Randomized Optimization
(ML Assignment 2)

Silviu Pitis
GTID: spitis3
silviu.pitis@gmail.com

1 Neural Network Optimization

A Dataset recap (MNIST: Handwritten digits)

As in Project I, I use a subset of the full 70,000 MNIST images, with the following training, validation, test splits:

- **Training:** First 5000 samples from the base test set
- **Validation:** Last 5000 samples from the base test set
- **Test:** All 5000 samples from the base validation set (using the typical 55000-5000 training-validation split of the base data)

B Description of Experiments

I implemented Randomized Hill Climbing (HC), Simulated Annealing (SA), and Genetic Algorithms (GA) in Tensorflow/DEAP and used each approach to train a very simple neural network on my dataset. The neural network is fully-connected and has a single hidden layer with 100 units that use tanh activations. The loss function is defined as the mean categorical cross entropy over the 10 labels. Since this architecture differs somewhat from the one used in Project I (which also used a powerful optimizer), I also ran a (backpropagation) baseline using gradient descent.

Although I simplified the network considerably, as compared to Project I, this network still has **`101770`** weights. While this seems large, it is very small for modern neural networks. Due to the size of this parameter space, my thoughts going into this experiment were that all randomized methods would completely fail due to the curse of dimensionality. Last semester, one of the questions on our CS6601 exam was: which of these methods could be used to train a neural network. I answered that they all *could* be used to train a neural network, but good luck getting them to work in any reasonable amount of time! We'll see below that my hypothesis was spot on.

Gradient Descent (baseline)

I use minibatch gradient descent with a minibatch size of 50 and a learning rate of 0.1. Weights are initialized with the Xavier Glorot[1] initialization. The model trains for 37 epochs before the validation loss starts going up and training terminates. It achieves accuracies of 99.8%, 92.8% and 92.1% on the training, validation, and test sets respectively. These results are fairly consistent between trials (initialization does not matter much). Training takes about 5 seconds.

By contrast, the 2-layer network with 350 ReLU units in each layer, trained with Adam, which I used in Project I, was able to score approximately 95%. on validation/test.

Randomized Hill Climbing

I assumed that by "randomized" hill climbing, the assignment was asking for two kinds of randomness: first, random restarts once hill climbing gets stuck, and second, random selection of

1 See Glorot & Bengio (2010), http://jmlr.org/proceedings/papers/v9/glorot10a/glorot10a.pdf.

candidate weight updates (neighbors). I tried a couple different options for choosing candidate updates, each time doing 20 random restarts.

Random restarts

For each restart, weights were initialized uniformly in [-0.5, 0.5]. This is much larger than the Xavier Glorot initialization used for the Baseline (for the first layer, that would be sqrt(6 / (784 + 100)) => [-0.082, 0.082]). My reasoning is that this allows for even more exploration of the weight space, although I'm not convinced this is helpful given that I only did 20 restarts. My reason for limiting the number of restarts to 20 was that, even with GPU-accelerated evaluation, each training episode took 10-30 seconds or more, which is very long for such a simple network. More restarts would be prohibitive.

Methodology and preliminary experiments

At each time step, my network chooses a set of weights that neighbors the best set of weights so far, evaluates the network with the new set of weights. If the result (training loss) is better than the current best set of weights, the best set of weights is updated. All evaluations are done on loss function with respect to the entire training set (that is, cross entropy loss, not accuracy). Because evaluations on 5000 examples at once are expensive, I did the preliminary tests with respect to poorly performing configurations using only the training set (no validation). Training with a single restart was terminated if the algorithm failed to find a better neighbor after 3000 consecutive tries, and also, if validation was used, if the validation loss started to rise.

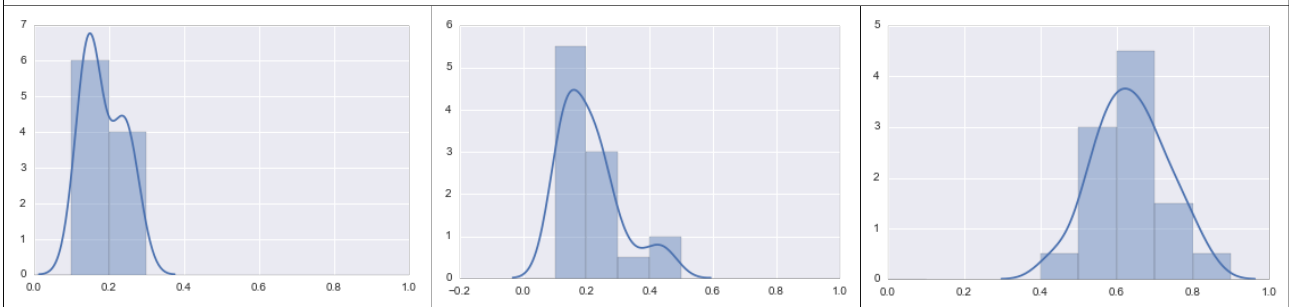
I tried a few different configurations for choosing neighboring sets of weights. All involve incrementing the vector of all weights by a small delta. In each case the delta was determined as the sum of two components: a random vector and a gradient vector.

Each component of the random vector was chosen from a normal distribution with zero mean and standard deviation σ.[2] It is also possible to think about this as a product of a randomly unit direction multiplied by a scalar slope, where the scalar slope is proportional to σ. Preliminary experiments (detailed results omitted) showed that an annealed σ (or slope) performs best, so for each configuration I initialized σ as 0.01. Each time the algorithm failed to find a better neighbor after 100 tries, σ was annealed (multiplied) by a factor of 0.95.

Initially I did not use the gradient component of the weight vector, and chose random neighbors. This did not work very well, and so I decided to add the gradient vector multiplied by some factor (the learning rate). I tried various configurations to see how important the gradient component was. I found that as the learning rate approached the gradient descent baseline learning rate (0.1), the algorithm's performance improved dramatically (shown in Figure 1).

Figure 1. Histograms showing performance of neighbor selection configs over 20 restarts

In each case, final accuracy on the training set is shown on the x-axis (which goes from 0.0 to 1.0). As you can see, increasing the gradient component greatly increases the efficacy of HC.



(A) No gradient component | (B) Gradient factor = 0.001 | (C) Gradient factor = 0.01

2 I also tried a uniform distribution in a very preliminary experiment, and it did not make a significant difference. My reasoning for ultimately choosing a normal distribution is that it is sparser is large components, which makes it closer to changing only 1 of the weights, which I understand is a common approach to choosing neighbors in discrete cases, but is impractical here with over 100,000 parameters.

Best configuration, and comparison to gradient descent baseline

The best configuration used had a gradient factor (learning rate) of 0.1 (the same as the gradient descent baseline). The differences of this configuration and the baseline are that (1) the learning rule here uses full batch learning, as opposed to mini-batches, (2) some randomness is introduced via the random neighbor component, and (3) we do 20 random restarts with a wider weight initialization. Although this best configuration consistently scored above 85% on the training set (using the validation stopping rule), it was only able to score about 87.8% on the validation set and 86.5% on the test set. This indicates that my randomized hill climbing rule overfits the data faster than the gradient descent baseline. I believe this is because (1) Xavier Glorot initializations are much better than the wide range used here, and (2) the gradient noise introduced by mini-batch learning (which only approximates the full batch gradient) is a better regularizer than the gradient noise introduced by adding a small noise component to the full batch gradient. Because hill climbing took much longer than the baseline to train (about 4 minutes for 20 restarts), I did not try more configurations to see if I could match the baseline results.

Simulated Annealing

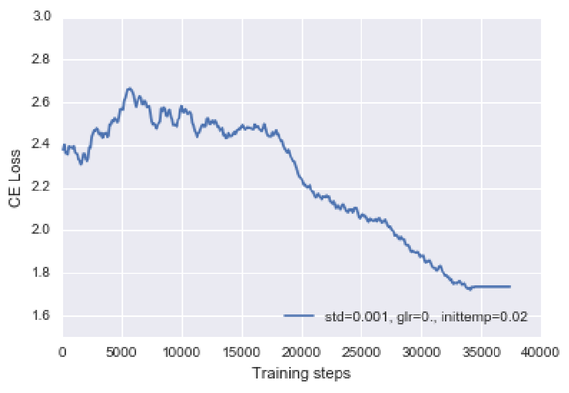
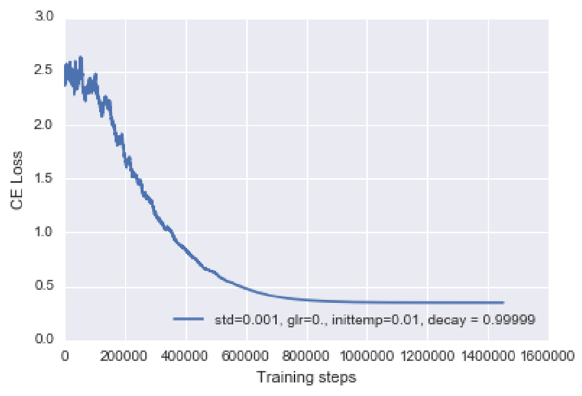
Simulated annealing is similar to random restart hill climbing, except that instead of using random restarts to escape local maxima, we accept bad steps (weight changes that increase loss) with some probability p, determined by a Boltzmann distribution. The key factors that determine the acceptance probability for bad steps are (1) how bad it is (difference between new loss and old loss), and (2) the so-called “temperature”. When temperature is higher, the probability of accepting bad steps is higher. Temperature is initialized to some value, and then decreased (‘annealed’) during training. I chose an exponential annealing schedule, so that at each step the current temperature was multiplied by some factor $0 < a < 1$.

Candidate weight increments at each step were determined according to a random vector (with normally distributed components). Unlike with HC, where a gradient bias was necessary in order to achieve a good fit, I was surprised to find out that with proper tuning of the initial temperature and annealing rate, simulated annealing was able to fit the training data about as well as the gradient descent baseline, even when no gradient bias was used.

Convergence was determined in the same manner as HC. Because poor neighbors were accepted with positive probability, the algorithm only converged once temperature was effectively zero (see also the section on validation below).

Below are two graphs showing the learning progress of two different simulated annealing configurations. Both use a standard deviation of 0.001 for the random component and no gradient bias when determining candidate weight increments. Note that the second graph has 40x as many training steps as the first graph, and that the y-axes have different ranges. These parameters were determined after several initial experiments, using temperatures much higher than 1 (which all diverged) decay rates as low as 0.9 (which all converged too quickly to bad values).

Figure 1. SA training progress for two different temperature configurations

 Initial temp: 0.02, annealing factor: 1 - 1e4	 Initial temp: 0.01, annealing factor: 1 - 1e5

Validation, performance and generalization

Running the second configuration above (1.6 million training steps) took over 2 hours on a GPU. I had not run the experiment with early stopping / periodic evaluation of validation performance because I was not expecting it to come anywhere close to over-fitting territory (my philosophy on this is to first get a reasonable fit on the training set, and only then start worrying about validation performance). Due to the time requirements, and still lackluster performance (below), I did not repeat the experiment with early stopping / validation.

Unfortunately, although it was able to achieve decent accuracy on the training set (89.0%), the second configuration above badly overfit (relative to gradient-based methods), achieving accuracies of only 81.8% on validation and 80.8% on test. I was surprised by this, having expected better generalization than the more aggressively fitting gradient-based optimization methods.

Therefore, it would seem that following the gradient is actually a good inductive-bias to have, at least for this data set (gradient descent on this data tends not to overfit until training reaches about 90% accuracy). In retrospect, this makes sense: even though following gradient more aggressively fits the data, this aggression relies on picking out the best direction, and directions that simultaneously benefit more training examples (i.e., that do not memorize a specific training example) are going to generalize well. By contrast, picking a random direction, like SA does, is going to stumble into pockets that improve the fit only with respect to a select few training examples, leading to more memorization.

This spells bad news for SA is general: it is a method for finding the best fit on some set of data, but the best fit (global optimum) on that set of data will not necessarily generalize the best to the unseen data. For example, many configurations of the simple neural network described here are going to fit the training data perfectly (there are multiple global optima with respect to the training data). Some will generalize better than others. It seems that SA does not really care which global optimum is lands on (should we have enough patience to let it land on one). On the other hand, if gradient descent lands on a global optimum, it will have gotten there by choosing directions that benefit the most training examples, which will tend to bias it towards global optima that generalize well.

Genetic Algorithms

Instead of keeping a single set of parameters at any given time like HC and SA, GA-based optimization keeps multiple sets of parameters (a population). The method of producing the next population is also quite different: in addition to mutating some members of the population (i.e., replacing them with a neighbor, as in HC an SA), we also “crossover”, “breed”, or “mate” pairs of members to form new sets of parameters that are not neighbors of any prior set of parameters. This is interesting because the members of our population can bounce around and explore entirely new areas of the parameter space (but see * below). Although the mutation and the breeding are blind to the fitness value (unlike HC and SA), GA-based optimization works by a “survival of the fittest” principle: at each time-step, we apply a “selection” operator that favors keeping the fitter members of the population and discards the rest.

To implement GA-based optimization, I use Python’s DEAP library (https://github.com/DEAP/deap), which provides most of the necessary components (selection, mutation, and crossover operators, in addition to abstractions that manage the population and its members, and to automate the training loop). I defined an “individual” to a flattened real vector of all 101770 weights in my network, and the fitness of that individual to be the negative of my networks cross entropy error when evaluated using that set of weights.

Parameters and Validation

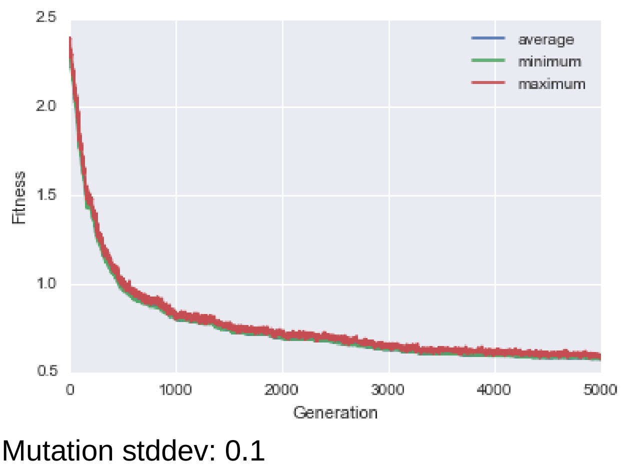
Like HC and SA, GA-based optimization was extremely computationally expensive, as compared to the baseline gradient descent. Computing 50 iterations (generations) took between 5-20 seconds or more, depending on the parameters used (discussed below). This is because, with a population size of 50, each generation requires about 20-30 fitness evaluations. DEAP offered a several hyperparameters to play with, more than any algorithm we have studied so far, and I did not have (and still do not have) good intuitions about them. I was able to conduct some preliminary tests on the first 50 generations of learning, which led me to do full tests of two different configurations. The hyperparameters I experimented with are as follows:

- **Overall genetic algorithm and selection criteria**: I tried both eaSimple+selTournament, and eaMuPlusLambda+selBest. The latter is the version we discussed in class (pick the k best of each population, and breed them to form the other members of the population). The former first samples a full population via subsampling, and then replaces some members with crossovers. This did not seem to make a big difference in results.
- **Crossover probability:** I tested a variety of values. Values that were too low resulted in very slow learning over early generations. Values that were too high did not seem to improve learning, and required more fitness evaluations per generation, which slowed down training. I settled on using a probability of 0.4, but I do not know if this the best choice for performance in later generations, since tests were only done on the first 50.
- **Crossover operation:** I tested a 1-point crossover (exchanging everything to the right of a random point, as in the lecture) and 2-point crossover (exchanging the middle area between two random points). I found the 1-point crossover to be more effective, but I can think of no reasonable explanation for this, and think it may be an anomaly.
- **Mutation probability and mutation:** The mutation used was to increment indpb% of the parameters by a normal distribution with stddev sigma. Mutpb% of the population was mutated on each generation. Preliminary experiments found that any intermediate values of indpb and mutpb showed similar results. I settled on indpb=0.05, and mutpb=0.2. The stddev value (i.e., which neighbors to choose when mutating) had a very large effect on the rate and quality of learning (values below).

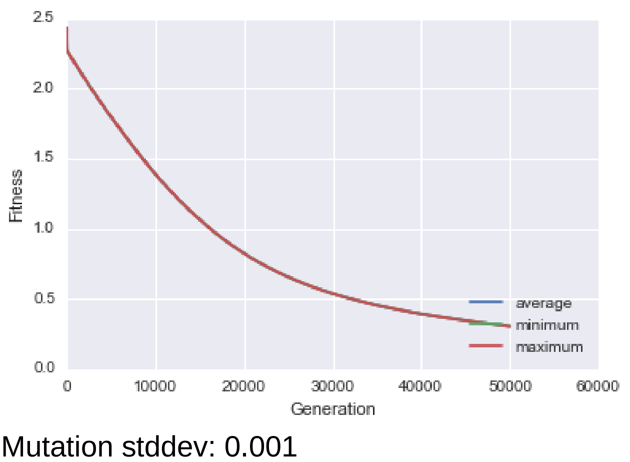
Configurations, Performance and Generalization

The two configurations I ended up doing full tests of used stddev values of 0.1 and 0.001 (other hyperparameters as stated above). The former configuration learned faster, requiring 5,000 generations. The latter required 50,000 generations to learn, which took about 1.5 hours. The learning curves (training only) are shown in figure 3.

Figure 3. GA training progress for two different mutation configurations



Mutation stddev: 0.1



Mutation stddev: 0.001

As you can see, the high stddev configuration learned faster, but flattened out much earlier, which suggests that mutation sizes would benefit from annealing. In terms of accuracy the high stddev configuration scores 82% on train, 80.1% on validation and 78.9% on test. The low stddev configuration scores 91.4% on train and 89.2% on validation (I accidentally overwrote the data before computing test). Although it did not quite reach the gradient descent baseline (it is possible that it could, given a few more hours of training), the great generalization performance was very surprising to me after seeing how simulated annealing faired. I am at a loss for a good explanation for why this generalized better than SA. It could be chance, it could be specific to this problem, or it could have something to do with the mutations were sparse, as opposed to the dense neighbors I used for SA.

C Neural Network Summary

Learning algorithm	**Best generalization accuracy**	**Time to train**
Gradient descent basline	92.8%	~5 seconds
Randomized hill climbing	<30%	-
Randomized hill climbing w/ gradient component	86.5%	~4 minutes
Simulated annealing	80.8%	>2 hours
Genetic algorithms	89.2% (validation)	~1.5 hours

Overall, I don't think I will be using randomized optimization algorithms to optimize neural network parameters any time in the future. No algorithm was able to get close to the gradient descent baseline, and no algorithm was able to do so in any reasonable amount of time. Moreover, this was *after* I reduced the number of parameters from about 500K to a more manageable 102K.

2 Discrete Optimization over Bitstrings

The second part of this assignment asked that I compare the algorithms over 3 bitstring optimization problems. Each of the three problems should highlight the strengths and weaknesses of a specific algorithm.

Experimental Design: In the MIMIC paper, the experimental design assumes that the maximum is known and achievable. It is also not stated how many trials were run. I argue that this is poor design, as we will not in general know what the maximum is, and there is potentially high variance for every trial. Therefore, I take a different approach: although the maximum can be computed in each of my chosen problems, I do not precalculate it and count the number of iterations until it is reached. Instead, I use a *patience* parameter. Patience specifies the number of tries (learning steps) the algorithm attempts in order to improve its current best maximum before giving up (terminating). The algorithm then returns its current best maximum, even though that maximum may very well be a local maximum. Additionally, I run 30 trials for each algorithm and compute both best case and average results over all 30 trials. This reduces the impact of variance due to randomness.

Hyperparameters: There were a lot of hyperparameters to choose for each algorithm, but I stuck to some "typical" values across all experiments, with some tweaking of the patience parameters so as to not have too large a disparity in wall clock time between algorithms (I'm an equal opportunity experimenter). The one hyperparameter I tinkered with more than others was the temperature settings of simulated annealing, since I found it to be sensitive to different values. The precise values used are available in the IPython notebook if you are interested, but I do not believe they are that important (the purpose of this exercise was to demonstrate strengths/weaknesses, not tune the algorithms, and preliminary experiments showed that tuning did not have a significant effect).

Neighbors for RHC and SA: I defined a neighbor of a bitstring, for purposes of RHC and SA, to include all bitstrings at a hamming distance of 1.

GA Configuration: The GA used used a one-point crossover operation, together with the eaSimple+selTournament style learning algorithm (each as described in the neural network section). Mutations consisted of flipping each bit of a mutated individual with probability mutpb.

MIMIC: For MIMIC, the probability density estimator used is very different from what we learned in lecture / Isbell's paper. I used Tensorflow to build a densely connected projection from a hidden layer of latent variables, all uniformly distributed. At each iteration a gradient descent step is taken towards maximizing the probability that the top X best examples would be generated by the network's projection. This works nicely, and is very fast. I believe the neural network approach is superior to the dependency tree based approach because it is naturally capable of dealing with continuous parameters, it does not favor any individual parameter (there are no parent-child relationships between parameters), and the complexity of the generating network is easily modified / expanded for more difficult domains. For example, in the experiments below I used a single logistic regression layer, which projected its outputs from a uniformly distributed latent layer. I believe this limits the joint probability distributions that can be represented, but we can easily expand its capacity by adding a hidden layer (e.g., using a 2-layer projection). We can also use this approach to extend MIMIC in order to optimize continuous parameters, by using a neural network that models a continuous probability distribution (e.g., a GAN architecture would extend nicely to the continuous situation).

A Sum+Parity: Simulated Annealing

Sum+parity is a problem I made up that creates a fitness function that has lots of local optima, but where the “depth” of the local optima is configurable. Note that the “depth” of local optima is relative to the overall shape of the fitness function (if the overall shape stays the same, even if things get absolutely further apart, we could, e.g., increase the temperature on SA to maintain the same level of rejection). The fitness of a bitstring is defined as its sum plus a parity bonus when the sum is an even number. My intention of designing this problem was to:

1. show that random hill climbing fails when there are lots of evenly distributed local optima, and
2. highlight the effect of local optima depth on the performance of simulated annealing.

I expected SA to work better than GA and MIMIC for shallow local optima, but otherwise, I did not have a good hypothesis for what would happen with GA and MIMIC.

I tested all algorithms at two levels of local optima depth:

1. a “shallow” problem, with parity bonus = 1.1. In this problem, [1 0 1 0] evaluates to 2 + 1.1 = 3.1. The neighbors between this local optima and the global optima ([1 1 1 1]) have fitness equal to 3, so we can say that the local optima are 0.1 units deep.
2. A “deep” problem, with parity bonus = 5. Here [1 0 1 0] evaluates to 2 + 5 = 7, and we can say that local optima are 4 units deep.

The results are below. In each case, bitstrings of length 50 were optimized.

	Shallow (parity bonus = 1.1)			**Deep (parity bonus = 5)**		
	Max / avg fitness	**Avg fitness evaluations**	**Avg wall clock time**	**Max / avg fitness**	**Avg fitness evaluations**	**Avg wall clock time**
RHC	39.1 / 37.0	40119	0.43s	45 / 40.7	74467	0.32s
SA	**51.1 / 50.9**	**15443**	**0.25s**	45 / 39.6	40047	0.44s
GA	51.1 / 48.7	4813	0.54s	**55 / 52.5**	**4586**	**0.57s**
MIMIC	**51.1 / 51.0**	**1717**	**0.89s**	**55 / 54.7**	**1818**	**0.70s**

I would argue that Simulated Annealing is the best algorithm for the shallow version of the problem. It achieves almost as good average results as MIMIC in a third of the time and almost always finds the maximum. We see that random hill climbing, as expected, gets easily stuck in local maxima. The depth of the local maxima do not affect the performance of RHC, GA or MIMIC, but they do affect the performance of Simulated annealing, which, as expected, performs noticeably worse on the deep problem. While we could try to adapt simulated annealing to the deep problem by increasing the temperature, this has only limited effect: the shape of the fitness function is different, and the depth cannot be directly accounted for via temperature adjustment.

I was surprised that GA performed very well on both shallow and deep problems, though in retrospect I recognize that there is a lot of structure inherent in the problem that favors the inductive bias of the GA. MIMIC performed well in both cases.

B Product of Sums of Consecutive Ones: GA / RHC

I thought this problem would demonstrate a situation in which GA outperforms MIMIC, because the joint probability distribution of good solutions is difficult to model (bad for MIMIC), whereas the fitness of bitstrings subdivides quite nicely into two consecutive crossover portions (good for GA). The fitness function is defined to be the product of the sums of consecutive ones, so that the string [1 1 1 0 1 1 1 1] has fitness 3 * 4 = 12.

I was surprised to learn that RHC also does very well on the problem; in retrospect, it is easy to see that the number of local optima here is not that large, so that RHC does not get stuck as often as in the first problem.

The results are below. In each case, bitstrings of length 50 were optimized.

	Max / avg fitness (in millions)	**Avg fitness evaluations**	**Avg wall clock time**
RHC	**1.33 / 1.13**	**16706**	**0.35s**
SA	1.02 / 0.85	20175	0.53s
GA	**1.33 / 0.99**	**5083**	**0.61s**
MIMIC	1.24 / 0.98	3030	0.76s

Note that while GA did outperform MIMIC in terms of everything but the number of fitness evaluations, MIMIC was still very competitive (it was very hard for me to find a problem on which my version of MIMIC did not do very well).

C Noisy Evaluation: MIMIC

Although the strength of MIMIC is quite clear from the deep Sum+Parity problem, I wanted to come up with a problem that MIMIC would work for, but where other algorithms would fail completely. I came up with the following stochastic optimization problem. Stochastic optimization is a common problem characterized by a noisy fitness fitness (e.g., stochastic gradient descent is optimization over a stochastic cost function; see, e.g., Kingma & Ba (2014) (https://arxiv.org/abs/1412.6980)).

First, we define a random vector of length equal to our bitstrings (I used a uniform distribution between -0.5 and 0.5 for each component). This vector never changes, and the same vector is used when evaluating all algorithms. Then, we define the “True” fitness of a bitstring to be the dot product of that bitstring with the random vector, but we only allow our algorithms to see the “noisy” fitness evaluation, which is equal to dot product of the bit string and a noisy random vector. The noisy vector is equal to the base random vector + noise vector whose components are normally distributed with mean zero and some standard deviation (I used 0.25).

I expected RHC and SA to fail completely on this (a single bad noise result is very hard to recover from, since they keep only a single point estimate at any given time). I expected both GA and MIMIC to do better, with MIMIC coming out ahead because its sampling method is naturally able to deal with noise (it is already a noisy learner!).

Results were measured using the “true” fitness, and are shown below. In each case, bitstrings of length 200 were optimized. To get a better understanding of the effect of noise, the noiseless results are also shown below, using the same base random vector.

	Noisy			Noiseless		
	Max / avg fitness	**Avg fitness evaluations**	**Avg wall clock time**	**Max / avg fitness**	**Avg fitness evaluations**	**Avg wall clock time**
RHC	8.1 / 4.6	12115	0.13s	25.4 / 25.3	45713	0.22s
SA	8.5 / 5.0	34373	0.69s	25.4 / 25.4	67355	0.38s
GA	14.0 / 11.5	5160	2.47s	22.2 / 20.1	6060	3.35s
MIMIC	**22.6 / 20.6**	**5656**	**0.30s**	25.4 / 25.4	11615	0.44s

The difference in results is striking. MIMIC is able to deal with noise very effectively, and find a solution that is very close to the global optimum of 25.4 (see noiseless results).

RHC and SA, as predicted, fail completely. Although they can deal with the noiseless version of the problem with little difficulty, neither is able to travel through a noisy fitness function. This suggests that both would also fail at mini-batch based optimization of the neural network in the first part of this assignment (mini-batches introduce noise to the cost function).

This problem also demonstrates a scenario in which MIMIC has a clear advantage over GA, which cannot even optimize the noiseless version of the problem (query as to why; unfortunately, the nature of GAs still remains a bit of a mystery to me).

3 Further work

Probably the coolest thing in this assignment was seeing how easy MIMIC was to implement and how effective it was with a neural network based probability density estimator. My version of MIMIC would be very capable of optimizing “HyperNetworks” (Ha et al. (2016) – https://arxiv.org/abs/1609.09106), which is something I’m interested in trying in the near future.

The least cool thing in this assignment was that GA still remains quite a bit opaque. I see them as being very arbitrary, with little theoretic underpinning. But MIMIC seems to have better effects, and is theoretically sound, so why use GA? Although I was able to cherry pick a problem where a GA using 1-point crossover outperformed MIMIC using a dense projection from a uniformly distributed latent space, this was only because I tuned the problem to the GA’s inductive bias. I could easily pick a probability estimator for MIMIC that achieves good results (e.g., I think a left-to-right sequence generator in the form of very small RNN or HMM would easily solve the Product of Sums of Consecutive Ones), but this also relies on my background knowledge. This being said, I could see certain scenarios in which GA with appropriate chosen, domain-specific mutations and crossover operations could be useful. E.g., I think problems like that in Jozefowicz et al. (2015) (http://www.jmlr.org/proceedings/papers/v37/jozefowicz15.pdf), which used GAs to find effective recurrent architectures, are particular well suited for GAs because they do not have a well defined domain. I can see ways of extending MIMIC to such a scenario (e.g., HMMs and RNNs naturally generate variable length sequences---not a fixed domain), but it is not as straightforward.