Natural Computing Coursework 2

November 2020

Abstract

The following review documents the performance of naturally inspired algorithms when used to optimise neural networks. Firstly, the performance of Particle Swarm Optimisation and the effect in changes in its many parameters is considered. This is used to optimise the weights of a given neural network. Secondly, the architecture of the network and how it is optimised using Genetic Algorithms. Thirdly using Genetic Programming to again optimise the architecture of the neural network and exploring different activation functions.

1 Introduction and Overview

Particle swarm optimization (PSO) is an algorithm inspired by the movement of organisms in a bird flock or fish school, that optimizes a problem by iteratively trying to improve a candidate solution with regard to a given measure of quality, usually a fitness of how well it is learning the data. It solves a problem by having a population of candidate solutions, here particles, and moving these particles around in the search-space. Each particle's movement is influenced by its local best known position for exploration of the search space, but is also guided toward the best known positions in the search-space for exploitation, which are updated as better positions are found by other particles, this uses the notion of "Swarm Intelligence" since all the particles can gain information form one another. This in turn, is expected to move the swarm toward the best solutions. PSO is an effective optimisation strategy for a range of problems.

In this scenario we will use it to optimise the weights of a neural network. In particular, we are trying to classify the Spiral Data-set described in the Google Tensorflow Playground [4].

Our data consists of 264 samples, where our features are 2d coordinates $\{x_1, x_2\}_i$, and with labels y_i which are a binary class.

We will train on 50% of this data, and test on the other 50%. We will further split the test set into a validation and test set, 30% and 20% of the full dataset respectively. We will then optimize the network weights using the training set, optimize the model and optimizer's hyperparameters using the validation set, and finally evaluate our final model on the test set to guage generalizability.

We have decided to use the third implementation of the task: Use the implementation from a neural network package of your choice or write the code yourself. We have made all the algorithms from scratch and used them for our analysis.

What is also important to note is that we have decided to put all the graphs and tables in the appendix, this is in order to be able to explain the results thoroughly and completely. When referencing for example **A.2** this will refer to the second figure in **Appendix A**.

2 Particle Swarm Optimisation

What is important to understand is how we are interpreting the problem to find the optimal weights of a neural network using PSO. The main and most important aspect one may ask themselves is what do the particles look like and what do they represent. The particles will represent the weights and biases of the neural network. One can picture putting them of all in a vector with dimension equal to the total number of weights and biases in the neural network.

We can represent them in the following way:

$$w = \left(w_1{}^1_1, ..., w_n{}^1_k, ..., w_{ij^n}, b_1^1, ..., b_n^1, ..., b_i^n\right) \tag{1}$$

Where each $w_i{}^n_j$ corresponds weight of the ith node of the nth layer going to the jth node in the next layer. Likewise for the biases b_i^n is the bias for the ith node in the $n+1$th layer. It is also important to note that the value of n will be at most the number of layers in the neural network and that the values of i and j will depend on the amount of nodes in each layer of the neural network.

2.1 Fitness Function

We will use the Binary Cross Entropy fitness function. This will give us a metric of how well matched the predicted labels are to the target labels. So, when we implement the PSO algorithm we will have a set of particles that represent weights as described before. To calculate the new fitness of each particle we need to evaluate the particles (the weights) at all the data points that we had in the training set and then calculate the fitness of the function accordingly, note that we will train by minimising this fitness function:

$$BCELoss = -\frac{1}{N}\sum_{i=1}^{N}\left(y_i \cdot log(\hat{y}_i) + (1-y_i) \cdot log(1-\hat{y}_i)\right) \tag{2}$$

Where y_i is the label for the $\{x_1, x_2\}_i$ value of our training data set and $\hat{y}_i$ is the predicted out of the neural network when we pass in the weights, in our case our the particle. We calculate the output directly as a 2-class classification problem using the sigmoid function. Once the weights are updated, each particle's network is fed all training cases and the fitness is calculated. The fitness for each particle is compared to their personal best, and if it's lower, the personal best value is updated. After all particle fitness's have been calculated, the

lowest fitness among them are compared to the global best fitness. If one of the new fitness's is lower than global best, then the global best value is updated with the particle that has the lowest fitness so far. As we train the neural network, we calculated the above equation to find the fitness of each particle and updated it accordingly in each run of the PSO algorithm.

So to recap we train the weights of each particle with the training set and then we optimize the hyper-parameters using the validation set. This is to avoid over-fitting to the training set. Finally we can then assess how well the final model generalises by running the model on the held out test-set, and calculating a fitness and an accuracy.

2.2 Search Space

Due to the intrinsic nature of the problem proposed, the spiral data set is not linear and as a result it is suggested that we use input transformations, therefore we create 6 features. We transform our coordinates $X = \{x_1, x_2\}$ into the following:

$$\phi(X) = \{x_1, x_2, sin(x_1), sin(x_2), {x_1}^2, {x_2}^2\} \tag{3}$$

We will also follow the guidance that we should work with one hidden layer of 4-8 neurons, of which we choose 8 neurons. This in turn corresponds to a search space of 65 dimensions in our case. This is because we are using a 3-layer neural network with one hidden layer, since it is known that this structure can solve the spiral problem given we have transformed the input features, as a result we are using a network 6-8-1, 6 input 8 nodes in hidden layer and 1 output node.

Analysing the structure of the network we are using, we find that we have $6 * 8 + 8 = 56$ weights from the first layer to the hidden layer including the biases.Then we are using $8 * 1 + 1 = 9$ weights including the bias From the second layer to the output layer. As a result, our total search space has dimension $56 + 9 = 65$. Hence the particles as described in (7) when using PSO will have 65 dimensions. This neural network structure will be the one we will be using throughout the analysis of the hyper-parameter search.

2.3 Initial Results

We started by creating a PSO with certain parameters which can be used as a baseline for our later experiments. The parameters for the baseline where: $\omega = 0.1$, $\alpha_1 = 1.6$, $\alpha_2 = 2.4$, search range = 1.0 and population size = 30. We tried to find the weights for the neural network with layers 6-8-1 with *tanh* as the activation function for each node, for all the PSO analysis, this means they will use all the ϕ transformations on the data as inputs equation **(3)**. From the graph **A.1** we found the performances on the training and validation sets at each epoch in the training schedule. After 10000 epochs, we achieved these baseline results: Training set accuracy:65% and validation set accuracy: 50% **A.2**. Note that the loss may decrease whilst the accuracy does not increase.

We decided to run a hyper parameter exploration to find the optimal parameters for the PSO. We searched for the all combination of the parameters that is ω, α_1, α_2 and search range and kept the population size constant at 30. For ω we searched for values between 0.1 up to 1. Searched for values of α_1 and α_2 that added up to 4, between 0 and 4.2, this is know from literature to be what these values should add up to and we searched for different search ranges. Then we ran the PSO with the possible combinations of these parameters.

After running the hyper parameter search we found that the best PSO parameters that created the best performance with loss of: 0.42 and 75% accuracy on the validation set. With the following parameters: $\omega = 0.3$, $\alpha_1 = 3.4$ and $\alpha_1 = 0.6$, search space = 1000 and population size of 30. What we can see from the graph **A.3** is that the loss does decrease over the number of epochs which means it is indeed optimising for the loss.

What is also nice about our result that it does better than a random classifier as well, we would expect that any if we achieved better than 50% classification accuracy on the validation. In our case the optimal parameters for the PSO achieve 75% accuracy on validation set this is better than the performance of the baseline PSO, as seen in figure **A.3**.

2.4 Comparison to model with linear inputs

The graph **A.4** shows the comparison of the best PSO found when we search the hyper parameters and a PSO with only linear inputs, in this case the structure will be 2-8-1 instead of 6-8-1 since we only put in the linear terms. We can see, problematically, that the accuracies are almost the same. We expected a performance hit when using only linear inputs but in fact we do not get one. We see that after 200 epochs, the optimized weights already have lower loss, but the accuracy does not improve.

This shows that when using the linear classifier with one hidden layer the task cannot be done successful, this is due to the nature of the classification problem since it needs non linear transformation to learn the spiral data set. We can also see though, that the network seems not to be utilizing the nonlinear features provided to it. This warrants additional investigation.

2.5 Effect of the PSO parameters

We will analyse the PSO parameters in the context of this problem. The equation used in PSO is for the new velocity and position are given by:

$$v_{i,t+1} = \omega v_{i,t} + \alpha_1 r_1 (p_i - x_{i,t}) + \alpha_2 r_2 (g - x_{i,t}) \tag{4}$$

$$x_{i,t+1} = x_{i,t} + v_{i,t+1} \tag{5}$$

Firstly, the accelerates constant α_1 and α_2 represent the particle stochastic acceleration weight toward the personal best and the global best. Therefore we can see that if α_1 is greater then the particle will be more exploitative and tend to the personal best likewise α_2 dictates much exploration around the space the particle does. When analysing these 2 parameters we decided to explore them as a pair. In our case we kept all the parameters of the PSO the same and varied α_1 and α_2 and trained each PSO on them. What we found is that the maximum accuracy on the validation set was achieved when $\alpha_1 = 3.4$ and $\alpha_1 = 0.6$ form graph **A.5**. This shows that in our particular

problem the particles that are more exploitative resulted in better accuracy this is because α_1 is greater than α_2. This could be due to the nature of our data, since its non linear exploration can hinder the loss since exploring the area can result in incorrect classification for a large amount of points and so the particles would prefer to reach the areas where they know they have a low loss. However it is still important to have the correct balance of exploration and exploitation and so the reason for why α_2 is so low with respect to α_1 is because the these parameters do rely on some other underlying assumptions like what input features we used or what what value of ω was close to 0 which was used with these α's.

The ω of the particle describes the previous velocity influence on current velocity. Controlling its selection may tune the global and local search ability of PSO. As a result if inertia is close to 0 then the particle has no memory and will decide to explore the space since since it only depends on global and personal best positions and not current velocity. Likewise if ω is large then the particle will take large steps and will try and search globally. We decided to explore the ω parameter and varied it from -1 to 1, the results can be seen in figure **A.6**. We can see that the best value for the validation accuracy was 65% when ω was close to 0, this suggests that for our problem the particle was more exploratory since it had no memory as explained above and so the equation of the velocity depended on global and personal best positions and not current velocity.

This does agree with our results since, we have found that the value of the α's for our problem have made the particles more exploitative whilst the values for ω make the particle more exploratory hence this interaction between these two parameters shows that there is a balance between exploration and exploitation for our solution.

As a concluding remark we have found these values for the PSO to be optimal for our given fitness function, to minimize the BCEloss. However using different fitness functions could result in different PSO parameters that could increase the validation accuracy of the network on the spiral data-set.

3 Genetic Algorithm

Genetic Algorithms is a search heuristic inspired by Darwin's natural selection. This is commonly used for optimization and search problems. The algorithm creates a population of chromosomes, where each chromosome encodes a solution to a particular problem. Just like in natural selection the best chromosomes, (the ones with better fitness for the problem) will be chosen to breed for future generations thus generating fitter solutions for the given problem over time. It also relays on biologically inspired operators such as mutation, crossover and selection. Before continuing, when referring to a chromosome this is the same a member of the population in the GA. A gene is the individual element of a chromosome which can be mutated.

The canonical Genetic Algorithm is described in figure **A.7** shows an outline of how we implemented the algorithm to optimise the shape of the neural network.

An overview of how the Genetic Algorithm works in the context for neural networks, we use GA to optimise the shape of a neural network for a given problem. We can encode the length of each chromosome as the number of layers and each number represents the number of neurons in each layer. Section 3.1 will further expand how we have implemented this in our problem. When evolving a population using a genetic algorithm you need to evaluate how well each member of the population is preforming. For each member of the population we train the neural network structure which is encoded in this member using the training set, and then calculated the fitness on the validation set. The members with the lowest validation fitness where given more chance of being picked in the selection process of the algorithm since they encoded better network structures for the problem.

Since it is expensive to train each member of a population at each epoch, what we did to overcome is use a cash to save values for the fitness on the training and validation sets for each different structures encoded in the chromosomes. This reduced the time of the genetic algorithm since when a member of the population has been trained it will not have to be retrained.

We then use selection to create the intermediate population. The selection we used a Roulette Wheel Selection, where we took the reciprocal of the fitness in order to give lower fitness a higher chance of being picked this is because we want to minimize fitness in our case so lower fitness should be more likely to be picked. Finally we do crossover and mutate on this intermediate population and then update the current population as this intermediate population. This is repeated for the number of generations we decide. Section 3.4 explains the Operators and Parameters of the GA in depth.

3.1 Problem Encoding

Encoding the problem is important to define correctly since this will allow us to use the GA appropriately. We know that in a GA we can encode each chromosome to have the same length. In our case we let the length of the chromosome by the number of layers in our neural network.

$$chromosome = [n_1, n_2, ...n_i, 1] \tag{6}$$

Where n_1 is a value between 1 and 6 corresponding to the nodes in the input layer, the values n_j for $2 \leq j \leq i$ correspond to the number of nodes in the $j-1$ hidden layer. The last value 1 will always be the same and is the output node of the neural network. All chromosomes will have the same length in the population.

For example $[3, 2, 4, 1]$ would encode a four layer neural network with two hidden layers. The number of nodes per layer is simply the number at that index. So in this case the first layer would have 3 input nodes, the first hidden layer would have two hidden nodes, the second hidden layer would have four nodes and finally the output layer would have one node.

We have also implemented managed to encode the GA such that it is compatible with a varying number of hidden layers. For example $[3, 0, 4, 1]$ would mean that the first hidden layer would have no nodes and so we would simply ignore it when creating the structure of the neural network for this chromosome and so we would create a neural network with three layers instead: three input nodes, four nodes in the hidden layer and one output node. This encoding is the same as $[3, 4, 1]$. Which means when we are working with chromosomes of length n we can create neural network structures of varying length up to and including n.

3.2 Controlling the Complexity of the Evolving Network

In order to control the complexity of the evolving network we implemented several restrictions on the encoding of the chromosome.

Firstly is that all the chromosomes are of fixed length which is decided when one initialises the GA. The variable *length* is an integer that controls the length of all the chromosomes when we initialize the population. Once this value is set all the chromosomes will have the same length for all generations. However this is very powerful, because since we have implemented our GA such that it can have a varying number of hidden layers with a certain *length* it encodes also all the smaller network structures too.

Take for example that we set $length = 6$ this would mean that all our chromosomes in the initial population would have length 6, which means they can encode a neural network with up to six layers (four hidden, output and input) however they can also encode a NN with five, four or three layers. Since if they contain a 0 gene then we simply ignore it and make a smaller NN structure. For example the chromosome $[3, 4, 0, 6, 0, 1]$ has length 6 but it encodes a NN with four layers namely $[3, 4, 6, 1]$ which is then created and evaluated.

Secondly, we can also control the complexity of the evolving network by controlling the number of hidden nodes in each layer. We do this again by a setting the variable *max_hidden_nodes* when we initialize the GA, this is an integer that controls the maximum number of hidden node each layer can have. Therefore if it set to 10 then this means that each layer can have at most 10 nodes per layer. To ensure this is consistent when mutating a gene which corresponds to a hidden layer it can change it to any integer up to this *max_hidden_nodes* variable.

However what is important to also note is that we have two special genes namely the first gene which corresponds to the input layer and the last gene which corresponds to the output layer. In our case we want a binary output since we need to classify points in the spiral data set. To take this into account is that the final gene will never be altered and will always have value of 1. Which ensures that we always have one node in the output layer. Likewise we want our neural networks to always have inputs, therefore we never let the first gene have value 0, this would simply mean that this chromosome will not have any inputs. What is also important is that the inputs can only be a subset of the set of phi transformations equation (3).

One can asks themselves which inputs does the chromosome in the population of a GA use? We know that there can be up to six inputs, but if the chromosome doesn't use all of them which one does it pick. To simplify this we choose the first n of the transformations. For example, if we have the chromosome $[3, 4, 0, 6, 0, 1]$ we can see that the first gene which corresponds to the input layer has value 3. In this case we would use the first 3 transformations of $\phi(X)$, so our inputs for this neural network would be $\{x_1, x_2, sin(x_1)\}$. This is a clear limitation of our implementation of the GA since it cannot strictly take the non linear transformations, however this is taken into account when implementing the GP.

3.3 Initialisation function for the initial population

To initialize the population of the GA one needs to specify several parameters: *length*: size of the chromosomes. P: size of the population an integer. *max_hidden_nodes*: maximum number of hidden nodes per hidden layer. *min_hidden_nodes*: minimum number of hidden nodes per hidden layer. Once we have these values we can use this algorithm which creates the initial population for the GA to evolve. We iterate over P to create P new members to add to our population. For each member we create them in the same fashion and then add them to a list of all the members. To create a member we do the following:

The first gene will be a random number between 1 and 6 by construction of the inputs since the input cannot be empty as explained above. The last gene will be always be set to 1 so we always have one output for all the encoding of the structures of the NN. For the remaining genes which correspond to the hidden layers, we set all of them to a random number between *min_hidden_nodes* and *max_hidden_nodes*. Doing this will allow us to have control over the maximum number of layers in the NN that the GA can produce which will be exactly *length*. Will also allow us to control the number of nodes in the hidden layers.

3.4 Fitness Function

Again we will use the Binary Cross Entropy (equation (2)) as the fitness function to measure how well each chromosome is preforming. This will give us a metric of how well matched the predicted labels are to the target labels.

However what is important to understand is how do we go from a chromosome encoding to evaluating the fitness of the NN encoded on spiral data set. We do this by first dropping all the zeros in the chromosome, then create a neural network with the reaming chromosome. This neural network can be trained using PSO using the training instances which correspond to the correct inputs (which will be indicated by the first gene). Once we have trained this NN we can evaluate the BCELoss on the validation set which we use as the fitness for this specific chromosome. This can then be used for the selection of the chromosome.

3.5 Operators and Parameters of the GA

In this section we will comment on the operators and parameters used in the GA. What is also important to note is that we need to define which parameters of the PSO we use since we train each neural network with this method. The Operators we will focus on will be selection, crossover and mutation. The Parameters we will focus on will be the parameters used for the PSO, and then particular ones for the GA which include: Population size, length size, *max_hidden_nodes*, *min_hidden_nodes*.

1. **Operators**:

   - Selection: We have used the roulette wheel selection where the best fitness will be more likely to be chosen this was appropriate for our case since we just want to pick the lowest fitness with higher probability, so we added to the roulette wheel the inverse of the fitness so lower values are more likely to be picked.
   - Crossover: This is has been maintained the same where crossover happens with a certain probability within each of the members of the population. We have decided to implement crossover in our problem since it will allow for exploitation, this is because

better results can breed to create better results, however what we realized was that in doing so, the fitness could decrease. Take for example the chromosomes $[6, 8, 0, 1]$ and $[6, 0, 8, 1]$ which both encode the NN with structure 6-8-1. However if these two breed at the mid-point then the resulting offspring would be: $[6, 8, 8, 1]$ and $[6, 0, 0, 1]$, We have now destroyed a good candidate solution and created two new structures that do not preform as well as the original chromosomes.

- Mutation: We can mutate all the genes except the final one which will always be equal to 1. The hidden layers can be mutated with a value between *min_hidden_nodes* and *max_hidden_nodes*. Whilst the first gene which corresponds to the input node can only be mutated to a value between 1 and 6. We include this to increase the exploration of our solutions created in our population. Mutation is particularly important to get longer chromosomes encode smaller network structures since it gives them a chance to get 0 in a gene.

2. **Parameters**:

   - Population size: we will vary between 30 and 60. This is because we will use the convention of using a population of 10*D where D is the length of the chromosome. [2].
   - Dimension (D): the dimension will vary between 3 and 6 to see the performance on different sizes, we decided to uses these values since we know that the problem is solvable with one hidden layer so we searched up to 4 hidden layers.
   - Number of generations: we will be kept constant at 250, we want to give the populations time to evolve. But also have a compromise with the computational time to evaluate this algorithm.
   - mutation probability: we will keep it the same at 0.05. However in the performance we will investigate how important it is for this task since mutation will allow give the algorithm more exploitation.
   - crossover probability: we used Single-point crossover with probability 0.7.
   - PSO parameters: for all the neural network structures we train them using the optimal PSO parameters found after the hyper parameter search, the number of epochs will be 1000 to reduce computational time since BCE loss converged after this number of epochs. **A.4** shows this is the case.

3.6 Performance of the GA

1. We first started with doing hyper-parameter optimisation, we changed the following parameters: population size (2 - 6) and dimension (20 - 60). We kept constant the mutation probability (0.05), crossover probability (0.7) and number of generations (250) for all parameters. After doing this we found that using longer dimensions (6 and 5) resulted in a worse accuracy on the validation set, like wise using dimension of 2 couldn't solve the problem. The optimal dimension used for the GA was sizes of 3 or 4, this is what is expected since we know that the problem could have been solved with one hidden layer and chromosomes with length 4 can easily encode a three layer NN. The results found can be seen in the table **A.8**. The best solution for dimensions of length four was $[5, 8, 0, 1]$ with validation accuracy of 75.4%.

   However one could expect that since longer chromosomes can encode smaller structures that we needed a way to increase exploitation, one way this could have been achieved is by increasing the mutation probability such that it is more likely for a gene to be equal to 0. This would mean that the structures would be smaller over time and would smaller structures which in turn could find better solutions.

2. The second results we wanted to see is if higher dimensions chromosomes tried to encode smaller network structures. Running tests on dimensions of size six we found that it was difficult to encode smaller network structures with a fixed mutation rate of 0.05. Then we varied the mutation rate for the GA (0.05, 0.15, 0.25), this should increase the chance of a 0 and in turn a lower structure. We found that the best mutation rate for the problem is 0.15 which resulted in the best validation accuracy, what is important to note that mutation rate of 0.25 was too high and changing the genes more often has a detrimental effect on the GA. **A.9**.

Concluding remarks: What we can see is that the GA for larger dimensions of the chromosomes they are dependent on the mutation rate, this is because longer chromosomes encode longer but not necessary better solutions to the GA. What could be done is increase the change of a 0 gene to reduce the complexity of the chromosome. What is also important to see is that the GA is a highly computationally expensive algorithm, results could have been better if we would have increased the number of generations so let the GA under go more mutation over time.

4 Genetic Programming

Genetic Programming is a search heuristic algorithm similar to GA. This is commonly used for optimization and search problems. The members of the population are encoded as trees, the population can evolve in a similar fashion to GAs using operators such as mutation, crossover and selection.

4.1 Problem Encoding

Encoding the problem is important to define correctly since this will allow us to use the GP appropriately. We will want more control over the parameters and the inputs we will use, we will build on top of the encoding used in the GA, which took into account varying lengths of hidden layers. We will encode the Cartesian Genetic Programming in this way:

$$chromosome = [[\text{binary list of integers}], [n_1, a_1], [n_2, a_2], ..., [n_i, a_i], 1] \tag{7}$$

The first entry, [binary list of integers] corresponds to a list of length 6 with will correspond to possible inputs which will be used after the ϕ transformation (Equation (4)) is applied to the training data. For example if we have $[0, 0, 1, 0, 1, 1]$ then we would use the third fifth and sixth, as the inputs which correspond to $\{sin(x_1), x_1^2, x_2^2\}$.

The genes in the middle which correspond to the hidden layers are explained as following: $[n_i, a_i]$. n_i is the number of hidden nodes in that layer, which as explained before will be maximum value of *max_hidden_nodes*, it can take value 0 which means that there are no nodes and so in this case we would ignore it that layer. a_i is a new variable which is the key corresponding to the activation function used for that layer. The activation functions we are using correspond to the dictionary: $\{A$: sigmoid function, B: tanh, C: ReLu$\}$.

The final gene remains the same as before and corresponds to the output node.

Take for example the chromosome: $[[1, 0, 0, 0, 1, 1], [2, A], [4, B], [0, C], 1]$, this would be translated as a 4 layer neural network with two hidden layers. The inputs would be the first and two last transformations of ϕ. The first hidden layer has two nodes and activation function A, the second hidden layer has four nodes and activation function B and one output node. We drop $[0, C]$.

We can then select, crossover and mutate in a similar fashion as in the GA. The only difference would be mutation the first gene where we can change every entry of it, and the genes in the hidden layers we can mutate the activation functions used too. Therefore our terminals would be the number of hidden layers and non-terminals the activation function used.

4.2 Operators and Parameters of the GP

The operators and the parameters used are the same of that of GA, this is because we have encoded the CGP in the form of a string and as a result the operators and parameters will be the same as in GA. However what will be different is that we will always have that the dimension of the chromosomes to always be 6 this is because it takes into account all network structures of up to 6 layers, and doing this allows us to control the complexity of the graphs created, this means that our graphs created never grow in size beyond ten layers. Which follows the same principle as section 3.2 of the GA. We also kept the probabilities of crossover and mutation constant, at 0.7 and 0.05 these values are used in literature, to ensure a balance between exploration and exploitation, in particular mutation for exploration[3] and crossover for exploration[1]. The PSO used to train all the structures will be the same using 1000 epochs to train the NN, since PSOs converged after this number of iterations for our given problem.

4.3 Results of the GP

1. The performance of the GP was very similar to that of the GA due to the encoding. We found that just like in GA when initializing longer members they do not find good solutions due to the mutation rate being small and so the amount of zero chromosomes is low and as a result will be hard to create smaller network encoding structures. The best structure found by the GP found was: $[[1, 0, 0, 1, 1, 1], [8, A], [0, C], 1]$ when D = 4 **A.10**. However what is nice to see is that most of the GP chose to use the non linear inputs in the first gene which suggests that these inputs have a strong influence on how well it preforms.

2. Our second test was to find out if long-thin architectures are better than short-thick ones? To do this we set the dimensions of all chromosomes to 6 and then we set the maximum number of nodes per layer as 8. While keeping all the other parameters constant. This means that the GP can generate any neural network of at most 6 layers with at most 8 hidden units in each layer. We set the population size to 10. The main idea behind this was to see to where the population of GP diverged to. We ran this for 250 generations.

   After having run the test on this we could see that the GP preferred long thin architectures rather than short thick ones. However this could be the case because of the way we initialize the population. We started with dimension 6 and 8 hidden nodes, this means that for a layers the small chance of no hidden nodes is 1 in 8 which means that when we initialize the population they will be naturally very long. Thus this depends on the initial population as well as on the mutation rate which in this test was 0.05, since if the mutation rate to create 0 would be greater then the architecture could explore smaller NN structures and prefer short think ones, the result we got was: $[[1, 0, 1, 1, 0, 1], [5, B], [1, A], [0, B], [2, A], 1]$ with validation fitness of 0.6536. Another reason for this is because we ran the PSO for training for only 1000 epochs which increasing the number could have found more accurate fitness for shorter chromosomes and thus they would have been more likely picked in the roulette wheel. However we do see that the chromosome has successfully encoded a 0, so running for more iteration could have resulted in a smaller and thicker one.

3. The third test was to find if GP preferred small or large populations. We set the the possible number of members of the populations to be either 4, 10 or 50 and so we will do three separate tests all with 250 generations. We found that for small populations the GP gets stuck at a local minimum after around 50 generations and couldn't explore. This was due to the fact that the mutation rate that we used for this test was too low as explained previously and so couldn't explore more architectures. This showed that the GP preferred larger populations for this problem since it could explore more space of solutions **A.11**, hence we could not confirm this for CGP in our case. What is also important to note is that the validation loss even for the best population is high, this was due to the dimension we used of 10, since we wanted the CGP to be able to explore different sizes of structures, however this was not beneficial in our case.

4.4 Outlook on other tasks

Another part of the network structure that can be optimised with GP is regularization. What could have been done we could have done L-n regularization to improve the structures of the NN created. We could have encoded in each genes that encoded a hidden layer, we could have added an extra parameter for a regularization function that can be applied to that layer, this would mean that the structure can now also take this into account.

References

[1] Y. Ammar H.H.; Tao. *ingerprint registration using genetic algorithms. In Proceedings of the 3rd IEEE Symposium on Application-Specific Systems and Software Engineering Technology, Richardson, TX, USA,* Mar. 2000.

[2] Pedro Diaz-Gomez and Dean Hougen. *Initial Population for Genetic Algorithms: A Metric Approach.* Jan. 2007.

[3] Borhan Kazimipour. *Why is the mutation rate in genetic algorithms very small?* Jan. 2013.

[4] *Tensorflow Playground.* URL: `https://github.com/tensorflow/playground`.

Appendix A Graphs and Results

A.1 PSO baseline performance over 10,000 epochs

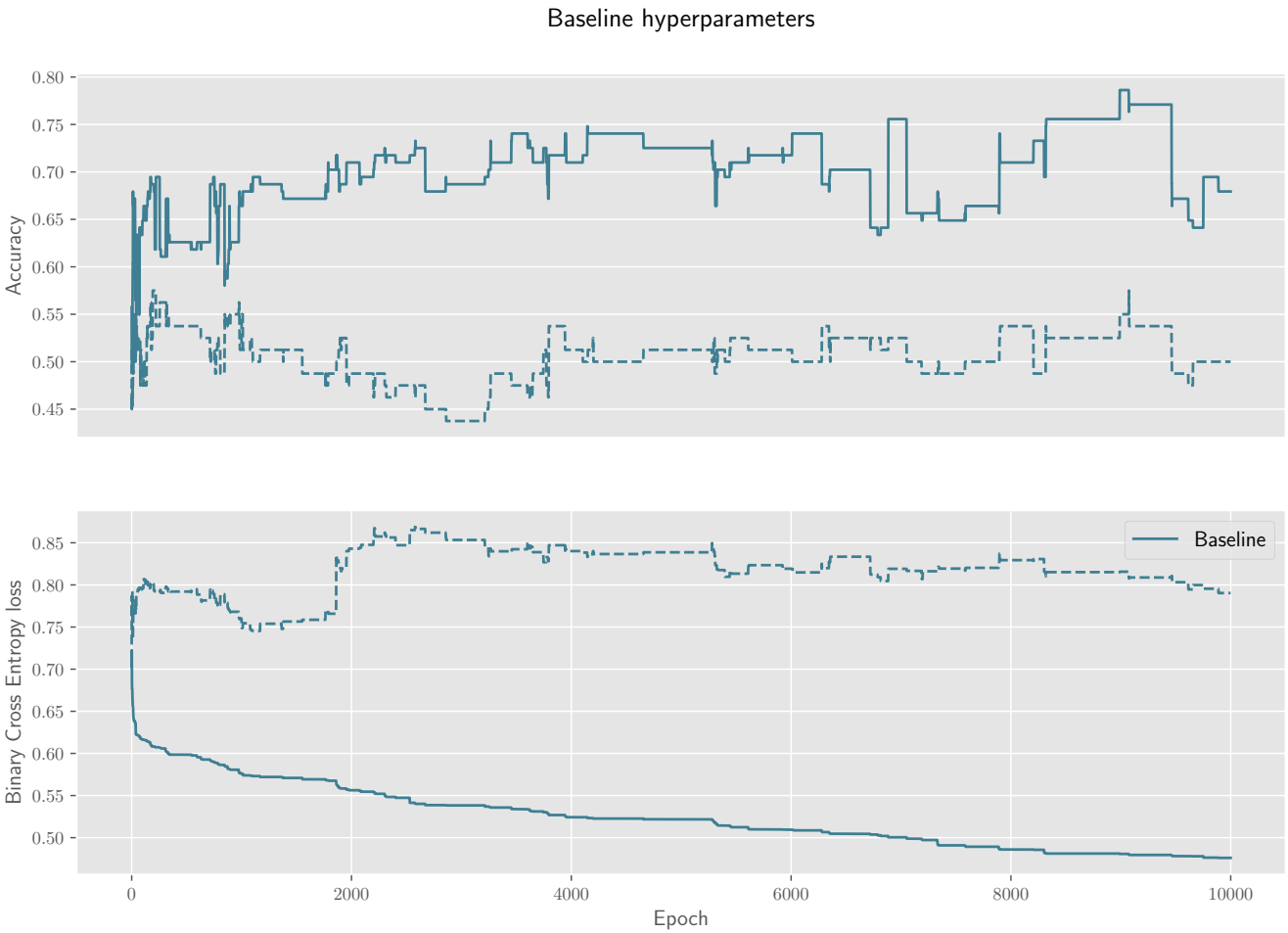
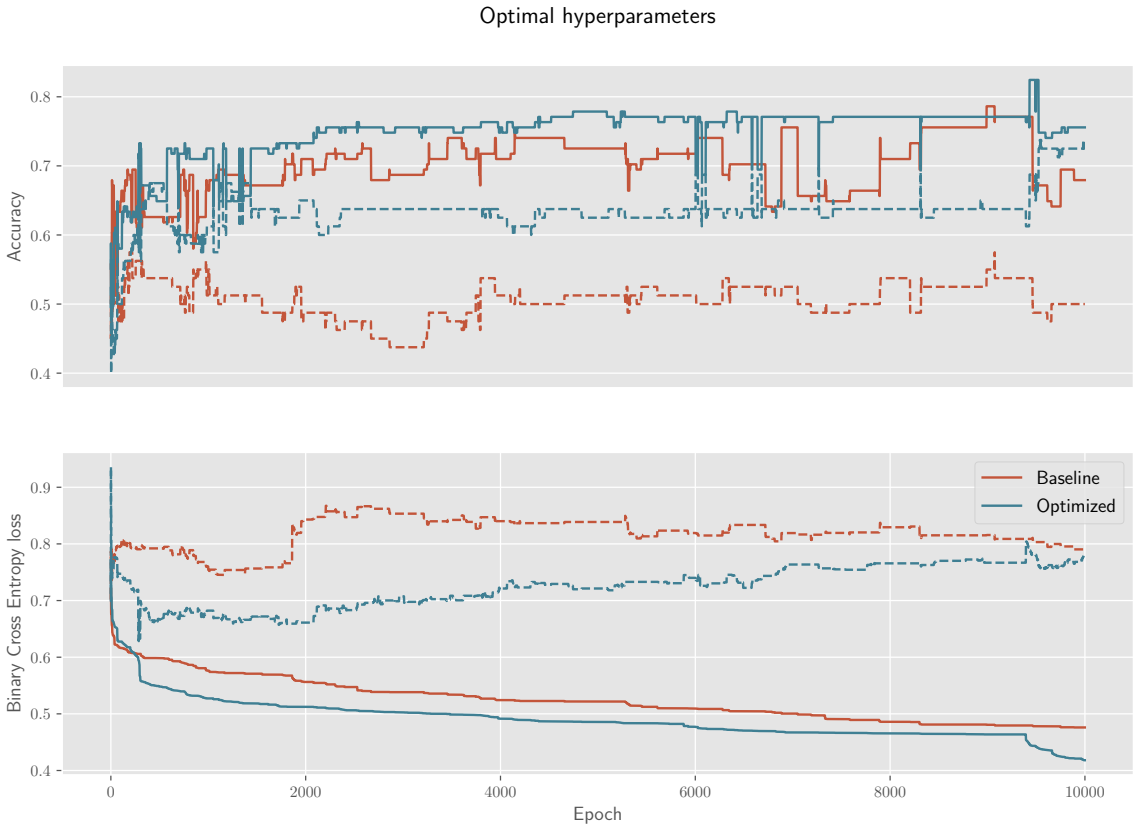


Figure 1: Baseline hyperparameters: inertia=0.1; a1=1.6l a2=2.4; population size=30; search range=10; seed=12345324

A.2 Results for the baseline PSO

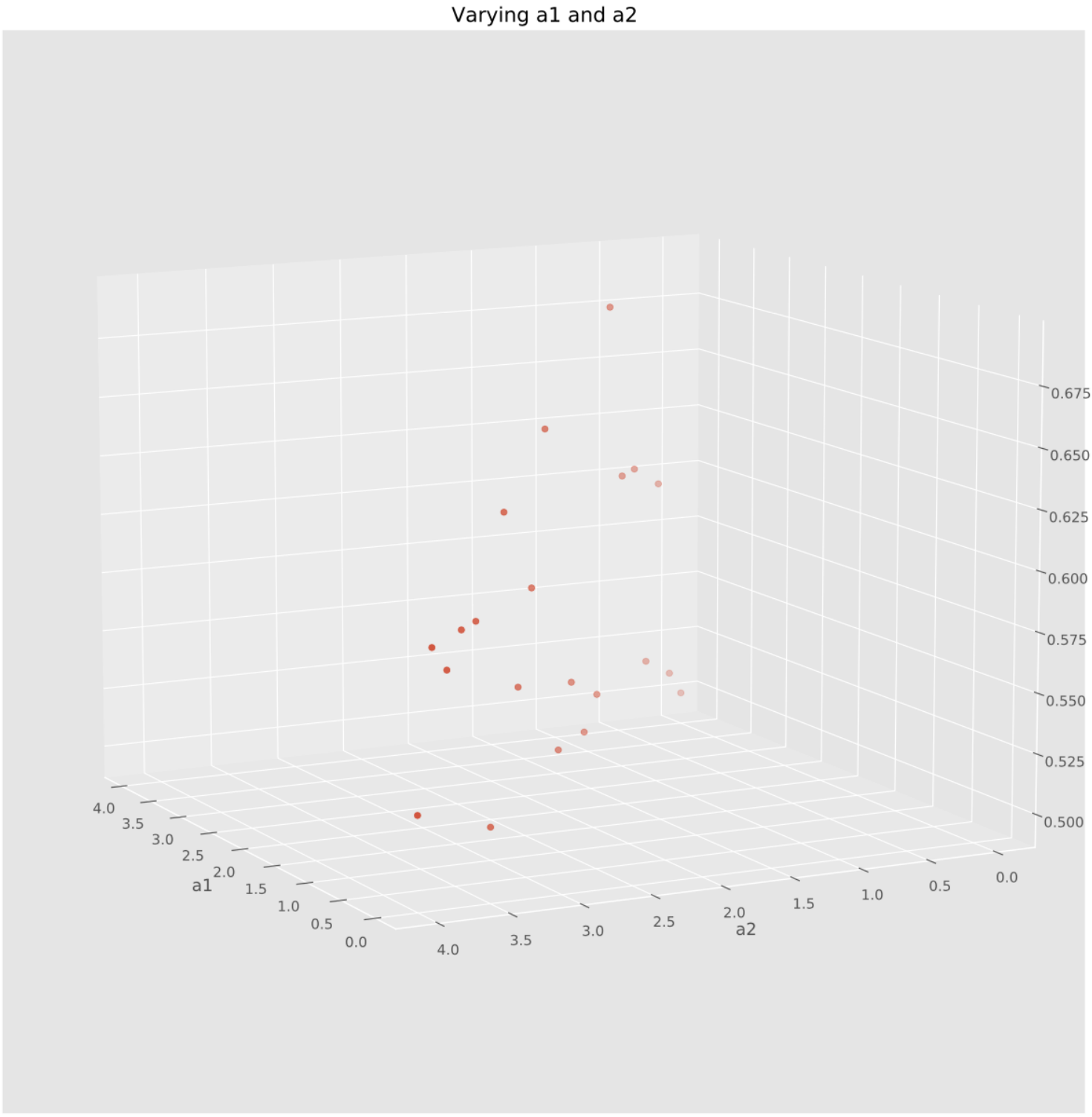
	Accuracy	**BCE Loss**
Training Set	67.94%	0.476
Validation Set	50.00%	0.790

A.3 PSO baseline comparison with best PSO found after hyper-parameter search



A.4 Best PSO comparison with linear inputs

A.5 Varying α_1 and α_2 while keeping other parameters constant



A.6 Varying ω while keeping other parameters constant.

A.7 Canonical Genetic Algorithm used to find the best network structure for the neural network for the 2 class spiral problem

Algorithm 1: Canonical Genetic Algorithm

```
for number of generations do
    evaluate fitness: for each member in population;
    selection based on fitness: intermediate population;
    crossover: intermediate population;
    mutate: intermediate population;
    update: population = intermediate population;
```

A.8 Results for GA over different sizes of dimension

Dimension	3	4	5	6
Validation Accuracy	72.3%	75.4%	65.4%	55.2%
Best network structures	[6,5,1]	[6,8,0,1]	[4,7,8,0,1]	[5,4,8,7,1]

A.9 Results for GA over different mutation probabilities with fixed dimension and population (6, 60)

Mutation rate	0.05	0.15	0.25
Validation Accuracy	55.2%	62.3%	52.4%

A.10 Results for GP over different sizes of dimension

Dimension	3	4
Validation Accuracy	70.3%	75.3%
Best Network Structure	[[0,1,1,1,0,1],[6,A],1]	[[1,0,0,1,1,1],[8,A],[0,C],1]

Dimension	5	6
Validation Accuracy	68.4%	57.2%
Best Network Structure	[[0,0,1,0,1,1],[8,A],[4,C][0,C],1]	[[0,1,1,1,0,1],[5,A],[8,B],[7,B],[0,C],[1,B],1]

A.11 Results for GP over different population sizes with dimension of chromosomes = 10

Population Size	4	10	50
Validation Loss	0.8813	0.7451	0.5313