Report 2: Genetic Algorithms

Eric Scott, supervised by Roy Villafañe, Andrews University

04 April, 2009

For CPTR 495, Independent Stmudy in Computational Intelligence

1 Beyond Gradient Descent

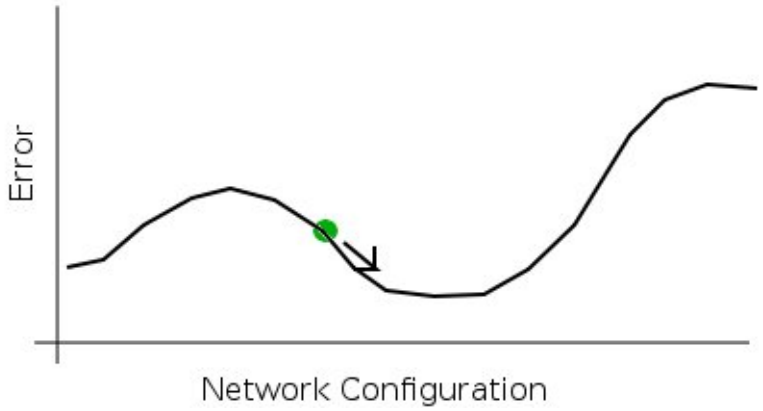
Computational Intelligence is largely concerned with function optimization. Not only are CI tools useful for modelling and optimizing complex nonlinear problems in science and industry, but the task of, say, training an Artificial Neural Network (ANN) is itself an optimization problem. The possible solutions to an optimization problem exist in a huge, sometimes infinite manifold (a.k.a. solution space, problem space, fitness landscape, energy landscape) that is impossible to search exhaustively or to solve analytically.

It's worth noting that important analogies exist here to thermodynamics and the tendency of a system to settle into a given state. Statistical mechanics has a well-established framework that can save us the trouble of re-visualizing the problems at hand as we please for our specific application, and Boltzmann is a name that occurrs repeatedly in the study of adaptive systems. Just like a system of water molecules at a low temperature forms a crystal by settling into a low energy, high-entropy state, and chemical reactions in the presence of a catalyst are assisted to surmount a potential energy barrier, entropy, potential wells, energy barriers, etc are all terms that have very real meaning in analyzing an endless array of dynamical systems and their behaviors. CI tools work by definition very much like natural systems – in a complex, nonlinear, and statistical way.

Artificial Intelligence in general is, then, concerned with finding ways of efficiently plotting a search path through the space of possibility so that a high quality solution is found in a minimal amount of time. A globally

optimal solution is usually not possible, but heuristic algorithms and various stochastic processes have been developed to try and make the task of finding a fairly good local optimum tractable. In the modelling of nonlinear systems, for example, often the only way to find out what the system will do is to run a simulation of it or to observe it in the natural world. At this point CI tools are useful for developing a sort of digital intuition regarding how the system works and how we can control it or optimize it – i.e. to develop heuristics that make robust predictions out of the otherwise infinite landscape of possible explanations. We can then design a CI system that evolves and learns in a stochastic manner, taking advantage of what we know to set up a framework that, as it stochastically converges into a low energy state, settles upon a viable solution to the problem.

Gradient descent is one example of an optimization paradigm, and forms the basis of training an ANN via backpropogation (As discussed in my previous report). Also known as "hill climbing," this greedy approach simply takes the most obvious local step towards a more optimal solution on the manifold. In continuous systems this is determined via calculus (the "gradient"), but it can be applied to discrete systems as well simply by comparing all of the solutions in the immediate neighborhood and moving to the best one. The idea is that if we go in that direction, there will be an even better solution the other side of the one we choose, and so on, until we reach a peak or valley (Recall that we may define either low or high points in the landscape to be optimal, depending if we are visualizing the manifold as "fitness" or "error" – optimization is either minimizing or maximizing a function).



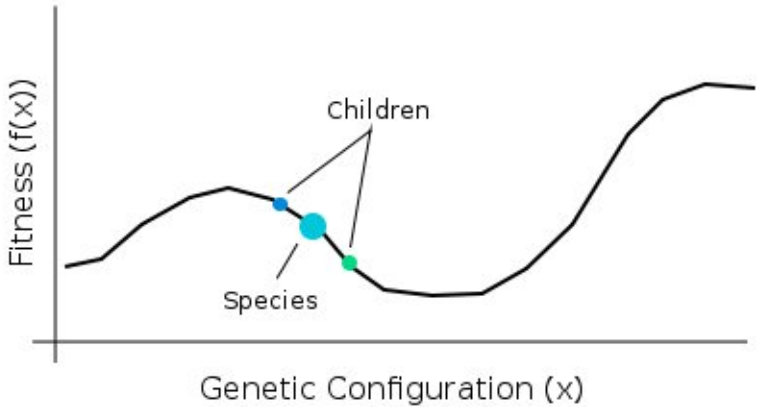
The long-term behavior of the system results in convergence on locally optimal solutions. Much like chemical reactions without catalysts, however,

more optimal solutions (lower energy states) are likely to exist on the other side of a ridge. Thus it is important not only that we converge on local solutions by sliding into potential wells ("basins of attraction"), but that a significant diversity of the landscape is explored.

Simulated annealing, then, is an algorithm that extends gradient descent by imitating thermodynamic processes in cooling metals. By "heating up" the system, i.e. dislocating and "shaking" the vectors that define the path of our search, we allow the system to "jump" into areas that are sub-optimal (Which we normally want to avoid), at which point we might be lucky enough to end up drawn into the potential well of another, more optimal solution than we at first found. In this way we explore more of the manifold, while still being drawn towards the local optima in each explored section between "shakes."

2 Genetic Algorithms

Another alternative to a vanilla greedy method is to, instead of having a "ball" that rolls around, define a "parent" who spawns children with slightly different configurations. Candidates with higher fitness (Lower error) are more likely to "survive" to the next generation. This part of the process is known as a "beam search," and as one of my textbooks puts it, "the effect is that the [better] state says to the others, 'Come over here, the grass is greener!'"[1]



[1]Stuart Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. (2003), 116.

The analogies to evolution are already apparent. We round off the model by incorporating the concepts of "mutation" and "crossover" (sexual selection), and are left with one of the most powerful information-generating processes artificial intelligence has at its disposal: Genetic Algorithms (GAs).

In general, the field of Evolutionary Computation (EC, which we will consider a subfield of CI) is composed of several paradigms which developed somewhat independently in the 60's and 70's. The trend has been towards unification since the 80's, but the kinks are still being worked out as to how to unify them all into one coherent unit. If nothing else we should still hold there to be precise definitions that distinguish between Genetic Algorithms, Evolutionary Programming (Which includes only mutation, not crossover), Evolution Strategies, and Genetic Programming (Which focuses on evolving *programs* built out of actual computer code). Partical Swarm Optimization is also sometimes considered an evolutionary paradigm, though I will leave it until a future report which will cover swarm and ant colony alogorithms.

2.1 Selection

We have noted that while it is important for a stochastic optimization algorithm to converge on effective solutions, being drawn too greedily towards basins of attraction yeilds poor results. Thus it is important to explore a large variety of locations in the landscape. This forms the core of the concept of *selective pressure*: too much pressure leads to premature solutions that are sort of quick and dirty, while too little may cause the system to drift aimlessly over the landscape with little direction or success.

Determining the parameters or method by which selection takes place in a genetic algorithm is what determines its success. How many individuals should be in our population? How often should mutation occur? How do we choose how likely an individual is to survive? These questions as of yet largely fall to the intuition of the CI specialist, and must be learned as an art with regard to the specific problems at hand. Every class of problem has a different characteristic topography to its fitness landscape, and what works for one application may not be effective in another scenario. The question of how to optimize these parameters forms the domain of *Evolutionary Strategies*, which considers the problem of not only evolving the solutions but also of evolving the evolutionary parameters themselves.

Several mechanism for implementing selection have been developed. Engelbrecht lists, among others, random selection (Which is about as useless as

it sounds), proportional selection (Where the probability an individual survives is proportional to its fitness), tournament selection (Which selects the best candidates from a random subset of the population), rank-based selection (Which sorts individuals by fitness and then selects from each rank with a predefined probability), Boltzmann selection (Which is based on simulated annealing), and elitism (The most fit individuals always survive – This has a rather high selective pressure). In my code I've implemented a version of proportional selection known as "roulette wheel selection."

2.2 Gray Code

Random mutation via bit-flipping suffers a drawback when we realize that a small change in a bitstring does not generally correspond to a small change in the value it represents, i.e. binary has a large and variable *Hamming distance*. For example, you must flip four bits to transform a binary seven (0111) into a binary eight (1000). If we want to ascend smoothly up the potential well of an evolutionary maximum, however, we do not want it to be difficult for small mutations to occur.

A solution is provided by Gray coding, which provides a mapping of binary numbers to encoded versions with a Hamming distance of 1. Eberhart and Shi provide the following algorithm for defining Gray code:

$$G_i = XOR(B_i, B_{i-1}) \tag{1}$$

Where G_i represents the i^{th} bit from the left of the Gray code, and B_i is the i^{th} bit of the binary code, i starts at 2, and $G_1 = B_1$ (Or, if you like, i starts at 1 and $B_0 = 0$). For example, the bitstring $\vec{B} = 01101011$ can be transformed as follows.

$$\begin{array}{rr} & 01101011 \\ \text{XOR} & 00110101 \\ \hline & 01011110 \end{array}$$

Clearly this operation is equivalent to $\vec{G} = XOR(\vec{B}, (>> \vec{B}))$, where $>>$ is the binary operater shift-right (i.e. divide by 2). This makes it straightforward and efficient to implement in code as a one-line function.

A more difficult problem is the conversion of Gray code back into binary. A moment's consideration shows that no simple shift operation will provide the inverse function. After getting on Wikipedia and reading about how

Frank Gray originally visualized the definition of the code that bears his name (He called it "reflected binary code"), I was able to intuit that an accurate algorithm is:

$$G_n = B_n \tag{2}$$
$$B_i = XOR(B_{i+1}, G_i) \tag{3}$$

Where n is the length of the bitstrings. Using this fact we can proceed from right to left along $\vec{G}$ bit by bit until $\vec{B}$ has been fully constructed. I have not proved this algorithm on paper, but I trust it intuitively and it has worked in every test case.

3 Code Status

A GA framework has been implemented and briefly tested on several toy problems, including the training of a neural network. I would have liked to produce graphs showing the performance difference based on different parameters, but between the simplicity of the problems tried and time constraints it didn't get done. I also tried to multithread the system, anticipating future applications on systems with multiple cores, but bugs and performance issues have kept me constrained to a single thread as of yet.