Randomized Search

Eric Gu
November 6, 2018

This notebook will explore 3 random local search algorithms:

- randomized hill climbing
- simulated annealing
- a genetic algorithm

We will use the three algorithms instead of backpropagation to find good weights for a neural network, training on one of the datasets we explored in Assignment 1. Additionally, we will explore two well-known optimization problems using the same three algorithms.

Written using ABAGAIL. Data sourced from Kaggle.com.

Neural Network Optimization

Dataset Recap

For our neural network, we'll explore Breast Cancer in Wisconsin—a relatively small, less balanced dataset on breast cancer diagnostics published in a paper from the University of Wisconsin. It was donated to the UCI Machine Learning repository in 1995 and is available on Kaggle.com.

There are 569 instances and 30 features.

Features are computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. They describe characteristics of the cell nuclei present in the image. Ten real-valued features are computed for each cell nucleus. The mean, standard error and "worst" or largest (mean of the three largest values) of these features were computed for each image, resulting in 30 features.

After shuffling the rows, we split the data for training, testing, and cross validation. 80% of the data was used for training, the remaining 20% for testing. We used 5-fold cross-validation.

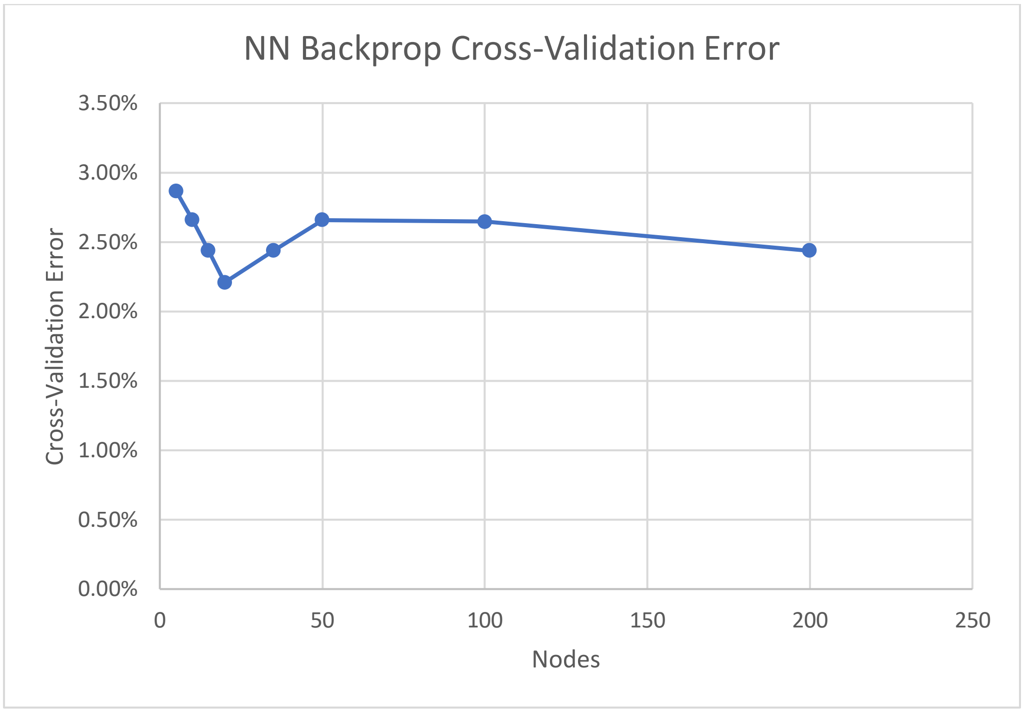
Description of Experiments

Using existing code from ABAGAIL, we implemented Randomized Hill Climbing (RHC), Simulated Annealing (SA), and a standard Genetic Algorithm (GA) to train a neural network on my dataset. Because of the small size of the dataset, the neural network is fully-connected and has only 1 hidden layer with units that use tanh activations (more convenient for neural networks). Loss function was L2-norm, or sum of squares error.

To start, we used backpropagation and gridsearch CV to adjust hyperparameters and the number of nodes in the hidden layer, and to provide a baseline performance for comparison against the random search algorithms. For all algorithms, the neural network used a dynamic learning rate between 0.000001 and 50, with an accuracy threshold of 1e-10 to ensure training ends only when the algorithm is very sure the output is correct.

Backpropagation

Because the dataset is so small, we saw in Assignment 1 that even 25 training iterations was sufficient to converge on a solution using `lbfgs` solver. A rule of thumb is to use a hidden layer size somewhere between the input layer size and output layer size, or around (30 + 1) / 2 = 16. Using 200 iterations, we search for the optimal number of nodes in the hidden layer to use in this particular problem for the remaining algorithms.



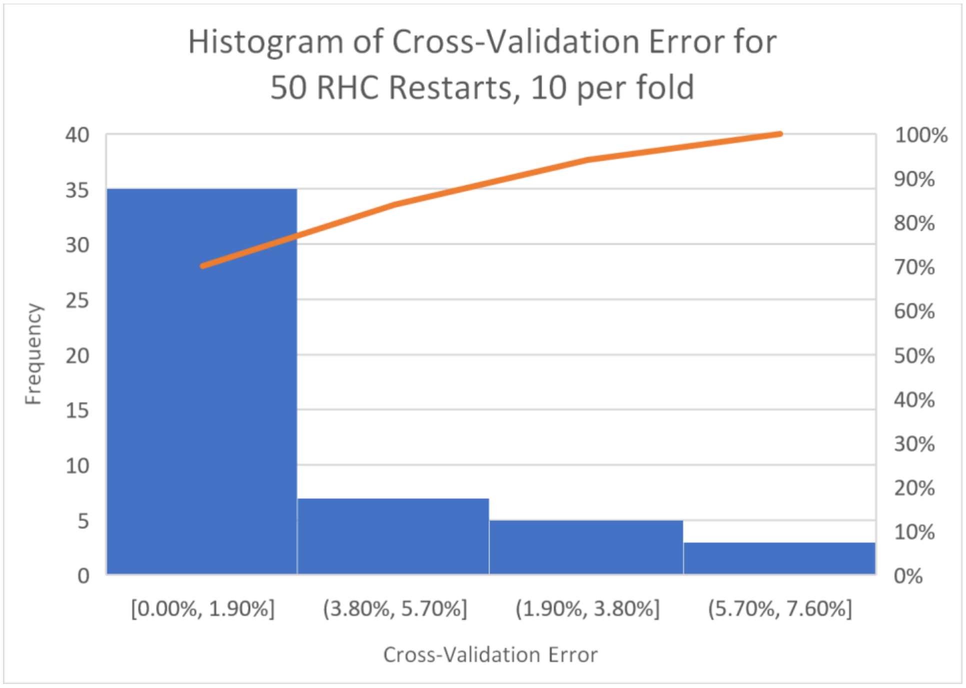
It turns out that the best network has 20 hidden layer nodes, where the average cross-validation error is 2.21%. Exceptional! For comparison, in Assignment 1, the MLPClassifier with ReLU activation from sci-kit learn achieved an almost identical error of 2.2%. When we fit the neural network on the training set and evaluate against the test data, we find a test error of 7.018%.

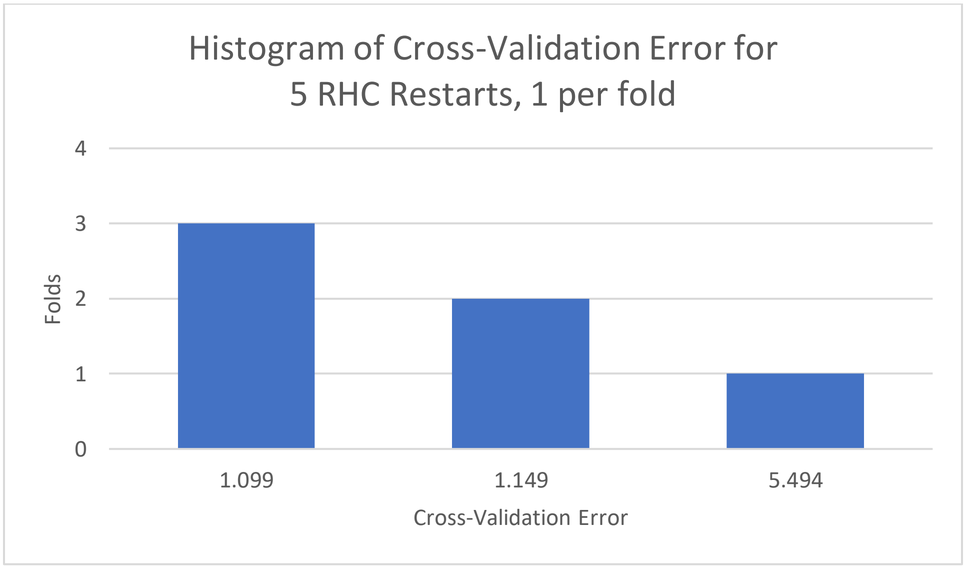
Randomized Hill Climbing

We implemented “randomized hill climbing” in more than one sense: 1) the algorithm randomly chooses a move from the moveset at each state, and if the neighbor’s value (loss) was better than the current state, it moves to that neighbor; 2) the algorithm runs until it has no neighbors or it reaches the maximum number of steps (we set to 2000), and then restarts with new random weights on a different fold of training data. The algorithm runs 10 restarts on each fold, which totals 50 restarts. Weights are randomly assigned between [0, 1], and neighbors are determined by randomly choosing a weight and adjusting it by a random value uniformly distributed between [-0.5, 0.5].

Curiously, this gave *worse* test results (4.386% error) compared to a version of the algorithm that ran without random restarts each fold (3.509% error), and naturally took 10 times longer to run. That is, when the algorithm ran only once on each fold, the model's cross-validation error was higher (1.099% vs 0.000%), but the test error ended up lower. This is a clear indication that running more random restarts and choosing the model based on the best cross-validation score results in overfitting. It should be noted that a 0% validation error is extremely unlikely and is a sign of the relatively small problem space.

There are no hyperparameters to be tuned. From running the algorithm only once on each of 5 folds, we found that the lowest validation error was 1.099%, which would indicate better out-of-sample performance than backpropagation. We use that network to evaluate the test data, and get a test error of 2.632%, significantly lower than backprop just as predicted.





The training time for RHC is significantly higher primarily because of a couple reasons:

1. Random moves take significantly longer to converge on a solution than backpropagation, which uses gradients (2000 iterations vs. 200 iterations).
2. Cross validation is used to select the weights for the neural network that will go on to evaluate test data and make a final prediction, so the model is fit to most of the training data at least *k* times, whereas backpropagation needs only to train one model. This compounds the performance difference from more iterations per model as mentioned in reason #1.

However, for the same reason the training time takes longer, RHC performs better on test data because it takes more time to search out the problem space in directions that may not be immediately obvious when using ordinary gradient descent. More importantly, the random restarts help to find better starting weights that lead to a better local optimum for out-of-sample data compared to ordinary backpropagation.

Simulated Annealing

Simulated annealing is like RHC, except that instead of using random restarts to escape local minima, the algorithm occasionally accepts bad moves to neighbors. The probability of this is determined by the Boltzmann distribution, which starts with some “temperature” that cools (or “anneals”) over time, thereby reducing the probability of accepting bad moves the longer the algorithm runs. At higher temperatures, the algorithm behaves like a random walk, and at lower temperatures, it converges like stochastic hill-climbing.

By running gridsearch CV for 400 iterations on each fold for many cooling rates between 0.8 and 0.999 and temperatures between 1e3 and 1e15, we determined the best hyperparameters were starting temperature = 1e8 and cooling rate = 0.80. That range of possible hyperparameters sufficiently covers the range of reasonable behavior desirable in a simulated annealing model. However, its error on the cross-validation folds were alarmingly high, and the cause could not be determined—it’s possible that variance in the training data happened to fall along the CV folds.

Using those hyperparameters, we train on the entire training set for 2000 iterations before evaluating on the test data, just like RHC. We achieve a training error of 1.538% and a test error of 3.509%. The test error is somehow identical to RHC’s, which suggests one of several possibilities: the test data has that amount of noise and both models are accurately predicting all other instances, or the topography of the problem space has many similar routes to the same minima or many identically valued minima, or an enormous coincidence has occurred. At any rate, because simulated annealing needs to run only once on the training data, it completes quite quickly—slower than backprop, but much faster than RHC.

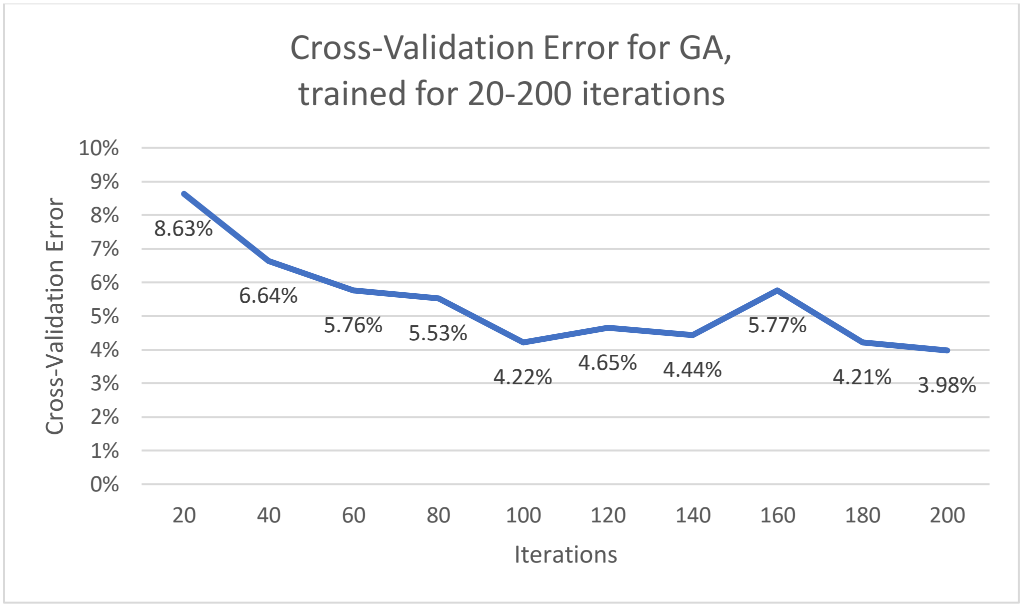
Genetic Algorithms

Instead of modifying a single set of weights (which we’ll call an “instance”) at a time like RHC or SA, GA-based optimization generates multiple sets of weights, called a population. We use a standard implementation of genetic algorithms from ABAGAIL, which both “mates” pairs of instances to create offspring and “mutates” random members of the new population. This results in interesting behavior unlike that observed in RHC or SA, because instances can change dramatically and explore new areas of the problem space instead of incrementally changing single weights at a time.

In each generation, the error measure is calculated for each individual and determines the probability distribution for selecting it for mating, mutation, or survival into the next generation—the better it performs on that fold of training data, the more likely it is to be selected. Offspring are created from a 1-point crossover of the attributes (neural network weights) from 2 randomly selected parents; they take the first bits from the second parent at a randomly determined dividing point in the bit string, and the remaining bits from the first parent. Any leftover quota in the new generation's population is randomly directly filled from the old generation according to the same probability distribution mentioned earlier. All new generation instances are then mutated with according to the mutation probability.

To fit hyperparameters for our genetic algorithm, we ran all combinations of population ratio = {0.10, 0.15, 0.20, 0.25}, mating ratio = {0.02, 0.04, 0.06}, and mutation rate = {0.02, 0.04, 0.06} on 5-fold cross-validation. This gridsearch recommends a population ratio of 0.20, mating ratio of 0.06, and mutation ratio of 0.04.

The algorithm did not perform very well when fitting the different hyperparameters during cross-validation, with the lowest k-fold cross-validation error of an instance being 6.644%. However, when using the tuned hyperparameters and fitting for more iterations, the cross-validation error improved to 3.98% at 200 iterations. Including 'training iterations,' tuning all 4 hyperparameters in tandem takes over an hour of wall clock time but finding the hyperparameters and then tuning iterations with those hyperparameters fixed takes only a few minutes.



After running the tuned GA at 200 iterations on the test data, we see a training error of 7.033% and a test error of 6.140%. In general, the cross-validation performance measured against training iterations left something to be desired; the unsteady downward trend suggests that 1) more training data would help, and 2) that genetic algorithm performance is unreliable due to its stochastic nature, even with the same CV folds.

Neural Network Summary

Algorithm	Training Time	Training Error	Lowest Cross Validation Error	Test Error
Backpropagation	0.172 s	0.000%	2.647%	7.018%
Randomized Hill Climbing	5.430 s	1.111%	1.099%	3.509%
Simulated Annealing	1.335 s	1.538%	12.419 %	3.509%
Genetic Algorithm	5.479 s	7.033%	3.976%	6.140%

Interestingly, the RHC and SA algorithms had higher training error than backpropagation but lower test error. It's difficult to say why they were able to outperform gradient descent, but it's likely that their ability to randomly search the state space enabled them to find more generalizable solutions that resisted overfitting to the training data. In a way, backpropagation could be thought of as "too greedy" for a reasonably complex problem with comparatively little data like this one. The genetic algorithm barely outperformed the baseline on test data, but it came with tremendous computational requirements that make it an unlikely candidate for practical applications.

Two Randomized Search Optimization Problems

In this section, we will apply all three search algorithms to the max k-coloring and four peaks problems to highlight the strengths of genetic algorithms and simulated annealing, respectively.

Max K-Coloring

A graph is K-Colorable if it is possible to assign one of k colors to each of the nodes of the graph such that no adjacent nodes have the same color. The fitness function is defined to be the iterations required to find if k colors can be assigned to the graph, which ought to be minimized.

This problem is NP-complete. Multiple solutions may exist for a given graph, so genetic algorithms are well-suited for converging on one or more "good" solutions for k-coloring a graph, even if it's not necessarily the best one. Hill-climbing algorithms may struggle to necessarily find a k-coloring in the first place due to their greedy strategy, and struggle to cheaply evaluate an incremental change for graph coloring problems.

We randomly generate a graph of 50 vertices with 4 adjacent nodes per vertex, and K = 8 possible colors. Each state is generated and stored as a permutation-based encoding for RHC, SA, and GA.

We try to solve using RandomizedHillClimbing, SimulatedAnnealing, and StandardGeneticAlgorithm from ABAGAIL without random restarts. We train both RHC and SA for 20,000 iterations, using temperature = 1e12 and cooling rate 0.1 for SA. For the genetic algorithm: single crossover, the population size = 200, mating ratio = 0.05, and mutation ratio =

0.30, trained for 50 generations. These hyperparameter values are empirically sourced from the ABAGAIL test library.

Algorithm	**Found Max K-Coloring**	**Fitness Score**	**Average Wall Clock Time**
Randomized Hill Climbing	No	340	0.079 s
Simulated Annealing	No	340	0.077 s
Genetic Algorithms	Yes	350	0.019 s

As predicted, GA performed the best by far, managing to converge on a solution and running in shorter time. The solution space for this K-Coloring problem likely has many plateaus with neighboring states that do not offer meaningful improvement to the fitness function, which would explain the poor performance for RHC and SA. GA, on the other hand, performs better because any successful crossover is representative of some of the underlying structure of the graph, and can simultaneously "see" more of the state space at once.

Four Peaks

Given an N-dimensional input vector $\vec{X}$, the four peaks evaluation function to be maximized is defined as:

$$f(\vec{X},T) = \max\left[tail(0,\vec{X}), head(1,\vec{X})\right] + R(\vec{X},T)$$

where

$$\begin{aligned} tail(b,\vec{X}) &= \text{number of trailing } b\text{'s in } \vec{X} \\ head(b,\vec{X}) &= \text{number of leading } b\text{'s in } \vec{X} \end{aligned}$$

$$R(\vec{X},T) = \begin{cases} N & \text{if } tail(0,\vec{X}) > T \text{ and } head(1,\vec{X}) > T \\ 0 & \text{otherwise} \end{cases}$$

The parameter space is discrete-valued and can be thought of as bit strings. There are two global maxima: 1) there are T + 1 leading 1's followed by all 0's or 2) when there are T + 1 trailing 0's preceded by all 1's. There are also two suboptimal local maxima, which occur with a string with all 1's or all 0's.

The problem has relatively few suboptimal local optima, but the "gravity well" or basin of attraction for those local optima grow as T grows larger—fewer and fewer instances in the problem space are able to meet the qualification needed to earn the large reward from R($\vec{X}$, T), so the best neighbor for many states is to increase the leading 1's or trailing 0's in the bit string to improve the value given by the max function. This takes you farther away from the global maximum.

Interestingly, as the dimensionality of the problem space, *N*, grows, so too does the "depth" of both local and global optima. The potential reward of finding a global optimum grows in proportion to the size of the search space. The downside, of course, is that that global optimum is harder to find.

We construct the problem where N = 50, T = N / 10, and solve using RandomizedHillClimbing, SimulatedAnnealing, and StandardGeneticAlgorithm from ABAGAIL with 10 random restarts each. We train both RHC and SA for 200,000 iterations, using temperature = 1e11 and cooling rate 0.99 for SA. For the genetic algorithm: single crossover, the population size = N * 10, mating ratio = 0.50, and mutation ratio = 0.05, trained for 1000 generations.

Algorithm	**Fitness Score**	**Average Wall Clock Time**
Randomized Hill Climbing	50	0.049 s
Simulated Annealing	85.2	0.134 s
Genetic Algorithms	85.4	0.186 s

Hill climbing fails to find the global optimum even once with 10 random restarts—though T is not large, it's still unlikely for the random initialization to land in the small, relatively steep basin of attraction for the two highest peaks. It runs by far the fastest, nearly 3 times faster than simulated annealing.

Simulated annealing, on the other hand, found the global maximum because the local maxima were "shallow" enough (N was not large) and the large temperature and slow cooling rate allowed for more than sufficient exploration. Simulated annealing is consistent, too: when randomly restarted, SA found the global maximum 8 out of 10 times.

Genetic algorithms were too random at N = 50. The problem state allows for 2^{50} possible states, intractable for a population size of only 50. Since GA can't take advantage of exploring neighbor states, it relies on chance and having individuals in the population close enough to the optimal solution. However, with population = 500 and proportionate mating and mutation rates, the algorithm exceeds the fitness of simulated annealing through brute force, with about a 50% performance loss in speed.