The Code

The code is designed in such a way that anyone can use it without knowing the inner workings of the various algorithms, or without having any knowledge of MATLAB, in which the entire project is coded. The current version of the code aggregates older algorithms - BioGP, EvoNN and cRVEA, as well as newly developed algorithms – EvoDN and deepRVEA, packaged in a root folder named MASTERCODE. The files and folders contained in the root folder will be briefly discussed in subsection 1 of this chapter. The information provided in that subsection should enable any new user to use the code. The second subsection will delve into the details of how single layer Artificial Neural Networks (ANNs) Deep Neural Networks (DNNs) are handled in the program and will detail the various mathematical definitions proposed by the author. The third subsection will detail how those data structures and mathematical definitions are used in the EvoDN and deepRVEA to train DNNs on discrete datasets.

1. Outline Of MASTERCODE

The folders and files contained in MASTERCODE are described tables 1 and 2 respectively.

|  |  |
| --- | --- |
| Folder Name | Description |
| BioGP | Contains BioGP algorithm for training of Genetic tree models and their optimization via PPGA |
| cRVEA | Contains cRVEA algorithm for constrained optimization using surrogate models |
| deepRVEA | Contains deepRVEA algorithm for training of DNNs via RVEA |
| EvoDN | Contains EvoDN algorithm for training of DNNs via PPGA |
| EvoNN | Contains EvoNN algorithm for training of ANNs via PPGA |
| RVEA | Contains RVEA algorithm for unconstrained optimization using surrogate models |
| Output | Contains the output of the code |
| Tools | Contains tools for advanced users |

Table 1. Subfolders in the root folder

|  |  |
| --- | --- |
| File name | Description |
| Autorun.m | This is a wrapper for the entire code. User will use this file to run the code |
| Configuration.m | User can use this file to change the parameters used in the different algorithms |
| Constraints.m | This file contains the mechanisms for constraint handling and should not be edited |
| Default.mat | This file contains the default parameters to be used in the algorithms, as suggested by the author |
| README.md | This file contains brief instructions to use the code |

Table 2. Files in the root folder.

Before you begin

Before starting training, the user should put the dataset, on which to train, in the root folder. The dataset should be in xls format and should not contain any spaces in its name (support for more formats to be added later, read README.md for details). It is important that the file should be user editable. Sometimes files downloaded from the internet are “Protected” by Microsoft Office as a security measure. These files can’t be edited and will produce errors in the code. To overcome this problem, simply open the file in Microsoft Excel, and when prompted, select the “Enable Editing” option. Furthermore, the first row of the sheet should contain independent variable names in the format “X1, X2, X3…” and the dependent variable names in the format “Y1, Y2, Y3…”. The second row of the sheet can contain descriptive names for the respective columns, the program will ignore this row.

Running Autorun.m

A snippet of the file Autorun.m is shown on the next page. Training on multiple algorithms can be done consecutively by assigning multiple Algorithm names to the variable Training\_Algorithms, as shown in the figure. The Algorithm names have to be same as the folder name they are stored in. Multiple optimization algorithms can be run in a similar way. The name of the excel sheet to be used for training must be assigned to the variable Problems. Multiple sheet names can also be assigned at once.



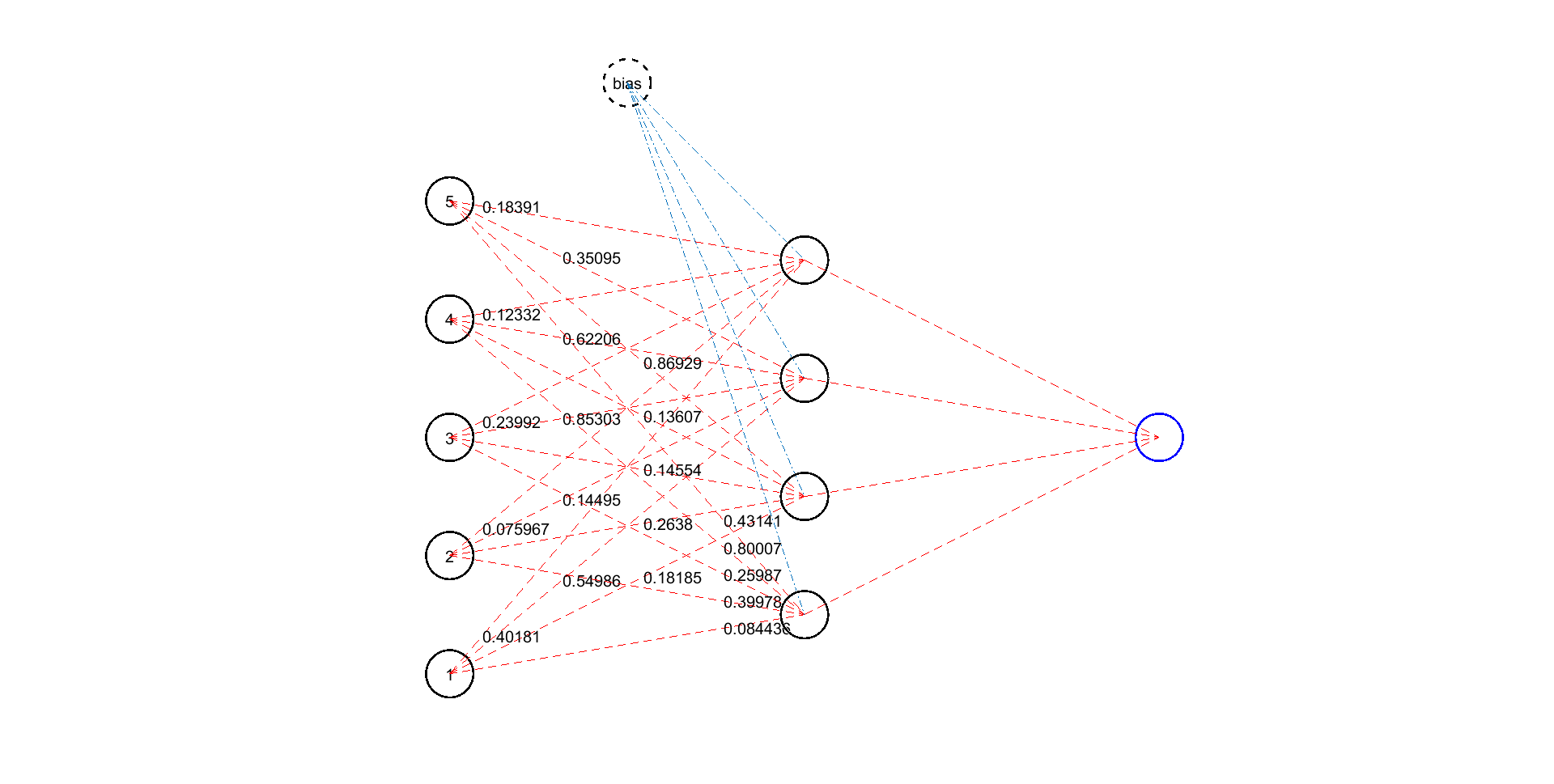
The variables in\_index and out\_index are used to designate the column indices of the independent and dependent variables respectively. param\_name will contain the names of the files which contain the algorithm parameters. The method to create those files will be discussed shortly. do\_training and do\_optimization are boolean variables which enable/disable training and optimization respectively. use\_defaults overrides param\_name and forces the program to use the default configuration file defaults.mat. Once all these variables have been assigned properly, executing Autorun.m will commence training/optimization. The outputs, including the model file, model accuracy, SVR results, and optimization results will be saved in /Output//<Problem \_name>//<Config\_filename>//<Training\_Algorithm\_name> folder.

Creating the configuration files

Parameters can be set manually by using custom configuration files. These files can be created using the Configuration.m program. The various algorithm parameters of BioGP, EvoNN and RVEA have been discussed before (insert citations). Most parameter variables are self-explanatory and the file is well commented and easy to understand. Executing this file will create a configuration file in the root folder with the name given by the output.name variable.

2. Deep Artificial Neural Networks

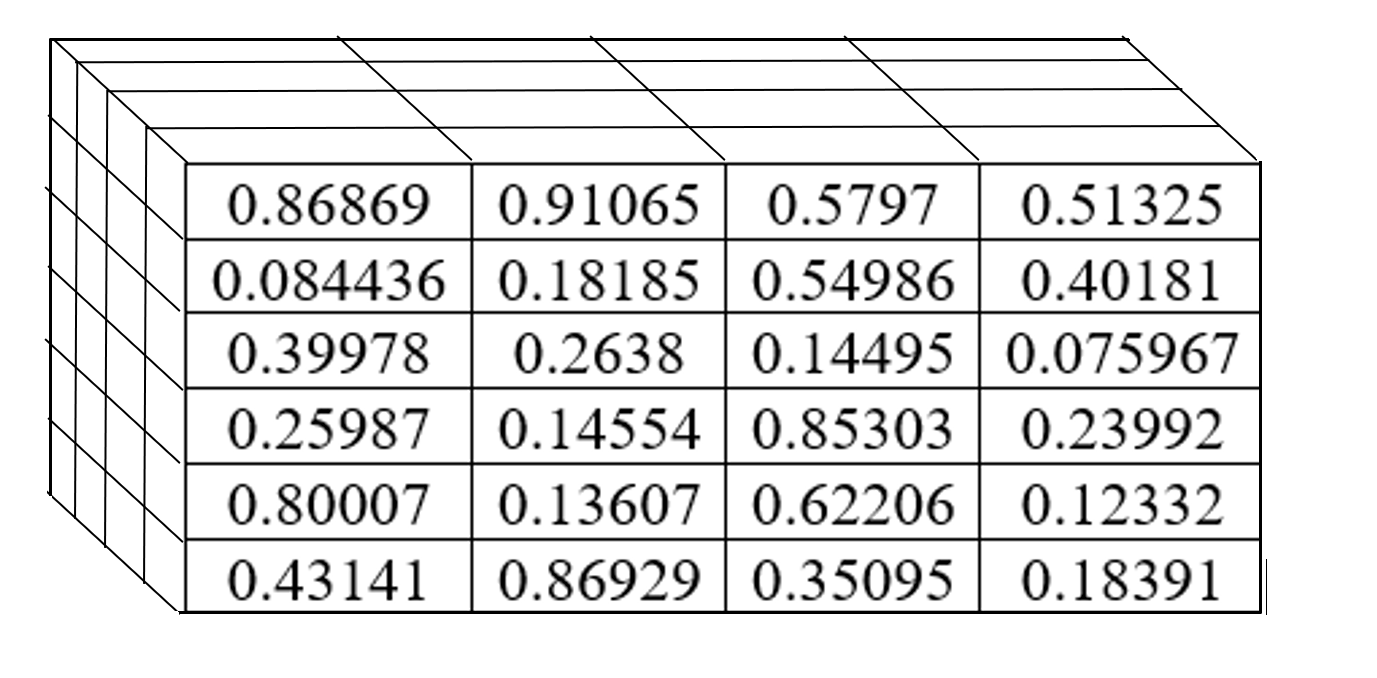
EvoNN creates ANN models with only one hidden node-layer. The second half of the network is generated by Linear Least Square technique, and as such, the GA only has to manage the first half of the network. This can be represented as a 2-D matrix of dimensions [num\_input\_nodes, num\_out\_nodes]. One extra row is added to store “bias” values. These new 2-D matrices can be stacked together to form a population of ANNs.

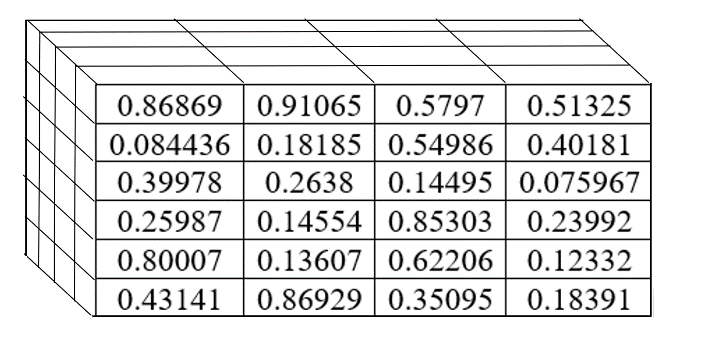


Genetic Algorithm

LLSQ

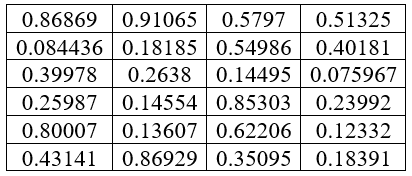
Biases

Weights



Individual[num\_in\_nodes+1, num\_out\_nodes]

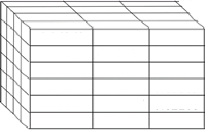
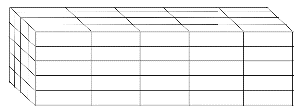
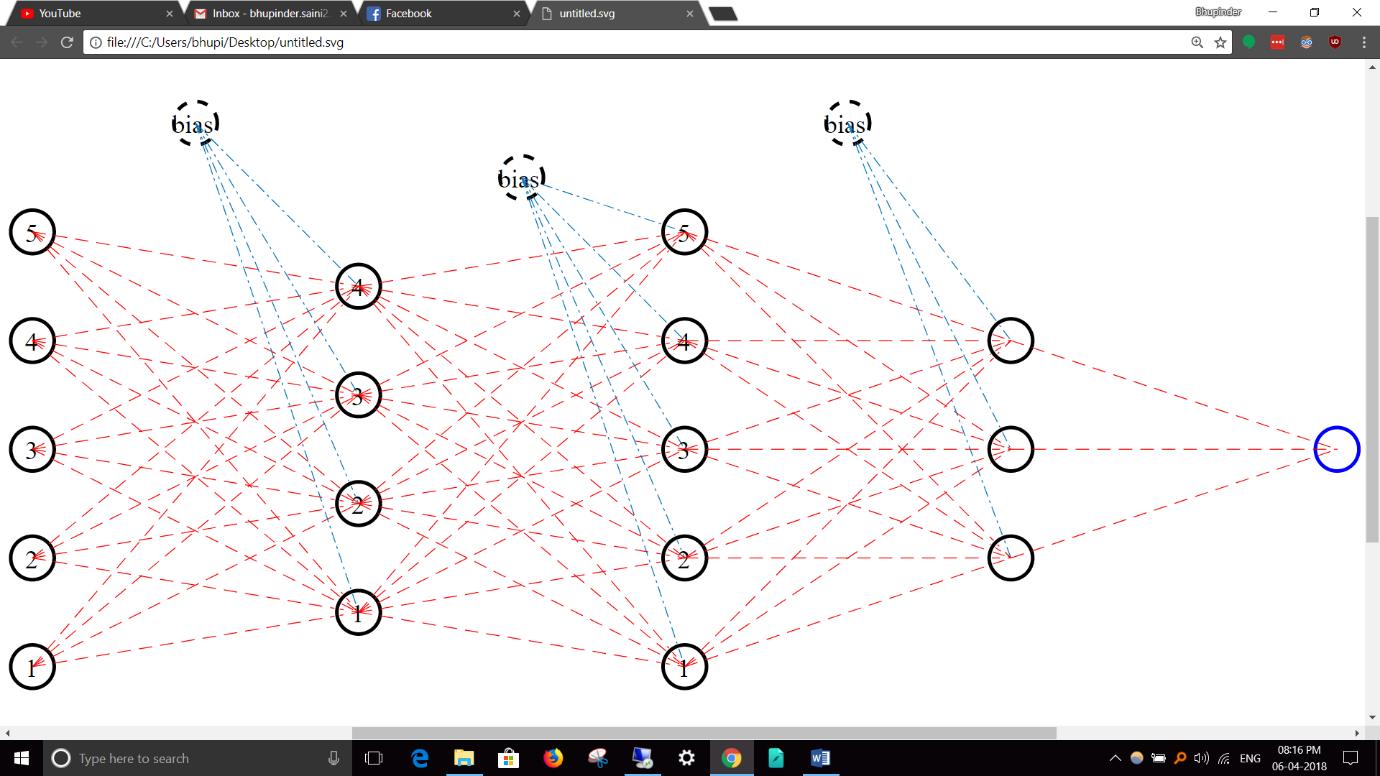
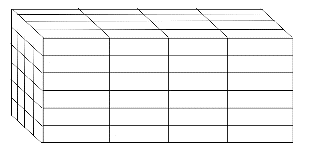
Population[Pop\_size, num\_in\_nodes+1, num\_out\_nodes]



EvoDN and deepRVEA work with DNNs. All but the final layer of the network are optimized by genetic algorithms, and the final layer is optimized by LLSQ. The DNN structure can be defined using the variable NNet\_str in Configuration.m. NNet\_str = [x1 x2…xn] will create a neural network with n hidden layers, with x1 nodes in the first hidden layer, x2 nodes in the second hidden layer, and so on. As each layer has, in general, a different number of connections, they cannot be represented by 2-D matrices of the same dimensions. To bypass this issue, while handling the entire population, the program groups together the weight and biases of equivalent layers from the whole population together in 3-D matrices as before, and then groups together the many 3-D matrices together in a cell structure. This makes the handling and manipulation of the population easier.

Genetic Algorithm

LLSQ



Population{1}[Pop\_size,in+1,x1]

Population{2}[Pop\_size,x1+1,x2]

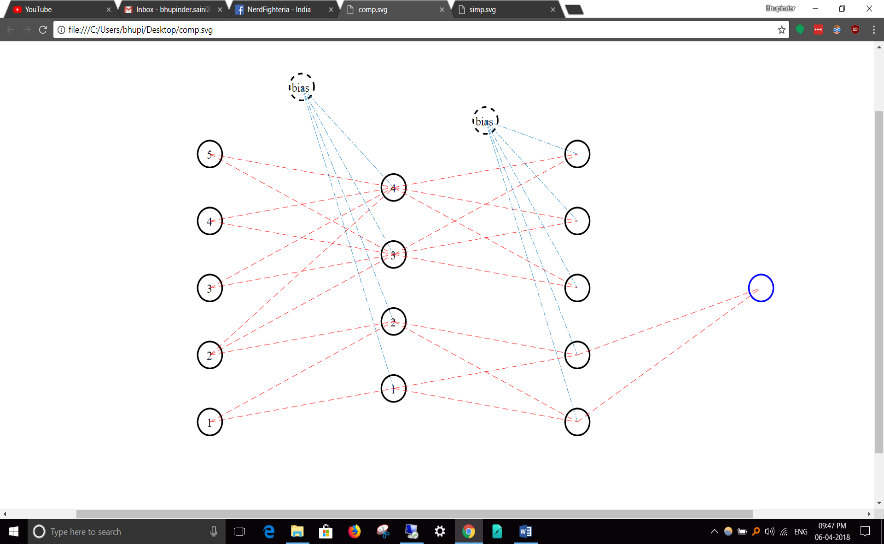
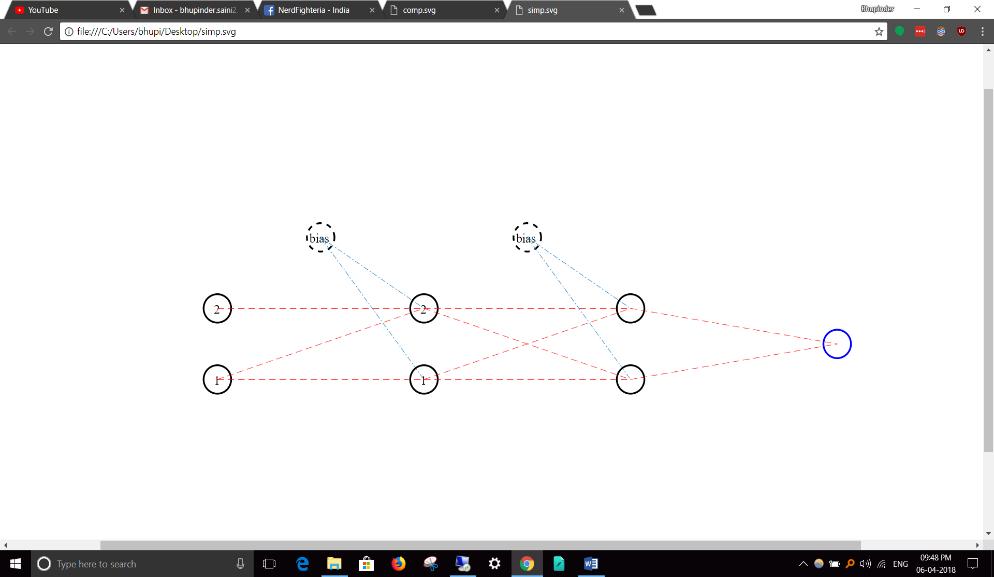
Population{3}[Pop\_size,x2+1,x3]

Population{Num\_layers}

Complexity and Error

Earlier ANN algorithms defined complexity as the number of active (non-zero) connections between the first two layers. While that is a good measure of complexity for ANNs with one hidden layer, the definition cannot be generalized to DNNs. Firstly, a complex looking DNN can actually be much simpler than it looks if there are some inactive connections. An extreme case is shown in the figure below. Both DNNs are equivalent, however, one looks much more complicated than the other. Moreover, weaker connections contribute very less to the final output, and must not be treated equivalent to stronger connections while measuring complexity. Both of these problems can be overcome by using the following definition of complexity:

Where is the complexity, and is the matrix of the weights of the connections between the ith and (i+1)thlayer. As the product function is used, any inactive connection will discount all the connections that precede or succeed it. It also ensures that connections with larger magnitude of weight values contribute more to the final complexity value.



Root Mean Square Error (RMSE) is the definition chosen in EvoNN and BioGP for error calculations.

Where is the expected value of the objective, is the output of the model, and is the total number of data points. EvoDN and deepRVEA also use this definition but provide the choice to use the Logistic Regression Cost Function definition of error as well, which has a special characteristic being globally convex, and thus having only one minima (insert citation).

Crossover and Mutation

While the crossover and mutation techniques discussed in <previous chapter> are effective for single layer neural networks, they are inefficient when many layers and nodes come into the picture. Combined with large population sizes and big datasets, the computational complexity of those techniques make optimization practically infeasible. To tackle those problems, new definitions of crossover and mutation have been proposed which either drastically reduce the time complexity of the process without changing the overarching mathematics, or by using simpler mathematical processes to achieve similar outcomes as that of the previous techniques. To achieve crossover between two individuals, instead of going over each connection and checking for a random probability of crossover, a random number of connections to be swapped is generated by a binomial random number generator and the swap is done parallelly, at once. This is mathematically equivalent to the older crossover technique but is much more time efficient. For mutation a slightly altered version of self-adapting mutation technique is used.

Where, is a constant, is the mutation parameter, and is the self-adaptation factor, as defined in (previous chapter). The definition of was chosen such that it represents the variation in the genetic pool:

Where A and B are randomly chosen individuals. Doing this procedure repeatedly for each connection is time consuming, hence a new definition is proposed:

Where is a random integer, and is the standard deviation in the population. This calculation only has to be done once per generation and hence is relatively very efficient. And like the previous definition, this definition also represents the variation in the population.

3. Evolutionary Deep Neural Networks And deepRVEA

EvoDN and deepRVEA are based on the previously described algorithms EvoNN and RVEA. They create and train DNN models on discrete data using the Predator-Prey Genetic Algorithm and Reference Vector Guided Evolutionary Algorithm respectively. Many changes, some subtle some large, were made to the EvoNN code for optimization purposes and to make training of deep models feasible in an acceptable timeframe.

Pseudocode: EvoDN



Pseudocode: deepRVEA

