



ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

"Σχεδιασμός και υλοποίηση τεχνητής νοημοσύνης σε παιχνίδια χρησιμοποιώντας νευροεξελικτικούς αλγόριθμους."

"Design and implementation of artificial intelligence in games using neuroevolution algorithms"

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Του

ΜΠΑΝΑΚΑΚΗ ΚΩΝΣΤΑΝΤΙΝΟΥ

Βόλος, Ιούλιος 2017



ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

**"Σχεδιασμός και υλοποίηση τεχνητής νοημοσύνης σε παιχνίδια
χρησιμοποιώντας νευροεξελικτικούς αλγόριθμους."**

**"Design and implementation of artificial intelligence in games
using neuroevolution algorithms"**

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Του

ΜΠΑΝΑΚΑΚΗ ΚΩΝΣΤΑΝΤΙΝΟΥ

Επιβλέποντες :

Τσομπανοπούλου Παναγιώτα
Αναπληρωτής Καθηγήτρια Π.Θ.

Τσουκαλάς Ελευθέριος
Καθηγητής Π.Θ.

Εγκρίθηκε από την διμελή εξεταστική επιτροπή την εβδομή Ιουλίου 2017

(Υπογραφή)

.....

Τσομπανοπούλου Παναγιώτα
Αναπληρωτής Καθηγήτρια Π.Θ.

(Υπογραφή)

.....

Τσουκαλάς Ελευθέριος
Καθηγητής Π.Θ.

(Υπογραφή)

.....
ΜΠΑΝΑΚΑΚΗΣ ΚΩΝΣΤΑΝΤΙΝΟΣ

Διπλωματούχος Μηχανικός Ηλεκτρονικών Υπολογιστών, Τηλεπικοινωνιών και Δικτύων του
Τμήματος Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Πανεπιστημίου Θεσσαλίας

© 2017– All rights reserved

Περίληψη

Ένα πρόβλημα που αντιμετωπίζουν τα σύγχρονα ηλεκτρονικά παιχνίδια, είναι ότι η συμπεριφορά των χαρακτήρων που δεν χειρίζονται από κάποιον παίκτη είναι προσχεδιασμένη και προμελετημένη και δεν αλλάζει καθόλου κατά την διάρκειά του παιχνιδιού. Οι προγραμματιστές τεχνητής νοημοσύνης συνήθως προσπαθούν να λύσουν αυτό το πρόβλημα φτιάχνοντας τεράστια συμπεριφορικά δένδρα που όμως και πάλι οδηγούν σε προσχεδιασμένες συμπεριφορές, αυτές οι συμπεριφορές μπορεί να είναι αρκετές αλλά είναι πεπερασμένες. Αυτή η διπλωματική εργασία παρουσιάζει έναν εναλλακτικό τρόπο προσέγγισης αυτού του προβλήματος. Γενετικοί αλγόριθμοι και τεχνικά νευρωνικά δίκτυα θα χρησιμοποιηθούν για να εξελιχθούν αυτές οι συμπεριφορές. Εν τέλει, οι γενετικοί αλγόριθμοι θα αναλάβουν την εξέλιξη από τα βάρη των τεχνητών νευρωνικών δικτύων. Αυτοί οι νευριεξελικτικοί αλγόριθμοι θα χρησιμοποιηθούν για να ελέγξουν απλοϊκούς χαρακτήρες που δεν ελέγχονται από κάποιον παίκτη για να καταφέρουν να κερδίσουν τον αντίπαλό τους, δηλαδή τον παίκτη. Ωστόσο για να αναδυθούν επιτυχείς συμπεριφορές πρέπει το πρόβλημα να έχει κατανοηθεί σε βάθος.

Abstract

One problem in modern computer games is that the non player character has behaviours that are scripted and never change. The artificial intelligence developers usually try to solve this problem by writing enormous behavioural trees, but this also leads to scripted behaviors, this behaviors might be many but they are measurable. This thesis will present an alternative way of solving this problem by using genetic algorithms and artificial neural networks to create evolved behaviours. The genetic algorithm eventually will take the responsibility to evolve the weights of an artificial neural network. This evolutionary algorithms will be used to control simple non player characters so they can beat their opponent, the player, in a simple game. However the only way for successful behaviors to emerge is to comprehend the problem very well.

Contents

Contents	9
1. Thesis Structure	11
2. Genetic Algorithm	12
2.1 Introduction	12
2.2 Software Genetic Algorithms	12
2.3 Fitness Function	13
2.4 Selection Techniques	15
2.5 Scaling Techniques	16
2.6 Crossover	17
2.7 Mutation	20
2.8 Limitations	21
3. Artificial Neural Networks	22
3.1 Introduction	22
3.2 Artificial neural network	22
3.3 The Perceptron	23
3.4 Activation Function	24
3.5 Feed Forward Network	26
3.6 Recurrent Networks	27
3.7 How the networks learn	28
4. Neuroevolution	29
4.1 Introduction	29
4.2 Direct representations.	29
4.2.1 Conventional direct representation.	29
4.2.2 Symbiotic, Adaptive Neuro Evolution.	30
4.2.3 Neuro Evolution of Augmented Topologies.	30
4.3 Developmental representations.	33
4.4 Indirect representations.	34
4.5 Real-time Evolution.	35
5. Implementation	36
5.1 Introduction	36
5.2 Genetic algorithm for movement.	36
5.2.1 Genome representaion.	36
5.2.2 Mutation	36
5.2.3 Crossover and Parent Selection	37
5.2.4 Fitness function	37
5.2.5 Results	37

5.3 Improving and enhancing the simulation.	38
5.3.1 Genome Representation.	38
5.3.2 Crossover and Selection.	38
5.3.3 Mutation.	38
5.3.4 Fitness functions.	39
5.3.5 Results.	39
5.4 The enemy of the agents.	43
5.5 Brain out of Neurons.	43
5.5.1 Genome Representation.	43
5.5.2 Network Topology.	43
5.5.3 Population Pool.	44
5.5.4 Crossover and selection.	44
5.5.5 Mutation.	44
5.5.6 Fitness Functions.	44
5.5.7 Results.	45
6. Conclusions.	47
7. Bibliography	48

1. Thesis Structure

The chapters of this thesis can easily be divided into two different categories. The first is about the background knowledge that is needed and the second is the implementation of this theoretical background.

In the 2nd chapter genetic algorithms are studied. Starting with a small introduction of how evolution works according to biology and continuing on how this idea can be implemented on digital devices. In the end of this chapter limitations of genetic algorithms are presented.

The 3rd chapter is all about the foundations of how artificial neural networks work. The artificial neural networks are also inspired by biology and the first paragraph of the chapter tries to make a connection between the physical neuron and the artificial one. After that a small historical flashback on how the idea of neural networks started and then the chapter tries to give as much information about how different artificial neural networks work.

The idea behind neuroevolution is demonstrated on the 4th chapter. This chapter tries to categorize different neuroevolution algorithms according to their genome representation while at the same time displays key information about them. The last paragraph of the chapter describes briefly how neuroevolution algorithms can be applied in real-time systems such as game.

The 5th chapter is all about using the knowledge of the previous three chapters into a game. It starts with an easy task, which is the movement of an agent from point A to a point B. Then it tries to evolve different species of agents that are doing the same thing. After introducing the player, the agents acquire an artificial brain with the help of artificial neural networks that is being evolved to beat the player with the help of genetic algorithms.

Chapter 6 presents the conclusions of this work while chapter 7 contains the bibliography.

2. Genetic Algorithm

2.1 Introduction

The genetic algorithms grow and evolve over time to converge upon a solution to a particular problem. This behavior is the same in the living organisms. They evolve for many generations to become more and more successful at many tasks, this might be survival reproduction and many more.

The whole idea behind the genetic algorithms is driven by biology. The living organisms consist of a collection of cells. Each cell contains the same information, in the form of DNA, which is called chromosome. Individual chromosomes are built from smaller building blocks called genes. These genes are composed by nucleotides. These nucleotides are linked together into long chains of genes and encode a certain trait of the organism. This trait creates a chromosome and the collection of all the chromosomes is also known as genome. The genome is all the information necessary to reproduce an organism, this is how cloning works.

Evolutionary wise an organism can be considered “successful” if it mates and gives birth to child organisms which on their turn will go on and reproduce. To do this every organism must be good at many tasks, such as finding food or avoiding predators etc etc. The measure of success of an organism in all of its given tasks is called fitness. The more fit an organism is, the more likely it is to create an offspring, “survival of the fittest”.

Creating offsprings when two organisms mate and reproduce will essentially create a mix of the parents genomes which will consist from an entirely new set of chromosomes. This process is called recombination or crossover.

Given that said it's easy to see that the offsprings will only have traits that their parents have, but this it's not quite true. Sometimes an error might occur and the offspring will have an entirely new trait or lost an old one. These errors are called mutations. The probability of a mutation is very small, but it might be very significant in a large amount of generations. Some of the mutations might give the offspring advantages, other might give disadvantages and other might have no effect at all.

2.2 Software Genetic Algorithms

Step One: Generate the initial population of individuals randomly. (First generation)

Step Two: Evaluate the fitness of each individual in that population (time limit, sufficient fitness achieved, etc.)

Step Three: Repeat the following regenerative steps until termination:

1. Select the best-fit individuals for reproduction. (Parents)
2. Breed new individuals through crossover and mutation operations to give birth to offspring.
3. Evaluate the individual fitness of new individuals.
4. Replace least-fit population with new individuals.

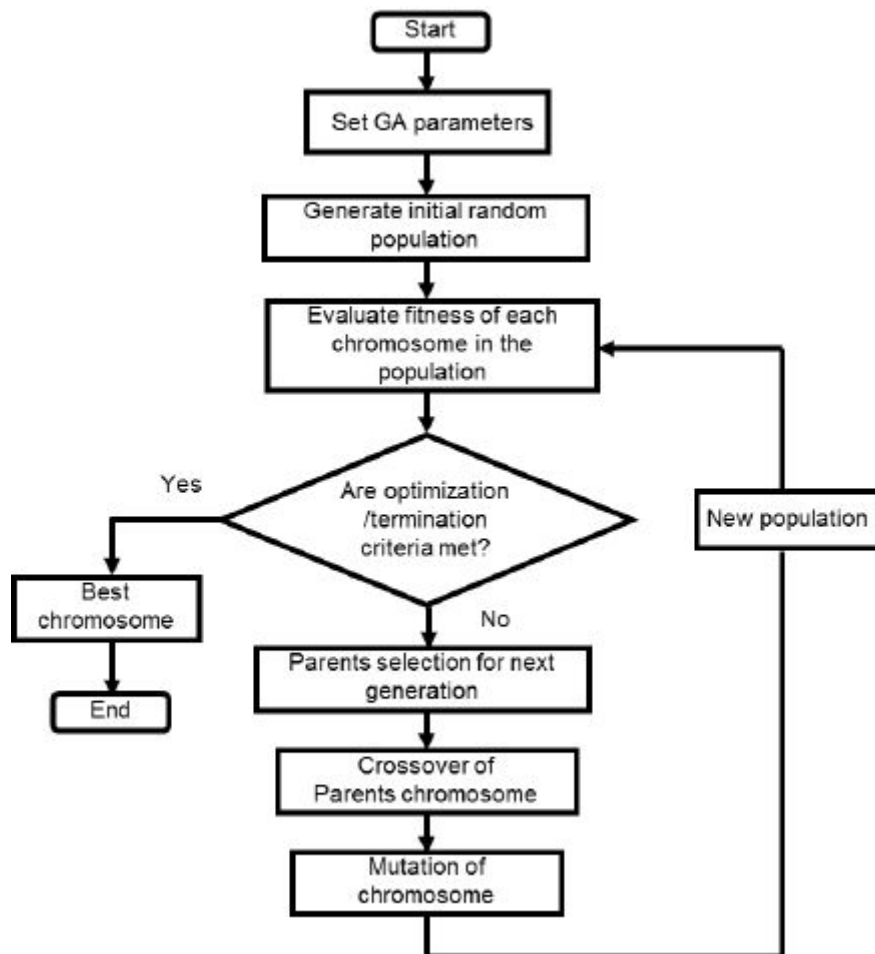


Figure [1]
Genetic algorithm visualization

One of the most key steps of the algorithm is the encoding of the chromosome. The initial population of a genetic algorithm is created by giving a random value in its of the genes. Whenever the whole length of the chromosome is filled with random value then an individual of the initial generation is created.

Step three of the algorithm is often called an epoch. It is the loop of the algorithm which more or less ties everything together. The epoch is also called generation.

2.3 Fitness Function

A fitness function is a particular type of objective function that is used to summarise how close a given design solution is to achieving the set of aims. This function is responsible in genetic algorithms to assign the proper fitness score to each individual after testing or simulating his behaviour.

Even though the final solution of the problem is designed by the computer, the human is the one that designs the fitness function. It's not always an easy task, a function that is designed poorly will either converge to an inappropriate solution or will not converge at all.

Moreover, the fitness function must not only correlate closely with the designer's goal, it must also be computed quickly. Speed of execution is very important, as a typical genetic algorithm must be iterated many times in order to produce a usable result for a non-trivial problem.

As shown in figure [2] there is a possibility that the population will stuck in a local maxima instead of finding the global one.

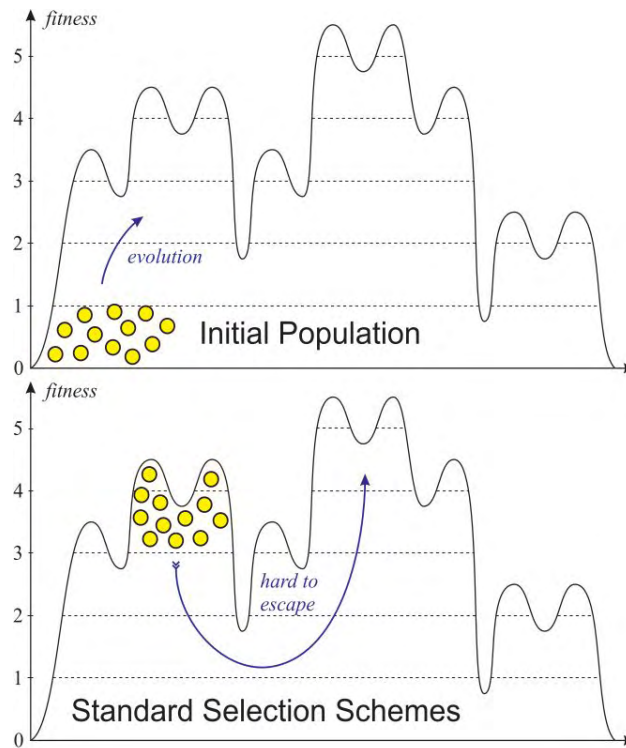


Figure [2]
Genetic Algorithm stuck in Local maxima.

A genetic algorithm may have more than one parameter in its fitness function. A two parameter fitness landscape would be in 3D as shown in figure [2].

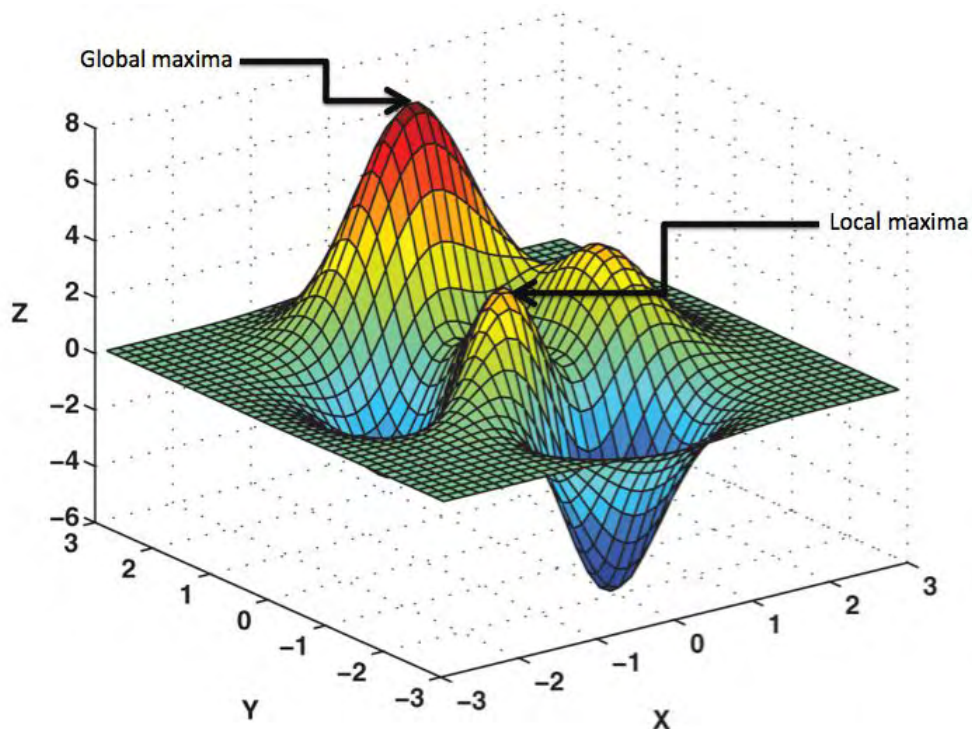


Figure [3]
A two-parameter fitness landscape.

The x and the y axis represents the parameters and the z axis represents the fitness. If you have more than two parameters then it can't be represented but they are hyperspaces. However the algorithm may still stuck in a local maxima.

2.4 Selection Techniques

Selection is how you choose individuals from the population to provide a gene base from which the next generation of individuals is created. How a genetic algorithm chooses who will be the parents for the next generation plays a very important role in how efficient your algorithm is. If the fittest individuals are chosen all the time, the population may converge too quickly at a local optima and stuck there. On the other hand if individuals are selected completely random, then the genetic algorithm will probably take some time to converge (if it ever does at all). Essentially every genetic algorithm must have its strategy that will make it converge fairly quickly but yet enables the population to retain its diversity.

A good strategy is to give individuals a better chance of being selected as a parent according to their fitness score. The most common way of implementing fitness proportionate selection is roulette wheel selection. It does not guarantee that the fittest member goes through to the next generation, but it has a good probability of doing so. This technique can be visualized with a pie chart (figure [4]). It can now choose an individual just by spinning the wheel, tossing a ball, and grabbing the chromosome that the ball stopped.

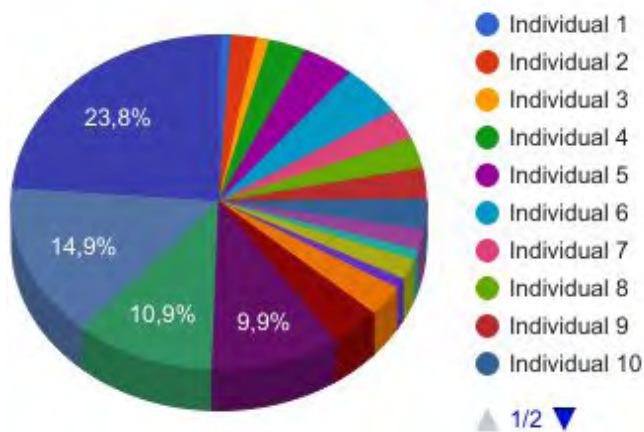


Figure [4]

A roulette wheel selection of chromosomes.

Stochastic Universal Sampling is an attempt to minimize the problems of using fitness proportionate selection on small populations. Basically, instead of having one wheel which spun several times to obtain the new population, stochastic universal sampling uses n evenly shaped pointers which are only spun once. The amount of pointer is equal to the amount of offsprings required. Each pointer is away from its previous one by an angle of $360/n$ degrees.

Moreover Tournament Selection algorithm can be used. The algorithm selects n random individuals from the population and then the fittest of these genomes is chosen to be added to the new population. This process is repeated as many

times as is required to create a new population of genomes. Any individual selected are not removed from the population and therefore can be chosen any number of times.

The last technique is very efficient. The only drawback is that sometimes it might lead to very quick convergence that might have no diversity in it. For this reason a variation of the algorithm was created. The variant chooses randomly a number between zero and one. If the random number is less than a predetermined constant, for example 0.75, the the fittest individual is chosen to be a parent. Otherwise if the number is greater than 0.75 the weaker individual is chosen.

To help the genetic algorithm converge more quickly elitism is often used. In each epoch before the selection loop the most fit organisms of the current generation have a guaranteed slot in the next generation. This way it is certain that the solution obtained by the genetic algorithm will not decrease from one generation to the next. This selection technique enhances the standard methods of selection.

2.5 Scaling Techniques

Using selection on unprocessed fitness score can give a genetic algorithm that works. On the other hand if the scores are scaled in some way before the selection take place it will often lead to make the genetic algorithm perform better.

Rank Scaling can be a great way to prevent too quick convergence, particularly at the start of a run when it's common to see a very small percentage of individuals outperforming all the rest. The individuals in the population are simply ranked according to fitness. Then their new fitness score that is assigned to them is based on their rank as shown in Table [1].

This technique avoids the possibility that a large percentage of each new generation is being produced by a very small number of highly fit individuals, which can quickly lead to premature convergence. Rank scaling assures that the population remains diverse. Unfortunately the population may take a lot longer to converge, but this big diversity provided by this technique leads to a more successful solution to the problem that the genetic algorithm trying to solve.

Individual	Score	New Fitness
1	3.4	3
2	6.1	4
3	1.2	2
4	26.8	5
5	0.7	1

Table [1]
Fitness scores before and after ranking.

Sigma Scaling on the other hand tries to fix the problem in rank scaling. It attempts to keep the selection pressure constant over many generations. At the beginning of the genetic algorithm, when fitness scores can vary wildly, the fitter individuals will be allocated less expected offspring. Toward the end of the algorithm, when the fitness scores are becoming more similar, the individuals will be allocated more expected offspring.

The formula for calculating each new fitness score using sigma scaling is :

if $\sigma=0$ then fitness=1

else

$$\text{newFitness} = (\text{oldFitness} - \text{averageFitness}) / 2 * \sigma$$

where σ represents the standard deviation of the population.

Standard deviation is the square root of the population variance. The variance is a measure of spread within the fitness scores. Figure [5a] and [5b] shows a population with low and high variance. The

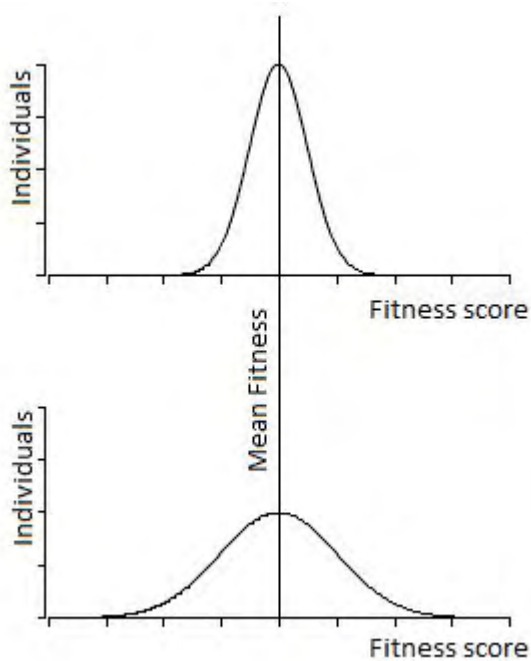


Figure [5a] and [5b]

5a shows a population with low spread.
 5b shows a population with high spread.

hump in the middle of the graph represents the average fitness score. Most of the population's scores are clustered around this hump. The variance is the width of the hump at the base.

To calculate the variance, first calculate the mean of all fitness scores.

$$mean = \frac{score1+score2+...+scoreN}{N}$$

Then the variance is given by the following mathematical formula.

$$variance = \frac{\sum_{x=1}^N (score_x - mean)^2}{N}$$

Where score_x is the fitness score of the current individual and N is the population size. Once the variance has been calculated, it's very easy to calculate the standard deviation.

$$\sigma = \sqrt{variance}$$

Instead of a constant selection pressure, sometimes the selection pressure must vary. A common scenario for this is one in which it is required in the beginning a low selection pressure to retain the diversity, but as the genetic algorithm converges closer to a solution, the fitter individuals must be mainly selected to produce offsprings.

2.6 Crossover

Crossover rate is simply the probability that two chosen chromosomes will swap their bits to produce two new offsprings. A typical value is around 0.7, although some problems may require much higher or lower values.

The simplest crossover operator is single point crossover. It simply cuts the genome at a random point and then switches the ends between the parents. It is very easy and quick to implement and is generally effective to some degree with most types of problems.

Parent1:	101000 1010
Parent2:	110111 0101
Child1:	1010000101
Child2:	1101111010

Table [2]
 Single point crossover.

Two point crossover cuts the genome in two random points instead of just one. Then the operator swaps the two blocks of genes between these two points. This way combinations of genomes are produced that the single point crossover is not able to produce.

Parent1:	101 000 1010
Parent2:	1101 11 0101
Child1:	101 111 1010
Child2:	110 000 0101

Table [3]

Two point crossover.

There is no reason to limit the amount of crossover point to two. Sometime, depending on the encoding of your algorithm's genome, it might be beneficial to use multiple crossover points. One easy way of implementing this Multi-point crossover is just to go to each position of the chromosome and randomly swap the genes based on the crossover rate. This process is also known as parametrized uniform crossover. The most common crossover rate value for this type of crossover are between 0.5 and 0.8. For some types of problems it works very well but on others it might mix the genes in a chaotic way.

Parent1:	101 000 1010
Parent2:	1101 11 0101
Child1:	1111101000
Child2:	1000010111

Table [4]

Multi point crossover.

The hard part about crossover is that it has to produce an offspring that is valid. Some problems, like the traveling salesman problem, each of the offsprings must have each different genes only once. In the traveling salesman problem the gene is encoded with the numbers of the cities. If the genome has the city number two twice and the city number one none, after the crossover, then the offspring is not valid.

To create valid offsprings on this type of problems partially mapped crossover algorithm is the most commonly used. Assuming an eight city problem, the algorithm first chooses two random crossover points. Then a mapping of the two center sections of the parents is created as shown in table [5]. The last step is to iterate through each parent's genes and swap the genes wherever a gene is found that matches one of those listed on the previous step, table [6].

Parent1:	2.5.0.3.6.1.4.7	The mapping
Parent2:	3.4.0.7.2.5.1.6	3 is mapped to 7
Parent1:	2.5.0.x 3.6.1 x.4.7	6 is mapped to 2
Parent2:	3.4.0.x 7.2.5 x.1.6	1 is mapped to 5

Table [5]

partially mapped crossover , splitting the parents and creating a mapping.

Step 1 [3 and 7]	Step 2 [6 and 2]	Step 3 [1 and 5]
Child1: 2.5.0.7.6.1.4.3	Child1: 6.5.0.7.2.1.4.3	Child1: 2.1.0.7.6.5.4.3
Child2: 7.4.0.3.2.5.1.6	Child2: 7.4.0.3.6.5.1.2	Child2: 7.4.0.3.2.1.5.6

Table [6]
partially mapped crossover , generating the offsprings.

Order based crossover is an other algorithm that spawns valid permutations. To perform this crossover several genes from the parent are chosen at random. After that the order of thoses genes is imposed ont he respective gene in the second parent. Table [7] demonstrates how this algorithm works. Assuming again an eight city traveling salesman problem and the random genes are the genes in the positions two, three and six.

Parent1: 2.5.0.3.6.1.4.7	Parent1: 2.5.0.3.6.1.4.7
Parent2: 3.4.0.7.2.5.1.6	Parent2: 3.4.0.7.2.5.1.6
Child1: 3.4.5.7.2.0.1.6	Child2: 2.4.0.3.6.1.5.7

Table [7]
Order based crossover.

Position-based crossover is very similar to order based crossover. Instead of the order of the genes this algorithm operates on the position of them. Using the sma example, with the same parents and the same random selected genes, the algorithm at first moves the selected genes from parent1 to offspring and keeping them in the same position. To fill the rest of the blanks it iterated through parent2 genes and fill them with genes that have not already appeared. Table [8] demonstrates how this algorithm works.

Parents	Offspring1	Offspring2
Parent1: 2.5.0.3.6.1.4.7	Offspring1: *.5.0.*.*.1.*.*	Offspring2: *.4.0.*.*.5.*.*
Parent2: 3.4.0.7.2.5.1.6	Offspring1: 3.5.0.4.7.1.2.6	Offsprin2: 2.4.0.3.6.5.1.7

Table [8]
Position-based crossover.

2.7 Mutation

Mutation rate is the probability that a gene within a chromosome will be altered. This rate is usually very small especially for binary encoded genes, typically 0.001. So after an offspring is produced mutation takes place by checking every single gene in the offspring's chromosome. Some problems does not have limitation on the mutation and the new value can be a random one within the constraints of the problem. But this is not the case in every single one of them, sometimes the mutation operator must produce valid offsprings. Again as an example is the traveling salesman problem.

The first algorithm is the exchange mutation operator. This algorithm produces valid children by choosing two random genes in a chromosome and swapping them.

5 . 3 . 2 . 1 . 7 . 4 . 0 . 6 \longrightarrow 5 . 4 . 2 . 1 . 7 . 3 . 0 . 6

Scramble mutation does also the same job but it chooses two random genes and "scramble" this genes with the genes between them to create the offspring

1 . 2 . 3 . 4 . 5 . 6 . 7 . 8 \longrightarrow 1 . 2 . 5 . 3 . 6 . 4 . 7 . 8

Displacement mutation selects two random genes and takes the chunk of the chromosome between them and reinsert it at a random position. This position can't be the original. Displacement mutation algorithm is quite interesting because it has a high converge rate at start and at the end it needs a lot more iterations to find a near best solution.

1 . 2 . 3 . 4 . 5 . 6 . 7 . 8 \longrightarrow 1 . 4 . 5 . 6 . 2 . 3 . 7 . 8

One of the most effective mutation is the insertion mutation. This is almost the same as displacement mutation but instead of a whole chunk of genes insertion mutation only selects one gene and displace it into the initial chromosome.

1 . 2 . 3 . 4 . 5 . 6 . 7 . 8 \longrightarrow 1 . 4 . 2 . 3 . 5 . 6 . 7 . 8

Inversion mutation is a very simple operator. It selects two random points in the chromosome and reverse the genes between them.

1 . 2 . 3 . 4 . 5 . 6 . 7 . 8 \longrightarrow 1 . 2 . 6 . 5 . 4 . 3 . 7 . 8

Displaced inversion mutation operator is a combination of Displacement mutation and inversion mutation. Selects two random points in the chromosome, reverses the order of the genes between this two points, and then displaces them somewhere along the length of the original chromosome.

1 . 2 . 3 . 4 . 5 . 6 . 7 . 8 \longrightarrow 1 . 6 . 5 . 4 . 2 . 3 . 7 . 8

2.8 Limitations

Genetic algorithms is a very good way to try to find the optimal way to solve a problem. Though using a genetic algorithm to figure out the most aerodynamic airplane you will never end up with a boat.

Very complex problems often have very expensive fitness functions. In real world problems a single function evaluation may take several hours. Furthermore they do not scale up well with complexity. When the number of elements which are exposed to mutations is large there is often an exponential increase in search space size. This makes it extremely difficult to use the technique on problems such as designing an engine or a plane. In order to make such problems tractable to evolutionary search, they must be broken down into the simplest representation possible.

Genetic algorithms gives solutions that are better to other solution that they might earlier have found. There might be times that finding a stop criteria might be extremely difficult.

One of their biggest limitations is that they tend to converge in local optima instead of global optimum of the problem. In other words this means that they don't know how to sacrifice short-term fitness to gain long-term fitness. Though the likelihood of this happening depends on the architecture of the fitness function itself. Different fitness functions, increasing the rate of mutation or by using other selection and crossover techniques might help the algorithm to find the global optimum. Furthermore big diversity in the initial population might also help.

Moreover they are not indicated for problems with dynamic data sets. The genomes will start to converge towards a solution which may no longer be valid for later data. Several methods has been proposed to make genetic algorithms more effective with dynamic data. Some of them is triggered hypermutation, increased mutation rate when the solution quality drops, and random immigrants, which is a method of generating and adding random individuals in the population pool.

Genetic algorithms cannot effectively solve problems which their only fitness measure is right or wrong. They cannot make decision as there is no actual problem that the algorithm can converge to. However if what we are looking for is a ratio of successes to failures then this is a problem suitable for fitness measure.

Lastly, for specific optimization problems, other optimization algorithms might be more efficient than a genetic algorithm in terms of speed of convergence. For example an ant colony optimization algorithm solution to the traveling salesman problem will have advantages over a genetic algorithm approach. In general the suitability of a genetic algorithm is dependent on the amount of knowledge of the problem; well known problems often have more specialized approaches.

3. Artificial Neural Networks

3.1 Introduction

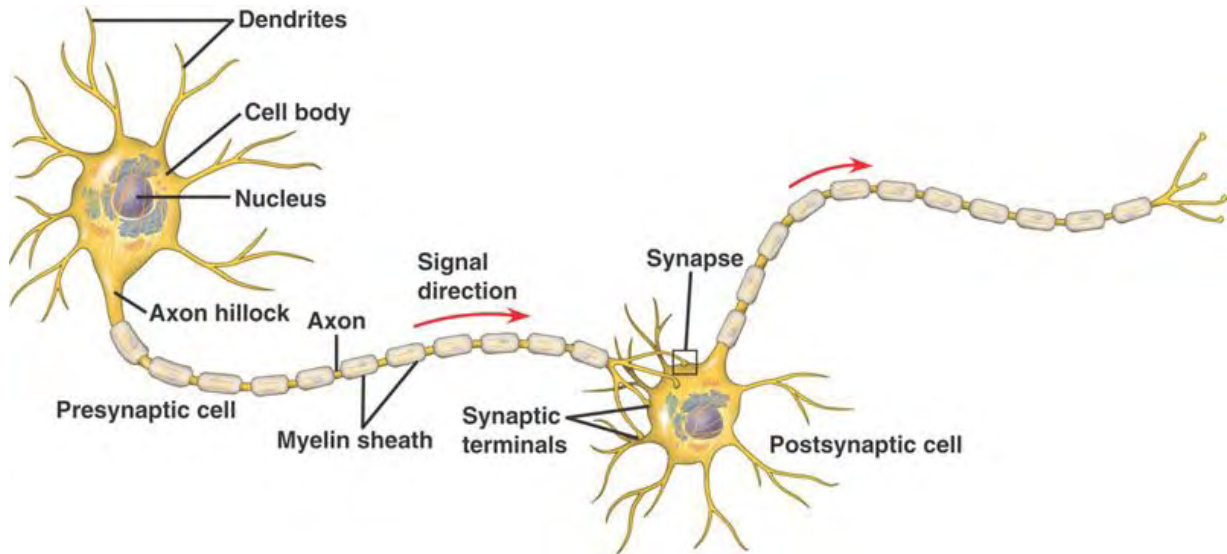


Figure [6] [source](#)
two connected neurons

An artificial neural network attempts to mimic the way the biological brain works. The brain consists of a very large amount of neurons. The neuron consists of several parts. As shown in figure 1 the neurons are connected with each other Axon to Dendrite. This connection is called Synapse. When the Axon is stimulated it transfers to the synaptic gap between the Axon and the Dendrite some of its vesicles. The dendrites transfer this information to the Soma (cell body) of the neuron. When the stimulus is enough the second neuron transfer through its axon this information to the next neuron.

The input of the neurons is binary, they either are stimulated or not. Each neuron might have lots of synapses with different neurons. Each synapse has each own “weight”. This weight is multiplied by the weight giving a result to the neuron. The neuron has the ability to collect this results by summing them up. If the sum is greater than the threshold then the neuron is stimulated.

The artificial neural networks are inspired by this model.

3.2 Artificial neural network

The beginning of the artificial neural networks have started in the late 1800s with scientific attempts to study how a human brain works. In 1890, William James published the first work about brain activity patterns. In 1943, McCulloch and Pitts produced a model of the neuron that is still used today. This model can be divided into two parts: a summation over weighted inputs and an output function of the sum.

Donald Hebb, in 1949, in his publication “The organization of behavior” outlined a law for synaptic neuron learning. In his honor, this law, was named after him (Hebbian learning). It is one of the most simple and straightforward learning rules for artificial neural networks.

In 1958 the perceptron algorithm was created by Frank Rosenblatt.

3.3 The Perceptron

Perceptron is an algorithm for supervised learning for binary classifiers. Binary classifiers are functions that can decide if an input belongs in a specific class or not. The perceptron is a linear classifier and is useful for solving problems where the input classes were linearly separable in the input space. The perceptron function:

$$f(x) = 1 \text{ if } w * x + b > 0 \text{ and } f(x) = 0 \text{ otherwise}$$

Where x is a vector of values, w is a vector of weights, $w*x$ is the dot product of these two vectors and b is the bias, sometimes referred as threshold.

Perceptron is a learning algorithm. It starts with a random bias and tries to find the correct one for its given problem. The problem has to start with a set of training data. This data will be processed by the perceptron function one by one and spatially the bias will alter to become the decision boundary of the problem. If the training data are not linearly separable the algorithm will never terminate.

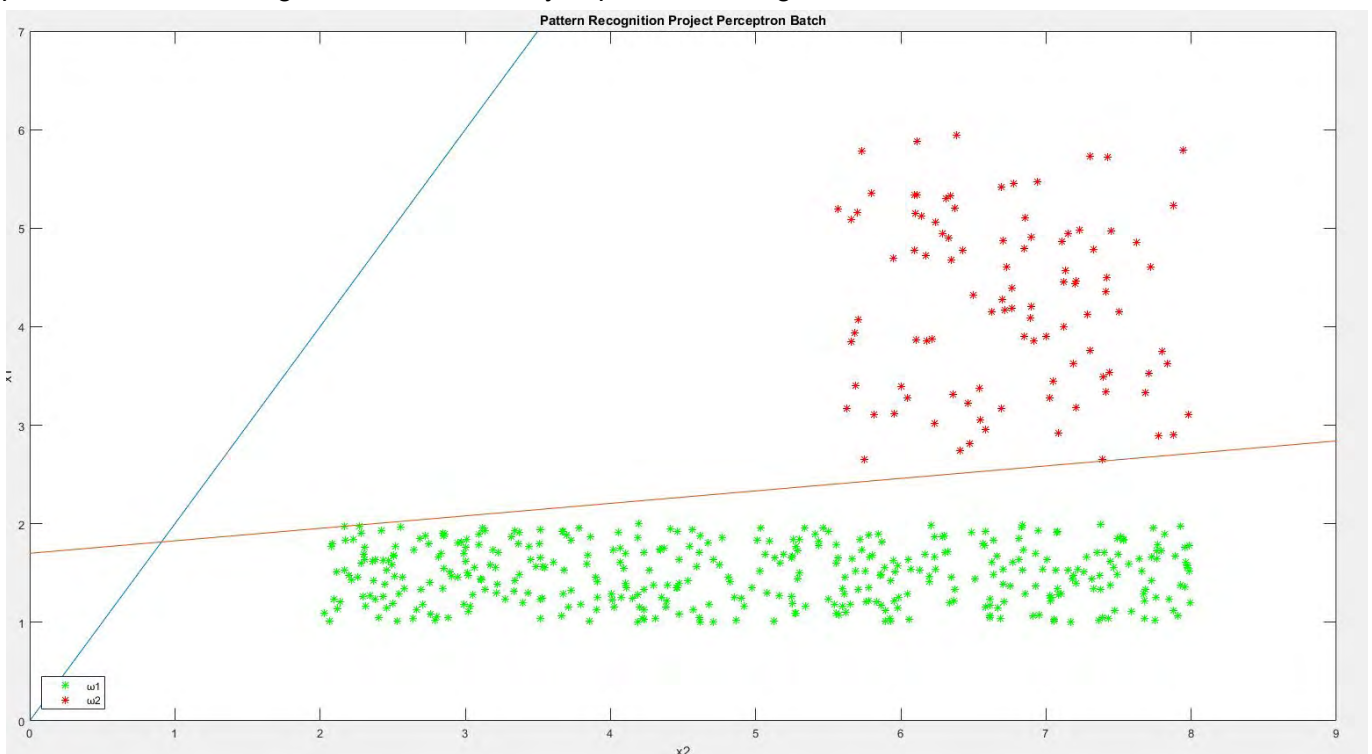


Figure [7]

Two classes of points linearly separated by perceptron.

After the algorithm has terminated, it can predict if a given input belongs to a class or not. The steps of the algorithm in a better visualization:

1. initialize the weights and the threshold. It can be initialized to 0 or to a small random value.
2. for each example in our training set perform the following steps:
 - a. Calculate the actual output
 - b. update the weights

$$w(t+1) = w(t) + (d - y(t)) * x$$

where d is the desired output, x is the input and $y(t)$ is the output of 2a.

It wasn't until 1990 when Hecht-Nielsen showed that multilayer perceptrons, perceptrons that give their output to another perceptron as an input, can solve nonlinear problems. But to do so they require more sophisticated learning algorithms such as the backpropagation.

3.4 Activation Function

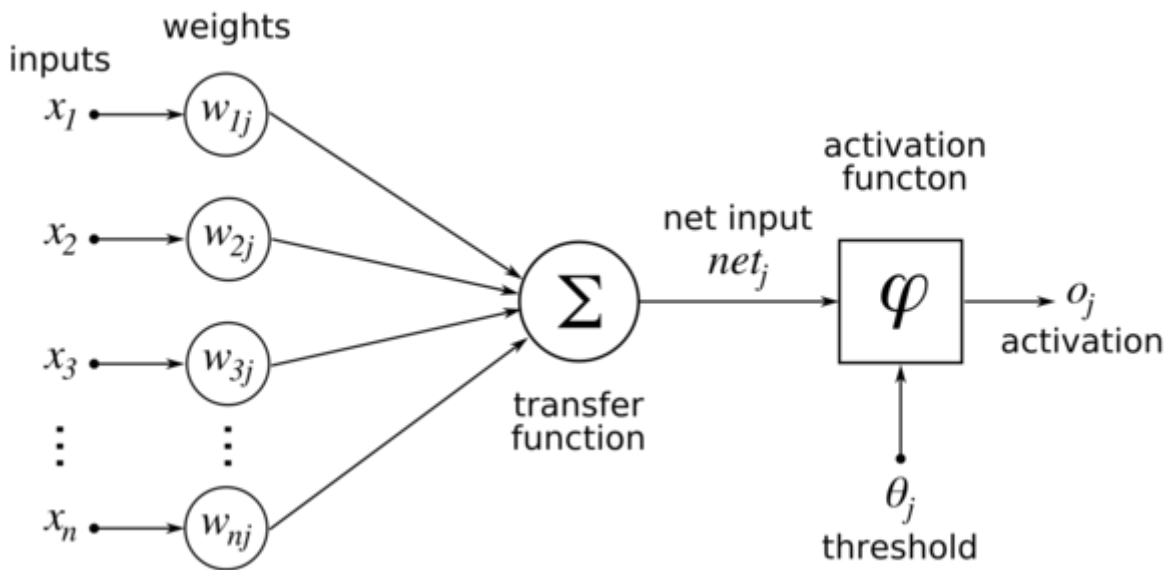


figure 8 [\[source\]](#)
An artificial neuron.

The output of an artificial neuron is defined by an activation function. There are activation functions that works like switches, “fire” or not, and are called linear functions. However, it is the nonlinear functions that allows neural networks to compute answers for nontrivial problems using only a small number of neurons. This function is also called the transfer function in artificial neural networks.

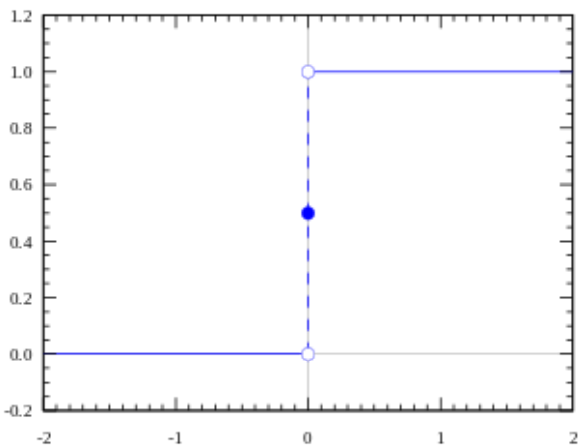


Figure [9]
Step activation function.

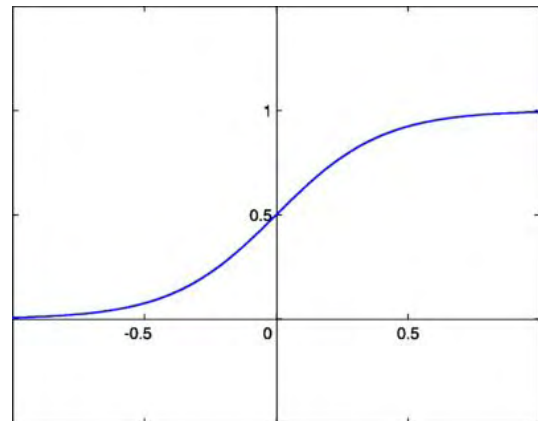


Figure [10]
Sigmoid activation function.

The step function was originally used by perceptron. The output is a certain value, A_1 , if the input sum is above a certain treshold and A_0 if the input sum is below a certain threshold. The values use by perceptron is $A_1 = 1$ and $A_0 = 0$. This step functions are useful for binary classification problems.

One of the most common and useful activation functions is the log-sigmoid function, also known as a logistic function, is given by the relationship:

$$\sigma(t) = \frac{1}{1+e^{-\beta t}}$$

Where the β is a slope parameter.

Commonly the log-sigmoid function is just called sigmoid. However you can construct a sigmoid function using hyperbolic tangent function instead of this relation, in which case it would be called a tan-sigmoid. The sigmoid function has the property of being very similar to the step function, but with the addition of a region of uncertainty. Sigmoid functions are very similar to the input-output relationships of biological neurons, although not exactly the same.

One of the main reasons that sigmoid function is very commonly used is because of its easiness to calculate its derivative, which helps for calculating the weight updates in certain training algorithms. The derivative when $\beta=1$ is given by:

When $\beta \neq 1$, the derivative is given by:

$$\frac{d\sigma(\beta, t)}{dt} = \beta[\sigma(\beta, t)[1 - \sigma(\beta, t)]]$$

Lastly, softmax function is often used in the final layer of a neural network classifier. This function is a generalization of the logistic function that “squashes” the input values in the range [0,1]. The function is given by:

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}}$$

Where K is the set of neurons in the output layer.

3.5 Feed Forward Network

The first and the simplest type of artificial neural networks was the feedforward ones. A feed forward network does not contain any cycle between the units that consists it. Therefore the information moves in only one direction, forward, from input nodes to hidden nodes if there are any and to output nodes.

The simplest feedforward Network is a single layer perceptron as shown in figure 3. It consists of a single layer of output node. The inputs are fed directly to the outputs through a series of weights. Multiple perceptrons grouped in layers create a neural network.

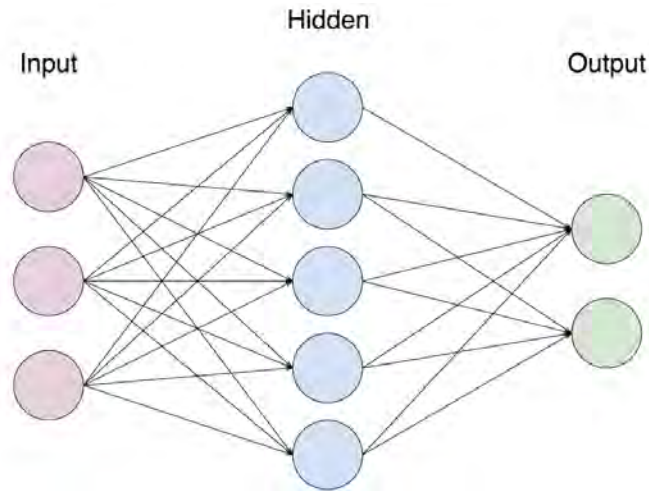


Figure [11]
A feed-forward neural network.

In a feed forward network the weights from a neuron to itself is zero. The same is true for the weight of a neuron for a neuron of the previous layer. A feed forward network where every node from the first layer connects with a non zero weight to each and every node in the next layer is called fully connected. Otherwise it is called either sparsely connected or non-fully connected network. The percentage of available connections that are not zero is also known as connectivity of the network.

3.6 Recurrent Networks

In a recurrent neural network, the weight matrix for each layer contains input weights from all other neurons in the network, not just neurons from the previous layer. This additional complexity from these feedback paths can have a number of advantages and disadvantages in the network.

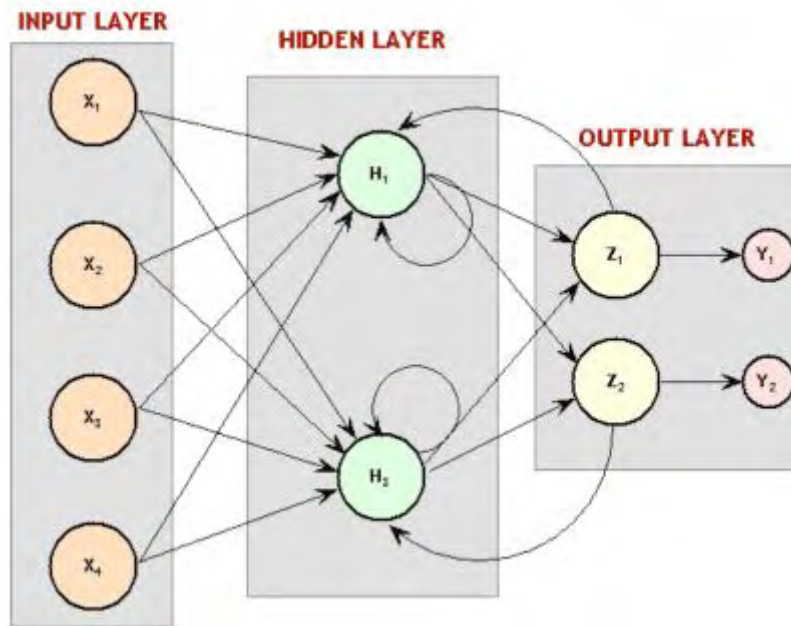


figure [12] [\[source\]](#)

A simple recurrent neural network example.

A way to think about recurrent neural networks is that they have a “memory” which captures information about what has been already calculated. These networks are very commonly used these days in natural language processing problems, it’s easier to predict the next word in a sentence if you know which words came before it. One more thing is that if it is needed to be taken into account a sequence of 5 words then at least a 5 layer neural net is needed. The hidden layers might be considered as the memory of the network.

3.7 How the networks learn

There are three major abstract learning ways to train an artificial neural network. These are supervised learning, unsupervised learning and reinforcement learning.

In Supervised learning, a set of training data, consisted with both the input and the output, is processed while the neural net is trying to find a function, or class of functions, that matches the examples. Commonly used cost function is the mean-squared error, gradient descent and backpropagation algorithms tries to minimize the average squared error between the network's output and the target values over all the example pairs. Supervised learning is used to problems in the field of pattern recognition and regression.

Unsupervised learning is used in situations where the cost function is known but there is no data set that minimizes that cost function over a particular input space. For this reason the unsupervised networks begins with a cost function and the input data. The artificial neural network tries to find a suitable input-output relationship that minimize the cost function.

Lastly in reinforcement learning the input data is not given in a training set, but rather generated by an agent's interaction with the environment. At each point in time t , the agent performs an action y_t , which is the output of the behaviour, and the environment generetes an observation x_t and an instantaneous cost c_t , according to some dynamics that are usually unknow. The aim is to discover a policy for selecting actions that minimizes some measure of a long-term cost. There are numerous algorithms available for training neural network models; most of them can be viewed as a straightforward application of optimization theory and statistical estimation. Reinforcement learning, due to its generality, is studied in many other disciplines, such as game theory, simulation based optimization, multi agent system, swarm intelligence, statistics, genetic algorithms and others.

4. Neuroevolution

4.1 Introduction

After seeing that the artificial neural networks are inspired by biology it is easily assumed that we can combine them with an evolutionary algorithm. This combination is called neuroevolution and belongs to the reinforcement learning algorithms. They require minimal information which most of the time might be only the network's performance at a task. For example, the outcome of a game, whether the player won or lost.

There are many ways to classify this algorithms but two main approaches exist. The first one divides the algorithms in two main categories. The algorithms are either able to evolve only the connection weights of a fixed network topology, often called conventional neuroevolution, or they are able to evolve both the topology of the network and its weights, often referred as TWEANNs which stands for Topology and Weight Evolving Artificial Neural Network.

The other approach is based on the most difficult part in a neuroevolution algorithm. This part is the representation of the genotype. The choice of representation is crucial, it determines what kind of crossover and mutation can be applied. Three main categories of representations are given from Floreano: 1) direct , 2) developmental and 3) Indirect or implicit.

4.2 Direct representations.

Direct representations have 1-to-1 mapping between the genotype and the phenotype. This means that every part of the genome can be translated into a specific part of the neural network. It is easier to understand how the network is constructed but as the networks are getting bigger the genotype gets bigger in a similar amount, this means a lot more computation time. Most direct representations have a fixed neural network topology. Given that said is easily understandable that the only task for the evolutionary algorithm will be to find the best weights that optimizes the network to solve the problem.

4.2.1 Conventional direct representation.

Weight vectors was one of the earliest attempt to represent the genotype of a fixed topology. It was literally a vector of all the connection weights of the neural network. This way of representations are usually referred as conventional neuroevolution methods because these were the first successful attempts at neuroevolution. Mutation was mostly done by letting every weight have a small chance of changing between generations. Crossover could be done by swapping parts of the vector. The early tries of weight vectors had a large chance of premature convergence. A lot of word have been done since then, mainly trying to keep the population diversity high. The fixed topology of weight vectors methodology is quite restricting but in some cases might perform better.

4.2.2 Symbiotic, Adaptive Neuro Evolution.

Moriarty and Miikkulainen came up with an algorithm to improve the diversity of conventional neuroevolution methods. Instead of using a vector representation of an entire network, it evolves a population of individual neurons. They called this method Symbiotic, Adaptive Neuro Evolution. Each neuron has an weighted connection with the input and the output layer in addition to the vector of its weights for the hidden layer. The fitness of the neuron is determined by its average performance in a sample of random networks. The random networks are formed by randomly selecting a subset of the population of neurons. The network is evaluated on the target task and the neurons will be awarded with the fitness of the network that will be divided evenly among them. Neurons will be then ranked according to their average fitness and the top, usually, 25% will be used for crossover, usually one-point crossover method. Mutation would change the individual weights with a small probability.

4.2.3 Neuro Evolution of Augmented Topologies.

One of the most successful direct representation is the Neuroevolution of Augmented Topologies. It has been developed by Stanley and Miikkulainen at the University of Texas in 2002. The algorithm alters both the weighting parameters and the structures of the network, attempting to find a balance between the fitness of evolved solutions and their diversity thus it is a TWEANN algorithm.

As the developers claim this is succeed due to three key characteristics: (1) employing a principled method of crossover of different topologies, (2) protecting structural innovation using speciation, and (3) incrementally growing from minimal structure.[\[Reference\]](#)

The initial population in NEAT starts with no hidden nodes. Even though random initial population that consists out of random topologies may ensure a high starting diversity, it often lead to produce many problems for TWEANNs. The most serious one is that this evolved solutions will not be minimal. The population starts with many unnecessary nodes and connections which they haven't withstand any evaluation to justify their existence. NEAT starts minimally and searches for the solution in the lowest-dimensional weight space possible of all generations. The hard part in starting minimal is that topological innovations would not survive.

The representation that NEAT uses has each genome encoded as a list of connections. Each connection specifies the in-node, the out-node, the connection weight, whether or not the connection is enabled and an innovation number, the last one will be explained below.

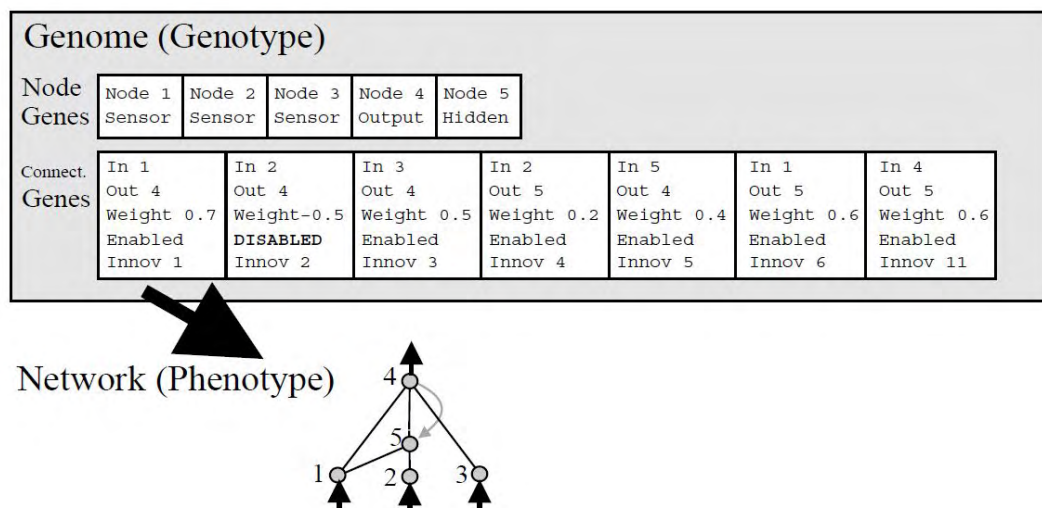


Figure [13]

The genotype and the phenotype of NEAT representation.

Source: Evolving Neural Networks through Augmenting Topologies

Mutation in NEAT can change both the connection weights and the network structures. Connection weights mutate as in any NE system. Topology mutation occurs in two ways as shown in figure 2. In the add connection mutation, a single new connection gene with a random weight is added connecting two previously unconnected nodes. In the add node mutation, an existing connection is split and a node is placed where the old connection was used to be. The old connection is disabled and two new connections are added to the genome. The new connection leading to the new node receives a weight of 1, and the new connection leading out receives a weight as the old connection.

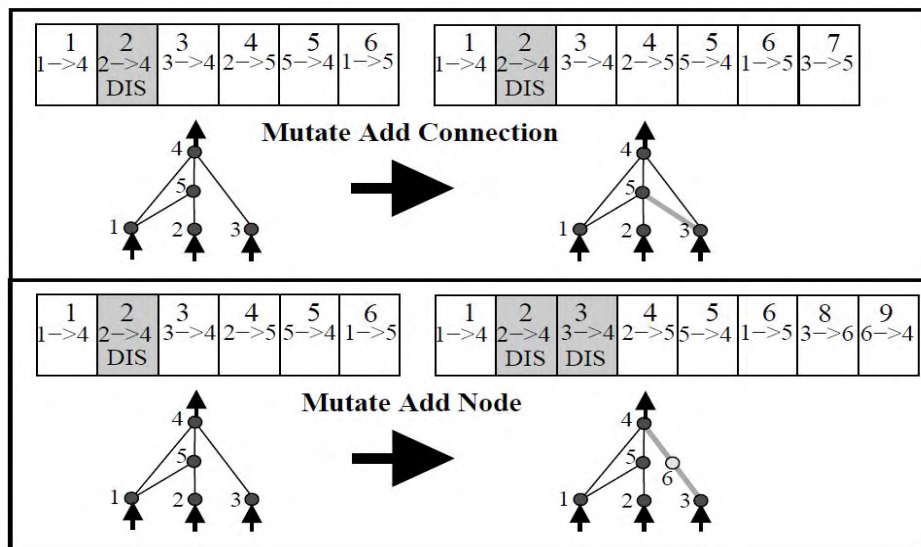


Figure [14]

Topology mutation operators in NEAT.

Source: Evolving Neural Networks through Augmenting Topologies

Historical markings are used in NEAT to clarify if two genes match up between any individuals in a topologically diverse population. To do so a global innovation number is needed and it will be incremented and assigned to the gene. This innovation number represents a chronology of the appearance of every gene in the system. Given that said it is easily assumed that in an add connection mutation the global innovation will be increased once and in add node mutation twice. There is a need of keeping a list of the innovations that occurred in the current generation to avoid giving the same structural innovation the same number. With the innovation number of each gene will be used in the crossover operation to find out which genes are matching (even with different weights), and also to find either the disjoint or the excess ones.

The crossover operation between two parents will start by defining the disjoint and the excess genes. The next step is to combine the overlapping parts of the two parents as well as their different parts. Matching genes are inherited randomly, whereas disjoint genes and excess genes are inherited from the more fit parent. This way the system can form a population with diverse topologies. However, as Stanley and Miikkulainen showed, such population are unable to maintain topological innovations because smaller structures optimize faster than larger structures. The solution that they suggested is to protect the innovation by speciating the population.

The idea of speciation is to divide the population into species in a way that similar topologies are in the same species. Again the historical marking offers an efficient solution. The number of excess (E) and disjoint (D) genes as well as the average weight difference between the matching genes (W) in a pair of genes (W) can give the compatibility distance δ .

$$\delta = \frac{c_1 * E}{N} + \frac{c_2 * D}{N} + c_3 * W$$

The coefficients adjust the importance of the three factors, and the number of genes in the larger genome N , normalizes the distance. A threshold δ_t is also needed to show whether a gene belongs to a species or not. If it is not compatible with any existing species then a new species is created.

With explicit fitness sharing all organisms in the same species must share the fitness of their species. Thus, a species cannot afford to become too big even if many of its organisms perform well. Therefore no species is likely to take over the entire population. The adjusted fitness is calculated by dividing the actual fitness by the sum of the distances between this gene and every other gene in the species. In the end the reproduction begins with the elimination of the lowest performing individuals of the species. Then the entire population is replaced by the offsprings of the remaining organisms in each species. This way the topological innovations are protected so they can evolve and be more optimized. Lastly if the maximum fitness of a species does not improve significantly after a number of generations then the genes of this species are not allowed to reproduce.

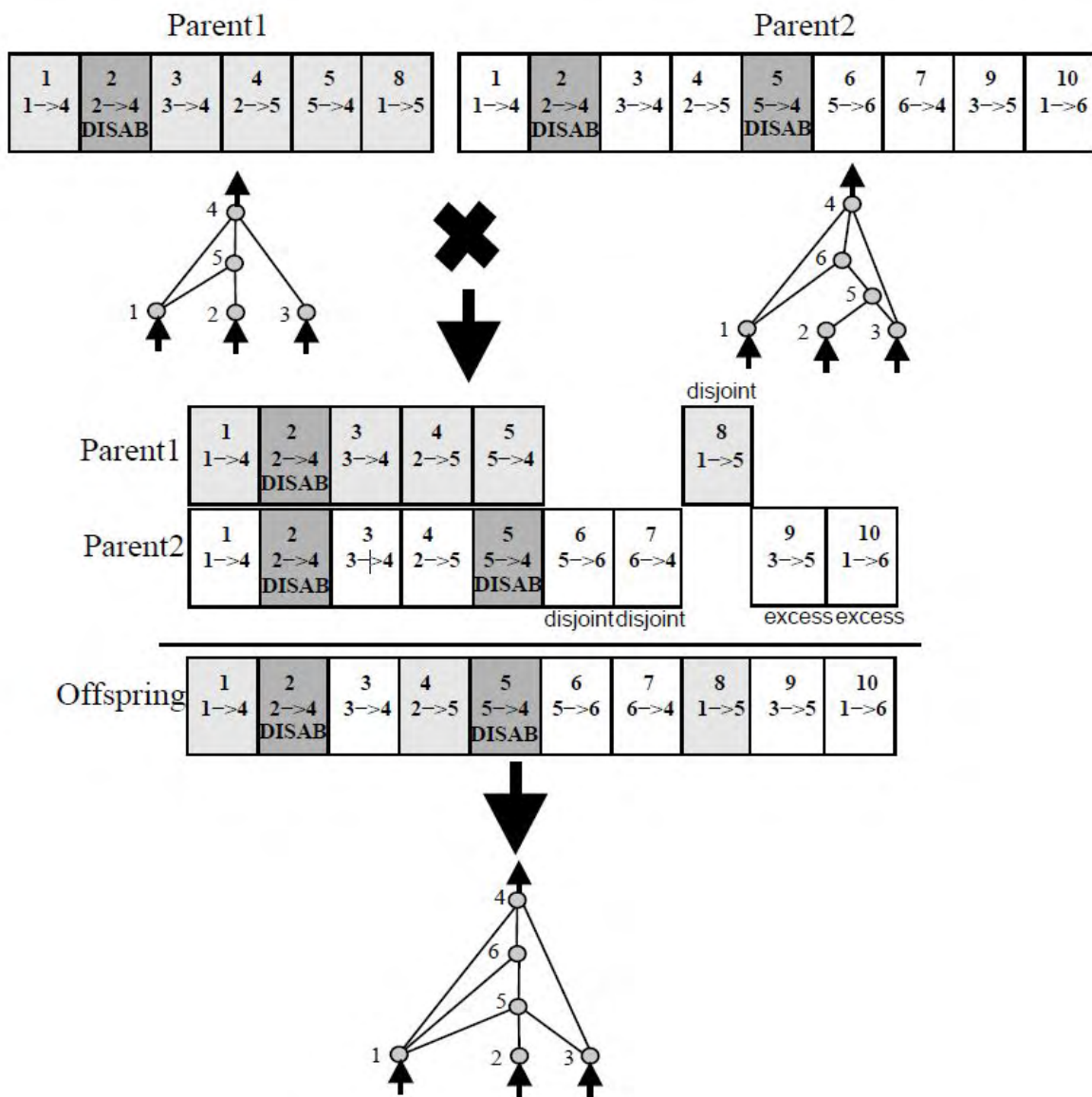


Figure [15]

Showing the innovation number of each connection and the crossover operation between two parents with equal fitness.
Source: Evolving Neural Networks through Augmenting Topologies

4.3 Developmental representations.

The second category of representations is the developmental one and its main reason of existence is the problem of direct representations of dealing with larger networks. To do so an assumption must be made that larger neural networks are consisted by smaller modules. The evolutionary algorithms can exploit this modularity of the assumption. This assumption was made by Gruau who also developed Cellular Encoding. It uses a symbolic expression, *s-expressions*, to represent the nodes of the network. Each node is divided into two child nodes and a specific rule that describes how the links between the new nodes and the existing would be established.

Kitano developed a developmental representation that consist of evolvable and fixed rules. He used a connectivity matrix consisting of 1's and 0's. Instead of encoding the entire matrix, the representation was in the form of 2x2 matrices of symbols and some roles which specified how to develop the entire matrix. Recursively, each symbol in a 2x2 matrix of symbols would develop into its own 2x2 matrix until a point is reached where each 2x2 matrix is a predetermined matrix consisting of 1's and 0's as shown in figure 4.

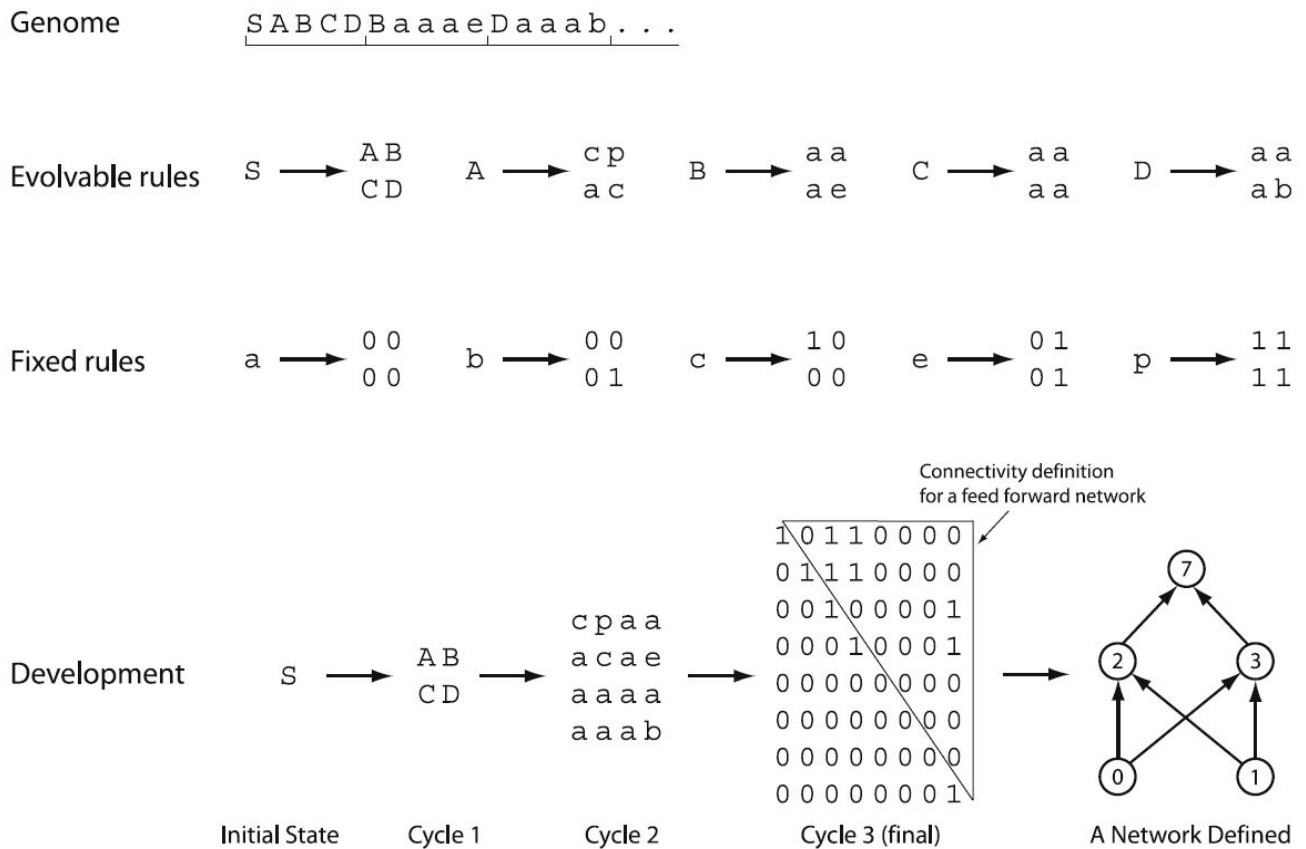


Figure [16]
Kitano's grammar encoding.
Source: Neuroevolution: From architecture to learning

4.4 Indirect representations.

The third and the last category of representations is the indirect ones. This is the most recent approach and they are directly inspired by the biological genes and DNA, especially from the gene regulatory network mechanism. With this mechanism biological genes can be represented as a sequence of characters that can be divided into regions, such as the coding, the promoter and the terminator region. One of the most successful approach so far is the Analog Genetic Encoding.

The AGE representation is a sequence of characters from an alphabet. Sequences of characters are predefined as device and terminal tokens that signal the start and the end of a coding region. The representation of each node is encoded in the following way: Device Token - Sequence of characters that indicate the input weights of the node - Terminal Token - Sequence of characters that indicate the output weights of the node - Terminal Token. To decode this representation an interaction map is needed. With this map the actual weight is calculated by combining the input and the output characters of the two nodes connected by this weight.

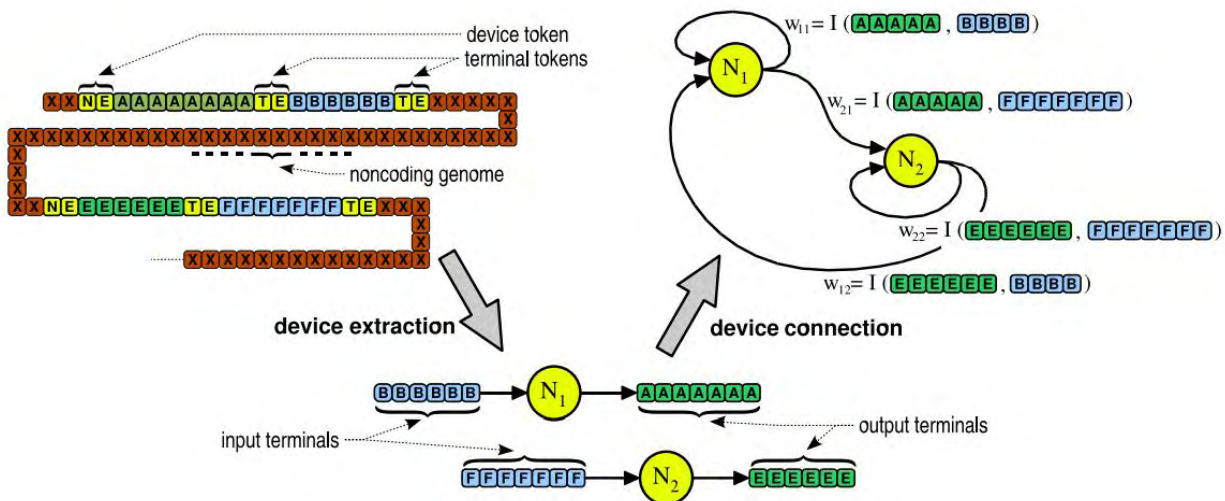


figure [17]

The AGE representation of both the genotype and the phenotype.
Source: Neuroevolution with Analog Genetic Encoding.

	A	B	C	D	E	F	G	...	U	V	W	X	Y	Z
A	5	2	1	0	-1	-2	-5	...	-5	-2	-1	0	1	2
B	2	5	2	1	0	-1	-2	...	-5	-5	-2	-1	0	1
⋮														

and

	A	...	Z
-	-3	...	-3

figure [18]

The interaction map of the AGE representation.
Source: Neuroevolution with Analog Genetic Encoding.

The most fascinating thing about AGE is that if a device token is not followed by enough terminal tokens then that part of the genome is considered invalid. This way mutation and crossover operators can be applied to the whole genome without a need of protection. The initial population is created by generating individuals with a given number of neurons and with random terminal tokens.

4.5 Real-time Evolution.

In most video games the behavior of the non-playing characters(NPCs) is usually scripted, so the player can exploit a weakness for many times but this weakness will never be repaired. A way to keep the game interesting is to make the NPCs learn from the interaction with the player. This way the behaviour of the NPCs will improve as the game is played. In short this learning mechanism can be achieved by simply having a pool of the whole population stored and sorted. Individuals used in the game are spawned from this pool. Immediately after an individual is killed, it is replaced by mutating one of the better performers in the population. This way the population doesn't evolve in waves as an epoch is processed, but rather, continuously. This technique requires fast turnaround of game agents in order to work properly. If the NPCs dies too slow, then it is unlikely they will evolve at a satisfactory pace.

This real-time evolution is easily applied to NEAT to create real time NEAT or rtNEAT, but there are some minor changes that has to take place as listed in Real-Time Neuroevolution in the NERO Video Game . Instead of having the whole population tested and evaluated to form the next generation, every few frames two high fitness agents are selected to produce an offspring that will replace another lower fitness one. The individual that is being removed must be chosen carefully to preserve speciation and of course there has to be a minimum amount of time that the individual have played before evaluated and removed. Replacing an old agent with a new one doesn't necessary means that the body has to be replaced also, sometimes you can just alter the way it thinks and responds to stimuli.

The last step is to find the satisfactory pace of agent evolution. Let I be the fraction of the population that is too young and therefore cannot be replace. Then I is calculated by having the minimum time, m , that an agent is alive between all alive agents divided by the population size, $|P|$, times the number of ticks, n , between replacements. This means that if the portion of the population that is ineligible at any given time is known, the time between two replacements can be calculated.

$$I = \frac{m}{|P|*n} \Leftrightarrow n = \frac{m}{|P|*I}$$

5. Implementation

5.1 Introduction

Evolutionary techniques are very important in the modern life and they are very widely used. These algorithms are very powerful in real world application problems, they can control and evolve physical devices. The evolution is on the phenotype of the device. Given some constraints they will produce a very close to optimum design for the given task. On the other hand control of devices includes evolution of controllers that are capable of driving mobile robots, automobiles and even rockets.

Furthermore evolutionary algorithms are very good to simulate artificial life or any other kind of simulation. They can simulate from simple hunting and mating behaviors to more complex ones, such as the investigation of what conditions are necessary for a certain behavior to evolve. There has been done a lot of work in this domain and many experiments has shown how behaviors like foraging, pursuit and evasion, hunting and herding, collaboration and even communication may emerge in response to environmental changes that pressures the individuals.

Last but not least evolutionary algorithms has a vast domain of application in games. Other than the most obvious application of adaptable and evolutionary enemy behaviours that enhances the experience of the player, they can be also used as a game tester while the game is still on the development. This tester can evolve a plan of play to beat the game with the best available way. The result of such a plan will so if something in the game is completely out of balance since the evolutionary algorithm will exploit it. Moreover neuroevolution has been used to evolve the player's weapons. This way content is created by the player and not by the developer and the player usually likes the uniqueness of his intelligent weapon.

5.2 Genetic algorithm for movement.

The thing that the first genetic algorithm is trying to achieve is to move some agents form a starting point to the finish line. The agents are learning to travel a small distance over some time. It is a very simple but fundamental approach to get the basic feeling of a genetic algorithm.

5.2.1 Genome representaion.

The genome is represented by a list of smaller nodes that contains the acceleration of the agent for a specific amount of time. The time is measured in frames. The given acceleration is in $[-8,8]$ units of force.

Due to the physics engine it must be noted that each of the agents has a mass. Thus any force in the space between $[-4,4]$ it will result to no movement at all. Forces in the space $(4,8]$ will result to an acceleration of the body of the agent to the right. On the other hand forces in $[-8,-4)$ will accelerate the body to the left of the screen.

There is a specific amount of genes for each agent that accelerate the agent's body for a specific number of frames.

5.2.2 Mutation

To mutate a gene of the agent's genome the acceleration value of the gene is replaced with a new random value between $[-8,8]$.

5.2.3 Crossover and Parent Selection

Single point crossover as described previously is used in this simple problem. The selection is done with the implementation of a simple roulette wheel selection algorithm.

5.2.4 Fitness function

The fitness of each agent is calculated using three different positions, the starting position (SP), the current position (CP) that the agent stopped moving and the desired position (DP) that the agent must reach. The fitness can be easily described with 3 different equations.

- 1) $fitness = 0$ if $CP > SP$.
- 2) $fitness = 10000$ if $\|CP - DP\| < 0.4$.
- 3) $fitness = \frac{100}{(\|CP - DP\|)^2}$ elsewhere.

5.2.5 Results

After some testing the standard conversion is between the generations five to twelve using a mutation rate of 5%. The model will converge even with very small populations.

Even though the bots learn to travel the given distance within the given time due to the nature of the problem and the fitness function model there are two behaviours that are not desired. The first one is that sometimes the agent goes out of the borders, before the starting point or after the desired point, this might mean that the agent might not be rendered for a small period of time. The second one is that sometimes might reach the desired position in a shorter period of time and stay there without moving for a significant period of time.

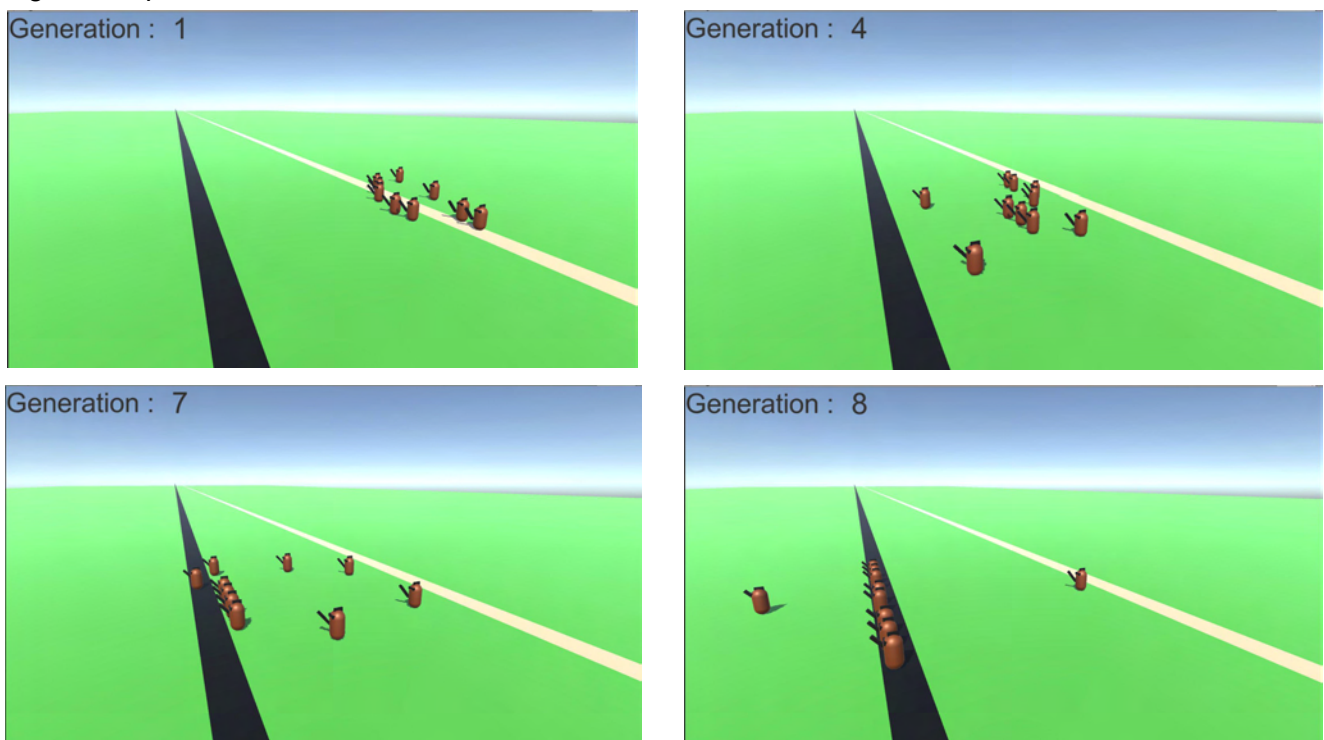


Figure [19-22]

Screenshots from the simulation of the genetic algorithm that learns to the agents how to travel from one point to another.

5.3 Improving and enhancing the simulation.

The second approach to the problem of traveling from one point to another includes two things. First there is a need to fix the problems that have emerged from the first approach. And secondly to add agents that can jump or even fly. For the first problem both the genome and the fitness functions must be changed. For the second there must be added more functionality to the genes of each agent.

5.3.1 Genome Representation.

The genome is represented as a list of nodes that now consists from two parts in the simple moving agent, walkers, and three on the ones that jump or fly. The first part is still the vertical acceleration. For the walkers is still the same constraints as explained on the first approach. The jumpers have a random number between $[-1.75, 1.75]$ and the fliers $[-1, 1]$. Note that due to the fact that there is no friction while they are on the air there is no dead space that they can not accelerate.

The second parameter of the gene is the time that the acceleration is applied. This will help agents evolve to reach their target in a specific amount of time. For the walkers the time constraints are $[1, 30]$, for the jumpers $[50, 120]$ and for the fliers $[6, 9]$. Time is measured in frames.

The third and final parameter of the gene is the vertical force that will be applied on the body to make it take off the ground. This force is applied once in the first frame of the gene. For the jumpers the minimum and the maximum vertical force is between $[0, 450]$ and for the fliers $[65, 70]$.

5.3.2 Crossover and Selection.

Single point crossover is used and the selection is done with the implementation of a simple roulette wheel selection algorithm.

5.3.3 Mutation.

Mutation might be happening separately on each of the parameters of each gene. This means that in most cases only one of the parameters will change but there is a possibility that the others will also change. This change is by choosing a random number within the constraints of the parameter given in the genome representation section. The mutation rate for the walker is 5%, for the jumper is 2% and for the flier is 5%.


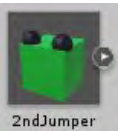
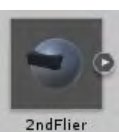
	Horizontal Acceleration	Frames	Vertical Acceleration	Mutation Rate
	$[-8, 8]$	$[1, 30]$	$[-, -]$	5%
	$[-1.75, 1.75]$	$[50, 120]$	$[0, 450]$	2%
	$[-1, 1]$	$[6, 9]$	$[65, 70]$	5%

Table [9]
Constraints and information of each different agent's genome.

5.3.4 Fitness functions.

The Fitness function can be divided into three separate ones for the walker and the jumper and in four for the flier. The first part of the function is quite similar to the initial fitness function that was used on the first approach for all three different agents. The adjustments are made so there are better results.

1. $fitness += 0$ if $CP > SP$.
2. $fitness += 900$ if $\|CP - DP\| < 0.4$.
3. $fitness += \frac{100}{(\|CP - DP\|)^2}$ elsewhere .

The second part is the one that helps the agents learn to stay between the borders. This is simple done by awarding the agent for each frame that the agent is after the starting point and before the desired destination point.

4. $fitness += A$ if $CP < SP$ and $CP > DP$. A equals 0.005 for all the species.

The third part is for the time of arrival. This fitness is rewarded only the first time the agent crosses the desired destination point.

5. $fitness += B * [(Fs * DT)^2 - (t - DT * Fs)^2]$ B equals 0.003 for all species.

Where Fs stands for Frames per second and it is 50, DT is the desired time of arrival to the desired destination and t is the actual time of arrival.

Lastly the fliers has a fourth part that alters the fitness and it deals with the height constraints. It works as the second part of the equation but for the height instead of the horizontal distance.

6. $fitness += C$ if $3 \leq CH \leq 5.5$, whete CH stands for current height.

5.3.5 Results.

For the walkers the generation that they start to converge to find a solution is shifted between the fifteenth and twentieth generation. It starts to arrive with small deviation at the desired point at the desired time of arrival usually after the thirtieth generation.

The jumpers usually perform very well. They usually arrive at the desired destination within the first ten to twelve generations but they usually need more to find the sweet spot of timed arrival but they do after the thirty fifth generation.

As for the fliers they don't perform as well as the others. Even though they try to float around the air it is very difficult for them to stay within the height limits. Other than that they usually find their way to the desired position with one way or the other at around the tenth generation, but they arrive with small time deviation at about the thirty fifth generatio. Lasty they find their way floating around in desired height at generation forty five or so.

For this simulation i have divided my screen in four pieces. The top left shows the training of the walkers. The top right show the training of the fliers. The bottom left shows the training of the jumpers. Lastly the bottom right shows the best individual of the previous generation of all the three species.

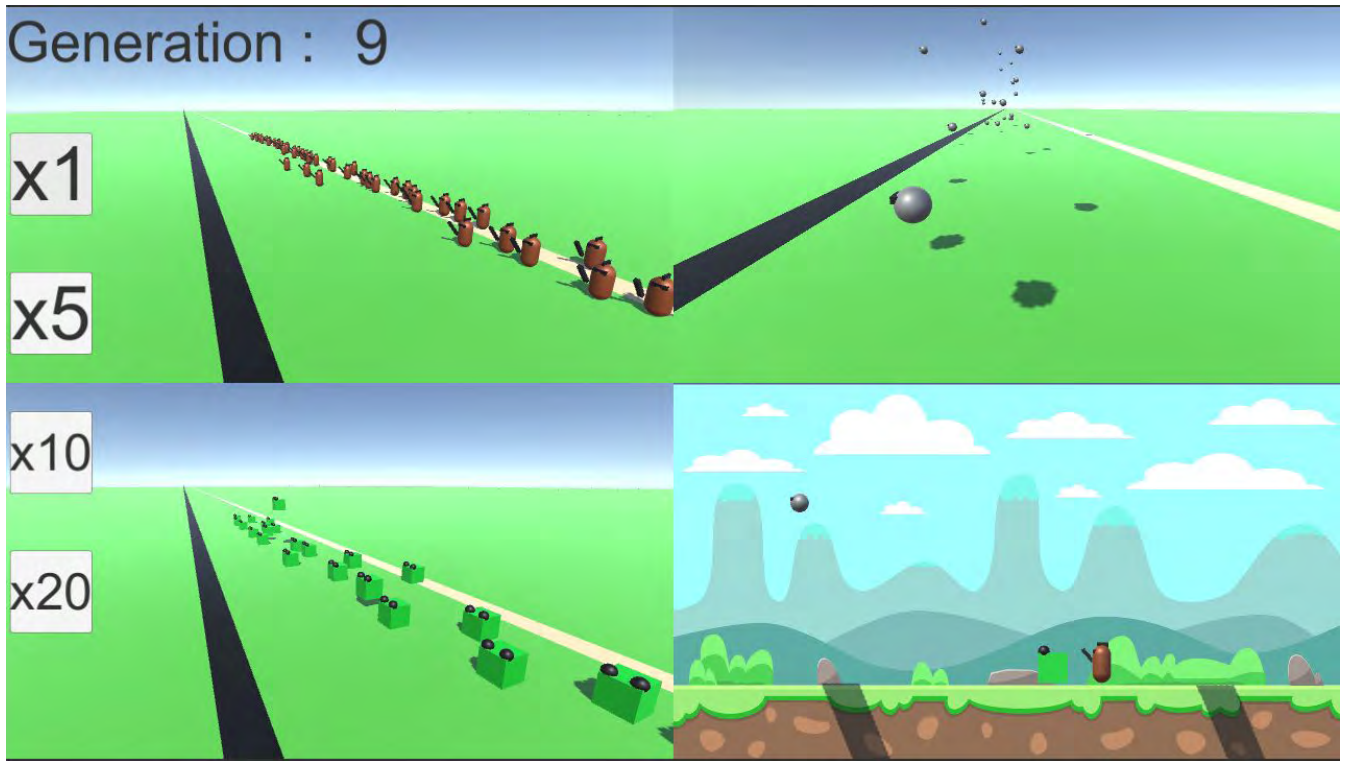


Figure [23]
Best flier individual crosses the desired horizontal position.

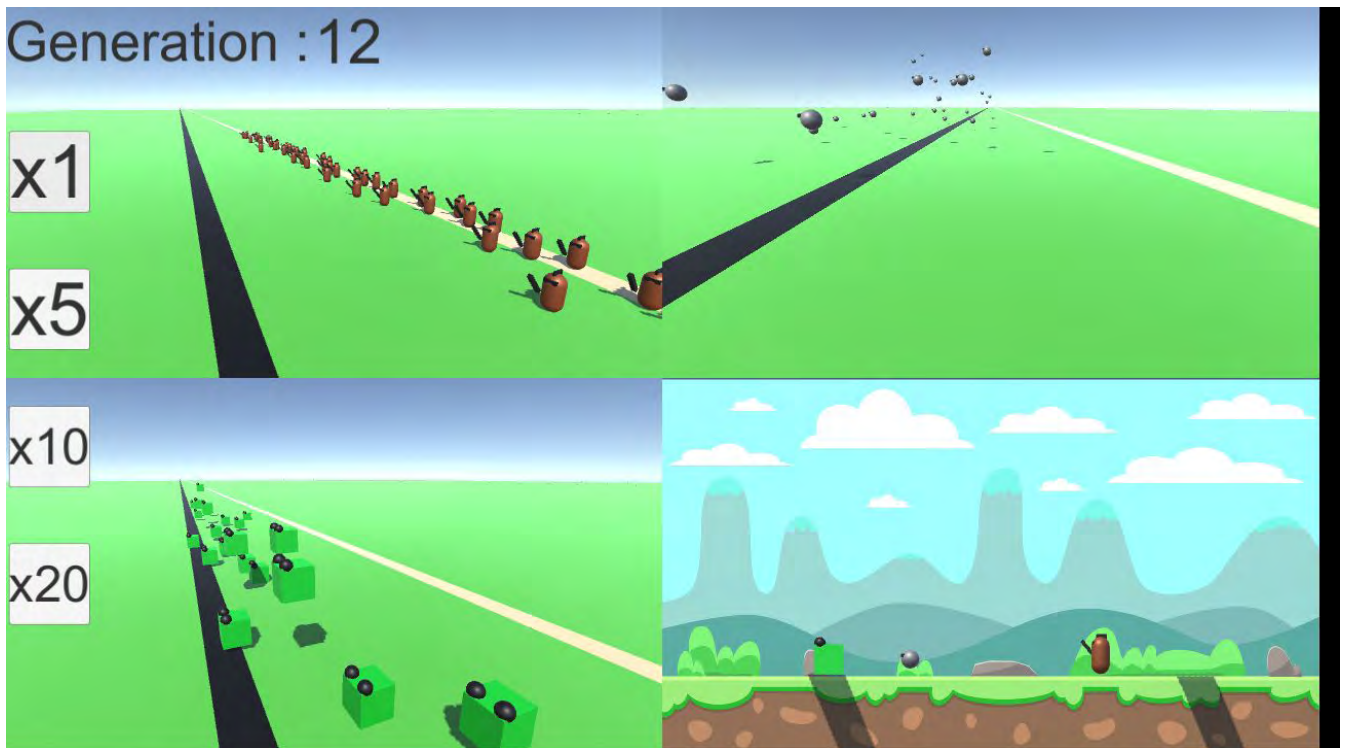


Figure [24]
Best jumpe individual crosses the desired horizontal position.

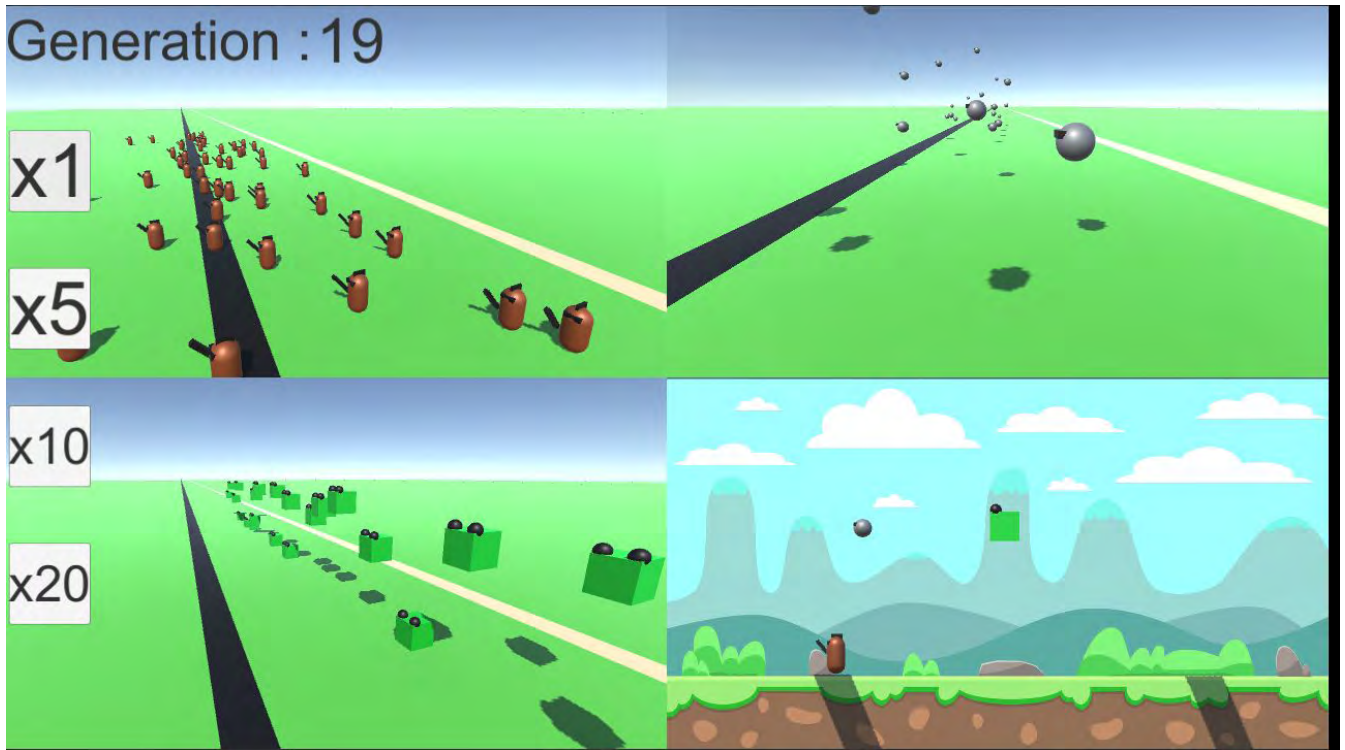


Figure [25]
Best walker individual crosses the desired horizontal position.

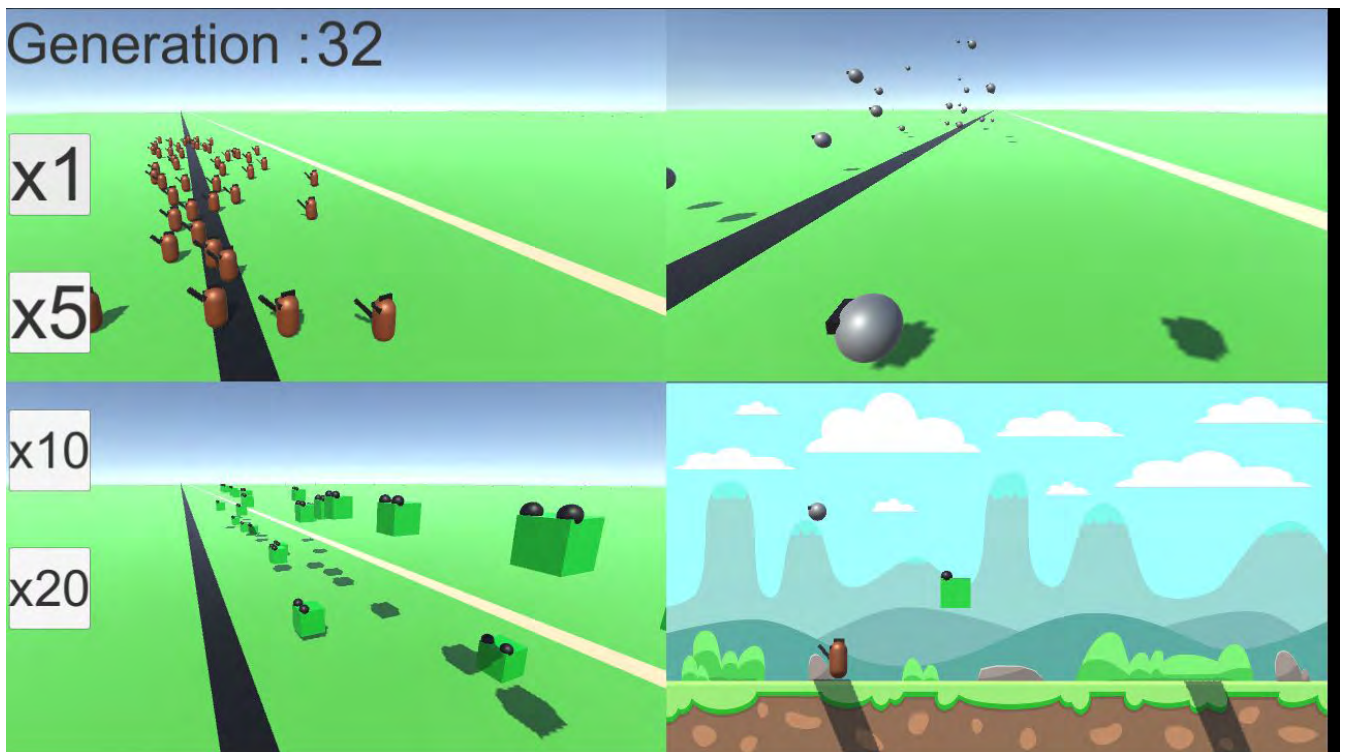


Figure [26]
Best flier and walker crosses the desired horizontal position near the desired time.

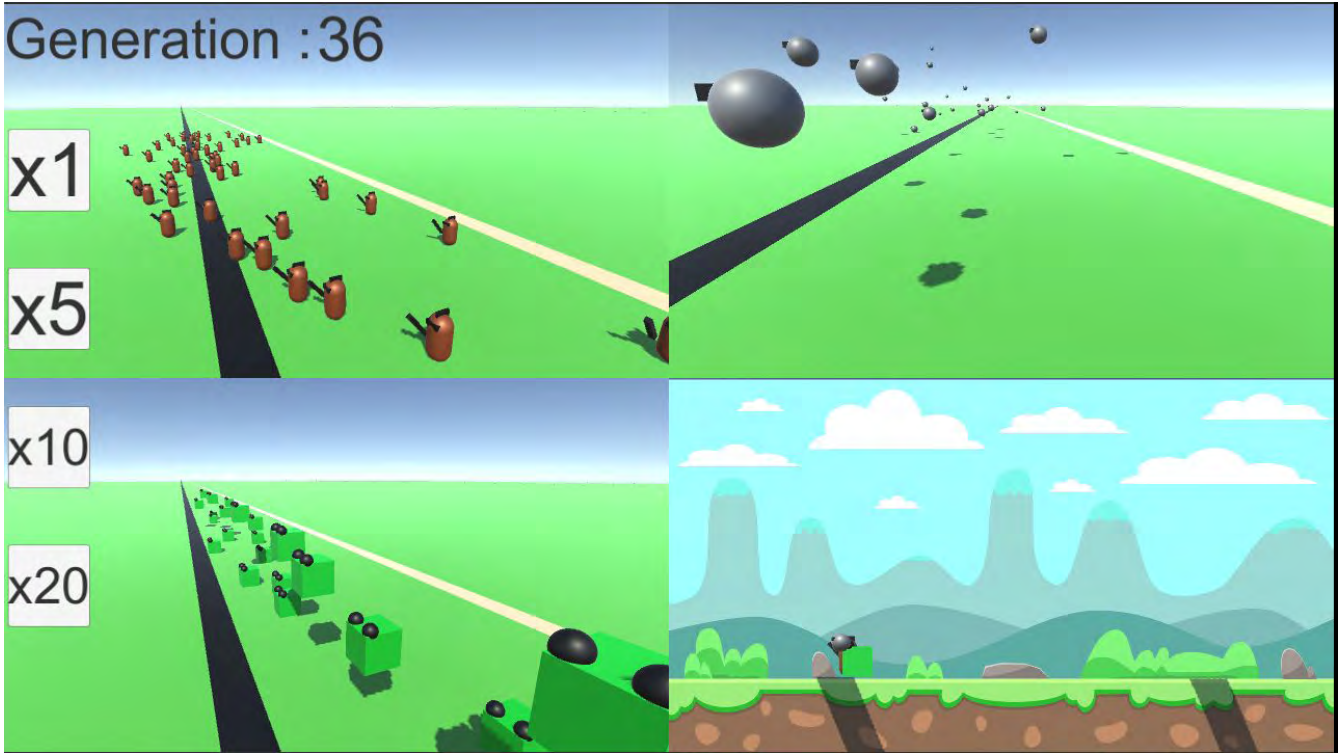


Figure [27]

Best individual of each species crosses the desired horizontal position near the desired time.

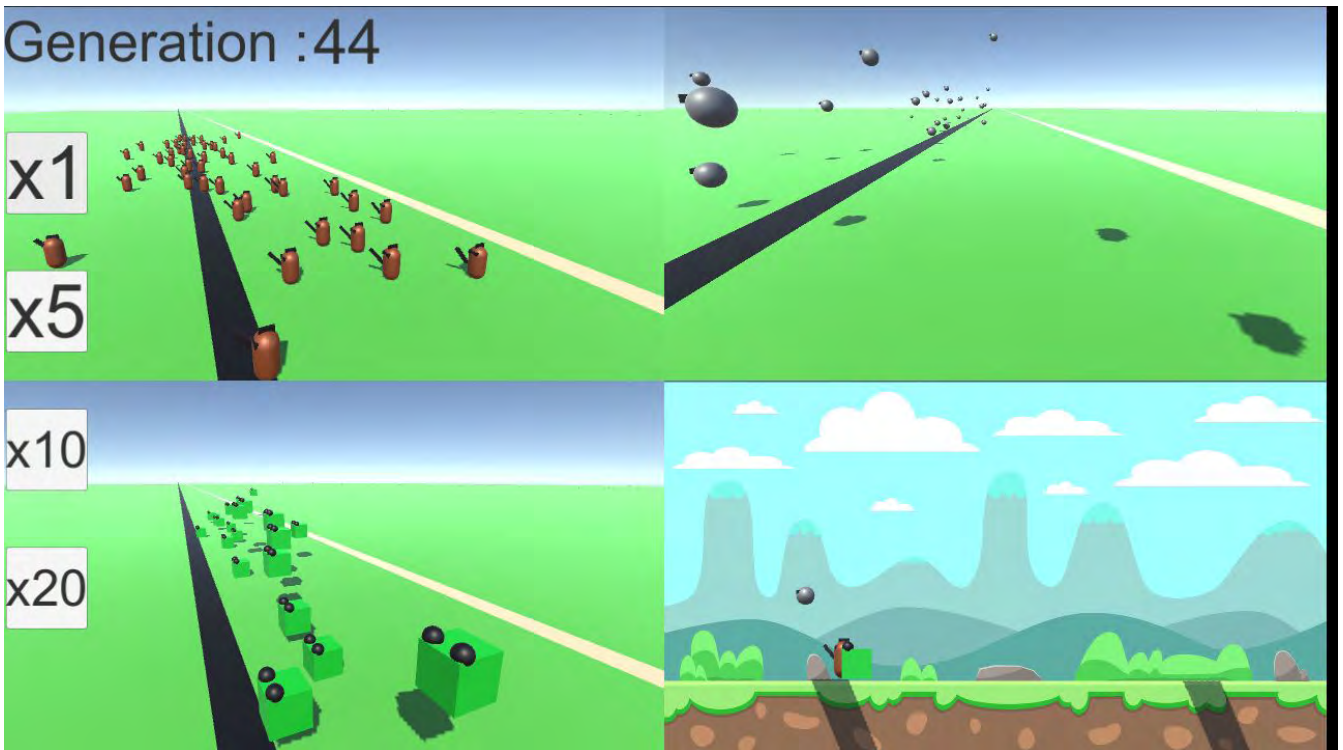


Figure [28]

Best individual of each species crosses the desired horizontal position near the desired time, and the flier is within the vertical height limit for a big period of time.

5.4 The enemy of the agents.

One key component in the behaviour of every evolutionary agent is his rival, also known as the player of the game. Each player has his own play style, this means that with a scripted agent the game will be easier for some and harder for others. On the other hand agents that evolve based on the skill of the player will gradually give the player a continuous challenge.

To have agents evolve their behaviors around the player, they have to somehow understand and react based on the player's strategy and/or his game mechanics.

The playing character in this simple simulation is someone that holds a gun that periodically shoots towards the incoming hordes of his enemies. The second mechanic is that the player can choose when the playing character will jump into the air to shoot enemies that are otherwise unreachable.

Given that said each non player character in the game will have to know where the flying bullets are and where the barrel of gun is in the world at any given time. By default there are at most three bullets existing in the same time. Furthermore the simulation takes place in a two-dimensional world so only the x and the y values are needed. If the agents have this particular information they can develop very interesting behaviours.

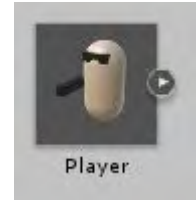


Figure [29]
The model of the player.

5.5 Brain out of Neurons.

The third implementation is all about neuroevolution. The algorithm will evolve neural networks with fixed topologies to create unique behaviours. The player that was introduced in the previous paragraph will be the cornerstone of this evolution. Each species will have different network topologies to try to solve their problem as well as they can.

5.5.1 Genome Representation.

The genome in this implementation is rather simple. A list of all the weights of the artificial neural network is everything that it takes. However a structure in this list has been made so it can be handled properly. The neural network consists of a list of neuron layers. A neuron layer embodies a list of neurons and each neuron a list of weights that includes its bias.

5.5.2 Network Topology.

All the neural networks can be examined in three stages. The input layer, the hidden layers and the output layer. The input layer is the same for all the different species of the simulation's agents. This are the eight points in their two dimensional world space as described before.

The walker has two layers of neurons in his hidden layer. The first has ten neurons and the second five. The jumper has also two layers of neurons but they have twelve and eight. After a lot of testing the flier ended up with three layers of neurons consisting of eighteen twelve and six neurons. This topologies have changed a lot during the simulations.

Lastly the output layer of the network represents the type of movement the agent must make. Due to the fact that sigmoid function of the sum of each neuron output the output values vary between $[0,1]$. Thus the movement of the walker must be divided into two separate movements, going forth or going backwards, this means that two neurons are necessary for the output layer of the walker. The jumper and the flier needs a third one that signals the vertical force. Again due to the fact that the output is between $[0,1]$ adjustments must be made so each agent will get the right amount of force.

The walker has the output of his neural network multiplied by eight and minus eight. After multiplication the sum of the two outputs is the horizontal acceleration that is applied to the agent's body. For the jumper the horizontal coefficients are five and minus five and the horizontal acceleration is calculated with the same way, as for vertical coefficient it is four hundred and twenty five. For the flier the horizontal coefficients are one and a half and minus one and a half and the vertical is twenty.

5.5.3 Population Pool.

Each species has each own population pool. This pool is gradually filling up with random agents that represent the initial population. An agent must have its fitness calculated to enter this pool, in other words it has to die. Once its maximum capacity is reached and new agents must be created the selection, the crossover and the mutation takes place as usual but they create only one new agent. This new agent is sent out to try out his strategy. Once he is dead and the fitness is calculated he must compare it with the worst individual of the pool. If he performed better then the worst fit is taken off the pool and the new agent becomes a part of it.

5.5.4 Crossover and selection.

One point crossover is being used for this implementation also but with a small twist. The point that is selected for the crossover to take place can't be in the middle of the weights of a neuron. Only whole neurons are transferred to the new agent. The structure of the genome help us to divide the genome at the right point. As for the selection classical roulette wheel is being used.

5.5.5 Mutation.

For the mutation every single weight of the genome must be tested and changed if necessary. The mutation rate for all three species is rather high standing at twenty percent. If a weight is mutated then it takes a new value between $[-1,1]$.

5.5.6 Fitness Functions.

The fitness functions are quite similar to those of the second implementation. The goals of each agent has not be changed.

5.5.7 Results.

For the walkers mainly two behaviors appear. The first one is to wait as far as they can and when they see the player jumping to run as fast as they can to cross the finish line. The second one is that sometimes they run towards the player no matter if he is shooting towards them. The first behavior is quite expected given that they realize the bullets and the player as a threat. They usually evolve fast enough to challenge the player.

For the jumpers the results was amazing. They converge to their behaviour extremely fast. In most cases they synchronize their jumps to the bullet fire rate and they jump between them. It has been noticed that they even learn to jump backwards if the player jumps so they can avoid the next incoming bullet. This general behaviour together with the walkers makes it really hard for the player not to lose after a few minutes of gameplay.

Sadly the fliers are not performing that well. They can't really find a way to float around the air of their world. In rare cases that they kinda succeed they have strange behaviours such as trying to pass over the bullets which makes them fly out of the screen. But this behaviour needs a lot of time to be developed and there is no challenge in them for the player. The main issue of the fliers is that the can't come up with a plan to bit gravity.

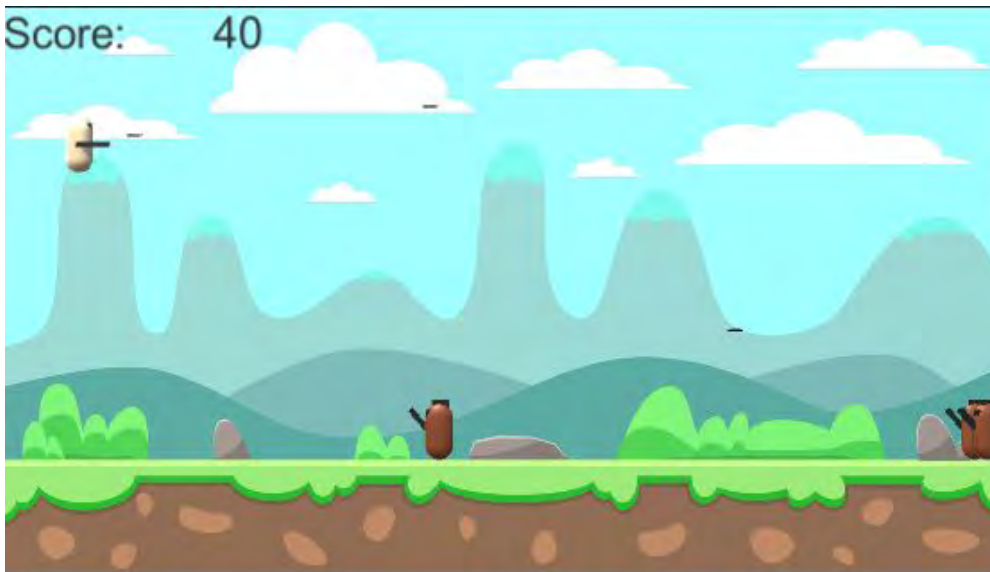


Figure [30]

A walker find its way to success while the player is on the air.



Figure [31]

A jumper that synchronized its jumps to avoid the bullets.

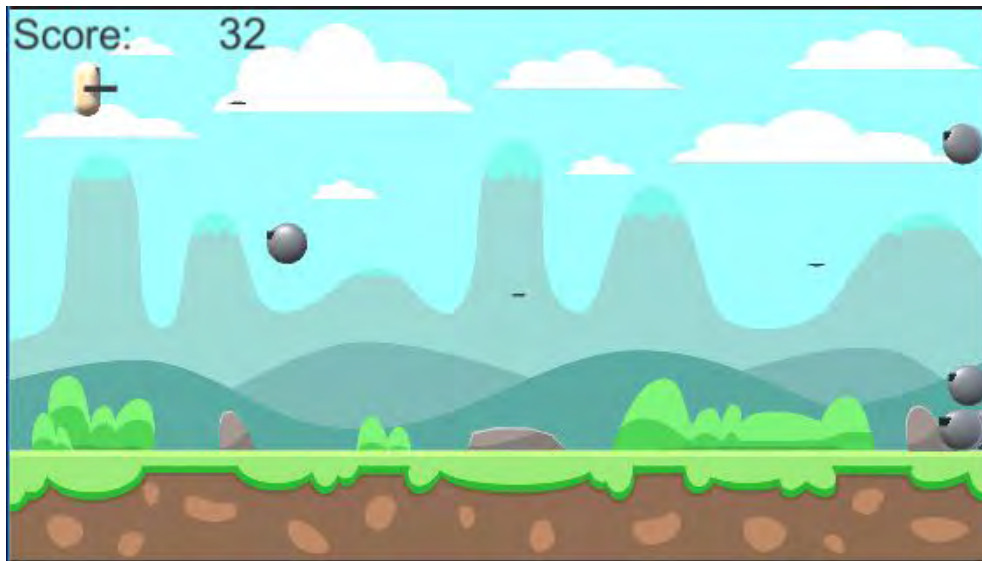


Figure [32]

A flier that succeeded to fly all the way from start to finish.

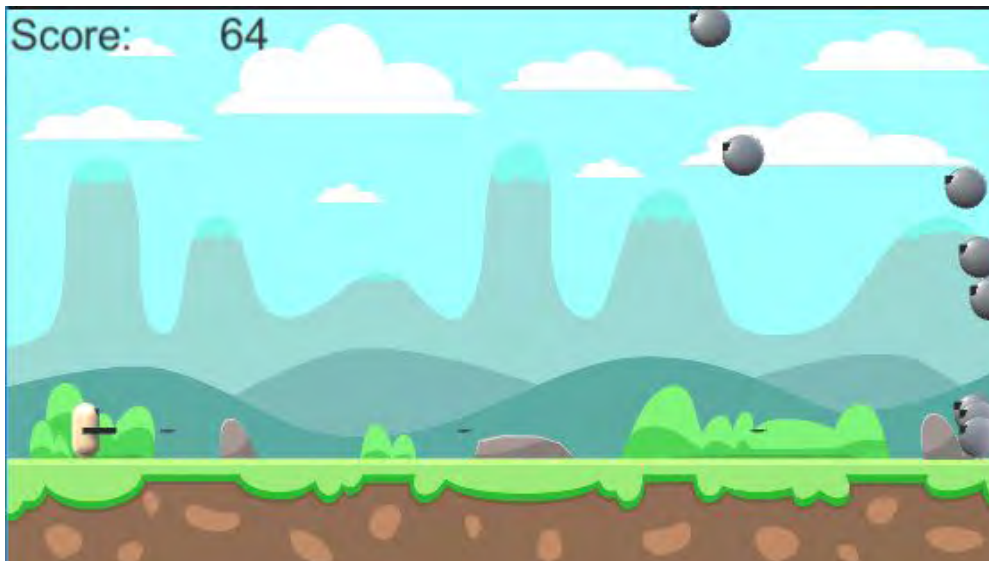


Figure [33]

The general behaviour of the fliers.

6. Conclusions.

Genetic and neuroevolution algorithms are a very powerful tool. These algorithms can surely be used to evolve unpredictable behavior that counters the player's weaknesses. Other than that, if there is enough time they can give an enormous amount of information to the development team about the balance of the game. But these two applications are not always as easy as they may sound.

They can make a game a lot more interesting but they have some serious limitations. The biggest of them all is the limitation of the developer's mind. The better the developer has understood the problem the better the algorithm can work. Other than that being stuck in a local optima is always a possibility. In chapter five it was very clear that some time some species performed a lot better than others and sometimes there were agents that didn't perform at all.

As of the implementation the non player characters performed very good in general. Though further work must be done to solve bigger problems.

Lastly it is worth mentioning that neuroevolution algorithms are being used on self driving cars and other types of controllers. This by each own makes them very important because these controllers will be the future of humanity, though no one knows for sure.

7. Bibliography

- [1] Ai techniques for game programming. By Mat Buckland.
- [2] The Nature of Code. By Daniel Shiffman (2012).
- [3] An introduction to genetic algorithms. By Mitchell Melanie (1996).
- [4] Genetic algorithm. https://en.wikipedia.org/wiki/Genetic_algorithm.
- [5] Selection (genetic algorithm). [https://en.wikipedia.org/wiki/Selection_\(genetic_algorithm\)](https://en.wikipedia.org/wiki/Selection_(genetic_algorithm))
- [6] Crossover (genetic algorithm). [https://en.wikipedia.org/wiki/Crossover_\(genetic_algorithm\)](https://en.wikipedia.org/wiki/Crossover_(genetic_algorithm))
- [7] Mutation (genetic algorithm). [https://en.wikipedia.org/wiki/Mutation_\(genetic_algorithm\)](https://en.wikipedia.org/wiki/Mutation_(genetic_algorithm))
- [8] Evolution strategy. https://en.wikipedia.org/wiki/Evolution_strategy
- [9] Genetic Algorithm. By Tom V. Mathew.
- [10] Niching Methods for Genetic Algorithms. By Samir W. Mahfoud (1995)
- [11] Artificial Neural Networks. Wikibooks.
https://upload.wikimedia.org/wikipedia/commons/1/13/Artificial_Neural_Networks.pdf
- [12] Artificial Neural Networks. https://en.wikipedia.org/wiki/Artificial_neural_network
- [13] Perceptron. <https://en.wikipedia.org/wiki/Perceptron>
- [14] Backpropagation. <https://en.wikipedia.org/wiki/Backpropagation>
- [15] Activation function. https://en.wikipedia.org/wiki/Activation_function
- [16] Softmax function. https://en.wikipedia.org/wiki/Softmax_function
- [17] Feedforward neural network. https://en.wikipedia.org/wiki/Feedforward_neural_network
- [18] Recurrent neural network. https://en.wikipedia.org/wiki/Recurrent_neural_network
- [19] Recurrent neural network tutorial, by Denny Britz.
<http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/>
- [20] Reinforcement learning. https://en.wikipedia.org/wiki/Reinforcement_learning
- [21] Neuroevolution. <https://en.wikipedia.org/wiki/Neuroevolution>
- [22] An overview of neuroevolution techniques. By Vincent Hoekstra (2011).
- [23] Neuroevolution: From architectures to learning. By Dario Floreano, Peter Durr and Claudio Mattiussi (2008).
- [24] Evolving Neural Networks through Augmenting Topologies. By Kenneth O. Stanley and Risto Miikkulainen.
- [25] Neuroevolution with Analog Genetic Encoding. By Peter Durr, Claudio Mattiussi and Dario Floreano.
- [26] Comparison of NEAT and HyperNEAT on a strategic Decision-Making Problem. By Jessica Lowell, Kir Birger, and Sergey Grabkivsky.
- [27] Real-Time Neuroevolution in the NERO Video Game. By Kenneth O. Stanley, Bobby D. Bryant, and Risto Miikkulainen
- [28] Sane neurogenetic algorithm. <http://satirist.org/learn-game/methods/ga/sane.html>