.290653228759766, 2.5163779258728027, -3.946957588195801, -9.462818145751953, 2.2301690578460693]

[0, 1, 0, 1, 0]

[-4.2906532662833605, 2.5163778872381677, -3.946957610498142, -9.46281815829018, 2.2301690114943007]
```

Question 1.8

To implement rprop and Lamarckian learning approach, I implemented a for loop in every generation that added the weights of every offspring into the network. I then used the torch implementation of 'Rprop' to improve each individual 30 times, before pulling the final weights out and applying them back onto the corresponding individual and re-evaluating its fitness values.



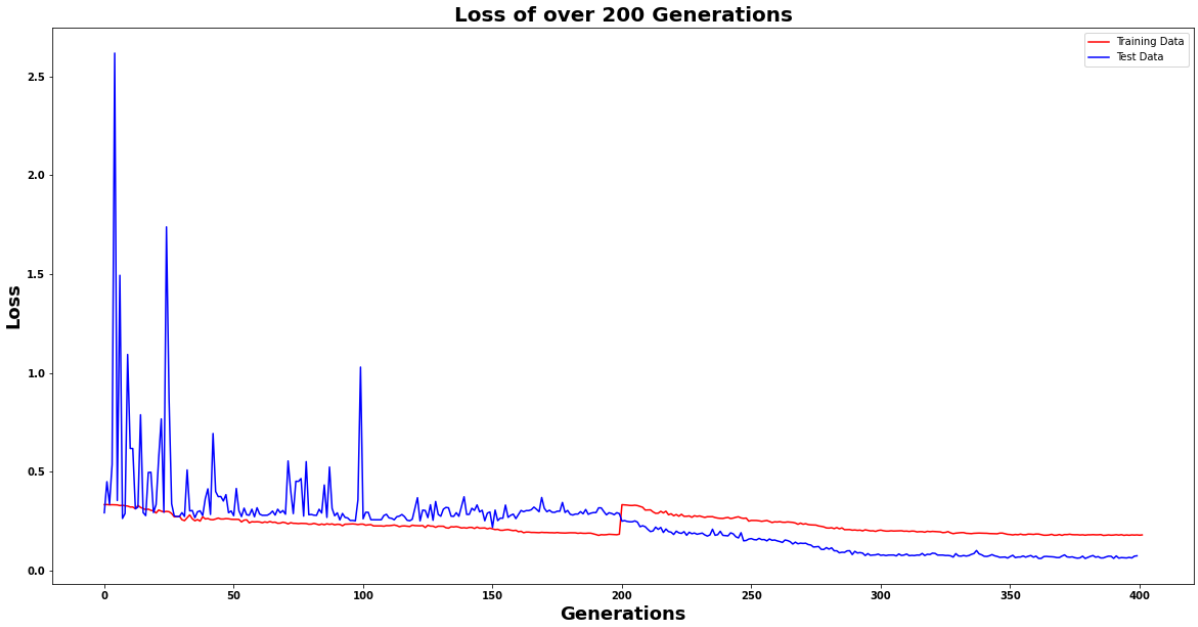


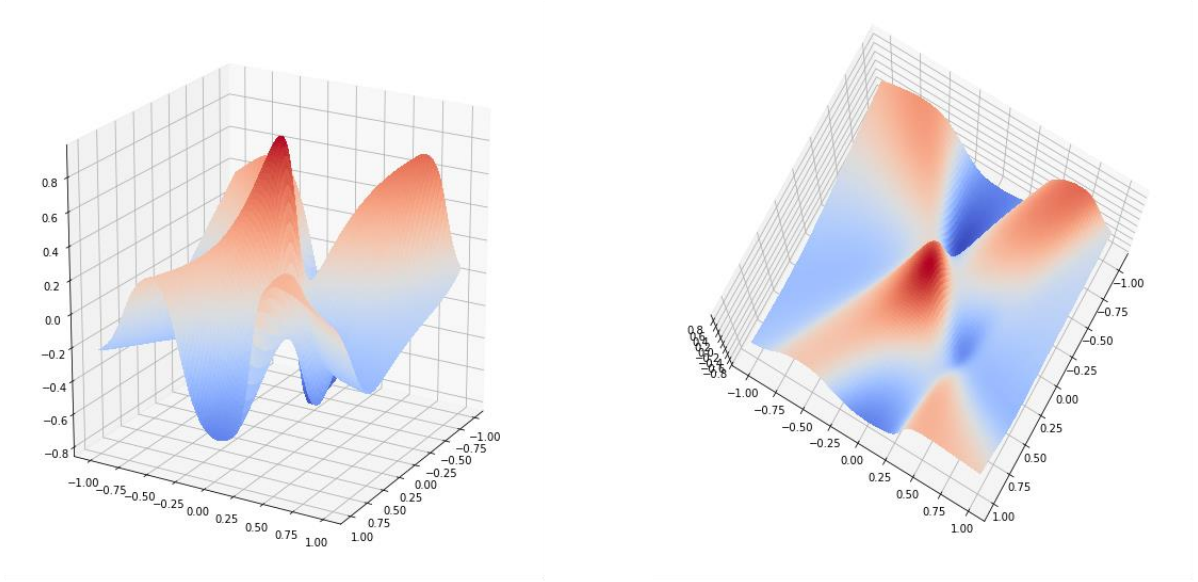


Question 1.9

To implement the Baldwinian learning approach, I created a copy of the offspring after their generation, feeding that copy into the 'Rprop' function instead of the initial offspring. Once the copy had been optimised, I set the fitness values of that copy to the original offspring, without transferring over the weights.

I again kept the hyperparameters the same as 1.5 and 1.8 to ensure a fair comparison.





The Baldwinian method of learning performed significantly worse than the Lamarckian method.

This is because the Lamarckian method passes the phenotypic changes such as weights and the fitness values that were developed during the 'Rprop' optimisation onto the following generation, resulting in changes to the genotypic data of that generation.

The Baldwinian method however only passes on the phenotypic fitness values, with the weights lost with that generation, resulting in the next generation having the same genotypic data as the one preceding it.

Appendix

For this coursework, I used Colab as my IDE of choice. As a result, my code is split up into distinct sections, as they complete distinct tasks.

Each section has been given a title, indicating what it is that they achieve.

For my code to work, it is recommended to run these sections in order as they go down, as some sections will not work if prior sections have not been run.

Please note that in the hyperparameters listed in the 'Common Code' section, the current iterations value is for 1.8 and 1.9. To repeat the graphs for 1.5, this will need to be changed to 10,000.

Imports

```
import random
import matplotlib.pyplot as plt
import matplotlib
import numpy as np
import torch
import torch.nn.functional as F
from torch import nn

from matplotlib.ticker import LinearLocator
from mpl_toolkits.mplot3d.axes3d import Axes3D
from sympy.combinatorics.graycode import gray_to_bin
from sympy.combinatorics.graycode import bin_to_gray
from deap import creator, base, tools
```

Evaluation Function

```
def eval_function(x1, x2):
    return np.sin(3.5 * x1 + 1) * np.cos(5.5 * x2)
```

Question 1.1 – 3D Surface Plot

```
xrange = np.linspace(-1.0, 1.0, 100)
yrange = np.linspace(-1.0, 1.0, 100)
X, Y = np.meshgrid(xrange, yrange)
Z = eval_function(X, Y)

fig = plt.figure(figsize=(7, 7))

ax = fig.add_subplot(1, 1, 1, projection='3d')

surf = ax.plot_surface(X, Y, Z, cmap=matplotlib.cm.coolwarm,
                       linewidth=0, antialiased=False)

ax.set_zlim(-1, 1)

# Add a color bar which maps values to colors.
fig.colorbar(surf, shrink=0.5, aspect=5)
```

```
plt.show()
```

Question 1.2 – Generate Values and Plot

```
x1_values = []
x2_values = []
y_values = []

for i in range(1100):
    x1 = random.uniform(-1, 1)
    x2 = random.uniform(-1, 1)
    y = eval_function(x1, x2)
    x1_values.append(x1)
    x2_values.append(x2)
    y_values.append(y)

x1_train = torch.as_tensor(x1_values[0:1000], dtype=torch.float)
x2_train = torch.as_tensor(x2_values[0:1000], dtype=torch.float)
y_train = torch.as_tensor(y_values[0:1000], dtype=torch.float)

x1_test = torch.as_tensor(x1_values[1000:1100], dtype=torch.float)
x2_test = torch.as_tensor(x2_values[1000:1100], dtype=torch.float)
y_test = torch.as_tensor(y_values[1000:1100], dtype=torch.float)

fig = plt.figure(figsize=(20, 10))

train = fig.add_subplot(1, 2, 1, projection='3d')
train.scatter3D(x1_train, x2_train, y_train, c=y_train)

test = fig.add_subplot(1, 2, 2, projection='3d')
test.scatter3D(x1_test, x2_test, y_test, c=y_test)

plt.show()
```

Data Pre-processing

```
x_train = [[x1] for x1 in x1_train]
x_train = [x_train[i] + [(x2_train[i])] for i in range(len(x2_train))]

x_test = [[x1] for x1 in x1_test]
x_test = [x_test[i] + [(x2_test[i])] for i in range(len(x2_test))]

y_train = [[y] for y in y_train]
y_test = [[y] for y in y_test]

x_train = np.array(x_train).astype(np.double)
y_train = np.array(y_train).astype(np.double)

x_test = np.array(x_test).astype(np.double)
y_test = np.array(y_test).astype(np.double)

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)
```

Question 1.3 – Create Neural Network

```
class Net(torch.nn.Module):
    def __init__(self, n_feature, n_hidden, n_hidden2, n_output):
        super(Net, self).__init__()
        self.hidden = torch.nn.Linear(n_feature, n_hidden)
        self.hidden2 = torch.nn.Linear(n_hidden, n_hidden2)
        self.out = torch.nn.Linear(n_hidden2, n_output)

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

feature=2
hidden=6
output=1
net = Net(n_feature = feature, n_hidden = hidden, n_hidden2 = hidden, n_output = output)
```

Question 1.4 – Weight Functions

```
def weightsOutofNetwork():
  hidden_weights_1 = net.hidden.weight.tolist()
  hidden_weights_2 = net.hidden2.weight.tolist()
  output_weights = net.out.weight.tolist()

  hidden_bias_1 = net.hidden.bias.tolist()
  hidden_bias_2 = net.hidden2.bias.tolist()
  output_bias = net.out.bias.tolist()

  flattened_hidden_weights_1 = [item for sublist in hidden_weights_1 for item in sublist]
  flattened_hidden_weights_2 = [item for sublist in hidden_weights_2 for item in sublist]
  flattened_output_weights = [item for sublist in output_weights for item in sublist]

  output = []

  output.extend(flattened_hidden_weights_1 + flattened_hidden_weights_2 +
flattened_output_weights + hidden_bias_1 + hidden_bias_2 + output_bias)

  return output

def weightsIntoNetwork(ind):
  hidden_weights_1 = ind[:12]
  hidden_weights_2 = ind[12:48]
  output_weights = ind[48:54]
  hidden_bias_1 = ind[54:60]
  hidden_bias_2 = ind[60:66]
```

```
  output_bias = ind[66:67]

  hidden_weights_1 = [hidden_weights_1[i:i+2] for i in range(0, len(hidden_weights_1), 2)]
  hidden_weights_2 = [hidden_weights_2[i:i+6] for i in range(0, len(hidden_weights_2), 6)]
  output_weights = [output_weights]

  hidden_weights_1_tensor = torch.FloatTensor(hidden_weights_1)
  hidden_weights_2_tensor = torch.FloatTensor(hidden_weights_2)
  output_weights_tensor = torch.FloatTensor(output_weights)

  net.hidden.weight = torch.nn.Parameter(hidden_weights_1_tensor)
  net.hidden2.weight = torch.nn.Parameter(hidden_weights_2_tensor)
  net.out.weight = torch.nn.Parameter(output_weights_tensor)

  hidden_bias_1_tensor = torch.FloatTensor(hidden_bias_1)
  hidden_bias_2_tensor = torch.FloatTensor(hidden_bias_2)
  output_bias_tensor = torch.FloatTensor(output_bias)

  net.hidden.bias = torch.nn.Parameter(hidden_bias_1_tensor)
  net.hidden2.bias = torch.nn.Parameter(hidden_bias_2_tensor)
  net.out.bias = torch.nn.Parameter(output_bias_tensor)

def test():
    weights = weightsOutofNetwork()

    weights[0:3] = [1, 1, 1]

    weightsIntoNetwork(weights)

    weights = weightsOutofNetwork()

    for i in range(0,12):
        if i < 3:
            bold_weight = "\033[1m" + str(weights[i]) + "\033[0m"
            print(bold_weight)
        else:
            weight = weights[i]
            print(weight)

test()
```

Common Code

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

popSize = 50
dimension = 67
numOfBits = 30
iterations = 10000 #(200 for 1.8 and 1.9)
dspInterval = 10
nElitists = 5
```

```
omega = 5
crossPoints = 2
crossProb = 0.7
flipProb = 1. / (dimension * numOfBits)
mutateprob = 0.7
maxnum = (2 ** numOfBits)-1

toolbox = base.Toolbox()

toolbox.register("attr_bool", random.randint, 0, 1)

toolbox.register("individual", tools.initRepeat, creator.Individual,
                 toolbox.attr_bool, numOfBits * dimension)

toolbox.register("population", tools.initRepeat, list, toolbox.individual)

loss_values = []
loss_func = torch.nn.MSELoss()

def chrom2real(c):
    indasstring = ''.join(map(str, c))
    degray = gray_to_bin(indasstring)
    numasint = int(degray, 2)
    numinrange = -20 + 40 * numasint / maxnum
    return numinrange

def seperatevariables(ind):

    def chunks(l, n):
      n = max(1, n)
      return (l[i:i+n] for i in range(0, len(l), n))

    return [chrom2real(dv) for dv in chunks(ind, numOfBits)]

def weights_to_chromosome_gray(list_of_weights):
    chrom = []

    for number in list_of_weights:
      number_in_boundaries = np.clip(number, -20, 20)
      number_in_range = int(((number_in_boundaries + 20) * (maxnum-1)) / 40)
      binary_number = format(number_in_range, '030b')
      gray_coded_number = bin_to_gray(binary_number)
      for y in range(len(gray_coded_number)):
        chrom.append(int(gray_coded_number[y]))

    return chrom

def initialisePopulation():
  pop = []
  for i in range(0, popSize):
    start_weights = [random.uniform(-1,1) for y in range(dimension)]
```

```
    individual = creator.Individual(weights_to_chromosome_gray(start_weights))
    pop.append(individual)
  return pop

toolbox.register("evaluate", eval_indv)

toolbox.register("mate", tools.cxUniform, indpb=crossProb)

toolbox.register("mutate", tools.mutFlipBit, indpb=flipProb)

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

loss_values_train = []
loss_values_test = []
ind_best_list = []
```

Question 1.5 – Genetic Algorithm

```
def eval_indv(indv):
  sep = seperatevariables(indv)
  weightsIntoNetwork(sep)
  out = net(x_train)
  loss = loss_func(out, y_train)
  loss_values.append(loss.item())
  return 1.0 / (loss.item() + 0.01),


with torch.no_grad():

  pop = initialisePopulation()

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

  for ind, fit in zip(pop, fitnesses):

      ind.fitness.values = fit

  fits = [ind.fitness.values[0] for ind in pop]

  g = 0

  while g < iterations:

      g = g + 1
      print("-- Generation %i --" % g)

      offspring = tools.selBest(pop, nElitists) + toolbox.select(pop,len(pop)-nElitists)

      offspring = list(map(toolbox.clone, offspring))

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

```
            toolbox.mate(child1, child2)

            del child1.fitness.values
            del child2.fitness.values

        for mutant in offspring:

            toolbox.mutate(mutant)
            del mutant.fitness.values

        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

        if g%dspInterval == 0:

            fits = [ind.fitness.values[0] for ind in pop]

            length = len(pop)
            mean = sum(fits) / length
            sum2 = sum(x*x for x in fits)
            std = abs(sum2 / length - mean**2)**0.5

            print("  Min %s" % min(fits))
            print("  Max %s" % max(fits))
            print("  Avg %s" % mean)
            print("  Std %s" % std)

        pop = offspring

        best_ind = tools.selBest(pop, 1)[0]

        loss_train = ((1.0/best_ind.fitness.values[0]) + 0.01)
        loss_values_train.append(loss_train)

        val_best_individual = seperatevariables(best_ind)

        weightsIntoNetwork(val_best_individual)

        test_output = net(x_test)

        test_loss = loss_func(test_output, y_test)
        loss_values_test.append(test_loss.item())

    print("-- End of (successful) evolution --")

    best_ind = tools.selBest(pop, 1)[0]
    print("Best individual LOSS is " , (1.0/best_ind.fitness.values[0]) + 0.01)

plt.figure(figsize=(20, 10))
```

```
plt.figure(figsize=(20, 10))

plt.title("Loss of over 10000 Generations", fontsize=20, fontweight='bold')
plt.xlabel("Generations", fontsize=18, fontweight='bold')
plt.ylabel("Loss", fontsize=18, fontweight='bold')
plt.xticks(fontweight='bold')
plt.yticks(fontweight='bold')

plt.plot(np.array(loss_values_train), 'r')
plt.plot(np.array(loss_values_test), 'b')
plt.legend(["Training Data", "Test Data"])
```

Question 1.6 – 3D Surface Plot

```
xrange = np.linspace(-1.0, 1.0, 100)
yrange = np.linspace(-1.0, 1.0, 100)

X, Y = np.meshgrid(xrange, yrange)

grid = np.c_[X.ravel(), Y.ravel()]

grid = torch.nn.Parameter(torch.from_numpy(grid)).type(torch.float)

output = net(grid)

output = output.detach().flatten()

output = output.reshape(100, 100)

output = np.array(output)

fig = plt.figure(figsize=(20,10))

ax = fig.add_subplot(1,2,1, projection='3d')

p = ax.plot_surface(X,Y,output, rstride=1, cstride=1, cmap=matplotlib.cm.coolwarm, linewidth=0,
antialiased=False,
                    zorder=0)
p = ax.view_init(20, 30)

ax = fig.add_subplot(1,2,2, projection='3d')

p = ax.plot_surface(X,Y,output, rstride=1, cstride=1, cmap=matplotlib.cm.coolwarm, linewidth=0,
antialiased=False,
                    zorder=0)
p = ax.view_init(80, 30)
```

Question 1.7 – Weights to Chromosome

```
maxnum = 2**30
def weights_to_chromosome_gray(list_of_weights):
    chrom = []
```

```
    for number in list_of_weights:
      number_in_boundaries = np.clip(number, -20, 20)
      number_in_range = int(((number_in_boundaries + 20) * (maxnum-1)) / 40)
      binary_number = format(number_in_range, '030b')
      gray_coded_number = bin_to_gray(binary_number)
      for y in range(len(gray_coded_number)):
        chrom.append(int(gray_coded_number[y]))

    return chrom

def test():
  n = weightsOutofNetwork()

  print(n[:5])
  print("\n")

  gray_bits = []
  gray_bits = weights_to_chromosome_gray(n)
  print(gray_bits[:5])
  print("\n")

  test_chrom_weights = seperatevariables(gray_bits)
  print(test_chrom_weights[:5])
  print("\n")

test()
```

Question 1.8 – Rprop Learning and Lamarckian Approach

```
def eval_indv(individual):
  sep = seperatevariables(individual)
  weightsIntoNetwork(sep)
  optimizer = torch.optim.Rprop(net.parameters(), lr=0.01)
  for i in range(30):
        out = net(x_train)
        loss_optimizer = loss_func(out, y_train)
        optimizer.zero_grad()
        loss_optimizer.backward()
        optimizer.step()
  new_weight = weightsOutofNetwork()
  individual[:] = weights_to_chromosome_gray(new_weight)
  out = net(x_train)
  loss = loss_func(out, y_train)
  loss_values.append(loss.item())
  return 1.0 / (loss.item() + 0.01),

pop = initialisePopulation()

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

for ind, fit in zip(pop, fitnesses):
```

```
    ind.fitness.values = fit

fits = [ind.fitness.values[0] for ind in pop]

g = 0

while g < iterations:

    g = g + 1
    print("-- Generation %i --" % g)

    offspring = tools.selBest(pop, nElitists) + toolbox.select(pop,len(pop)-nElitists)

    offspring = list(map(toolbox.clone, offspring))

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

        toolbox.mate(child1, child2, crossProb)

        del child1.fitness.values
        del child2.fitness.values

    for mutant in offspring:

       if random.random() < mutateprob:
          toolbox.mutate(mutant)
          del mutant.fitness.values

    for individual in offspring:

     weightsIntoNetwork(seperatevariables(individual))

     optimizer = torch.optim.Rprop(net.parameters(), lr=0.01)

     for i in range(30):
        out = net(x_train)
        loss_optimizer = loss_func(out, y_train)
        optimizer.zero_grad()
        loss_optimizer.backward()
        optimizer.step()

     new_weight = weightsOutofNetwork()
     individual[:] = weights_to_chromosome_gray(new_weight)
     individual.fitness.values = eval_indv(individual)

    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

    if g%dspInterval == 0:
```

```
        fits = [ind.fitness.values[0] for ind in pop]

        length = len(pop)
        mean = sum(fits) / length
        sum2 = sum(x*x for x in fits)
        std = abs(sum2 / length - mean**2)**0.5

        print("  Min %s" % min(fits))
        print("  Max %s" % max(fits))
        print("  Avg %s" % mean)
        print("  Std %s" % std)

    pop = offspring

    best_ind = tools.selBest(pop, 1)[0]
    loss_train = ((1.0/best_ind.fitness.values[0]) + 0.1)
    loss_values_train.append(loss_train)

    val_best_individual = seperatevariables(best_ind)

    weightsIntoNetwork(val_best_individual)

    test_output = net(x_test)
    test_loss = loss_func(test_output, y_test)
    loss_values_test.append(test_loss.item())

    ind_best_list = tools.selBest(pop, 1)[0]

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

sep = seperatevariables(best_ind)

weightsIntoNetwork(sep)
print("Best individual LOSS is " , (1.0/best_ind.fitness.values[0]) + 0.01)
```

Question 1.9 – Baldwinian Approach

```
def eval_indv(individual):
  sep = seperatevariables(individual)
  weightsIntoNetwork(sep)
  optimizer = torch.optim.Rprop(net.parameters(), lr=0.01)
  for i in range(30):
        out = net(x_train)
        loss_optimizer = loss_func(out, y_train)
        optimizer.zero_grad()
        loss_optimizer.backward()
        optimizer.step()
  new_weight = weightsOutofNetwork()
  individual.new_weights = new_weight
  out = net(x_train)
```

```
  loss = loss_func(out, y_train)
  loss_values.append(loss.item())
  return 1.0 / (loss.item() + 0.01),

pop = initialisePopulation()

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

for ind, fit in zip(pop, fitnesses):

    ind.fitness.values = fit

fits = [ind.fitness.values[0] for ind in pop]

g = 0

while g < iterations:

    g = g + 1
    print("-- Generation %i --" % g)

    offspring = tools.selBest(pop, nElitists) + toolbox.select(pop,len(pop)-nElitists)

    offspring = list(map(toolbox.clone, offspring))

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

        toolbox.mate(child1, child2, crossProb)

        del child1.fitness.values
        del child2.fitness.values

    for mutant in offspring:

       if random.random() < mutateprob:
          toolbox.mutate(mutant)
          del mutant.fitness.values

    for individual in offspring:

     weightsIntoNetwork(seperatevariables(individual))

     optimizer = torch.optim.Rprop(net.parameters(), lr=0.01)

     for i in range(30):
        out = net(x_train)
        loss_optimizer = loss_func(out, y_train)
        optimizer.zero_grad()
        loss_optimizer.backward()
        optimizer.step()
```

```
      new_weight = weightsOutofNetwork()
      individual[:] = weights_to_chromosome_gray(new_weight)
      individual.fitness.values = eval_indv(individual)

    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

    if g%dspInterval == 0:

        fits = [ind.fitness.values[0] for ind in pop]

        length = len(pop)
        mean = sum(fits) / length
        sum2 = sum(x*x for x in fits)
        std = abs(sum2 / length - mean**2)**0.5

        print("  Min %s" % min(fits))
        print("  Max %s" % max(fits))
        print("  Avg %s" % mean)
        print("  Std %s" % std)

    pop = offspring

    best_ind = tools.selBest(pop, 1)[0]
    loss_train = ((1.0/best_ind.fitness.values[0]) + 0.1)
    loss_values_train.append(loss_train)

    weightsIntoNetwork(best_ind.new_weights)

    test_output = net(x_test)
    test_loss = loss_func(test_output, y_test)
    loss_values_test.append(test_loss.item())

    ind_best_list = tools.selBest(pop, 1)[0]

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

sep = seperatevariables(best_ind)

weightsIntoNetwork(sep)
print("Best individual LOSS is " , (1.0/best_ind.fitness.values[0]) + 0.01)
```