F20BC - Coursework Report

# Implementation

The approach we took was to create classes for the following: Artificial Neural Network, Particle Swarm Optimisation, Particle, and Hyperparameters.

## Hyperparameters Profile

We created hyperparameter profiles to store all the hyperparameters we plan to adjust. These include the number of layers and number of neurons per layer, activation functions for each layer, and the adjustable parameters for the PSO, which are the inertia, cognitive, social, and global component as well as the number of iterations and the number of particles. All of this helped us to make the testing of different configurations of the PSO and ANN easier.

## Artificial Neural Network

The ANN class stores layers sizes, activation functions, weights and biases. The weights are initialised randomly while the biases are just filled with 0’s, the weights could also be initialised to 0’s as we do not work with the ANN directly as the weights and biases come from the particles. Activation. Functions and layers sizes come from the Hyperparameter profiles. As we are using PSO to optimise the weights and biases we only need the ANN class to have forward propagation method to get the output. The forward propagation works by; for each hidden layer, multiplying the input matrix and the weight matrix together, then adding the resultant matrix with the bias matrix then finally applying the activation function of the layer to the output. After getting the outputs for each hidden layer we then do the same on the output layer consisting of 1 neuron as this is a binary classification problem, applying our activation function for the output layer we get the prediction.

## Particle

The particle class stores the velocity and position of the particle and handles the position update of the particle. It also stores the particles best position and best fitness value. The method that calculates the new position has been completed with boundaries of the min and max values found in the dataset, which turned out to be min: -13 & max: 18. The update of position is simply adding the current position and the velocity matrices together, if the new position falls outside the boundary we randomly reset the particles position within the boundary.

## Particle Conversion

In order to discuss how to convert a Particle into a Neural Network, we need to, conceptually, understand how a Neural Network could be converted into a Particle (which wasn’t necessary in the implementation, but provided an understanding of how to convert a Particle to an ANN). It would be done in 5 steps:

1. Go through every layer (except the input layer)
2. Get the weight matrix from the previous layer to this layer
3. Flatten the weight matrix into a vector
4. Append the biases of this layer onto the end of this vector
5. The concatenation of all of these layers’ vectors in order produces a Particle’s Position

With that understood, the conversion of a Particle into a Neural Network works as follows:

1. Firstly, there exists a helper function “get\_particle\_layer\_counts”, which, based on an architecture of a Neural Network (an array of numbers representing number of neurons in each layer – the first number is no. neurons in the input layer, the last is no. neurons in the output layer).
2. This function allows us to get the number of weights and biases in each layer, based on the architecture of the Neural Network.
3. Secondly, there exists a helper function “get\_layer\_vectors”, which, given a Particle, converts its position into an array of vectors. Each vector in this array represents all the weights *and* biases for that layer, as described in the ANN-to-Particle conversion.
4. With that, the conversion of a Particle to a Neural Network works as follows:
   1. We take in a Particle, and, using “get\_layer\_vectors”, get the arrangement of weights *and* biases in each layer, based on its position
   2. Since we know, conceptually, that biases would be at the end of each layer’s vector, and that the number of biases would equal the number of Neurons in that layer, we extract them using a slicing operation.
   3. This leaves us with a flattened vector of weights
   4. Based on the number of neurons of the previous layer, m, and the no. neurons of this layer, n, we use NumPy’s “reshape” method to convert the flattened weight vector into a Weight matrix
   5. This is done for every layer except the input, producing weights and biases
   6. The resulting Neural Network is returned

## Particle Swarm Optimisation

Particle swarm optimisation has 4 functions, the PSO algorithm itself, finding informants of a particle, accessing the fitness of the particle and updating the velocity of the particle. All these functions are explained in more detail in the sections below. Initialising the PSO is done by first extracting all the relevant hyperparameters from the profile that has been chosen, as well as the data to be used and the labels from the data.

### PSO algorithm

The algorithm consists of first updating the particles position as described in the Particle section, then checking the fitness of the particle, replace the global and personal best positions accordingly if the fitness of the particle is higher than previously set positions, lastly we update the velocity. We do all of this for a set amount of iterations, in the end getting the global best fitness.

### Getting informants

Before Informants are acquired, the array of all particles is copied, and the particle looking for its informants is removed from this copy, producing just a list of possible informants.

Note – in our implementation, the minimum number of informants is 1. Therefore, it was chosen to enforce the minimum number of Particles in PSO to be 2, since, if there is only 1, the particle will remove itself from the list of possible informants, and there would be no informants to choose from.

Using NumPy’s “random.choice” method, N particles, where N is number of informants, are selected randomly, without replacement.

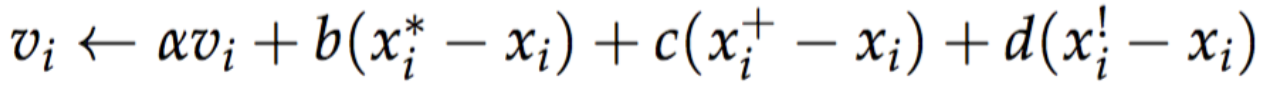
As one can see, it was chosen to randomly select informants at each time step, instead of pre-selecting informants for each particle at the beginning and retaining them.

This was done for two reasons:

1. It would be less memory-intensive to pick random informants for particles rather than storing a Particle’s informants in memory.
2. If all of a Particle’s preselected informants converge on a Local Optimum, there would be no way for the Particle to do any better. However, if its informants are selected randomly, then there is a chance for the Particle to come across informants whose positions are close to the Global Optimum.

### Updating velocity

The velocity update function was implemented using this equation:



First we take the velocity of each particle and multiply it by the inertia (the amount of the velocity we should keep, alpha), Then we will make sure the particle partially moves towards the best position previously discovered by itself (weight b in the equation), then towards the best position discovered by its informants(weight c in the equation), and finally previously discovered global best(weight d in the equation). This is done for each dimension in the velocity vector. The weights b,c, and d are all a random value between 0 and the value of hyperparameters β, γ, and δ this introduces non linearity into the algorithm allowing the particles to search more space without being influenced by the weights all the time, which could lead to a local maxima instead of the global one. The new velocity will then be used in the next iteration of PSO to update the position of the particle within the search space.

### Accessing fitness

For the fitness of the particle the specification mentioned to just use the accuracy of the artificial neural network for the given data and not worry about overfitting. The fitness function takes in the entire dataset, and the labels separately, then convers the particle into a neural network method for which is described in Particle Conversion section, then we run every datapoint in the dataset through the forward propagation method of our artificial neural network and get the predicted values, which then are converted into the classes by using a simple threshold where any values blow 0.5 are classed as 0 and values above 0.5 are class 1. We then compare the predicted labels against the actual labels and get the accuracy by this simple formula: (number of correctly predicted labels / total number of labels \* 100) giving us the percent accuracy of the network.

# Experimental investigation

# Results

# Discussion and Conclusions

# References