Let's look at...

Machine Evolution

As you will see later in this course, neural networks can "learn", that is, adapt to given constraints. For example, NNs can approximate a given function. In biology, such learning corresponds to the learning by an individual organism. However, in nature there is a different type of adaptation, which is achieved by evolution. Can we use evolutionary mechanisms to create learning programs?

Fortunately, on our computer we can simulate evolutionary processes faster than in real-time. We simulate the two main aspects of evolution:

- Generation of descendants that are similar but slightly different from their parents,
- Selective survival of the “fittest” descendants, i.e., those that perform best at a given task.

Iterating this procedure will lead to individuals that are better and better at the given task.

Let us say that we wrote a computer vision algorithm that has two free parameters x and y. We want the program to "learn" the optimal values for these parameters, that is, those values that allow the program to recognize objects with maximum probability p.

To visualize this, we can imagine a 3D "landscape" defined by p as a function of x and y. Our goal is to find the highest peak in this landscape, which is the maximum of p.

We can solve this problem with an evolutionary approach. Any variant of the program is completely defined by its values of x and y and can thus be found somewhere in the landscape. We start with a random population of programs. Now those individuals at higher elevations, who perform better, get a higher chance of reproduction than those in the valleys.

Reproduction can proceed in two different ways:

- Production of descendants near the most successful individuals ("single parents")
- Production of new individuals by pairs of successful parents. Here, the descendants are placed somewhere between the parents.
Machine Evolution

The fitness (or performance) of a program is then a function of its parameters x and y:

![Fitness vs Parameters](image)

Machine Evolution

The initial random population of programs could look like this:

![Initial Population](image)

Machine Evolution

Only the most successful programs survive…

![Surviving Programs](image)

Machine Evolution

… and generate children that are similar to themselves, i.e., close to them in parameter space:

![Generating Offspring](image)

Machine Evolution

Again, only the best ones survive and generate offspring:

![Repeating Process](image)

Machine Evolution

… and so on…

… until the population approaches maximum fitness.

![Approaching Maximum Fitness](image)
Genetic Programming
Instead of just varying a number of parameters, we can evolve complete programs (genetic programming). Let us evolve a wall-following robot in grid-space world. The robot’s behavior is determined by a LISP function.
We use four primitive functions:
- \( \text{AND}(x, y) = 0 \) if \( x = 0 \); else \( y \)
- \( \text{OR}(x, y) = 1 \) if \( x = 1 \); else \( y \)
- \( \text{NOT}(x) = 0 \) if \( x = 1 \); else \( 1 \)
- \( \text{IF}(x, y, z) = y \) if \( x = 1 \); else \( z \)

Genetic Programming
The robot receives sensory inputs \( n, ne, e, se, s, sw, w, \) and \( nw \). These inputs are 0 whenever the corresponding cell is free, otherwise they are 1.
The robot can move either north, east, south, or west.
In genetic programming, we must make sure that all syntactically possible expressions in a program are actually defined and do not crash our system.

Genetic Programming
We start with a population of 5000 randomly created programs and let them perform.
We let the robot start ten times, each time starting in a different position.
Each time, we let the robot perform 60 steps and count the number of different cells adjacent to a wall that the robot visits.
There are 32 such cells, so our fitness measure ranges from 0 (lowest fitness) to 320 (highest fitness).

Genetic Programming
Example for a perfect wall-following robot program in LISP:

Genetic Programming
The best-performing program among the 5000 randomly generated ones:

Genetic Programming
In generation \( i + 1 \),
- 500 individuals are directly copied from generation \( i \)
- 4500 are created by crossover operations between two parents chosen from the 500 winners.
- In about 50 cases, mutation is performed.
Genetic Programming
Example for a **crossover** operation:

Mutation is performed by replacing a subtree of a program with a randomly created subtree.

Genetic Programming
After **six generations**, the best program behaves like this:

And after **ten generations**, we already have a perfect program (fitness 320):

Here is a diagram showing the maximum fitness as a function of the generation number:

Genetic Programming
You could simulate an evolutionary process to improve your **Isola playing algorithm**.

The easiest way to do this would be to use evolutionary learning to find the optimal weight vector in your static evaluation function, i.e., optimal weighting for each evaluation feature that you compute.

For example, assume that you are using the features $f_1$ (number of neighboring squares) and $f_2$ (number of reachable squares).

In each case, you actually use the **difference** between the value for yourself and the value for your opponent.

Game Player Evolution
Then you could use weights $w_1$ and $w_2$ to compute your evaluation function:

$$e(p) = w_1 f_1 + w_2 f_2$$

So the performance of your algorithm will depend on the weights $w_1$ and $w_2$.

This corresponds to the example of the computer vision algorithm with two free parameters.

Thus you could use an evolutionary process to find the best values for $w_1$ and $w_2$ just like in that example.
Game Player Evolution

But how can you determine which individuals survive and procreate?

Well, one possibility would be to hold a tournament in which all individuals compete (or many smaller tournaments), and only the best individuals will reach the next generation, i.e., the next tournament.

The other individuals are deleted and replaced with new individuals that use similar weights as the winners.

This way you will evolve algorithms of better and better performance, or in other words, you will approach the best values for $w_1$ and $w_2$.

Game Player Evolution

You could slightly modify the game code to implement this principle of evolution.

When you have obtained the best values for $w_1$ and $w_2$ (or in your case maybe $w_1$, $w_2$, …, $w_{37}$), just transfer these values into your original program.

Your program should now play significantly better than it did prior to its evolutionary improvement.

Try it out!