Function Approximation by a Polynomial Using a Genetic Algorithm

Contents

1. Abstract

Genetic algorithms are one of the best ways to solve a problem for which little is known. They are a very general algorithm and so will work well in any search space. All you need to know is what you need the solution to be able to do well, and a genetic algorithm will be able to create a high quality solution. Genetic algorithms use the principles of selection and evolution to produce several solutions to a given problem.

Genetic algorithms tend to thrive in an environment in which there is a very large set of candidate solutions and in which the search space is uneven and has many hills and valleys. True, genetic algorithms will do well in any environment, but they will be greatly outclassed by more situation specific algorithms in the simpler search spaces. Therefore you must keep in mind that genetic algorithms are not always the best choice. Sometimes they can take quite a while to run and are therefore not always feasible for real time use. They are, however, one of the most powerful methods with which to (relatively) quickly create high quality solutions to a problem. [2].

Now it makes sense to give some key terms of genetic programming:

- **Individual** - Any possible solution
- **Population**- Group of all individuals
- **Search Space** - All possible solutions to the problem
- **Chromosome** - Blueprint for an individual
- **Trait** - Possible aspect of an individual
- **Allele** - Possible settings for a trait
- **Locus** - The position of a gene on the chromosome
- **Genome** - Collection of all chromosomes for an individual

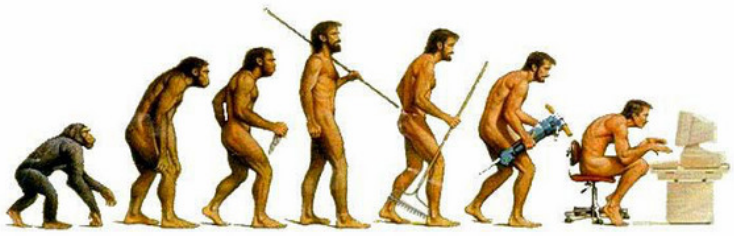

Figure 1. Evolution

In a genetic algorithm, a population of strings (called chromosomes or the genotype of the genome), which encode candidate solutions (called individuals, creatures, or phenotypes) to an optimization problem, is evolved toward better solutions. Traditionally, solutions are represented in binary as strings of 0s and 1s, but other encodings are also possible. The evolution usually starts from a population of randomly generated individuals and happens in generations. In each generation, the fitness of every individual in the population is evaluated, multiple individuals

are stochastically selected from the current population (based on their fitness), and modified (recombined and possibly randomly mutated) to form a new population. The new population is then used in the next iteration of the algorithm. Commonly, the algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population. If the algorithm has terminated due to a maximum number of generations, a satisfactory solution may or may not have been reached.

A standard representation of the solution is as an array of bits. Arrays of other types and structures can be used in essentially the same way. The main property that makes these genetic representations convenient is that their parts are easily aligned due to their fixed size, which facilitates simple crossover operations. Variable length representations may also be used, but crossover implementation is more complex in this case. The fitness function is defined over the genetic representation and measures the quality of the represented solution. The fitness function is always problem dependent. For instance, in the problem of approximation of a set of points by a polynomial (2), the fitness of the solution is the sum of differences between the values in initial points and the values of approximated line.

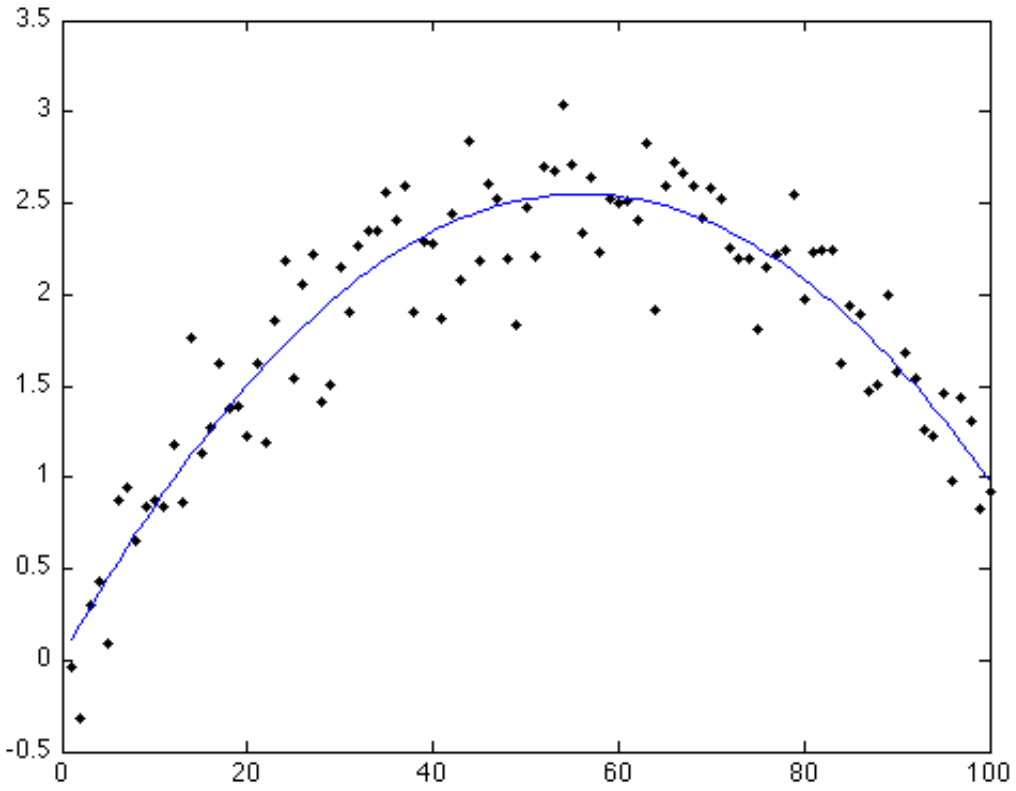

Figure 2. Polynomial approximation to noisy polynomial data

Once the genetic representation and the fitness function are defined, a GA proceeds to initialize a population of solutions (usually randomly) and then to improve it through repetitive application of the crossover, mutation

and selection operators.

Initially many individual solutions are (usually) randomly generated to form an initial population. The population size depends on the nature of the problem, but typically contains several hundreds or thousands of possible solutions. Traditionally, the population is generated randomly, allowing the entire range of possible solutions (the search space). Occasionally, the solutions may be ”seeded” in areas where optimal solutions are likely to be found.

During each successive generation, a proportion of the existing population is $selected$ to breed a new generation. Individual solutions are selected through a fitness-based process, where fitter solutions (as measured by a fitness function) are typically more likely to be selected. Certain selection methods rate the fitness of each solution and preferentially select the best solutions. Other methods rate only a random sample of the population, as the former process may be very time-consuming.

The next step is to generate an extended generation population of solutions from the selected ones. For each new solution to be produced, a pair of ”parent” solutions is selected for breeding from the pool selected previously. By producing a ”child” solution using the above methods of crossover and mutation, a new solution is created which typically shares many of the characteristics of its ”parents”. New parents are selected for each new child, and the process continues until a new population of solutions of appropriate size is generated.

These processes ultimately result in the next generation population of chromosomes that is different from the initial generation. Generally the average fitness will have increased by this procedure for the population, since only the best organisms from the first generation are selected for breeding.

As a general rule of thumb genetic algorithms might be useful in problem domains that have a complex fitness landscape as mixing, i.e., $mutation$ in combination with $crossover$, is designed to move the population away from local optima that a traditional hill climbing algorithm might get stuck in. Observe that commonly used crossover operators cannot change any uniform population. Mutation alone can provide ergodicity of the overall genetic algorithm process (seen as a Markov chain).

Examples of problems solved by genetic algorithms include:

- Function optimization.
- Query optimization in databases.
- Various problems with graphs (the traveling salesman problem, coloring, finding matchings).
- Setup and training of artificial neural networks.
- Layout tasks.
- Scheduling.
- Game strategy.
- Approximation theory.
- Artificial Life.
- Bioinformatics (protein folding).

Thus, simple generational genetic algorithm procedure consists of the following steps [1]:

- Set the fitness function.
- Choose the initial population of individuals.
- (Cycle begin)
    1. Crossover (birth of new individuals)
    2. Mutation
    3. Evaluate the individual fitness of individuals in population
    4. Selection (Select the best-fit individuals for reproduction in next population)
    5. If the termination condition is true (time limit, sufficient fitness achieved, etc.) then (end of cycle), else (cycle begin)

Genetic algorithm execution is a parallel process. That is, there is no dependency across the chromosomes of a population for the process of fitness evaluation and genetic operations. Hence, the entire population can be operated in parallel within a generation. Thus, genetic algorithm can be efficiently implemented using CUDA.

2. Task definition

The aim is to implement a simple genetic algorithm in order to find an approximation of function:

$$f(x) = \sum_{i=0}^{p} c_i \cdot x^i, \quad x \in R$$

The maximum power of the polynomial f is equal to 4. Consequently, total number of function parameters is 5. The polynomial f coefficients are unknown, but a set of values $f(x_i)$ is given on some grid of nodes $N = \{x_i\}_{i=0}^{h}$.

A single solution (or individual) of a population is a set of polynomial coefficients $C_i = \{c_k\}_{k=0}^{p}$.

An individual fitness function is the maximum error ε – difference between $f(x)$ and polynomial $g_{C_i}(x)$, built using C_i, evaluated on N:

$$\varepsilon = \max_{x_i \in N} \left(f(x_i) - g_{C_i}(x_i) \right)$$

It is necessary to find the best set of coefficients C_i, which will give the polynomial $g(x)$ the best approximation of $f(x)$.

3. Proposed method

The proposed method for approximation of a function by a polynomial using a genetic algorithm is described in the listing below.

```
Fitness(individuals, points)
{
    for each individual
    {
```

```
        //In the problem of function approximation by a polinomial fitness can be evaluated as follows:
        for each point from points
        {
            f_approx = 0;

            for (i=0; i < number of polynomial params in each individual; i++)
            {
                f_approx += individual.param[i] * (point_x)^i;
            }

            sumError of current individual += (f_approx – point_y)^2;
        }

        fitness of current indivdual = sumError; // The least value of fitness is, the better ↩
            individual fits the model
    }
    return array of fitnesses;
}

Crossover(individuals)
{
    loop until new population is full //for number_of_parents times
    {
        parent1 = randomly select from individuals;
        parent2 = randomly select from individuals;

        crosspoint = random between 1 and individual_size_in_bits–1;
        child1 = bits from beginning of parent1 till crosspoint + bits from parent2 starting from ↩
            crosspoint;
        child2 = bits from beginning of parent2 till crosspoint + bits from parent1 starting from ↩
            crosspoint;

        //  For example:
        //parent1 == [0 0 0 0 0 0 0 0]
        //parent2 == [1 1 1 1 1 1 1 1]
        //crosspoint(random between 1 and 7) = 3
        //  then
        //child1  = [0 0 0 1 1 1 1 1]
        //child2  = [1 1 1 0 0 0 0 0]
    }
    return parents + new individuals;
}

Mutation(individuals)
{
    //first individual is usually left without changes to keep the best individual
    mutNumber = random between second_individual_start_bit and ↩
        number_of_bits_in_individuals;//usually from 1% to 2% of bits in individuals

    loop mutNumber times
    {
        number of bit to mutate = random between 0 and number of bits in individuals;
        inverse individuals[number of bit to mutate];
    }

    return individuals;
```

```
    //  For example:
    //individuals == [1 1 1 1 1 1 1 1 1 1]
    //mutNumber = 2
    //loop 2 times:
    //   1st: num_of_bit_to_mutate = 2
    //        inverse individuals[2]   ->   [1 1 0 1 1 1 1 1 1 1]
    //   2nd: num_of_bit_to_mutate = 7
    //        inverse individuals[7]   ->   [1 1 0 1 1 1 1 0 1 1]
    //return         [1 1 0 1 1 1 1 0 1 1]
}

Selection(individuals, fitnesses)
{
    sort individuals corresponding to their fitnesses \
    individuals with small (good) fitness value to the beginning \
    individuals with large (bad) fitness value to the end;
    return sorted individuals;
}

Genetic_algorithm(points) // points are the data to approximate by a polynomial
{
    population = Initialize first population (with zeros or some random values);

    loop until (generationNumber > maxGenerationNumber) or (bestFitness < aimed fitness)
    {
        generationNumber++;

        current_fitnesses = Fitness(population, points);

        population = Crossover(first half of population with the best fitnesses)

        population = Mutation(population);

        population = Selection(population, current_fitnesses);

        bestFitness = fitness of first individual;
    }

    return first individual of population with the best params of a polunomial;
}
```

4. Implementation requirements

4.1. Input data

- The set of points on a surface (500–1000);
- The size of population P (1000–2000);
- E_m , D_m – mean and variance for $Mutation$ to generate the random number of mutated genes;
- $maxIter$ - the maximum number of generations, $maxConstIter$ - the maximum number of generations with constant value of the best fitness.

The initial population C_0 is set randomly (it is possible to initialize it with zeros).

4.2. Output data

- The time of processing on GPU;
- The set of coefficients of the polynomial that approximates the given set of points;
- The best fitness value;
- The last generation number (number of evaluated iterations).

4.3. Implementation

The program is required to work on Linux machine. The use of CURAND library for generating random values is mandatory. It is advisable to use the Thrust library.

5. Outcome

1. Getting familiar with a simple genetic algorithm..
2. Getting familiar with CUDA applications development and CURAND and Thrust libraries.

References

[1] John R. Koza. Genetic programming: On the programming of computers by means of natural selection. *Complex Adaptive Systems*, 1992.

[2] Melanie Mitchell. *An Introduction to Genetic Algorithms*. A Bradford Book; Third Printing edition, 1998.