Module Code: CS3AI18
Assignment report title: Project report
Student Number: 26011251
Date of Completion: 05/03/21
Actual time spent on the assignment: 35
Assignment evaluation:

1. Following the pseudo code from lectures and online tutorials made the assignment relatively straight-forward
2. Completing the code for the continuous optimisation was the hardest part, after that the rest was just a matter of editing a few of the functions for the combinatorial optimisation and the knapsack problem
3. The assignment has been very beneficial as I am going to use the genetic algorithm in my final project

Abstract:

The report details the implementation of genetic algorithms for use in solving continuous and combinatorial optimisation programs. The genetic algorithm is a search heuristic which "reflects the process of natural selection where the fittest individuals are selected for reproduction in order to produce offspring of the next generation" [1]. The algorithm is able to solve both problems consistently. The performance of the algorithm has been tested through the variation of its parameters, and the optimal setup has been found. Comparisons have also been made between the performance of a ranked roulette selection system and a more basic top 10 approach.

Task 1: Continuous optimisation implementation

Subtask 1.A.

The first subtask asked for the completion of the genetic algorithm code for the 'sum squares' continuous optimisation problem, expressed below, where $x = (x_1, x_2, ….., x_n)$, and it variables are within the range [*lb, ub*], minimising to values of 0.

$$f(x) = \sum_{I=1}^{n} i x_i^2$$

Six main steps are involved in the algorithm, each coded as a separate function within the python code. The first step of the algorithm is to create a population of individuals. The individual is a list of n genes, each generated randomly within the range of [*lb, ub*]. A population of individuals is created according to the population size.

Eg. $ind_1 = [\ 3, 5, 10, -2\ ]$

$ind_2 = [\ 4, -6, -1, 7\ ]$

$ind_n = [\ x_1, x_2, x_3, x_4\ ]$

where n is 4, the [*lb, ub*] are [-10, 10] , and the population size is n

The fitness of each individual within the population is then computed using the fitness function, in this case the sum of squares, with the genes acting as the x values and the gene length I. The desired target for f(x) is 0 so a lower score is better. As such the code returns negative values for the fitness so when sorted from lowest to highest the best scores (closer to 0) are put at the end of the list, making it personally easier to understand. Three other fitness methods have also been written for comparison between each generation of individuals, returning: the average fitness of a generation, the best fitness value of a generation, and the gene combination for the fittest individual.

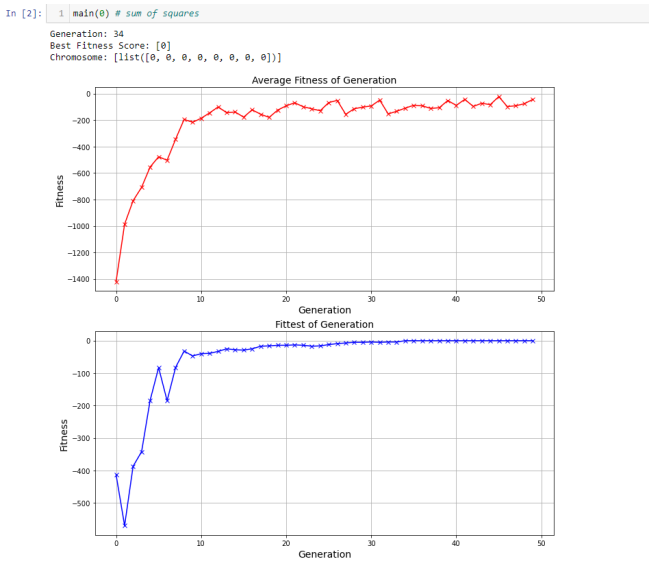
After calculating the fitness score for each individual, a desired number of 'strong parents' are randomly selected from the population. The algorithm makes use of ranked roulette selection. Each individual is given a weighting based upon how their fitness score is placed within a ranking, with fittest individuals having a greater probability to be picked than the worst. Ranked selection helps avoid premature convergence, defined as "when the genes of some high rated individuals quickly attain to dominate the population, constraining it to converge to a local optimum" [2]. Maintaining genetic diversity is key to make sure the parents are able to produce fitter children in order to find the solution to the optimisation problem, as such the ranking system is utilised before using a roulette wheel selection of the parents.

The next stage is the crossover of the genes to create the next generation of individuals. While the number of children is less than the desired amount, two parents are picked from the selection and have their genes mixed. The crossover rate determines whether crossover will take place, or whether the child will be a ‘clone’ of the mother or father. If crossover does take place, a point along the chromosome will be randomly chosen; the father’s genes to the left of the point will combine will the mother’s genes to the right of the point, forming the child’s chromosome. Similarly the father’s genes to the right of the point may be combined with the mother’s genes to the left of the point. `The crossover point and the side the parent’s genes are taken from are chosen randomly so the children are not always made from an even split of each parent, and so their genes are not taken from the same side of each parent each time. This acts to increase the genetic diversity of the next generation.

In order to cultivate further genetic diversity, each child is passed through the mutation function. Each gene within a child has a chance to mutate, determined by the mutation rate. If the gene is selected for mutation, it will be recalculated within the original bounds. Mutation ensures that the children are sufficiently different enough from their parents and one another, increasing the likelihood of finding the optimal combination of genes over subsequent generations.

The last step of the algorithm is to survive the children into the next generation. An excess of children are created during crossover so that the fittest can then be selected, as done before with their parents. This differs from the standard process in Genetic Algorithms where the number of children created is the same as the number of parents [3]. Borrowing from Evolutionary Programming practice, creating an excess of children and then selecting according to their fitness allows for a larger pool of genetically diverse candidates to pick from than if only the required number of children were created during crossover.

The surviving children then become the next generation of parents. The process is repeated until the limit to the number of generations is reached.



From the results it is clear that the algorithm is able to solve the minimisation problem, achieving the target of 0, and the expected values for x. With a population size of 50, a strong population size of 10, a crossover rate of 8 (80% crossover chance), and a mutation rate of 2 (20% mutation chance), the algorithm achieved the target after 34 new generations, though from the shape of the graph it is clear that it started to achieve consistently low best/average fitness scores after only 10 new generations.

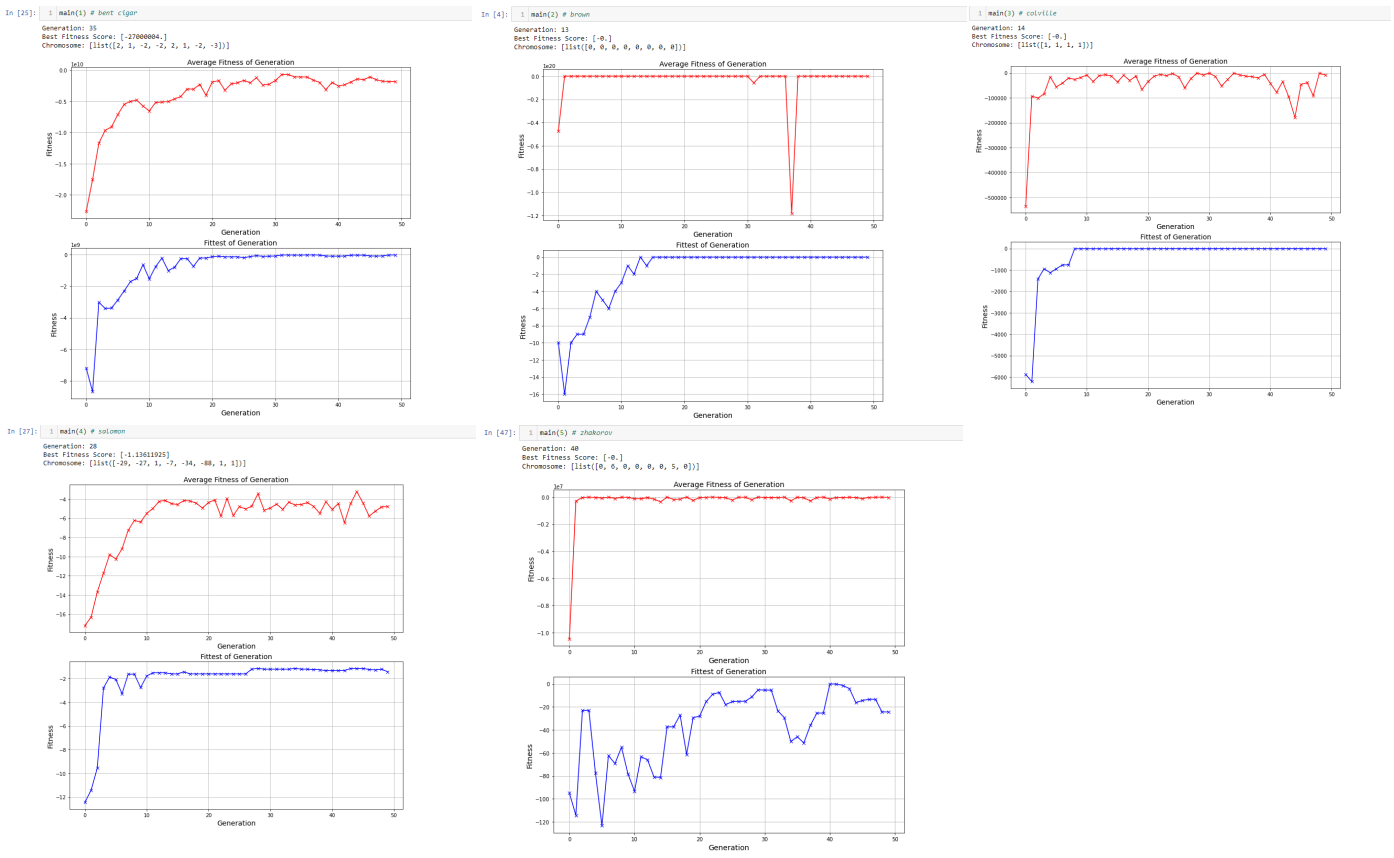
Subsequent runs of the algorithm produced solutions after 43, 33, 21, 45, and 31 new generations, showing that the algorithm will not always produce the correct solution at the same rate; this is to be expected as

the random nature of the selection, crossover, and mutation, lead to diverging scenarios each time the algorithm is run.

Subtask 1.B.

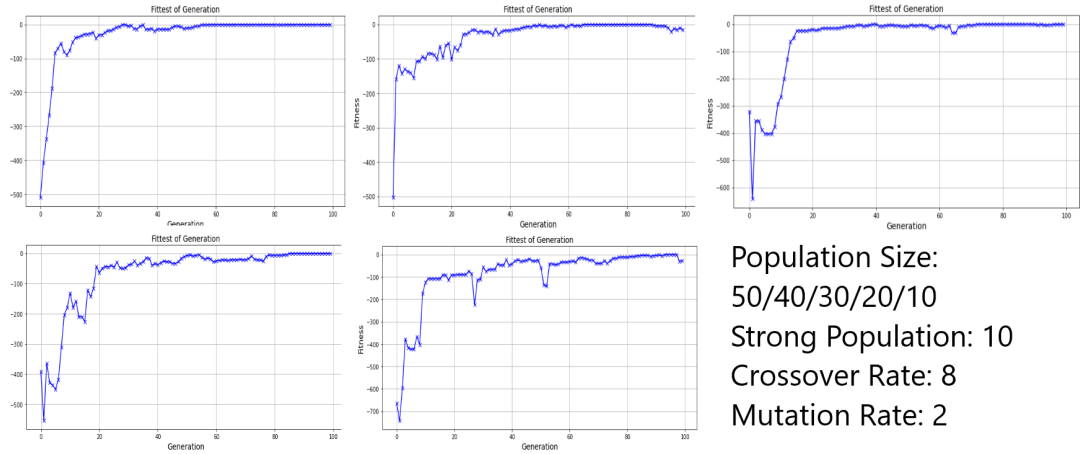
The second subtask required the implementation of five other continuous optimisation functions. From a GitHub repository of optimisation problems [4], five were chosen and implemented as fitness functions within the code: bent cigar, brown, colville, salomon, and zhakarov. The implementation of the other functions only required adjustments to the fitness methods and the addition of a way to change the genome length, and the lower and upper gene bounds. Only the colville function had a set genome length and minimised to (1,1,1,1), the others were all set to a genome length of 8 and minimised to (0,….,0).
Of the five additional functions, brown, colville, and zhakarov (the latter with unexpected genes of 6 and 5 instead of 0), where able to find solutions to the optimisation problems within 50 new generations.

The bent cigar and salomon solutions converged on fitness scores very close to their targets of 0, suggesting that in additional runs with more optimal variables (population size, strong population etc.) they perhaps would have been able to find the optimal solutions.



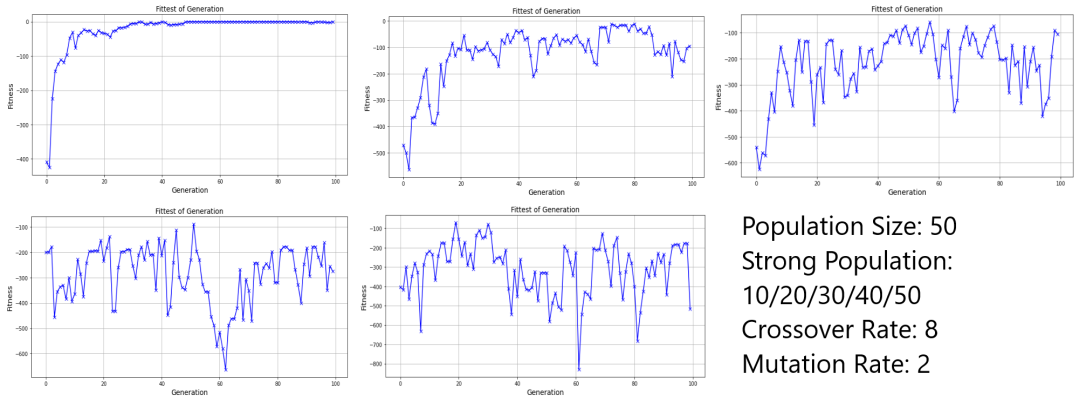
Subtask 1.C.

Lowering the population size by 10 in each subsequent run has lowered the rate at which the algorithm converges on the target of 0. From the shape of the graphs it is clear that a decrease in population makes improving the fitness score harder as there is less genetic material, and consequently less genetic diversity from which to select the best performing individuals from.



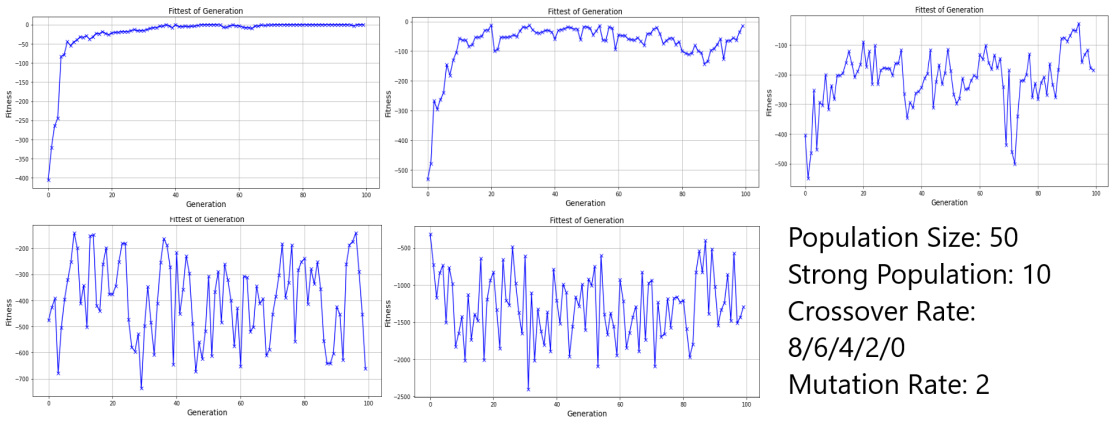
Population Size: 50/40/30/20/10
Strong Population: 10
Crossover Rate: 8
Mutation Rate: 2

Increasing the number of selected individuals has shown to make the fitness level fluctuate greatly between generations. With a strong population the same size as the total population (no selection) , the best fitness score fails to converge; without the selection of the strongest of individuals, the subsequent generations cannot improve.



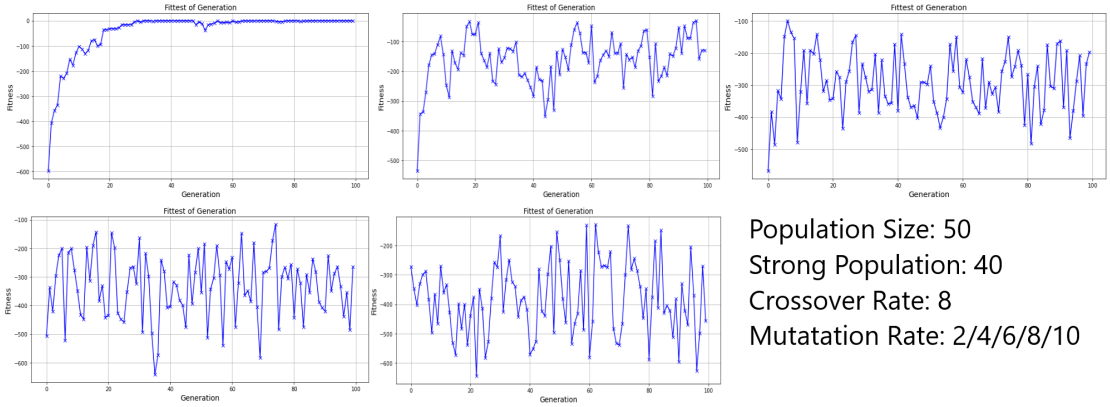
Population Size: 50
Strong Population: 10/20/30/40/50
Crossover Rate: 8
Mutation Rate: 2

With a decreasing crossover rate (fewer unique children), the fitness of each generation also fluctuates greatly, failing to converge and find the correct solution. If unique children cannot be created through crossover, the likelihood of creating a new and more successful gene combination is low, as such, it is important to maintain a high crossover rate.



Population Size: 50
Strong Population: 10
Crossover Rate: 8/6/4/2/0
Mutation Rate: 2

As the mutation rate increases the effective of the algorithm greatly diminishes, just as with an increasing selection size or increasing crossover rate. It the mutation rate is high the chromosome of a child will be changed too much, nullifying its creation from a pool of best performing parents. Assigning a low mutation rate is key for a successful genetic algorithm as it ensures genetic variance amongst the children without changing their genes too much as to completely change their performance.



Population Size: 50
Strong Population: 40
Crossover Rate: 8
Mutatation Rate: 2/4/6/8/10

The testing leads to the conclusion that a large population, with a small number of selected parents, subjected to a high rate of crossover, and a low rate of mutation, is the optimal setup of parameters for the algorithm to correctly solve the sum of squares optimisation problem.

Subtask 1.D.

Though the algorithm works well in solving the sum of squares optimisation problem, alternate methods of selection exist which may improve upon the results. The algorithm already implements ranked roulette selection to improve the selection of a genetically diverse set of the fittest individuals. Borrowing from Evolutionary Programming practice, an excess of children is created as that allows for a second round of selection between generations, allowing for a greater likelihood of selecting appropriate individuals.

A somewhat crude method of selection is simply choosing the top performing individuals from the population, without random selection. With a strong population selection of 10, the top 10 fittest individuals will be selected from the population. Though this may lead to less diversity found when included weaker individuals, it may lead to a quicker discovery of the optimal solution. The results over five runs of ranked roulette selection and five of top 10 selection do indeed support this hypothesis. The average run implementing top 10 selection was found to have shortened the number of generations required to find the optimal solution by nearly 50% compared to ranked roulette selection, solving the optimisation problem after 12 generations as opposed to 22.

Generation to Achieve Target Fitness (popSize 50, strongPop 10, crossoverRate 8, mutationRate 2, generations 100)

	Attempt 1	**Attempt 2**	**Attempt 3**	**Attempt 4**	**Attempt 5**	**Avg (rounded)**
Ranked Roulette Selection	Gen 20 Fitness 0	Gen 29 Fitness 0	Gen 16 Fitness 0	Gen 27 Fitness 0	Gen 19 Fitness 0	Gen 22 Fitness 0
Top 10 Selection	Gen 10 Fitness 0	Gen 14 Fitness 0	Gen 10 Fitness 0	Gen 17 Fitness 0	Gen 8 Fitness 0	Gen 12 Fitness 0

Subtask 2.A.

With the algorithm already completed for the continuous optimisation problem, only small adjustments are required to configure the code for the combinatorial optimisation problem. The function 'sum 1s in a binary string' is expressed below, where $x = (x_1, x_2, ..., x_n)$, and its variables are within the discrete set {a, b,…}.

$$f(x) = \sum_{I=1}^{n} x_i$$

For the minimisation and maximisation to 0 (x = (0,0,…,0)), and 1 (x = (1,1,…,1)) respectively, the discrete set is {0,1}. With a population generated (gene values either 0 or 1), the parents are again selected via the ranked roulette method, the fitness is calculated differently however. For the minimisation to 0, the individuals' fitness scores are calculated from the number of 0 genes within the chromosome, for a maximum fitness score equal to the gene length. For the maximisation to 1, the individuals' fitness scores are calculated from the number of 1 genes present within the chromosome, for a maximum fitness score equal to the gene length.

The rest of the algorithm works the same way as for the continuous optimisation problem, the only key difference found within the mutation method. Instead of randomly calculating the new gene, bit-flipping takes place, the 0s converted to 1s and vice versa.

The algorithm is able to successfully solve for both the minimisation and maximisation of the function. As there are only two variables within the discrete set, the optimisation problem is very easy to solve; both the optimal solution for the minimisation and optimisation of the function can be found within the first generation of new parents, there is even a good chance of creating the optimal individual (0,0….,0)/ (1,1…,1) within the initial population.

Subtask 2.B.

Three additional combinatorial optimisation problems have been implemented within the code. The first is for guessing the correct combination against a generated passcode. Like the board game 'Mastermind', each individual is generated with genes in the range of 0-9. The fitness is calculated by how many of the genes are the right number in the correct position relative to the passcode. The algorithm is able to consistently solve the problem, guessing the correct four-digit passcode well within the 100 generation limit each time.

The second new implementation is for the tension/compression spring design problem, minimising the volume of a coil spring under a constant tension/compression load. The three gene variables within the function are: the number of spring's active coils, the diameter of the winding, and the diameter of the wire. Each variable has its own discrete set of values. The algorithm is able to find a solution for a fitness value of 0.00025625, close to the optimum solution of 0.0126652327883 [5].

The third implementation is of the welded beam design problem. The system cost of welding a beam to a rigid member is minimised from thickness of the weld, length of the weld, length of the material, and thickness of the material. Running the algorithm for the function and parameters results in a minimised value of 0.00788822 every time; the code in its current state cannot properly solves this function.

Subtask 3.A.

The code has been reconfigured to solve the combinatorial knapsack problem. Given a set of items, each with a set weight and value, how must a bag be packed so that the total weight limit is less than or equal to the bag's limit, and the total value of the items is as large as possible?

With a list of five items with weights and values, the population of individuals is generated with binary gene values. For each 1 gene within the chromosome, the weight and value of the items are added to a running total. If the weight of the chromosome exceeds the weight limit, a fitness value of 0 is return, else the fitness score is returned as the value of the bag. The chromosome with the greatest value is the fittest individual.

The algorithm is able to consistently determine the optimal solution within 10 generations. Solving the problem is imperative for goods shipping businesses as maximising the value of each container reduces storage space wastage, conferring higher profits per delivery.

Subtask 3.B.

Genetic algorithms can be applied to configure the most appropriate hyper-parameters for neural networks. The first stages are: create a population of neural networks, assign random hyper-parameters within a range to each neural network, and then train the networks either simultaneously or one by one. The fitness of each network is determined by its training cost, with the lowest training cost designated as the fittest and vice versa. Through following the basic steps of the genetic algorithm and creating subsequent generations, the hyper-parameters for lowest training cost can be found and the network optimised.

One major downside of training neural networks via genetic algorithms is that running many neural networks is very resource intensive, both time-wise and resource-wise [6]; only those with the time and money can realistically run many large neural networks simultaneously. Another issue is that of overfitting. While the genetic algorithm may find optimum settings for the neural network's current use-case, the network may not be optimised for general use [7]; if the network is too specifically tuned, it will underperform when applied to other scenarios.

Bibliography:

1. Vijini Mallawaarachchi Introduction to Genetic Algorithms — Including Example Code 08/07/17
2. Elena Simona Nicoară Mechanisms to Avoid the Premature Convergence of Genetic Algorithms 2009
3. Dr Varun Ojha Artificial Intelligence CS3AI18 / CSMAI19 Lecture - 2/10: Problem Solving (Evolutionary Algorithms)
4. Nathan Rooy GitHub repository
5. Yuksel Celik 1 [0000-0002-7117-9736] and Hakan Kutucu1 [0000-0001-7144-7246]
Solving the Tension/Compression Spring Design Problem by an Improved Firefly Algorithm
6. Suryansh S. Genetic Algorithms + Neural Networks = Best of Both Worlds 26/03/18
7. Backpropagation vs Genetic Algorithm for Neural Network training