To the Graduate Council:

I am submitting herewith a thesis written by Isaac Sherman entitled "On the Role of Genetic Algorithms in the Pattern Recognition Task of Classification." I have examined the final paper copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Bruce MacLennan, Major Professor

We have read this thesis
and recommend its acceptance:

Dr. Bruce MacLennan

Dr. Hairong Qi

Dr. Catherine Schuman

Accepted for the Council:

Dixie L. Thompson

Vice Provost and Dean of the Graduate School

On the Role of Genetic Algorithms in the Pattern Recognition Task of Classification

A Thesis Presented for

The Master of Science

Degree

The University of Tennessee, Knoxville

Isaac Sherman

May 2017

dedication...

Acknowledgements

I would like to thank...

Some quotation...

Abstract

Abstract text goes here...

Contents

List of Tables

List of Figures

Chapter 1

Introduction

April 12 deadline- two passes (a week each) gives us roughtly last week of march for rough draft. Look up correlation coefficient //

A Genetic Algorithm (GA) is a biologically inspired form of computing. GAs can be used for many different purposes, from optimization to classification to design and testing. They can be applied anywhere that a solution can be encoded and then autonomously evaluated. However, they are most well understood as a general solution to optimization as a form of stochastic gradient descent, similar conceptually to simulated annealing. In this paper, I attempt to narrow the role of GAs as they pertain to pattern recognition and classification. In the remainder of this chapter I will describe GAs in general terms and discuss the motivations of this paper. In chapter 2, I will discuss the specifics of the GAs which I use in this paper in detail. In chapter 3 I will describe the experiments conducted and discuss the results. In Chapter 4 I will describe future avenues of study which are available.

Genetic Algorithms Introduction Imagine a colony of rabbits. The rabbits are quite content to munch on clovers and thistles and the like. Some rabbits are much more content than others, and have significant girth. One day foxes find the rabbits. There are many more rabbits than foxes, and the foxes can't eat all of them. The heaviest rabbits are both the slowest and the most appetizing to the foxes, and they are the first to go, but many die. They have failed the evolutionary filtering process. However, the rabbits that survive are thinner, and the most successful are likely faster. These are the rabbits which survive to populate the next generation.

GAs encapsulate this process, though usually with much less mayhem. The algorithm is as follows:

```
Population := RandomInitialization()
        while True:
                For each Solution in Population:
                        Evaluate Solution
                        Assign Fitness to Solution
                newPopulation = SurviveAndBreed(Population)
                Population = newPopulation
        end while
```

Stopping Conditions This algorithm can run for a predetermined number of generations, indefinitely, or until another specific criteria is met. Consider the problem of finding a way of combining 4 operands with 3 operators to achieve a particular value. There are often many ways to solve this problem. If one was to use a GA to solve the equation, the algorithm could stop as soon as it had a valid solution values for a, b, c, d, and the 3 operators which satisfied the equation. For example:

$$a\ op_1\ b\ op_2\ c\ op_3\ d = x$$

$$3 \times 7 \times 3 + 5 = 68$$

$$9 \times 7 + 8 - 3 = 68$$

Solution Encodings Now lets look specifically at what is meant by a solution. First, GAs usually have some encoding based ultimately on a string of 0s and 1s, called a bitstring[1]. Continuing with our example, we know that there are 4 operands which have a value between 0 and 9, or 10 values total. To encode that in a bitstring we use the values 0000-1010. We could also include some error correcting code to randomly reassign the bits if they go outside of the values, ensuring that any solution randomly generated contains valid data (we could also use more bits and/or arbitrarily assign values via modulo arithmetic, but this is the most instructive method for our current purposes). Next, we have the operators, which can be $+$, $-$, $*$, $/$. These fit nicely within the 00-11 bit range, with no need of error correction. So we have a bitstring with the form xxxx-xx-xxxx-xx-xxxx-xx-xxxx[2], 22 bits which satisfy the constraints of our problem. The second line of the above example would be 0011-10-0111-10-0011-00-0101.

Fitness Functions Following along with the algorithm, we need to assign fitness. In this case the closer one is to the solution, the better, usually [3]. There is research showing that a fitness function should be differentiable, at least as far as the solution space. Point discontinuities aren't an issue if they occur outside of the solution space, which is important for us because we're going to make use of one. Specifically, the function

$$F(s) = \frac{1}{|E(s) - x|}$$

Where E(s) is the result returned by evaluating s, the numeric string. So if we had the string 3-6*6+2, then E(s)= -16. Assuming x is still 68, then the fitness for that particular s would be $\frac{68}{84} = 1.19E-2$, extremely low. When E(s) = x or F(s) = ∞ we break out of the infinite loop. This would be a discontinuity, but the function is still differentiable across where we're evaluating it.

[1]Other genetic alphabets are possible as well, though less common.

[2]We use the hyphens here only for readability- the bitstrings contain only 1s and 0s.

[3]There are examples of x where this won't work as well- in a shortened version of our example problem, for instance, if the target is 25, $2 \times 3 \times 4$ will quickly dominate the fitness landscape but isn't actually any closer to a valid solution than $5 \times 5 + 9$.

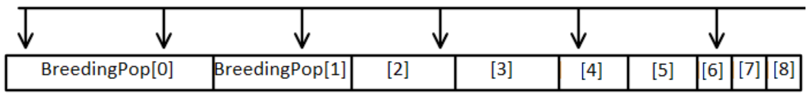

Figure Figure 1.1: An example of RUS. The bar represents the cumulative fitness of the population. The arrows represent which members of the population go on to become a breeding member. In this example, 3 gets skipped even though it has higher fitness than 4 and 6, while 0 gets copied into the breeding population twice.

Breeding and Survival Now that we have a fitness function, we can evaluate the entire population and determine who has the highest fitness. With a random seed, the first generation is usually not very fit. Regardless, the next step is to see who survives and who breeds into the next generation. Survival is usually an arbitrary matter of copying the best solution(s) *in toto* to the next generation. [4] This is typically referred to as elitism, and it is usually a percentage around 10% of the population that is uncritically copied into the next generation. Properly done, this guarantees monotonically non-decreasing fitness of the best solution from one generation to the next.

The next step is to pick some fraction of the population as breeding stock. The most fit are generally given preferential treatment, but not exclusive preference. This is in large part because in GAs as in life, genetic diversity is a critical trait to the overall fitness of a population. For GAs, it means that diversity speeds convergence to ensure that even the less fit have a chance to propagate into the next generations. The method we employ, a fairly standard one, is Roulette Uniform Selection. To get an intuition for the algorithm, see Figure 1.5. In it, you can see that RUS chooses markers (represented by the arrows) equally spaced between the beginning of the population and the end. Everywhere a marker falls means a copy of that solution gets passed into the breeding population for the next generation. This is usually guaranteed to get at least 1 copy of the most fit individual, but everything else is based on chance.

[4]Some variations have ages, where all solutions will live a certain amount of time, and more fit solutions have longer lifespans than less fit solutions. These will not be covered further in this document.

With a breeding population in place, we can begin the breeding process. This is typically accomplished via an operation called crossover, which I will explain below.

A	*1001*	*10*	*0011*	*11*	*1001*	*00*	*0111*
B	**0011**	**01**	**1000**	**01**	**0001**	**10**	**1001**

We begin with two solutions. Crossover takes and returns 2 bitstrings. It also has several subtypes, which I will illustrate in sequence. First is one-point crossover. This means that before crossing, a crossover point is chosen and offspring are produced as a copy, and when this point is reached, the offspring cross over and begin taking material from the other parent.

A'	*1001*	_1_**1**	**1000**	**01**	**0001**	**10**	**1001**
B'	**0011**	**0**_0_	*0011*	*11*	*1001*	*00*	*0111*

Two point crossover is similar to one point, except that crossing happens twice.

A'	*1001*	*10*	_00_**00**	**01**	**0001**	**10**	**1**_111_
B'	**0011**	**01**	**10**_11_	*11*	*1001*	*00*	_0_**001**

Notice that here the final digit of A' and the second digit of B' becomes invalid (1111 is not between 0000 and 1001)- this is not controlled for in Crossover, but rather handled later by evaluating the offspring.

The final standard type is uniform crossover, which makes a check at each bit to crossover. It results in much more mixing, depending on the probability of a crossover event. One advantage is that it treats the beginning and end as the same, which can't be said for either of the first two. Another is that it allows the designer to specify, directly and exactly, how much gene mixing should occur on average.

One thing that might be noticed is that regardless of where the crossover point is, with these methods if both A and B contain a particular bit in the same location, it will appear in both offspring. This is one reason that diversity is important- if the population becomes too homogeneous, it will be unable to change except through random mutation, which we discuss shortly.

One other form of crossover we will deal with is one that counters this potential vulnerability. It is a shifted crossover, so that one bitstring shifts forward a random number of bits, and then crossover proceeds normally.
Other forms of crossover are possible, though they are not as widely represented in the literature. One is a modified uniform crossover that checks to cross only at each ”word”, that is at each column representing a digit or operator in our example. This gives the designer some control over how much contiguous information is exchanged per crossover. Other variations with three or more parents, or even random asexual reproduction are possible. Logical operations are viable, though again care must be taken to not increase homogeneity overmuch.

Mutation Mutation is fairly straightforward. Usually after crossover and before insertion in the next generation, that is, only affecting the offspring of breeding and not elitism, each bit in the bitstring has a chance to change. The algorithm is perhaps the most instructive:

```
for (Solution s in Crossover(A,B)):
        For bit in s.Bits:
                if(MutationChance > Random(0,1)) bit = !bit
        end for
end for
```

There are some implementations that vary in that they will change random values to 1 or 0 rather than flipping bits (in other words, values will change about half as often). A standard value for mutation is small, about $\frac{1.0}{Solution.Length}$ which means on average 1 bit will change per solution per generation.
What is less straightforward are the effects of mutation over a population. While values of 0 often stagnate in local minima, an upper correlate doesn't seem to exist-one could set mutation high, say 25%, and turn off crossover entirely, and proceed

with elitism and mutation alone and arrive at solutions. In practice, this is much slower than using crossover. A general rule of thumb is to keep mutation low, and increase it even more slowly to combat stagnation.

Drawbacks While GAs have great versatility, there are some drawbacks which significantly limit their utility.

Swiss-Army Chainsaw First, GAs are not the perfect solution for anything. At their core, they are a biased-walk[5]. This means that while they will come to a local optima, there is no guarantee they will achieve a global optimum. If there exists a tailored solution for a problem, using that will probably work faster and better.

And Quick to Anger Second, GAs are subtle things. The fitness function, not given its due in this writeup, can be the difference between a quickly converging solution and processors spinning their wheels for days or months coming to one sub-satisfactory solution after another. A GA will optimize the fitness function- and that's all it will do. So if you want to maximize a metric, say accuracy for a classifier, be aware that it might do exactly that by simply guessing the most prevalent answer in the dataset. This will get it to a local optima, and it might be surprisingly difficult to get it out of it.
Furthermore, there are hyperparameters which effect the GA directly but can be difficult to tease out. What should the elitism percent be? It probably depends on your other parameters. There are ways of optimizing these, and they themselves might be amenable to a further GA, except that evaluating them is time consuming- should your fitness function be speed of convergence? Or the best solution arrived at within 100 generations? That might seem too long, but use fewer generations and you run the risk of invoking too much sensitivity to starting conditions to draw

[5]With a random-walk on one side of the spectrum and a guided approach on the other.

meaningful conclusions. There are some rules of thumb to assist with these situations, but they only mitigate the problem, they don't eliminate it entirely.

Toolset Finally, a GA is only as good as its toolbox. On Earth, that toolkit was physics. All of physics, and massively, embarrassingly parallel at that. That's difficult to take advantage of digitally, where you not only have the responsibility of developing an encoding but also defining the universe your population lives in. For instance, the heart of this paper is whether to use a GA to optimize a classifier or to use a GA to classify things. The classifier has theory underpinning its toolkit, the GA has only whatever fitness function and encoding it is supplied. Or, to get back to our example, it might speed convergence to increase mutation rates and restrict crossover to occur only at the breakpoints of binary words. The downside of this is that GAs are supposed to be able to solve any problem, and while they can, they also require a certain amount of customization to not waste everyone's time. The harder the problem, the more customization that is usually required. And at that point, if a tailored solution of some kind exists it's probably easier to code up and implement than an equivalent GA. At their core, the GA is a biased walk, but building in paths will hopefully make that walk much quicker.

Theoretic Underpinnings Holland's Schema theorem.

Chapter 2

Derivation of the Navier-Stokes Equations

Chapter 3

Computational Fluid Dynamics

Chapter 4

Conclusions

Bibliography

Bibliography

Fermi, E. (1956). *Thermodynamics*. Dover Publications. 5

Lamb, H. (1895). *Hydrodynamics*. Cambridge University Press, Cambridge, UK. 5

Liepmann, H. and Roshko, A. (2001). *Elements of gasdynamics*. Dover Pubns. 5

Saad, T. and Majdalani, J. (2010). On the Lagrangian optimization of wall-injected flows: from the Hart-McClure potential to the Taylor-Culick rotational motion. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Science*, 466(2114):331–362. 5

Appendix

Appendix A

Summary of Equations

A.1 Cartesian

some equations here

A.2 Cylindrical

some equations also here

Vita

Vita goes here...