**Assignment 2 Analysis  
Myles Lefkovitz  
 gth836x; 901929700  
 mlefkovitz@gatech.edu**

**Part 1 – Local Random Search on a Neural Network**

1. **Dataset – Adult Income data:**

On the University of California Irvine’ Machine Learning Repository I found an Adult Income Data Set[[1]](#endnote-1) that classifies adults into one of two income categories: ‘>50K’, or ‘<=50K’. The ‘>50K’ category identifies individuals that earned more than $50,000 in the given year, 1994. The ‘<=50K’ category identifies individuals that earned less than or equal to $50,000. $50,000 in 1994 is approximately $81,000 in today’s terms. The data has 13 attributes, 5 of which are real-valued, and 8 of which are categorical.

The 5 real-valued attributes and their minimum and maximum values are: age [17 – 90], education-num [1 – 16], capital-gain [0 – 99,999], capital-loss [0 – 4,356], hours-per-week [1 – 99].

The 8 categorical attributes and a brief description are: workclass (type of employer), education (level of education, duplicated in the education-num real-valued attribute), marital-status, occupation (type of job), relationship (family status), race, sex, native-country. See figures 1-8 below for an examination of these 8 attributes.

The data set’s 2 categories, ‘>50K’ and ‘<=50K’, represent approximately 24% and 76% of the instances in the data set, respectively. To create a high-performing simplistic model, we could uniformly classify every individual in the ‘<=50K’ category. We should consider a learner successful only if it categorizes adult incomes correctly more than 76% of the time.

1. **Backpropagation**

When completed backpropagation in assignment 1, cross validation settled on alpha=0.005 and hidden layer size = 3 as the best set of hyperparameters. See the model complexity chart in figure 1 below. There are a total of 47 input notes and one output node.

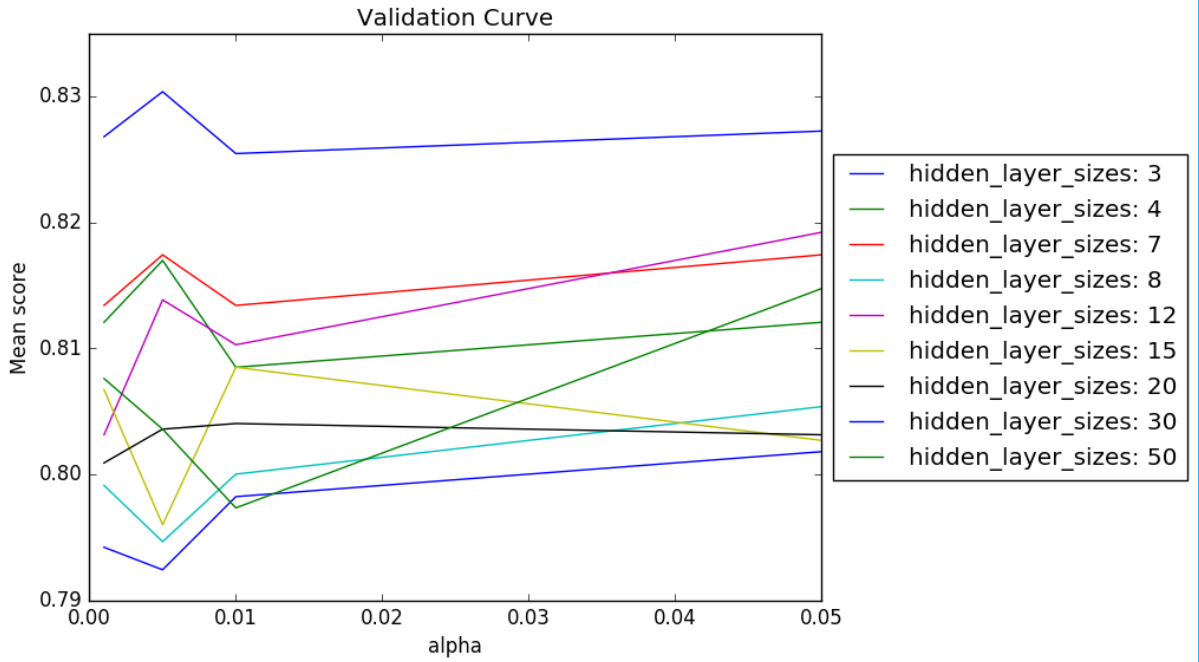


Figure 1. Neural Network Model Complexity Chart

This model’s performance against training data is 86.8%  
This model’s performance against CV data is 83%  
This model’s performance against test data is 83.5%  
This model took approximately 49 minutes to run

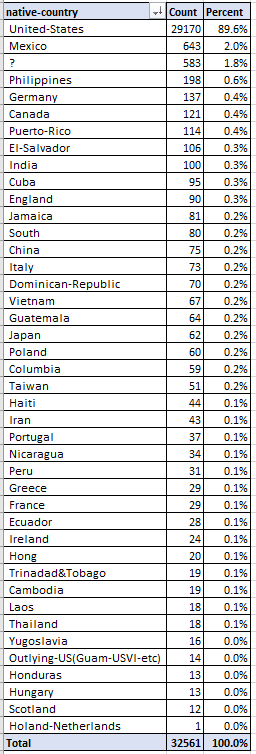
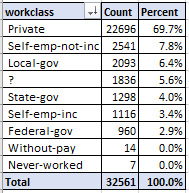
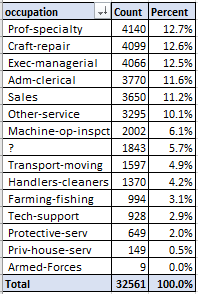
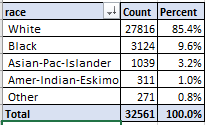
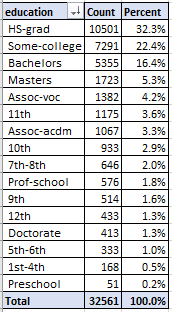
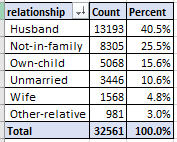
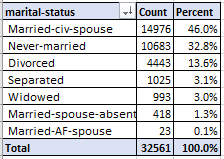
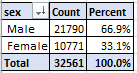


Figure 9. Adults by sex

Figure 8. Adults by marital status

Figure 7. Adults by relationship

Figure 6. Adults by education

Figure 5. Adults by race

Figure 3. Adults by work class

Figure 4. Adults by occupation

Figure 2. Adults by Native Country

1. **Randomized Optimization**

**Randomized Hill Climbing**

Randomized Hill Climbing is an algorithm that selects a random starting point and iteratively finds the best point in its neighborhood and move to it. Using this process the algorithm ‘climbs’ the ‘hill’ to the optimum.

A random hill climbing algorithm was used to train the weights of the adult income neural network described above. Below, you can find the accuracy of the algorithm at different numbers of iterations. At 20,000 iterations the algorithm correctly predicted 83% of the instances in the training set and 80.9% of the instances in the test set and took 834s (around 14 minutes).

|  |  |  |  |
| --- | --- | --- | --- |
| Iterations | Train Accuracy | Test Accuracy | Time (s) |
| 50 | 67.81% | 69.48% | 1.599 |
| 100 | 24.96% | 24.48% | 3.322 |
| 500 | 76.96% | 78.23% | 15.954 |
| 1000 | 77.01% | 78.23% | 32.823 |
| 2000 | 78.21% | 79.06% | 69.573 |
| 5000 | 78.80% | 79.90% | 155.691 |
| 10000 | 80.31% | 80.10% | 364.867 |
| 20000 | 83.08% | 80.94% | 834.851 |

**Simulated Annealing**

Simulated Annealing is an algorithm that selects a random starting point and iteratively selects a point at random in its neighborhood to potentially move to. The move is made if the point is closer to the optimum, or probabilistically dependent on the temperature otherwise.

A simulated annealing algorithm was used to train the weights of the adult income neural network described above. Two hyperparameters were tuned:

1. Initial Temperature (10^11, 10^8, 10^5)
2. Cooling rate (0.99, 0.95, 0.90)

Iterations were capped at 10,000 for simplicity. Some combinations of initial temperature and cooling rate were thrown out. In the end 4 were evaluated:

1. T = 10^11, Cooling rate = 0.95
2. T = 10^8, cooling rate = 0.95
3. T = 10^5, cooling rate = 0.99
4. T = 10^11, cooling rate = 0.90

For higher cooling rates, you’d expect to need more iterations before converging to the optimum (the temperature stays higher for more iterations). For lower cooling rates, you accept some risk that the algorithm will cool too quickly and miss the global optimum.

The evaluation output below shows that at 10,000 iterations, the algorithm with initial temperature 10^8 and cooling rate 0.95 performed just as well as the algorithm with initial temperature 10^11 and cooling rate 0.90. They each classified 79.2% of training instances and 79.69% of test instances correctly. The algorithms took 383 and 359 seconds to find the weights respectively (around 6.5 and 6 minutes).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Train Accuracy** | | | |
| **Iterations** | **10^11, 0.95** | **10^8, 0.95** | **10^5, 0.99** | **10^11, 0.90** |
| **50** | 75.1% | 55.7% | 75.0% | 68.7% |
| **100** | 74.2% | 73.5% | 25.0% | 76.8% |
| **500** | 25.4% | 55.4% | 23.2% | 76.8% |
| **1000** | 34.9% | 75.0% | 25.0% | 77.1% |
| **2000** | 75.0% | 77.7% | 78.8% | 76.8% |
| **5000** | 78.7% | 80.6% | 76.9% | 78.6% |
| **10000** | 78.2% | 79.2% | 76.8% | 79.2% |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Test Accuracy** | | | |
| **Iterations** | **10^11, 0.95** | **10^8, 0.95** | **10^5, 0.99** | **10^11, 0.90** |
| **50** | 75.42% | 57.40% | 75.52% | 68.75% |
| **100** | 73.96% | 73.85% | 24.48% | 78.23% |
| **500** | 25.00% | 53.44% | 21.77% | 77.92% |
| **1000** | 32.19% | 75.21% | 24.48% | 78.54% |
| **2000** | 75.52% | 78.44% | 79.69% | 78.23% |
| **5000** | 79.38% | 77.40% | 77.92% | 79.27% |
| **10000** | 79.38% | 79.69% | 77.92% | 79.69% |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Time (s)** | | | |
| **Iterations** | **10^11, 0.95** | **10^8, 0.95** | **10^5, 0.99** | **10^11, 0.90** |
| **50** | 1.639 | 1.603 | 1.665 | 1.625 |
| **100** | 3.253 | 3.377 | 3.576 | 3.236 |
| **500** | 15.5 | 15.971 | 16.207 | 16.587 |
| **1000** | 31.59 | 31.891 | 32.026 | 33.889 |
| **2000** | 72.631 | 67.173 | 66.496 | 64.847 |
| **5000** | 166.806 | 172.218 | 232.083 | 166.185 |
| **10000** | 376.698 | 383.365 | 399.799 | 359.033 |

**Genetic Algorithm**

Genetic Algorithms combine instances in populations to attempt to reach the global optimum.

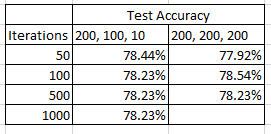
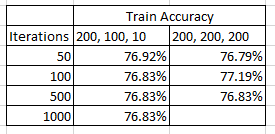
A genetic algorithm was used to train the weights of the adult income neural network described above. Two hyperparameters were tuned:

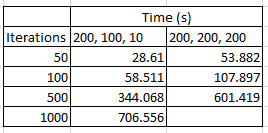
1. # of instances mated
2. # of instances mutated

With population size held constant at 200 (around 10% of the training set) two sets of hyperparameters were evlautes:

1. # of instances mated = 100; # instances mutated = 10
2. # of instances mated = 200; # instances mutated = 200

The evaluation output below shows that the parameters don’t seem to change much. Neither does adding additional iterations (other than processing time consumed). Genetic algorithms seem to perform very well with few iterations, but don’t seem to improve with additional computation time.

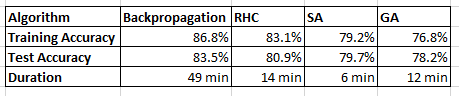




1. **Summary**

As shown above different algorithms have different performance rates when training neural networks. In this example, backpropagation was most closely approximated by using random hill climbing. The algorithm performs only marginally worse on the test and training set and took less than 1/3rd the training time. Simulated annealing and the genetic algorithm also performed better than chance on the test set (76% as shown in the introduction) and all have shorter training times than backpropagation.

In the additional graphs below, note the test accuracy convergence around 80% and the time different algorithms take with different iterations. Genetic algorithms seem to take much more time per iteration, which makes sense considering the multiple steps required to compare and match instances.



**Part 2 – 3 Problem Domains**

1. **Count Ones**

Count Ones is a well understood problem where the parameter space is a bit string (a string of 1s and 0s) and the optimal result is the string of all 1s.

We can use randomized optimization to solve this problem quickly and reliably. The evaluation function is simply the sum of the bits in the bit string (000011101 sums to 4). By starting with a random string, the optimization algorithms can search through the neighborhood of strings to find the single global maximum.

**Algorithm performance:**

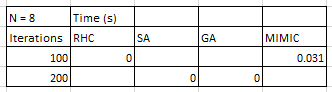
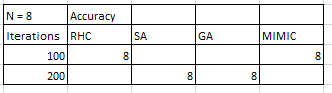
With enough iterations and a sufficiently small parameter space, any optimization algorithm should eventually be able to solve the count ones problem. Since the domain is unstructured, we would expect RHC and SA to perform better than GA and MIMIC.

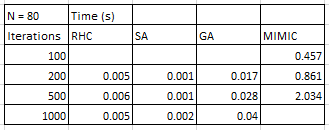
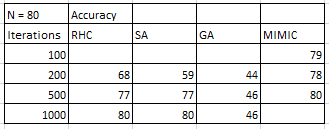
With an initial N (bit string size) = 8, we see that any of the algorithms (RHC, SA, GA, MIMIC) can solve this problem in under .1 second with 200 or fewer iterations.

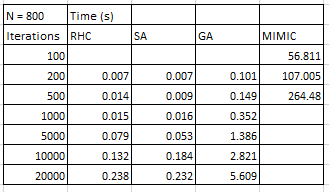
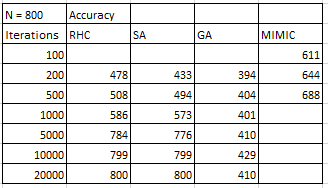
Paring N up to 80, we increase the complexity. We should expect GA and MIMIC to take more iterations to achieve the same results. It takes RHC and SA 1000 iterations to converge on the optimum. It only takes MIMIC 500 iterations to converge. GA does not come close to converging after 1000 iterations. It takes MIMIC >2 seconds of clock time to run 500 iterations, vs <0.006 seconds for RHC and SA.

Increasing the N to 800, we further increase complexity. We could expect GA and MIMIC to fail to converge in reasonable time, while RHC and SA should still succeed. Indeed, we see convergence in 20,000 iterations for both RHC and SA. Total clock time for either algorithm is under .25 seconds. MIMIC approaches 5 minutes of runtime at 500 iterations and is far from convergence at that point.

See detailed output below:







**Summary:**

We see that on simple versions of this problem, or with very limited iterations, MIMIC is able to more closely approximate the optimum, but as we increase the difficulty of the problem, we see that, as expected, SA reaches the optimum in less clock time (and possibly fewer iterations).

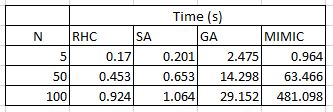
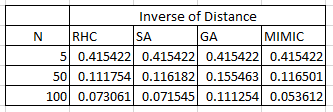
1. **The Traveling Salesman Problem**

The traveling salesman problem defines the optimization problem where an algorithm must traverse every node in a connected network while traveling the minimum distance in that network.

**Algorithm performance:**

With enough iterations and a sufficiently small parameter space, any optimization algorithm should eventually be able to solve the traveling salesman problem. Since the domain is structured, we would expect GA and MIMIC to perform better than SA and RHC.

As we can see in the output below, with only 5 nodes, any of our algorithms converge on the solution for the traveling salesman problem and do so relatively quickly (<1 second). For problems with many nodes, genetic algorithms stand out with the ability to find much better solutions without too much cost.



**Summary:**

We see that on simple versions of this problem any algorithm can converge on the optimum solution, but as we increase the complexity of the problem, we see that, as expected, GA is able to use the domain’s structure to find a better path than its competitors.

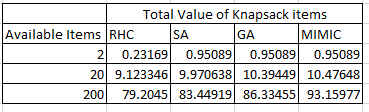
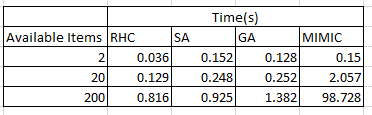
1. **Knapsack Problem**

The knapsack problem defines the optimization problem where an algorithm must add items to a knapsack such that the total value of included items is maximized and the weight constraint is not exceeded. An algorithm must identify which items to include and which to exclude.

**Algorithm performance:**

With enough iterations and a sufficiently small parameter space, any optimization algorithm should eventually be able to solve the knapsack problem. Since the domain is structured, we would GA and MIMIC to perform better than SA and RHC.

To compare the algorithms, I first selected a simple problem space: 2 available items, 1 copy of each, with a knapsack that holds 1 unit of weight and has 1 unit of volume. As you can see in the analysis below, SA, GA, and MIMIC all performed similarly and had the same total value with similar runtimes. The available items were increased to increase the complexity of the problem. It’s easy to see MIMIC succeeding with Genetic Algorithms not far behind.

**Summary:**

We see that on simple versions of this problem, or with very limited iterations, many algorithms are able to closely approximate the optimum, but as we increase the difficulty of the problem (even slightly) we see that, as expected, MIMIC reaches higher maxima. This supports the initial hypothesis that MIMIC and GA perform better on structured problem domains.

1. **Summary**

These three problem domains, Count Ones, Traveling Salesman, and Knapsack, can all be optimized using the randomized optimization techniques discussed in this course, but each algorithm performs differently on each problem domain. The simulated annealing algorithm performs best on unstructured problems like count ones. Genetic algorithms perform best on a structured problem like the traveling salesman problem. MIMIC is able to perform best on a complex structured problem like the knapsack problem.

1. http://archive.ics.uci.edu/ml/datasets/Adult [↑](#endnote-ref-1)