NAT cw2 report

Si Han Ang, Ching Ling Yeung

Task 1

In this task, we've adapted the python code for PSO given in the course materials to optimise the weights of our implementation of a neural network for the two_spirals.dat dataset.

1. The binary cross entropy function

$$-\sum(Y * log(AL) + (1 - Y) * log(1 - AL))/m$$

   where AL is the is predicted classification and Y the actual classification, is used to compute to cost of predictions, which is in turn used as a fitness function for the PSO. In this case, we would want to minimize the fitness value (alternatively, one could use $1 - fitness$ if we want to maximize the fitness values instead).

2. From doing some trials in the playground, we found that with one hidden layer, generally working with 8 neurons is better than 4 neurons in terms of converging to a lower test loss. We therefore decided to work with a network with an input layer, a hidden layer with 8 neurons, and an output layer.

   We've also found that results improved if we have in addition to the linear x and y terms, we also added x^2, y^2, and $sin(x)$ terms, which is what's used in this task (excluding part 4).

3. We ran a grid search (an exhaustive search) to determine which PSO parameters will give the best result in terms of achieving a higher fitness using the GridSearchCV function from the sklearn library with 2-fold cross validation. Using only 2-fold cross validation presents a tradeoff between accuracy and the runtime, where we opted for the latter.

   In the grid search, we fixed the population to 20, time step to 2500, and search range to 0.1 to help reduce the runtime of the search.

   We first determined the best activation functions to use in the neural network. From the three defined activations functions RELU, Sigmoid, and tanh, the grid search shows that having tanh as the first activation function and sigmoid as the second gives the best results.

   We then ran the grid search to determine the best values for the rest of the PSO parameters with values between 0.5 to 1.5, with 0.1 increments.The result are as follows:

   - The inertia $w = 1.2$
   - The attraction to personal best $a_1 = 1.5$
   - The attraction to global best $a_2 = 1.5$

Running the PSO with the above configuration and changing the population to 30 and the time step to 5000 (to try to get better results while keeping the runtime at a reasonable length), the neural networks gives predictions with accuracy ranging from 0.64 to 0.91, depending on the random initialization of each particle's position and velocity. After 100 runs, the average accuracy of the PSO is 0.719, with a standard deviation of 0.0654

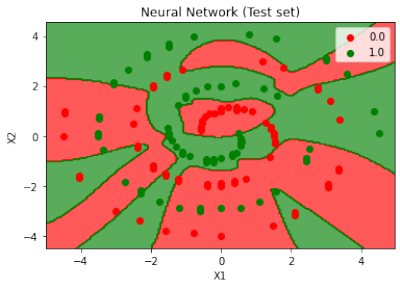

Figure 1: Instance of PSO resulting in 0.64 accuracy

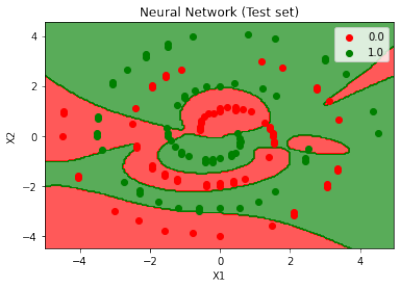

Figure 2: Instance of PSO resulting in 0.72 accuracy

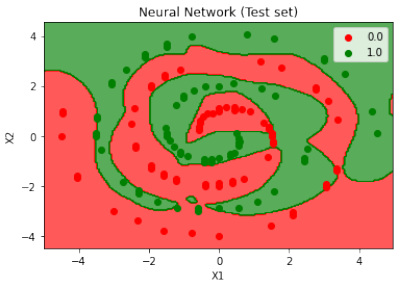

Figure 3: Instance of PSO resulting in 0.91 accuracy

4. If only linear input features are used (i.e. we only have x and y terms), a network with more hidden layers will be needed to get a good classification as the data itself is not linearly separable.

   We ran the PSO against a neural network with two hidden layers, one with 4 neurons and tanh as the activation function, and the other with 2 neurons with RELU as the activation function, as a well as an output layer with sigmoid as the activation.

   Using the same configuration as the previous part ($w = 1.2, a_1 = 1.5, a_2 = 1.5$) and running 10 trials of the neural network (significantly smaller to conserve time), we get on an average accuracy of 0.563 and a standard deviation of 0.0469.

   Running the grid search again this time on the deep neural network, we get the result of $w = 0.8, a_1 = 1.5, a_2 = 1.5$. With this configuration, we get an average accuracy of 0.58 and standard deviation of 0.0475 running the network 100 times.

   Running the deep neural network with only linear outputs gives a significantly worse result than the initial network with only one hidden layer but with non linear outputs, which was expected as the data is not linearly separable. The results here could potentially be improved by adding even more hidden layers to try to capture the non linear features of the data, but we'll need to run more experiments to determine how effective this would be.

5. Following the grid search, we did a more thorough exploration of the parameters of the PSO.

w	1.0	1.1	1.2	1.3	1.4
mean	0.676	0.688	0.730	0.679	0.698
std	0.0681	0.0663	0.0461	0.0733	0.0618

table of the accuracy with only varying w (i.e. inertia) for the PSO

It seems 1.2 is indeed the choice with the average highest accuracy, with a relatively small standard deviation. This probably means that this is around the point at which we've struck a good balance between exploration and exploitation.

a_1	1.3	1.4	1.5	1.6	1.7
mean	0.661	0.688	0.686	0.675	0.695
std	0.0373	0.0470	0.0448	0.0650	0.0530

Table 1: Accuracy with only varying a_1 (i.e. attraction to personal best) for the PSO

While $a_1 = 1.7$ and $a_1 = 1.4$ does give a higher accuracy on average, it does have a slightly higher standard deviation, and the difference between the average accuracy isn't too significant. It would be difficult to conclude from this whether $a_1 = 1.5$ is indeed the best parameter for getting a high accuracy.

a_2	1.3	1.4	1.5	1.6	1.7
mean	0.661	0.685	0.681	0.712	0.0727
std	0.0373	0.0470	0.0460	0.0781	0.0640

Table 2: Accuracy with only varying a_2 (i.e. attraction to global best) for the PSO

Here we see both values of 1.6 and 1.7 give a much higher average accuracy than the 1.5 we used in the previous tasks. These two values however does have a much higher standard deviation. It is possible that as the grid search only considered values between 0.5-1.5, the actual best parameter has not been found yet. In this case, it seems like the best value for the a_2 parameter should be higher (possibly higher than 1.7).

Task 2

In this task, we ran the GA on a deep neural network with back propagation using python's genetic algorithm library. The PSO from task 1 is not used here mainly due to the runtime required. We defined the fitness function as a combination of training loss, test loss and complexity measure indicated in part 4 of this task. The loss function that we used is the binary cross entropy function that is used and defined in Task 1.

We first ran a grid search to determine how many hidden layers we should work with for the subsequent GA search:

layers	1	2	3	4	5
min	0.389	0.374	0.55	0.275	0.411
mean	0.517	0.546	0.662	0.612	0.649
std	0.0976	0.0842	0.121	0.114	0.0908

Table 3: Fitness Evaluations across various No. of Hidden Layers

Since searching with 1 and 2 layers are relatively trivial (this can be done with brute force), we decided to focus on the case with 4 hidden layers,which seems to have relatively better results

among the other number of layers we've tested. We used tanh as the activation for the hidden layers, and sigmoid as the activation function of the output layer.

1. We encoded the structure of the network in two different ways. In the first method, we encoded the structure through a connectivity matrix, where an element is 1 if there is a connection between the two neurons, and 0 otherwise.

   The GA determined that the best structure is [9, 10, 10, 10, 1], with a fitness of 0.673.

   In the second method, we encoded the structure as the number of hidden neurons within the hidden layer.

   With this method, the best structure is [7, 5, 7, 10] with each number representing the number of neurons in that layer. This achieved a fitness of 0.383.

   Comparing the two, we see that the encoding using the second method achieved a much better fitness than the first.

2. We've defined 4 initialization function for the weights of the neural network: He initialization, Xavier initialization weight, Alt initialization, and Normal initialization. We then encode this with having 1 if the initialization function is being used as part of the initialization of the network, and 0 otherwise. For the GA evolution of the initialisation functions, we decided to only use an equally weighted training and test loss as the fitness function.

   We ran the GA with the following parameters:

   - iteration: 10
   - population size: 50
   - mutation probability: 0.3
   - elit ration: 0
   - crossover probability: 0.01
   - parent's portion: 0.3
   - crossover type: uniform

   Interestingly, the result from the GA shows that using only He's initialization function gives the best fitness, as opposed to using a mixture of different initialization functions.

3. From the trials we ran, the a learning rate of 0.01 seems to be good enough.

   We ran a grid search on the various configurations of crossover probabilities, mutation probabilities and population sizes for 10 generations to assess their impact on the GA's performance. The results are summarised in the tables below:

Crossover Probability	0.01	0.1	0.3	0.5
min	0.371	0.386	0.38	0.374
mean	0.399	0.405	0.404	0.404
std	0.012	0.008	0.011	0.012

Table 4: Effect of Crossover Probabilities on GA's Performance

Mutation Probability	0.01	0.1	0.3	0.5
min	0.394	0.386	0.394	0.371
mean	0.408	0.406	0.406	0.39
std	0.005	0.008	0.006	0.012

Table 5: Effect of Mutation Probabilities on GA's Performance

Population Size	10	25	50
min	0.385	0.371	0.386
mean	0.403	0.40	0.405
std	0.009	0.014	0.009

Table 6: Effect of Population Sizes on GA's Performance

Based on our findings from Tables 4 to 6, the performance of GA seems to be best for a relatively large mutation probability and low crossover probability. Population sizes do not appear to have noticeable effects on the performance of GA.

4. The complexity is controlled by including it as a factor in the fitness function of the GA. The fitness function of the GA would then be a multi objective function, combining the training loss, test loss, and the complexity of the network:

$$Fitness = 0.45 * trainLoss + 0.45 * testLoss + 0.1 * complexity$$

The training loss and test loss is equal to try to get a good balance between having a good fit and preventing over fitting, and the complexity is a smaller factor as the main goal is to get a good classification. The best evolved network has a fitness of 0.383 (the solution mentioned in part 1 of this task).

Task 3

In this task, we used a python implementation of genetic programming "tinyGP" written by moshesipper[2]. Our attempt at adapting this implementation to the task was ultimately unsuccessful.

1. In this task, we attempted to follow the approach by Koza and Rice (1991)[1]. In particular, we attempted to fix the root of the tree to be a sigmoid activation function and tried to implement rules analogous to those highlighted in the paper to ensure the output is a valid artificial neural network model. Instead of having a linear threshold processing function followed by a weighting function as in the paper, we tried to simplify it by having an activation wrapper function (either tanh or sigmoid) which applies the respective activation function on the sum of the 2 inputs.

   The main procedure runs by recursively combining the existing node with a left node and a right node until a tree is formed.

2. The code uses tournament selection as the basis for selecting good individuals and has a setting for bloat control. We adapted the error function to be the binary cross entropy function and the fitness function to be $1/(1 + (binarycrossentropy) + 0.01 * (sizeofindividualtree))$.

   The following parameters are used for GP:

   - population size: 50
   - minimum depth: 2
   - maximum depth: 5
   - generations: 50
   - tournament size: 5
   - crossover rate: 0.8
   - mutation probability : 0.2
   - bloat control

3. We would adapt the fitness function defined in task 2 to also incorporate classification error with the weights for each factor calibrated.

   $Fitness = \alpha * trainLoss + \beta * testLoss + \gamma * complexity + \omega * classificationError$

4. No, we are unable to confirm these findings. However, we think that cross-over might lead to less than desirable trees due to the random placement of the sub-trees as it requires stringent rules to formulate a neural network model.

5. It is worth noting that this application is only premised on the feed-forward neural network model. It would be interesting to adapt and use genetic programming to more complicated deep learning models like recurrent neural networks and convolutional neural networks. Recurrent neural network are especially interesting as it requires the encoding to also account for bi-directional information flow between neurons of different layers.

[1]

References

[1] John R. Koza and James P. Rice. Genetic generation of both the weights and architecture for a neural network. In *In Proceedings of International Joint Conference on Neural Networks*, 1991.

[2] M. Sipper. Tiny genetic programming in python. https://github.com/moshesipper/tiny_gp, 2019.

[1] We have adapted the neural network codes from the Coursera course on Neural Networks and Deep Learning by DeepLearning.Ai for task 1.