James Liu
CS 4641, Isbell, Spring 2014
March 16th 2014
Randomized Optimization Assignment

Randomized Optimization applied to Neural Networks

For this section, I used the same spambase dataset I used in the Supervised Learning assignment. Additionally, these tests are more for determining if randomized optimization methods perform as well as non-random methods can. Thus, as all of them are using the same underlying representation of a neural network of the a similar structure, conditions for overfitting and other phenomenon pertaining to the classifier are all the same, so none of the tests conducted in this section tested on a separate testing set or used cross validation, but rather are compared to two standards based on the training accuracy at a given iteration and total time required to train as a measurement of how well these randomized optimizations can develop models that explain the data.

All of the results shown in this analysis will be compared to two standards: Weka's ZeroR algorithm and the results from the neural network tests seen in my Supervised Learning assignment (the results have been included in the SupervisedNeuralNetworks excel file), particularly the L=0.3, M=0.2, 2500 epochs test on spambase (standard settings for a backpropogated neural network without overfitting). ZeroR simply chooses the most common class for any given instance, and can be seen as a baseline for performance. The second standard of a backpropogated neural network is used as basis of comparison for performance as well as time used, since both are using the same underlying model to train. It is expected that randomized optimization will have significantly better training times, avoid local minima better, and more easily deal with non-differentiable functions than backpropogation with momentum does.

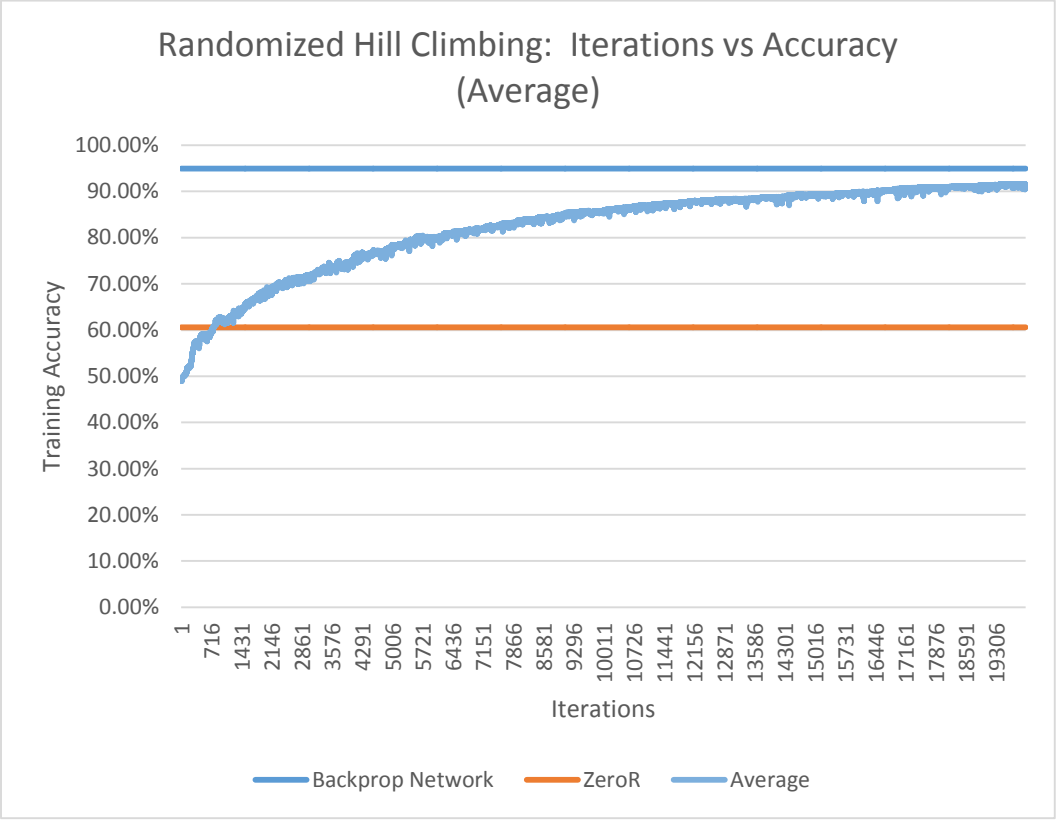
Algorithm	Training Accuracy	Training Time	Extra Notes
ZeroR	60.5955%	<1ms	
MultiLayerPerceptron	94.9529%	1833.45 seconds	L=0.3, M=0.2, at 10000 epochs, not cross validated

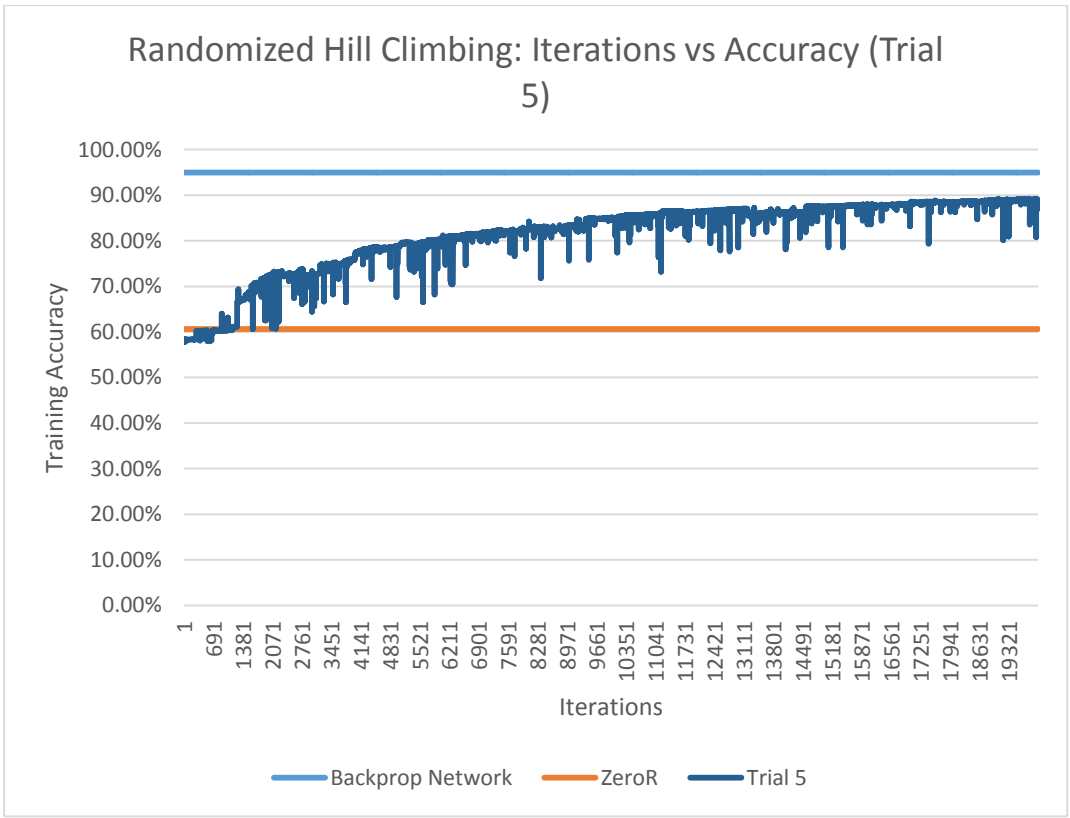
The neural networks used in these tests were set up with 30 hidden nodes, a number that I found to maximize the overall accuracy and the same number used in the backpropogation networks used in the first assignment. Too few and the network was not complex enough to describe the target concept, too many and the network became too complex and thus didn't converge fast enough, both of which reduced the final classifier's accuracy.

The tests I created tested the classifier created by the algorithms at each iteration. While neural networks are fairly fast at classifying instances, there are still 2000 or so instances in the dataset that were being tested each iteration, and over the course of the 1,000 or 20,000 iterations, this adds a significant amount of time to the training times of each algorithm. Thus, every algorithm in this section is adjusted to remove the extra testing from the training time, and thus may differ from the actual values seen when running these tests.

Finally, due to the randomized nature of these algorithms, I took the average case of several trials for each parameter set for each algorithm to make sure that the results recorded weren't purely coincidence. Randomized Hill Climbing was run a total 10 times, while Simulated Annealing and Genetic Algorithms ran a total of 3 trials per parameter configuration.

Randomized Hill Climbing



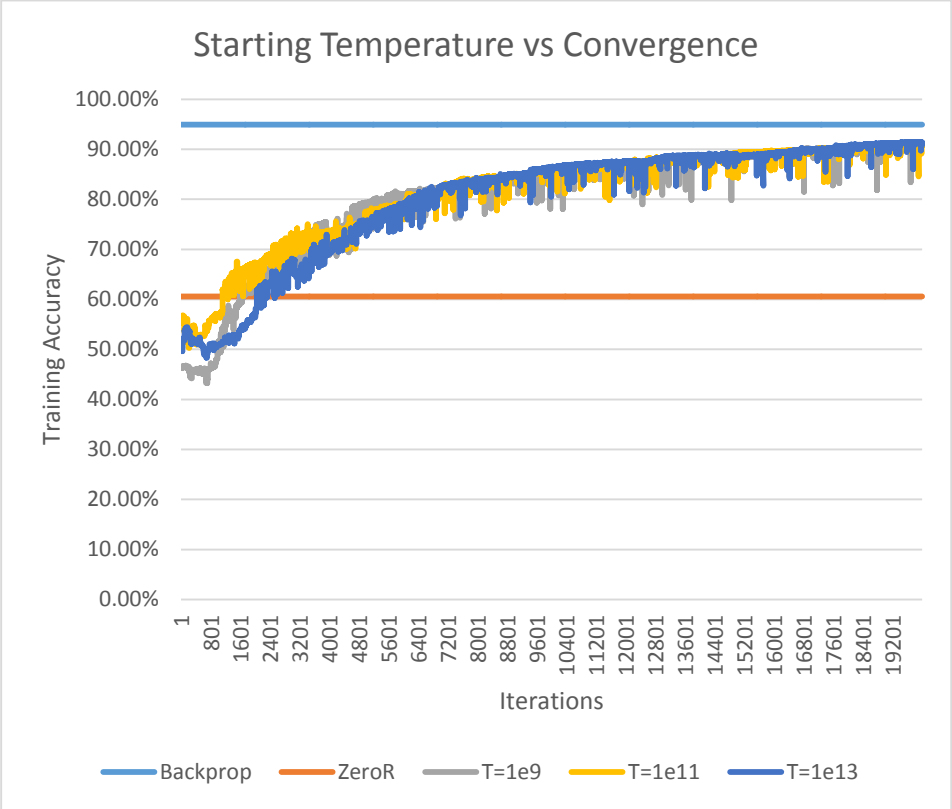


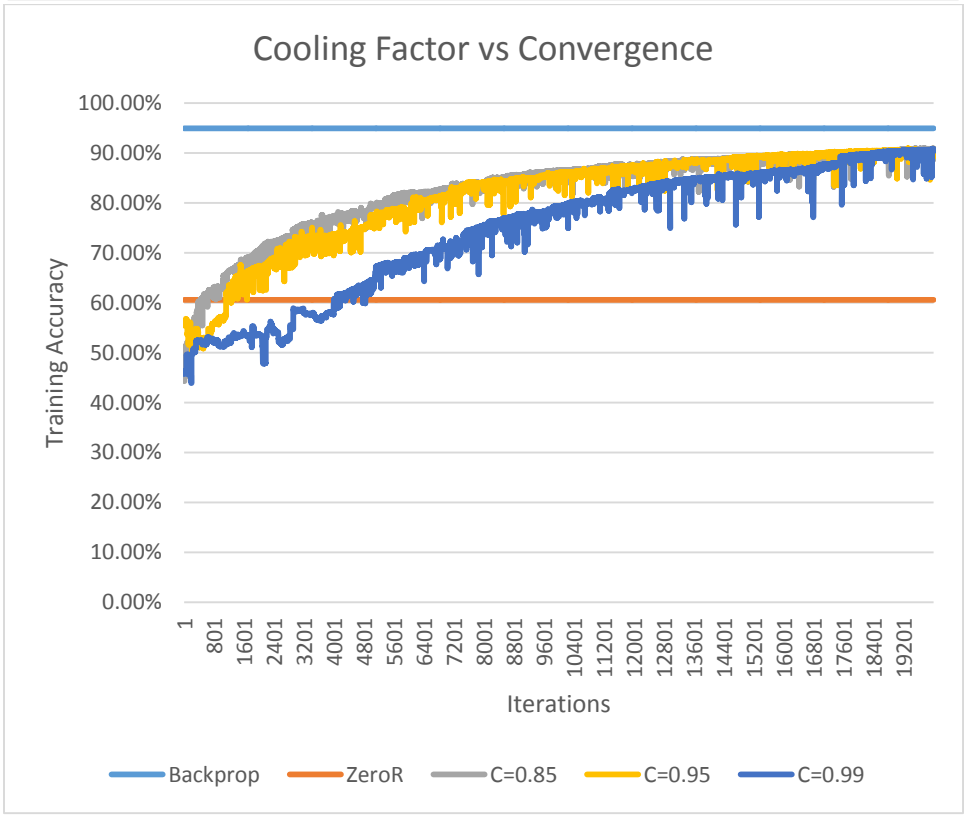
A total of 10 trials was performed with randomized hill climbing, each running to a full 20,000 iterations, with an average final training accuracy of 91.52%. As noted with the average case plot, the training accuracy of randomized hill climbing seems to monotonically increase with respect to the number of iterations performed, at least up to the $20{,}000^{th}$ iteration. This seems odd given the way neighbor function the neural network optimization problem is set up to choose a neighbor by changing only one of the weights by a maximum of 1. Given that the spambase dataset has a total of 58 attributes, many of which are sparse and/or irrelevant and that the underlying representation used is a neural network, the search space is enormous compared to the area in which neighbor function samples from. The fact that all ten trials tended to converge toward a single value suggests that the optimization problem presented by spambase has numerous local maxima that are of close proximity to each other that increase in value as they approach the absolute maximum. The other possible explanation is that there is only one local optima to the function, but the trials done with genetic algorithms show that that cannot be true.

It took on average 3316.46 seconds (adjusted) to complete 20,000 iterations. Thus, while randomized hill climbing generated a neural network that works almost as well as those created through backpropogation, it took three times the time to get close. Furthermore, the average accuracy vs iterations plot shows a logarithmic relationship between the two: further iterations would give diminishing returns in increase in accuracy, usually what one would see in a gradient descent algorithm. As randomized hill climbing can be viewed as a randomized version of gradient descent that is capable of dealing with non-smooth functions, the diminishing returns over time suggest that the underlying function being optimized is continuous and differentiable, as randomized hill climbing presents a similar but strictly worse performance than backpropogation does.

Finally, it should be noted that the graph plotting the accuracy of a single trial is not monotonically increasing like is expected of a greedy algorithm like RHC. There is the occasional spike in error, where it either suddenly increases or decreases before quickly recovering. As a sum of squared error was used as an evaluation function, the tests were making the assumption that by minimizing it by maximizing its inverse we could increase accuracy because the noise in the set was Gaussian. The algorithm is likely choosing a point with a lower sum of squared error, but the training accuracy could decrease because of noise that is only approximately Gaussian: it's close enough to a Gaussian distribution that minimizing sum of squared error will on average increase accuracy, but not close enough that it happens with every case/iteration.

Simulated Annealing



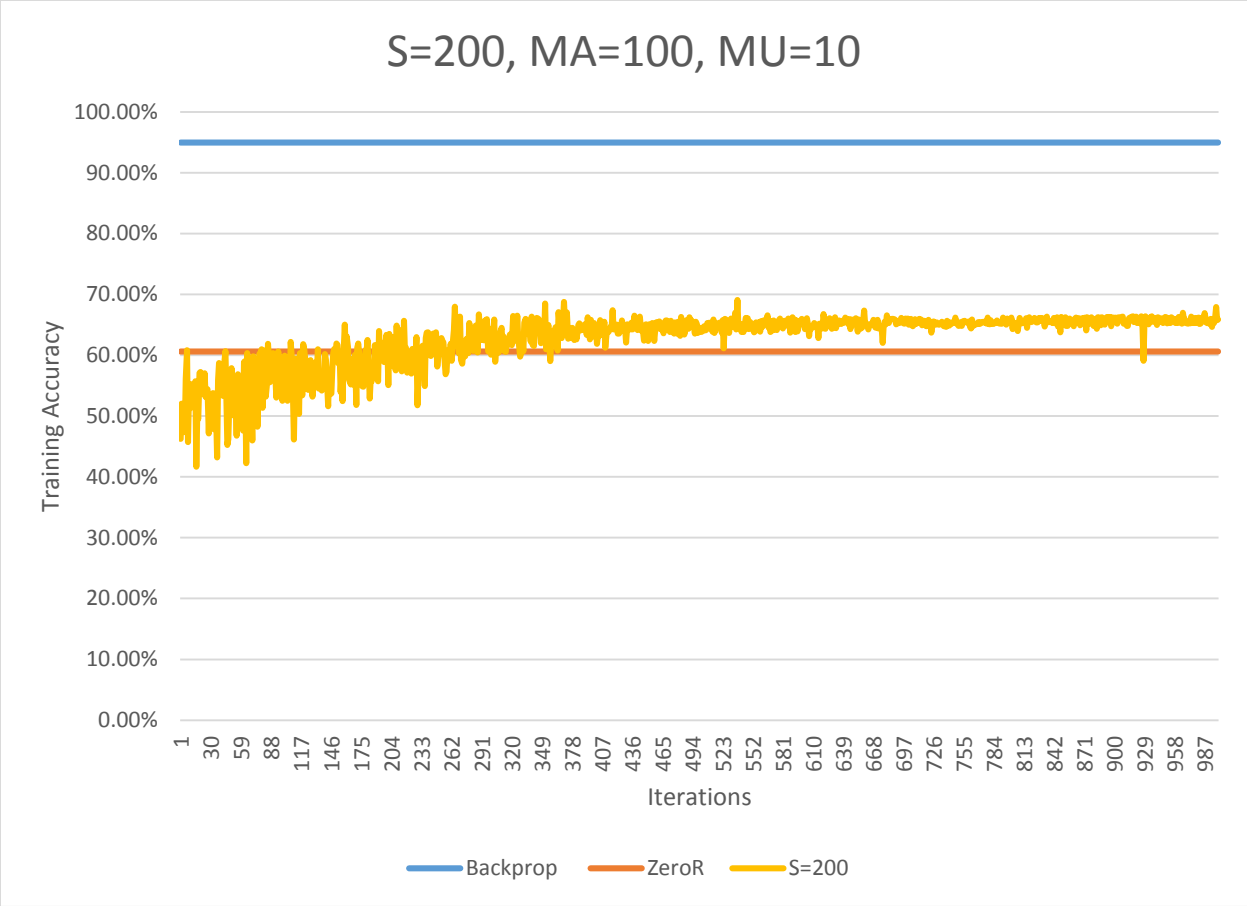


I took a total 15 trials, each 20,000 iterations, to test simulated annealing, 3 for each parameter configuration. Using T=1e11, C=0.95 as and T=a standard, I tested using altered starting temperatures (T={1e9, 1e11, 1e13}) and cooling factors (C={0.85, 0.95, 0.99}), to test the effects each have on how the algorithm behaves. Due to how similar the simulated annealing and randomized hill climbing are in execution, the two share many properties, such as occasional spikes in training accuracy from a non-Gaussian noise in the dataset, need for a high number of iterations to converge, etc. The main difference between the two is simulated annealing's capability of making a "bad decision", and thus allow it to escape local maxima. However, since randomized hill climbing already preforms well enough on spambase, most of the improvements made by simulated annealing do not improve, and sometimes hinder the overall performance. As for the time used, simulated annealing took at least 3000 seconds to complete all 20000 iterations, much like randomized hill climbing, which is to be expected, due to the similarity of the two algorithms.

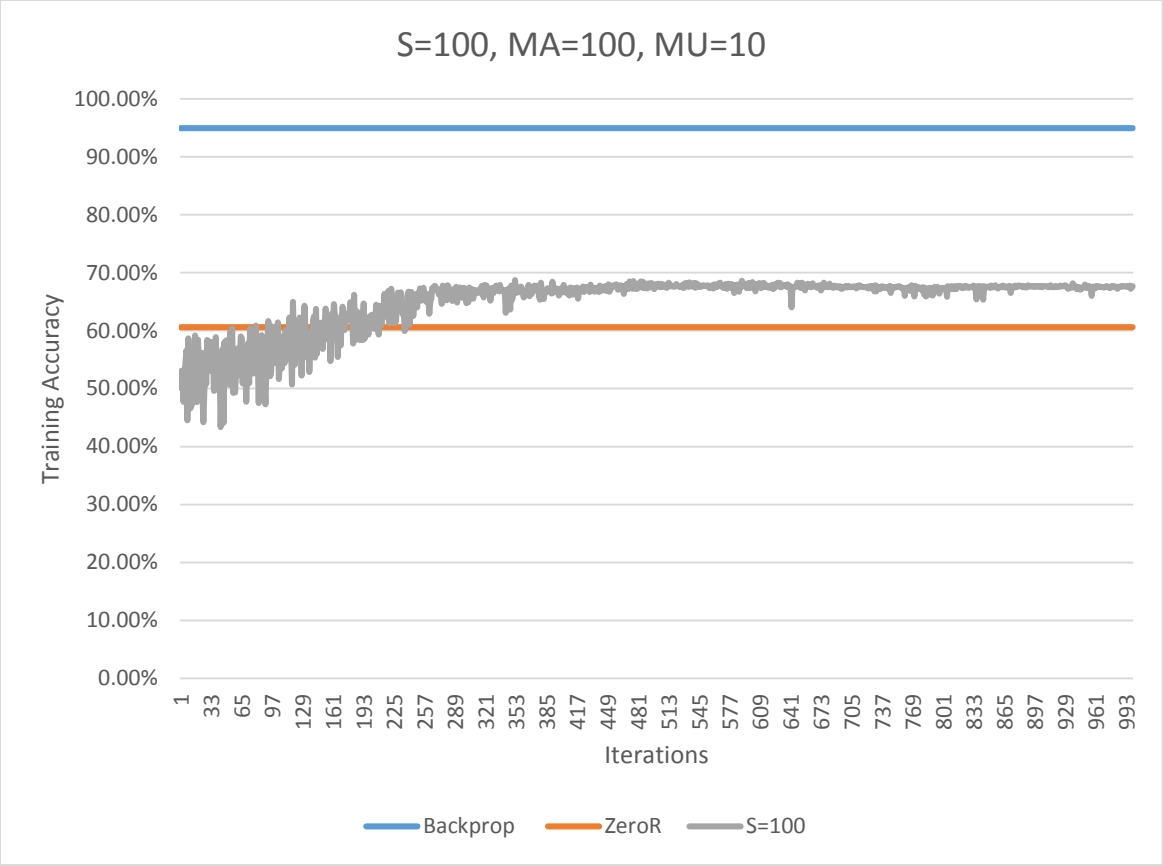
Overall, the addition of the temperature-based probability to take a jump to a worse solution will slow convergence to a single value, which is expected as there is no guarantee moving to such a neighbor will ultimately produce more optimal solutions. The current temperature directly affects the likelihood that the algorithm will jump to a point less optimal than the current one. With the starting temperature tests, it's easy to see a slight variance in the beginning due to differences in temperature. The T=1e9 tests takes to the the logarithmic increase much faster than the T=1e11 tests do. Similarly, the T=1e13 tests take much longer to start to stop making bad decisions and start to converge. It's also interesting that the time difference between each's tendency to converge starts varies linearly: the time between when the T=1e9 and T=1e11 tests is approximately the same as those between the T=1e11 tests and the T=1e13 tests, despite there being an exponential increase in starting temperature. This can be explained by the fact that the cooling factor, the rate at which temperature decreases over time, is multiplicative. It's a constant less than 1 that is multiplied into the current temperature after every iteration. As such, temperature exponentially decreases over time. As starting temperature increased exponentially, the difference in time to converge increased linearly.

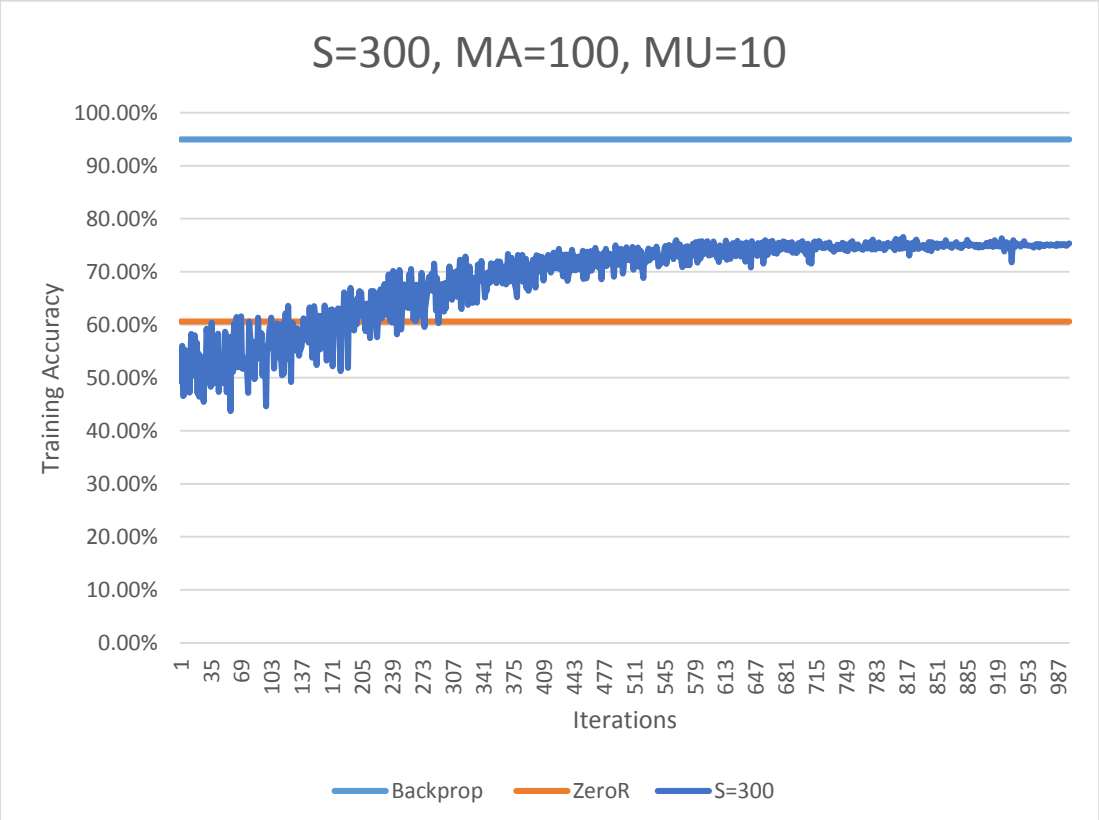
The cooling factor determines by how much is the temperature reduced each iteration, and could be seen as a measure of deviation from randomized hill climbing. A cooling factor closer to 1 will result in slower temperature decrease. The C=0.85 tests very quickly takes to the logarithmic increase seen with randomized hill climbing on spambase. The C=0.95 test has a slight delay, but also quickly assumes the logarithmic pattern, although not as tightly as the C=0.85 tests. The C=0.99 tests show the algorithm making bad decisions as far as the 11,000 iteration, whereas the C=0.95 and C=0.85 stopped well before that.

Genetic Algorithms



Just as I performed a set of three trials for each parameter set, I tested genetic algorithms in the same manner, each to 1,000 iterations of the algorithm. Using S=200 (population size), MA=100 (number to mate per iteration), and MU=10 (number to mutate per generation, plotted above, as a standard, I tested the algorithm by varying only one of the three parameter s to see what effects it would have on the overall performance of the algorithm.


S=100, MA=100, MU=10
100.00%
90.00%
80.00%
70.00%
60.00%
50.00%
40.00%
30.00%
20.00%
10.00%
0.00%
Training Accuracy
1
33
65
97
129
161
193
225
257
289
321
353
385
417
449
481
513
545
577
609
641
673
705
737
769
801
833
865
897
929
961
993
Iterations
Backprop
ZeroR
S=100


S=300, MA=100, MU=10
100.00%
90.00%
80.00%
70.00%
60.00%
50.00%
40.00%
30.00%
20.00%
10.00%
0.00%
Training Accuracy
1
35
69
103
137
171
205
239
273
307
341
375
409
443
477
511
545
579
613
647
681
715
749
783
817
851
885
919
953
987
Iterations
Backprop
ZeroR
S=300

The first nine tests were to test the effect of changing the population size on the algorithm. The results are rather interesting. Changing the population size changed where the algorithm converged to. This supports the idea that there are indeed multiple optima in the optimization problem of creating a neural network for spambase. Increasing population seems to decrease the speed at which genetic algorithms converge, as notably, the fluctuations in accuracy for the S=100 tests are rather small after iteration 250 compared to that of S=200 and S=300. An explanation for this is that a larger population size maintain a wider variety of genes than those of smaller populations. Seemingly suboptimal genes, ones that hinder the optimality of each solution, are bred out much faster in smaller populations than they are in larger populations. Often some combination of suboptimal alleles may perform better than a single optimal gene set. This would equivalent to having a larger "neighborhood" in randomized hill climbing or higher temperature in simulated annealing: slower convergence, but it's easier to get out of suboptimal local optima. This also explains how the S=300 tests performed better overall than the others, but seemingly took longer to find the better solution.

The values I chose to test mating rate and mutation rate with did not produce particularly interesting results, but are viewable in the Genetic Algorithms excel file included. The mating rate and mutation rate are the factors that allows genetic algorithms to deviate from the initial population. Changing the mutation rate is akin to changing the momentum on a backpropogated neural network. Increasing it will increase the time it takes to converge, but increase the chances of finding better optima, but if it is too high, it will overshoot, if it is too low, it will not escape suboptimal local optima. Likewise, changing the mating rate is akin to changing the learning rate. The algorithm will check more varied populations as a result, but may overshoot if set too high. However unlike backpropogation, the process is randomized and is not limited to differentiable functions.

Overall, each trial took at least 10,000 seconds to complete all 1,000 iteration. It's a fair step up from the amount of time used for RHC or SA. It is a significant increase in amount of computation need per iteration, each instance in the population needs to be tested for fitness, mated, mutated, etc., compared to producing a random neighbor. Genetic algorithms generally do not work well with problems that require computationally expensive evaluation functions, as it has to evaluate the entire population at each iteration. It also did relatively poor compared to the other two on spambase. One explanation is that spambase's number of attributes created too large of a search space for the algorithm to work through. Even though the individuals were relatively spread apart, the unfit genes were quickly removed, and converged to a local maximum closest to the best initial starting point.

Optimization Problems

Simulated Annealing Advantages: FlipFlop

Flip Flop is a function that returns the number of times bits alternate in a bitstring, including the first bit (nothing to anything is a flip). "0011" would return 2. "0101" would return 4. The optimal solution produces a constantly alternating bitstring with a return value equal to the length of the string. The function is also known to have a large number of local maximums.With each bit as a separate attribute in a set of instances. I tested all four algorithms using the following parameters and a FlipFlop problem of bitstrings length 180:

Algorithm	Parameters	Average Result (of twenty trials)	Total Time (seconds)
Randomized Hill Climbing	20,000 iterations	146.3	0.425791146
Simulated Annealing	20,000 iterations, T=100, C=0.95	177.7	0.48696624
Genetic Algorithm	1,000 iterations, S=200, MA=100, MU=20	147.55	0.284263
MIMIC	1,000 iterations, S=200, K=100	153.95	25.87919232

Of the four, only simulated annealing got close to producing the optimal solution. There is an easy explanation as to why this occurs. Simulated annealing works best in a small discrete space where neighbors and local optima are close by. In this case, each neighbor for simulated annealing was a flip of one bit in the bitstring. These conditions also make randomized hill climbing work well too. However, randomized hill climbing is unwilling to make a "bad decision". It will often get to a bitstring where changing any of the bits would result in no gain, and get stuck. Simulated annealing, given the proper temperature (set to 100 in this case, as the search space is fairly small), can easily get out of these plateaus, whereas RHC, genetic algorithms tend to converge to and stay stuck there. MIMIC on the other hand required significantly more time, and potentially more to successfully converge toward the absolute maximum. Runtime wise, simulated annealing is very fast for its level of performance compared to the other three algorithms.

Genetic Algorithms Advantages: TravelingSalesman

To test the advantages of genetic algorithms, I used the classic Traveling Salesman problem, where one needs to find the shortest path to visit all of the nodes in a graph with N vertexes. The problem I wrote for this generates a random graph with a specified number of vertexes and random valued edges and tries to apply the aforementioned algorithms to solve it by maximizing the inverse of the total length of the path. I ran the following algorithms as comparisons with the following parameters on a graph of size 50:

Algorithm	Parameters	Average Result (of twenty trials)	Total Time (seconds)
Randomized Hill Climbing	20,000 iterations	0.122163969	0.151215803
Simulated Annealing	20,000 iterations, T=1e12, C=0.95	0.126667605	0.303176502
Genetic Algorithm	1,000 iterations, S=200, MA=150, MU=10	0.170790477	0.41422137
MIMIC	1,000 iterations, S=200, K=100	0.101461259	24.86249917

Of the four of them, genetic algorithms did the best. This is easily explained by the fact that genetic algorithms works with entire sets of potential solutions, and only continues with ones that are slightly altered or mixed versions of those that did the best out of the previous generation, whereas RHC, SA deal with only one possible solution at a time, and with an high branching factor that is associated with this problem, the search space becomes too large for them to properly go through without additional iterations. MIMIC on the other hand looks at the probability that the true maximum is in an area. As the search space for this function is very large and has many local maximums, it would require more than the allotted 1,000 iterations to approximate the absolute maximum more accurately, resulting in poor results. Another thing to note is the time required to execute the entire optimization problem. Genetic algorithms took marginally more time than randomized hill climbing and simulated annealing to complete. Though it has been mentioned before that they are generally very time consuming. That is only partially true, genetic algorithms run time depends on the number of iterations done, population size, but most importantly the runtime complexity of the evaluation function. Evaluating a neural network is far more intensive than summing the

lengths of a set of edges in a graph, which is why GAs only required an almost negligible amount of increased time for this problem as compared to with the neural network optimization problems.

MIMIC Advantages: Four Peaks

Four peaks is a function that that takes a bitstring of length N and a "trigger" position T. It maximizes at bitstrings that have contiguous 0s or 1s up to the trigger point, and then the complementary bit contiguously until the end of the string. It has a total of four local maxima, two of which where the bitstring is entirely 0s or entirely 1s, which produces a value of N. The absolute maximum is found at the aforementioned string with a value of 2N – T – 1. I ran the aforementioned algorithms with the following parameters on a FourPeaks problem of N=85, T=8

Algorithm	Parameters	Percent Suboptimal (out of 20) (value of 85)	Percent Optimal (out of 20) (value of 161)	Total Time (seconds)
Randomized Hill Climbing	20,000 iterations	100%	0%	0.130296445
Simulated Annealing	20,000 iterations, T=1e12, C=0.95	90%	10%	0.208812035
Genetic Algorithm	1,000 iterations, S=200, MA=100, MU=20	0%	0%	0.102043757
MIMIC	1,000 iterations, S=200, K=5	30%	70%	3.562616823

With the exception of genetic algorithms, all of the algorithms used generated either the suboptimal or optimal solutions every trial. None of them did as well as MIMIC did. As there are only four local maxima, each spread a fair distance apart (in fact each is N changes from each other, far beyond the scope of the neighbor function given to RHC and SA), RHC and SA failed to produce successful results. Genetic algorithms failed to find any of the local maxima, and may have needed additional iterations to properly converge, as the results had no coherent pattern to them, and looked like they were far from converging anytime soon. This is likely caused by the mutation factor of 20, which is far too high for this kind of problem. MIMIC on the other hand, performed far better than any of the others, primarily because it creates a distribution of the most likely locations for local maxima. By repeating this process by sampling from the previous iteration's distribution, it is increasingly likely to find better maximums. Over the iterations, suboptimal local maximums will be removed as better ones are found, further narrowing the search, until it converges on a single value, the absolute maximum. MIMIC is effective in the sense that it requires the least number of iterations to produce a good result out of the four algorithms we've looked at, as the result from each iteration increases the chances that the next iteration will produce better results, which compounds. However, it does not work well with discontinuous functions because high values in these functions may not lead to better maximums. Functions with a large number of local maximums of the same height like FlipFlop, as it would result in a near uniform distribution, which does not improve upon previous iterations. The number of instances to keep to generate the next distribution also determines how fast MIMIC converges: too high and the distribution does not improve, too low and the distribution is skewed toward a small subset of local maximums encountered. Finally, the one other downside of MIMIC is its runtime, as seen with previous tests, MIMIC takes a significantly longer time, ten to possibly a hundred times longer than RHC, SA, and even GAs, due to the added complexity of finding the best K samples out of a N instance group and building a distribution around them.