## Finding Weights for a Neural Network

### Introduction

I chose to implement my own neural network and use it with the Pokémon classification dataset I created for previous assignment. The neural network tries to use a Pokémon’s type (one of nine discrete values), height, and weight to classify an entry as being one of 34 Pokémon. There were three entries in the training data for each Pokémon. The neural network I used had 11 input nodes (one for each of the possible types plus one for height and one for weight), a single hidden layer with 22 nodes, and 34 output nodes. This is the same structure I used in the previous assignment. The output layer was normalized. The heuristic I used to quantify how well the neural network performed was the sum of all the values for the correct output node over all the training data. For example, if I had three training data whose labels were Pikachu, Bulbasaur, and Charmander and the neural network’s output nodes for those Pokémon were .5, .9, and .4 respectively, then the heuristic’s value would be 1.8.

### Hill Climbing

I chose to perform hill climbing on each of the weights and biases in sequential order. There were a total of 1046 weights and biases in the neural network. My algorithm starts with the first weight and finds which value maximizes the heuristic described in the previous section. It uses a slight modified hill climbing to achieve this. It then moves on to the next weight. It continues looping through each of the weights until it makes it through all the weights without being able to increase the value of the heuristic. At this point, the algorithm has reached a local maximum and it quits. Since weights are continuous, I made a slight modification to how the hill climbing algorithm works. The algorithm starts out by taking steps of size 1. Once it gets to a point where increase or decrease the weight by 1 lowers the heuristic it stops. Then, starting from that point, it repeats the process with steps of size .1. It repeats this with step size of .01 and .001. I didn’t think smaller values would matter so I stopped there.

This method resulted in the neural network getting stuck at a local maximum in 100% of tests I ran. When I trained a neural network on the data in the previous assignment, it was able to achieve ~90% accuracy on the test set. Using the hill climbing method, the neural network achieved, on average, 14.7% accuracy. The highest accuracy I observed was 21/102 Pokémon correctly classified (since there were three instances of each Pokémon in the training data this meant that it could correctly classify 7 out of the 34 Pokémon). It was also an slow way to train the neural network. Since I wrote my own neural network code, and because each step in the hill climbing required all 102 training data to be run through the neural net 3 times (one to get the value of the heuristic function for the step below the current value, the step above the current value, and the current value itself), the algorithm usually took more than half a minute to complete on my computer. Weka’s back propagation, when finding weights for the same neural network structure on the same data, got >85% accuracy in around 7 seconds.

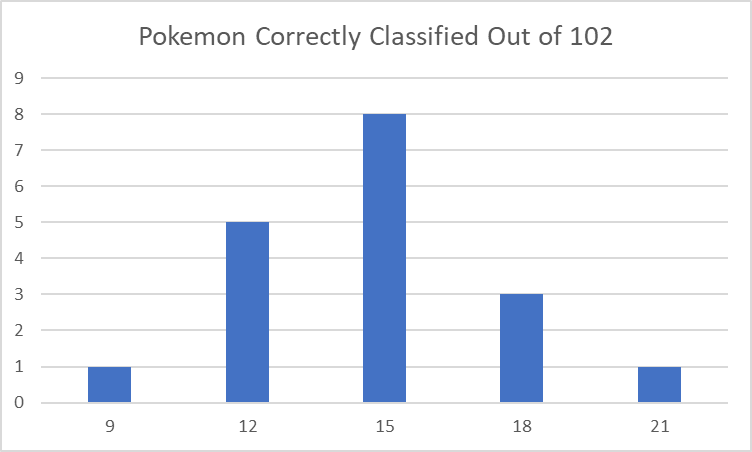


Figure 1: Histogram of the number of Pokémon that were correctly classified by my neural network after hill climbing optimization was completed.

I suspect that the way I chose to implement hill climbing contributed to the poor results. Since my algorithm hill climbs each weight one-by-one, I think that it may lead to the neural network strongly favoring a few outputs while ignoring others. I also think that my heuristic contributed to the issue. Since the output is a measure of how confident the neural network is that a given input corresponds to a specific output, it might have to lower its confidence in a given output in order to model the data more accurately. This means that there are many local maximums and so the hill climbing algorithm is likely to get stuck.

### Simulated Annealing

I had some difficulty deciding how to apply simulated annealing to the problem of optimizing the neural network. I decided to, for each iteration, select a random weight or bias in the neural network. Then add a random number sampled for the normal distribution to this weight. The neural network generated by this addition constitutes the new sample. From there I just used the standard simulated annealing algorithm. I checked the output of the heuristic function for both the new and old neural network. If the heuristic for the new neural network was higher, the algorithm jumps to that sample. Otherwise, it probabilistically jumps to the new sample based on the temperature. Once the temperature gets to a very low value, the algorithm stops.

While simulated annealing performed better than hill climbing, it still didn’t do very well. My test of the algorithm took 627 seconds to complete and only correctly classified 36 out of 102 Pokémon. This is still much slower and much less accurate than Weka’s back propagation. I suspect that the dimensionality of the problem overcame simulated annealing’s ability to get out of local maxima. Since there are over a thousand weights to tune, there are probably too many local maxima for simulated annealing.

### Genetic Algorithm

The way I chose to implement the genetic algorithm optimizer for finding neural network weights was as follows. Firstly, I initialize 1000 neural networks with random values. The random initial weights for these neural networks were much larger than the random weights I assigned for the hill climbing and simulated annealing neural networks. I did this to allow for more ‘genetic’ variation in my population. The next step was to evaluate each neural network’s fitness. I used the same fitness function as I used for the previous two randomized optimization methods. Then I ordered them by fitness. The least fit 500 were culled and were replaced by the offspring the most fit 500. This was done by ‘breeding’ the first and second most fit then the third and fourth most fit and so on. Each pair created two offspring. Each of the two offspring had a 50% chance of inheriting each neural network layer from either parent. Put another way, the son of two neural networks had a 50% chance of inheriting his father’s hidden layer or his mother’s hidden layer and the same was true for the output layer. The daughter had a separate 50% chance for each layer. This was a slightly problematic way of implementing things because there is a chance that both the son and daughter could be exactly the same as the father or exactly the same as the mother. Then some genetic diversity would be lost. However, given enough individuals, I didn’t think this would matter. After creating the two children, I added a random number (sample from the gaussian distribution) to each weight and bias to increase genetic diversity. I had this process repeat 10,000 times.

This method works slightly better than simulated annealing. The fitness of the most fit individual increased quickly at first but once it was correctly classifying around 24 of the 102 Pokémon, the improvement slowed down rapidly. The best it was able to do was correctly classify around 40 out of 102 Pokémon. One problem was that genetic diversity quickly collapsed. The difference between the best fitness and the average fitness shrunk quickly. I suspect that the way I implemented the genetic algorithm had something to do with this. But I also think that this is inevitable. The hypothesis space is too massive and, given enough time, a population will tend to get stuck in one part of it. Another problem with this approach was that it was the slowest method by far. Perhaps 1000 individuals were too many. Each generation took roughly a second to complete on my machine. This means that the 10,000 generations required around 3 hours to complete. For the results I ended up getting, this is an atrociously long time. For comparison, Weka’s backpropagation took roughly 7 seconds.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Algorithm | Hill Climbing | Simulated Annealing | Genetic | Backprop | Random Search |
| Correctly classified | 21 | 36 | 44 | 90 | 20 |
| Time to complete | 30-120 seconds | 627 seconds | 3 hours | 7.7 seconds | 146 |

Figure 2: Accuracy and completion time of all methods. Out of the randomized search algorithms genetic had the highest accuracy while simulated annealing was the best in terms of accuracy/time tradeoff. Hill climbing performed only a little better than completely random search. Backpropagation was, by far, the best. Though Weka probably used other tricks to improve performance so it’s not an apples to apples comparison.

## Optimization Problem #1: ‘To be or not to be? That is the question.’

The first optimization problem is designed to highlight the advantages of my genetic algorithm. The problem is copied from ‘The Coding Train’ YouTube channel. The problem is simple: the goal is to generate the string ‘To be or not to be? That is the question.’ The heuristic function takes in an array of 41 bytes, converts those bytes into characters, and returns the number of characters that match the target string. This problem is ideal for genetic algorithms because it can be solved piece-by-piece. If one individual has the first letter correct, and other individual has the second letter correct, then one of their offspring are likely to have both letters correct. This kind of problem is interesting because it has such a ‘lumpy’ heuristic. A letter is either correct or it isn’t. There is no way to tell that you’re getting closer to a letter being correct. This means that hill climbing gets stuck. Simulated annealing and even MIMIC have a hard time solving this problem.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithm | Hill-Climbing | Genetic Algorithm | Simulated Annealing | MIMIC |
| Average letters correct out of 41 | 0.6 | 41 | 41 | 41 |
| Average completion time (seconds) | .006 seconds | .198 seconds | 33.538 seconds | 85.39 seconds |
| Average number of iterations | 1.4 | 138.6 | 5,298,314 | 1007.8 |

Figure 3: A comparison of the performance of different optimization methods for the ‘To be or not to be? That is the question’ problem.

Based on the data, my hypothesis that the genetic algorithm would perform the best was completely accurate. The reason for this, as stated above, is that individuals solved parts of the problem and their offspring benefitted for their parents’ partial solutions. My genetic algorithm was fairly simple. First create 1000 individuals and initialize them all with string of random bytes. Order them by fitness. The least fit 500 are removed each generation. The first and second most fit create two offspring, he third and fourth most fit create two offspring, and so on. To create the offspring, I first created a clone of the ‘father’ and ‘mother’ individuals, we’ll call these offspring ‘son’ and ‘daughter’. Then, for each byte in the array, there is a 50% chance that the ‘son’ and ‘daughter’ have that specific byte swapped. Then, for each byte, there is a 10% chance that it is replaced with a random byte.

Hill climbing was, by far, the worst performing algorithm on this problem. This is because of the ‘lumpy’ heuristic. For any byte that is more than two bits off from the correct value, hill climbing will not be able to find the correct value. The reason for this is that I defined a neighbor as any string of bytes that differs by exactly one bit. To improve the hill climbing algorithm, I could try to expand what is defined as a neighbor. However, this would only go so far. Hill climbing is simply unsuited to this problem.

Simulated annealing did better than I excepted. I thought that since there aren’t really hills, more like steppes, it would perform poorly. However, given a long enough cooling duration, it almost always solved the problem. The main issue with simulated annealing was how long it took. I suspect, given enough time, I could fine-tune it and lower the average completion time by five seconds or so. Even if I did that, it would still be orders of magnitude slower than the genetic algorithm. I don’t think there is any way around this. Simulated annealing is slow on purpose.

MIMIC was a different story entirely. I coded the algorithm myself so that might have something to do with its poor performance, but I suspect that MIMIC is just bad at solving this problem. The first issue I ran into was that the probability distribution tended to collapse before it found a solution. To prevent this, I prevented the probability distribution from every collapsing completely. That is to say, the probability of x given y is never more than 98.5%. This improved things but then a second issue cropped up. Once MIMIC has mostly solved the problem (e.g. it has 35 out of 41 letters correct), it is vastly more likely to sample individuals from the distribution with a lower fitness than the individual that went into creating the distribution. To combat this, I created each new distribution using only the most fit members. If there is one individual with a fitness of 40 and the rest have lower fitness scores, then the only individual used to create the new probability distribution is that one. With these changes MIMIC was able to solve the problem, but slowly.

## Optimization Problem #2: Bumpy Hill

The second optimization problem is meant to show the strength of simulated annealing. Since simulated annealing is designed to work in situations where there are many local optima, I decided to create a one-dimensional optimization problem with many local optima. The heuristic function takes in an array of four bytes, converts that into a floating-point number, runs that number through a function of many overlapping sines and cosines, and scales the value before returning it. Since the function has essentially infinite optima (since the peak of each sine or cosine is an optima), I felt that simulated annealing would perform much better on this than other methods.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithm | Hill-Climbing | Genetic Algorithm | Simulated Annealing | MIMIC |
| Average final fitness | 4983 | 6235.2 | 6528.8 | 6476.6 |
| Average completion time (seconds) | .0025 seconds | .1111 seconds | 27.268 seconds | .3376 seconds |
| Average number of iterations | 4.6 | 127.2 | 3912021 | 41 |

Figure 4: A comparison of the performance of different optimization methods for the ‘Bumpy Hill’ problem.

What is interesting here is that this wasn’t the huge win for simulated annealing that I though it would be. While it performed the best from a results perspective, it came at a huge time cost. I suspect that if I ran the genetic algorithm or MIMIC 10 times and took the best result, either would perform better than simulated annealing. I think that this might have to do with how I’m defining neighbors. Since the input to the heuristic function is essentially a fixed-point number, and since a neighbor is any value that differs by one bit (which depending on the bit could be a very different number), the simulated annealing function might be jumping around too much to work. My implementation of simulated annealing could be improved by limiting the neighbors it considers to neighbors that are numerically close not just close in a bit sense.

One interesting thing to point out about MIMIC is that the probability distribution didn’t necessarily collapse on a good heuristic value. However, MIMIC tended to find byte strings with high fitness along the way. I chose to save the highest fitness it found and count that instead of the fitness it ended on. I think that the reason it didn’t necessarily collapse on high fitness byte string is that changing just one bit can have a drastic impact on the fitness of the byte string. I think the high fitness byte strings got lost in the resampling process.

## Optimization Problem #2: US Map Coloring

The final optimization problem highlights the strengths of MIMIC. I didn’t have a strong intuition as to what kinds of problems MIMIC would be good at, so I went with one from the paper. I chose to do the graph coloring. Specifically, the graph that was being colored was a map of the United States. The fitness function takes in 12 bytes of data. Each byte represents the colors of four states. This means that each state gets two bits (or four possible colors) to represent it. For each instance where two states that touch have the same color, the fitness function subtracts one from the final fitness. This problem is good fit for MIMIC because a probability distribution is a good way of representing the kind of logic a person would use to color the map (e.g. ‘If Washington is red, then Oregon should be blue’).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithm | Hill-Climbing | Genetic Algorithm | Simulated Annealing | MIMIC |
| Average final fitness | -3.6 | 0 | 0 | 0 |
| Average completion time (seconds) | .006 seconds | .059 seconds | 3.645 seconds | .663 seconds |
| Average number of iterations | 13.2 | 98.6 | 621457 | 21.8 |

Figure 5: A comparison of the performance of different optimization methods for the ‘US Map Coloring’ problem.

I expected MIMIC to outperform the other methods by more than it did. Based on the data, I can’t say whether the genetic algorithm or MIMIC was superior. If the problem had a costlier fitness evaluation function, MIMIC would have been the hands-down winner. However, due to the time cost of each MIMIC generation, the genetic algorithm achieved similar results in less time. I suspect that, with some tuning, I could have gotten MIMIC to run faster. For example, I’m using 1000 samples, which is probably more than I need. Also, since I implemented MIMIC myself (and not in an optimal way), it could probably be much faster.

Hill climbing, once again, was the worst performing algorithm. It consistently got stuck in local optima. I was surprised at the efficacy of the genetic algorithm. The power of a genetic algorithm is its ability to combine partial solutions to problems. However, two partial solutions to the graph coloring problem won’t necessarily go together. There were a handful of instances where the genetic algorithm had trouble converging on a solution, but for the most part it solved the problem quickly. Simulated annealing, as always, seems to be a reliable, but slow, way to solve the problem. Since hill climbing tended to get close to solving the problem, it’s no surprise that simulated annealing was enough to get it the rest of the way.