

2D1431 Machine Learning

Lab 4: Genetic Algorithms

Örjan Ekeberg

December 7, 2004

1 Introduction

Genetic Algorithms is a class of learning algorithms based on parallel search for an optimal solution. The algorithms as well as most of the terminology is inspired from evolutionary processes in nature. A *Fitness Function* defines what is to be optimized and the choice of fitness function is what defines the task to be solved.

The parallel search is normally done synchronously in timesteps that are called *generations*. In each generation, a number of search paths are maintained, and in analogy with evolution, these are called *individuals*. The whole set of individuals in a generation is referred to as the *population*. The central idea in genetic algorithms is to preserve and create variations of the individuals that seem most promising and remove the others. Both the variation and the selection can be done in a number of ways and what is best often turn out to be task specific.

1.1 Selection

The selection is normally done by calculating the fitness value for each individual and preserving only those individuals who have the best score. This can also be made in a probabilistic way by assigning a probability of “survival” depending on the fitness score. Often a very simple rule is used, such as keeping the best 25% of the individuals and removing the rest.

If a too aggressive selection strategy is used there is a risk that after a while all individuals in the population become almost identical. When the selection allows also some less fit individuals to survive, the algorithm is more likely to escape from locally maximal points. On the other hand, a very mild selection makes the algorithm unnecessarily slow, since there is

very little new room for novel individuals. The population size is usually kept constant so the selection has to remove enough individuals to make room for all variations generated.

1.2 Variation

The process of creating variations of successful individuals is normally done as two different processes: *mutations* and *crossovers*. Mutation means that small random changes are made to the individuals, typically by perturbing some of the parameter values defining the individual. In practice mutations, together with the selection mechanisms, behave like a gradient following algorithm. Random points close to a good individual are sampled and if better points are found they are incorporated into the population by means of the selection mechanisms.

Crossover operations are ways of combining sub-solutions from different individuals into single new ones. This recombination process has no counterpart in traditional optimization algorithms and can in some situations result in considerable speedups. If each individual consists of a set of parameters, then the crossover operation can simply be to create a new individual by randomly picking parameters from either the first or the second “parent”. If both parents had found good subsets of parameter values, then the new individual may end up with both good subsets. To be really efficient, the internal representation must be chosen to increase the likelihood for this to happen.

2 Exercise 1: Optimization

Our first task will be to use a genetic algorithm for a straightforward numerical optimization task. We will choose a mathematical function where we already know the solution. This makes it easy to see how well the algorithm is doing. We will use two functions, both chosen so that it is hard to find the optimum by a simple gradient following method.

2.1 First function

The first function to use, f , is defined as

$$f([x, y]) = [1 + \cos(15r)]e^{-r^2}, \quad \text{where } r = \sqrt{x^2 + y^2}$$

This function is shown in figure 1. This function has its optimal point at the origin, but any method trying to find this point must be capable of passing

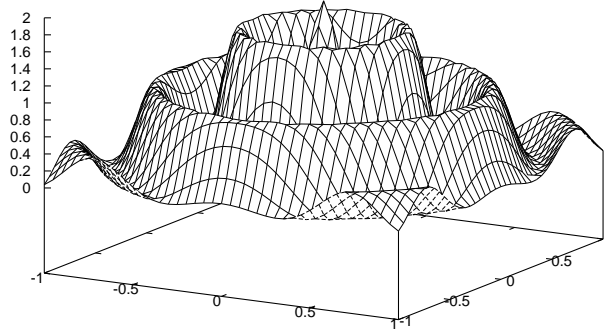


Figure 1: First function ($f = [1 + \cos(15r)]e^{-r^2}$) used for the optimization task.

the rings of locally optimal points surrounding the center.

To make the implementation in Matlab simple, we will use the two numbers x and y as the *chromosome*, i.e. as the internal representation of each individual. The whole population can now be represented as a $N \times 2$ matrix, where each row is an individual.

1. Your first task will be to write the Matlab code for the fitness function. This is simply an implementation of the function $f([x, y])$. Write this as a function which receives a vector as input and returns a number.
2. Next, create a population of 50 individuals at random positions in the (x, y) space. Use random numbers in the range $[-1 : 1]$. Use your fitness function to calculate a vector of fitness values and use the Matlab-function `sort` to reorder the population so that the best individuals come first.

Note that the `sort` function in Matlab returns a second vector containing the permutation indices which becomes very useful for reordering the whole population.

3. Make a function `mutate` which creates new individuals by making small

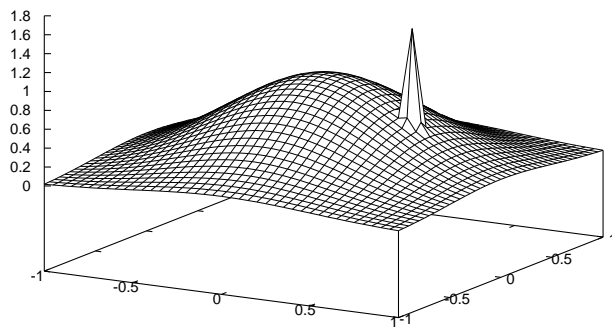


Figure 2: Second function ($g = e^{-2r^2} + 2e^{-1000s^2}$) used for the optimization task.

random perturbations to the numbers of a set of old individuals. Both input and output should be $N \times 2$ matrices, for arbitrary values of N .

4. Make a **crossover** function which creates new individuals by taking the x -value from one individual and the y -value from another.
5. Put everything together into a program which finds the optimal point.
 - How many generations are needed in general to find a point close to the true optimum?
 - Does it matter whether or not you use the crossover operation?

2.2 Second function

Now we will use the same program to find the optimum of another tricky function. As the second test function, g , we will use:

$$g([x, y]) = e^{-2r^2} + 2e^{-1000s^2}$$

$$\text{where } r = \sqrt{x^2 + y^2}$$

$$\text{and } s = \sqrt{(x - 0.5)^2 + y^2}$$

This function is illustrated in figure 2. Here, there is a narrow peak containing the optimal point at $0.5, 0$, while most of the parameter space has a gradient leading to a less optimal point. The question is if the optimization algorithm will be able to find the narrow peak.

Again, answer the questions:

- How many generations are needed?
- Does the crossover operation make a difference?

3 Exercise 2: Feedback Control

The term *Genetic Programming* or *Evolutionary Programming* generally refers to a technique where programs or algorithms are evolved to solve a given task. A central question is how to represent the program and in most cases there is no point in working with source code in ordinary programming languages. Typically, a more compact representation such as instruction lists, evaluation trees or graphs are used.

In practice, the choice of program representation limits the search space for the genetic search and has a large impact on the performance. By choosing a program representation which supports only potentially feasible solutions, the search can go much faster. In this lab, we will use a genetic algorithm to search for a control program capable of steering a car when backing up with an attached trailer. The program will be restricted to a sequence of ten instructions executed on a minimal virtual machine. We will also use a minimal instruction set for this machine; in fact, only two machine instructions are available.

In order to calculate the fitness values, we have to simulate the backing car and trailer, as it is controlled by different programs generated by the genetic algorithm. We will use a simple performance measure, the distance the trailer has moved after 50 s of simulated time.

Most of the Matlab code is already available and your task is to understand what is going on and to make one improvement of the code.

You will notice that the program will take quite some time to run. This is not uncommon for genetic algorithms in general, where execution time for real tasks are often in the order of CPU-days or weeks, rather than seconds or minutes.

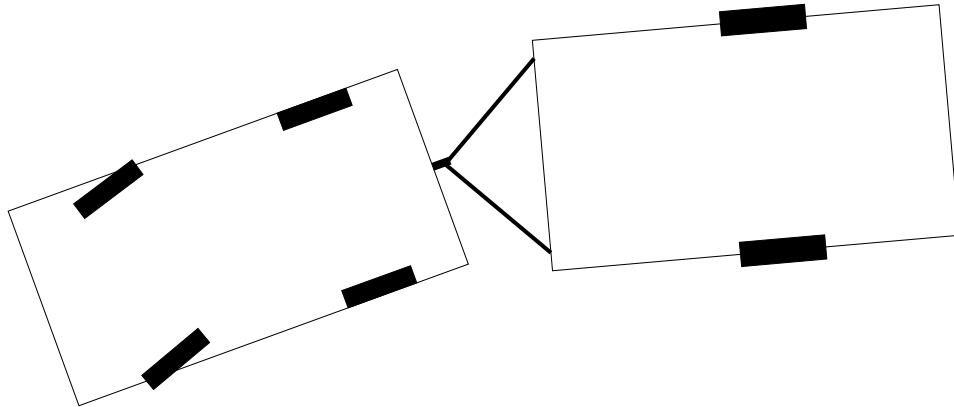


Figure 3: The task is to control the steering of a backing car with an attached trailer. The car is moving at a constant speed backwards.

3.1 Mathematical model of the system

Figure 3 illustrates the mechanical setting. The car is moving backwards at a constant speed, and the objective of the control program is the steer the front wheels of the car so that the trailer continues straight.

In this section we will formulate the mechanical problem mathematically, in the form of differential equations. If you find this too hard to follow, don't despair. You do not really have to understand the details to complete the lab.

3.1.1 The car

We will assume that the car is moving backwards with a constant velocity v . Figure 4 illustrates the geometry which determines the movement of the car. The first part in formulating a mathematical model of the movement will be to understand how the pivot point (x, y) will move.

Let us first take a look at a point p , located right between the rear wheels of the car. The rear wheels will constrain the movement of p to move only along the axis of the car. Thus, the change in p is given by v (the velocity) and Φ (the orientation of the car).

The orientation of the car (Φ) will, however, not stay constant but changes gradually as the car turns. The angular velocity ($\frac{d\Phi}{dt}$) of this turn will be depend in the angle of the front wheels (ω). The angular velocity of the car will be $v/B \cdot \tan \omega$.

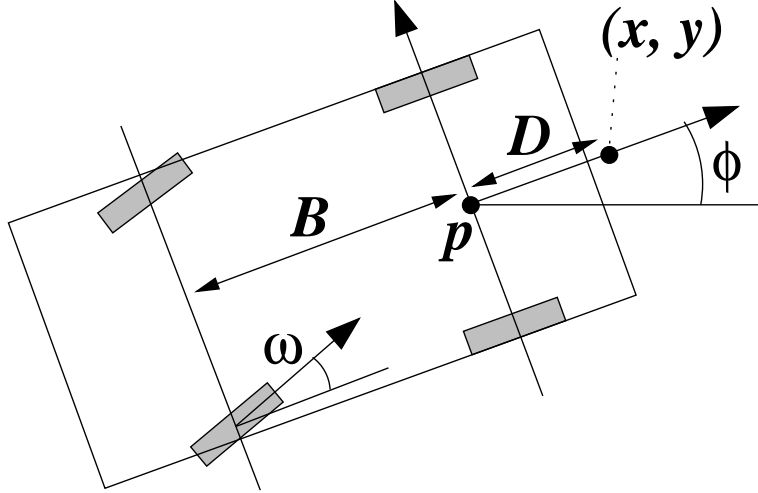


Figure 4: Notations used for the car. The point x, y is where the trailer is attached to the car. The angle of the front wheels (ω) determines how fast the car turns, i.e., how fast the angle Φ changes. The angle Φ , in turn, determines the direction in which the center point p is moving.

An important point to keep track of is the pivot point (x, y) . This point can be calculated given p , D , and Φ , but we will instead use x and y as primary state variables and skip the calculation of p . The movement of (x, y) is composed of the translation of p plus the rotation $\dot{\Phi}$ of the whole car. The movements of x , y , and Φ can be precisely described as a set of first order differential equations:

$$\begin{aligned}
 \dot{x} &= v \cos \Phi - \dot{\Phi} D \sin \Phi \\
 \dot{y} &= v \sin \Phi + \dot{\Phi} D \cos \Phi \\
 \dot{\Phi} &= \frac{v}{B} \tan \omega
 \end{aligned} \tag{1}$$

Given the values of the starting position (x_0, y_0, Φ_0) , the constants v , B , D , and a controller defining the steering ω , we are now able to integrate (1) numerically to get the position of the car any time in the future.

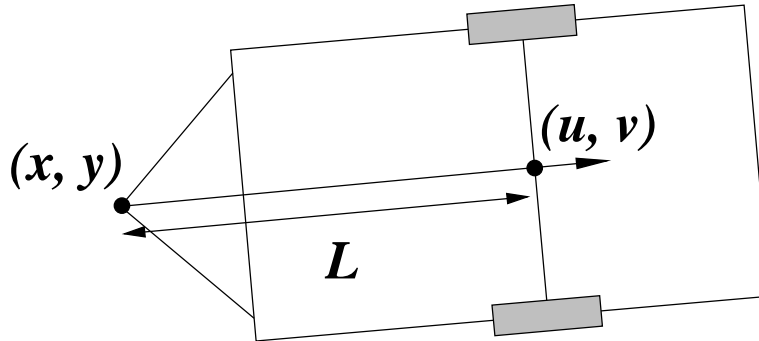


Figure 5: Notations used for the trailer. The point x, y and its movements (\dot{x}, \dot{y}) is completely given by the model of the car. The trailer follows by moving the point u, v such that the distance to x, y is preserved. In addition, u, v can only move parallel to the axis of the trailer, as constrained by the wheels.

3.1.2 The trailer

How does the trailer move? The movements of the center of the trailer (u, v) is constrained by two things. Firstly, the wheels ensure that it only moves in the direction given by the line from (x, y) to (u, v) . Secondly, the distance from (x, y) to (u, v) is constant (L).

$$\begin{aligned} \dot{u} &= \frac{u - x}{L} \cdot \frac{\dot{x}(u - x) + \dot{y}(v - y)}{L} \\ \dot{v} &= \frac{v - y}{L} \cdot \frac{\dot{x}(u - x) + \dot{y}(v - y)}{L} \end{aligned} \quad (2)$$

The movement of the car and trailer is implemented in the function `trailer` (in file `trailer.m`). This function is called by the numerical solver (`ode23`) which in turn is called by our fitness function (`fitness.m`).

3.2 The virtual machine

We will use a minimal virtual machine which has three registers and implements two machine instructions.

The registers are

Abbreviation	Index	Description
PC	1	Program counter, next instruction to execute
ω	2	Output: steering angle (radians)
Φ	3	Input: angle between car and trailer (radians)

In the Matlab program, these registers reside in a global vector called `machine`.

The machine instructions are

Abbreviation	Index	Arguments	Description
Skip-If-Greater	1	<i>mem, val</i>	Skip next instruction if register <i>mem</i> is greater than <i>val</i>
Store	2	<i>mem, val</i>	Store <i>val</i> in register <i>mem</i>

The program consists of ten instructions stored in a row-vector. This vector is 30 elements long since each instruction uses three slots: one for the instruction code (1 or 2), one for the register index, and one for the value. The register indices are actually stored as the index minus two so that the modifiable registers are addressed with positive integers (we do not want the instructions to change the program counter). Note that both the main program (`gp.m`) and the mutation function (`mutate.m`) must know about this coding in order to ensure that only valid programs are generated.

3.3 Your task

Your first task is to run the program and study what the algorithm does. You may need to set up the cputime limit to be able to run the program (command `"limit cputime 1000"` e.g.).

- What strategies are the evolved controllers using?
- Look at the best evolved program (stored in `pop(1,:)`) and decipher what it is actually doing. Translate the program into readable "assembler code". Does it use the available input in any meaningful way?
- Make a rough comparison of the programs in the end. Are they very different or just variations of the same theme?

Your next task is to make *one modification* of the system. There are many things that can be done and you are free to choose any of the suggested improvements below. You are also free to make other things (of the same complexity) that you may find more interesting to explore.

Note that you only need to make
one of
the modifications listed below

Implement crossovers

The current system does not use the crossover operation. One would imagine that crossovers could be quite useful here. Good sequences of instructions could work even better if combined. Implement a simple crossover operation which randomly picks a point in the program (an index) and makes a new individual by taking the instructions before the index from one parent and those after from the other.

Test and see if you get any noticeable differences.

Add new instructions

The instruction set of the virtual machine is extremely small. This is made on purpose to limit the search space. Can you think of other instructions that could possibly work better? Implement one or two more instructions and see if that helps. Does it even become harder because the search space is larger? Can you find another small set that works better?

Varying program length and memory size

The program length was arbitrarily set to ten instructions. Investigate if more or less size improves the situation. Also see what happens when the memory (number of registers) is increased.

Varying learning parameters

The genetic algorithm uses a number of parameters and normally these parameters play a big role. In particular, the amount of randomness injected by the mutation operation is important for determining the spread of values over the population. Another important issue is how the selection process is done.

Vary the amount of mutation randomness and how aggressively the selection process removes non-fit individuals and study the consequences. In particular, notice if you see any difference in how different the evolved programs become.

```

% Run the Genetic Programming algorithm

N = 20; % Size of population
P = 10; % Program size

pop = rand(N,P*3); % Initial population
fit = zeros(N,1);

for i = 1:N
    for j = 1:P
        % Instruction must be 1 or 2
        if pop(i,j*3-2) > 0.5
            pop(i,j*3-2) = 1;
        else
            pop(i,j*3-2) = 2;
        end
        % Proper memory address range
        pop(i,j*3-1) = floor(pop(i,j*3-1)*2);
        pop(i,j*3) = pop(i,j*3) - 0.5;
    end
end

for generation = 1:10
    %% Calculate the fitness from one test run
    for i=1:N
        fit(i) = -fitness(pop(i,:), i);
        drawnow;
    end

    hold('off');

    [f, perm] = sort(fit); % Sort according to fitness

    pp = perm(1:N/4);
    pop = [pop(pp,:); mutate(pop(pp,:)); mutate(pop(pp,:)); mutate(pop(pp,:))];
end

```

```

function f = fitness(chrom, ix)
    global prog
    global nextstep
    global machine

    prog=chrom;
    nextstep = 0;           % When to execute the VM again
    machine = [1, 0, 0];  % Memory of the VM [PC, w, phi]

    t = [0 50];           % Simulated time interval
    xinit = 5;
    yinit = -20 + 2*ix;
    vinit = 1.0;
    phiinit = 0;
    % Start trailer randomly at one of two angles
    if rand>0.5
        offset = 0.2;
    else
        offset = -0.2;
    end

    [t, x] = ode23(@trailer, ...
        t, ...
        [xinit, ...
        yinit, ...
        phiinit, ...
        xinit + 5*cos(phiinit+offset), ...
        yinit + 5*sin(phiinit+offset), ...
        vinit]);

    plot(x(:,1), x(:,2), 'b');
    hold('on');
    plot(x(:,4), x(:,5), 'r');
    axis([0,50,-25,25], 'square');

    [m, n] = size(x);
    xdist = x(m,1)-xinit;
    ydist = x(m,2)-yinit;
    dist = xdist.*xdist + ydist.*ydist;
    f = sqrt(dist);       % Fitness is how far we got

```



```

function fdot = trailer(t, x)
    D = 1;
    B = 2;

    %% Do we need to advance the virtual machine?
    global nextstep
    if t > nextstep
        nextstep = t + 1;
        dostep = 1;
    else
        dostep = 0;
    end

    %% Calculate angle between car and trailer
    phi = unwrap(atan2(x(5)-x(2), x(4)-x(1)));
    angle = unwrap(phi - x(3));

    %% Call controller
    w = controller(dostep, angle);

    v = x(6);
    dv = 0;
    if abs(angle) > pi/2
        %% Stop the car if trailer is not behind the car
        dv = -5*v;
    end

    dphi = v * tan(w) / B;
    dx = v * cos(x(3)) - dphi * D * sin(x(3));
    dy = v * sin(x(3)) + dphi * D * cos(x(3));
    L = (dx*(x(4)-x(1)) + dy*(x(5)-x(2))) ...
        / ((x(4)-x(1))^2 + (x(5)-x(2))^2);
    fdot = [dx,
            dy,
            dphi,
            L*(x(4)-x(1)),
            L*(x(5)-x(2)),
            dv];

```

```

%% The controller
function w = controller(dostep, angle)
    %% Parameters for the controller
    global prog
    global machine

    if dostep==1
        [dummy,progsz] = size(prog); %% Get program size

        machine(3) = angle;

        pc = machine(1); %% Get the program counter

        instr = prog(pc); %% Get current instruction
        mem = prog(pc+1);
        val = prog(pc+2);

        pc = pc+3;

        if instr==1 %% Skip if greater
            if machine(mem+2) < val
                pc = pc+3;
            end
        end
        if instr==2 %% Store
            machine(mem+2) = val;
        end

        if pc > progsz
            pc = 1;
        end
        machine(1) = pc;
    end

    w = machine(2);

```

```

function m = mutate(mm)
    [rows,columns] = size(mm);

    m = mm;
    for i = 1:rows
        for j = 1:columns/3
            if rand<0.2
                if rand<0.5
                    m(i,j*3-2) = 1;
                else
                    m(i,j*3-2) = 2;
                end
            end
            if rand<0.2
                m(i,j*3-1) = floor(rand*2);
            end
            if rand<0.2
                m(i,j*3) = m(i,j*3) + 0.1 * randn;
            end
        end
    end
end

```