# CSC 492/592 BioInspired Computing

## Homework 2

Ian Carlson Derek Stotz

March 15, 2015

# Contents

# List of Figures

# List of Tables

# Document Preparation and Updates

Current Version [1.0.0]

*Prepared By:*
*Ian Carlson*
*Derek Stoz*

***Revision History***

| ***Date*** | ***Author*** | ***Version*** | ***Comments*** |
|---|---|---|---|
| ***2/2/15*** | *Ian Carlson and Derek Stoz* | *1.0.0* | *Wrote the Whole Thing* |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

# 1

# Evolutionary Algorithms - Text Chapter 3

## 1.1 Problem 3.1 and 3.2

Implement the various hill-climbing procedures and the simulated annealing algorithm to solve the problem exemplified in Section 3.3.3. use a real-valued representation scheme for the candidtate solutions (variable x). By comparing the performance of the algorithms, what can you conclude?

For 3.2, also implement an genetic algorithm to do the same.

For the simple hill climbing, try different initial configurations as attempts at finding the global optimum. Was this algorithm succesful?

Discuss the sensitivity of the algorithms in relations to their input parameters.

### 1.1.1 Problem Information

The goal of the problem in 3.3.3 referenced by the text was to find the global maximum of the equation $g(x) = 2^{-2(\frac{x-.1}{.9})^2} sin(5\pi x)^6$. By inspection of the graph given in the textbook, we know that the optimal x value is 0.1, and the global maximum is 1. All algorithms were evaluated by whether or not they reliably converged to this answer.

### 1.1.2 Deterministic Hill Climb

#### Algorithm Description

The first algorithm we implemented was the deterministic hill climb. In this program, we chose a random starting x value, and input a fixed number of time-steps. At each time-step, the algorithm checks to the left and right of the current position by a fixed step-size. If a better solution is found, the algorithm moves one step in that direction. This repeats until no improvement can be made, or the maximum number of iterations is reached.

#### Algorithm Performance

This algorithm, in general, was not succesful. Even when the starting domain was restricted between -1 and 1, the algorithm only converged to the global maximumin approximately 10% of the trials. In the other 90% of the trials, the starting x value was outside the "hill" of the global maximum, and it would converge to to a local maximum instead.

#### Sensitivity

We considered the sensitiviy of the algorithm to 3 input parameters: starting x value, step size, and number of time steps.

This method is extremely sensitive to the starting x value. If there is not a monotonicaly increasing slope from the starting value to the global maximum, deterministic hill climb will not find it. This algorithm does, however, always find the nearest local maximum, and does so very quickly.

The resolution of the final result depends on the step size, but as long as the step size is reasonably small it does not effect whether or not the algorithm will converge on a local maximum.

The final result is sensitive to the maximum number of iterations. If the distance between the starting x value and the x value of the local maximum is greater than the number of iterations multiplied by the step-size, the local maximum will not be reached when the program terminates.

### 1.1.3 Stochastic Hill Climb

#### Algorithm Description

The stochastic hill climb was a slight variation on the deterministic hill climb. Instead of looking left and right by a fixed step size, we instead would pick a random floating point number from a uniform distribution between -p and p, where p was one of our input parameters. At each time step, we would generate this random number and add it to the current x value. If the new location had a higher fitness than the old location, we would replace the current x value with that new x value. This algorithm terminated once the maximum number of iterations was reached, or if the fitness of the current x value was equal to the global maximum, which we knew to be 1 by examination of the graph of the function.

#### Algorithm Performance

This algorithm was hit and miss and highly sensitive to paramters, as will be discussed in the next section. In general, this algorithm performed better than the deterministic hill climb because it had the potential to jump out of local maxima. However, it also had the potential to jump away from the global maximum rather than converging.

For this particular problem, we were able to find a set of parameters that nearly always converged to the global maximum in the time allowed, but from a computational perspective, it wasn't much better than random search.

#### Sensitivity

We analyzed three input parameters: starting x value, perturbation size - referred to as p above, and maximum number of iterations.

The starting x value had some impact on whether the algorithm would converge, but not nearly as much as in the deterministic version because the algorithm had the potential to jump out of local maxima in favor of maxima with higher fitness. Of course, if the starting x value was randomly generated very close to the global maximum, it nearly guaranteed the algorithm would converge to the global maximum.

The most important parameter was the perturbation size. A very large perturbation allowed the algorithm to jump out of local maxima much more frequently, but it also took much longer to converge to any maxima, and may not converge at all if the maximum number of iterations was too small. If the perturbation size was very small, the algorithm would quickly converge on a maxima, but be unable to jump out of local maxima. The proper value for this parameter seems to be highly problem dependant, and is affected by the maximum allowable number of iterations. If an infinite number of iterations were allowed, having a very large perturbation size would be allowable. The algorithm may jump between maxima with a high frequency, but the algorithm never chooses a new x value with a lower fitness than the current one, so it would eventually find the global maximum.

The maximum number of iterations on its own had little effect on the final solution except in the case noted in the discussion about perturbation size.

### 1.1.4 Shotgun Hill Climb

#### Algorithm Description

This algorithm is also known as iterated hill climb in the text, but we liked the Wikipedia name a little better. The basic premise of shotgun hill climb is to run a hill climb algorithm many times with different starting positions, and keep track of the best of all runs. For this particular problem, we chose to iteratively

apply the stochastic hill climbing method. Since the stochastic hill climb nearly always converged, applying it with many different starting x values basically guaranteed convergence. In our application, we chose the starting x value randomly for each iteration, but another method would be to uniformly sample the feature space.

#### Algorithm Performance

This algorithm worked very well. For this problem, it might be the fastest and most reliable solution. However, for problems with very large and complex feature spaces, fully sampling the space might not be desirable or even possible. At the very least it is an easy first attempt for finding a least a very good local maximum of a complex feature space.

#### Sensitivity

The most interesting feature of the shotgun hill climb is its insensitivity to starting parameters. Even if the hill climb algorithm were set up with poor starting conditions or input parameters, applying the algorithm multiple times typically still manages to force convergence to a solution. We purposefully chose input parameters that caused poor performance with our stochastic hill climb, but the shotgun hill climb still converged to the global maximum every time we ran it.

### 1.1.5 Simulated Annealing

#### Algorithm Description

Simulated annealing attempts to combine the ability of the stochastic hill climb to jump out of local maxima with the high probability of convergence on a given maxima of the deterministic hill climb. The primary feature is the introduction of the temperature parameter T, which starts out high and diminishes. Much like the stochastic hill climb, a random perturbation is chosen at each time step. However, when T is high, there is a high probability that the algorithm will choose to make the jump even though the fitness at the new location might be lower. As T reduces, it becomes less and less likely that the location will jump to a less fit solution.

#### Algorithm Performance

We had trouble getting the simulated annealing algorithm to converge reliably. This was probably just because we were bad at picking the parameters. This algorithm seems finicky at best, and unreliable at worst.

#### Sensitivity

We analyzed the following paramters: T - the initial temperature, k - the cooling rate, p - perturbation amount. The initial temperature selection seems crucial and very problem specific. In addition, the initial T and the value for k are strongly related. A high T and low k will cause the algorithm to fail to converge in the alotted time. A low T and high k will force the algorithm to end prematurely when T hits near zero. Tuning these two paramters appears complex and more or less trial and error.

### 1.1.6 Simulated Annealing With a Twist

#### Algorithm Description

We had a thought, when we were having trouble with getting simulated annealing to work, that worked out pretty well so we thought we would include it. In traditional simulated annealing, the probability of accepting a random jump even if it is "worse" than the current location is high when T is high, and decreases as T decreases. Instead, we wrote an algorithm much like the stochastic hill climb, but instead of changing the probability of accepting a jump to a "worse" x value, we changed the size of the allowable jump as T

decreased. In other words, when T is high, the perturbations were allowed to be very large. As T diminished, the size of the jumps was reduced and we only accepted a jump when it increased the fitness of the solution.

#### Algorithm Performance

This performed much better overall than our traditional simulated annealing, as it allowed us to jump out of local maxima with very high frequency at the beginning, but allowed us to converge very accurately to a maxima at the end.

#### Sensitivity

Our modified simulated annealing algorithm seemed much less sensitive to input parameters. However, it did depended on having a high enough initial perturbation size to jump out of local maxima, which is problem specific information. In addition, the rate at which T diminishes appears to have some affect on how likely the algorithm was to get trapped in a local maxima.

### 1.1.7 Evolutionary Algorithm

#### Algorithm Description

Problem 3.2 requested a Genetic Algorithm for finding the maxizing x for the function $g(x) = 2^{-2(\frac{x-.1}{.9})^2} sin(5\pi x)^6$, where the mutation and crossover was done bitwise. Implemented in python, this was a new experience for us. However, the python bitwise operations turned out to be reasonably useable, and using some trickery we were able to find reliable ways to mutate and crossover python's abstract integers.

The starting population filled by generating n numbers between 1 and 0xFFFFFFFF. When fit, the number is divided and shifted over to reflect a floating point value between -1 and 1. This allowed us to use bitwise operators to modify representations of floating point values.

The overall algorithm was a traditional GA. The population was fit by plugging the values into the function, and the results were sorted. The top [selection rate] percent of "dudes", as we called our individuals, were then bred to create the next population. Breeding was done by choosing a crossover point in the bit string and copying the left half of one onto the other by means of bitmasking. Mutation, done [mutation rate] percent of the time after breeding, was done by flipping one random bit in the genome of the dude.

At the end of one generation, the best fit dude was printed and the loop started over again. After a specified number of iterations, the overall best dude was printed to the screen, showing its fitness and x value.

#### Algorithm Performance

This genetic algorithm performed exceptionally well. With the default parameters in the code, the best fit dude had a fitness of 1.0 approximately 90% of the time (or rather, .99999... with too many 9's for python to keep track of). While the computation time required for such consistently optimal results was much greater than any of the other optimization algorithms applied to this problem, the results were clearly the best.

#### Sensitivity

The genetic algorithm required quite a bit of tweaking in order to settle into finding 1.0 ninety percent of the time. First of all, increasing the population size greatly stabilized the results, giving more consistently optimal x values in earlier generations. However, the population size had the greatest effect on performance. Increasing the maximum generations increased the runtime linearly, but did not always guarantee an eventual convergance. With our population size of 750, a generation count of 40 seemed to be a sufficiently lengthy lifetime for the program. The mutation rate, while not really necessary, did lenghen the amount of time that the program deviated before sticking around one fitness value. Our selection rate was kept low, as we preferred a higher mutation rate as a deviation mechanism for the aforementioned reason. In addition, a lower selection rate helped keep the runtime low and the results focused on better results.

## 1.2 Problem 3.7

Determine using genetic programming, the computer program (s-expression) that produces exactly the outputs presented in Table ?? for each value of x. The following hypotheses are given:

- use only functions with two arguments, i.e. binary trees
- largest depth allowed for each tree: 4
- Function set: {+,*}
- Terminal set: T={0,1,2,3,4,5,x}

Table 1.1: Desired Input Output Value Pairs.

| -10 | -9 | -8 | -7 | -6 | -5 | -4 | -3 | -2 | -1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 153 | 120 | 91 | 66 | 45 | 28 | 15 | 6 | 1 | 0 | 3 | 10 | 21 | 36 | 55 | 76 | 105 | 136 | 171 | 210 | 253 |

### 1.2.1 Algorithm Description

While complex in execution, the approach for solving this problem was simple. An initial set of binary trees $P_O$ was generated, each with one member of the function set (or one nonterminal) as the root, and two members of the terminal set (terminals) as children of the root node. Once the initial population was generated, the following steps were performed in a loop for a fixed number of iterations:

- Evaulate the fitness of each tree
- Sort the trees by fitness
- Replace the bottom 50% using the crossover operation on the top 50%
- Randomly mutate 10% of the new population

While most of the code written was to manage the trees and put together some half-ways readable output, there are a handful of operations worth examining in detail.

#### Evaluation

Evaluation of the fitness of each tree was done by computing the sum of squared differeces between the desired output given in table 1.1, and the actual output when a given x value is supplied to the tree. A fitness of 0 was considered optimal.

#### Crossover

Crossover was done by randomly selecting two parents from the best 50% of the population. One parent was randomly selected to be the base parent, and the other the secondary parent. The child was initially created as a clone of the base parent. Then a random node was selected from both the child and the secondary parent. This could be any node, including the root node of either tree. The selected node in the child tree was then replaced with the node from the secondary parent tree. If the maximum depth would be violated by this exchange, the nonterminal nodes at the maximum depth were replaced with terminals, and their children were deleted.

#### Mutation

Mutation was done by selecting a random node in the tree, deleting it, and randomly regenerating it. The node selected was determined by picking a random number from a uniform distribution between 0 and $N-1$. The selected node could be any node in the tree, including the root node. If a new node was generated as a nonterminal, the children nodes were also randomly generated. It was therefor possible for a tree to be completely destroyed and randomly rebuilt in a mutation. The probability for this being $\frac{1}{N}$ where $N$ was the number of nodes in the tree. In the random node generation, the maximum depth was enforced by requiring that any node generated on the maximum depth level be generated as a terminal.

#### Alternative Mutation

A second theme for mutation was considered, but not implemented. In this alternative scheme, a node would be selected at random in the tree just like in the method above. However, instead of destroying the node and all of its children (if any), the node's type would remain the same and only the value would change. Hence, nonterminals would remain nonterminals, but the operation could change from + to * or vice versa. Likewise, a terminal would simply change to another terminal value. This idea, while valid, was discarded on the grounds that the method stated above would more thoroughly explore the space by allowing for much more radical changes via mutation.

### 1.2.2 Results

Running the genetic program with a population of 100 individuals and 3000 generations consistently gives a tree the represents the equation $2x^2 + 5x + 3$ that showed a total sum of squared differences error of 2. We compared this with using excell to find a polynomial fit to the given points. Excel gave us the equation $y = 2.001x2 + 4.987x + 2.8666$. Given that were are constrianed to using integer values, this appears to be the optimal achievable solution. There are many different trees that represent the same polynomial, and the algorithm may or may not generate the same one every time. The tree often also contains no-ops, like adding 0, or adding something multiplied by zero. An example run is shown in Figure 1.1.

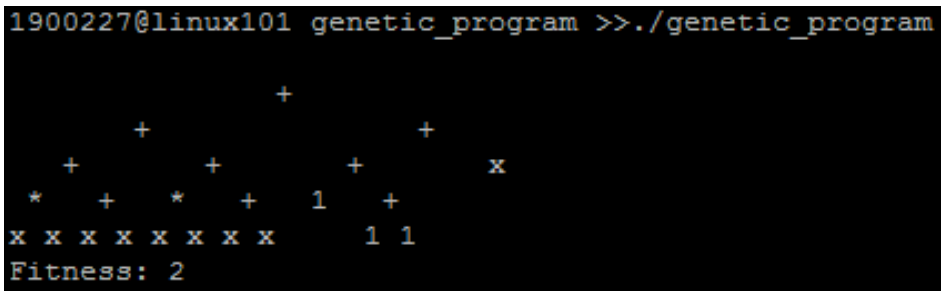


Figure 1.1: A Sample Genetic Program Run

# 2

# Text Chapter 4 - Artificial Neural Networks

## 2.1 Problem 4.17

Apply the MLP network trained with the backpopagation learning algorithm to solve the character recognition task in 4.4.2.

Determine a suitable network architecture and test the resultant network sensitivity to noise in the test data. Test different noise levels from 5-50%.

## 2.2 Network Description

To solve this problem, we implemented a three layer neural network with 120 input nodes, 50 hidden layer nodes, and 8 output nodes. There are 120 input layer nodes so that each pixel has one node. There are 8 output nodes so the network can decide if the test image matches each of the training images independantly. It is entirely possible for the network to make a test image to multiple training images, or none. The hidden layer was more difficult to nail down. We ended up settling on 50 hidden layer nodes because it seemed to minimize the error we were seeing, but as long as there were "enough" hidden layer nodes, the exact number didn't seem to have much impact.

## 2.3 Network Training

The test and training data for the network were a set of 8 "images" represented by arrays of integers that were either 1, filled, or 0 unfilled. The network was trained against each of the training set images 500 times. This wasn't a hard and fast number, but one that seemed to provide a middle ground between untrained and overtrained. Training for fewer cycles than this led to a high chance of misidentification even at low noise levels. Training for more cycles than this led to a high chance of not matching a test image to any of the training images even at low noise levels. In all cases, the network suceeded at matching the test images correct to the training images when no noise was present.

The training images were presented in batches, such that the network would be trained against each image once before moving on to the next set. However, the order in which the images were presented was randomized for each set.

## 2.4 Network Testing

The network was tested against images at noise levels between 0 and 50%. At each noise level 1000 noisy test images were randomly generated from each training image and presented to the network. The results of the testing are shown in Appendix A

## 2.5 Network Performance

The neural net performed very well at low noise levels, but at higher noise levels it rejected most of the images as not belonging to the training set. We theorize that we just didn't find the sweet spot for the number of training epochs

# 3

# Swarms - Text Chapter 5

## 3.1 Problem 5.1

Write a pseudocode for the simple ACO (SACO) algorithm considering pheremone evaporation, implement it computationally, and apply it to solve the TSP instance presented in section 3.10.4. discuss the results obtained. Remove the pheremone evaporation term(equation 5.3), apply the algorithm to the same problem, and discuss the results obtained.

### 3.1.1 Problem Information

The goal of the problem in 5.1 referenced by the text was to find the optimal path through every city on a given grid (the Travelling Salesman Problem discussed in 3.10.4).

### 3.1.2 Pseudocode

```
Initialize t, best, place all ants on city 0
While t < max_iterations
   For each ant
      Calculate probability to move to each city:
         pij = ([pheromones on edge ij^alpha] * [visibility of edge ij^beta])/
             ([sum of all pheromone values^alpha] * [sum of all visibilities^beta]) if j has not
                 been used.
             0 otherwise.
      Normalize p-values
      Use roulette wheel randomization to select a j
   Evaluate ant solutions
   If a better ant solution has been found
      Update best
   Update pheromone values
   Increment t
Print Best
```

### 3.1.3 Algorithm Description

The algorithm was adapted directly from the book's pseudocode - the SACO (Simple Ant Colony Optimization) with pheromone evaporation (at first). On the first iteration, a collection of ants generate random paths through the city graph, with their selections biased towards shorter paths. The ants start at the first city and find paths through each other city. On successive iterations, the ants also take into consideration pheromone

values along candidate edges when selecting cities, which are updated whenever an ant traverses along an edge. With pheromone evaporation used in the algorithm, a percentage of the pheromone is removed before being updated by the ant. This helps prevent the ants from converging onto a local minimum.

### 3.1.4 Algorithm Performance

The SACO algorithm worked fairly well. With the default parameters, the ants consistently found a path shorter than any that were generated through our attempts at random pathing (less than a length of 70), and found a path of less than 60 more than 50% of the time.

Removing the evaporation caused the algorithm to find best paths of between 58 and 60 about 90% of the time. While this is a nice result, and "bad" paths were much less common than with the evaporation on, the lack of deviane means that the occurance of "very good" paths, of length 56 or less, is also less common.

## 3.2 Problem 5.8

Apply the PS algorithm described in 5.4.1 to the max problem of ex 3.3.3. Compare the relative performance of the PS algorithm with that obtained using a standard GA.

### 3.2.1 Problem Information

The goal of the problem in 5.8 referenced by the text was to find the global maximum of the equation $g(x) = 2^{-2(\frac{x-.1}{.9})^2} sin(5\pi x)^6$.

### 3.2.2 Algorithm Description

We implemented a simple particle swarm optimization algorithm. A specified number of particles were initialized at random point between -1 and 1, and then began to move randomly. Using a set of accelerations and velocity restrictions, random changes to the particle velocities were applied in the general direction of the "best" neighbors and the "best" overall particle (as determined by the maximizing function, with the weights towards either metric provided by alpha and beta constants). After a determined number of iterations, the best particle was then printed.

### 3.2.3 Performance

The particle swarm performed well, finding an x with a fitness of .99999 or better every time. However, as with all things in this assignment, the parameters we used traded off precision for performance. So, while the results were more consistently better than any of the hill climbing techniques, it did take slightly longer than hill climbing.

Conversely, while it did not find an x with a fitness resolved at 1.0 nearly every time, as the Genetic Algorithm did, it was faster than the GA and still provided very good, consistent results. In my opinion, the particle swarm solution to this maximizing problem was a good intermediately sized one which provided good results relatively quickly.

# 4

# Immunocomputing - Text Chapter 6

## 4.1 Problem 6.1

Use a bone marrow algorithm to define genes for gene libraries to be used to generate the inital population of a genetic algorithm to solve the TSP presented in CH 3 and CH 5 (figure 6.24).

### 4.1.1 Problem Information

As described, this problem requested that we generate an initial population for a genetic algorithm to use when solving the travelling salesman problem depicted in figure 6.24 in the book. It also elaborated upon the description with the following (paraphrased):

Gene length Lg = 4, number of libraries n = 8, and library length (number of genes in each library) Ll = 4. As one gene from each library will be selected, the total chromosome length is L = Lg x n = 4 x 8 = 32, that corresponds to the number of cities in a tour.

Each gene will be defined as a sequence of four cities known to be part of an optimal route.

A repair algorithm must be used in order to generate permutations of L integers while maintaining most of the genes intact. This is because the TSP problem has the contraint that no city can be visited more than once. to illustrate this problem and one possible way of solving it, consider the example presented in figure 6.25 in the book.

### 4.1.2 Algorithm Description

Bone marrow algorithms are simple techniques used primarily to build antibodies out of gene libraries for artificial immune systems. Generally, they can also be utilized to build genomes out of a collection of random genotypes stored in such a gene library.

Building a list of cities using the bone marrow algorithm was straightforwards. First of all, we began by creating a random set of gene libraries by randomly sampling stretches of cities (of length 4, of course) from a hardcoded optimal route, as specified in the problem. Next, we used these gene libraries to build enough routes to fill a population, taking a random stretch of cities from each gene library per route. After each route was built, however, we applied a repair algorithm in order to fit the constraints of the travelling salesman problem. This involved replacing all repeated cities with randomly ordered cities which were unused before the repairing. In the end, the python script prints out all routes in the initial population. If one were to then implement a GA for the travelling salesman problem, the population could be used to begin selection and breeding.

### 4.1.3 Results

As it was outside of the scope of the problem, we did not evaluate the routes generated by the bone marrow algorithm. However, the variety was apparent while the stretches of pre-built genes were clearly present in the genomes.

# A

# ANN Test Results

Table A.1: ANN Test Results - Noise Level 0%.

| Image | Num Tests | Correct | Incorrect | Correct Multiple | Incorrect Multiple | No Match |
|---|---|---|---|---|---|---|
| Image 0 | 1000 | 1000 | 0 | 0 | 0 | 0 |
| Image 1 | 1000 | 1000 | 0 | 0 | 0 | 0 |
| Image 2 | 1000 | 1000 | 0 | 0 | 0 | 0 |
| Image 3 | 1000 | 1000 | 0 | 0 | 0 | 0 |
| Image 4 | 1000 | 1000 | 0 | 0 | 0 | 0 |
| Image 5 | 1000 | 1000 | 0 | 0 | 0 | 0 |
| Image 6 | 1000 | 1000 | 0 | 0 | 0 | 0 |
| Image 7 | 1000 | 1000 | 0 | 0 | 0 | 0 |

Table A.2: ANN Test Results - Noise Level 5%.

| Image | Num Tests | Correct | Incorrect | Correct Multiple | Incorrect Multiple | No Match |
|---|---|---|---|---|---|---|
| Image 0 | 1000 | 993 | 0 | 0 | 0 | 7 |
| Image 1 | 1000 | 995 | 0 | 0 | 0 | 5 |
| Image 2 | 1000 | 969 | 0 | 0 | 0 | 31 |
| Image 3 | 1000 | 963 | 0 | 0 | 0 | 37 |
| Image 4 | 1000 | 996 | 0 | 0 | 0 | 4 |
| Image 5 | 1000 | 988 | 0 | 0 | 0 | 12 |
| Image 6 | 1000 | 994 | 0 | 0 | 0 | 6 |
| Image 7 | 1000 | 999 | 0 | 0 | 0 | 1 |

Table A.3: ANN Test Results - Noise Level 10%.

| Image | Num Tests | Correct | Incorrect | Correct Multiple | Incorrect Multiple | No Match |
|---|---|---|---|---|---|---|
| Image 0 | 1000 | 915 | 0 | 0 | 0 | 85 |
| Image 1 | 1000 | 915 | 0 | 0 | 0 | 85 |
| Image 2 | 1000 | 800 | 0 | 0 | 0 | 200 |
| Image 3 | 1000 | 776 | 0 | 0 | 0 | 224 |
| Image 4 | 1000 | 926 | 0 | 0 | 0 | 74 |
| Image 5 | 1000 | 874 | 0 | 0 | 0 | 126 |
| Image 6 | 1000 | 906 | 0 | 0 | 0 | 94 |
| Image 7 | 1000 | 947 | 0 | 0 | 0 | 53 |

Table A.4: ANN Test Results - Noise Level 20%.

| Image | Num Tests | Correct | Incorrect | Correct Multiple | Incorrect Multiple | No Match |
|---|---|---|---|---|---|---|
| Image 0 | 1000 | 462 | 1 | 1 | 0 | 538 |
| Image 1 | 1000 | 326 | 2 | 0 | 0 | 672 |
| Image 2 | 1000 | 264 | 2 | 0 | 0 | 734 |
| Image 3 | 1000 | 233 | 2 | 0 | 0 | 765 |
| Image 4 | 1000 | 535 | 1 | 0 | 0 | 464 |
| Image 5 | 1000 | 438 | 0 | 0 | 0 | 562 |
| Image 6 | 1000 | 488 | 2 | 1 | 0 | 511 |
| Image 7 | 1000 | 575 | 0 | 0 | 0 | 425 |

Table A.5: ANN Test Results - Noise Level 30%.

| Image | Num Tests | Correct | Incorrect | Correct Multiple | Incorrect Multiple | No Match |
|---|---|---|---|---|---|---|
| Image 0 | 1000 | 103 | 8 | 0 | 0 | 889 |
| Image 1 | 1000 | 27 | 6 | 0 | 0 | 967 |
| Image 2 | 1000 | 64 | 8 | 0 | 0 | 928 |
| Image 3 | 1000 | 54 | 11 | 0 | 0 | 935 |
| Image 4 | 1000 | 151 | 3 | 0 | 0 | 846 |
| Image 5 | 1000 | 134 | 4 | 0 | 0 | 862 |
| Image 6 | 1000 | 180 | 10 | 0 | 0 | 810 |
| Image 7 | 1000 | 239 | 4 | 0 | 0 | 757 |

Table A.6: ANN Test Results - Noise Level 40%.

| Image | Num Tests | Correct | Incorrect | Correct Multiple | Incorrect Multiple | No Match |
|---|---|---|---|---|---|---|
| Image 0 | 1000 | 19 | 10 | 0 | 0 | 971 |
| Image 1 | 1000 | 3 | 18 | 0 | 0 | 979 |
| Image 2 | 1000 | 3 | 15 | 0 | 0 | 982 |
| Image 3 | 1000 | 5 | 19 | 0 | 0 | 976 |
| Image 4 | 1000 | 28 | 19 | 1 | 0 | 954 |
| Image 5 | 1000 | 33 | 4 | 0 | 0 | 963 |
| Image 6 | 1000 | 43 | 11 | 0 | 0 | 946 |
| Image 7 | 1000 | 49 | 5 | 0 | 0 | 946 |

Table A.7: ANN Test Results - Noise Level 50%.

| Image | Num Tests | Correct | Incorrect | Correct Multiple | Incorrect Multiple | No Match |
|---|---|---|---|---|---|---|
| Image 0 | 1000 | 1 | 25 | 0 | 1 | 974 |
| Image 1 | 1000 | 0 | 23 | 0 | 0 | 977 |
| Image 2 | 1000 | 0 | 32 | 0 | 0 | 968 |
| Image 3 | 1000 | 0 | 17 | 0 | 0 | 983 |
| Image 4 | 1000 | 0 | 17 | 0 | 0 | 983 |
| Image 5 | 1000 | 3 | 18 | 0 | 0 | 979 |
| Image 6 | 1000 | 7 | 19 | 0 | 0 | 974 |
| Image 7 | 1000 | 5 | 13 | 0 | 0 | 982 |