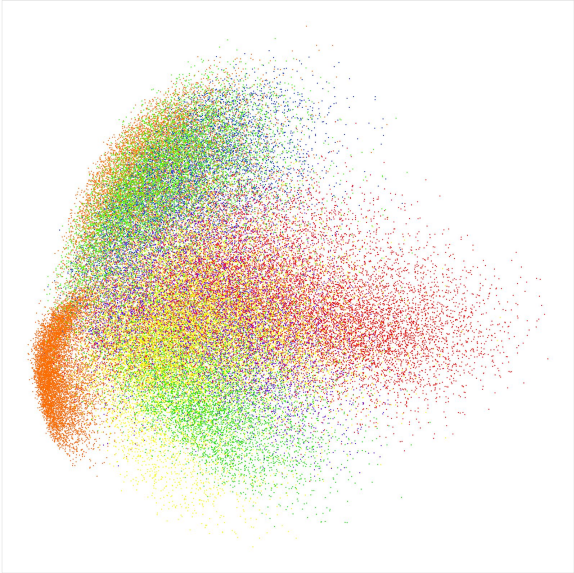
Ashwini Iyer
CS 4641
ML Project 2
March 9, 2018

Randomized Learning Report

Datasets & Justification

I chose the MNIST data set as my classification data set. The MNIST dataset contains images of handwritten digits with approximately 50,000 training images and 10,000 testing images, where each sample is a 28 x 28 image with no color channels. This dataset is well benchmarked and was used previously in the supervised learning assignment. To avoid handling high-dimensional data and issues discovered previously in the supervised learning assignment, PCA was performed on the dataset, and a smaller training set was sampled from the dataset with reduced dimensionality. Different dimensionality reductions were tried, and below is a visualization of PCA performed down to 2 dimensions,

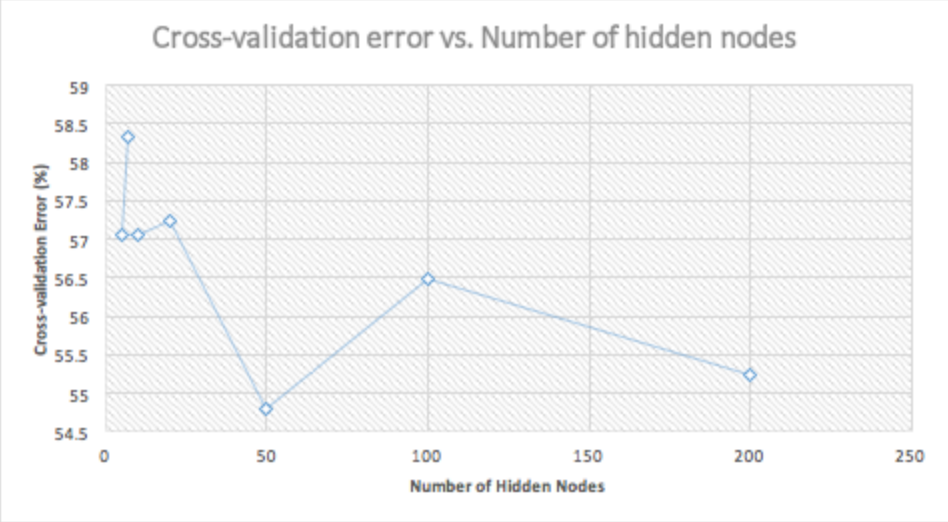


where the horizontal and vertical axes represent the two transformed dimensions, and the colors correspond to the 10 labels of digits. The transformed data is cleaner and results in higher accuracies for smaller amounts of data as the variance is explained in the reduced dimensions succinctly.

Randomized Optimization on Neural Networks

For this section I will be using Randomized Hill Climbing, Simulated Annealing, Genetic Algorithms and MIMIC instead of Back Propagation on a neural network for the MNIST classification problem I addressed in the last experiment.

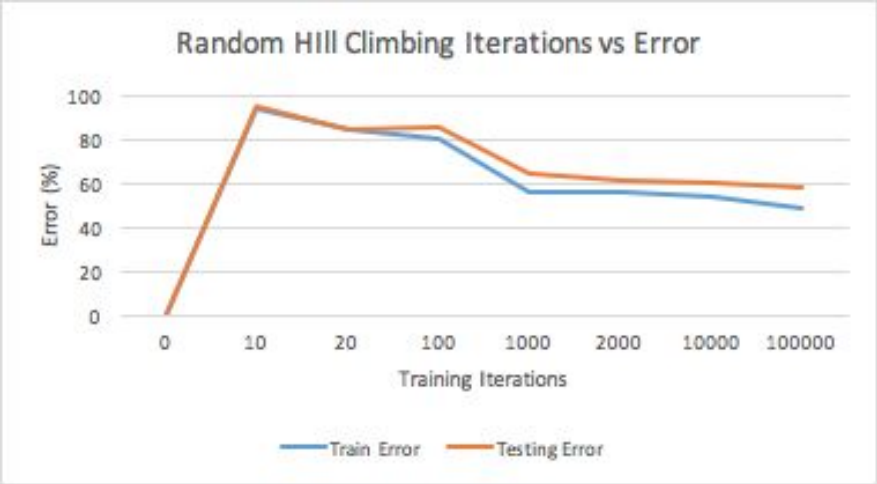
To best test the optimization methods of randomized hill climbing, simulated annealing, and genetic algorithms on the MNIST data, I needed to determine the best hyperparameters for my network as well as the parameters within each of the optimization problems. The first hyperparameter search I performed was for the number of hidden nodes.



The more hidden nodes in the neural network, the longer training takes and the greater possibility the model can overfit the data. However, with too little hidden nodes, the model is not complex enough to model the data and can underfit. This classic bias vs. variance tradeoff results in a sweet spot here for 50 hidden nodes with the lowest cross validation error. Cross-validation was performed with 10 folds with the training data being 70% of the dataset, and the testing data was the other 30%. The neural networks used to compare the different optimization networks had 30 hidden nodes, as this hyperparameter resulted in a similarly low validation error and faster training time.

Randomized Hill Climbing

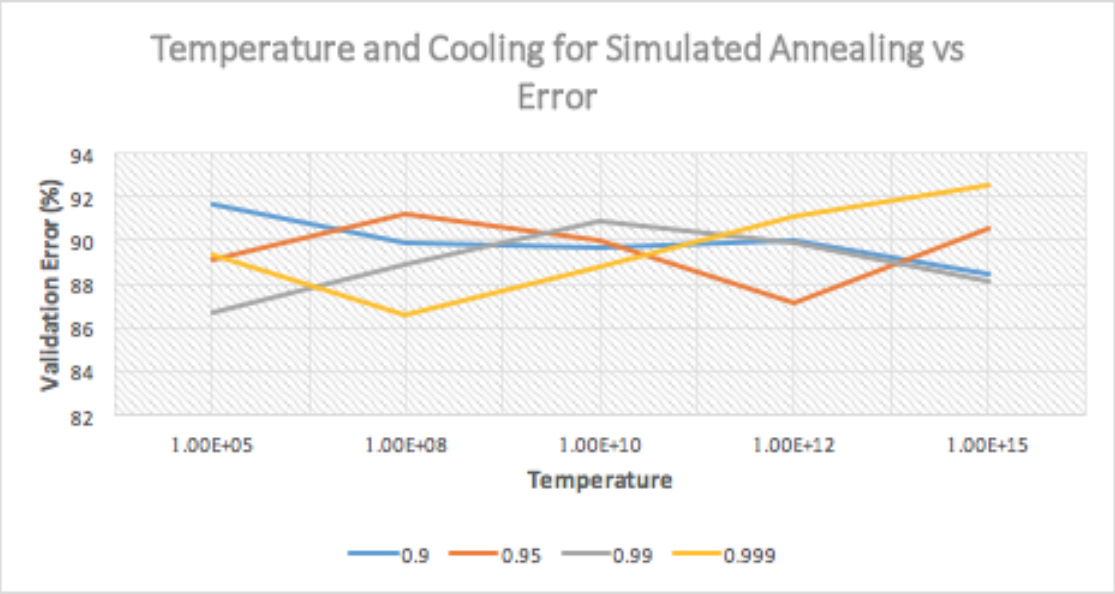
For Randomized hill climbing, there were no algorithm specific hyperparameters, so the following graph is the results of error vs. training iterations for both training and testing errors. A total of 100,000 iterations were run with an average training error of 48.8% and an average testing error of 58%. It took a total of 247 seconds to run.



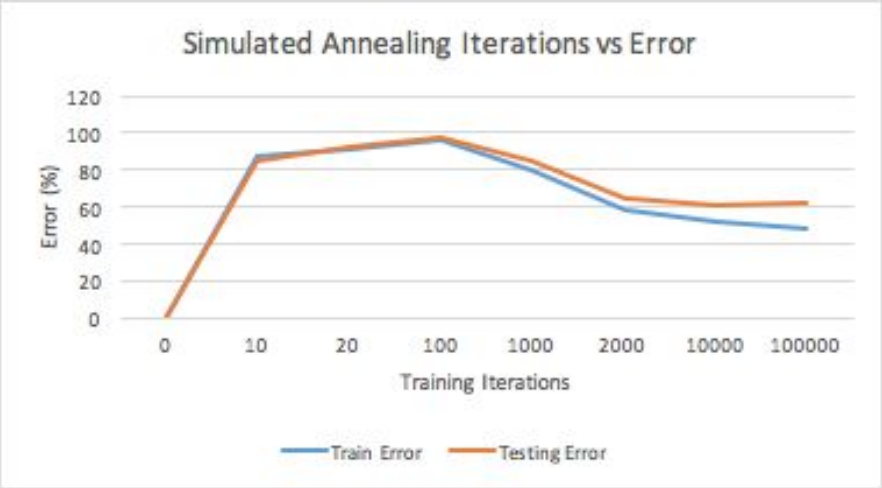
The training and testing error both decrease as the number of iterations increases, and the testing error stops decreasing around 10,000 iterations. The testing error remains higher than the training error throughout the experiment. Overfitting has been reduced a lot due to cross validation, and there is only a slight difference between training and testing error towards the end. This performance on the MNIST dataset for a simple neural network on a small amount of dimensionality reduced dataset is decent for a hill climbing approach. To decrease the chances of hitting a local maxima and stalling, random restart for Randomized Hill Climbing was checked, but it was discovered that the Randomized Hill Climbing from ABIGAIL performed more consistently. Given that the neural network optimization problem involves choosing a neighbor by changing one of the weights by at most 1, it is interesting to me that, up until 10,000 iterations, as the number of iterations increases the training accuracy increases as well. In addition, the hill climbing converged to around the same error when multiple trials of training were performed suggesting that the transformed MNIST dataset may have a cluster of multiple local maxima that increase in value as they near the absolute maxima.

Simulated Annealing

Simulated Annealing is a method for finding a solution to an optimization problem, and more specifically works well when the problem requires identifying a maximum or minimum. This is because Simulated Annealing strength is its ability to not get caught in a local maxima. In general sense it identifies a solution by generating a random initial solution by looking at the area nearby. If the nearby solution is better it moves to it. Simulated Annealing has hyperparameters of temperature and cooling which can affect movement based on iterations (cools over time). I performed a hyperparameter search to best identify the simulated annealing model using cross validation. Below is a graph of the temperature and cooling vs. the validation error.



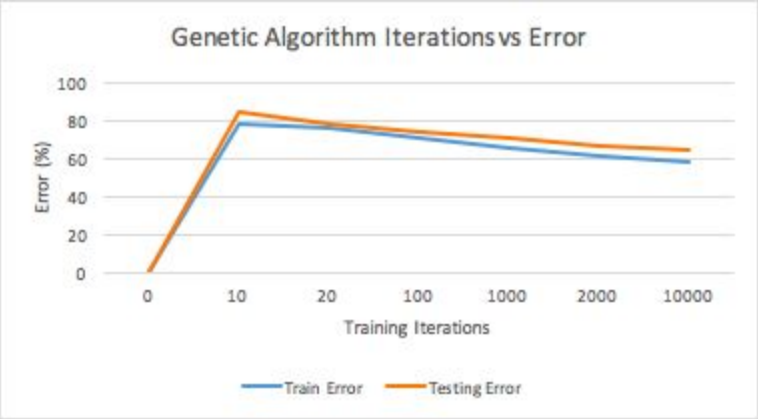
This graph represents the relationship between temperature and the validation error. The colors each represent the different cooling parameters and the results are the gathered validation error after 100 iterations. The probability to make a jump out of current local maxima is based on temperature and the cooling factor reduces this temperature over time, so the algorithm converges to a single maxima. This is performed with the intuition that jumps may not provide optimal solutions and may extricate the model to worse local maxima. It is logical then that a more aggressive cooling parameter of 0.9 performs better with higher temperatures like 1e15 than lower temperatures like 1e5 as initially the model initially jumps out of bad decisions quite rapidly and also cools its jumping to allow for convergence. The below graph represents the relationship between the training and testing errors over the iterations. A total of 100,000 iterations were run with an average training error of 48.7% and an average testing error of 62.3%. The total time to execute was 250 seconds, which is approximately around the same order of magnitude as random hill climbing.



Simulated Annealing and Randomized Hill Climbing are very similar in terms of how they execute and they both need a high number of iterations to converge. The main benefit of using Simulated Annealing over Randomized Hill Climbing is the fact that Simulated Annealing can avoid local maximas because it is allowed to reach suboptimal solutions. In other words, it can make “bad decisions”. I later compare these algorithms directly.

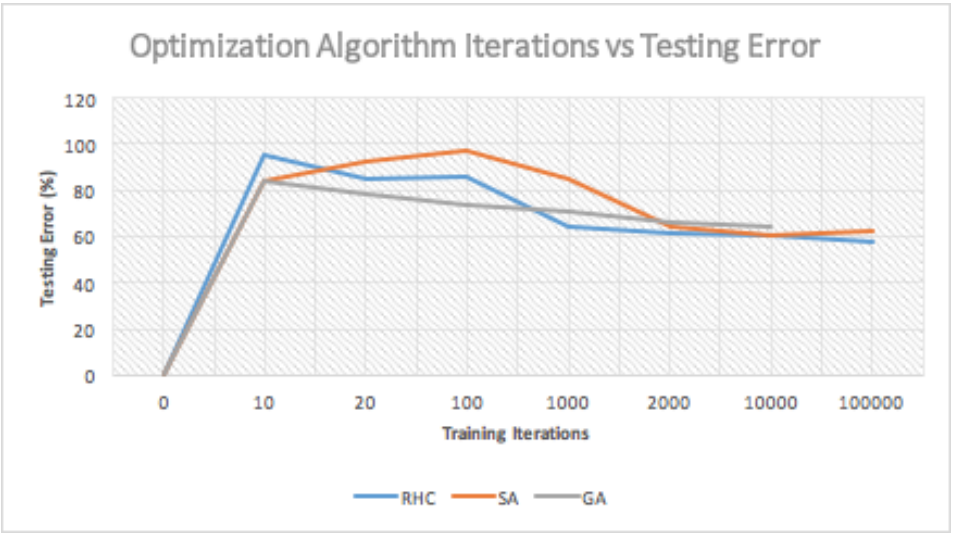
Genetic Algorithms

A Genetic Algorithm is a search heuristic that is based on the concept of natural selection. This is because it starts with an initial population and determines which to continue with based on a fitness function. In our specific case, a similar hyperparameter search was performed for genetic algorithms across population ratios ranging from 0.10 to 0.25, mate ratios, and mutate ratios. As 3 parameters are harder to visualize, the reported optimal hyperparameters were a population ratio of 0.15, mate ratio of 0.04, and mutate ratio of 0.04. 0.15 was a smaller population size which would increase the speed at which the algorithm converges as a smaller population size would have less variety of genes. Suboptimal genes would be pruned through breeding. An increased mutation ratio would increase convergence time but also increased the likelihood of converging to better optima. However, a very large mutation ratio can also overshoot maxima. The graph below represents the relationship between the Genetic Algorithm iterations and both the testing and training error.



A total of 10,000 iterations were run with an average training error of 58.4% and an average testing error of 64%. For 10k iterations, Randomized Hill Climbing, Simulated Annealing, and Genetic Algorithms took 24.9s, 24.7s, and 697.8s respectively. Genetic algorithms took significantly more time than Randomized Hill Climbing or Simulated Annealing, which were one to two orders of magnitude faster. But, this makes sense because in this algorithm for each instance the population needs to be tested for fitness, compared to producing a random neighbor, mated and mutated. It takes so long because it has to evaluate the whole population at each instance, as a result it would not be well suited for problems with computationally expensive evaluation functions.

Comparison of Algorithms



The graph above compares the testing error of the three optimization approaches across different training iterations. In terms of wall clock time, the training times for random hill climbing and simulated annealing were 30x faster than the training time using genetic algorithms. This is partially why, I did not perform a 100k iteration experiment for genetic algorithm. For time

efficiency, random hill climbing and simulated annealing approach a lower testing error than genetic algorithms and even beat out genetic algorithms after 10,000 iterations have been performed. To improve each algorithm as mentioned previously, for each algorithm, a hyperparameter search was performed to find the best parameters with cross-validation. The data was transformed by PCA to find dimensions with maximum variance and reduce dimensionality of the data to improve training time and reduce search space. Among the algorithms, random hill climbing and simulated annealing performed the best with testing errors of 60.6% and 60.6% respectively. The best performance is determined by the best testing error on unseen data at training iterations that indicate model convergence (around 10k iterations). Random hill climbing and simulated annealing were orders of magnitude faster in clock time but are less iteration efficient (arguably sample efficient). Random hill climbing performs better than simulated annealing at early iterations, but they both converge around the same testing error.

Optimization Problems

Genetic Algorithms Advantages

I chose the Traveling Salesman Problem to demonstrate the advantages of Genetic Algorithms. A genetic algorithm is a search heuristic that reflects the process of natural selection where only the fittest members are chosen for reproduction. I chose to test the advantages of Genetic Algorithms with the Traveling Salesman Problem. The Traveling Salesman Problem addresses the question of having a graph with N vertices and needing to find the shortest path to visit all the nodes exactly once. It is not difficult to find an initial solution that visits every node, but it will not be close to the optimal. I chose to use this to demonstrate the advantages of Genetic Algorithms because they start with a solution and make improvements to it, which is necessary to go from a solution that works to the optimal one. To implement the Traveling Salesman Problem I used a program that makes a graph with a specified number of vertices and edges. It then applies Random Hill Climbing, Simulated Annealing, Genetic Algorithms and MIMIC to the graph. I performed random hill climbing for 200k iterations. Simulated annealing was performed with a temperature of 1e12 and cooling of 0.95 for 200k iterations. Genetic algorithms and MIMIC were both performed for 1000 iterations. The table on the next page shows the comparisons between them in terms of time and score.

Table 1: Traveling Salesman (Input Size: 100 Iterations: 20)

	Score Average	Total Time (s)
RHC	0.073953096	1.899016377
SA	0.074289286	5.869660786
GA	0.090184154	14.18606192
MIMIC	0.045077403	4305.869219

Table 2: Traveling Salesman (Input Size: 50 Iterations: 20)

	Score Average	Total Time (s)
RHC	0.119831785	1.299920994
SA	0.120121525	4.77602994
GA	0.166999792	5.992023884
MIMIC	0.090395999	318.4787539

Genetic Algorithms perform the best for this problem. Random Hill Climbing and Simulated Annealing do not perform as strongly because, unlike Genetic Algorithms, they only consider one possible solution at a time and due to the fact that the Traveling Salesman Problem has such a high branching factor they have to go through multiple iterations to handle the search space. MIMIC will also struggle with the search space because it looks for the probability the true maximum is in the area and would need many iterations to handle the sample space and find an accurate true maximum. This is shown by the fact that MIMIC takes significantly longer than the other algorithms to execute. On the other hand, Genetic Algorithms do not have this problem because they work with the entire set of possible solutions.

MIMIC Advantages

I chose Four Peaks to demonstrate the advantages of MIMIC. Four Peaks is a function that takes a bitstring input of size N and a trigger point T as a parameters. It then maximizes at bitstring with continuous zeros or ones up until the trigger point and then complementarity up until the end. It has four local maxima, and the absolute maxima is found in the string with 2N - T - 1 as its value. I ran the Random Hill Climbing, Simulated Annealing, Genetic Algorithms and MIMIC on the Four Peaks problem with the following parameters: N = 50, T = 5. I performed 20 trials and noted the average performance.

Table 1: Four Peaks (Input Size: 50 T:5 Repeat: 20)

	Score Average	Total Time (s)	Suboptimal (score of 50)
RHC	52.2	0.908589529	95%
SA	83	2.800239903	25%
GA	78.6	1.16250268	0%
MIMIC	91.8	25.56851403	5%

Genetic Algorithms do not find any local maxima and fail to converge. All the others resulted in varying amounts of suboptimal solutions, and MIMIC had the highest score average and percent of trials that resulted in optimal solutions (a score of 94). MIMIC did the best and Randomized Hill climbing performed the worst with the lowest score average of 52.5 and highest suboptimal trial performance of 95% I believe that Randomized Hill Climbing and Simulated Annealing performed poorly because as explained previously, in this problem there are only four local maxima each spread N away from the other. This distance of N is beyond the neighbor function given to Randomized Hill Climbing and Simulated Annealing, so as a result they were unable to produce valuable results. MIMIC performed the best, with only 5% suboptimal and an average score of 91.8, because by nature of the algorithm it creates a probability distribution of the locations of the local maxima requires the least number of iterations to produce valuable results. Additionally each iteration value compounds because in later iterations always samples from the previous iteration distribution. Thus as we iterate the suboptimal will be removed and eventually allowing the algorithm to converge, allowing for fewer total iterations. One thing to consider in general when using MIMIC is its total run time. As we can see in all the executions of MIMIC it takes significantly more time than the others to run. It is not necessarily a flaw when the problem is this well suited to MIMIC, but in other cases it is definitely something to consider.

Simulated Annealing Advantages

I chose Flip Flop to demonstrate the advantages of Simulated Annealing. Flip Flop is a function that returns the number of times bits alternate in a bitstring. The optimal solution for flip flop is an alternating bitstring of the same length as the original string. Flip Flop is a function known to have a lot of local maximums since each bit is treated as a separate attribute. I tested Random Hill Climbing, Simulated Annealing, Genetic Algorithms and MIMIC on bit strings of length 100, 150 and 180.

Table 1: Flip Flop (Length: 100 Iterations: 20)

	Score Average	Total Time (s)
RHC	79.25	1.417082228
SA	98.5	3.059669507
GA	87.3	1.16250268
MIMIC	86.65	149.1328341

Table 2: Flip Flop (Length: 150 Iterations: 20)

	Score Average	Total Time (s)
RHC	122.15	1.872929275
SA	148	3.509898431
GA	124.05	1.975540217
MIMIC	128.05	290.7946916

Table 3: Flip Flop (Length: 180 Iterations: 20)

	Score Average	Total Time (s)
RHC	145.75	2.012255595
SA	177.5	3.62439526
GA	145.55	2.087058856
MIMIC	153.05	414.3010526

It is clear from the tables that Simulated Annealing is the one that got closest to the optimal solution. As mentioned above the optimal solution would be an alternating string of the size of the input, and as shown in the tables the Simulated Annealing sizes were by far the closest to the input string size. I believe that Simulated Annealing worked the best because it works best in spaces where neighbors and local optima are nearby and in flip flop each neighbor is just the flip of one bit in the bitstring. It is curious to me that Randomized Hill Climbing did not perform as well because it also works best in the previous stated situation. But, I think this is because Randomized Hill Climbing will get stuck when it gets to a string where changing any bits will not reduce entropy, but Simulated Annealing will not have this problem if it is given a good temperature which in this case it was. Simulated Annealing was marginally slower than Randomized Hill Climbing and Generic Algorithms, but MIMIC was much slower in all the cases. I was curious about how this would perform with different input sizes, so I also ran this on all the algorithms with inputs of 100, 150 and 180. I maintained 20 iterations for all of them. It is interesting that no matter the input size the trends between the algorithms remained the same. Simulated Annealing consistently got the closest to the optimal and took around the same amount of time for each input size. While MIMIC consistently took the most time and only occasionally performed slightly better than Genetic Algorithms and Randomized Hill Climbing.