# COM3013 Computational Intelligence Coursework 2

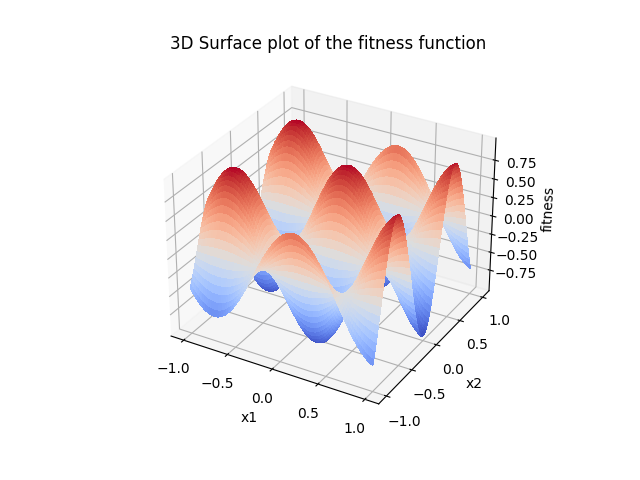
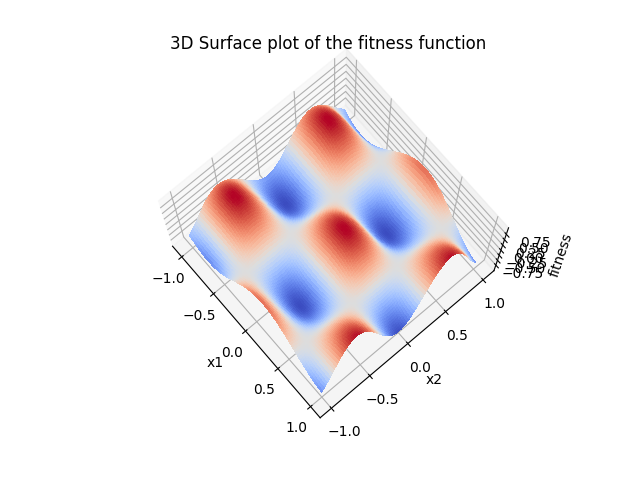
Shourya Raj Sharma | 6530100 | [ss02184@surrey.ac.uk](mailto:ss02184@surrey.ac.uk)

The main objective of the task was to train feed-forward multi-layer perceptron networks with two hidden layers to approximate the following function:

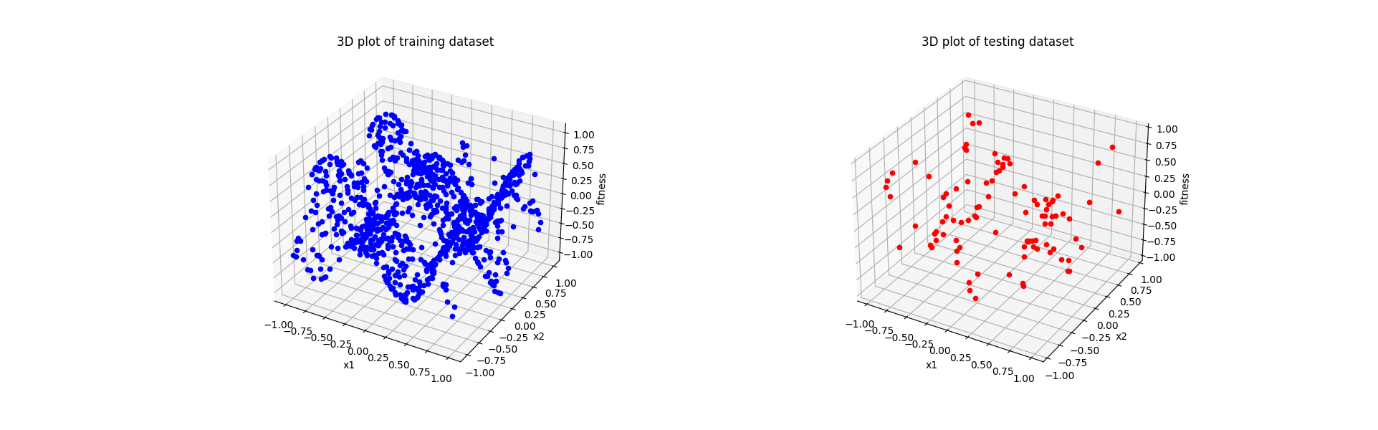
* 1. Visualize the function with a 3D surface plot, in the given range. Use of NumPy for numbers is recommended.

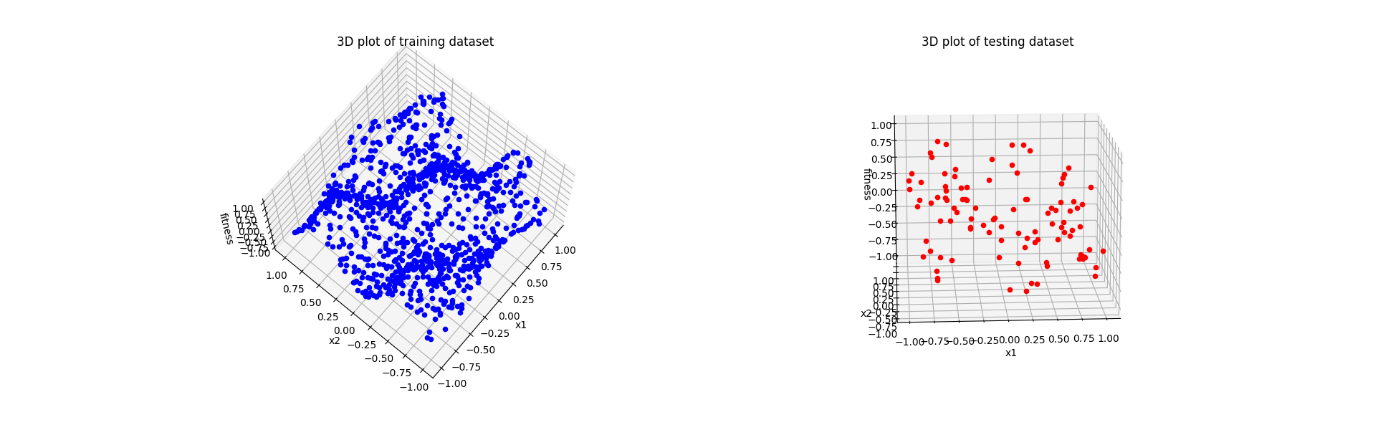
To do this, an initial fitness function matching the one above was defined, as well as a 3D surface plot function to utilise the fitness function.

1. # initial y function
2. def fitness(x1, x2):
3. f = np.sin(3.5\*x1 + 1)\*np.cos(5.5\*x2)
4. # f = 2 + 4.1\*(x1\*\*2) - 2.1\*(x1\*\*4) + (1/3)\*(x1\*\*6) + (x1\*x2) - 4\*((x2-0.05)\*\*2) + 4\*(x2\*\*4)
5. return (f)
7. # 3D surface plot of fitness function
8. def plot3D():
9. fig = plt.figure()
10. ax = fig.add\_subplot(projection='3d')
11. x1 = np.linspace(-1, 1, 100)
12. x2 = np.linspace(-1, 1, 100)
13. x1, x2 = np.meshgrid(x1, x2)
14. Z = fitness(x1, x2)
16. ax.plot\_surface(x1, x2, Z, rstride=1, cstride=1, cmap=cm.coolwarm, linewidth=0, antialiased=False, zorder=0)
17. ax.set\_xlabel('x1')
18. ax.set\_ylabel('x2')
19. ax.set\_zlabel('fitness')
20. plt.title('3D Surface plot of the fitness function')
21. plt.show()



* 1. Randomly generate 1100 samples for and within [-1, 1]. Calculate the corresponding y values for the 1100 samples. Use 1000 of them as the training dataset, and the other 100 samples as a test dataset. It is recommended to store your data as a tensor. Visualise the training and test data (two separate plots) in a three-dimensional graphic.





* 1. Assume that the neural network has six hidden neurons in each of the hidden layers and the network is fully connected. There is a threshold / bias connection for all hidden nodes and the output node. The activation function used is a sigmoid function in the hidden neurons, and a linear activation function in the output neuron. Write the code that creates the network.

1. class Net(torch.nn.Module):
2. # initialise two hidden layers and one output layer
3. def \_\_init\_\_(self, n\_feature, n\_hidden, n\_output):
4. super(Net, self).\_\_init\_\_()
5. self.hidden = torch.nn.Linear(n\_feature, n\_hidden) # hidden layer
6. self.hidden2 = torch.nn.Linear(n\_hidden, n\_hidden) # 2nd hidden layer
7. self.out = torch.nn.Linear(n\_hidden, n\_output) # output layer
9. def forward(self, x):
10. x = F.sigmoid(self.hidden(x)) # activation function for hidden layer
11. x = F.sigmoid(self.hidden2(x)) # activation function for hidden layer 2
12. x = self.out(x)
13. return x
15. device = torch.device("cuda:0" if torch.cuda.is\_available() else "cpu") # cuda is the GPU
16. # intitialize neural network to GPU
17. model = Net(n\_feature=2, n\_hidden=6, n\_output=1).to(device)
    1. Write the following two functions:

* weightsOutofNetwork: extracts all the weights of the network and puts them in one list; returns this list.

* weightsIntoNetwork: takes as input a list of all weights of the network and uses them to set the weights of the network

Write a test to check if the weights are retrieved and inserted in the network correctly. Show the following test results in your report: use weightsOutofNetwork to retrieve all weights; print out the first layer of weights (connecting input to hidden); change three of these weights; insert all weights back in the network using weightsIntoNetwork; use weightsOutofNetwork to retrieve all weights again and print out the first layer to check. Highlight the changed weights, e.g., in bold.

1. # extract weights out of neural network
2. # returns array of weights
3. def extractWeightsOutOfNetwork(nn):
4. outweights = []
5. for param in nn.parameters():
6. data = Tensor.cpu(param.data) # convert cpu to tensor to perform numpy operations
7. flattened = (np.array(data).flatten()).tolist()
8. outweights += flattened
9. return outweights
11. # inputs weights into neural network
12. # input: array of weights, network being modified
13. def inputWeightsIntoNetwork(arr, nn):
14. weights = np.asarray(arr)
15. nn.hidden.weight = torch.nn.Parameter(torch.from\_numpy(weights[:12].reshape(6, 2)).to(device))
16. nn.hidden.bias = torch.nn.Parameter(torch.from\_numpy(weights[12:18].reshape(1, 6)).to(device))
17. nn.hidden2.weight = torch.nn.Parameter(torch.from\_numpy(weights[18:54].reshape(6, 6)).to(device))
18. nn.hidden2.bias = torch.nn.Parameter(torch.from\_numpy(weights[54:60].reshape(1, 6)).to(device))
19. nn.out.weight = torch.nn.Parameter(torch.from\_numpy(weights[60:66].reshape(1, 6)).to(device))
20. nn.out.bias = torch.nn.Parameter(torch.from\_numpy(weights[66:67].reshape(1, 1)).to(device))
21. return nn

Using the extractWeightsOutOfNetwork method, it can be shown that the model initially contains the following weights:

[**-0.18409693241119385, 0.38351136445999146, 0.26195228099823, 0.3215683102607727, 0.0595211386680603, 0.5543354153633118, -0.36236831545829773, -0.22076770663261414, 0.45321351289749146, 0.39072638750076294, 0.6025742888450623, -0.6496381759643555,** 0.29979485273361206, -0.1291700005531311, -0.09141683578491211, -0.27062129974365234, -0.6663400530815125, 0.6079911589622498, -0.28171953558921814, 0.12552547454833984, -0.15029221773147583, 0.17436236143112183,

0.14592105150222778, 0.04953697323799133, -0.10457947850227356, 0.060629189014434814, 0.2759896516799927, -0.10034462809562683, -0.07143360376358032, 0.14576715230941772, 0.2394466996192932, -0.0881163477897644, -0.29311132431030273, -0.4063488245010376, -0.21109214425086975, -0.13730919361114502, -0.21595339477062225, 0.3568522334098816, -0.39930984377861023, 0.20799648761749268, -0.2570364475250244, 0.1814124584197998, -0.10060983896255493, -0.28118574619293213, 0.03469979763031006, 0.008986979722976685, 0.27488255500793457, -0.07337099313735962, 0.2845766544342041, -0.3381407856941223, -0.18659497797489166, 0.32653355598449707, -0.09253856539726257, 0.05812731385231018, 0.07438746094703674, 0.15162736177444458, 0.26534080505371094, 0.3142687678337097, 0.2742518186569214, -0.3983507454395294, 0.36885106563568115, -0.19457481801509857, 0.023155510425567627, 0.29285353422164917, 0.1048046350479126, -0.2843838930130005, 0.17574095726013184]

Of these, the weights from the first layer (connecting input to hidden) would be the first 12 (as seen in bold).

We can verify the validity of this first layer by also printing net.hidden.weight, which will print the first layer’s weights as follows:

Parameter containing:

tensor([[-0.1841, 0.3835],

[ 0.2620, 0.3216],

[ 0.0595, 0.5543],

[-0.3624, -0.2208],

[ 0.4532, 0.3907],

[ 0.6026, -0.6496]], device='cuda:0', requires\_grad=True)

The weights presented in the above tensor match the first 12 presented in the array as well. The weights 0.3835, 0.2620 and 0.3216 were then changed to 0.5, 0.4 and 0.3 respectively, with inputWeightsIntoNetwork then being used to modify the net to use the updated weights. The weights from this updated net can then be as follows, with the initial 12 weights once again highlighted in bold, and the changed values highlighted in italics.

[**-0.18409693241119385, *0.5, 0.4, 0.3*, 0.0595211386680603, 0.5543354153633118, -0.36236831545829773, -0.22076770663261414, 0.45321351289749146, 0.39072638750076294, 0.6025742888450623, -0.6496381759643555,** 0.29979485273361206, -0.1291700005531311, -0.09141683578491211, -0.27062129974365234, -0.6663400530815125, 0.6079911589622498, -0.28171953558921814, 0.12552547454833984, -0.15029221773147583, 0.17436236143112183, 0.14592105150222778, 0.04953697323799133, -0.10457947850227356, 0.060629189014434814, 0.2759896516799927, -0.10034462809562683, -0.07143360376358032, 0.14576715230941772, 0.2394466996192932, -0.0881163477897644, -0.29311132431030273, -0.4063488245010376, -0.21109214425086975, -0.13730919361114502, -0.21595339477062225, 0.3568522334098816, -0.39930984377861023, 0.20799648761749268, -0.2570364475250244, 0.1814124584197998, -0.10060983896255493, -0.28118574619293213, 0.03469979763031006, 0.008986979722976685, 0.27488255500793457, -0.07337099313735962, 0.2845766544342041, -0.3381407856941223, -0.18659497797489166, 0.32653355598449707, -0.09253856539726257, 0.05812731385231018, 0.07438746094703674, 0.15162736177444458, 0.26534080505371094, 0.3142687678337097, 0.2742518186569214, -0.3983507454395294, 0.36885106563568115, -0.19457481801509857, 0.023155510425567627, 0.29285353422164917, 0.1048046350479126, -0.2843838930130005, 0.17574095726013184]

The code for these changes is as follows:

1. print("================================================Creating Network =================================================")
2. # Create model with two hidden layers and 6 neurons in each
3. net = Net(n\_feature=2, n\_hidden=6, n\_output=1).to(device)
5. # Extracted weights from network
6. extractedWeights = extractWeightsOutOfNetwork(net)
8. # Test to see if new weights can be inputted
9. print("================================================Weights================================================")
10. print(extractedWeights)
11. print("================================================Weights from first layer================================================")
12. print(net.hidden.weight)
13. print(extractedWeights[:12])
14. print("================================================Changing 3 Weights==============================================")
16. # Change weights
17. extractedWeights[1] = 0.5
18. extractedWeights[2] = 0.4
19. extractedWeights[3] = 0.3
20. net = inputWeightsIntoNetwork(extractedWeights, net)
22. newWeights = extractWeightsOutOfNetwork(net)
23. print(newWeights)
    1. Use a binary coded genetic algorithm (with Gray coding) for optimising the weights of the neural network to fit the function, by minimising the mean squared error on the training dataset. Your evaluation function should extract weights from the chromosome and insert all weights in the network using weightsIntoNetwork, the loss of the network on the training data should then be used as the fitness of the individual. Use 30 bits for encoding each weight and limit weights to the range [-20,20]. It will help to initialise your individuals to a smaller range (e.g. [-1,1]). Experiment with hyperparameters to get a good result. Show your complete code, with hyperparameters clearly named at the top. Give a brief justification for your choice of hyperparameters. Show a plot of the training and test error across the generations.

The complete code can be found in the appendix, with relevant code sections being screenshotted and explained here.

With the way the neural network was composed, the number of weights and biases in total was 67. This meant that the number of dimensions for each individual that would be processed in the Genetic Algorithm (GA) would be 67. With the number of bits in each weight being 30, the numOfBits variable was set to 30, and the totalBits variable was set to 2010 (67 \* 30), as this would be the total number of bits present in an individual. The popSize was set to 50, as this is the standard minimum population size that is traditionally used in a binary coded genetic algorithm. Need to also talk about crossProb, flipProb, mutateProb and maxNum.

1. device = torch.device("cuda:0" if torch.cuda.is\_available() else "cpu") # cuda is the GPU
3. loss\_func = torch.nn.MSELoss()
4. totalBits = 67\*30 # [(input size + 1) \* numOfHiddenNeurons + (numOfHiddenNeurons + 1) \* output] \* numOfBitsPerWeight
5. popSize = 50
6. dimension = 67 # number of dimensions
7. numOfBits = 30 # number of bits per weight
8. numOfGenerations = 30 # number of generations to run
9. nElitists = 1
10. crossPoints = 2 #variable not used. instead tools.cxTwoPoint
11. crossProb = 0.6
12. flipProb = 1. / (dimension \* numOfBits) #bit mutate prob
13. mutateprob = .1 #mutation prob
14. maxnum = 2\*\*numOfBits-1
16. creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
17. creator.create("Individual", list, fitness=creator.FitnessMin)
19. # intitialize neural network to GPU
20. model = Net(n\_feature=2, n\_hidden=6, n\_output=1).to(device)
21. torch.save(model, 'model.pt') # save model to folder
22. toolbox = base.Toolbox()

Once these initial hyperparameters had been defined, the

* 1. Show a 3D surface plot of the function implemented by the neural network across the range [-1,1]. To do this compute the network’s output for a grid of values uniformly covering the range. i.e. similar to 1.1 except using network output instead of the mathematical function.
  2. Write a function which takes a list of weights (output from weightsOutofNetwork) and returns a chromosome (i.e. a long list of binary bits, in Gray coding). The function should check if weights need to be pushed back into the range [-20,20]. Write a test to test your function: test if weights can be put into a chromosome, and retrieved, and turned back into weights. Show the results of your test for a small number of weights. Note that numbers retrieved will not be exactly the same as those input because of a loss of accuracy in conversion.
  3. Embed the Rprop learning in the genetic algorithm as a local search method (lifetime learning). Use the Lamarckian learning approach, i.e., the weight changes in the lifetime learning are encoded back to the genotype. Implement 30 iterations of local search in each generation. Implement the above memetic algorithm by extending the code above. Explain hyperparameter choices. Show a plot of the training and test error across the generations, and also a 3D surface plot of the function implemented by the neural network across the range [-1,1].
  4. Implement the Baldwinian learning approach to replace the Lamarckian approach in Question 1.8 and plot the results. Show a plot of the training and test error across the generations, and also a 3D surface plot of the function implemented by the neural network across the range [-1,1]. [6 marks]. Analyse the difference in the results compared to those obtained in Question 1.8 and discuss possible reasons [3 marks]

## Appendix

1. import random
2. import numpy as np
3. import matplotlib.pyplot as plt
4. from matplotlib import cm
5. import torch
6. from torch.functional import Tensor
7. import torch.nn.functional as F
8. from sympy.combinatorics.graycode import GrayCode
9. from sympy.combinatorics.graycode import gray\_to\_bin, bin\_to\_gray
10. from deap import creator, base, tools, algorithms
11. import copy
13. class Net(torch.nn.Module):
14. # initialise two hidden layers and one output layer
15. def \_\_init\_\_(self, n\_feature, n\_hidden, n\_output):
16. super(Net, self).\_\_init\_\_()
17. self.hidden = torch.nn.Linear(n\_feature, n\_hidden)
18. self.hidden2 = torch.nn.Linear(n\_hidden, n\_hidden) # hidden layer
19. self.out = torch.nn.Linear(n\_hidden, n\_output) # output layer
21. # connect up the layers: the input passes through the hidden, then the sigmoid, then the output layer
22. def forward(self, x):
23. x = F.sigmoid(self.hidden(x)) # activation function for hidden layer
24. x = F.sigmoid(self.hidden2(x)) # activation function for hidden layer 2
25. x = self.out(x)
26. return x
28. device = torch.device("cuda:0" if torch.cuda.is\_available() else "cpu") # cuda is the GPU
30. loss\_func = torch.nn.MSELoss()
31. totalBits = 67\*30 # [(input size + 1) \* numOfHiddenNeurons + (numOfHiddenNeurons + 1) \* output] \* numOfBitsPerWeight
32. popSize = 50
33. dimension = 67 # number of dimensions
34. numOfBits = 30 # number of bits per weight
35. numOfGenerations = 30 # number of generations to run
36. nElitists = 1
37. crossPoints = 2 #variable not used. instead tools.cxTwoPoint
38. crossProb = 0.6
39. flipProb = 1. / (dimension \* numOfBits) #bit mutate prob
40. mutateprob = .1 #mutation prob
41. maxnum = 2\*\*numOfBits-1
43. creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
44. creator.create("Individual", list, fitness=creator.FitnessMin)
46. # intitialize neural network to GPU
47. model = Net(n\_feature=2, n\_hidden=6, n\_output=1).to(device)
48. torch.save(model, 'model.pt') # save model to folder
49. toolbox = base.Toolbox()
51. # initial y function
52. def fitness(x1, x2):
53. f = np.sin(3.5\*x1 + 1)\*np.cos(5.5\*x2)
54. # f = 2 + 4.1\*(x1\*\*2) - 2.1\*(x1\*\*4) + (1/3)\*(x1\*\*6) + (x1\*x2) - 4\*((x2-0.05)\*\*2) + 4\*(x2\*\*4)
55. return (f)
57. # 3D surface plot of fitness function
58. def plot3D():
59. fig = plt.figure()
60. ax = fig.add\_subplot(projection='3d')
61. x1 = np.linspace(-1, 1, 100)
62. x2 = np.linspace(-1, 1, 100)
63. x1, x2 = np.meshgrid(x1, x2)
64. Z = fitness(x1, x2)
66. ax.plot\_surface(x1, x2, Z, rstride=1, cstride=1, cmap=cm.coolwarm, linewidth=0, antialiased=False, zorder=0)
67. ax.set\_xlabel('x1')
68. ax.set\_ylabel('x2')
69. ax.set\_zlabel('fitness')
70. plt.title('3D Surface plot of the fitness function')
71. plt.show()
73. # generates 1100 samples of x1 and x2 values
74. # returns x1, x2, and y (output from the fitness function for every x1 and x2)
75. def generate1100SamplesforX1andX2():
76. X1 = np.random.uniform(-1, 1, 1100)
77. X2 = np.random.uniform(-1, 1, 1100)
78. Y = fitness(X1, X2)
79. return X1, X2, Y
81. # splits x1 and x2 into training and testing datasets
82. # returns tuple of training and testing data
83. def splitDataInto1000TrainingAnd100TestingSamples(X1, X2, Y):
85. # Training dataset
86. # transforms x1 and x2 into 2d array of tensors
87. X\_train = torch.from\_numpy(np.vstack((X1[:1000], X2[:1000])).T).to(device)
88. # transforms y (output) into a tensor
89. Y\_train = torch.from\_numpy(Y[:1000]).to(device)
91. # Testing dataset
92. # transforms x1 and x2 into a 2d array of tensors
93. X\_test = torch.from\_numpy(np.vstack((X1[1000:], X2[1000:])).T).to(device)
94. # transforms y (output) into a tensor
95. Y\_test = torch.from\_numpy(Y[1000:]).to(device)
96. training = (X\_train, Y\_train)
97. testing = (X\_test, Y\_test)
99. return training, testing
101. # creates 3D scatter plot of the training and testing dataset
102. def visualizeTrainingandTesting(training, testing):
103. X\_train, Y\_train = training
104. X\_test, Y\_test = testing
105. fig = plt.figure(figsize=(26, 6))
106. ax = fig.add\_subplot(1,2,1, projection='3d')
107. for i in range(len(X\_train)):
108. ax.scatter3D(X\_train[i][0], X\_train[i][1], Y\_train[i], c='blue', marker='o')
109. # plt.pause(0.000001)
110. ax.set\_xlabel('x1')
111. ax.set\_ylabel('x2')
112. ax.set\_zlabel('fitness')
113. ax.view\_init(80, 30)
114. plt.title('3D plot of training dataset')
115. # ax.scatter(X1\_test, X2\_test, Y\_test, c='black', marker='o')
116. ax = fig.add\_subplot(1,2,2, projection='3d')
117. for i in range(len(X\_test)):
118. ax.scatter3D(X\_test[i][0], X\_test[i][1], Y\_test[i], c='red', marker='o')
119. # plt.pause(0.000001)
120. ax.set\_xlabel('x1')
121. ax.set\_ylabel('x2')
122. ax.set\_zlabel('fitness')
123. plt.title('3D plot of testing dataset')
124. plt.show()
126. # extract weights out of neural network
127. # returns array of weights
128. def extractWeightsOutOfNetwork(nn):
129. outweights = []
130. for param in nn.parameters():
131. data = Tensor.cpu(param.data) # convert cpu to tensor to perform numpy operations
132. flattened = (np.array(data).flatten()).tolist()
133. outweights += flattened
134. return outweights
136. # inputs weights into neural network
137. # input: array of weights, network being modified
138. def inputWeightsIntoNetwork(arr, nn):
139. weights = np.asarray(arr)
140. nn.hidden.weight = torch.nn.Parameter(torch.from\_numpy(weights[:12].reshape(6, 2)).to(device))
141. nn.hidden.bias = torch.nn.Parameter(torch.from\_numpy(weights[12:18].reshape(1, 6)).to(device))
142. nn.hidden2.weight = torch.nn.Parameter(torch.from\_numpy(weights[18:54].reshape(6, 6)).to(device))
143. nn.hidden2.bias = torch.nn.Parameter(torch.from\_numpy(weights[54:60].reshape(1, 6)).to(device))
144. nn.out.weight = torch.nn.Parameter(torch.from\_numpy(weights[60:66].reshape(1, 6)).to(device))
145. nn.out.bias = torch.nn.Parameter(torch.from\_numpy(weights[66:67].reshape(1, 1)).to(device))
146. return nn
148. # creates 3D surface plot of the fitness function
149. def neuralNetwork3DSurfacePlot():
150. X = np.linspace(-1, 1, 100)
151. Y = np.linspace(-1, 1, 100)
152. combined = torch.from\_numpy(np.vstack([X, Y]).T).to(device)
154. X, Y = np.meshgrid(X, Y)
155. Z = model(combined)
156. Z = Tensor.cpu(Z).detach()
157. fig = plt.figure()
158. ax = fig.add\_subplot(projection='3d')
159. ax.plot\_surface(X, Y, Z, rstride=1, cstride=1, cmap=cm.coolwarm, linewidth=0, antialiased=False, zorder=0)
160. ax.set\_xlabel('x1')
161. ax.set\_ylabel('x2')
162. ax.set\_zlabel('y')
163. plt.title('3D surface plot of fitness function implemented by the neural network')
164. plt.show()
166. def plot(maxArr):
167. print("plotting................................................................")
168. # maxArr = maxArr.detach().numpy()
169. gen = []
170. for i in range(numOfGenerations):
171. gen.append(i)
173. plt.plot(gen, maxArr, label="Best Individual")
174. # plt.plot(gen, avgArr, label="Average Individual")
175. plt.legend()
176. plt.xlabel("Generation")
177. plt.ylabel("Fitness")
178. plt.title("Fitness of best individual from the testing dataset across the generations")
179. plt.show()
181. # Convert chromosome to real number
182. # input: list binary 1,0 of length numOfBits representing number using gray coding
183. # output: real value
184. def chrom2real(c):
185. indasstring=''.join(map(str, c))
186. degray=gray\_to\_bin(indasstring)
187. numasint=int(degray, 2) # convert to int from base 2 list
188. numinrange=-20+40\*numasint/maxnum
189. return numinrange
191. # Convert array of weights to chromosome
192. # input: array of weights
193. # output: list binary 1,0 of length numOfBits representing number using gray coding
194. def real2Chrom(weights):
195. output = [] # create output array to hold individual
196. for i in range(len(weights)): # clamp weights between -20 and 20
197. if weights[i] < -20:
198. weights[i] = -20
199. elif weights[i] > 20:
200. weights[i] = 20
201. numasint = (weights[i] + 20)\*maxnum/40 # convert weight to integer
202. binary = bin(int(numasint))[2:].zfill(30) # convert to binary
203. gray = bin\_to\_gray(binary) # convert to gray-coded digit
204. output.append(gray)
205. output = list(''.join(output))
206. for i in range(len(output)):
207. output[i] = int(output[i])
208. return output
210. # Return loss value of the neural network on the current individual
211. # input: list binary 1,0 of length numOfBits representing number using gray coding
212. # output: loss value of individual
213. def getWeightFitness(individual):
214. individual = np.array(individual) # convert
215. reshaped = individual.reshape(67, 30) # reshape to array of weights
216. weights = []
217. for ind in reshaped:
218. ind = chrom2real(ind)
219. if ind < -20:
220. ind = -20
221. elif ind > 20:
222. ind = 20
223. weights.append(ind)
224. weights = np.asarray(weights) # create array of weights as real numbers
225. inputWeightsIntoNetwork(weights, model) # update model with new weights
226. out = model(training[0]) # input training data and predict network output based on data
227. loss = loss\_func(out, training[1]) # # compare output with labeled data
228. return loss.item(),
230. # Implements lamarckian learning local search on each individual
231. # input: list binary 1,0 of length numOfBits representing number using gray coding
232. # output: updated list binary 1,0 of length numOfBits representing number using gray coding
233. def lamarckianOptimize(individual):
234. individual = np.array(individual) # convert individual to numpy array
235. reshaped = individual.reshape(67, 30) # reshape individual to 67x30
236. weights = []
237. for ind in reshaped:
238. ind = chrom2real(ind) # convert weights to real number
239. weights.append(ind)
240. weights = np.asarray(weights)
241. # print("original weights: ", extractWeightsOutOfNetwork(model))
242. inputWeightsIntoNetwork(weights, model) # input weights into network
243. y = model(training[0])
244. originalLoss = loss\_func(y, training[1]) # calculate loss
245. i = 0
246. loss = 0
247. updatedWeights = extractWeightsOutOfNetwork(model) # extract weights from
248. optimizer = torch.optim.Rprop(model.parameters(), lr=0.005) # intialize RProp optimizer
249. while i < 30: # run 30 iterations of optimization to find better weights
250. out = model(training[0]) # input training data and predict network output based on data
251. loss = loss\_func(out, training[1]) # compare output with labeled data
252. optimizer.zero\_grad() # clear gradients for next train
253. loss.backward() # backpropagation, compute gradients
254. optimizer.step() # apply gradients
255. if originalLoss > loss: # update weights if loss result is better
256. updatedWeights = extractWeightsOutOfNetwork(model)
257. originalLoss = loss
258. i += 1
259. newInd = real2Chrom(updatedWeights) # convert weights to gray coded individual
260. return newInd
262. # Implements baldwinian learning local search on each individual
263. # input: list binary 1,0 of length numOfBits representing number using gray coding
264. # output: tuple loss value of optimized individual
265. def baldwinianLearning(individual):
266. individual = np.array(individual) # convert individual to numpy array
267. reshaped = individual.reshape(67, 30) # reshape individual to 67x30
268. weights = []
269. for ind in reshaped:
270. ind = chrom2real(ind) # convert weights to real number
271. weights.append(ind)
272. weights = np.asarray(weights)
273. inputWeightsIntoNetwork(weights, model) # input weights into network
274. i = 0
275. loss = 0
276. y = model(training[0])
277. originalLoss = loss\_func(y, training[1]) # calculate loss
278. optimizer = torch.optim.Rprop(model.parameters(), lr=0.005) # intialize RProp optimizer
279. while i < 30:
280. optimizer.zero\_grad() # clear gradients for next train
281. out = model(training[0]) # input training data and predict network output based on data
282. loss = loss\_func(out, training[1]) # compare output with labeled data
283. loss.backward() # backpropagation, compute gradients
284. optimizer.step() # apply gradients
285. i += 1
286. if originalLoss > loss: # update original loss if loss result is better
287. originalLoss = loss
288. return originalLoss.item(),
290. def plotLearning(lamarckian, baldwinian):
291. print("plotting................................................................")
292. # maxArr = maxArr.detach().numpy()
293. gen = []
294. for i in range(numOfGenerations):
295. gen.append(i)
297. plt.plot(gen, lamarckian, label="Lamarckian Learning", color="blue")
298. plt.plot(gen, baldwinian, label="Baldwinian Learning", color="green")
299. plt.legend()
300. plt.xlabel("Generation")
301. plt.ylabel("Fitness")
302. plt.title("Fitness of best individual from across the generations")
303. plt.show()
305. # Generates dataset
306. print("================================================Dataset================================================")
307. dataset = generate1100SamplesforX1andX2()
309. # Splits dataset into training and testing dataset
310. print("================================================Splitting Dataset================================================")
311. training, testing = splitDataInto1000TrainingAnd100TestingSamples(\*dataset)
312. # print(type(training[0]))
313. # Visualize training and testing dataset
314. # visualizeTrainingandTesting(training, testing)
316. print("================================================Creating Network =================================================")
317. # Create model with two hidden layers and 6 neurons in each
318. net = Net(n\_feature=2, n\_hidden=6, n\_output=1).to(device)
320. # Extracted weights from network
321. extractedWeights = extractWeightsOutOfNetwork(net)
323. # Test to see if new weights can be inputted
324. print("================================================Weights================================================")
325. print(extractedWeights)
326. print("================================================Weights from first layer================================================")
327. print(net.hidden.weight)
328. print(extractedWeights[:12])
329. print("================================================Changing 3 Weights==============================================")
331. # Change weights
332. extractedWeights[1] = 0.5
333. extractedWeights[2] = 0.4
334. extractedWeights[3] = 0.3
335. net = inputWeightsIntoNetwork(extractedWeights, net)
337. newWeights = extractWeightsOutOfNetwork(net)
338. print(newWeights)
340. print("================================================Running GA================================")
342. # Attribute generator
343. # define 'attr\_bool' to be an attribute ('gene')
344. # which corresponds to integers sampled uniformly
345. # from the range [0,1] (i.e. 0 or 1 with equal
346. # probability)
347. toolbox.register("attr\_bool", random.randint, 0, 1)
349. # Structure initializers
350. # define 'individual' to be an individual
351. # consisting of numOfBits\*dimension 'attr\_bool' elements ('genes')
352. toolbox.register("individual", tools.initRepeat, creator.Individual,
353. toolbox.attr\_bool, totalBits)
355. # define the population to be a list of individuals
356. toolbox.register("population", tools.initRepeat, list, toolbox.individual)
357. #----------
358. # Operator registration
359. #----------
360. # register the goal / fitness function
361. toolbox.register("evaluate", getWeightFitness)
363. # register the crossover operator
364. toolbox.register("mate", tools.cxTwoPoint)
366. # register a mutation operator with a probability to
367. # flip each attribute/gene of 0.05
368. toolbox.register("mutate", tools.mutFlipBit, indpb=flipProb)
370. # operator for selecting individuals for breeding the next
371. # generation: each individual of the current generation
372. # is replaced by the 'fittest' (best) of three individuals
373. # drawn randomly from the current generation.
374. toolbox.register("select", tools.selBest, fit\_attr='fitness')
376. arr = []
377. bestInitial = 0
379. # Run GA optimizer for numOfGenerations generations
380. # inputs: x (population), boolean to decide local search method
381. def main(x, boolean):
382. fitArr = []
383. # print(pop)
384. #random.seed(64)
386. # create an initial population of individuals (where
387. # each individual is a list of integers)
388. pop = copy.deepcopy(x)
389. # for individ in pop:
390. # sep=separatevariables(individ)
391. # print(sep[0],sep[1])
393. # Evaluate the entire population
394. fitnesses = list(map(toolbox.evaluate, pop))
395. #print(fitnesses)
396. for ind, fit in zip(pop, fitnesses):
397. #print(ind, fit)
398. ind.fitness.values = fit
400. print(" Evaluated %i individuals" % len(pop))
401. # Extracting all the fitnesses of
402. fits = [ind.fitness.values[0] for ind in pop]
403. bestInitial = tools.selBest(pop, 1)[0].fitness.values[0]
404. # Variable keeping track of the number of generations
405. g = 0
406. # print(fitnesses)
407. # Begin the evolution
408. while g < numOfGenerations:
409. # A new generation
411. # Run lamarckian learning on each generation
412. if boolean is True:
413. for ind in pop:
414. ind = lamarckianOptimize(ind)
415. fitnesses = list(map(toolbox.evaluate, pop))
417. # Run baldwinian learning on each generation
418. else:
419. fitnesses = list(map(baldwinianLearning, pop))
421. for ind, fit in zip(pop, fitnesses):
422. ind.fitness.values = fit
424. best\_ind = tools.selBest(pop, 1)[0]
425. fitnessBest = best\_ind.fitness.values[0]
426. arr.append(fitnessBest)
428. g = g + 1
429. print("-- Generation %i --" % g)
430. # Select the next generation individuals
431. offspring = tools.selBest(pop, nElitists) + toolbox.select(pop,len(pop)-nElitists)
432. # Clone the selected individuals
433. offspring = list(map(toolbox.clone, offspring))
435. # for individ in offspring:
436. # print(individ)

439. # Apply crossover and mutation on the offspring
440. # make pairs of offspring for crossing over
441. for child1, child2 in zip(offspring[::2], offspring[1::2]):
443. # cross two individuals with probability CXPB
444. if random.random() < crossProb:
445. #print('before crossover ',child1, child2)
446. toolbox.mate(child1, child2)
447. #print('after crossover ',child1, child2)
449. # fitness values of the children
450. # must be recalculated later
451. del child1.fitness.values
452. del child2.fitness.values
454. for mutant in offspring:
456. # mutate an individual with probability mutateprob
457. if random.random() < mutateprob:
458. toolbox.mutate(mutant)
459. del mutant.fitness.values
461. # Evaluate the individuals with an invalid fitness
462. invalid\_ind = [ind for ind in offspring if not ind.fitness.valid]
463. fitnesses = map(toolbox.evaluate, invalid\_ind)
464. for ind, fit in zip(invalid\_ind, fitnesses):
465. ind.fitness.values = fit
466. #print(" Evaluated %i individuals" % len(invalid\_ind))
468. # The population is entirely replaced by the offspring
469. pop[:] = offspring
471. if boolean is True:
472. np.save('lamarckian', arr)
473. else:
474. np.save('baldwinian', arr)
476. if \_\_name\_\_ == "\_\_main\_\_":
477. x = toolbox.population(n=popSize)
478. x2 = copy.deepcopy(x)
479. main(x, True)
480. # neuralNetwork3DSurfacePlot()
481. arr = []
482. model = torch.load('model.pt').to(device)
483. main(x2, False)
484. # neuralNetwork3DSurfacePlot()
485. print("================================================Extracting Weights out of Network================================================")
486. extracted = extractWeightsOutOfNetwork(model)
487. print(extracted)
488. print("================================================Converting weights to individual==============================================")
489. real = real2Chrom(extracted)
490. print(real)
491. l = np.load('lamarckian.npy')
492. b = np.load('baldwinian.npy')
493. print(l)
494. print(b)
495. plotLearning(l, b)
496. plot(b)