Universidad de Valladolid

A nature-inspired didactic approach to Genetic Algorithms & Neural Networks

Alejandro Barón García
Alberto Calvo Madurga

supervised by
Manuel Barrio Solorzano

Algorithms and Computation
Voluntary project

Academic year 2018-2019

"An algorithm must be seen to be believed"-Donal Knuth

Contents

1. Introduction

The aim of this project is to give a visual (therefore didactic) approach to how Genetic Algorithms work. Our main goal is to have a first contact with these algorithms and learn how they work.This approach will be taken with the help of some practical examples on these algorithms.

Apart from implementing the Genetic Algorithm we will give an slight insight on Neural Networks. We will be building two examples on how to use GAs:

- Using GAs as fittest individual creator in order to develop a task
- Using GAs as Neural Networks tuner in automatons coordination tasks (*Coordinated Automatons*)

This project will be developed within the scope of the *Algorithms and Computation* subject in the academic year 2018-2019, as students of the Double Degree in Computer Science & Statistics at University of Valladolid. All the work and code displayed in this document will be available at our github repositories :

- Alejandro Barón García: *https://github.com/AlejandroBaron/didactic-GAs*
- Alberto Calvo Madurga: *https://github.com/AlbCalv/didadtic-genetic-algorithms*

1.1 Tools

In order to make the project, we are going to use *Python 2.7.15rc1*. We chose Python because of its versatility and its current rise in the field of IA and Algorithms.

Some specific libraries are used to:

- **Pygame 1.9.1release** as our simulation environment. Pygame is a Free and Open Source python programming language library for making multimedia applications,which has been chosen because we won't be deploying complicated simulations since the main objective is to see how the algorithms work. While two simulators will be made, one for each example, for accelerating the analysis of convergence one of them will be isolated in one point of the task.
- **Matplotlib.Pyplot** which provides a MATLAB-like plotting framework. We use it in order to display the different trajectories made by our individuals.

- **NumPy** which is the fundamental package for scientific computing with Python. Its most important use in this example is the powerful way of treating N-dimensional array objects.

No libraries will be used for the task of creating the two main algorithms (neural network and genetic algorithm), the goal of the project is to code them by ourselves.

We want to thank you for getting interested in our project and hopefully our readers enjoy the work as much as we enjoyed building it from scratch

2. The Peppered Moth Case

2.1 Context

The case of the Peppered Moth (*Biston betularia*) is a classic example in the teaching of evolution and one of the earliest tests of Charles Darwin's natural selection. During the Industrial Revolution the air pollution had been increased so much that this moth suffered a colour change in the population. The white bark of the trees turned black due to the carbon particles that were contaminating the air.

The frequency of dark-coloured moths increased so that before 1811, the dark-coloured (or melanic) form was still not known, and by the end of the 19th century it had almost completely outnumbered the original light-coloured type with a record of 98%

Later Bernard Kettlewell investigated the adaptation of the peppered moth between 1953 and 1956 and noticed that in a clean environment such as in Dorset, the light-colored body was an effective camouflage while the dark colour was beneficial in a polluted environment like Birmingham.

These days, with the desertion of carbon as a energy source, things have changed again. In fact, if the population of dark-coloured moths was 98% in 1959, in 2003 was only 2%.

Figure 2.1: Dark-coloured moth

Figure 2.2: White-coloured moth

2.2 Introduction

We want to simulate how moths change their appearance in order to camouflage themselves from predators. They achieved this by changing the colour so they could rest in the trunk of trees (more specifically in the birch trees) avoiding detection.

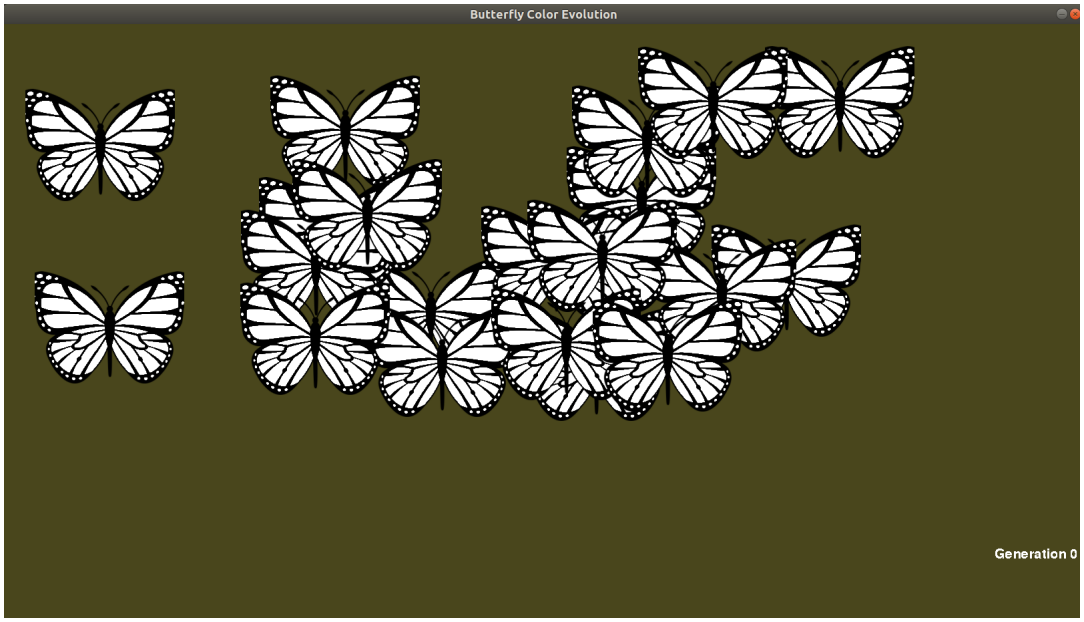
This example will be used in order to introduce the genetic algorithms in its simplest form. Down below the five basic phases applied to this example are explained.

2.3 Genetic Algorithm

2.3.1 Initial population

The process begins setting up the individuals that will form the **Population**. In our case we have 20 individuals starting with white-coloured body, such as if the Industrial Revolution had just started. Each of them has four "genes" which maps directly with the values in the **RGB** code.

We also set the background that simulates the trunk of the tree with an specific colour that will be the colour to reach for our individuals as the goal.



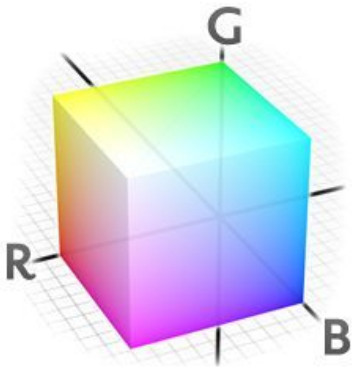
Originally, Peppermoths were white
20 individuals represent the population

2.3.2 Fitness function

The **fitness function** determines how well an individual is performing the task, in other words, how can an individual compete with other individuals. The function gives a **fitness score** to each individual which will determine the probability for reproduction.

In our example we set a function *rgbdistance()* that will recieve two tuples with the values of parameters R,G and B in both moth's colour and tree's trunk, and returns the euclidean distance between them.

$$rgbdistance(RGB_1, RGB_2) = \sqrt{(R_1 - R_2)^2 + (G_1 - G_2)^2 + (B_1 - B_2)^2}$$



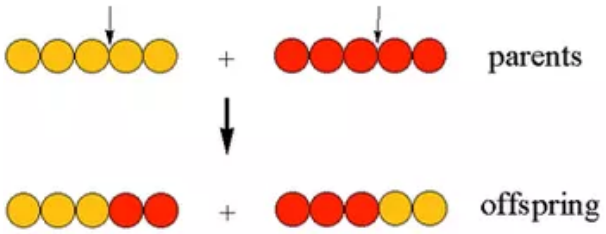
2.3.3 Selection

Once we have calculated how well the individuals have performed in the task, we have to choose an arbitrary number of individuals based on their fitness score. This selection is done with the idea that the fittest individuals will have more probabilities to survive and reproduce in the future.

The number of individuals chosen in this case is 10, corresponding to half of the population.

2.3.4 Crossover

This phase is the most significant one in a genetic algorithm. For each pair of fittest individuals, called parents, their genes are mixed up so the children will have more options to survive than the individuals in past generations.

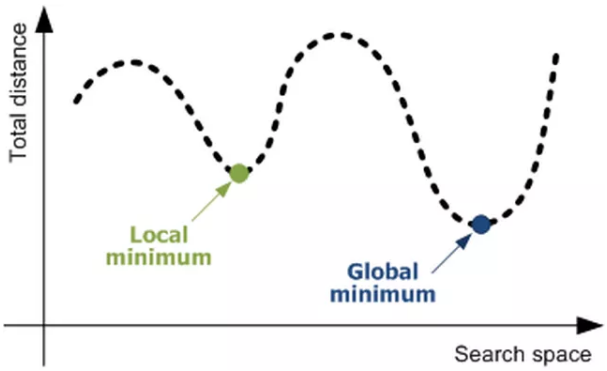


Crossover Phase
Source: http://janmonschke.com/Genetic-Algorithms/presentation/#/16

The number of pairs that will reproduce is going to be 10, and each pair will have two children. The decision of crossover in our case is that the first children will receive the genes as a weighted mean of the RGB values present in both of the parents. The second children will receive directly one parent's RGB values.

Once we have the twenty children we make the decision of changing one of them for another one with random genes. This is made in order to avoid a local minimum.

Local Minimum. A local minimum, also called a relative minimum, is a minimum within some neighborhood that need not be (but may be) a global minimum.



Local minimum problem
Source: https://stackoverflow.com/questions/9457116/in-regards-to-genetic-algorithms

Sometimes the algorithm finds a solution that it's not optimal globally, due to the random initial values of the population, so with this change made, the twenty individuals that shape the **offspring** are now ready for the next generation.

2.3.5 Mutation

In the offspring formed, some of the genes may suffer a slight **mutation**. This phase is very important in order to maintain diversity within the population and prevent premature convergence (local minimum problem).

The decision taken in our algorithm is that every value in the RGB code mutates in a minimum way. This is achieved by multiplying the value for the result of a random observation in a normal distribution with a mean of one and standard deviation of 0.1. Normal distribution was chosen due to the fact that many natural processes follow a normal distribution.

$$MutatedColour(i) = Colour(i) * N(1, 0.1), i \in \{R, G, B\}$$

Intermediate Generations:Only the ten fittest individuals survive and are able to mute. Parents' colours are inherited and slightly mutated for the next generation

2.3.6 Termination

The ideal algorithm terminates if the population converges.In other words,there are two possible situations when the algorithm is considered to have converged:

- Once a certain generation the offspring produced that is not significantly different from the previous generation, but the task has not been perfectly covered *(local minimum/maximum)*
- Once the algorithm has fulfilled the desired task *global minimum/maximum*

In this practical example, we managed to reach a optimal solution.

Optimal Generation:Individuals end up adapting to the background and having similar colours (slightly different due to mutations)

2.3.7 Final Observations

As we said at the beginning of this chapter, this example is made in order to provide a visual and didactic approach to genetic algorithms.

Once we have decided that the genes of an individual, the rest of the phases are simple and intuitive ,such as selecting the fitness function relate with the difference between the RGB values of two colours or making the crossover by getting the mean of the values).

With this base, now we confront a quite more difficult task of mixing this genetic algorithms with neural networks .

3. Coordinated Automatons

3.1 Introduction

The case we chose to take on is Automatons Coordination. We want to see how these Coordinated Automatons coordinate if trained by a Genetic Algorithm.

The movement decision module was not clear at first, but we finally decided to choose a Neural Network as the autonomous module. The role of the Genetic Algorithm will be the one of adjusting the Neural Network parameters (which will be covered deeper at **Section 3.3.2**), such as:

- Weights and Bias

Other parameters (hyperparameters) can be genetized as well, but we didn't implemented this function, such as:

- Number of layers (or the network depth)
- Neurons per layer (or the network width)

Disclaimer: We Failed

We failed while trying to find an optimal solution for the algorithm. Therefore, the following information should be taken from an academic and didactic approach for future examples, not for this specific case but as a general approximation to GAs + Neural Networks. We will discuss the possible reasons why we failed at the end of the document.

As the solution couldn't converge, we provide some ways to understand the problem (such as trajectory graphics) and tweak of parameters. This also will be explained further on.

Coordinated Automatons as ants

The problem we decided to simulate was inspired by how ants cooperate. With the coordination of multiple "simple" beings they can achieve huge tasks like carrying food for the colony. With a far more simple approach, we will create small circular automatons whose goal will be to push a bigger circle to a specified place.

The aim of the model is to find an optimal movement module both for reaching the goal and coordinate all the ants.

Ants coordinate to achieve a common goal: Food for the colony
Source: https://www.malaysia-wildlife-and-nature.com/malaysian-hymenoptera.html

3.2 Choosing the movement environment

As mentioned before, the project simulation environment will be built in Pygame, because it's easy to use. However, this environment does not have any physics implemented(just for collisions but we will be implementing our for circular bodies and Pygame's is mainly oriented to rectangles), it's just for graphics. Therefore we will have to build the physics module from scratch.

3.2.1 Making the Simulation Engine

We will deploy some basic physics to the environment. There is no need in coding elaborated collisions (i.e. ellastic or inellastic).

One class for the actors, and a main function with its physics and movement is, under our perspective, more than enough. This class will represent both the Coordinated Automatons and the Objectives to be pushed.

Creating the actors

Our class Player,representing an actor inside the simulator, in terms of an object,inherits from Pygame.Sprite, having the following **atributes**:

- image: Image representing the Actor
- rect: a Python.Rectangle object

- centerx: current x position
- centery: current y position

Movement

In order to move the actors, we have merged the boundaries check and the movement in a single function, called ***move***(xIncrement,yIncrement) which adds the corresponding shift to each X and Y axis. Basically, in order to detect clashes with the boundaries, we say that one object is touching one border if its X position equals either 0+radius or the total Screen Width (same for Y with Screen Height).It should be noted that the values for Y axis are 0 in the top of the window and reaches the maximum Screen Height at the bottom of it. If this happens, we won't allow any movement towards the border.

Besides this, we face an extra problems that needed its own function to be created: overlapping.

Overlap

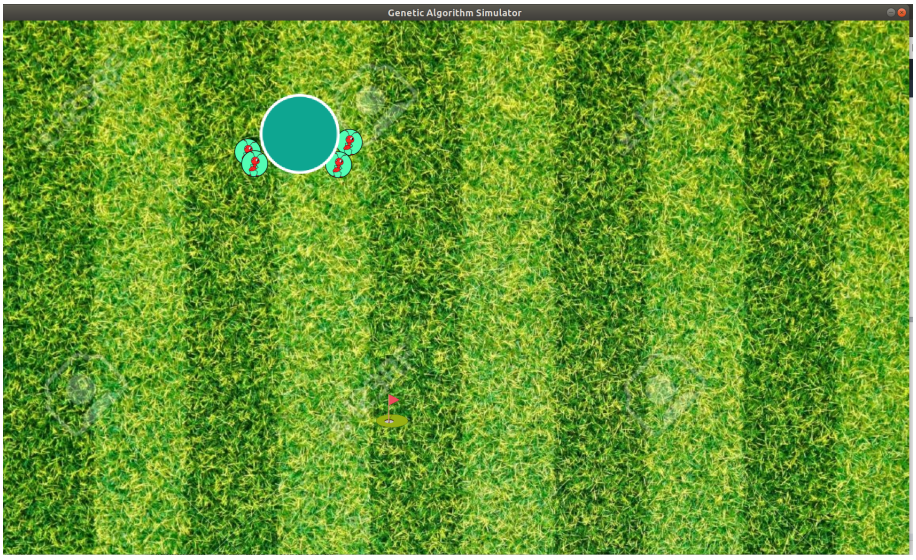
We faced another problem which we called overlap. When one Cell tries to push another body towards a border, our movement function (see above) does not allow the body to be pushed. We have to detect this situation (object being pushed towards one screen border while touching it) and prevent the pushing Cell to be shifted as well. We skipped this problem by adding a function that told us if one body was touching the X or Y border

Global movement

- **Resultant forces:** Every ant (cell) has to take a decision of movement every time, and when she doesn't crash into another one, everything is fine. The interesting point comes when more than one ant are trying to move the food.The idea its to check their intentions of movement so as to make the sum of the forces (taking care of the sign of the movement) and came with a resultant one. We make this decision from the point of physics where its logical to think in forces with specific value more than only telling the x,y direction movement.

$$resforce(A_1, A_2, A_3, A_4) = (\sum_{i=1}^{4} A_i.X, \sum_{i=1}^{4} A_i.Y)$$

- **Spawns directly attached to the food:** At first, we pretended to make the ants search for the food, then push it towards the anthill.However, we decided to spawn them attached directly to the food source.This idea might seem to be over specifying the goal that the ants have to achieve, but that's not true because in reality this will be the first thing the individual will do by instinct. With this improvement there will be less simulations in which ants don't touch the food, simple as that.

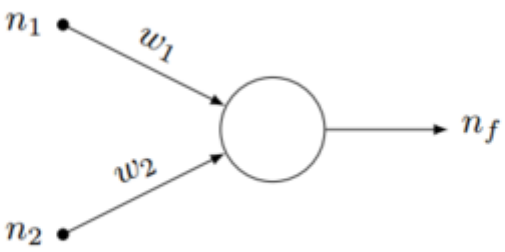


3.3 Neural Network: Our movement decision maker

3.3.1 What is a Neural Network and why to choose it

Neural networks are a form of machine learning which model themselves after the human brain.By combining neurons (basic calculus units), an artificial neural network is created, allowing the computer to learn by incorporating new data. We chose this model since they are optimal in complex environments which may not have a clear deterministic model apt for finding a solution.

Neural network models consist of three types of nodes, input nodes, hidden neurons and output nodes. Hidden neurons are grouped in groups called hidden layers.Each neuron has a set of weights, one for each input link from the rest of neurons of the previous layer. This structure is also known as **perceptron**



The output of each perceptron is defined by the activation function in the following way:

$$nf = \sigma(Wx + b)$$

Which will have an output according to the σ activation function(which we will described below), the input values x (n_1 and n_2 in the picture) and the bias b, which is a different constant for each layer used to adjust the neural network. Once all perceptrons get combined, we have the complex structure called **Neural Network**

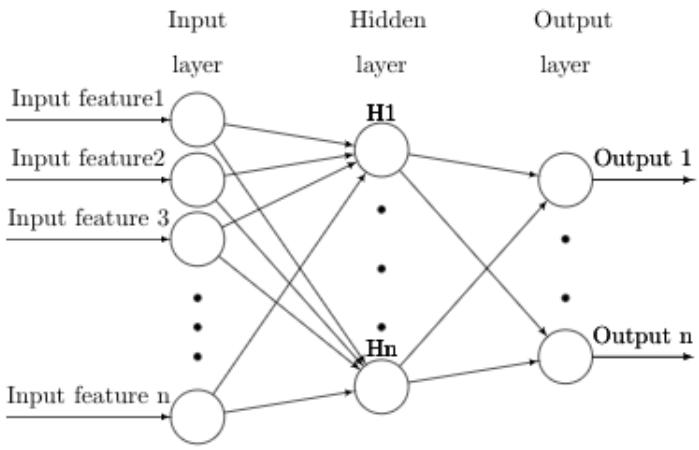
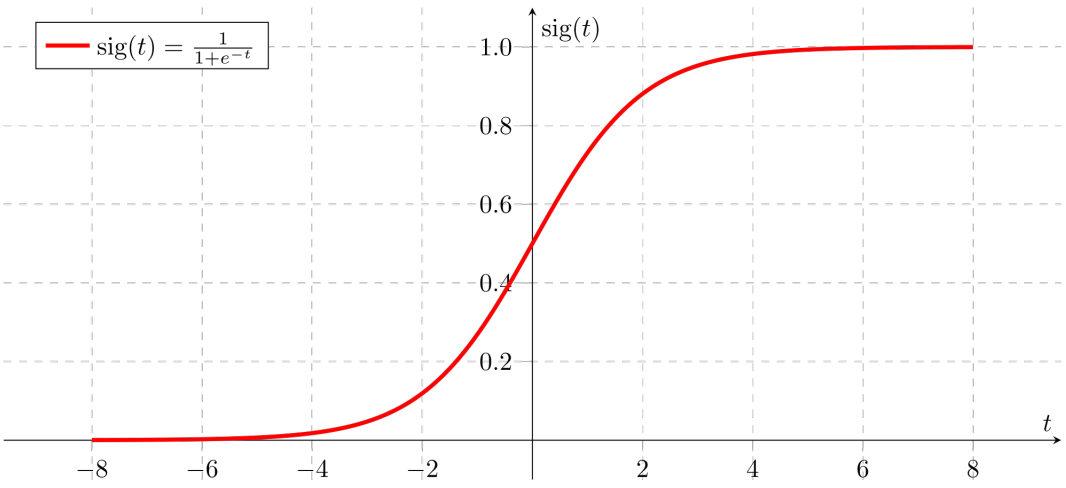


Figure 2.3: Feedforward Neural Network structure.

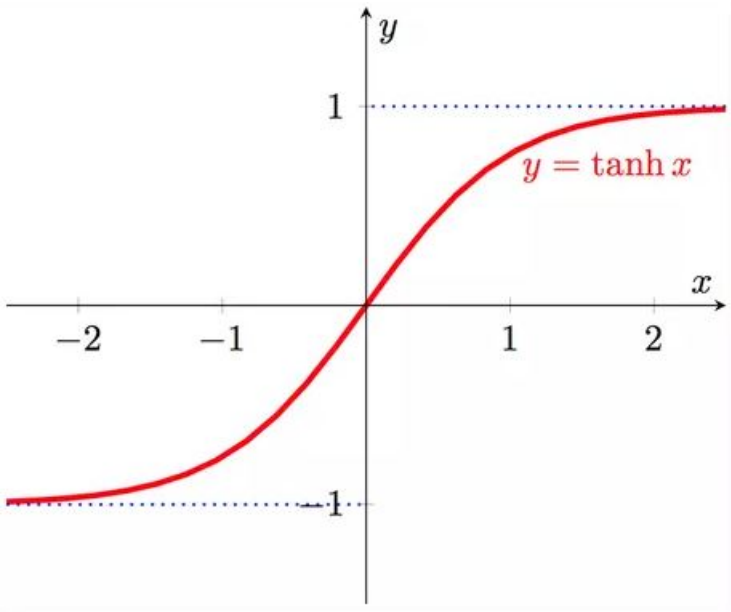
Activation Function

Activation Function decides whether a neuron shall send information to the next layer or not. Most of neural networks use the *sigmoid* function which maps $\mathbb{R}$ to the [0,1] interval.



Sigmoid function

However, since our movement must be forward (+1) or backward(-1) in the output layer we will use the *tanh* function, which maps $\mathbb{R}$ to the [-1,1] interval. Some authors propose the softmax function for this role, but we decided to go for the hyperbolic tangent.



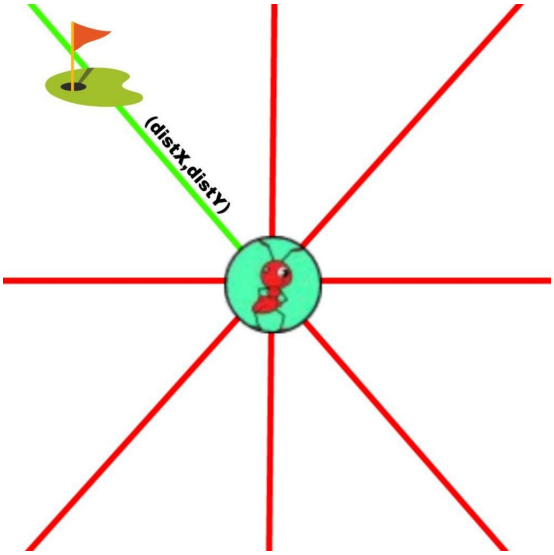
Hyperbolic tangent

3.3.2 Building the Neural Network

Selecting the input parameters

The cells have the mission of carrying the food to a set destiny. In order to achieve it, they have to know about distances. Each ant knows the X and Y distance from the food to the goal,making these the input parameters. With this information, our NN would have 2 input neurons. We tried this set of parameters as a first approach to the problem, trying to have the simplest input.

However, we felt that the model was not receiving enough information about it's environment, so we decided to modify the input. We gave each ant a kind of "sense".They will be able to sound out the environment in 8 directions. If one of the "beams" touches the goal, distance in X and Y from the ant to the goal will be given to the neural network as an input, otherwise 0. With this structure, we will have 8*2 inputs, two values per beam.



Neural Network's input is 0 be unless that beam is touching the goal

Choosing the number of neurons and layers (hyperparameters)

Hidden neurons are the neurons that are neither in the input layer nor the output layer. These neurons are essentially hidden from view, and their number and organization can typically be treated as a black box to people who are interfacing with the system. Additional layers of hidden neurons enables better power to the system, with the cost of additional complexity. Also having too few hidden neurons, can prevent the system from properly fitting the input data and reducing the robustness of the system.

In our case we started with two layers but the user will be able to extend the number to the desired one. This parameter can be changed if the user desires to restart the simulation once a local minimum is reached, but this will aspect be regarded further below.

Its easy to see that the outputs of our NN are going to be the movement decision with the food once we have the resultant forces.

3.3.3 Output Cost calculation

Let x_0 be the input parameters and let W_i, b_i be the weight matrix corresponding to layer i. The output, in a recursive form will be determined by:

$$output = x_{n\_layers+1} = \sigma(W_{n\_layers+1} x_{n\_layers} + b_{n\_layers+1})$$

This implies iterating for $n_n layers + 1$ to calculate the necessary $\sigma(W_i * x_{i-1})$ for the output. Given the number of elements in each matrix W_i ($nneurons\ layer_i * nneurons\ layer_{i-1}$.

We suppose the complexity for computating $\sigma(W_i * x_{i-1})$ is approximately $\mathcal{O}(n^3)$ (using Strassen algorithm and ignoring the cost of computing $\sigma()$ since it's a simple mathematical function over all the elements of the array). For calculus simplicity purposes, we'll consider n as the maximum dimension of all the W_i matrixes

where $n = max(nneurons, ninputs, noutputs)$. Finally, since we calculate $n_nlayers + 1$ results, we can say the complexity of the algorithm is :

$$\mathcal{O}(n^{log(7)} * (nlayers + 1))$$

Pseudocode

Algorithm 1 Neural network

```
1: procedure OUTPUT(input)                          ▷ O(n^log(7) * (nlayers + 1))
2:     x ← input
3:     for i in nlayers+1 do
4:         x ← σ(W_i * x)                           ▷ O(n^log(7))
5:     end for
6:     return x b
7: end procedure
```

3.4 Genetic Algorithm: the network tuner

Unlike other algorithms, NNs with their deep learning cannot be programmed directly for the task. Rather, they have the requirement of learning the information. There are several learning strategies such as supervised learning, unsupervised learning and reinforced learning.

In our case, we'll use genetic algorithms to complete our task of adjusting our NN. As we reviewed in **Chapter 2**, this kind of algorithm consists of an heuristic search to find an optimal solution

How are Genetic Algorithms combined with Neural Networks

There are three main paradigms in Machine Learning: **Supervised Learning, Unsupervised Learning** and **Reinforcement Learning**. In **Reinforcement Learning**, there is no need to have pairs that tell the model which input produces which output. This is just the right situation to use them. Due to the fact that we can't handle all the possible movement combinations, we will let the Neural Network train itself by letting it work on it's own and then telling it how good did it perform on the task.This is one of the main principles of Genetic Algorithms (fitness function) and therefore makes them competitive candidates for the teaching process.

Selection

The idea of this phase is to select the fittest individuals (according to a fitness function) and let them pass their genes to the next generation.This **fitness function** in our case will be the remaining distance between the food and the goal. The lower this value is, better the model has worked, so we will be looking for **optimal minimums**.

Individuals

In this example, our **population** is composed by Neural Networks. Each neural network will have its own **genes** which will be the weights matrixes for each layer **W** and bias **b**.

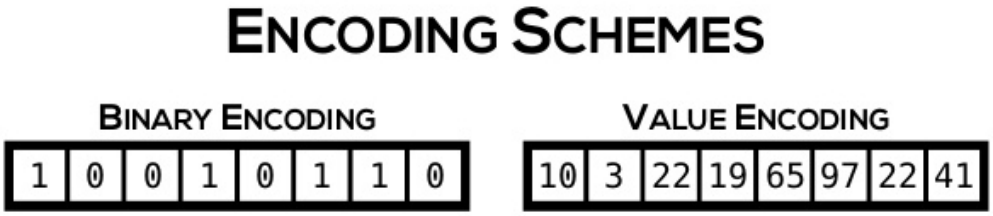
It's important to note the diference between having 4 ants in each simulation and 4 different neural network. Since our goal is to find one neural network that coordinates four ants, each one of them should have the same brain structure, therefore each set of ants will be using the same Neural Network.

Crossover

For each pair of parents to be mated, a crossover point is chosen at random from within the genes. **Offspring** will have their own **W** and **b** inherited as a mixture from their parents'.Matrix cells (each one representing a perceptron w input weight) will be randomly selected from each parents' layer matrixes and put together to conform a new **W** matrix. In the same way, biases **b** will be mixed to get the neural network child's bias. A third child will be an exact copy of one of the parents.

In our case we impose that the parents have to be different, in order to mix the genes. Also after the children are created, we choose one randomly that won't enter in the next generation. The substitute will be a randomly created individual. This is made in order to avoid local minimum.

Another possible way to mix genes: There are two main ways to combine weights in reinforcement training. It depends on the genes encoding. In our case, we chose to have a Matrix representation. Some authors propose a binary encoding, transforming each matrix into a binary string and trim it randomly. However, there are no significant evidences that this encoding is better than value encoding, so we chose the latter for our project.



Random Offspring: Just like we did in the Peppermoth example, we set one child as completely random (random set of weights and bias), in order to give our algorithm the chance to escape from a local minimum.

Mutation

Once the new offspring is formed, some of their genes are subjected to a mutation. This mutation will be a random percentage following a normal distribution. We decided to choose the normal distribution since we want light distribution tails, in other words, we don't want extreme percentage mutations.

Termination

Due to the complexity of creating our own neural network for this particular task, joined with the main genetic algorithm, the solution may not converge to the expected result.

Taking into account this problem, we allow dynamic changes in some parameters during the simulation:

- Change in the nº of layers of the neural network.
- Change in the nº of neurons by layers.
- Change in the variance of the mutation.

These changes happens when five generations are simulated and generation's average fitness does not improve.

There are two options that can be used in order to establish these changes:

- The user decides with change is made by introducing the information as an input when the algorithm reaches the "not improving" situation.
- The program chooses randomly which change is made. This option is probably the most interesting, it allows to simulate the program for long time in order to test the convergence of the algorithm

3.4.1 Cost

Let P be the population size and G the number of generations.

$$O(Genetic) = G * (O(Selection) + O(Crossover) + O(Mutation)$$

Each of these terms can be calculated in the following way

- $O(Selection) = O(P * log(P))$ since consists on ordering a vector of fitnesses and extracting the top 30%
- $O(Crossover) = O(P * O(mixingparents))$ which we couldn't calculate
- $O(Mutation) = O(P{*}wmatrixsize) = O(P{*}n^2)$ where $n = max(nneurons, ninputs, noutputs)$, see section 3.3.3

Therefore $O(Genetic) = O(G * P * (log(P) + O(mixingparents) + n))$

Pseudocode

Algorithm 2 Selection

input: individuals

1: $sorted \leftarrow sort\_by\_fitness(individuals)$
2: $fittest \leftarrow sorted[1 : (30\% * n\_individuals)$
3: **return** $fittest$
=0

Algorithm 3 Crossover

input: fittest **declare**: offspring

1: **for** k *in* $n\_individuals$ **do**
2: $parent1 \leftarrow random\_choice(fittest)$
3: $parent2 \leftarrow random\_choice(fittest)$
4: $child1 \leftarrow mix1(parent1, parent2)$
5: $child2 \leftarrow mix2(parent1, parent2)$
6: $child3 \leftarrow parent1$
7: $offspring.append(child1, child2, child3)$
8: **end for**
9: **return** $offspring$
=0

Algorithm 4 Mutation

input: offspring

1: **for** for child in offspring **do**
2: **for** w in child.weights() **do**
3: **for** w(i,j) in w **do**
4: $w(i, j) \leftarrow w(i, j) * rnorm(1, 10\%)$
5: **end for**
6: **end for**
7: **end for**
8: **return** $offspring$
9: =0

3.5 Non convergence analysis

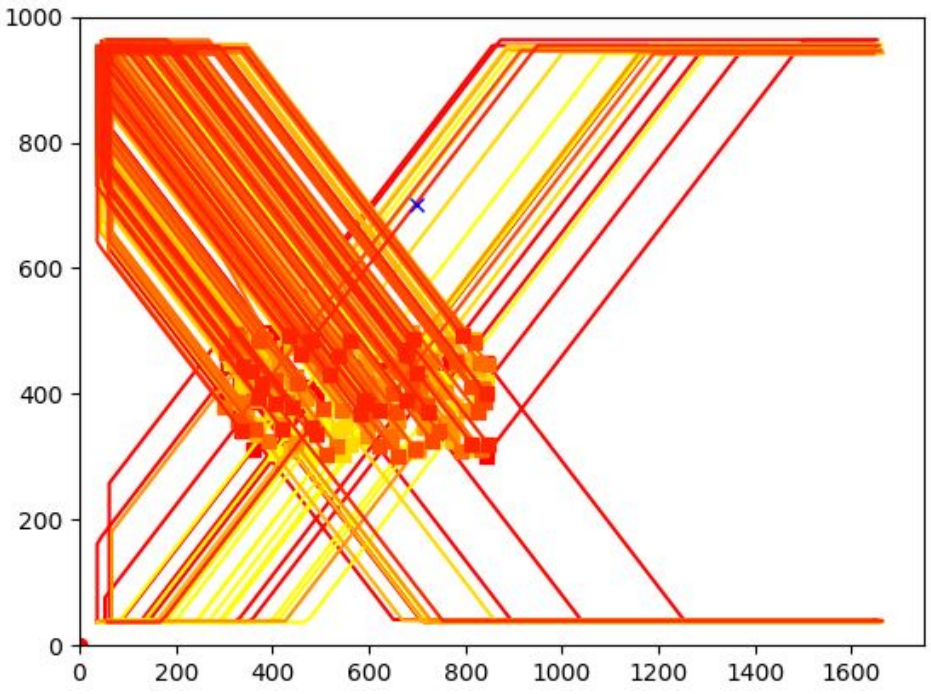
3.5.1 Movement analysis: trajectory tracking

As we have being telling all over the document, the main idea of the project is to provide a didactic and visual way to explain these algorithms. One of the ideas we found interesting in order to see how well our code works, is a track of the trajectories followed by our individuals.

In order to complete this task we use the *matplotlib.pyplot* library which provides a very comfortable way to make graphics. In our case, we have collected every movement decision for one individual and have joined their position changes so the final vision of and individual assembles to a snake where the start position relates to the head and the movement to the tail.

Our suspicion when we first tried to simulate the task in a visual environment, was that there were determined trajectories followed by the individuals, so this is a very useful way to draw conclusions about this problem.

Also in order to represent the different generations before we restart the simulation with different parameters, a colour gradient is made so the first generation has yellow colour and it ends up being red with the last generation.



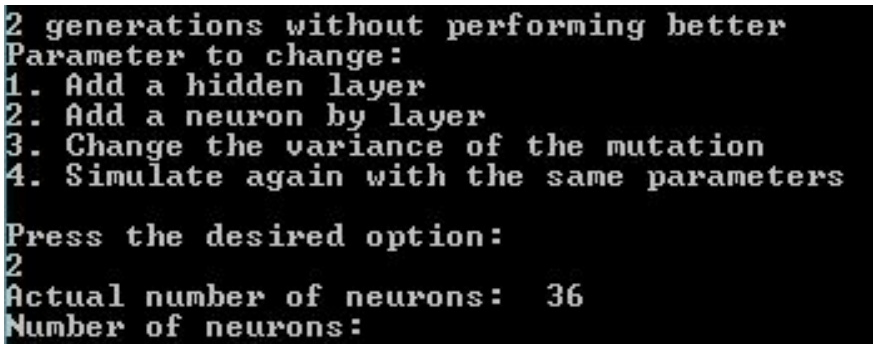
Trajectory tracking

Parameters tweaking

As we told before, our solution doesn't converge, so one idea is to interrupt the simulation when you reach a local minimum. This can be discovered if for some generations in a row, the avg. fitness hasn't improved.

One simple solution for trying to fix this is to change some parameters when this situation happens:

- Change the number of hidden layers
- Change the number of neurons by layer
- Change the standard deviation which affects directly to the mutation phase
- Simulate again with the same parameters



Parameters tweaking

4. Final thoughts

4.1 Files in the project

The different files used for each example are shortly described down below:

The Peppermoth Case

- **geneticbutterfly.py:** where the genetic algorithm is implemented.
- **set_picture_colour.py:** takes the basic butterfly photography and changes the colour of the skin
- **ButterflySim.py:** simulates the example visually.

Coordinated Automatons

- **GeneticAlgorithm.py:** where the genetic algorithm is implemented.
- **NeuralNetwork.py:** where the Neural Network is created.
- **Simulator.py:** simulates the example visually by importing this two files.
- **simulator_no_graphics.py:** avoids the visual part of the Simulator.py for speed reasons.
- **Plotter.py:** creates the trajectory plots with data found in **data.txt**.
- **data.txt:** data file used to create the trajectory graphics.
- **graphsim.py:** one trajectory plot per simulation

4.2 Possible Improvements

There are some possible measures we didn't take, but may help to solve or improve the situation.

- **Different Hyperparameters:** We tried multiple Hyperparameter sets, but none of them worked for our example. User can toy with this option via console in our program

- **Different mutation rates**: Also changeable by the user via console.
- **Try binary encoding**
- **Genetize the Hyperparameters:** As mentioned before, Hyperparameters can be genetized, having different neural networks for each individual in the generation (with different shapes). However, this was out of our reach and time
- **Increase population's size**
- **Increase the number of beams or modify network's parameter**
- **Change the fitness function** so it penalizes approaching the goal and then going away from it.

We are not disappointed with our failure. In the way, we learned a lot about Neural Networks and Genetic Algorithms, we had no prior knowledge about these algorithms. We enjoyed the learning process, discovering a whole new world in the algorithmic field. As Thomas Edison said :

'I have not failed. I've just found 10000 ways that won't work.'

Bibliography

[Brass & Brat] G. Brassard, P. Bratley. *Fundamentos de Algoritmia*. Prentice Hall, 2006

[Cormen] T.H. Cormen, C.E. Leiserson, R.L. Rivest. *Introduction to Algorithms*. MIT Press, 2001

[Mitchell] Melanie Mitchell *An Introduction to Genetic Algorithms*.

Useful third parties material

[1] https://es.wikipedia.org

[2] https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3

[3] https://www.techradar.com/news/what-is-a-neural-network

[4] http://janmonschke.com/Genetic-Algorithms/presentation/#/16

[5] http://homepages.inf.ed.ac.uk/pkoehn/publications/gann94.pdf

[6] https://towardsdatascience.com/how-to-build-your-own-neural-network-from-scratch-in-python-68998a08e4f6

[7] https://www.ijcai.org/Proceedings/89-1/Papers/122.pdf

Tools Documentation

[8] PyGame. http://www.pygame.org/docs/

[9] Numpy.http://www.numpy.org/

[10] PyPlot.https://matplotlib.org/api/pyplot_api.html