ML Assignment 2

Kuhu Gupta

1 Finding weights for Neural Network

1.1 Introduction

The first part of the assignment is about using randomized optimization to find the possible weights for a specific neural network. In assignment 1, I have used backpropogation to find optimal parameters for a neural network. The neural network took 6 features as input for US Permanent Visa Application to forecast the the outcome of the application into two categories, i.e accepted and denied. After running several tests, I found the optimal parameters of: 6-node input layer, one hidden layer with 100 nodes, one output node and about 500 iterations.
In assignment 2, it is mentioned to use three optimization methods for finding good weights:

- Random Hill Climbing
- Simulated annealing
- Genetic Algorithms

Each optimization techniques is discussed in detail in further sections including plots and analysis. Since I used US Permament Visa dataset in my past assignment I have used the same dataset for this assignment but there is one thing important to note that it tends to converge quite quickly because of the size of the dataset as well as consistent distribution.

1.2 Back-propagation Neural Network

The first algorithm used is back-propogation. Backpropagation is shorthand for "the backward propagation of errors," and works by computing an error at the output and distributed backwards throughout the network's layers. The error is minimized by calculating the error over various iterations. As discussed in assignment 1, the permanent visa is rather large and robust. It is quickly learnable by various different learners and in such backpropogation found significant success.

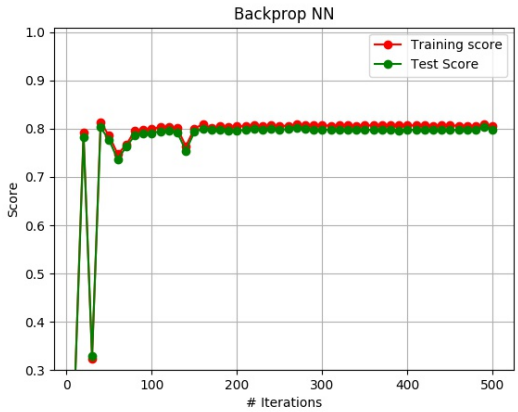

Figure 1: Backpropogation Success Rate vs. Iterations

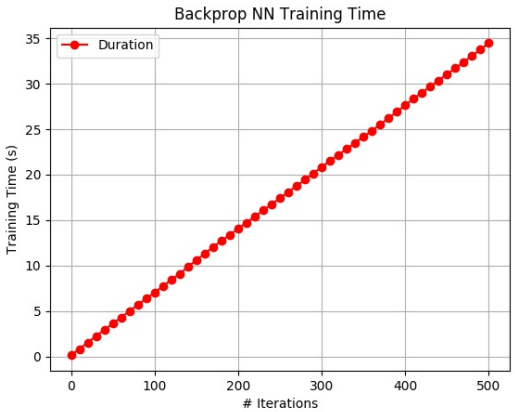

Figure 2: Backpropogation Training Time

As seen in figure 1 that around 50 iterations, the network behind to converge at around an 80% success rate. Seeing as the training and test core track either rather closely, it is apparent that the dataset is rather robust and consistent. One thing to note is that the training time scales linearly with the number of iterations which makes sense since the same amount of calculation with similar complexity are performed on each iteration of backprogogation.

1.3 Randomized Hill Climbing

The second optimization algorithm used is randomized hill climbing. Randomized hill climbing involves finding the optima by exploring a solution space, taking up random starting point and then moving in the direction of increased fitness on each iteration. Since we are trying to find optimal weights for a neural network, randomized hill climbing selects random weights and then moves in a direction so as to try to find a better result for that weight by trying to move up an optimization 'success hill'. If moving in both direction is not giving result it flips a coin to decide a direction. One important thing to take in consideration is that we are using randomized hill climbing, not random restart hill climbing. Therefore, there is a chance of the algorithm being caught at local maximas.

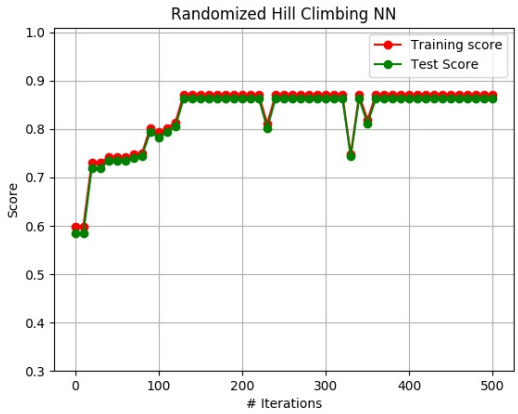

Figure 3: Randomized Hill Climbing Success Rate vs. Iterations

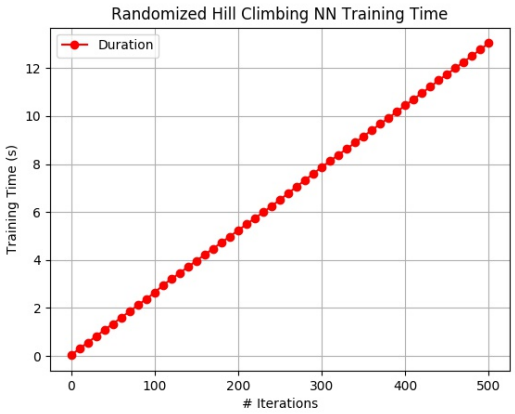

Figure 4: Randomized Hill Climbing Training Time

As seen in figure 3, a high training and test score is achieved at 130th iteration. This is can be subjected to random, but looking at the scores after the 130th iteration, the results seem to be rather consistent and stable. On observing the scores in figure 3, it is clearly visible that at various instances of randomized hill climbing, it stucks in local optimas and being unable to escape, such as 230th, 320th and 350th iterations. As seen in Figure 4 and similar to backpropogation, training time scales linearly with the number of iterations run. However, the training time is faster using randomized hill climbing as there will be reduction in calculations necessary. This happens because backpropogation requires calculations to minimize error by moving backwards through the network, on the other hand randomized hill climbing move in one direction which to find whether the new weights are optimal or not.

1.4 Simulated annealing

The third weight finding algorithm is simulated annealing. Simulated annealing involves running for a finite set of iterations, by taking up a random point and then sample in the neighborhood. After comparing the alternatives to the original solution, the algorithm will decide to either stick with the original point or move to the new one. The use of temperature and cooling parameter, clearly affects the learning. As the temperature increases, it more kind of a random walk while as it reduces it is more of hill climbing. At first, the algorithm is more open to worse solutions but eventually moves towards only accepting better solutions. I have tried multiple cooling parameters, such as, 0.15,0.35, 0.55 and 0.7 to find the best simulated annealing approach. Figure 4,5 and 6 shows the learning curves by number of iterations for simulated annealing approach with the above mentioned cooling parameters.
The cooling parameter of 0.15 and 0.35 take much longer to converge to the optimal solution than did a higher parameter. However, it could also be attributed to luck that the algorithm could have randomly found itself in an optimal solution earlier on and part of it is the also the way algorithm operates by its parameters. Moreover, the .7 cooling temperature here proved most effective.

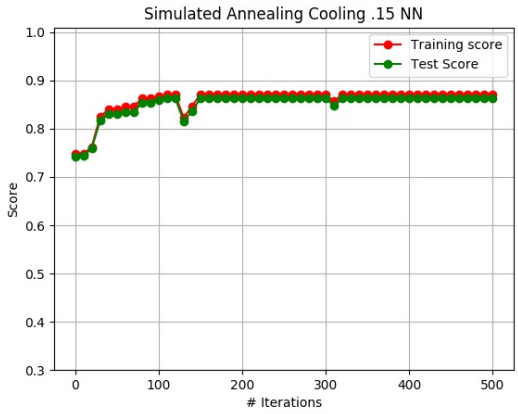

Figure 5: Simulated Annealing Success Rate vs. Iterations for 0.15 Cooling

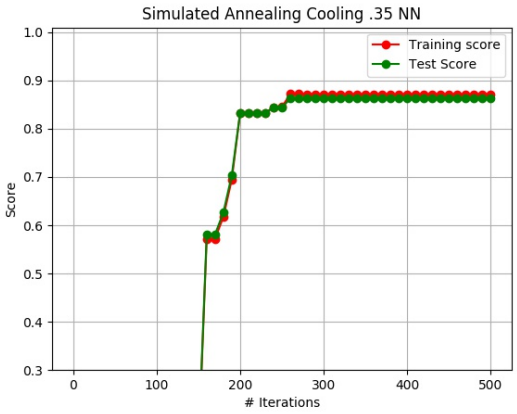

Figure 6: Simulated Annealing Success Rate vs. Iterations for 0.35 Cooling

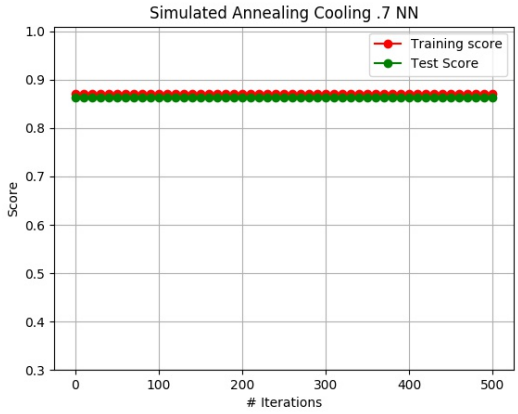

Figure 7: Simulated Annealing Success Rate vs. Iterations for 0.7 Cooling

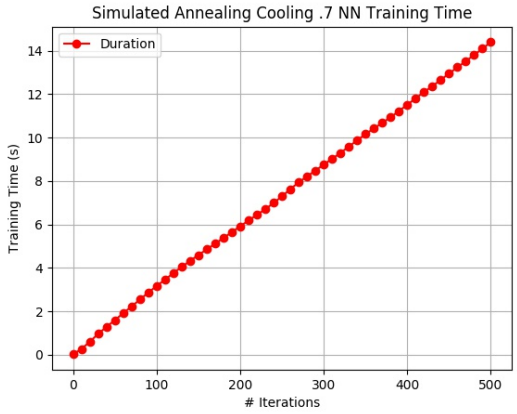

Figure 8: Simulated Annealing NN Training Time for .7 Cooling

1.5 Genetic Algorithms

The fourth weight finding algorithm is genetic algorithms. Genetic Algorithms simulate the process of natural selection which means those species who can adapt to changes in their environment are able to survive and reproduce and go to next generation. In simple words, they simulate "survival of the fittest" among individual of consecutive generation for solving a problem. The algorithm involves working with an initial solution and then making modification in an an attempt to improve the solution. These modifications are done in three ways, which are, Selection (give preference to the individuals with good fitness scores and allow them to pass there genes to the successive generations), Crossover(two individuals are selected using selection operator and crossover sites are chosen randomly. Then the genes at these crossover sites are exchanged thus creating a completely new individual) and Mutation(insert random genes in offspring to maintain the diversity in population to avoid the premature convergence). In the context of a neural network, we can use genetic algorithms to make modifications to our network's weights.

For the assignment, a population size of 50 is used to get a diverse initial sampling of possible solutions. However, I tried to vary the number of instances to mate (aka crossover) and to mutate throughout the trials.

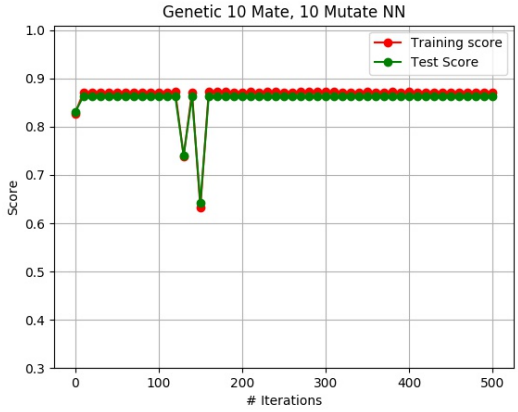

Figure 9: Genetic Algorithm Success Rate vs Iterations with 10 Mate, 10 Mutate, 50 population

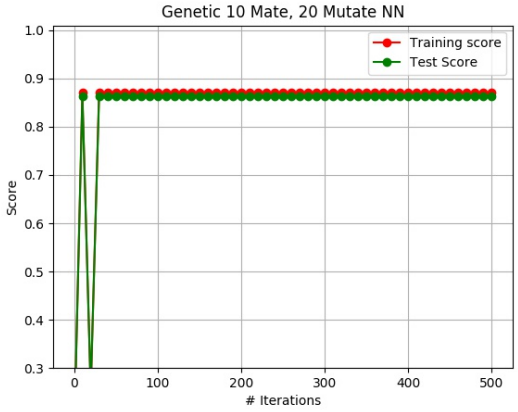

Figure 10: Genetic Algorithm Success Rate vs Iterations with 10 Mate, 20 Mutate, 50 population

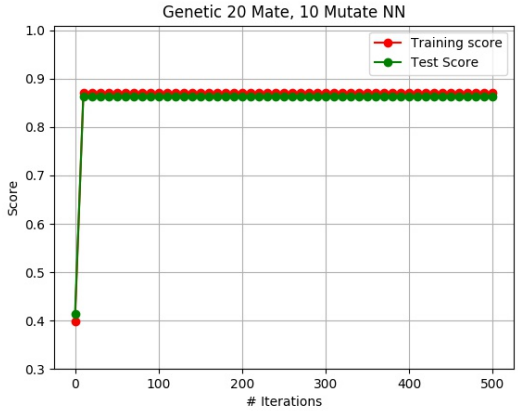

Figure 11: Genetic Algorithm Success Rate vs Iterations with 20 Mate, 10 Mutate, 50 population

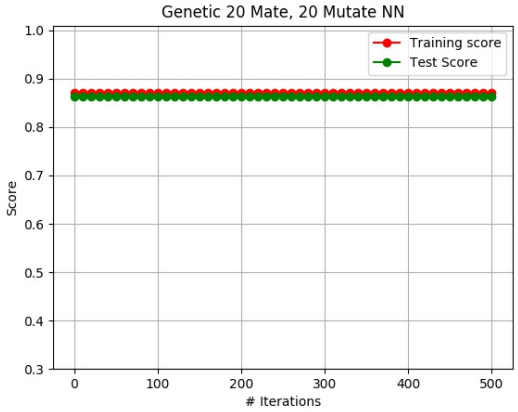

Figure 12: Genetic Algorithm Success Rate vs Iterations with 20 Mate, 20 Mutate, 50 population

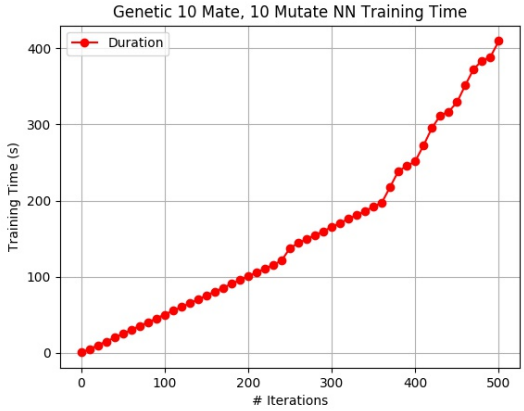

Figure 13: Genetic Algorithm Training Time for 10 Mate, 10 Mutate, 50 population

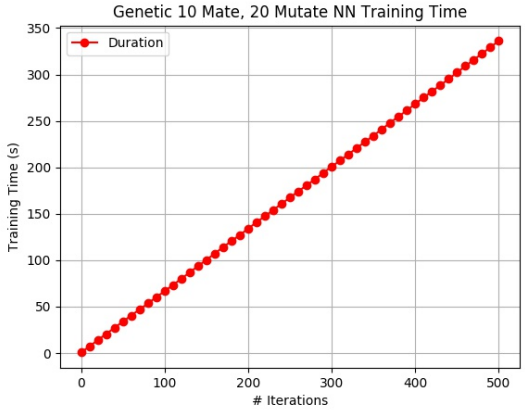

Figure 14: Genetic Algorithm Training Time for 10 Mate, 20 Mutate, 50 population

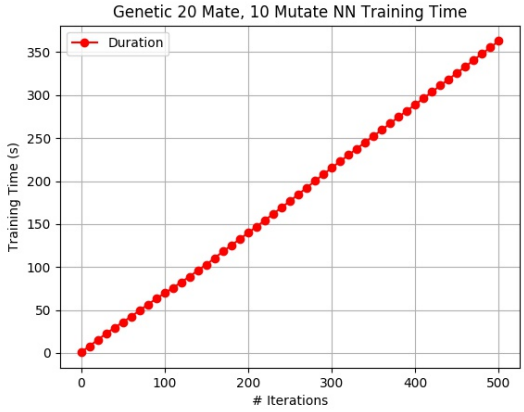

Figure 15: Genetic Algorithm Training Time for 10 Mate, 10 Mutate, 50 population

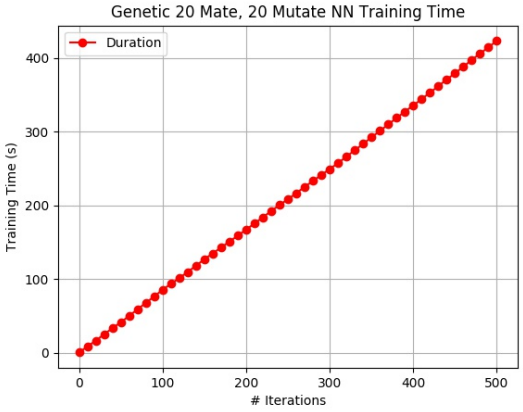

Figure 16: Genetic Algorithm Training Time for 10 Mate, 20 Mutate, 50 population

I have used different combinations of mate/mutate for my trials, which are, 10/10, 10/20, /20/10, and 20/20. All these trials were run with a population size of 50. As observed in Figure 9,10,11 and 12, both mating and mutating appeared to be highly effective in finding the optimal solution. In Figure 9, the optimal solution is converged quite quickly, though it gets stuck in local optima at around 120th and 160th. Also in Figure 10, it converges real quickly. Overall, the optimal solution converged quite quickly in under 30 iterations for each trial. This is partly due to the fact that the data is easily learnable and does not appear to maintain very many trapping local maximums.
Moreover, the training times scaled roughly linearly with the number of iterations ran. This does make sense because, from a performance perspective, the number of mutation calculations from iteration to iteration is more or less the same.

1.6 Conclusion

All the optimization algorithms produced more or less the similar results. However, what stood among all the factors was the amount of training time required, the tuning of parameters, and how many iterations were required to achieve a stable, optimal solution. All together, the genetic algorithm approach consistently produced the best results for my network. This makes sense, as the data was rather homogeneous and there appeared to be very few outliers. By learning the training data well, the learner was able to perform similarly well on the (nearly identical) test data.

2 Optimization Problem

The second part of the assignment involves three differnt optimization problems

- Continuous Peaks
- The traveling salesman
- Flip Flop

And the four optimization techniques from are used:

- Random Hill Climbing
- Simulated annealing
- Genetic Algorithms
- MIMIC

MIMIC is similar to the other algorithms used in part 1 of the assignment. It works to find the globally optimal solution. On the contrary to the other algorithms, it retains knowledge and structure of previous iterations and uses this information to more efficiently find better solutions. MIMIC is particularly strong in regards to problems that maintain patterns between subsets of their parameters across iterations. In the following sections, I will apply each of the optimization algorithms to each of the three optimization problems and carefully examine how each optimization problem's characteristics may favor one algorithm to another, as well as evaluate the efficiency of each.

2.1 Continuous Peaks

The continuous peaks problem is an extension of the four peaks problem that allows for a wide variety of local maximums. It contains many local optima in a 1D space This algorithm is particularly interesting due to the potentially large number of local maximums. Although this problem set was chosen for its simplicity, it highlights differences between random optimization algorithms well and applies to other examples like topography and optimization of surfaces. It is intuitive for a human to observe the global optima visually, however using random optimization and without processing each data point, the different performance between the random optimization algorithms perform is quite evident.

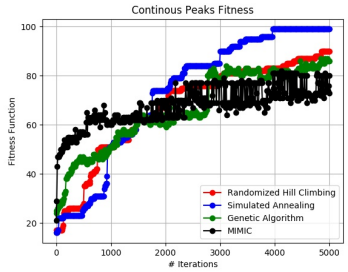

Figure 17: Fitness vs Number of Iterations of 4 Optimization Strategies

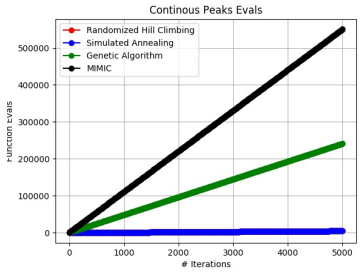

Figure 18: Function Evals vs Number of Iterations of 4 Optimization Strategies

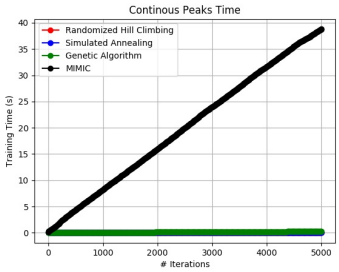

Figure 19: Training Time vs Number of Iterations of 4 Optimization Strategies

In order to ensure accuracy and a fair distribution, 5 different trials were used of 5000 iterations each. Each optimization strategy was also allowed to evaluate over a parameter range in an attempt to find the optimal parameters for that strategy.

For randomized hill climbing, there were no other parameters to tune. The algorithm was simply run for various iterations (multiples of 10) in a set of 5 trials. In terms of fitness function performance, it actually performed quite well and required no significant function evaluations. It was primarily fast to train due to the extreme simplicity of the algorithm itself since algorithm is geared towards climbing hills (which is what Continuous Peaks is doing), but it is much quicker because in constant risk of getting stuck with a local optimum and quit early.

The optimization problem of continuous Peaks, does not work well with Genetic Algorithms, as the problem itself does not lend itself well to mutation and creating multiple populations is not that advantageous. Here, I have used a population size of 100 and various configurations of 50, 30, and 10 mutations/mate combinations. Figure 17, 18 and 19 depicts the success rates when using 30 mutation and 30 matings. The performance of GA looks similar to randomized hill climbing which roughly makes sense due to the random nature of both algorithms. The number of function evaluations was higher for the genetic algorithm due to the amount of attempts at mutating and mating whereas the training time was practically irrelevant.

Mimic's parameters were set to 100 samples and a 'to-keep' number of 50. The threshold for the discrete dependency tree mimic used was sampled from the range 0.1, 0.3, 0.5, 0.7, and 0.9. While MIMIC started of relatively strong in terms of fitness performance as compared to other, however, it keep oscillating around the 60-80 range for the most part. It seems that MIMIC is facing difficulty on the continuous peaks due to the large amount of randomness in the data. The number of functional evaluations required for MIMIC was significantly higher than the other algorithms as seen in figure 18, as was its training time as seen in figure 19. Continuous peaks requires lots of iterations, and MIMIC's construction of trees slows it down considerably.

Simulated annealing was slightly more complex, where cooling coefficients of 0.15, 0.35, 0.55, 0.75, and 0.95 were used. It was found that the coefficient 0.15 was the most effective here. This low cooling rate was effective because it helped to slowly overcome any local maximums but not work too aggressively to skip any optimal solution. As seen in figure 17, around 3000k, Simulated Annealing took a clear lead over the other solutions in terms of fitness performance. Similar to randomized hill climbing, its number of function evals and training time was extremely low.

Overall, Simulated Annealing proved to be the best strategy for the continuous peaks problem as seen in Table 1, largely due to it's ability to handle large amounts of randomness well and to avoid getting caught in such local maximums. It's low training time and number of function evaluations further aided its case.

Iterations	Fitness	Training Time	Function Evals
0	16.0	5.5038e-05	11
10	16.0	7.64649999999e-05	21
50	22.0	0.000212433	61
100	22.0	0.000370814	111
200	23.0	0.000717487	211
300	23.0	0.001094822	311
500	26.0	0.001666152	511
750	31.0	0.002682947	761
1000	51.0	0.003609291	1011
1500	62.0	0.005517988	1511
2000	74.0	0.007139624	2011
2500	84.0	0.008715217	2511
3000	90.0	0.010594833	3011
3500	95.0	0.012718942	3511
4000	99.0	0.014618306	4011
5000	99.0	0.018770859	5011

Table 1: Simulated Annealing Results w/ .15 Cooling

2.2 The traveling salesman

The Traveling Salesman Problem (TSP) is a problem of minimizing the distance a salesman must travel to visit all cities. The generalized version is essentially finding the minimum distance to traverse all nodes within a graph of subgraph, where each edge has a certain distance. This problem is particularly interesting due to it's real-world implications, such as route planing for delivery trucks and everyday errands. It also has no known polynomial time solution

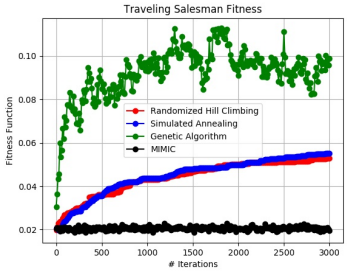

Figure 20: Fitness vs Number of Iterations of 4 Optimization Strategies

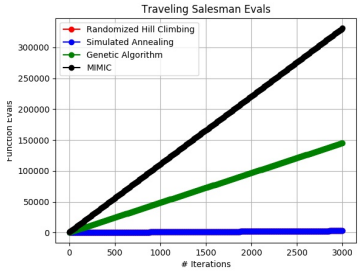

Figure 21: Function Evals vs Number of Iterations of 4 Optimization Strategies

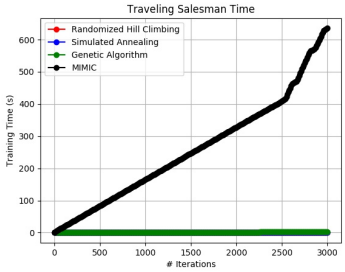

Figure 22: Training Time vs Number of Iterations of 4 Optimization Strategies

For this optimization problem, in order to ensure accuracy and a fair distribution, 5 different trials were used of 3000 iterations each. 50 Random points were generated and used as the destinations for our faux salesman. Each optimization strategy was also allowed to evaluate over a parameter range in an attempt to find the optimal parameters for that strategy.

For randomized hill climbing, there were no other parameters to tune. The algorithm was simply run for various iterations in a set of 5 trials. In terms of fitness function performance, it performed mediocre and required no significant function evaluations. It was similarly fast to train due to the extreme simplicity of the algorithm itself. The reason for the strategy's mediocrity can easily be attributed to the fact that random guessing does not perform well when trying to develop the most optimal path due to the large number of possibilities and low number of optimums.

Similar to the last problem, simulated annealing was slightly more complex, where cooling coefficients of 0.15, 0.35, 0.55, 0.75, and 0.95 were used. It was found that the efficient 0.55 was the most effective here. Interestingly enough, the simulated annealing approach very closely tracked the randomized hill climbing results. This was pretty cool, and clearly demonstrated the effect of the problem on the ability of a solution to perform well. As both were prone to picking random solutions and making marginal improvements on those solutions, they shared the same strengths and weaknesses for trying to plan an optimal path. Similar to randomized hill climbing, simulated annealing' number of function evals and training time was extremely low.

Iterations	Fitness	Training Time	Function Evals
0	0.0305500487733	0.011139866001	573
50	0.0535358597005	0.0619086620027	2984
100	0.0670653896105	0.106549986003	5404
200	0.0754973396223	0.190760427002	10194
300	0.0843530928911	0.276369227002	15021
500	0.0847430696768	0.430181398007	24690
750	0.0824751535861	0.62021527501	36716
1000	0.097111513149	0.799495104007	48778
1500	0.100315033444	1.153653522	72921
2000	0.0908447200981	1.51476069201	97068
2500	0.111066212921	1.875438519	121202
3000	0.0986916191186	2.27946003001	145282

Table 2: Genetic Algorithm Results w/ population 100, 30 mutations, 30 mating

The genetic algorithm employed used a population size of 100 and various configurations of 50, 30, and 10 mutations/mate combinations. The graph above depicts the success rates when using 30 mutation and 30 matings. As can be seen, it's performance was exceeding better than the other algorithms. The mutations and matings were able to successfully identify the most optimal paths and plan extremely well for the hypothetical salesman. This can be attributed to the fact that traveling salesman can be broken into various sub journeys that the genetic algorithm can learn and then attempt to combine for an optimal journey. The number of function evaluations was higher for the genetic algorithm due to the amount of attempts at mutating and mating–whereas the training time was practically irrelevant. The genetic algorithm was the clear winner in terms of performance for the TSP.

Mimic's parameters were set to 100 samples and a 'to-keep' number of 50. The threshold for the discrete dependency tree mimic used was sampled from the range 0.1, 0.3, 0.5, 0.7, and 0.9. MIMIC performed rather poorly on this problem. This was very likely due to the fact that pattern finding is of almost no use in this problem domain and that previous knowledge also is mostly unnecessary when trying to find optimal paths. The number of functional evaluations required for MIMIC was also significantly higher than the other algorithms, as was its training time. This was primarily due to the fact that MIMIC is significantly more complex to operate than the other algorithms. Overall, MIMIC was a very poor candidate for the traveling salesman problem.

Overall, the genetic algorithm proved to be the best strategy for the traveling salesman problem. It's ability to combine sub-journeys and paths into an optimal route was strongly advantageous. The results showed the strong benefit of mutations on this problem. Similarly, it's speed an relative simplicity were also very beneficial.

2.3 Flip Flop

The flipflop problem is a common optimization which involves counting the number of bits that alternate with its next neighbor in a bit string. This problem is interesting as the strings are randomized which leads to a possibilty of significant number of local mimimas and maximas. It is also particularly well suited for algorithms that can handle pattern matching and previous knowledge well.

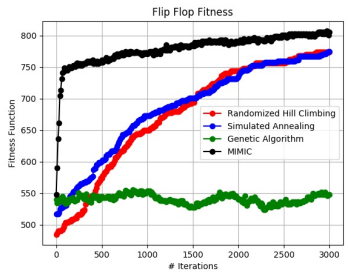

Figure 23: Fitness vs Number of Iterations of 4 Optimization Strategies

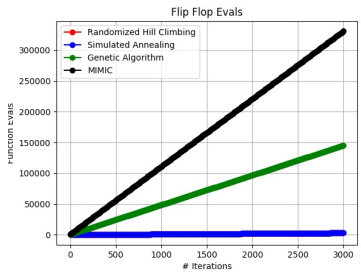

Figure 24: Function Evals vs Number of Iterations of 4 Optimization Strategies

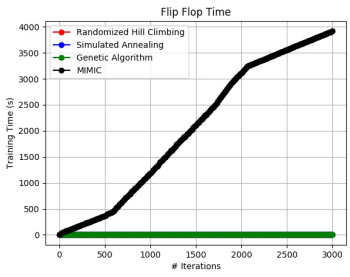

Figure 25: Training Time vs Number of Iterations of 4 Optimization Strategies

For this optimization problem, in order to ensure accuracy and a fair distribution, 5 different trials were used of 3000 iterations each. A bit string of length 1000 was used, where each bit was randomized as either 1 or 0. Each optimization strategy was allowed to evaluate over a parameter range in an attempt to find the optimal parameters for that strategy. The results can be seen in Figure 23,24 and 25.

For randomized hill climbing, there were no other parameters to tune. The algorithm was simply run for various iterations in a set of 5 trials. In terms of fitness function performance, it actually performed rather well with it's performance increasing over time. Randomized hill climbing required very limited function evaluations and it trained quickly due to its simplicity. The reason for the strategy's success can be attributed to the fact that random guessing is somewhat akin to the distribution of the input bit string in that the distribution for the next neighbor should, on average, change.

Similar to the last problem, simulated annealing was very similar to randomized hill climbing. Cooling coefficients of 0.15, 0.35, 0.55, 0.75, and 0.95 were used. It was found that the efficient 0.15 was the most effective in this case. Simulated annealing approach closely tracked the randomized hill climbing results yet again. Similar to hill climbing, the random variations of simulated annealing can roughly identify to the variations of the distribution of the input. Over time, as with randomized hill climbing, the performance of the strategy significantly increased. Also Similar to randomized hill climbing, its' function evals count and training time were extremely low.

The genetic algorithm employed used a population size of 100 and various configurations of 50, 30, and 10 mutations/mate combinations. Figure 23 depicts the success rates when using 30 mutation and 30 matings. The approach performed significantly worse than the others.The genetic algorithm had a very tough time recognizing the pattern that the input tended to follow and the mutations and matings had very little effect on finding an optimal solution. While t number of evals was higher for the this strategy due to the amount of attempts at mutating and mating,it did not help to achieve a better result.

Iterations	**Fitness**	**Training Time**	**Function Evals**
0	548.0	10.292430561	1000
50	713.0	54.323489845	6500
100	745.0	89.781651075	12000
200	753.0	159.161412402	23000
300	755.0	227.402023603	34000
500	761.0	364.872433042	56000
750	771.0	741.847804258	83500
1000	771.0	1186.13889593	111000
1500	785.0	2107.83467502	166000
2000	789.0	3107.1784734	221000
2500	799.0	3557.31754274	276000
3000	805.0	3916.14983514	331000

Table 3: MIMIC Results w/ .5 threshold

Mimic's parameters were set to 100 samples and a 'to-keep' number of 50. The threshold for the discrete dependency tree mimic used was sampled from the range 0.1, 0.3, 0.5, 0.7, and 0.9. MIMIC performed incredibly on this problem–mainly due to its ability to handle patterns very well and to keep track of historical knowledge. The number of evaluations required for MIMIC was definitely higher than the other algorithms, as was its training time, but its performance also far outweighed the other strategies. Overall, MIMIC proved incredibly successful at the flip flop problem.

2.4 Conclusion

Above, four different machine learning optimization strategies were applied to 3 different optimization problems. Interestingly enough, each optimization problem had its own inherit properties that proved more suitable for solving for different optimization strategies.

The continuous peaks problem, with its high amount of randomness and lack of predictable behaviour, proved to be optimized exceptionally well by simulated annealing. Simulated an-

nealing ability to randomly sample nearby solutions with effective cooling thresholds were very helpful for this problem in that it avoided getting trapped in local optimums but reduced the change of skipping over more significant nearby optimums.

The traveling salesman problem, which, in my opinion, proved to be the most interesting, was most aptly solved by a genetic algorithm. Travelling salesman required various sub journeys to be optimized and combined to find an optimal path about various points. Whereas random and pattern solutions failed badly, the genetic algorithm was able to mutate and combine to form optimal solutions quite well.

The final problem, flip flop, had an input that was highly repetitive and tended to follow a patterned distribution. In such, optimization strategies that sought to solve based on subgroups of parameters or previous knowledge performed rather well, whereas strategies that were left to chance or random combinations performed poorly. MIMIC, with it's ability to solve patterns and repetition well, as well as its ability to retain prior experiences, proved to be far and away the most optimal solution for this problem.