UNIVERSITY OF PIRAEUS – DEPARTMENT OF INFORMATICS

MSc «Advanced Informatics and Computing Systems - Software Development and Artificial Intelligence»

Master Dissertation

Dissertation Title	**Simulation and Comparison of Reinforcement Learning Algorithms**
Student Name	**Konstantinos Spyropoulos**
Father's Name	**Miltiadis Spyropoulos**
Student Id	**18023**
Supervisor	**Dionysios Sotiropoulos, Assistant Professor**

Submission Date: September 2022

Contents

Image List

Table List

Abstract

The field of Artificial intelligence in modern era is an important tool in the implementation of modern ideas and the development of many useful applications. Machine learning provides a variety of techniques and algorithms for problem solving where classical programming is limited. A similar problem is presented in this work, which is a theoretical and practical approach in the field of machine learning aiming in finding suitable techniques for solving complex problems.

Creating intelligent artificial agents to solve human-related problems is a major challenge for the artificial intelligence community. Particularly important is the understanding of the dynamic environment in which they operate and their interaction with it, just like humans do in the physical world. The field that specializes in creating such agents is called Reinforcement Learning. With the term Reinforcement Learning, we refer to the methods through which a system of algorithms 'learns' to interact within a structured environment after trial and error. This learning is done by exploring the environment through actions and rewards given by the environment to achieve a goal with optimal effort.

This research was developed on the idea of the simple chase game, which aims to create an environment and the agents that play in it. In this game, hunting agents are tasked with catching opponents, while the hunted agents try to stay free. The goal of the work is, apart from understanding the rules and getting the right decisions to achieve victory, finding new strategies that will lead to optimal results and comparing the most valuable algorithms.

The proposed approach has been chosen after analysing a wide range of Reinforcement Learning algorithms in a graphical emulation environment. The results achieved show the best implementation technique, but at the same time prospective improvements are also highlighted in the way of getting the right decisions but also in dealing with the more complex environment and rules of the game.

1. Introduction

1.1 What is Reinforcement Learning (RL)?

Learning in nature comes from interaction of living organisms with their environment. Due to this interaction, living organisms evolve, they learn from their actions and the resulting consequences and adapt in order to achieve certain goals that are extremely important to their survival. Reinforcement learning is based in this core idea and constitutes a major part in the machine learning field that faces decision making problems by gaining experience via environment interactions.

Specifically, inside a system, an agent takes actions and gains rewards which evaluate that action, depending on how much this action contribute to achieving a certain goal. The agent's final goal is to select the right decisions to maximize the aggregated reward.

What separates reinforcement learning methods from the other machine learning methods (e.g., supervised learning) is that in RL the system provides only partial feedback to the agent from its predictions. Simultaneously, those predictions can have major influence in long term future states of the agent. As a result, time plays an important role in agent's learning, where every decision directly and indirectly affects every possible state of the system until the completion of the goal.

Another important parameter of RL is that the training exists through trial and error. The agent does not have the full control of the environment which it exists, so it has to collect information which will determine its behaviour. It is important to also consider that using this method, it results to the exploration/exploitation dilemma which will be further analysed in the following sections.

1.2 Historical Remarks

The field of Reinforcement Learning is not a new concept, it has been improved and amended over the last 70 years of study. It initially approached simultaneously three different reinforcement learning methodologies (trial and error, optimal control and temporal difference), until finally around 1990 they merged into one method as we know it today.

- Trial and Error:

  This methodology was merged into machine learning field of study after Minsky research (1954) and the use of SNARCs (Stochastic Neural-Analogue Reinforcement Calculators). With the study of Clark and Farley (1955) in pattern recognition and Rosenblatt (1962), where the theory of neural networks and connected neurons was established, it was assumed that there is basically no difference between reinforcement and supervised learning. In 1961, Minsky also proposed the 'Credit Assignment Problem', which describes the difficulty of choosing the actions which contributed the most in fulfilling the final goal. Finally, John Andreae (1963-1977) ends the discussion between separating the two fields by creating STELLA, which learned through its interaction with the environment, making reinforcement learning an independent methodology.

- Optimal Control

  The study started in 1950 and it defines as *'the process of determining control and state trajectories for a dynamic system over a period of time to minimize a performance index'* (Sutto and Burto 2018). Bellman (1957) founded the well-known Bellman Function, which defines dynamically an equation using the state of a dynamic system and returns the optimal value function. Moreover, he proposed MDP (Markovian Decision Process) which is defined as *'a discrete stochastic version of optimal control'*.

- Temporal Difference

  Inspired by differential calculus, learning through temporal difference aims in predicting using a set of known variables of the current valuation of value function, a methodology like Monte Carlo (Hammersley 1964). It was developed and improved mainly after the study of Minsky (1954) and Samuel (1959).

Finally, those three assumptions were combined with the study of Watkins (1989) where the idea of Q-Learning was introduced. In the next decades, those ideas evolved and implemented over many different fields, starting with board games like backgammon, chess and Go (Tesauro 1994, Baxter 2000, DeepBlue IBM 1997, Google AlphaGo 2016). Many applications followed in Atari video games due to the use of simple control of the games and the graphical interface they provided. Until today, reinforcement learning is used in many applications like smart cars, advertising, web systems, robotic application, and chemistry.

1.3 Goal of the study and application

In this current study, a complete agent learning system was developed in a graphical environment that consists of certain rules. The main goal of the development is the usage of different techniques and algorithms and their comparison to find the one the fits in this specific problem. Simultaneously, throughout the experiments, certain conclusions were derived that describes the way each method helps in specific occasions of the problem. Equally important is the comparison of the decisions and actions of the agents with the human perspective and the different logic they follow in cases like the referred problem.

This and other similar experiments, which can be resolved using reinforcement learning, can be implemented in real world problems. Examples like these can be found anywhere in the real world, especially with the uprise of computational power of computer systems.

Self-Driving Cars

Many papers are recently published that refer to the technology of self-driving cars and the need of improving their characteristics, like changing lanes in the road, obstacle avoidance, maintaining speed and automated parking. Furthermore, an improved study is needed to optimize the control of the traffic lights in order to minimize traffic and accidents. Many of those applications are developed using the Q-Learning algorithm, which will be further analysed in the following sections.

Robotic Applications

In the recent years, many productions lines use robotic systems to improve accuracy and speed in high hazardous environments. By training the robot making specific movements in the best possible way, it can improve drastically the time of the execution and the power consumption by 40%. This can also be developed with Deep Q-Learning and QT-Opt which supports continuous movement in an environment.

Video Games

In the rising field of gaming industry, more and more companies are currently trying to surpass human capabilities and creating the best possible and realistic experience to the gamers.

Marketing and Advertising

Advertising certain products to certain groups of people is something very important to today's industry. By advertising in certain people, it minimizes the cost of the advertisement and maximizes the profits. Finally, with the uprising of the social media, we can see plenty of learning applications to present advertisements, depending on the user profile, his interests, even the people he is likely to communicate.

2. Definitions and Methodologies

2.1 Architecture and definitions

The main structure of a Reinforcement Learning system consists of two parts, the agent, and the environment. The environment defines the area where the agent takes actions, providing him the state for every time t. The agent, where the RL algorithm is implemented, decides based on the current state. In the following moment t, the environment provides to the agent the new state and its reward. Following up, the agent reviews and evaluates his actions using that reward and decides again his next actions. This loop continues until the environment sends a terminal state, where the episode ends.

We define episode as the above repetition until the terminal state. Terminal state can be send when the agent fulfils his purpose or when he surpasses a movement threshold in the environment.

The above logic can be described in Figure 1.

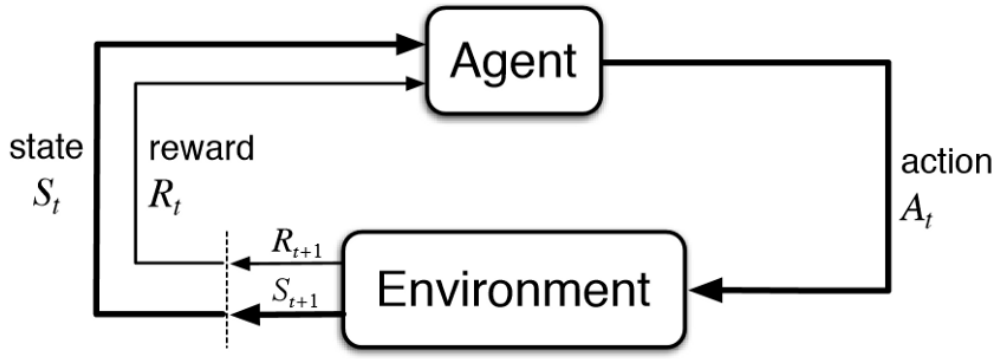

Figure 1: Basic RL Structure

Most RL algorithms follow the above tactic. The basic features that are used are:

- Agent: The entity that acts in the environment and receives reward for each action.
- Environment (e): The area that agent acts.
- State (S): The current state that the environment return to the agent.
- Action (A): All possible actions that the agent can do in each state.
- Reward (R): The reward given to the agent for every action.

A Reinforcement Learning system can also be defined by three important elements characterising the behaviour of the agent, policy, value function and model of the system.

Policy

The policy defines agent's behaviour at some point in time. It's the core of the agent as it's just enough to determine each behaviour. More specifically, the policy corresponds to the state in which the agent is located with the action it will make. In many cases it may be a simple function or a table, but there are cases where its logic may be much more complex by having extensive computational processes such as a search algorithm. More generally, policies are targeted procedures with specific possibilities for each action.

Value Function

Accordingly, value function refers to the rightfulness of an action in the long term. The value of each state is the sum of the reward that agent can receive in the future, starting from the present state, differentiating it from the reward that expresses the immediate desirable reward of the state. It is therefore understood that the assessment of value function is quite difficult to find than the reward, but it has a key role during the training procedure.

Model

Finally, the model mimics the behaviour of the environment. It is used to design and predict the following states before agent is found in them, learning the probability of transition from a condition s_0 to another s based on action a.

$$P(s_1 \mid (s_0, a))$$

Model-based και Model-free

In the context of reinforcement learning, systems are categorised as model-based and model-free methods, depending on whether these models are designed from the start or if they are set up during the trial-and-error process. Most of the time model-free systems are used as they do not require room for the storage of all state-action assemblies and, in addition, provide a more dynamic approach to changing environments.

Below, a comprehensive reference is made to Reinforcement Learning algorithms, which some of them are important for the development and creation of the final algorithm used. Some of them are the MDP, the Q-Learning and Deep Q-Learning algorithm, NEAT (Neuro-Evolution of Augmentation Topologies) but also other stochastic algorithms.

2.2 Markov Decision Process (MDP)

2.2.1 Introduction

The Markovian decision process (MDP) is a mathematical discrete stochastic process that is very important to study optimization problems through dynamic programming. Almost all Reinforcement Learning projects can be resolved using MDP or adapt to it. The basic idea is that the agent decides to choose the best action based on its present situation. Below follows a description of the major MDP properties as well as a brief description of solving a problem in a fully supervised environment.

2.2.2 MDP Mathematical Description

More detailed, according to Markov property, a state S_t depends only on the latter situation and ignoring all past situations:

$$P[S_{t+1} \mid S_t] = P[S_{t+1} \mid S_1, \ldots, S_t] \quad (1)$$

where S are the environment states.

So, we conclude that:

$$P_{ss'} = P[S_{t+1} = s' \mid S_t = s] \tag{2}$$

where $P_{ss'}$ is the probability of transition from the current state to the next state. Equation (2) is defined as the Markov Chain and consists of a series of states S_1, S_2, … where all of them obey to the Markov property.

Markov Reward Process is defined as the estimated reward of all possible situations that agent can travel from a state s. In particular:

$$R_s = \mathbb{E}[R_{t+1} \mid S_t = s] \tag{3}$$

In conclusion, the Markovian decision process (MDP), on the basis of and (1), (2) and (3), the transition probability of states and rewards are defined as:

$$R_s^a = \mathbb{E}[R_{t+1} \mid S_t = s, A_t = a] \tag{4}$$

$$P_{ss'}^a = \mathbb{P}[S_t + 1 = s' \mid S_t = s, A_t = a] \tag{5}$$

As the reward is temporary, it may be possible after an action to have a large reward but in the long term to achieve less than desirable. This long-term reward is defined as a Return, which is calculated using a discount (c). The logic behind the use of the discount is that we cannot be 100% sure of the future. It is therefore important to keep our future rewards in mind, but also to minimise their contribution through the discount.

More specifically:

$$G_t = R_{t+1} + \gamma R_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{6}$$

where $\gamma \in [0, 1]$.

The evaluation of "how good" is the choice of an action by the agent in a particular state is defined by the policy, which describes its decision-making behaviour by mapping out the probabilities of all the possible actions. The assessment of these options is carried out by the value function for the action to be taken and the state-value function. Those specific functions are evaluated through the agent's experience, and it is important to optimise it properly.

In more detail, policy is a probability distribution of actions α on the basis of the current state s.

$$\pi(a \mid s) = \mathbb{P}[A_t = a \mid S_t = s] \tag{7}$$

The value function presents the long-term value of the state s, meaning:

$$v(s) = \mathbb{E}[G_t \mid S_t = s] \tag{8}$$

Also, according to (8), state-value and action-value functions that are subject to a policy π are defined as:

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] \tag{9}$$

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] \tag{10}$$

Bellman equation gives a depiction of the value function. It consists of two separate parts:

- The immediate reward
- The value of future states after discount

$$\begin{aligned} v(s) &= \mathbb{E}[G_t \mid S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s] \end{aligned} \tag{11}$$

More specifically, an agent may go from a state s to a situation of s'. The state value calculates the expected value of the return from all subsequent states s'. Using recurrently the same definition for each subsequent state s', we are driven to the function (11).

In conclusion, we find below the Bellman equation for the optimal value:

$$v_*(s) = R(s) + \max_a \gamma \sum_{s'} P_{sa}(s') v_*(s') \tag{12}$$

where, by taking the values of each state of an MDP for all policies, we choose the one policy with the maximum value:

$$v_*(s) = \max_\pi v_\pi(s) \tag{12}$$

Accordingly, for each action, we choose the optimal policy:

$$q_*(s, a) = \max_\pi q_\pi(s, a) \tag{13}$$

2.3 Q-Learning and Deep Q-Learning

2.3.1 Main idea and Differences

With the term Q-Learning we refer to a Reinforcement Learning algorithm with the following characteristics:

- model-free: Evaluates the optimal policy without the need of a model of the environment and according to transition functions and rewards.
- off-policy: Evaluates the reward for each future action and updates the policy, without following a predefined behavior.

The agent uses the rewards of the environment to carry out the best possible action for the present state. In a more simplified version of the Q-Learning algorithm, this decision is taken in accordance with a Q table consisting of Q-values which match the actions with the possible state-action pairs. In the course of the training, this table is renewed, improving the Q-values that returns a greater reward.

It is understood that the greater the area or possible actions that the agent can take, the greater the Q table, so it is more difficult to train and maintain the Q table. The problem is even greater if the possible circumstances of the agent are in a continuous area or are overcrowding.

The Deep Q-Learning is a solution to this problem by replacing the table with a neural network, which accepts as input the possible situation and as an exit the Q-values corresponding to each operation that can be executed.

2.3.2 Q-Learning description

For each finite Markovian decision process, Q-Learning finds optimal policy maximising the value of the overall reward for all subsequent states, starting with the current one. It uses Temporal Differences (TD) to assess $Q^*(s,a)$, meaning the estimated discounted value, and therefore learns through its experience without knowledge of the environment.

Initially, Table Q is defined, which is the structure used to calculate the maximum estimated future reward for the action of each state. Initialisation shall be carried out at zero values with a MxN scale, with M number of s and N the number of actions α.

Then, the agent selects an action. The epsilon-greedy strategy is being implemented here. In detail, the choice of an action is initially carried out randomly. Gradually, and while agent acquires a better picture of the environment, epsilon decreases, so agent becomes greedier, choosing only actions that will certainly bring him a great reward, so it involves less exploration of the area. The balance between exploitation and exploration is very important. With greater exploration, the agent will wander aimlessly in the space following tactics that do not help him achieve the ultimate goal, and with great exploitation, the agent is in danger of being locked up at a local minimum.

After the execution of the action, the agent receives the reward from the environment and updates the Q table.

The values of Q table are updated through Q-function, which uses the Bellman equation taking as inputs the state (s) and the action (α).

$$Q^{\pi}(s_t, a_t) = \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ \ldots \ \mid s_t, a_t] \tag{15}$$

From (15), we conclude for finding the optimal value

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right] \tag{16}$$

where α is a hyperparameter that express the learning rate of the algorithm.

The function (16) describes the updating of the new Q-value by adding the old value plus the new estimate of reward.

This procedure is repeated until the training is stopped. Below is a diagram giving a brief description of the algorithm.

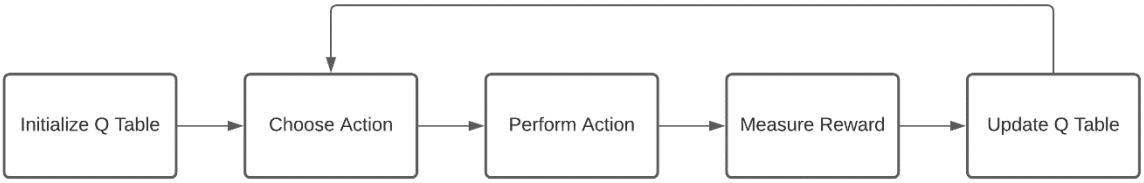

Figure 2: Q-Learning Algorithm diagram

2.3.3 Deep Q-Learning description

Deep Q-Learning algorithm is based on the same logic as Q, except that we use neural networks instead of a Q-table to estimate Q-values. In such a network, input is the current state and output are the Q-values for each of the actions.

For the implementation of such a network, the first key step is to store in memory the previous experience of the agent. This process is called replay memory:

$$e_t = (s_t, a_t, r_t, s_{t+1}) \tag{17}$$

The reason that replay memory is used is initially to make agent experience more efficient as it is reused during the course of training. This allows the system to learn from concrete examples multiple times. In addition, it helps not to "forget" past experiences but also to reduce the correlation between similar situations.

The next step concerns the training of the network. According to the equation (16), the network's loss of function can be calculated by finding the difference between Q value and the desired Q. In more detail, using the Bellman equation (12), the desired Q-value (Q-Target) is assessed:

$$\begin{aligned} Q_{target} &= R_{t+1} + \gamma max_a Q(S_{t+1}, a) \\ Q_{loss} &= R_{t+1} + \gamma max_a Q(S_{t+1}, a) - Q(S_t, A_t) \end{aligned} \tag{18}$$

An important challenge presented in line with the above logic is that the system uses the same parameters for the assessment of Q-target but also Q-value, resulting in each step of education, both values being moved. A solution is the implementation of different neural networks, with the Q-Target estimation network to have stable parameters values, and to be renewable by Q-value network after C steps.

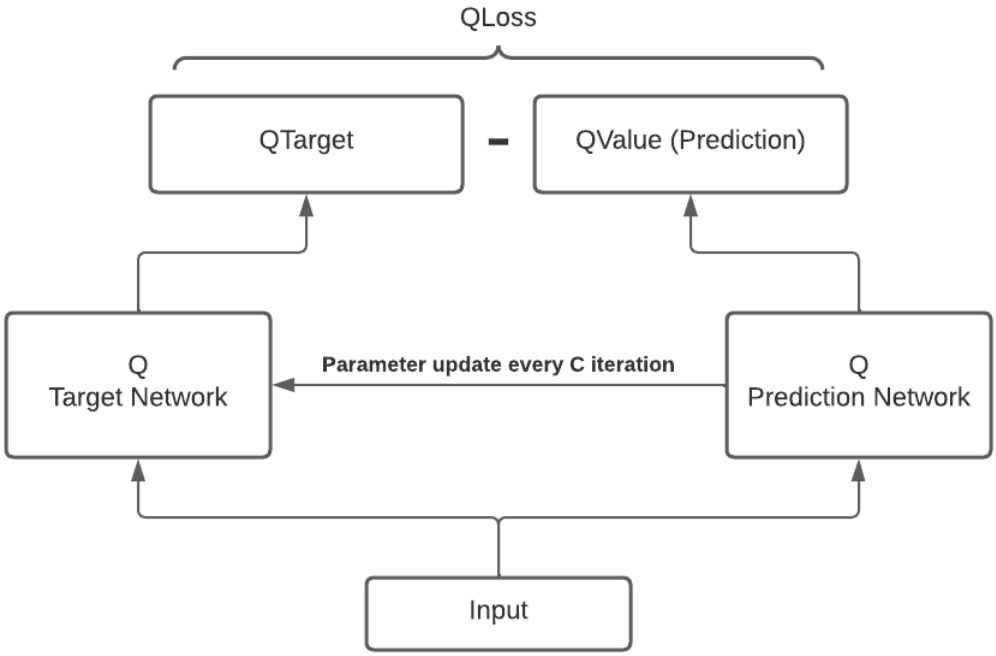

Figure 3: Different network implementations for Q-Loss

It is worth noting that during training each predicted action follows, like in the Q-Learning, an epsilon-greedy policy to increase the degree of exploration in every action of the training. Immediately after the calculation of Q-Loss, gradient descent is executed to reduce the error. Upon completion of a specific number of repetition C, the network weights shall be transferred to the target network. The whole process is repeated for a fixed number of episodes.

2.4 NeuroEvolution of Augmenting Topologies (NEAT)

2.4.1 Introduction to NeuroEvolution

The term Neuroevolution defines the Machine Learning method by which evolutionary algorithms are applied for the manufacture of artificial neural networks. Applications with this method over reinforcement learning problems have high levels of effectiveness, as compared to traditional methods of learning with stationary neural networks, they have a high generalisation that allows learning without clear objectives and with little feedback.

A big question in Neuroevolution is the simultaneous evolution of the topology of neural networks by training the weights of connected neurons. In need of overcoming this challenge, the NEAT algorithm, which belongs to the TWEANNs group (Topology and Weight Evolving Artificial Neural Networks) was developed. NEAT is showing that it outmatches evolutionary algorithms of a specific topology, that is, a very structured form of neural network, which only evolved the values of the trainable weights.

According to Kenneth O. Stanley and Risto Miikkulainen in their study in 2002 (Evolving Neural Networks through Augmenting Topologies), the basic idea of NEAT revolves around addressing the problems faced by TWEANNs and how the algorithms was designed to deal with them. These technical difficulties that emerge are:

1. Finding a genetic representation of networks that allows different topologies to apply crossover to each other.
2. The protection of topology, which takes a few generations to optimise, so that it does not disappear prematurely from the population, as the ultimate purpose is to find a simple topology that can only benefit from added complexity.
3. The minimisation of topologies during evolution without the use of a fitness function that calculates the complexity.

2.4.2 NEAT algorithm description

To solve the first challenge, the networks concerned are traced back to a representation called genome and the corresponding expression of the network in a form called phenotype. Each of the genomes includes a list of connecting nodes that describe:

- The in-node
- The out-node
- The weight of the node
- The enable bit which indicates the existence of a node in the network
- The innovation numbers

The innovation number is particularly important for the creation of new networks. For each new connection node (gene), we set an innovation number which is unique across the entire range of the training. The creation of new networks can be done through one of the following methodologies.

Mutation

The network mutation can be either on an existing network or adding a new structure to it, while at the same time changing the structure of the network but also the weights of the connection nodes. Structural mutations are implemented in two different ways:

a) by adding a connection where there is a merging of two existing nodes
b) by adding a node where a new node is created and the two corresponding connections of input and output

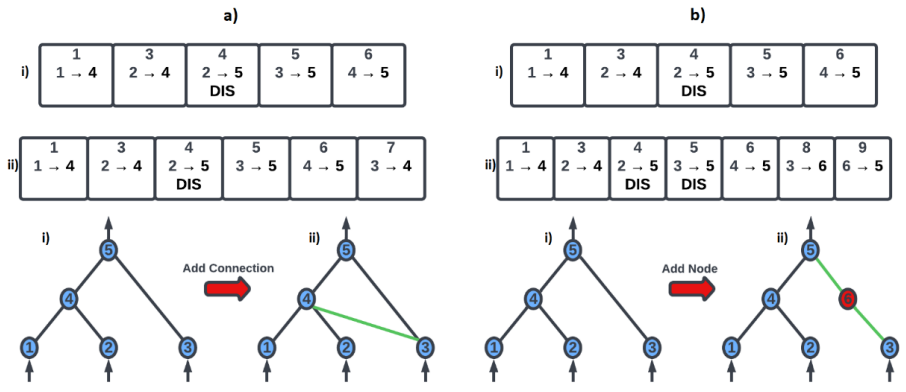

Figure 4: Genotype and Phenotype examples of mutation

Due to mutation, multiple different network topologies are created in the same position by referencing the innovation numbers of the nodes.

Crossover using innovation number

A major problem with crossover between networks is that it can create very complex or even non-functional networks due to a completely different structure and size. In the algorithm NEAT, this is easily resolved using the innovation number, a track number given after each new development.

Initially, for crossover to be applied between two genomes, the algorithm needs to know which genomes in the population are from a common ancestor. These can easily be found using the innovation number of each gene, which is unique. Thus, the origin of each gene in the system is known and we can easily find the pairs to be joined together, whereas genes that do not match are dismantled or concatenated.

More specifically, the genes between two genomes, which are the same then remain and are in line with each other. Those that don't match, so they have different innovation numbers, inherit from the best ancestor or the best random ancestor if both are just as good.

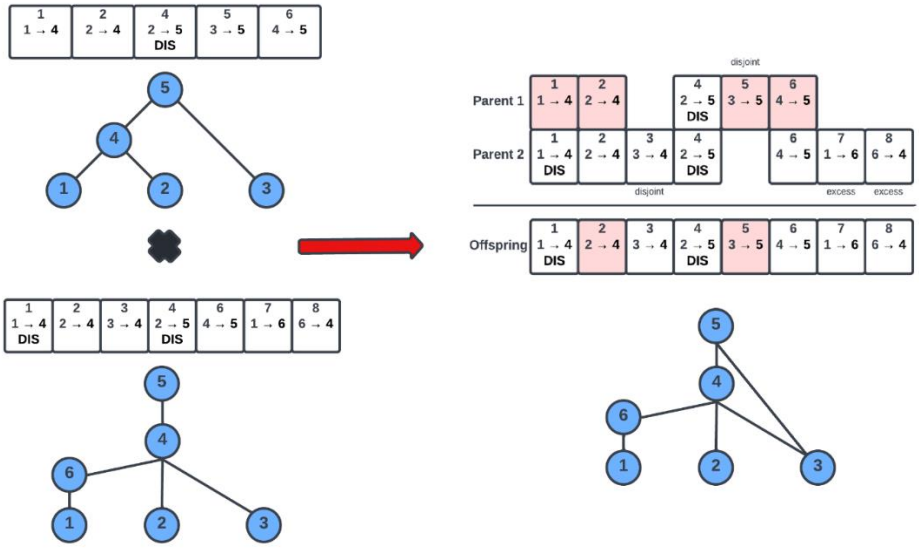

Figure 5: Genotypes and Phenotypes examples of crossover

In this way, NEAT applies crossover between two different networks without adding unnecessary complexity to the network or charging it with computational costly topological analysis. At the same time, any network structures can be merged without further analysis and in a rather simpler way.

Speciation

A major handicap presented during evolution is the fact that the new structures in their creation have an increased error. By adding a new connection or a new node without any optimisation of the network's weights, it would lead to a significant disadvantage vis-à-vis other networks of the population. One way of dealing with this problem is to protect these new networks by categorising the population into species based on their topological similarity (speciation). By this method, new networks are predicated to optimise their weights before they need to be compared to the rest of the population.

The seperation of networks is being implemented with the help of innovation numbers. The greater the number of non-like genes between two different genomes, the more disparate they are. The smaller the compatibility distance δ of two different structures, the more similar these structures are. This is expressed as:

$$\delta = \frac{C_1 E}{N} + \frac{C_2 D}{N} + C_3 \cdot W \tag{19}$$

where:

- E: the number of excess genes
- D: the number of disjoint genes
- W: the mean weight difference between two genomes
- N: the number of genes of the greatest genome
- c_1, c_2, c_3: the importance of the above parameters

The genomes belonging to the same category are represented by a random genome from this category of the previous generation. If a genome does not belong to any category, then this genome creates a new category with the same as a representative.

Similarly, the structures of the same category share the same weights among themselves, preventing the category from going beyond the total population, thus protecting new structures during training. This is the case with the use of the modified fitness function ($f'i$) for a structure i calculated on the basis of distance δ of all other structures j.

More specifically:

$$f'_i = \frac{f_i}{\sum_{j=1}^{n} sh(\delta(i,j))} \tag{20}$$

sh function is equal to 0 when δ(i, j) is above a threshold δ_t which we define, otherwise ii equal to 1.

Minimizing Dimensionality

Based on the overall concept of TWEANNs, the population is initialized with various random topologies so that the principle of diversity exists from the beginning. Instead, NEAT starts from a small-scale space without hidden nodes, and gradually increases the complexity of its structures, thus reducing the amount of training and increasing its efficiency compared to TWEANNs.

2.5 Alternative Methodologies

2.5.1 Methods

At the same time, in solving Reinforcement Learning problems, a variety of algorithms and variants are used. Those options are based on the nature of the problem itself, but also other parameters, such as the complexity of the problem, the availability or otherwise of data, the processing power of our system and others. Some of the predominant methods are described below.

2.5.2 Monte Carlo

Unlike previous methods, the Monte Carlo methodology is not required to be aware of the environment but only experience from interaction with the environment. The main difference with Markov's methodology is that in this case there are multiple situations that all behave like different problems related to each other.

In particular, the Monte Carlo base is subject to the idea of the average value of the results of many experiments. So also, in Reinforcement Learning problems, setting a state-value function and a particular policy, after many experiments, we end up with many returns. For all of those returns, as they express the long-term reward for each of the experiments, we find their average value and the result should be approaching the expected price.

The above logic is further analysed in the above algorithm.

```
Initialize:
        π, V, Returns(s)
Loop:
        Create episode with policy π
        For s in episode:
                G ← return after first s
                Append G to Return(s)
                V(s) ← Average(Return(s))
```

2.5.3 Genetic Algorithms

Genetic algorithms were designed by John Holland in the 1970s and are based on Darwin's ideas of biological evolution through the theory of natural selection. They belong to the wider group of evolutionary algorithms and are used to produce high-quality solutions to optimization or search problems.

The process begins with the creation of a random group of solutions called population. Each of these solutions is characterised by a set of parameters called genes and these compounds are called chromosome where each of the chromosomes is one of the solutions.

Finding the best solutions is done by creating the fitness function, which finds the fitness score of each solution, and the probability to be selected for reproduction is based on this score. During reproduction, the genes to be passed on to the next generation are produced in the following ways:

- Crossover
  The most important of the reproduction methods is to select two solutions and the association of parts thereof for the creation of a new offspring. The choice of these parts by parents is made using the appropriate choice of the crossover point.

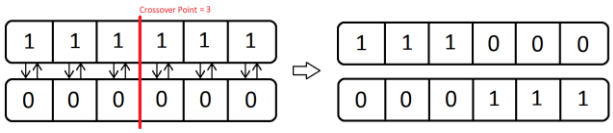

Figure 6: Crossover example

The offspring of that reproduction are added to the population.

- Mutation

  In this reproduction method changes are applied to an ancestor-solution in a random manner, the result of which is the offspring added to the population as well. This method is used to maintain diversity within the population but also to prevent premature generalisation of the result.

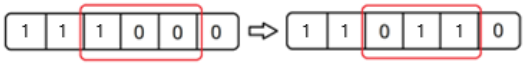

Figure 7: Mutation example

- Elitism

  A frequent occurrence of creating offsprings is the method of elitism, as it allows the new generation to pass the best of the solutions without undergoing any treatment. This method ensures that the quality of solutions from one generation to another is maintained.

Genetic algorithms are often used and most of their applications have very good results. However, this work has not been used as such, since one of their main disadvantages is generalisation at local extremities, especially in problems of great complexity. This means that they do not manage to assess the significance of long-term effects in relation to the short-term, which is contrary to Q-Learning. However, they are used to a certain extent not to find the best solution to the problem, but to find the best structure through which the best solution will be found, which was also analysed in the previous chapter with the description of NEAT algorithm.

Moreover, the final solution of genetic algorithm is the best only in comparison to every other solution of that genetic algorithm, so we are not certain that this solution is also the optimal one.

3. Environment and Simulation

3.1 Rules of the game

In the present work an environment simulation of a chase game between two teams of agents has been carried out. In particular, the blue team has a certain number of movements depending on the dimensions of the area in which they operate, until it catches the rival red team. Similarly, the red team in the same space should avoid blue.

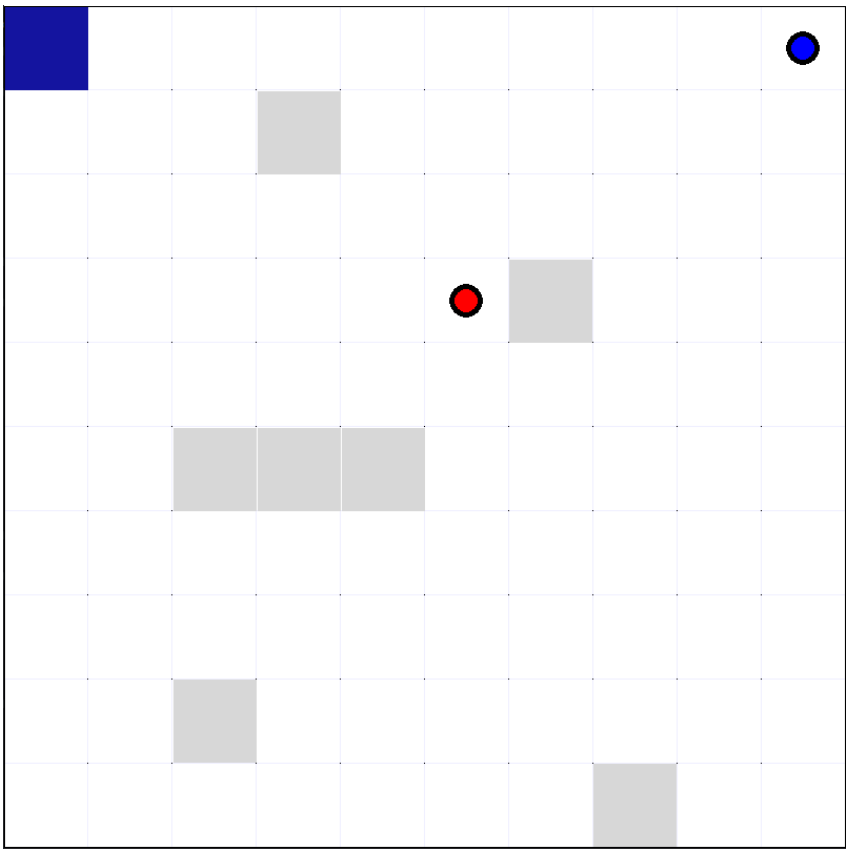

Figure 8: Simulation Environment 10x10 with obstacles

The area where agents act is a grid of multiple dimensions. Internally, the environment can also include obstacles either in the form of walls or covering entire cells. The experiments were developed in three different areas for variety. Those areas are 5x5, 10x10 and 10x10 with obstacles.

3.2 Simulation Environment Development

The creation of the simulation environment was made using the Python 3.9 programming language. For the development of the graphic interface, the pygame library was used, whereas the OpenAI library was used for the environmental structure.

OpenAI Gym is an environment for creating trainable agents. The reason chosen for the creation of the environment is easy integration with a wide variety of RL algorithms, as well as their meaningful comparison without major amendments to the code from implementation to implementation. This library has been used in many RL projects, such as Atari and MujoCo, but also Natural Language Processing projects, Simple Classification Problems, Game Theory, etc.

In addition to the existing environments provided by Gym, it allows to create custom environments and connect them to its interface. Some of the main methods he uses are:

- gym.make(env_name): Environment creation from the existing predefined environment. The development has been in another python script using pygame and is called through MapView2D class.
- env.reset(): Resets the environment in its initial state.
- env.render(): Builds and renders the environment.
- env.step(): Executes an action at every step. It is the most important method as it carried the learning algorithm along with the reward logic.

At the end of env.step (), four parameters are to be returned:

- observation: state of the environment at that moment
- reward: reward gained based on the previous action
- done: boolean value indicating the end of an episode
- info: diagnostic information that helps in debugging

In each step, the agent selects an action, and the env.step() returns the observation and the reward. The above implementation has been developed for a different number of experiments and different methodologies developed within class MapEnv(). Similarly, the design of the graphic environment in pygame and the rules of the game were developed in class MapView2D.

Finally, the code map_generator.py for the creation of new areas has been developed to carry out new experiments, but also a config.py script where the desired parameters for the train and test phase are recorded, like training algorithm, area selection, experiment goal, multi-processing operation, etc.

In total, the sub-sections are linked in accordance with the diagram below.

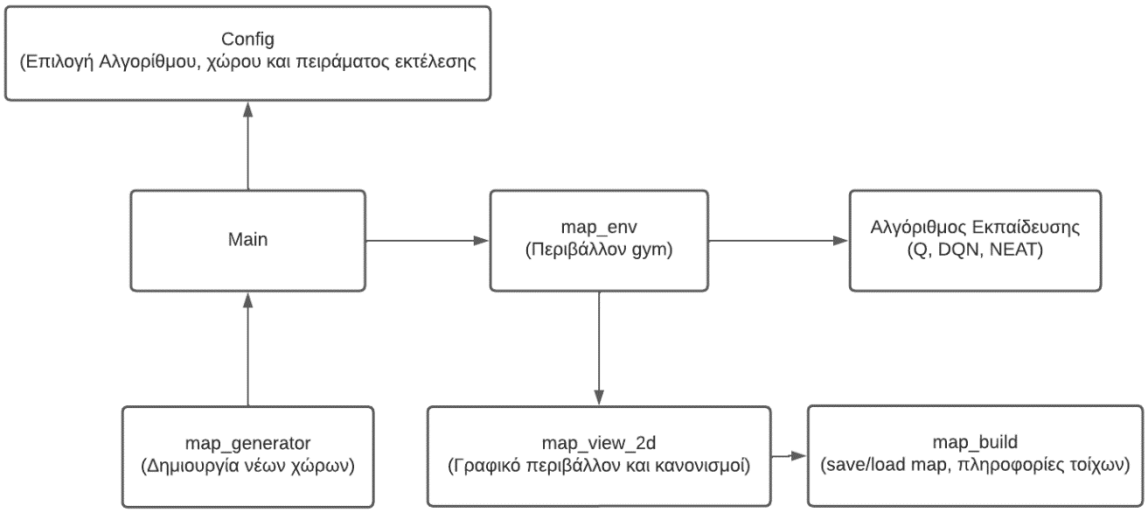

Figure 9: Simulation Code Development Diagram

4. Development

4.1 Algorithm Selection and Experiment Phases

Among the abovementioned Reinfocement Learning algorithms, the choice for the execution of the experiments was based on the nature of the problem and the complexity of the processes that the agents are required to carry out, all according to the processing capacity available to us. The algorithms selected and implemented in depth are Q-Learning, Deep Q-Learning and NEAT.

For each of the above algorithms, several increased complexity experiments were carried out to draw conclusions as to their comparison at each stage of the training. The phases of the experiments are as follows:

Experiment Phases:

- Static: Training the blue team to catch the red, which changes random positions and stays still in every game.
- Random Movement: Blue team training to catch the red one. The red team randomly moves in every step of the episode.
- 2Players: Simultaneous training of both teams.

4.1.1 Inputs – Outputs

Formatting the input and output into each algorithm is subject to a pre-processing. Initially, the input can be expressed in the positions of the two groups after the necessary pre-processing. At the phase of the experiments, input was given coded, either in one-hot encoding or normalized to 1, the coordinate values for each group. After the experiments phase, no change was made in relation to the coding, so input into the system was given as it stands. In more detail, the statements of the agent and the actions it performs per step can be expressed as follows:

Inputs:

- CHASER observation = [x_chaser, y_chaser, x_runner, y_runner]
- RUNNER observation = [x_runner, y_runner, x_chaser, y_chaser]

Outputs:

- CHASER action = ["N", "S", "E", "W"]
- RUNNER action = ["N", "S", "E", "W"]

In the early phases of the experiments, only the input and output of the blue group (chaser) were used. In the training experiments with the two groups, the initial aim was to train both groups at the same time and later to be trained in succession.

A challenge that had to be addressed is the behaviour of the agents with the obstacles. A thought about the recognition and treatment of obstacles is to add the positions of the obstacles to the input, as far as obstacles occupying a whole cell are concerned. Both groups needed more input to be able to interact with them, thus raising the space and time of education.

For the purposes of these conditions, the observation state and the action state have changed. This has greatly affected the complexity of the algorithms concerned, especially Q, as the scale of the table is greatly increased.

4.1.2 Rewards

An important parameter for conducting experiments is to maintain the reward for each phase of the problem. The reward given to the system for each of its action and its next state is to be the same in every different methodology.

The reward_logic() method was performed on every step of the training and, in all phases of experiments, the logic was quite simple as it was given a reward = 100 in the blue group if they catch the red, reward = -1 for every valid step and reward = -5 for each invalid step (step to wall or obstacle leading to a failure to change its status).

The differentiation of the negative reward for the validity of the step has been made on the reason that irregular movements of the agent have been observed several times, resulting in delaying the exploration and at the same time increasing the time of education. These movements were also considered unnecessary and should avoid them.

Similarly, the red team was taking a reward = 10 for every valid step, reward = 2 for every invalid step and reward = -100 each time it was caught by the blue team.

4.2 Experiments with Q-Learning

The first learning algorithm selected was Q-Learning. One of the main reasons was finite and relatively small, depending on the experiment, area for the environment. Therefore, the table of Q-values to be created will be small, so the training time is reduced.

Thus, in the first phase of the experiments and for a 5x5 area, Q-table that is initialised is 625x4.

4.2.1 Hyperparameters

The parameters used for all phases of the experiments are detailed below:

Number of episodes – (NUM_EPISODES)	30000
Max number of steps per episode – (MAX_T)	maze_size * 4
Number of steps for success (Streak) – (SOLVED_T)	(maze_size * 4) / 2
Number of successes (Streaks) to end training – (STREAK_TO_END)	100
Learning Rate (learning rate)	0.2
Exploration Rate (explore rate)	0.001
Discount Factor	0.99

Table 1: Hyperparameters of Q Learning

The selection of the hyperparameters was performed after many tests and changes, particularly between MAX_T and SOLVED_T. More detailed, in every episode, the agent has to make MAX _ T steps before the episode ends and SOLVED _ T steps so that the episode can be considered a success. This is a way of perceiving that the system is properly trained, as if an agent achieves 100 streaks, that is, the target before SOLVED_T steps, then the training stops and the Q table is considered to be properly updated.

4.2.2 Basic Algorithm Structure

The basic structure of Q-Learning algorithm is the following:

```
For episode in NUM_EPISODES:
        env.reset()
        For t in MAX_T:
                action ← select action from Q table
                observation, reward, done = env.step(action)
                Update Q table
                If STREAK_TO_END:
                        Save Q table and end training
```

The above logic was developed in the first two phases of the experiments.

In the experiment carried out by both groups (2Players), the same logic developed in each step for each of the agents. During the training, any agent who was able to achieve the necessary streak of success, his painting was stored, and the training continued. Finally, two Q-tables were created, which in the testing phase took turns acting.

4.3 Experiments with Deep Q-Learning

During the experiments with Q-Learning, many weaknesses were observed in training, mainly due to the exponential growth of time but also the size of Q-table and the complexity of the experiments. The solution to these weaknesses was the conversion of Q into Deep Q. Replacing the Q-table with a Dense neural network greatly reduced the learning area and the number of weights to be trained. In order to carry out the experiments, the training code developed using the rl.agents and keras library.

4.3.1 Hyperparameters

Below are presented the hyperparameter which were used in all phases of the experiments:

Number of episodes – (num_episodes)	100-200
Max number of steps per episode – (num_steps)	maze_size * 3
Warmup Steps	500-5000
Replay Memory	50000-200000
Learning Rate (learning rate)	0.0001
Epsilon	0.1
Gamma	0.95
Loss Function	mse

Table 2: Hyperparameters of Deep Q Learning

4.3.2 Basic Algorithm Structure

The basic structure of Deep Q-Learning algorithm is the following:

```
Initialize replay_memory, Q model weights
For episode in num_episodes:
        env.reset()
        For t in num_steps:
                If epsilon: action ← random, else: action ← select action from model
                observation, reward, done = env.step(action)
                Store in replay memory
                Sample random mini-batch from replay memory
                Gradient Descent to update weights
        Copy Prediction Network to Target Network
```

As far as the 2Players experiment was concerned, two substations of the above implementation were created and were trained in different times. First became the training of the blue team and then the red team. While red was training, the blue moved according to the model previously trained. Learning was not done at the same time as suggested in Q methodology. The reason why they were selected to take individual training is the obvious head start of the red group during the simultaneous training, so that the blue team does not manage to generalize properly.

4.3.3 Networks Structure

The main differences between the static structures of the networks were not affected by the size of the area. The complexity of the network was primarily affected by the goal of the problem, which helped to reduce the amount of networks, depending on the experiment to be carried out.

In more detail, the network structures for the blue group are set out below.

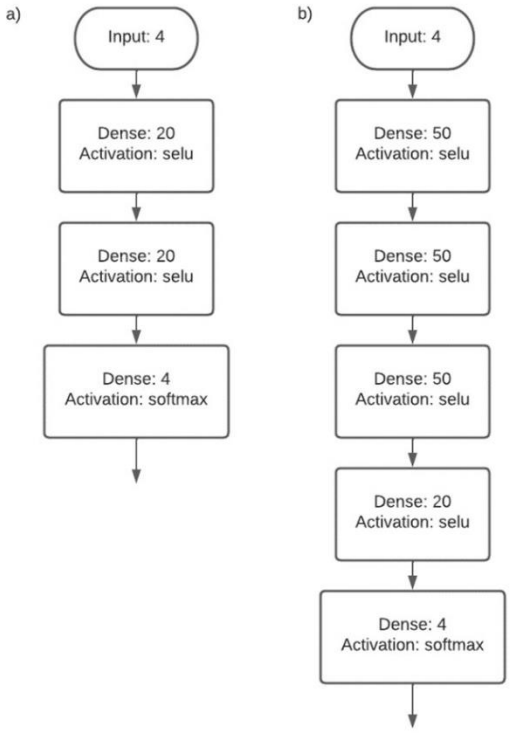

Figure 10: Neural Network Structures of Blue team a) Static, b) Random + 2Players

A major problem emerging at the end of the experiments is the difficulty of finding suitable networks but also the need to modify them until the best results are obtained, in combination with the choice of appropriate parameters.

4.4 Experiments with NEAT

The latest methodology implemented was the NEAT algorithm. Theoretically, NEAT is the most promising learning algorithm from all three. This is because it overruns in speed and volume from Q but also allows using neuroevolution the development of optimal, or near optimal, neural network topology. The second reason is also a great advantage in relation to Deep Q, which the basic structure of its network relates to a static structure which has been found after many tests and experiments of different topologies. The development of the code and learning algorithm was developed using the neat-python library, designed by the CodeReclaimers.

4.4.1 Hyperparameters

In NEAT, the following hyperparameters have been used for all phases of the experiments.

Number of generations – (generations)	<10000
Number of networks per episode – (runs_per_net)	maze_size * 3
Fitness Threshold	Μπλε:95, Κόκκινη:220
Replay Memory	50000-200000
Learning Rate (learning rate)	0.0001
Epsilon	0.1
Gamma	0.95
Loss Function	mse

Table 3: Hyperparameters of NEAT

4.4.2 Basic Algorithm Structure

The basic NEAT algorithm that was implementd using neat-python library, is the following.

```
Initialize population, fitness_function
Initialize genome from inputs,outputs
Initialize feedforward_net
For runs in runs_per_net:
        env.reset()
        observation, reward, done = env.step(action)
        next_action = net.activate(observation)
        calculate fitness
        evolve population through Crossover, Mutation
        divide population into species
```

As in previous experiments, in the 2Players, training has been carried out separately and simultaneously. Finally, separate training was selected for the same reasons as the implementation of Deep Q.

Particular attention should be paid to configuration file of NEAT. Configuration file contains all the important information about the training of agents.

At the below table is a presentation of the most important parameters of the configuration and their brief description.

NEAT		
fitness_criterion	mean	During training, fitness_fuction is calculated from the average of fitnesses per generation
fitness_threshold	95	Goal of fitness_function during evolution
pop_size	200	Size of genome per population
reset_on_extinction	True	Reset algorithm during extinction

DefaultStagnation		
species_fitness_func	max	Fitness calculation method
max_stagnation	20	Max number of generations for a species to be extinct if fitness does not improve
species_elitism	2	Number of protective species

DefaultSpeciesSet		
compatibility_threshold	3	Evaluation of same species based on genome distance

DefaultReproduction		
elitism	1	Protective genome through generations
survival_threshold	0.2	Species reproduction probability

DefaultGenome		
num_inputs	4	Number of inputs per network
num_hidden	0	Number of hidden nodes
num_outputs	4	Number of outputs per network
activation_default	clamped	Basic activation function
activation_options	sigmoid	Optional activation function
activation_mutate_rate	0.2	Probability of optional activation functions

Initialization of Weights and Biases		
bias_init_mean	0	
bias_init_stdev	1	
bias_replace_rate	0.1	
bias_mutate_rate	0.7	
bias_mutate_power	0.5	
bias_max_value	30	
bias_min_value	-30	
weight_max_value	30	
weight_min_value	-30	
weight_init_mean	0	
weight_init_stdev	1	
weight_init_type	gaussian	
weight_mutate_rate	0.8	
weight_replace_rate	0.1	
weight_mutate_power	0.5	
aggregation_default	sum	
aggregation_options	sum	
aggregation_mutate_rate	0.01	
compatibility_disjoint_coefficient	1	
compatibility_weight_coefficient	0.5	

New nodes and connection settings		
conn_add_prob	0.5	Add connection probability
conn_delete_prob	0.5	Remove connection probability
node_add_prob	0.2	Add node probability
node_delete_prob	0.2	Remove node probability
enabled_default	True	New connection activation
enabled_mutate_rate	0.01	Activation or deactivation of mutated connection probability

Initialization of outputs in network		
response_init_mean	1	
response_init_stdev	0	
response_init_type	gaussian	
response_replace_rate	0	
response_mutate_rate	0.01	
response_mutate_power	0	
response_max_value	30	
response_min_value	-30	

Connection options		
initial_connection	full	Node connection method
feed_forward	True	Feed forward method

Table 4: Config file parameters

5. Results

5.1 Q Results

The first conclusion resulting from the completion of all experiments was the ineffectiveness of the simple Q in complex problems. Although the time of training in small-scale problems was quite small, the more complex the problem became, the more difficult it was to generalize as to the solution to the problem.

This becomes quite evident in the table below. The times in the problems of two simultaneous training expresses the end of the first save of the first successful training.

Map	Time (hh:mm:ss)	Episodes	Max Episode Reward	Problem
5x5_empty	00:00:41	226	100	Static
	00:08:23	935	95	Random
	00:45:23	14586	80/20	2Players (Chaser/Runner)
10x10_empty	00:28:38	3532	85	Static
	07:12:01	25454	75	Random
	-	-	-	2Players
10x10_obst	00:31:15	3951	85	Static
	08:18:45	25810	70	Random
	-	-	-	2Players

Table 5: Q Learning Results

At the end of each execution, the algorithm ran 20 test episodes to calculate the accuracy. An episode is considered complete successfully if the agent achieves its target within a specified number of steps. This number varies between the agents, as each one acts as a basis for his own reward. In particular, the blue team successfully completes its target in less than (maze_size * 2) steps, and the red if it is not caught by the blue for more than (maze_size * maze_size) steps.

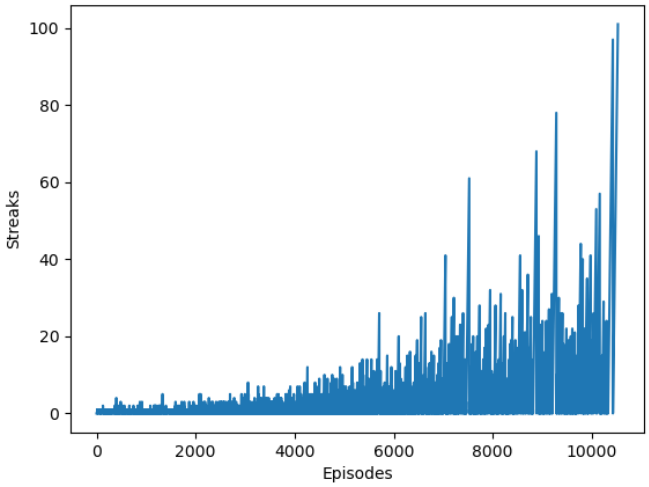

Figure 11: Consecutives successful tries (Streaks) per episode

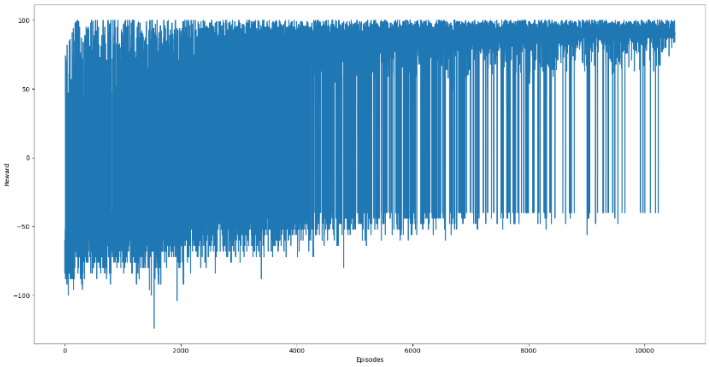

Figure 12: Reward per episode

In small areas like 5x5, Q had quite good results. If the area grew larger, the training time increased exponentially. By increasing the complexity and the scale of the Q-table, the algorithm could not generalize to the solution of the problem.

For example, in the area 10x10 without obstacles to the 2Players experiment, the maximum observation state for each agent is [10, 10, 10, 10] and action [4]. So, we end up training two Q-tables of 10000x4. In this example, the algorithm failed to reach a solution of up to 30000 episodes. This leads us to the conclusion that even if we increase the number of episodes and achieve the desired solution, it will hardly be able to generalize in experiments of increased complexity or larger areas.

In addition, increased computing power systems are required to record relevant experiments as the training of such an agent can last even days.

In conclusion, Q is a quick and effective algorithm to solve simple structured problems. By increasing complexity, however, resorting to other methodologies is necessary.

5.2 Deep Q Results

Regarding Deep Q, experiments certainly were better overall and vindicated its choice as an improved version of the simple Q-Learning algorithm. By increasing complexity, experiments ran into a reasonable time and comparatively much smaller than Q. An important factor was the fact that no table was made during training, but a network. The number of weights is much less than the size of a Q-table, so it makes it very much more trustworthy in testing.

For example, the results of the experiments, as well as their training times and maximum reward per episode, are presented.

Map	Time (hh:mm:ss)	Episodes	Max Episode Reward	Problem
5x5_empty	00:02:34	7	98	Static
	00:18:56	13	89	Random
	00:28:21	12	90	2Players Chaser
	00:28:21	17	212	2Players Runner
10x10_empty	00:07:15	10	95	Static
	00:27:55	26	86	Random
	00:45:16	14	89	2Players Chaser
	00:45:16	89	157	2Players Runner
10x10_obst	00:15:10	20	94	Static
	00:58:13	29	76	Random
	02:56:17	26	79	2Players Chaser
	02:56:17	54	221	2Players Runner

Table 6: Deep Q Results

In the above table, Max Episode Reward expresses the maximum reward by the average agent per episode. According to the rewards given, the most likely reward for the blue team is 100, whereas the red team is 250.

In experiments there was a slight superiority of the chaser, the blue team. This may be due to the use of the same network structure for both agents. The goal of the red group is more difficult as it is to face an already trained system, so a more complex model would have better results.

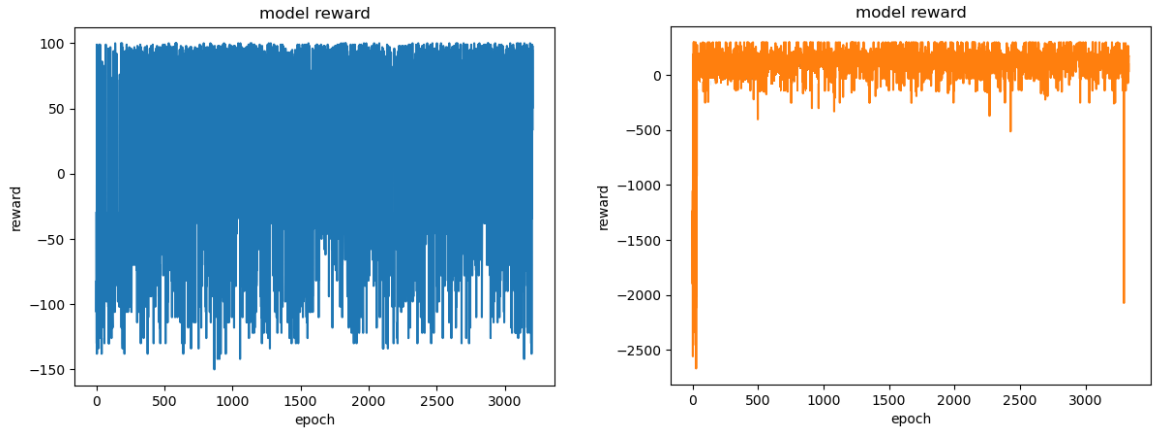

Figure 13: Model Reward Chaser-Runner during training at 10x10 area

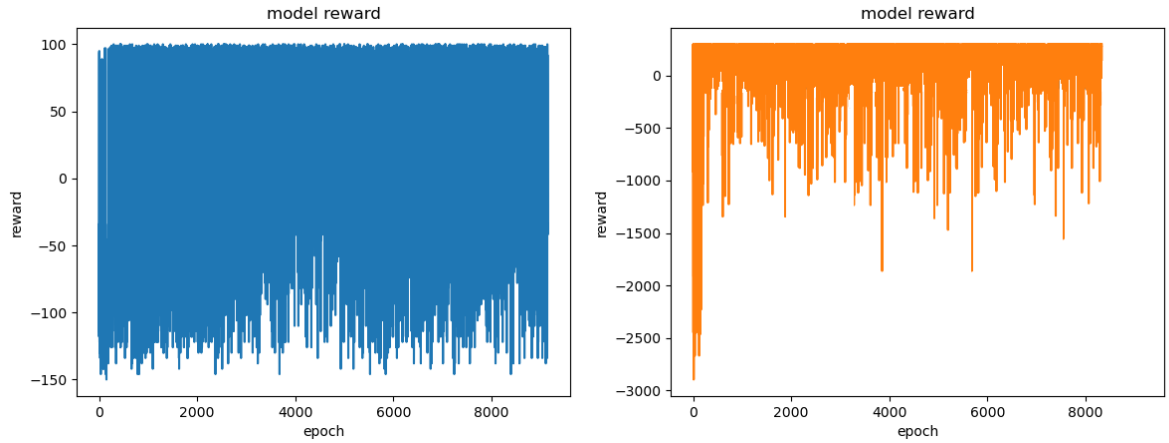

Figure 14: Model Reward Chaser-Runner during training at 10x10 area with obstacles

In addition, in the graphs above, the mean reward per epoch is presented, which expresses steps to achieve the target or exceeding the step limit. It is noted that the models actually generalize a lot faster than the epochs they ran into. This may also be due to the randomness of the examples. A failsafe to find the best model is a call-back implementation, a process that is embedded in the fit function of education and stores the best model after each episode until that moment.

5.3 NEAT Results

In comparison with previous algorithms, NEAT produced more satisfactory results. Finding the appropriate network structure was certainly a lead towards Deep Q which was trained in a stable structure. Overall, however, the training time was obviously greater than the rest, but not prohibitive.

The table below shows the results of the experiments.

Map	Time (hh:mm:ss)	Generations	Fitness Result	Problem
5x5_empty	00:00:09	10	98.2	Static
	00:08:59	752	93.7	Random
	00:14:45	761	92.7	2Players Chaser
	00:14:45	520	215	2Players Runner
10x10_empty	00:11:02	851	94.2	Static
	01:28:56	1121	87.6	Random
	02:10:12	1242	86.2	2Players Chaser
	02:10:12	810	180	2Players Runner
10x10_obst	00:32:04	2584	85	Static
	03:45:13	3650	82.4	Random
	04:56:30	3820	81.2	2Players Chaser
	04:56:30	2110	204	2Players Runner

Table 7: NEAT Results

The above results were achieved following a few changes in hyperparameters and tests. It is noted that for Fitness Result of the blue group, for each problem, other than "2Players Runner," the maximum fitness to be achieved is 100. Similarly, for the red group and "2Players Runner," the maximum fitness is 250 as it gets 10 reward for every valid move to a maximum number of movements 25.

It can be noted that as long as the area gets difficult, the chaser's fitness is decreasing. On the contrary, the runner has a fairly good effect, which is obvious as the chaser has not been sufficiently trained to catch the runner in each episode. In addition, the runner appears to have a lead with the addition of the obstacles.

During the experiments, most nodes from the network created have sigmoid activation function. Thus, the config file of NEAT increased the selection rate of the silence function in relation to clamped from 0.2 to 0.5.

The effectiveness of NEAT, particularly to the chaser, has led to an effort to improve the results, seeking optimal parameterisation. A very important change was the diversification of the way in which the reward was awarded. In more detail, for the runner, tests were carried out with the award of a reward = -100 when caught by chaser and reward = 4 for any other movement. In this way, more emphasis is placed on avoiding the opponent even if the movement that the agent is doing is not valid. At the same time, the maximal fitness of the runner changes from 250 to 100, as it now achieves a reward = 4 for a maximum number of steps 25.

Map	Time (hh:mm:ss)	Generations	Fitness Result	Problem
10x10_obst	04:21:40	1081	89.5	2Players Chaser
	04:21:40	2412	85	2Players Runner

Table 8: NEAT Results after parameterization

Thus, we achieved 8% improvement of fitness result for the chaser and 4% improvement for the runner.

The network structures for the last two experiments (2Players chaser, 2Players Runner) as well as the corresponding graphs are shown below:

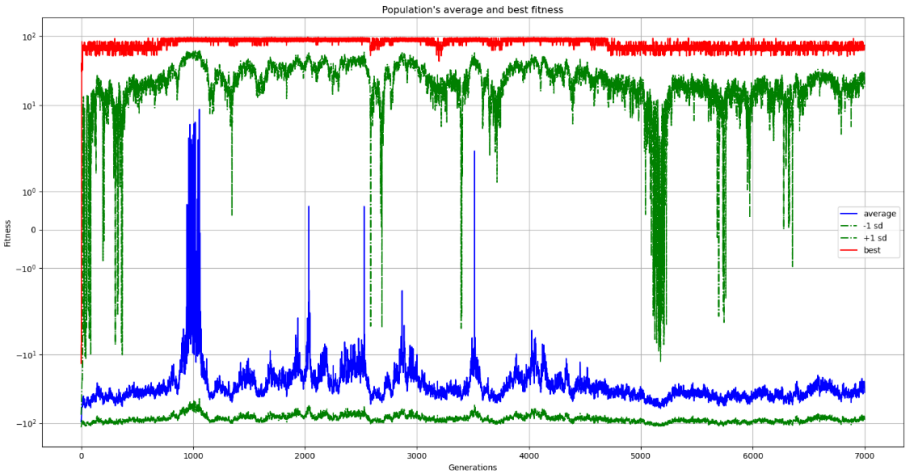

Figure 15: Mean and Max Reward during Chaser training

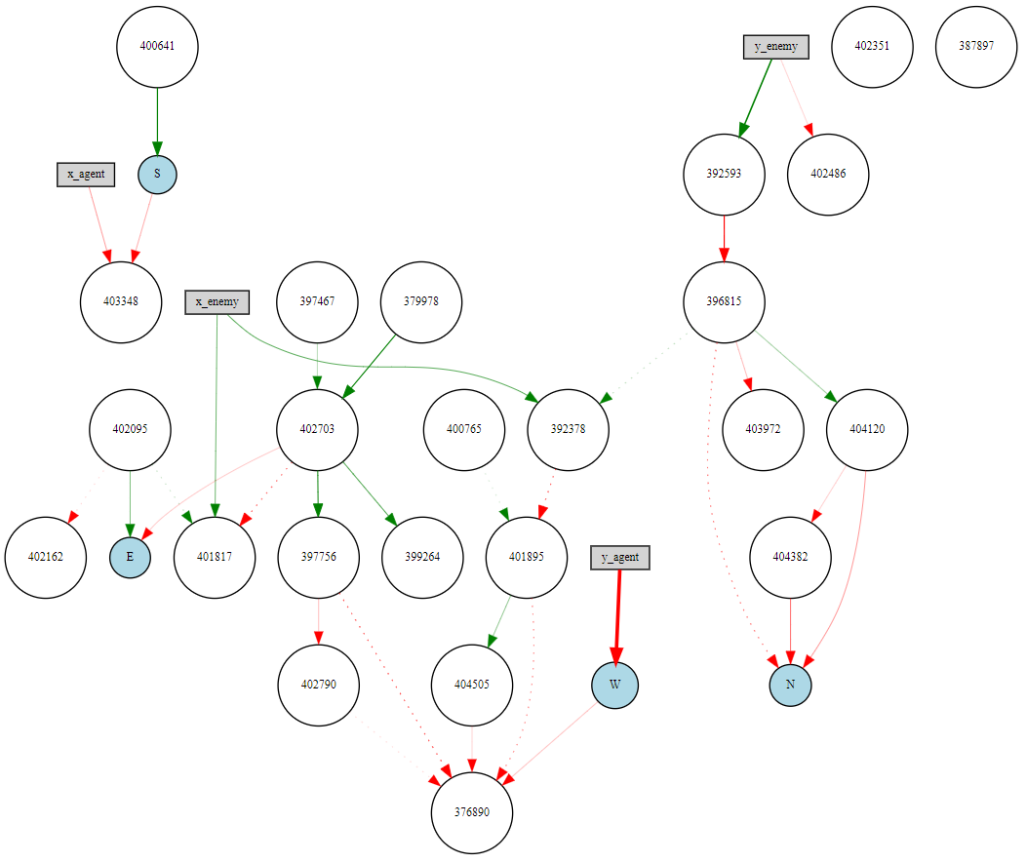

Figure 16: Neural Network Structure for NEAT Chaser

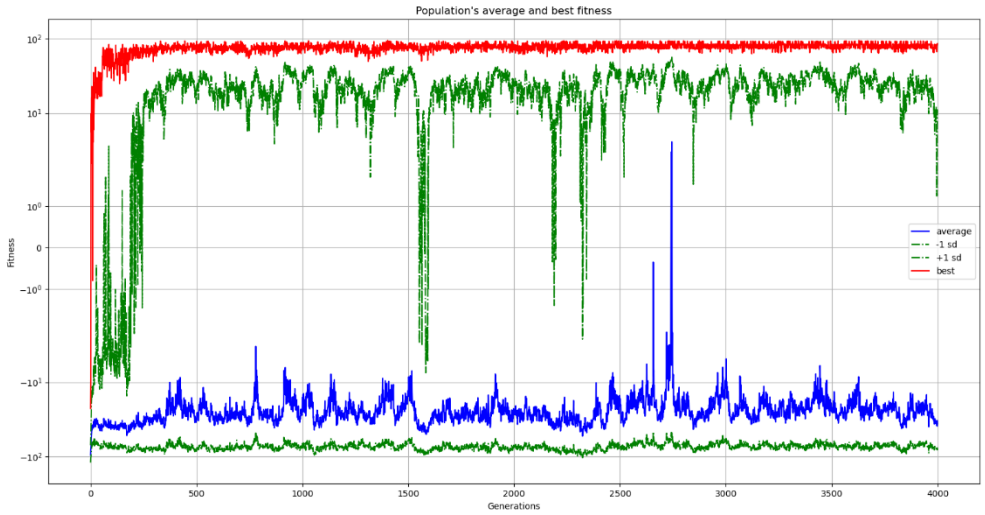

Figure 17: Mean and Max Reward during Runner training

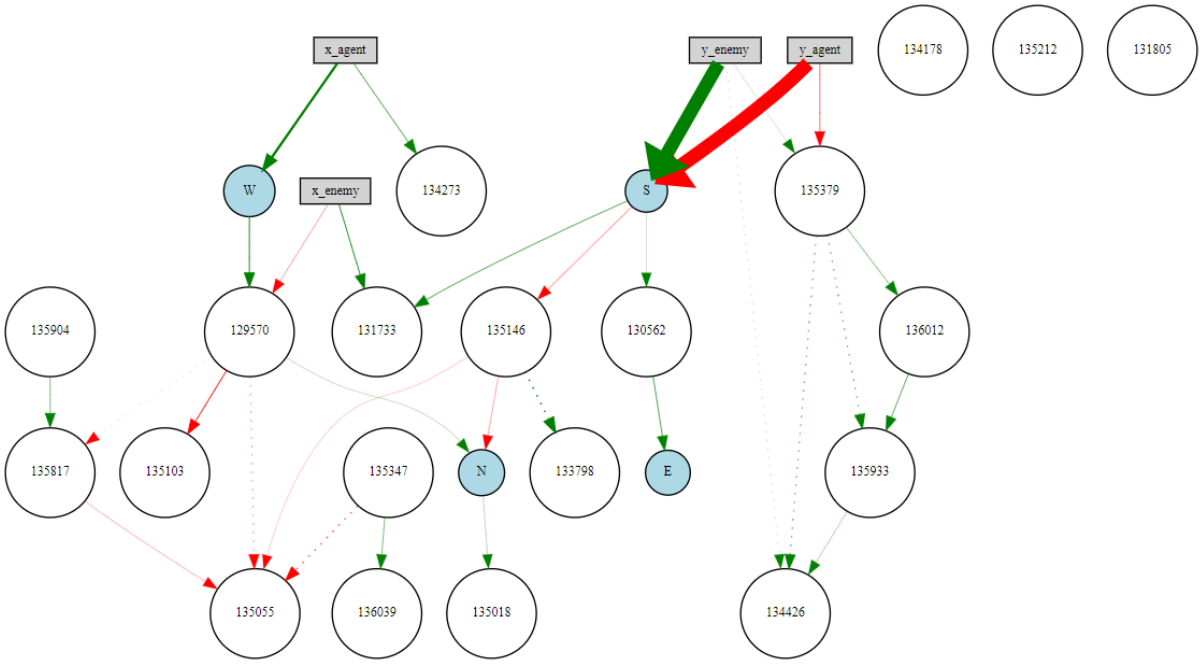

Figure 18: Neural Network Structure for NEAT Runner

5.4 Conclusion and Future Improvements

The general picture presented by the experiments shows the superiority of NEAT across the other algorithms in relation to this problem. Nevertheless, the low processing power in which experiments have been tested, in conjunction with the complexity of the experiments, makes it difficult to set up more complex networks created by neuroevolutionary sequence. Thus, by implementing experiments in systems of greater processing capacity, we would have been leading to better results and therefore in smaller training times, even in implementing experiments such as Q algorithm.

In addition, a majority of experiments have implemented using feedforward networks. A change that could yield improved results is the use of convolutional neural networks. In particular, in the Deep Q-Learning methodology, replacing Dense networks with Conv is a method that has been tested at times, for example the creation of an integrated Reinforcement Learning system in Atari games. As convolutional networks receive 2d inputs, the corresponding encoding should be implemented, in example conversion of them into a two-dimensional space or in the form of an image of the environment by different status (pixels), or to feed the network by a representation of the space per cell (10x10 2d array with a different value depending on the condition of each cell for 10x10).

A change in the system for already existing network structures could also be an alternate encoding. Different encodings have been tested in this study for the input of the observation into the network. One of them is the one-hot encoding, which transforms the network input into a one-dimensional table with 0 and in one of them we have 1. Each of these encoders is unique. The conclusions from this codification are the delay in the achievement of the target, but the results were quite like those analysed and in the previous section, posing the appropriate prospects for improving them. It is therefore a question of finding an appropriate encoding that will improve the present results.

This work was a learning approach using three different algorithmic logic. The results that emerge are desirable, but the relevant literature on similar techniques is very large. Other relevant learning pathways could be used in future research that can deliver the best results.

References

Andrew G.Barto, Richard S. Sutton. 2018. *Reinforcement Learning - An Introduction.*

Andrew Ng, Daishi Harada, Stuart Russell. n.d. “Policy invariance under reward transformations: Theory and application to reward shaping.”

Banzhaf, Wolfgang. 1998. “Genetic programming : an introduction on the automatic evolution of computer programs and its applications.”

Beakcheol Jang, Myeonghwi Kim, Gaspard Harerimana, Jong Wook Kim. 2019. “Q-Learning Algorithms: A Comprehensive.”

Bonate, Peter L. 2001. “A Brief Introduction to Monte Carlo Simulation.”

Bowen Baker, Ingmar Kanitscheider, Todor Markov, Yi Wu, Glenn Powell, Bob McGrew, Igor Mordatch. 2019. “Emergent Tool Use From Multi-Agent Autocurricula.”

Chan, Matthew. n.d. “gym-maze.”

Choudhary, Ankit. 2019. *A Hands-On Introduction to Deep Q-Learning using OpenAI Gym in Python.*

CodeReclaimers. 2015-2019. *NEAT Overview.*

Dimitri P. Bertsekas, John N. Tsitsiklis. 1996. “Neuro-Dynamic Programming.”

Foy, Peter. 2021. *Deep Reinforcement Learning: Guide to Deep Q-Learning.*

Kenneth O. Stanley, Risto Miikkulainen. n.d. “Evolving Neural Networks through.”

Kung-Hsiang, Huang (Steeve). 2018. *Introduction to Various Reinforcement Learning Algorithms. Part I (Q-Learning, SARSA, DQN, DDPG).*

Mahoney, Chris. n.d. *Reinforcement Learning.*

Miikkulainen, Kenneth O. Stanley and Risto. n.d. “Efficient Evolution of Neural Network Topologies.”

Nicholson, Chris. n.d. *A Beginner's Guide to Deep Reinforcement Learning.*

Schmidhuber, Jurgen. 2014. “Deep Learning in Neural Networks: An Overview.”

Scholz, Jan. 2019. “Genetic Algorithms and the Traveling Salesman Problem a historical Review.”

Sebastian Lang, Tobias Reggelin, Johann Schmidt, Marcel Müller. 2021. “NeuroEvolution of Augmenting Topologies for Solving a Two-Stage Hybrid Flow Shop Scheduling Problem: A Comparison of Different Solution Strategies.”

Simonini, Thomas. n.d. *Deep Q-Learning with Space Invaders.*

Szepesv´ari, Csaba. 2009. “Algorithms for Reinforcement Learning.”

Tensorflow. 2021. *Introduction to RL and Deep Q Networks.*

Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare, Joelle Pineau. 2018. “An Introduction to Deep Reinforcement Learning.”

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou. n.d. “Playing Atari with Deep Reinforcement Learning.”

Zaremba, Greg Brockman and Vicki Cheung and Ludwig Pettersson and Jonas Schneider and John Schulman and Jie Tang and Wojciech. 2016. “OpenAI Gym.”