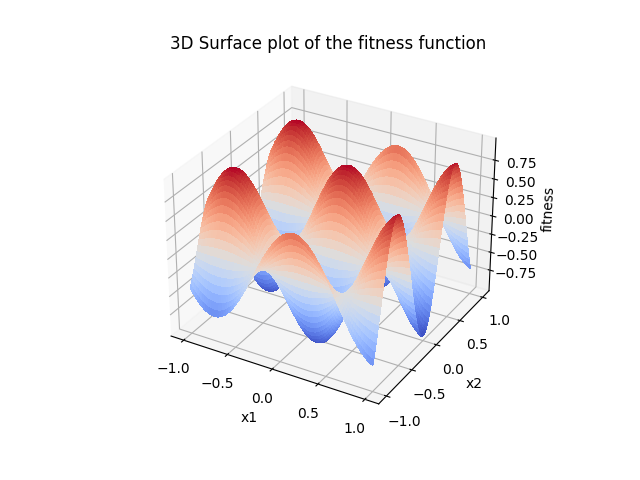
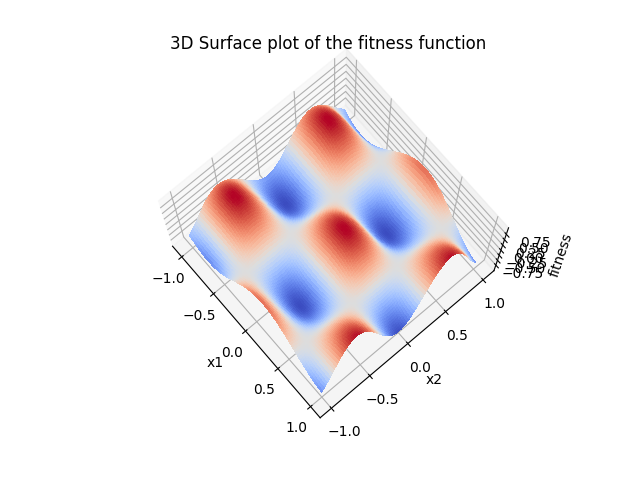
# COM3013 Computational Intelligence Coursework 2

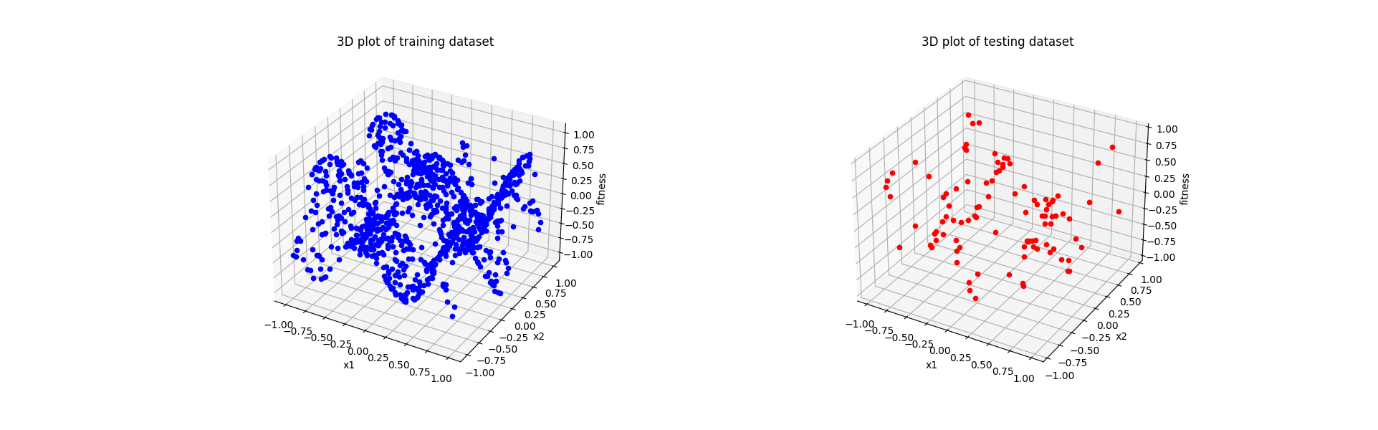
Shourya Raj Sharma | 6530100 | [ss02184@surrey.ac.uk](mailto:ss02184@surrey.ac.uk)

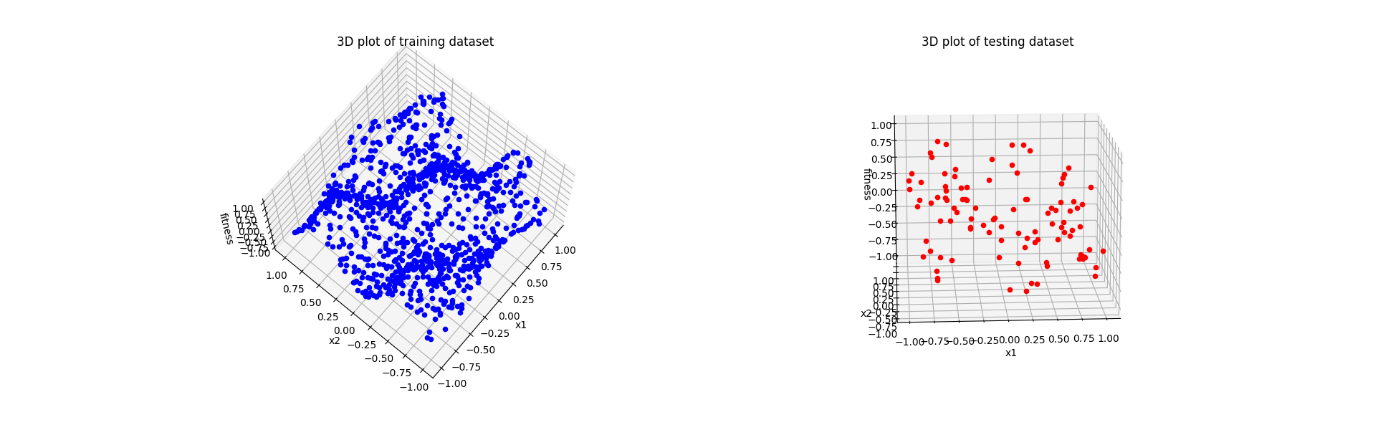
The main objective of the task was to train feed-forward multi-layer perceptron networks with two hidden layers to approximate the following function:

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



* 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 here.

* 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]