2024

PEDRO MIGUEL FIGUEIREDO DA SILVA

AN EVALUATION OF DEEP REINFORCEMENT LEARNING & NEUROEVOLUTION IN STEALTH GAME PROBLEMS

2024

PEDRO MIGUEL FIGUEIREDO DA SILVA

AN EVALUATION OF DEEP REINFORCEMENT LEARNING & NEUROEVOLUTION IN STEALTH GAME PROBLEMS

Dissertação apresentada ao IADE - Faculdade de Design, Tecnologia e Comunicação da Universidade Europeia, para cumprimento dos requisitos necessários à obtenção do grau de Mestre em *Creative Computing and Artificial Intelligence* realizada sob a orientação científica do Doutor Edirlei Soares de Lima professor auxiliar do IADE – Universidade Europeia.

Acknowledgements

I would like to thank my supervisor Doutor Edirlei Soares de Lima for his guidance and support. I am also grateful to my friends and, especially, to my parents for always encouraging me and making it possible for me to do what I enjoy most. Finally, I would like to thank my lovely girlfriend, without whom this thesis would not have been possible.

palavras-chave

Redes Neuronais Artificiais, Aprendizagem Profunda, Aprendizagem profunda por reforço, Neuroevolução, Videojogos Furtivos

resumo

A aprendizagem profunda (AP) é um campo em rápida evolução com potencial para revolucionar a indústria de videojogos. Embora as empresas de jogos tenham feito esforços para desenvolver diversas abordagens de implementação em jogos com recurso à AP, é uma metodologia pouco utilizada na indústria. Uma das principais razões é a falta de diversidade nos géneros de jogos aos quais esses algoritmos são testados e aplicados.

Os jogos furtivos são um dos géneros de jogos mais populares, caracterizados pela necessidade de equilibrar entre ações de fuga e deteção. Dentro deste género, os programadores de jogos recorrem a testes de garantia de qualidade para entender como é que os jogadores interagem com os diferentes níveis furtivos, o que torna o processo demorado. Potencialmente, a AP poderia ser usada para facilitar os processos de desenvolvimento de jogos e torná-los mais eficientes.

A presente dissertação visa investigar se a AP pode ser usada no desenvolvimento de jogos, particularmente em jogos furtivos. Como tal, o projeto foca-se no desenvolvimento de diversos ambientes de jogos de furtivos, otimizando algoritmos de AP para esses ambientes, comparando diferentes algoritmos de AP e medindo o custo computacional desses algoritmos.

Para alcançar os objetivos definidos, o projeto apresenta a seguinte metodologia: desenvolvimento de diversos cenários de jogos furtivos, implementação de modelos de AP de última geração, realização de ajuste de hiperparâmetros, avaliação de desempenho e realização de benchmarking computacional.

Os resultados obtidos mostram que, na maioria dos casos, os algoritmos de AP implementados foram capazes de aprender os diferentes níveis de jogos furtivos, demonstrando a plasticidade e adaptabilidade dos algoritmos implementados. Além disso, uma avaliação mais profunda de desempenho dos algoritmos mostrou distinções claras sobre quais algoritmos de AP com melhores desempenho, nomeadamente o algoritmo Rainbow-DQN dentro da aprendizagem por reforço profundo e o algoritmo GA para neuroevolução.

Apesar dos resultados bem-sucedidos, é necessária uma pes
mais profunda de modo a compreender melhor as possibilic
as técnicas de AP podem oferecer no desenvolvimento de
videojogos, dentro do género de jogos furtivos.

Keywords Artificial Neural Networks, Deep Learning, Deep Reinforc
Learning, Neuroevolution, Stealth Games.

abstract Deep learning (DL) is a rapidly evolving field with the potential to revolutionize the videogame industry. While there have been efforts by game companies to research DL approaches to game implementation, very little has been used in the industry. One of the main reasons is the lack of diversity in the game genres to which these algorithms are tested and applied to.
Stealth games are one of the most popular gaming genres, characterized by the need to balance both states of sneaking and detection. Within this genre, game designers have traditionally relied on quality assurance testers to understand how players interact with stealth levels, which can be a time-consuming process. Potentially DL could be used to facilitate the game development processes and make them more cost-effective.
This project aims to investigate whether DL can be used in game development, particularly in stealth games. As such, the project will focus on developing diverse stealth game environments, optimizing DL algorithms for these environments, comparing different DL algorithms, and measuring the computational cost of these algorithms.
To achieve the stated objectives, the project follows a structured methodology: developing diverse stealth game scenarios, implementing state-of-the-art DL models, conducting hyperparameter tuning, performing performance evaluation, and conducting computational benchmarking.
The results obtained from following this methodology show that in most cases, the implemented DL algorithms were able to learn all the different stealth game levels, indicating the strength and flexibility of the implemented algorithms. Also, a deeper evaluation of the algorithm's performance showed clear distinctions into which DL algorithms provided better performances, namely the Rainbow-DQN algorithm within the deep reinforcement learning and GA algorithm for neuroevolution.
Regardless of the successful results, further research is nee better comprehend the possibilities that the DL techniques to game development in the context of stealth games.

Table of contents

List of Figures

Glossary

ADAM – Adaptive Momentum

AI – Artificial intelligence

ANN – Artificial Neural Networks

CMA-ES – Covariance Matrix Adaptation Evolution Strategy

DL – Deep Learning

DQN – Deep Q Network

DRL – Deep Reinforcement Learning

FIFO – First In First Out

GA – Genetic Algorithms

IID – Independent Identically Distributed

Lr – Learning rate

MAE – Mean Absolute Error

MARL – Multi-Agent Reinforcement Learning

MDP – Markov Decision Process

ML – Machine Learning

MSE – Mean Squared Error

NEAT – Neuroevolution Augmenting Topologies

NPCs – Non-Player Characters

NS – Novelty Search

PCG – Procedural Content Generation

PER – Prioritized Experience Replay

QA – Quality Assurance

ReLU – Rectified Linear Unit

RL – Reinforcement Learning

RMSprop – Root Mean Squared propagation

RS – Random Search

SGD – Stochastic Gradient Decent

TD – Temporal Difference

1 – Introduction

The world of video games has long been synonymous with providing new experiences for its players, which is achieved by combining several artistic and technical fields. The games industry is constantly innovating in storytelling, art design, UI/UX interactions, graphic detail, and performance, among others. Interestingly, a field that exists in almost all games that has not seen the same amount of innovation is the artificial intelligence (AI) area.

In the last decade, the deep learning (DL) field, has possibly been the most popular topic in the fields of AI and Machine Learning (ML) due to its success in solving regression and classification problems by generalizing large amounts of data. Artificial neural networks (ANN), a data structure based on the biological brain, is the foundation of a DL model, with more complex architectures building on ANN.

While the field of DL has seen a significant evolution in the last decade (Croak & Dean, 2021), it is still barely used in games. Game developers mostly work on AI models that have been around for some time, like finite state machines and behavior trees. The traditional AI systems used by game developers require the scripting of all intended behaviors, where a developer must think and plan for all the possible interactions that an AI agent can do. Some of these interactions might be extremely difficult to develop, and these AI systems are known to get increasingly more complex and difficult to work with as they expand.

In the past decade, research concerning automated learning in game environments has increased, generally involving a type of ML methodology called Reinforcement Learning (RL). RL models consist of AI agents that can analyze their environment so that they take actions that lead them to better future rewards. The more performant and expressive algorithms from RL tend to use ANNs as their most important piece. This combination of methods forms the deep reinforcement learning (DRL) field. DRL models have shown great results in solving several types of videogames (Figure 1), including environments like Atari games, Doom, Minecraft, StarCraft, and car racing games (Shao et al., 2019).

While DRL is a recent and prominent DL approach applied to videogames, another DL technique known as Neuroevolution, has also been utilized in the context of videogames (Yannakakis & Togelius, 2018). Neuroevolution main purpose is to optimize ANNs using evolutionary algorithms to solve specific tasks. Notably, Neuroevolution has a storied and successful track record in learning various types of videogames (Risi & Togelius, 2017).

The success of DRL and Neuroevolution in solving several different game environments has shown game developers that they can be used in several game development problems.

Figure 1

Screenshots of games and frameworks used as research platforms for research in DL (Justesen et al., 2017)

1.1 – Motivation

While there have been some efforts by a few game companies, like Electronic Arts (Gordillo et al., 2021) and Ubisoft, to research how DL can be used in games, there is still very little of it being used in the market, as mentioned by Justesen et al. (2019). There are several possible reasons why this is the case (Togelius et al., 2024), for example, a lack of designer tools for DL, and a lack of developers with expertise in DL.

As mentioned by Justesen et al. (2019), most DL research is trying to optimize the algorithms to perform at a superhuman level in their given environment. For a player, interacting with a "perfect bot", being that competitively or cooperatively is probably not fun, as generally, players prefer an AI that behaves as humanly as possible. Game designers could also benefit from AI agents that learn to play games more like humans, as playtesting this AI would provide insight into how a real human would behave and react to changes in the game levels (Gutiérrez-Sánchez et al., 2021). Leveraging DRL for quality assurance (QA) and game balancing was also proposed by Kim & Wu (2021).

Since one of videogames main concerns is performance, video game developers are always looking to reduce the computational cost of their games. Using an AI model that takes too many of those resources is generally avoided, and DL-based models have the fame of being very slow to run (Justesen et al., 2017), which makes them unappealing for game developers.

The game genres that these algorithms are tested in are generally confined to the environments available in the most popular research platforms, the most notable being OpenAI Gym. Although these platforms provide a way to uniform and compare tests among researchers, for

game development this is not enough, as there exist several types of classic game genres that are not well represented in these research platforms, for example stealth games.

Many stealth games are available on the market, and each one is different from the others, although they all build upon the same core principles. The essence of a stealth game is about balancing two different opposing states, the first one is sneaking where the player is trying to navigate the level while avoiding being seen by enemies, and the other state is detection where the player is compromised and must survive until the game returns to the sneaking state (Cieślak, 2022; Wong, 2014).

Veteran game designers Benjamin Bauer and Randy Smith (Bauer, 2018; Smith, 2006), talk about stealth games level design, in which they mention that designers must rely on QA testers to understand if levels are having the desired effect on players. This time-consuming process happens when a level is created and when there is an alteration, big or small, to that level.

1.2 – Objectives

This project aims to answer the question if DL can be leveraged to facilitate game development in any way, particularly in stealth games.

For game developers to adopt DL into their games, the research in this field must contemplate more relatable game development problems, for example, fun AI for players to interact with, DL models that facilitate QA and game balancing, and more DL exploration of different classic gaming genres. Given the complexity of these challenges, this project adopts a strategic approach of pursuing specific, attainable objectives that contribute to addressing these overarching problems.

Therefore, the primary objectives of this project include:

i. Develop diverse stealth game environments suitable for training and evaluating state-of-the-art DL models.
ii. Establish a comprehensive methodology for optimizing DL state-of-the-art algorithms to effectively learn the created stealth game environments.
iii. Conduct thorough comparisons of state-of-the-art DL algorithms, highlighting their respective strengths and weaknesses in the context of learning stealth game environments.

iv. Accurately measure the computational cost of these algorithms in a game Development context.

By addressing these objectives, this research endeavors to provide valuable insights into the feasibility and benefits of integrating DL techniques into the gaming industry, particularly within the domain of stealth game development.

1.3 – Methodology

To achieve the stated objectives, this project follows a structured methodology that encompasses the following key steps:

i. Conduct thorough research on the state-of-the-art DL algorithms utilized in video games. Based on those findings examine their theoretical and technical components.
ii. Creating Diverse Stealth Game Scenarios: The project begins by developing different stealth game scenarios within the Unity 3D game engine (a popular game engine), incorporating common stealth mechanics found in such games.
iii. Integration of Game Data for DL Models: The implemented game levels are designed to seamlessly integrate with DL models, by collecting real-time game data, providing a comprehensive description of an environment's state.
iv. Implementation of State-of-the-Art DL Models: Various state-of-the-art DL learning models are directly developed from scratch inside the Unity 3D environment.
v. Hyperparameter Tuning: Hyperparameter tuning is conducted on all implemented DL algorithms. This process allows for the exploration of each algorithm's specific characteristics and optimal configurations.
vi. Performance Evaluation: A comprehensive performance evaluation is carried out by comparing and assessing the performance of all algorithms across the range of implemented stealth game scenarios/levels.
vii. Computational Benchmarking: To gauge their practicality, the computational efficiency of all algorithms is benchmarked for common DL tasks, using standard commercial hardware.
viii. Data Analysis: The project culminates in a thorough analysis of the data collected, providing insights and answers to the overarching research question.

1.4 – Document Outline

This document is made up of 6 chapters: Introduction (Chapter 1), Literature Review (Chapter 2), Testbed Environment (Chapter 3), Experimental Setup and Results (Chapter 4), Results Analysis and Discussion (Chapter 5), Conclusion, Limitations and Future Work (Chapter 6).

Chapter 1 gives an overview of the thesis. It contextualizes the relevance of the thesis themes, describes the current problems in the field and how they feed into the author's proposed objectives, and finally, it outlines the methodology taken to achieve those objectives.

Chapter 2 purpose is to detail the theory behind the algorithms used in this thesis. It first gives a classical explanation of RL, then details DL with a large focus on neural network basics. Later it details DRL and all the algorithms used in the thesis. This is followed by an overview of Neuroevolution methodologies with a focus on the algorithms used in the thesis. Finally, this chapter ends with an overview of DL state of the art in games.

Chapter 3 delves into the implementations of the thesis environment and tools created. It first explains how the game environment was created as well as describing its mechanics in detail. Later it gives an overview of the libraries created for the thesis project.

Chapter 4 focuses mostly on describing the experiments conducted as a part of this thesis. Initially, it explains the environment data collection process and outlines the methodologies employed for testing the implemented algorithms. Subsequently, it delves into the process of hyperparameter tuning for each algorithm. Finally, the chapter concludes by summarizing the overall results.

Chapter 5 main purpose is to make sense of the results presented in Chapter 4 and relate them to the proposed objectives described in Chapter 1. In doing so, three main aspects are discussed concerning stealth game environments, DL algorithms optimization, and final algorithm analyses.

Chapter 6 regards the thesis conclusion section, exploring this research main contributions to the video games field, its limitations, and possible future work.

The full project can be found at https://github.com/rorix14/Thesis-Project.

2 – Literature Review

This project focuses on Deep Learning and how it can be applied in stealth games. As such this chapter's purpose is to explain what RL, DL, DRL, and Neuroevolution are, while focusing on the algorithms and methodologies that are used in this project. Finally, a general overview of DL's state of the art in games and a more detailed look at existing research on DL in stealth games is given.

2.1 – Reinforcement Learning

There are three types of ML, supervised learning, unsupervised learning, and reinforcement learning which is the one used in this project. RL is inspired by the idea of learning through trial and error from performing actions in an environment, imitating one of the ways animals and humans learn. For example, when teaching a dog to do certain actions, a common way is to give treats to the dog once it performs the desired action given a command, the hope is that the dog associates the commands with the right actions. For RL, the intention is that an artificial agent, given enough time, can be able to perform similarly to a human in an environment.

This particularity of learning by interacting with the environment lends RL to be useful in many fields. Currently, RL applications are used in games, robotics, finance, healthcare, intelligent transportation, and energy optimization (Lonza, 2019).

To be able to talk more in-depth about RL and its methods, there are a set of base expressions and terms that need to be explained like states, rewards, policy, value function, and model:

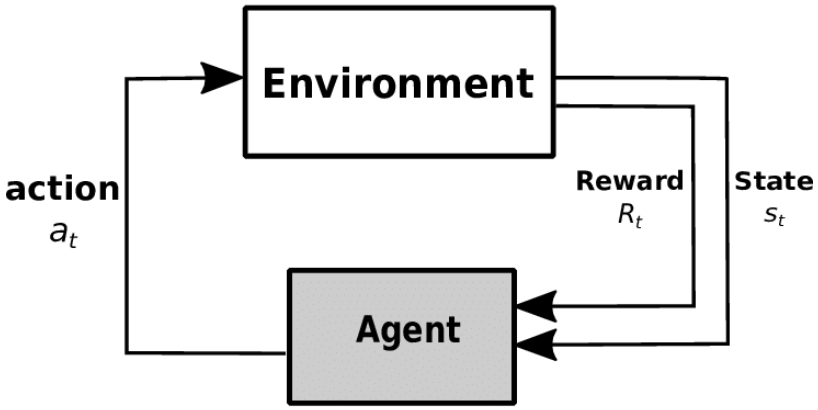
- States – A state is a set of features that represent a moment in an environment, for example in a chess game, a state could be represented by the positions of every piece on the board, or it could be represented by the image pixels of the entire board.
- Rewards – When an agent performs an action in an environment it receives a reward. This reward is a number that tells the agent how good the action was, informing it of his performance. Rewards can be given at different frequencies, if a reward is given at small intervals, it is called a dense reward, if it is given at large intervals, like only at the end of a game, it is called a sparse reward. Some environments can implement both sparse and dense rewards (Lapan, 2020).

- Policy – The policy defines an agent's action at a given state, sometimes referred to as the agent's "brain". In RL the best/optimal policy seeks to perform actions that will lead the agent to gather the maximum number of rewards. Policies can be deterministic $a = \pi(s)$ or probabilistic $\pi(a \mid s)$ (Lapan, 2020).
- Value Function – The value function gives the expected future reward sum (also known as the return) of a state if the agent starts from a given state while following a policy, in other words, the value function gives the expected return for all states for a given policy. This is useful because once the value function has converged it can be used to improve the current policy. If the return from performing an action in a state is known, then other actions can be used to see if they lead to a better result. There are two types of value functions, state-value $V_\pi(s)$ and action-value $Q_\pi(s,a)$, $V_\pi(s)$ is used to do policy evaluation and $Q_\pi(s,a)$ is used to find the optimal policy, also known as the control problem (Lonza, 2019).
- Model – In RL the model of the environment says what are its dynamics, meaning it gives the probability of reaching a state given the current state and an action, usually represented by $p(s' \mid s,a)$. Not all RL algorithms use a model of the environment, making them model-free instead of model-based (Lonza, 2019).

Figure 2 shows the core of most RL algorithms. There is an environment and an agent that performs actions on it so it can receive rewards, the agent also receives the environment state resulting from performing the last action, which will help it perform the next action.

Figure 2

A basic flow of RL (Mehrpouyan et al., 2018)

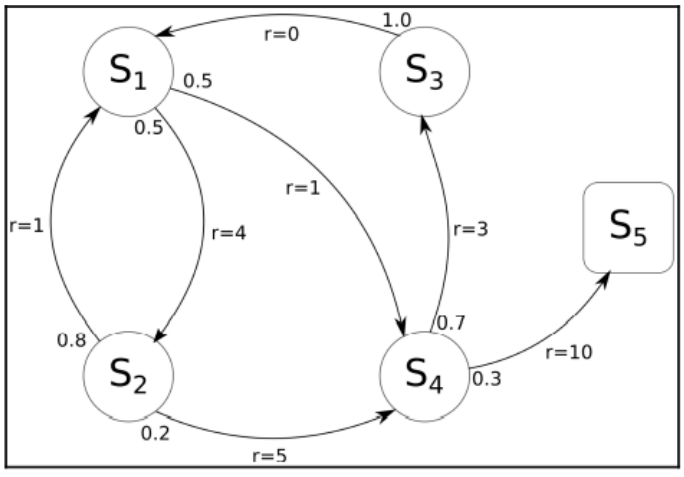


2.1.1 – Markov decision process

Most RL problems can be described as a Markov decision process (MDP), as such, they are the framework of most RL algorithms. MDPs make use of the expressions mentioned above, and a few more, to express a sequential decision-making mathematical model. In an MDP, an agent travels from state to state by performing actions, the probability of arriving at a next state given a state and action is given by the environment model, which in this case is an MDP. Each state transition also has a reward associated with it, this way the agent knows what reward it gets by performing an action at a given state. An MDP thus contains four main components; the state space S which contains the environment states, the action state space A which contains the set of all possible actions, the transition model that defines the probability of transitioning to another state $P(s', s, a) = p(s' | s, a)$, and the reward function R which says the reward collected for every state transition. MDPs also make a very important assumption which says that the transition to the next state only depends on the current state and not the sequence of previously visited states. This is called the Markov property and a system that makes use of this is called fully observable (Lapan, 2020; Palanisamy, 2018). Figure 3 shows a classic example of an MDP, this example contains five states with the final/terminal state being represented by a square. Every state contains transitions to other states, note that the sum of the transition values from a state always sums up to one as they represent probabilities, also every transition has a reward associated with it.

Figure 3

Example of an MDP (Lonza, 2019)



2.1.2 – Return

When an agent runs its policy in an MDP, it creates a sequence of states and actions $(S_0, A_0, S_1, A_1, ...)$ which is called trajectory τ, during this process the agent also collects a sequence of rewards, which is called the return and in its more basic form it can be defined as equation (1) (Lonza, 2019).

$$G(t) = r_0 + r_1 + r_2 + .. = \sum_{t=0}^{\infty} r_t \quad (1)$$

Some environments are continuous and never terminate, leading to infinite-sized trajectories, as such equation (1) of the return no longer fits, as its value will tend towards infinity. To solve this, a value between 0 and 1, called the discount factor, is added to equation (1), becoming equation (2) (Lonza, 2019).

$$G(t) = r_0 + \gamma r_1 + \gamma^2 r_2 + .. = \sum_{t=0}^{\infty} \gamma^t r_t \quad (2)$$

Equation (2) now gives priority to newer values instead of older ones, meaning that an agent will consider more immediate rewards than rewards in the far future. Generally, the discount factor values vary between 0.9 and 0.99.

Environments that are not infinite can be called episodic, and for them, the same return equation (2) is still used, just with one minor modification (3):

$$G(t) = r_0 + \gamma r_1 + \gamma^2 r_2 + .. = \sum_{t=0}^{k} \gamma^t r_t \quad (3)$$

Equation (3) can be put into a more simplified notation (4), which is more commonly used.

$$G_t = r_t + \gamma G_{t+1} \quad (4)$$

2.1.3 – Value Function and the Bellman equation

As mentioned before, the value function allows for policy evaluation, as it calculates the expected return of every state the policy goes through, which can be defined as in equation (5) (Lapan, 2020).

$$V_\pi(s) = E_\pi[G(t) \mid S_t = s] \quad (5)$$

The Bellman equation says that the expected return can be calculated recursively, meaning that the value of a state only depends on the value of the next state. This allows for $V_\pi(s)$ to be updated without the need to unroll the trajectory to the end, shown in equation (6) (Lanham, 2020).

$$\begin{aligned} V_\pi(s) &= E_\pi[r_t + \gamma G_{t+1} \mid S_t = s] \\ &= E_\pi[r_t + \gamma V_\pi(s') \mid S_t = s] \end{aligned} \quad (6)$$

The Bellman equation is used extensively through RL and most algorithms apply their variations on this equation.

2.1.4 – Temporal Difference Learning

Although there exist several methodologies for agents to learn their environments, this work focuses more on model-free approaches. This type of approach assumes that the environment's model is unknown, so the agent must take samples from the environment to estimate the expected return from a given state. The more samples it takes from all possible states the more accurate the estimation of the value function the agent will have. Temporal Difference (TD) is a model-free technique that uses the aforementioned approach, but in its case, it updates the value function on every step, meaning that every time it performs an action in a state and gets a reward, it immediately updates the value function. This technique is called bootstrapping. TD learning makes up the core of many of the most popular RL algorithms like, Q-learning and SARSA (Palanisamy, 2018).

As mentioned before the value function of a policy is the expected return of that policy starting from a given state. Since TD assumes the model of the environment is not known, it can only get an estimation of the value function. To do this the agent collects as many samples of the return as possible and averages them, as demonstrated in equation (7) (Lanham, 2020).

$$V_\pi(s) = E_\pi[G(t) \mid S_t = s] \approx \frac{1}{N}\sum_{i=1}^{N} G_{i,s} \quad (7)$$

TD uses bootstrapping, which means it updates the value function at every step, so if the equation (7) is used, it means that at every step the new return average for that state must be recalculated, which is not optimal. To optimize equation (7), the fact that the average of the return is being recalculated at every step can be used as leverage, equation (7) can be revised to use the previous average and the new sample value to calculate the new average return, seen in question (8) (Lonza, 2019).

$$V_{t+1}(s) = V_t(s) + \frac{1}{N}\left(G_t - V_t(s)\right)$$
$$= V_t(s) + \alpha(r + \gamma V_t(s') - V_t(s)) \quad (8)$$

In practice, a constant alfa is used instead of dividing by the number of samples. The α value usually ranges from 0.5 to 0.001. The objective is to improve the state value by a small amount toward the optimal value.

2.1.5 – Explore-exploit dilemma

Because TD learning relies on environment sampling to find the optimal value function, the probability of choosing any action on any state must be greater than zero, otherwise, convergence is not assured. While exploration is crucial to achieving an optimal value function, there are issues with doing too much of it. For example, reaching an optimal policy might become impossible, as the agent can constantly ignore what it has already learned. This tradeoff between exploration and exploitation creates the explore-exploit dilemma. Ultimately most TD algorithms like Q-learning try to use a balance between both, in the beginning, they do more exploration, and over time they slowly decrease as the optimal policy starts to be discovered.

There are several algorithms used to tackle exploration, probably the most common one (not necessarily the best) is epsilon (ε) greedy (Salloum, 2019). In ε-greedy the agent follows the policy $1 - \varepsilon$ amount of times, otherwise, it will perform a random action. So, if $\varepsilon = 0.3$, then on average, for every 10 actions, the agent will randomly choose an action 3 times. To avoid exploring too much at later stages it is common to decrease the value of ε over time, doing this makes sure that a policy can converge to a deterministic optimal policy. This strategy is called epsilon-decay.

2.1.6 – SARSA and Q-learning

Despite TD being used to solve for the value function, it cannot be used to improve and find the optimal policy. As it is, TD only gives the value of a state, this alone is not enough if an agent needs to decide what action gives the best return on a given state. For that, the agent needs to learn and apply the state-action function $Q(s, a)$. The state-action function maps states to actions, it tells of taking an action a on a state s. SARSA and Q-learning are both TD methods that implement the state-action function. SARSA just applies the state-action function to the TD equation (9) (R. Sutton & Barto, 2018).

$$Q_{t+1}(s, a) = Q_t(s, a) + \alpha(r + \gamma Q_t(s', a') - Q_t(s, a)) \quad (9)$$

The name SARSA comes from; the current state s, the current action a, the reward r, the next state s', and the next action a'. SARSA is an on-policy algorithm, meaning that the policy that collects samples from the environment (behavior policy) is the same that is being updated (target policy) while the agent is learning.

Q-learning differs from SARSA only in one way, it always takes the maximum state-action value, as can be seen in equation (10).

$$Q_{t+1}(s, a) = Q_t(s, a) + \alpha(r + \gamma \max_{a'} Q_t(s', a') - Q_t(s, a)) \quad (10)$$

This particularity makes Q-learning an off-policy algorithm, meaning that the policy used for exploration might not be the same that is being updated. In the case of Q-learning, the agent might not always choose the best next action, but it always updates the state-action function as if it did so (R. Sutton & Barto, 2018). The pseudocode for the Q-learning algorithm can be seen in Table 1.

Table 1

Q-learning algorithm pseudocode (Lonza, 2019)

Q-learning (Off-Policy) Algorithm	
1:	**procedure** Q-LEARNING
2:	Initialize $Q(s,a) = 0$ for all $a \in A$ and $s \in S$
3:	**for** $episode = 1..M$ **do**
4:	$s \leftarrow InitialSate$
5:	**for each** $episode$ **step do**
6:	**Select** a, **based on an exploration strategy, given** s
7:	**Take action** a_t**, observe** r, s'
8:	$Q_{t+1}(s,a) \leftarrow Q_t(s,a) + \alpha(r + \gamma \underset{a'}{max} Q_t(s',a') - Q_t(s,a))$
9:	$s \leftarrow s'$
10:	**if then** $s = terminal$ **then**
11:	$Q_t(s',a') \leftarrow 0$
12:	**end if**
13:	**end for**
14:	**end for**

2.2 – Deep Learning

The concept of deep learning was introduced by Dechter in 1986 and associated with ANNs by Igor Aizenberg and colleagues in 2000 (Zhang et al., 2018). Since then, the term deep learning has become a buzzword, with many authors having their own slightly different definitions (Zhang et al., 2018). Despite this, it is safe to say that the base of DL is ANNs, as such, for this work, DL can be defined as the study of ANNs. With that in mind, ANNs will be the focus of this section.

The concept of ANNs goes as far as 1965 when it was first introduced as a multilayer perceptron by Alexey Ivakhnenko. Despite its early conception, it was only several years later that more serious research on the topic took place (*AI Winter - Integrasi Komputer | Wiki EduNitas.Com*, n.d.). In 1980, research on speech recognition using ANNs returned, but unfortunately failed, being almost forgotten again until the late 90s. It was not until 2012 when Andrew Ng and Jeff Dean used an ANN to recognize cats in videos, that ANNs saw an explosion of interest that persists to this day (Lanham, 2020).

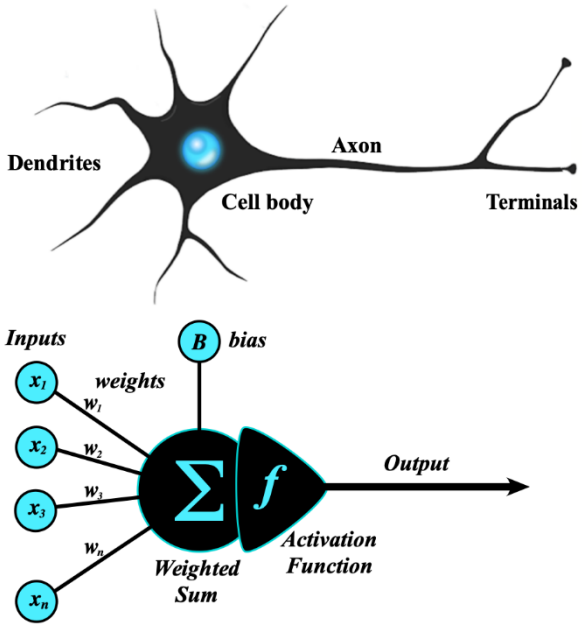
2.2.1 – Artificial Neural Network Model

ANNs are inspired by the biological neural networks of the human brain. These networks are made up of millions of neurons that pass information to each other. ANNs are also made up of networks of neurons, in their case, they are called perceptrons, as seen in Figure 4. In ANNs, a perceptron's responsibility is to take a set of inputs and process them in a way that ultimately becomes something meaningful to the user. A single perceptron alone cannot do much, so ANNs use layers containing hundreds of them. These layers are connected in sequence. For example, the perceptrons from layer one are all connected to the ones in layer two, which are then all connected to layer three, and so on (Aggarwal, 2018). This way the outputs of one layer become the inputs of the next layer. Fully connected layers like this are also called dense layers.

The interconnectivity of dense layers produces relationships between perceptrons resulting in models that frequently outperform many other ML models.

Figure 4

A biological neuron at the top and an artificial perceptron at the bottom (Elster, n.d.)



Each connection between neurons has a weight value associated with it, and each neuron can also have a bias value. When inputs are fed forward through the connections between neurons, they get multiplied by the corresponding weights. Once they reach the neuron, all the resulting

values from the weight input multiplication get summed together, with the neuron bias added (Nielsen, 2019). The result from this process makes the base output of a neuron. An artificial neuron connection can be represented by w_i, each input by $x_i$and the bias by b, so the output of a neuron can be defined as in equation 11.

$$y = w_1x_1 + w_2x_2 + w_3x_3 + \ldots + w_ix_i + b = \sum_{i=0}^{i=N} (w_ix_i) + b \quad (11)$$

The equation (11) can be simplified, by representing the weight input multiplication as the dot product of two vectors, shown in equation (12).

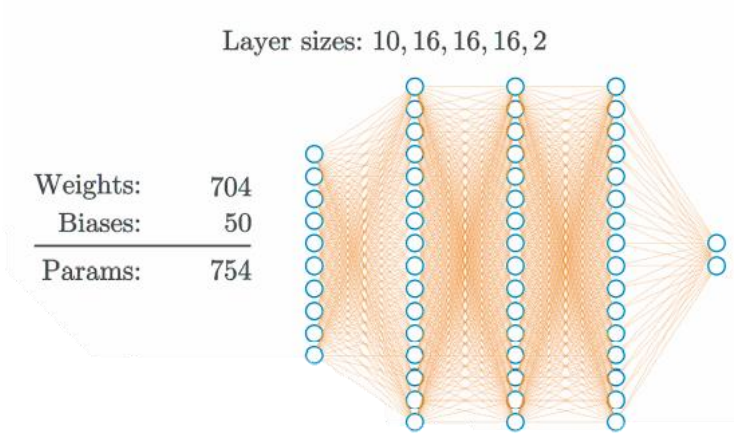
$$y = w \cdot x + b \quad (12)$$

Weights and biases constitute ANN trainable parameters, this means that the network can adjust its weights and biases to perform in an intended way. From equation (12), it can be reasoned that weights affect the slope of the function, while bias affects the offset of the function. Ultimately, the network slowly adjusts its weights and biases based on how wrong they are, hoping to output useful predictions when given data outside of their training (Lanham, 2020).

ANNs normally comprise an input layer, an output layer, and several dense layers in the middle of them. The input layers merely represent the input data. For example, if the input is an image by size of 32*32 then the input layer will be of size 1024, if the input is a 3D vector, then the input layer will be of size 3. The output layer represents the output of the network, and it's defined by the programmer to give results in the intended format. For example, if the network has to classify pictures of dogs by breed, then the number of neurons in the output layer should be the same number of dog breeds. Another example is to predict the value of a function, where the output layer would only need to have a single neuron (Chollet, 2021). An example of an ANN model is shown in Figure 5.

Figure 5

An ANN with an input layer of 10 neurons, 3 hidden layers of 16 neurons each, and 2 neurons for the output layer. Adapted from (Kinsley & Kukieła, 2020)



2.2.2 – Activation Functions

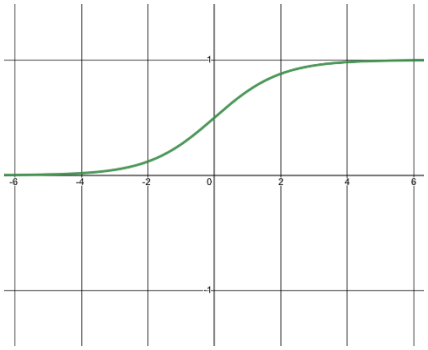
ANNs are meant to be used as function approximators, their objective is to output very similar values to what the real function would give. As mentioned before, the main equation used by neurons to transform data is $y = wx + b$ which is a linear function. This means that without a more complex function, neural networks would only be able to solve linear problems. For example, it would be impossible for an ANN to correctly solve a simple sinewave. Because of this, ANNs use nonlinear functions to transform their layer outputs from linear to nonlinear, these functions are called activation functions (Aggarwal, 2018). There are several different activation functions, and all of them have pros and cons. In general, ANNs will have two types of activation functions, one is used for all the network dense layers and the other is used in the output layer. In this literature review, only the most used activation functions will be explored, as they are also the ones used in the thesis project.

Historically one of the most used activation functions is the sigmoid function (13). It converts values to be in the range between 0 and 1, while keeping some of the granularity of the original values (Kinsley & Kukieła, 2020). The sigmoid makes larger values closer to one, while smaller values closer to 0. A plot of this equation (13) can be seen in Figure 6.

$$y = \frac{1}{1 + e^{-x}} \quad (13)$$

Figure 6

Sigmoid function plot (Kinsley & Kukieła, 2020)

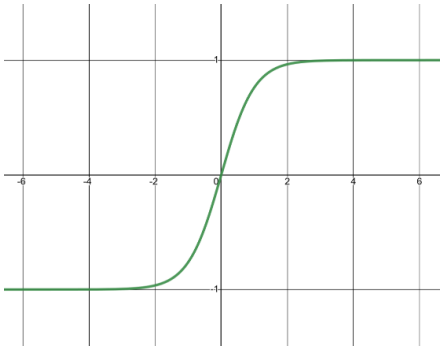


The hyperbolic tangent activation function (14) or Tanh for short, is a very similar function to the sigmoid, the only difference between the two is that the Tanh function outputs values in the range of -1 and 1 (Enyinna Nwankpa et al., 2021). A plot of this equation (14) can be seen in Figure 7.

$$y = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{14}$$

Figure 7

Tanh function plot (*Hyperbolic Tangent - MATLAB Tanh*, n.d.)

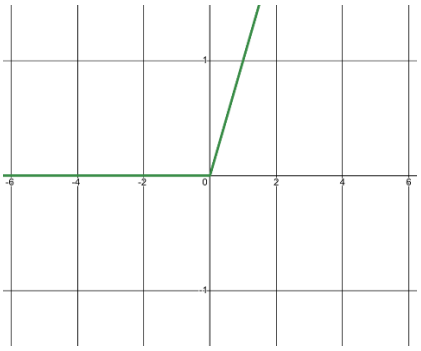


Currently, the most used activation function is the rectified linear activation function (15) also known as ReLU. ReLU is extremely close to being a linear function while being nonlinear, it can be defined as, if x is less or equal to 0 then y is 0, otherwise y is equal to x. Its popularity is due to its speed and efficiency (Enyinna Nwankpa et al., 2021). A plot of this equation (15) can be seen in Figure 8.

$$y = \begin{cases} x, & x > 0 \\ 0, & x \leq 0 \end{cases} \tag{15}$$

Figure 8

ReLU function plot (*A Gentle Introduction to the Rectified Linear Unit (ReLU) - MachineLearningMastery.Com*, n.d.)



While the before-mentioned activation functions can be used in all types of layers, they are mostly used in the dense layers. But some activation functions can only be used in the output layer, an example of that is the softmax activation function. This function is used in most classification problems as it can take non-normalized, uncalibrated inputs and produce a normalized distribution for all the classes the network has to predict (Nielsen, 2019). The distribution that the softmax function (16) produces represents the confidence scores for each class, which once summed add up to one. The predicted class is chosen from the output neuron that has the biggest confidence score, for example, if the output result is [0.2, 0.5, 0.3] then the network is predicting that the right answer is the second class.

$$S_{i,j} = \frac{e^{z_{i,j}}}{\sum_{l=1}^{L} e^{z_{i,l}}} \quad (16)$$

2.2.3 – Loss Functions

For an ANN to train, it must know how wrong or right its outputs are. To do this a neural network needs to calculate how much error its outputs have. The loss/cost function is the algorithm that quantifies how wrong an output is. The loss is the measure of this metric, as such the perfect model would have a loss of 0. Although perfect models are very rare, a neural network always tries to minimize the loss function during training, even if it cannot reach a loss value of 0 (Kinsley & Kukieła, 2020).

There are several types of loss functions, which are chosen to fit according to the type of neural network output. For multiclass classification networks, the categorical cross-entropy loss function is the most used (Goodfellow et al., 2016), and for regression, the most used functions

are the mean squared error (MSE) and the mean absolute error (MAE) (Q. Wang et al., 2022). Classification and regression problems are a staple of ML, because this thesis does not seek to solve these traditional problems it does not explore them in detail, as such for the interested reader the following book references give a comprehensive overview of the basics around these topics (Chollet, 2021; Kinsley & Kukieła, 2020).

The categorical cross-entropy loss function (17) measures the dissimilarity between the true class probabilities and the predicted probabilities of those classes. In this case, the predicted probabilities are usually obtained by applying a softmax activation function to the output layer of the network. The softmax function (17) ensures that the predicted probabilities represent a valid probability distribution, it guarantees that similar true and predicted distribution values close to 1 have a loss value that approaches 0.

$$L = -\sum_{i=1}^{N} y_i \log(\hat{y}_i) \tag{17}$$

The MSE (18) squares the difference between the predicted and the true value of the network outputs, and then averages the sum of those values. This method penalizes more harshly the further away a network output gets from the true value.

$$L = \frac{1}{N}\sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \tag{18}$$

The MAE (19) takes the absolute difference between the predicted and the true value of the network outputs, and then averages the sum of those values. This method gives less importance to outliers when compared to the MSE loss.

$$L = \frac{1}{N}\sum_{i=1}^{N} |y_i - \hat{y}_i| \tag{19}$$

In equations (17) (18) (19), y means the true value, $\hat{y}$ means the network output, and N means the number of outputs.

There are many more loss functions, even for regression problems (Q. Wang et al., 2022), but for this literature review, the focus is only on the ones to be used in the thesis project.

2.2.4 – Forward Pass Conclusion

Given all that has been described in this work so far about neural networks, an example of a full forward pass can be written as equation (20).

$$L = \frac{1}{J}\sum_{j}(y_j - w_2\left(max\left(0, w_1 x_j + b_1\right)\right) + b_2)^2 \quad (20)$$

Equation (20) represents a forward pass of an ANN with one dense layer with the ReLU activation function and an output layer with no activation function. Finally, the ANN loss value is calculated by utilising the MSE loss function.

2.2.5 – Chain Rule and Back-Propagation

As it was mentioned before, for a network to minimize its loss, it must adjust all its weights and biases. There are a few known ways to do this "adjustment", with the most popular method being stochastic gradient descent (SGD). This method involves the result of the calculation of the partial derivatives with respect to all the network weights and biases, using backpropagation with the chain rule to update all the network weights and biases with their gradients. The goal of SGD is to understand how much of an impact a single weight/bias had on the loss and adjust that weight/bias accordingly. For example, if a weight had a big impact on the network output and the loss for that output is high, then that weight is going to be adjusted more severely than a weight that had a negligible impact on the output (Bottou, 2012).

To understand the impact of an input on a function, the partial derivative with respect to that input must be calculated. Because the base function of a neuron in the network is $y = wx + b$, the partial derivative with respect to w and b, must be calculated to find the individual impact of those inputs. This means that the derivatives with respect to each input must be calculated separately (Nielsen, 2019). The partial derivatives of a function with respect to its inputs can be defined as in equation (21).

$$f(x, y, z) \rightarrow \frac{\partial}{\partial x}f(x, y, z), \frac{\partial}{\partial y}f(x, y, z), \frac{\partial}{\partial z}f(x, y, z) \quad (21)$$

The gradient is a vector containing all the partial derivatives solutions with respect to each of the function inputs and is denoted by the symbol ∇. The gradient of the function (21) can be represented in vector/matrix form as (22) (Kinsley & Kukieła, 2020).

$$\nabla f(x,y,z) = \begin{bmatrix} \frac{\partial}{\partial x} f(x,y,z) \\ \frac{\partial}{\partial y} f(x,y,z) \\ \frac{\partial}{\partial z} f(x,y,z) \end{bmatrix} = \begin{bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} \end{bmatrix} \quad (22)$$

As demonstrated before, the forward pass of a network can be described as a chain of functions, where the output of one part of the chain is the input to the next part of the chain. For example, the input layer feeds data into the first dense layer, that then gets passed to an activation function, which then goes on to the next layer, and so on until the data (by that time heavily modified) reaches the output layer that then feeds into the loss function. A simple forward pass can be generalized as in equation (23).

$$y = g(f(x))y = g(f(x)) \quad (23)$$

In this form (23), it is clear that the output of f influences g, as such, it can be reasoned that in some way x influences g. This means that the impact of x on g can be found by calculating the derivative of g with respect to x. The chain rule can be applied to find the derivative of the chain function. The chain rule states that the derivative of a function chain is a product of derivatives of all the functions in that chain (Kinsley & Kukieła, 2020). For example, applying the chain rule to equation (23), to find the derivative of g with respect to x, can be written like in equation (24).

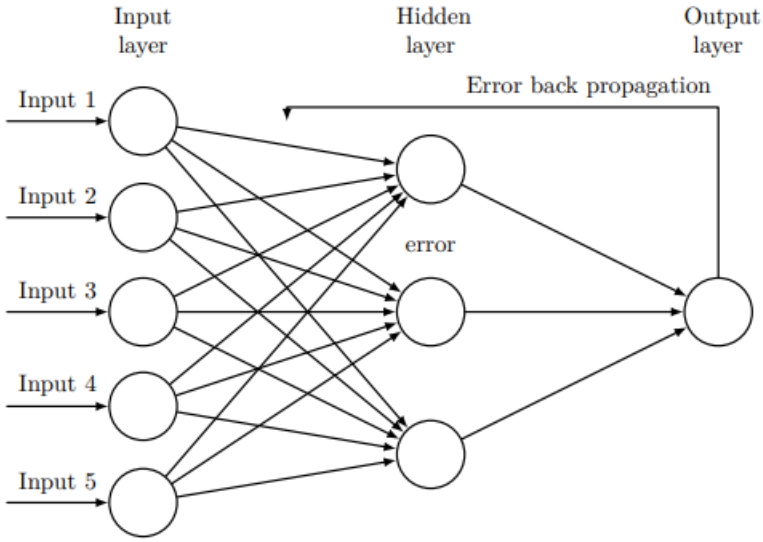
$$\frac{\partial}{\partial x} g\big(f(x)\big) = \frac{\partial g(f(x))}{\partial f(x)} \cdot \frac{\partial f(x)}{\partial x} = g'(f(x)) \cdot f'(x) \quad (24)$$

The combination of these three elements (partial derivatives, gradients, and the chain rule) forms the back-propagation algorithm, which is what allows neural network models to find the impact that individual inputs have on the loss value. In other words, back-propagation is used

to find the gradients of every input of the network, in this case, the gradients for the weights and biases. A visual representation of back-propagation is shown in Figure 9.

Figure 9

Back-propagation in a neural network example (Marta, 2016)



2.2.6 – Optimizers

Finding the gradients is just one step in improving the network. To fully improve the loss, the network weights and biases must be updated. This is where SGD comes in. SGD is considered an optimizer, and the main purpose of an optimizer is to adjust the network weights and biases given their gradient. Several types of optimizers have their own specific methods to update weights and biases, but most optimizers simply build off each other, with their base being SGD (Vani & Rao, 2019).

In its original form, SGD equation (25) just subtracts a fraction of the gradient for each weight and bias parameter.

$$\theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} L(\theta) \tag{25}$$

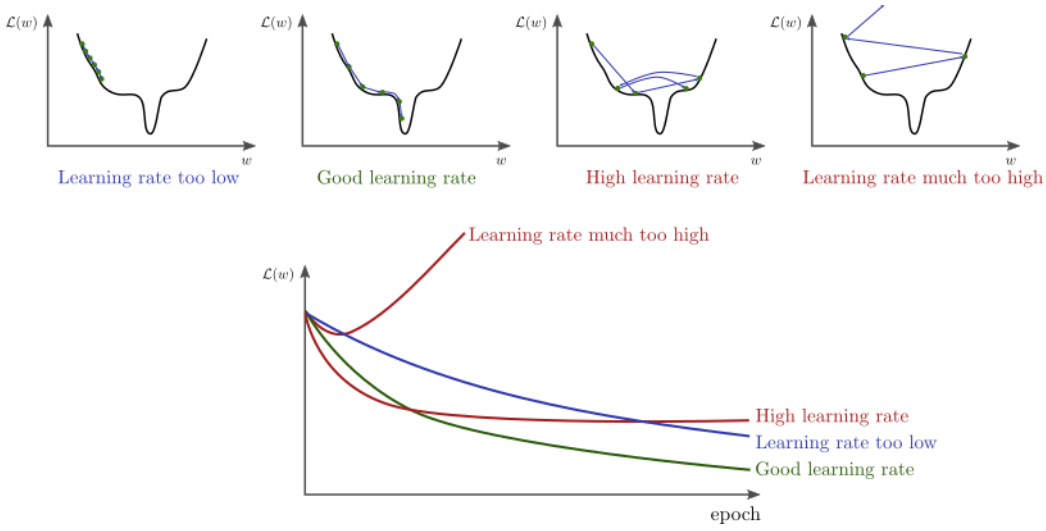
Generally, in SGD the full gradient is not applied, because it can lead to large variations in the weights and biases, which makes it very difficult for the network to converge to a minimum loss. For this reason, only a fraction of the gradient is used, as it allows for a more controlled

descent into the global minimum of the loss function. The variable that is used to decide how much of the gradient is used, is called the learning rate, and is denoted in equation (25) as the symbol α. The value of the learning rate falls in the range between 1 and 0, and depending on the problem, different values are used to achieve better results (Igiri et al., 2015), visual example of this can be seen in Figure 10. The learning rate falls into a category called hyperparameter, this is because its value is chosen by the developer, which is what defines whether a network parameter is a hyperparameter or not.

It is important to note that this method of slightly improving the weights and biases, using an optimizer, only works because it is performed hundreds or thousands of times for the same inputs. An iteration of this process is called an *epoch* and consists of having the network give a prediction on data (can be a single sample or a batch of samples), then calculating the loss based on the prediction, and finally using back-propagation to find the gradients of the network parameters, so an optimizer can be used to update them.

Figure 10

The impact that different Lr values have on a network loss over time (Hammel, n.d.)



While this use of SGD is good enough to find the minimum loss for some problems, it is still very basic. For this reason, SGD is usually paired with two other simple techniques, decay, and momentum, to improve its flexibility and performance.

The learning rate decay technique is very simple but very effective. The purpose of the decay is to decrease the learning rate as training goes on, although it decreases it less and less, as

epochs get completed during training, as demonstrated in equation (26). The objective of decaying the learning rate is to make smaller adjustments, to the weights and biases, so it prevents overshooting the loss function global minimum as the loss value gets closer to it (Nakamura et al., 2021). The decay rate variable is another hyperparameter.

$$learning\ rate = \frac{starting\ learning\ rate}{1 + decay\ rate * epoch\ number} \quad (26)$$

The usage of momentum in SGD was inspired by the real momentum that is experienced in real life. SGD with momentum tries to simulate what would happen to a ball rolling down a hill, if the ball finds a small hole or hill, the momentum will make it go past it and straight toward the bottom of the hill. In SGD, momentum works much the same, it tries to prevent cases where the model might get stuck towards a local minimum while going in the direction of the global minima (Choi et al., 2020).

The momentum uses the previous parameter update direction to influence the next update direction, by having it subtracted with the current gradient value, as seen in equation (27). As with the value of the gradient, only a fraction of the previous update is used. This is controlled by the hyperparameter momentum, which ranges in values between 0 and 1.

With the use of momentum, the weight and bias update formula from SGD must be slightly altered, as shown in equation 28.

$$v_t = \beta v_{t-1} - \alpha \nabla_{\theta_j} L(\theta_j) \quad (27)$$

$$\theta_j = \theta_j + v_t \quad (28)$$

Following these optimizations to the original SGD, several other optimizers started to appear. Currently, there exist several different optimizers that try to outperform the original SGD. Newer optimizers tend to build on the concepts of older ones. For example ADAM, currently the most popular optimizer, builds on top of another famous optimizer named RMSProp (Kinsley & Kukieła, 2020).

The RMSProp optimizer, short for Root Mean Squared Propagation, was first introduced by the father of backpropagation Geoffrey Hinton (Hinton et al., 2020). Similarly to other optimizers like AdaGrad, RMSProp uses a technique called the adaptive learning rate (29). This serves as a normalization update technique of weights and biases, which prevents the network values

from having too large of a discrepancy between each other. For example, some weights tend to get bigger updates than others, and the adaptive learning rate prevents those weights from getting into a range that is almost unobtainable by other weights, allowing these smaller ones to catch up. In equation (29), the symbol v is the normalization parameter, and the symbol ϵ is a hyperparameter that is merely there to prevent a division by 0, its value is usually something very small like 1e-7.

$$\theta_j = \theta_j - \alpha \frac{\nabla_{\theta_j} L(\theta_j)}{\sqrt{v_t} + \epsilon} \quad (29)$$

The unique part of RMSProp is how it updates the normalization parameter. The update consists of using a very similar function to SGD with momentum but uses the squared gradients instead, as shown in equation (30).

$$v_t = \beta v_{t-1} + (1 - \beta) \nabla_{\theta_j} L(\theta_j)^2 \quad (30)$$

As mentioned before, the most popular optimizer is Adaptive Momentum (ADAM) (Kingma & Lei Ba, 2014), which is also the optimizer used for this thesis project. ADAM uses the normalization parameter from RMSProp and the momentum concept from SGD, it uses momentum like in SGD but then uses the adaptive learning rate as it is used in RMSProp. Equation (31) represents the momentum update from ADAM. The normalization parameter update function is the same as the one used for RMSProp. The symbol β_1 is the hyperparameter used to control the impact of momentum, and β_2 becomes the hyperparameter to control the normalization parameter.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla_{\theta_j} L(\theta_j) \quad (31)$$

ADAM also implements a technique called bias mechanism correction, (not to be confused with the bias parameter). This technique is applied to both the momentum and the normalization parameter, and its purpose is to compensate for their initial values that tend to be close to 0, trying to "warm" them up in the initial stages of the training process (Goodfellow et al., 2016). The symbol $\widehat{m_t}$, in equation (32), corresponds to the bias corrected momentum, and the $\widehat{v_t}$ symbol, in equation (33), corresponds to the bias corrected normalization parameter. These bias corrected variables are then used in the final update equation (34).

$$\widehat{m_t} = \frac{m_t}{1 - \beta_1^t} \quad (32)$$

$$\widehat{v_t} = \frac{v_t}{1 - \beta_2^t} \quad (33)$$

$$\theta_j = \theta_{j-1} - \alpha \frac{\widehat{m_t}}{\sqrt{\widehat{v_t}} + \epsilon} \quad (34)$$

2.3 – Deep Reinforcement Learning

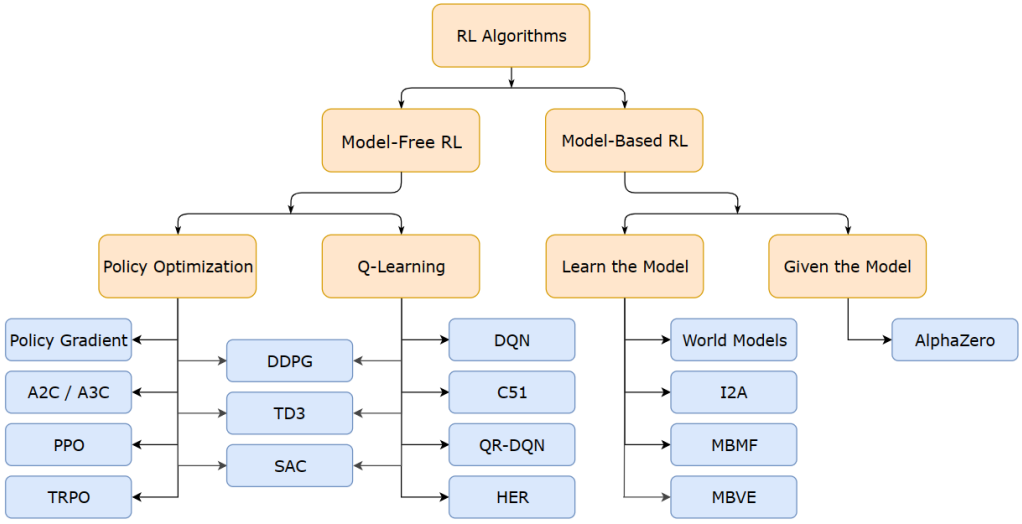
As shown in section 2.1, both SARSA and Q-Learning use tabular methods, meaning they need to store each action-state pair value in a table so they can be updated, which is a very common approach for many other RL algorithms. When using RL algorithms, the memory size required to use their tabular methods depends on the environment's number of possible states. In real life most problems have a near-infinite state space, which means that given the limitations of the currently available hardware, RL by itself cannot be used. Besides this, there are other relevant problems to this approach, tabular updating methods require that all states be visited several times for their true value to converge. This can mean that for large state spaces, some of those states can be rarely visited, which can lead to large over and under estimations of the true value. If RL algorithms could be able to generalize state values, then both aforementioned problems would be solved, this is where ANNs come in. ANNs like many other ML models are function approximators, their purpose is to give approximate results to a target function. In RL, function approximation methods can represent the value functions, removing the need for tabular update methods. Despite the ease of use of ANNs in many supervised learning problems, using them to solve RL problems can be more challenging. Despite this, researchers have developed several RL algorithms that use neural networks, giving rise to the field of DRL (Lonza, 2019).

DRL started to become popular around 2013 when researchers from DeepMind presented Deep Q-Network (DQN) the first DRL model to display superhuman ability in playing Atari 2600 games, using only raw pixel images and with no adjustment of the architecture or learning algorithm (Mnih et al., 2015). Another high-profile achievement of DRL came in 2017 when DeepMind used AlphaGo to learn the game of Go and defeat pro player and 18-time world champion Lee Sedol (Silver et al., 2016). DRL has recently grown so much in popularity because of these breakthroughs, which have led to a huge increase in DRL research and new algorithms in recent years, as shown in Figure 11.

This section's purpose is to explain in detail the DRL algorithms and techniques used in the thesis project.

Figure 11

Non-exhaustive taxonomy of DRL algorithms (Part 2: Kinds of RL Algorithms — Spinning Up Documentation, n.d.).



2.3.1 – Deep Q-Network

DQN is probably the most popular DRL method, as the name implies it takes the Q-Learning method and applies a neural network to it, replacing the tabular updates. In this scenario, the Q-value function is now the output of the neural network becoming $Q_\theta(s, a)$, where the symbol θ represents the weights and biases of the network.

As mentioned in the DL section, the goal of the network is to minimize the loss by adjusting its weights and biases. In this case, the previous Q-Learning method for updating the action-state values is adapted to be used as the network update function, as shown in equation (35). The objective is to optimize the network weights and biases, so its output approximates the optimal Q-value function (Palanisamy, 2018).

$$\theta = \theta - \alpha(r + \gamma \max_{a'} Q_\theta(s', a') - Q_\theta(s, a))\nabla Q_\theta(s, a) \quad (35)$$

In equation (35) the $r + \gamma \underset{a'}{max} Q_\theta(s', a')$ argument can be considered the target value y, as it is an estimation of the Q-value function true value, represented in equation (36). By doing so, equation (35) can be simplified, as shown in equation (37):

$$y = r + \gamma \underset{a'}{max} Q_\theta(s',\ a') \quad (36)$$

$$\theta = \theta - \alpha(y - Q_\theta(s,a))\nabla Q_\theta(s,a) \quad (37)$$

Describing the network update function like this, shows that this equation is the same as the derivative of the MSE loss function. This demonstrates that the MSE can be used to calculate the network loss in DQNs, as seen in equation (38) (Lonza, 2019), although in DRL this metric is not as useful as in supervised learning.

$$L(\theta) = (y - Q_\theta(s,a))^2 \quad (38)$$

Simply applying these techniques will allow neural networks to be used in Q-Learning, but this alone does not make the DQN algorithm. Two main factors from the vanilla Q-Learning implementation hurt the learning capabilities of neural networks: the lack of Independent and Identically Distributed (IID) data and the moving target problem. Neural networks tend to be more stable when the training data is IID. It is very common to see this happening in supervised learning, where datasets are always randomly shuffled. The opposite happens in RL, where the data that is fed to the network tends to be sequential and strongly related, which causes instability. The above equations show that the y value is calculated by the same network that is being updated, which causes the moving target problem, where the target values keep getting updated during training, which is another cause for instability (Lapan, 2020).

To solve these stability problems and increase reliability, DQN implements two techniques that are crucial to making neural networks work with RL. The first involves using a replay buffer/memory to deal with the problem of feeding the network sequential and strongly related data. The second addresses the moving target problem, by using a separate network to calculate the target values (Lanham, 2020).

The purpose of the replay memory (also known as experience replay) is to provide the used network IID data. The replay memory collects all the agent needed transition data (states, actions, rewards), so it can be used later during training (Mnih et al., 2015). When the network is performing SGD, it samples a random batch of the agent transition data from the replay

memory. In reality, storing all agents' transitions is impractical, as there can be an almost infinite amount of them, so the replay memory implements a system of FIFO (First In, First Out). The replay buffer stores all transitions until a certain threshold is reached, after which it starts to release older transitions in place of new ones.

To solve the moving target problem, DQN implements two networks instead of one. The first network, called the online network is used to interact with the environment and is also the one being constantly updated. The second network, called the target network, is used to predict the target values y, this network is only updated every N steps, with N usually ranging in values between 1000 and 10000 (Mnih et al., 2015). The intuition is that the target value remains fixed for every N iterations, diminishing training instabilities. In reality, the target network is an older version of the online network, when the target network updates it simply copies the current values of the online network.

To describe the use of the target network in calculating the target values, some alterations must be made to the update equation (35). In equation (39) $\hat{Q}$ symbolizes the target network, and θ' represents its weights and biases. The pseudocode for the DQN algorithm can be seen in Table 2.

$$\theta = \theta - \alpha(r + \gamma \max_{a'} \hat{Q}_{\theta'}(s',\, a') - Q_\theta(s,a))\nabla Q_\theta(s,a) \qquad (39)$$

Table 2

DQN algorithm pseudocode (Mnih et al., 2015)

Deep Q-learning algorithm with experience replay
procedure DQN
Initialize replay memory D with capacity N
Initialize action-value Q with random weights θ
Initialize action-value $\hat{Q}$ with weights $\theta' = \theta$
for $episode = 1..M$ **do**
$s \leftarrow InitialSate$
for each episode step do
Select a, **based on an exploration strategy, given** s
Take action a**, observe** r, s'
Store transition (s, a, s', r) **in** D
$s \leftarrow s'$
Sample random minibatch of transitions (s_j, a_j, s'_j, r_j) **from** D
Set $y = \begin{cases} r_j & if\ s'_j == terminal \\ r_j + \gamma \max_{a'} \hat{Q}_{\theta'}(s'_j,\ a') & otherwise \end{cases}$
$\theta \leftarrow \theta - \alpha(y - Q_\theta(s_j, a_j))\nabla Q_\theta(s_j, a_j)$
Every C **steps** $\hat{Q}_{\theta'} = Q_\theta$
end for
end for

2.3.2 – DQN variations & extensions

After its initial great results, DQN became the most popular DRL algorithm. Given this adoption, its shortcomings also started to become more evident, as such, many researchers started to develop new techniques that build upon and improve on the original DQN algorithm. While there are a few DQN variations (Hessel et al., 2018), this section focuses on the ones applied in this thesis project.

2.3.3 – Double DQN

One problem that Q-learning algorithms suffer from, is overestimating the Q-values. In the case of DQN, the overestimation comes from using the target network to calculate the y value. In the original Double DQN paper (Van Hasselt et al., 2016), the authors state that the overestimation comes from the max operation in the Bellman equation, as it uses the same values to both select and evaluate an action. The authors suggest decoupling the selection from

the evaluation to prevent this overestimation. This can be achieved by using the online network to select the action, but having the target network evaluate it. Now the target Q-value expression (36) can be written as equation (40).

$$y = r + \gamma \hat{Q}_{\theta'}(s', \underset{a'}{argmax}\ Q_{\theta}(s', a')) \quad (40)$$

In the original paper (Van Hasselt et al., 2016), the authors prove that this simple change solves the overestimation issue, calling this architecture double DQN.

2.3.4 – Dueling DQN

The Dueling DQN architecture comes from the idea that the Q-values that a DQN model tries to approximate, can be divided into two properties, the state value $V(s)$, and the state-action advantage value $A(s, a)$. By doing this split, the model will improve training stability, and faster convergence (Z. Wang et al., 2016). As mentioned before, the state value represents a state's expected reward, and the advantage value says how much of an improvement action a value is over the expected state reward. If $A(s, a)$ is a positive value, it means, the reward value that action a brings at state s is bigger than the average reward at that state. On the other hand, if $A(s, a)$ is negative, then it means the opposite, as the action is worse than the average. The Q-value function can be described as equation (41).

$$Q(s, a) = V(s) + A(s, a) \quad (41)$$

Unfortunately, using equation (41) during training can lead to a completely wrong interpretation of the relation between the two formulas. To prevent this, the original authors (Z. Wang et al., 2016) suggest that the mean value of the advantage should always be zero. They proposed subtracting the mean value of the advantage from the Q-value expression, thus the new $Q(s, a)$ function becomes the equation (42).

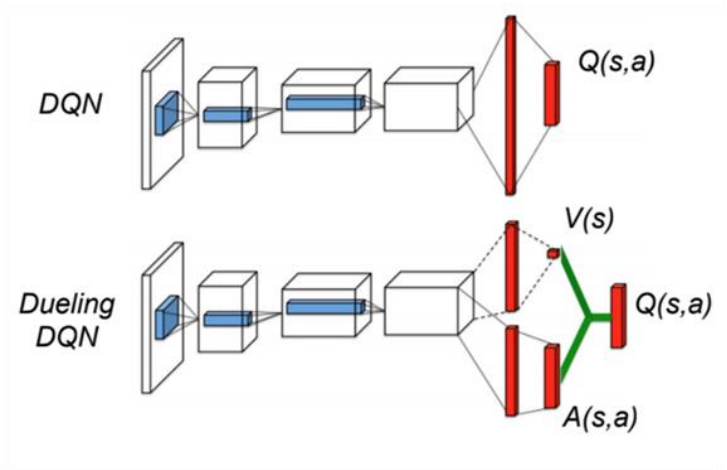
$$Q(s, a) = V(s) + A(s, a) - \frac{1}{|A|}\sum_{a'} A(s, a') \quad (42)$$

For a DL model to be able to predict two different functions (the value and advantage) the authors suggest an architecture that contains two streams, one that predicts the value function

and another one that predicts the advantage as shown in Figure 12. Both streams share the same input and output layer, of which the latter is responsible for implementing the equation (42) using the output of both streams.

Figure 12

DQN and Duelling DQN architecture (Z. Wang et al., 2016), the top model shows a typical DQN architecture that contains a single stream. The bottom model shows the suggested architecture for the Duelling DQN algorithm. It contains two streams (the value and advantage) and has a shared input layer (which in this case is a convolution neural network), and a shared output layer.



2.3.5 – N-step DQN

The concept of N-step was first introduced by Sutton (1988), it is a combination of TD methods that learn by doing a single step at a time, and methods that use the complete trajectory to learn, like Monte Carlo methods. By definition, the Bellman update used to calculate the target values in DQN (36) can be described as a recursive equation (43).

$$Q(s_t, a_t) = r_t + \gamma \max_a \left[r_{a,t+1} + \gamma \max_{a'} Q(s_{t+2}, a') \right]$$

$$Q(s_t, a_t) = r_t + \gamma r_{t+1} + \gamma^2 \max_{a'} Q(s_{t+2}, a') \quad (43)$$

The equation (43) shows how N-step DQN works, in this particular function the n-step is equal to one, and so the function only unrolls once, but this could be done for as many n-steps as desired, naturally, the unrolling would always stop once a terminal state is reached. The intuition is that all Q values will converge faster the more n-steps there are, but in reality, this does not happen. The equation (43) assumes that all rewards r were achieved on an optimal policy, which does not happen in DQN. DQN selects an action based on an exploration strategy, usually epsilon-greedy with decay. This means that, especially at the beginning of training, the DQN algorithm may choose random actions that might not be the most optimal ones, which can lead to rewards that are below their max value. This problem is amplified by the replay buffer, as it can continue to sample from transitions obtained from a bad policy. This means that the more unrolled the Bellman equation is, the higher the probability of the Q-values being incorrect also rises, which then leads to wrong Q function updates. This does not mean that the n-step technique is useless, Hessel et al. (2017) state that n-step can still help DQN converge faster as long as, n (the number of unrolls) is small, between 2 and 4, and the exploration factor ϵ is also small, ranging between 0.01 and 0.001. Equation (44) shows how the $Q(s, a)$ function can be generalized for any size of n, which in N-step DQN is used to calculate the target value.

$$Q(s_t, a_t) = \gamma^n \underset{a'}{max} Q(s_{t+n}, a') + \sum_{k=0}^{n-1} \gamma^k r_{t+k+1} \quad (44)$$

2.3.6 – DQN with Prioritized Experience Replay

DQN with Prioritized Experience Replay (PER) is an extension of the DQN algorithm that improves the learning efficiency of the network by prioritizing important experiences during the training process. The idea behind PER is to assign a priority value to each experience in the experience replay buffer based on its importance in the learning process (Schaul et al., 2015). This priority value is then used to determine which experiences should be sampled more frequently during training.

The PER algorithm assigns a priority value to each experience in the replay buffer, based on its importance in the learning process. The probability of an experience being chosen as a training sample is calculated using equation (45) where p is the priority of the sample and α is a hyperparameter. The priority can consider using the MSE error of the experience (in the original

paper the mean absolute error is used), which is a measure of how well the network's Q-values predicted the observed reward. Experiences with high errors are assigned higher priority values since they are more likely to improve the network's performance.

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \quad (45)$$

During the training process, experiences with higher priority values are sampled more frequently than those with lower priority values, which allows the network to focus on the most important experiences. This approach adds bias to the training prosses since data no longer has its IID properties due to not being uniformly sampled. To correct this bias the authors (Schaul et al., 2015) suggest multiplying importance-sampling weights with the MSE loss error of the experience. The equation (46) describes how the weights can be calculated. During training, the hyperparameter β should start with a value between 0 and 1, and slowly increase to 1, which will fully compensate for the bias introduced.

$$w_i = (N * P(i))^{-\beta} \quad (46)$$

2.3.7 – Noisy Networks DQN

The Noisy DQN algorithm is another extension of the DQN model that introduces a new form of exploration that replaces the before-used epsilon greedy strategy (Fortunato et al., 2017). The intuition behind Noisy DQN is that adding random noise to the network's weights and biases allows the model to explore more widely during the learning process.

As mentioned previously, the standard DQN algorithm relies on the epsilon greedy strategy for exploration, and while this method works particularly well for simple environments with short episodes, it struggles in environments that have sparser rewards where a larger sequence of actions is needed to achieve a reward. To address this issue, the Noisy DQN algorithm introduces noise directly into the network's weights, which adds a random component to the network's output.

To introduce noise, the Noisy DQN algorithm adds new parameters to the equation of a neural network linear layer (12). These new parameters are the sigma (σ) weights and biases which are also trainable parameters. Each of these new parameters gets multiplied by random numbers taken from a Gaussian distribution, as seen in equation (47).

$$y = (wx + b) + (\sigma^b * \varepsilon^b + (\sigma^w * \varepsilon^w)x)$$
$$y = (w + \sigma^w * \varepsilon^w)x + b + \sigma^b * \varepsilon^b \quad (47)$$

But before the random numbers can be used in equation (47) they must be factorized. To do this, the original paper written by (Fortunato et al., (2019) suggests using the formula (48).

$$\varepsilon = f(x)$$
$$f(x) = sign(x) * \sqrt{|x|} \quad (48)$$

During the training process, the network's weights are updated as usual using the backpropagation algorithm, however, the new sigma parameters are also updated using that same method. This allows the network to learn how to use the noise effectively to explore the state-action space, while also learning to process the Q-values.

2.3.8 – Categorical DQN

The Categorical DQN algorithm, also known as C-51, belongs to a group of algorithms that extend the DQN algorithm to predict the distribution of the return values (a distribution for every action) rather than the expected return values. The idea behind estimating the entire distribution is to take uncertainty into account, leading to more informed decision-making (Bellemare et al., 2017).

In the traditional DQN approach, the goal is to predict the expected return for each state-action pair. The C-51 algorithm tackles this limitation by representing the value function as a discrete distribution over possible return values.

To estimate this distribution, the C-51 algorithm defines a set of "atoms" that fall within a range of minimum and maximum values, that respectfully tend to represent the minimum and maximum possible rewards. This set of values is calculated by evenly spacing values within this range for a predefined length, forming the support vector. The predicted model distributions are represented by the probabilities assigned to these "atoms", allowing the algorithm to capture the full distribution of possible return values.

To be able to update the distributional estimates, the C-51 algorithm updates the Bellman equation used by the DQN algorithm (36) into the distributional Bellman equation (49). Both

equations are very similar, but the new equation uses a distribution for the prediction instead of a scalar.

$$Z(s,a) = R(s,a) + \gamma Z(s',\ a') \tag{49}$$

The resulting distribution from applying this new update (49), is used to train the model so it produces more accurate distributions for every state-action pair. To train the model more accurately, the C-51 algorithm uses the categorical cross-entropy loss, which compares the predicted distribution for action a to the new Bellman updated distribution.

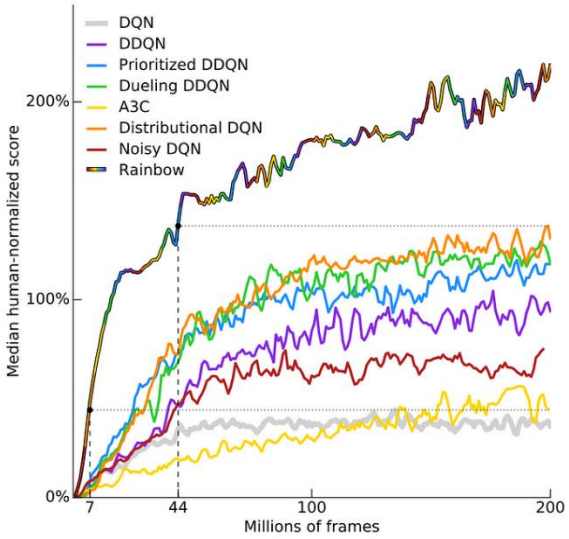
2.3.9 – Rainbow DQN

Rainbow DQN is one of the most advanced variants of the DQN algorithm, it combines all the previously mentioned DQN extensions to improve the stability and performance of the algorithm during training. Rainbow DQN was introduced by researchers at DeepMind in 2017 (Hessel et al., 2018), where it managed to produce state-of-the-art results on the Atari 2600 benchmark, both in terms of data efficiency and final performance, as shown in Figure 13.

More specifically, Rainbow DQN uses Double Q-Learning to tackle overestimation bias, PER to prioritize important transitions, Duelling networks to separate the estimation of state values and action advantages, multi-step learning to update the Q-values with multiple future rewards, distributional reinforcement to represent the distribution of returns instead of the expected return, and noisy linear layers for exploration.

Figure 13

Rainbow DQN performance (Hessel et al., 2017). Overall performance of the Rainbow DQN algorithm against the previous best-performing algorithms at the time on the Atari 2600 benchmark. The graph clearly shows that Rainbow DQN is the best algorithm for this benchmark.



2.4 – Neuroevolution

Neuroevolution has a rich history of use in the video game industry (Risi & Togelius, 2017; Togelius et al., 2024). As a subfield of Evolutionary Algorithms and DL, it aims to optimize ANNs by employing certain algorithms from Evolutionary Algorithms. Its primary objective is to solve for any given function $f(x)$, without any preconceptions. Unlike DRL, Neuroevolution is only concerned with the output of f, not its underlying mechanics. This implies that Neuroevolution has the potential to optimize for any function, which stands in sharp contrast to DRL (Lonza, 2019).

Neuroevolution diverges from DRL in its approach to training ANNs, drawing inspiration from biological evolution rather than relying on back-propagation. Neuroevolution adopts a training paradigm modelled after natural selection, which gives rise to a structured training cycle encompassing the following steps (Brown & Zai, 2020):

1. Population Initialization: The process creates a population of individuals for testing. Each individual is characterized by both a genotype and a phenotype. In the context of Neuroevolution, the genotype encapsulates all the weights, biases, and any other

relevant information related to an ANN, such as a genome ID. While, the phenotype represents the actual ANN structure, ready for use. The initial population is typically initialized with random weight and bias values.

2. Fitness Evaluation: With a population created, the algorithm proceeds to evaluate the performance of each individual on the specific problem it aims to solve. This can be finding the exit of a maze. After completing a simulation/episode, each individual receives a score based on their performance. The function responsible for assigning these scores is known as the fitness function, and the primary objective of Neuroevolution is to optimize for this function.

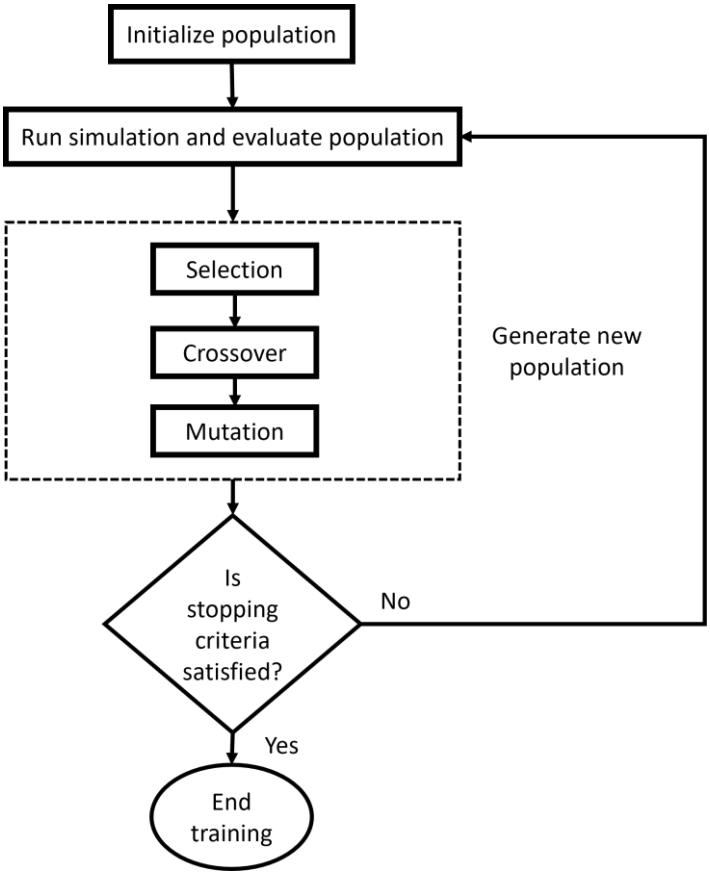
3. Selection: The next step, selection, focuses on breeding a new and improved population based on the fitness scores obtained by the individuals in the current generation. The process of selecting individuals may vary among different algorithms, but at its core, Neuroevolution tries to use the best genotypes in some way to create the next generation of individuals. This usually involves two different methods: crossover and mutation. Crossover combines the genes of two selected individuals to generate a new individual, while mutation introduces random alterations to the genotype of a chosen individual. The formation of a new population represents a new generation.

4. Iterative Process: The steps from 1 to 3 are repeated until some condition is met, such as when an individual achieves a certain fitness score threshold or a specified generation limit is reached.

While this structure forms the baseline of Neuroevolution, it's worth noting that each algorithm implements these steps in different ways (Telikani et al., 2022). A visual representation of this structure can be seen in Figure 14.

Figure 14

Neuroevolution main cycle, adapted from (Brown & Zai, 2020)



In light of the growing popularity of DRL, there has been a decline in Neuroevolution research as DRL seemed to offer policies with superior outcomes that were also easier to comprehend. However, in recent years, two noteworthy papers—one from OpenAI (Salimans et al., 2017) and another from Uber AI Labs (Such et al., 2018)— have demonstrated that two distinct Neuroevolution algorithms can yield better results than some of the most commonly used DRL algorithms on renowned DRL benchmarks.

These studies have also emphasized the advantages of utilizing Neuroevolution over DRL, particularly in the area of parallelization, where Neuroevolution algorithms can run on multiple machines simultaneously while achieving high performance (Lapan, 2020).

2.4.1 – Random Search

Random Search (RS) algorithms can be applied in numerous ways and, when implemented with the principles of Neuroevolution these methods resemble hill-climbing algorithms. They optimize for the best solution by adding noise to the current best solution, assessing whether these variations yield superior results. This process repeats until a certain condition is met (Lapan, 2020).

Neuroevolution RS is a straightforward algorithm. It begins by creating a single ANN, followed by the creation of a population where each individual represents a duplicate of this network with small perturbations made to its weights and biases. After evaluation, the network with the best fitness score is used to produce the next population of perturbed/noisy networks. This step could be considered the Mutation phase from Figure 14.

Despite its simplicity, this algorithm occasionally achieves better results than some of the most popular DRL and Neuroevolution algorithms. As a result, it is a commonly used algorithm in benchmark testing as it can provide useful insights into the performance of other algorithms (Hausknecht et al., 2014; Mnih et al., 2015; Such et al., 2018).

2.4.2 – Covariance Matrix Adaptation Evolution Strategy

The Covariance Matrix Adaptation Evolution Strategy (CMA-ES) is a popular Neuroevolution algorithm used for video game playing (Risi & Togelius, 2017; Togelius et al., 2024). In recent years, it has gained more traction after Salimans et al. (2017) demonstrated that CMA-ES can outperform some of the most popular DRL algorithms while requiring less training time.

The CMA-ES algorithm is similar to the RS algorithm described in the previous section, but there are a few key differences. Specifically, CMA-ES samples its perturbation noise from a Gaussian distribution, with each weight and bias being summed with the corresponding noise (ε), scaled by the noise standard deviation (σ). This transforms the linear network equation into a similar one used for Noisy DQN (47).

Following the evaluation phase, the CMA-ES algorithm updates the original weights and biases values by factoring in the fitness score of each individual (i), multiplied by the corresponding

noise (50). The intuition behind this is to steer the network's weights and biases towards the direction of the best-performing individuals.

$$\theta = \theta + \alpha \frac{1}{n\sigma} \sum_{\mathrm{i}=1}^{n} F(\theta + \sigma \varepsilon_{\mathrm{i}})\, \varepsilon_{\mathrm{i}} \tag{50}$$

It's worth noting that this approach shares similarities with SGD, which is not a coincidence. According to Uber Labs AI (Such et al., 2018), CMA-ES leverages this step to estimate gradients, but instead of descending, it performs gradient ascent. As a result, alternative optimizers, such as ADAM, may also be utilized to update the network.

The OpenAI research paper (Salimans et al., 2017) also highlights two methods that aid in improving CMA-ES's convergence stability and overall performance:

- Mirrored Sampling: This technique involves generating noise for an individual and then creating a copy of that individual's noise variables, but with negated noise values, which are used to create a new individual. This approach results in half of the population being composed of the negated version of the other half.
- Rank Transformation: In the standard CMA-ES methodology, fitness scores are normalized before doing the update step. Instead, rank transformation calculates the rank for each fitness score within the population, and then normalizes the ranked values.

2.4.3 – Genetic Algorithms

Genetic algorithms (GAs) are among the more versatile Neuroevolution algorithms due to the numerous variants available (Elsayed et al., 2010; Jenkins et al., 2019; Katoch et al., 2021). While GAs typically adhere closely to the structure depicted in Figure 14, the nuances that distinguish one GA model from another lie in the implementation of their selection, crossover, and mutation steps. How a GA implements these steps can heavily dictate its performance. In the context of this thesis, the following methods were used for the selection, crossover, and mutation steps: tournament selection, single-point crossover, and noisy injection:

- Tournament Selection: In this method, a group of individuals is randomly chosen from the population and their fitness values are compared. From this group, the two best

individuals are selected to undergo the crossover function. This process is repeated for every new member that needs to be created. The size of the group sampled from the population is referred to as the tournament size. A larger tournament size increases the likelihood of passing down the best-performer genes to the next generation, but this comes at the expense of gene diversity. Conversely, a smaller tournament size results in the opposite effect (Wirsansky, 2020).

- Single-Point Crossover: This technique selects a random point within the genome of both parents, referred to as the crossover point. The offspring of this crossover inherit genes situated to the right of the crossover point from one parent and genes on the left from the other parent. This mechanism ensures that the offspring carries genetic information from both parents (Wirsansky, 2020). Since an ANN genome can be represented as an array of weights and biases, single-point crossover is a well-suited method.

- Noisy Injection: The Noisy Injected involves randomly selecting a value from a Gaussian distribution, scaling it by a specific standard deviation, and then adding the result to one of the network's weights or biases (Buckland, 2002). This approach mirrors the technique employed by RS to generate a new population. However, in this approach, mutation is applied to an offspring resulting from the crossover method. The number of parameters altered by this metric is determined by the mutation rate value. This value determines the percentage of ANN parameters subject to mutation, typically maintained at a low level.

In addition to the selection, crossover, and mutation methods, GA algorithms offer the flexibility to incorporate additional techniques that extend beyond the core structure (Wirsansky, 2020). Within this project, two such techniques have been implemented: elitism and adaptive mutation:

- Elitism: The objective of elitism is to ensure that the best-performing individuals advance to the next generation. The number of top individuals carried over to the subsequent generation is controlled by the elitism number. This approach proves invaluable in safeguarding GA performance, particularly when the choice of selection, crossover, and mutation methods may otherwise result in performance fluctuations (Wirsansky, 2020).

- Adaptive Mutation: Adaptive mutation addresses the issue of having a fixed mutation rate. With constant mutation rates, both good and bad individuals undergo the same level of changes in their genes. This can prove problematic when a large mutation rate disrupts well-performing solutions, or a low mutation rate hinders the exploration of novel solutions and the discovery of better-performing genes. Adaptive mutation proposes to classify individuals as high or low quality by assessing their fitness score in comparison to the average score of the population. High-quality individuals will receive a lower mutation rate, while low-quality ones will experience a higher mutation rate (Marsili-Libelli & Alba, 2000).

2.4.4 – Neuroevolution Augmenting Topologies

Neuroevolution Augmenting Topologies (NEAT) and its variations are considered some of the most effective algorithms in Neuroevolution (Risi & Togelius, 2017). While NEAT follows the core principles of GAs, it introduces a pivotal distinction setting it apart from the algorithms discussed so far. In addition to adjusting connection weights, NEAT also evolves the topology of ANNs in order to find the optimal solution (Omelianenko, 2019). The main idea behind evolving topologies is that a population can start with the simplest possible networks, and from there evolve their topology in search of better solutions. As such, there is no need to predefine an ANN's topology, as NEAT ensures the discovery of the smallest topology capable of solving the target task at hand.

For NEAT to successfully evolve ANN topologies, it requires a crossover method capable of generating an ANN from two parents possessing distinct topologies. Additionally, mutation methods must not only alter the connection weights of an ANN but also modify its topology. Finally, a selection method must be employed to promote the diversity of ANN topologies within the population.

To facilitate the evolution of ANNs, the original NEAT authors (Stanley & Miikkulainen, 2002) developed a specific ANN genotype, as depicted in Figure 15. This NEAT genotype is comprised of two distinct types of genes: neuron genes and connection genes.

Neuron genes contain two primary parameters:

- Neuron ID: Helps distinguish one neuron from another.

- Neuron type: Describes a neuron's role within the ANN structure, categorizing it as input, output, or hidden.

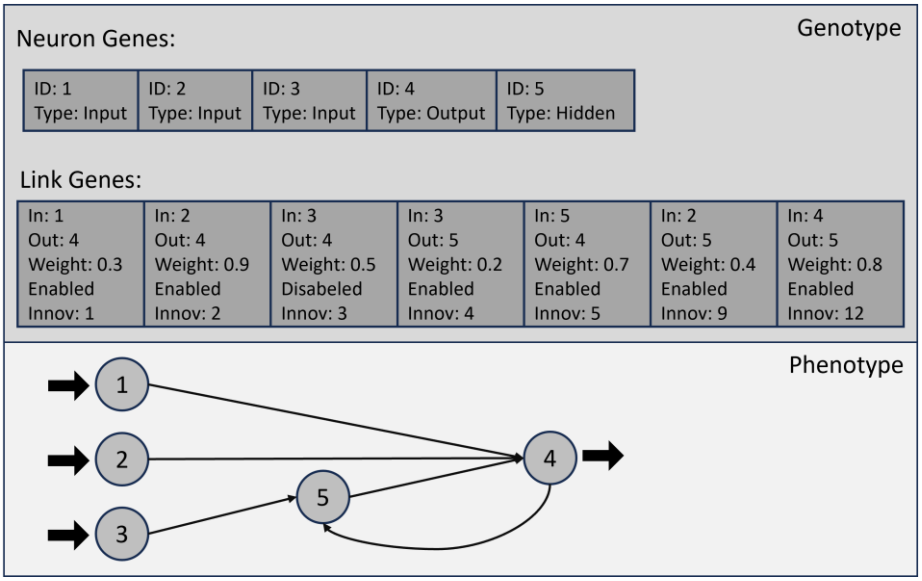
Connection genes, on the other hand, are primarily composed of five parameters:

- Input Neuron ID: The neuron from which the connection originates.
- Output Neuron ID: The destination neuron for the connection.
- Connection Weight: The value representing the connection's strength.
- Enabled Tag: A binary indicator that determines whether the connection or link is active within the ANN's phenotype. When set to 'false,' the link remains unused.
- Innovation Number: This number plays a pivotal role in enabling the crossover operator to generate offspring when combining ANNs with different topologies.

The genotype parameters mentioned above are the core elements in the NEAT algorithm, but they are usually not the only ones. Different NEAT implementations may add parameters to help with any specific implementation detail (Papavasileiou et al., 2021).

Figure 15

NEAT genotype to phenotype representation, adapted from (K. Stanley & Miikkulainen, 2002)



The NEAT algorithm, much like GAs, utilizes the noisy injection mutation method. However, because NEAT aims to introduce mutations to network topologies, it introduces two additional mutation methods: one for introducing connection links between neurons and another for adding new neurons to the network structure (Buckland, 2002). Whenever a new link or neuron is created, it is cross-checked with a database of previous structural innovations, which includes all previously created links and neurons along with their respective innovation numbers. If a new link or neuron is not found in the database, it is assigned a unique innovation number and added to the database. Otherwise, the structural change has already occurred and has a pre-existing innovation number. This innovative numbering system enables the NEAT algorithm to keep track of structural changes, which is essential for its crossover operation (Omelianenko, 2019).

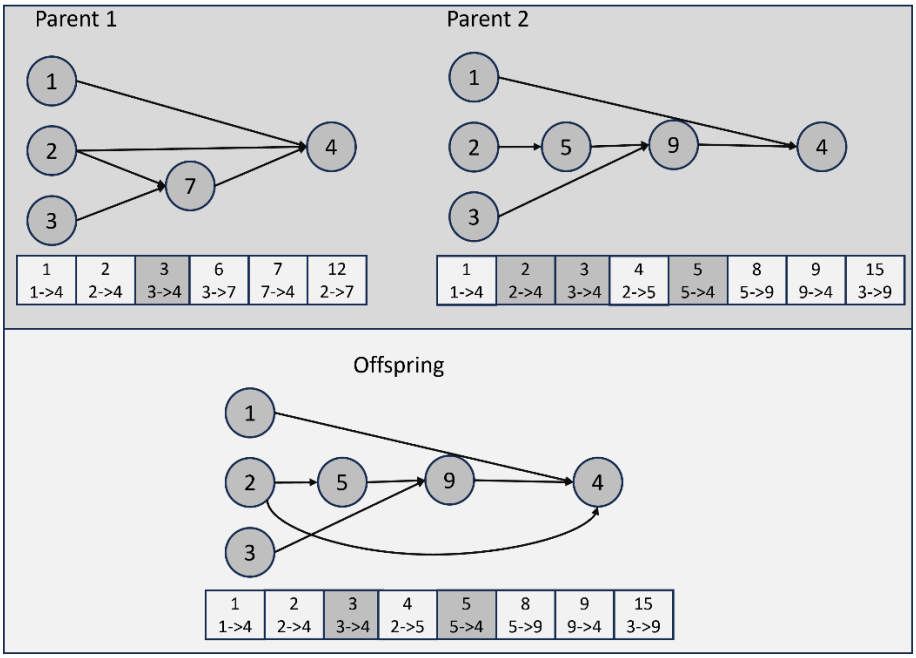
Because gene innovation numbers pass on through generations the NEAT crossover method can combine the genes of two parents successfully to form a new genotype, an illustration of this process can be seen in Figure 16. When comparing the genes of two parents, they can be categorized into three groups (Buckland, 2002):

- Matched Genes: These genes share the same innovation number in both parents, signifying a direct correspondence between their structural elements. Matched genes are randomly selected from either parent to be inherited by the offspring.

- Disjoint or Excess Genes: Genes that exist in only one of the parents can be categorized as either disjoint or excess genes. Whether a gene is considered disjoint, or excess depends on whether its innovation number falls within or outside the range of innovation numbers in the other parent. Only the disjoint and excess genes from the fitter parent are passed on to the offspring.

Figure 16

NEAT crossover representation, adapted from (K. Stanley & Miikkulainen, 2002). NEAT crossover, parent two is the fittest parent, genes in the darker grey are disabled.



When a genome incorporates a new structure, it can initially result in decreased performance as it may take several generations before the structure becomes more optimal, potentially even surpassing older genotypes. Regrettably, such diminished performance usually leads to the structure's elimination during the selection process. A similar phenomenon occurs when a population consists of genotypes of vastly different sizes. Smaller genotypes tend to optimize more quickly than larger ones, resulting in the premature elimination of larger not yet optimized genotypes. To address these challenges, the NEAT algorithm implements the concept of speciation (Omelianenko, 2019).

In NEAT, speciation involves categorizing and organizing genomes into different species as required. During the start of each generation, NEAT evaluates the genotypes of its population and determines their species based on a specific threshold. To assign a genotype to a species, NEAT compares its compatibility with the best-performing member of that species, using equation (51) (Papavasileiou et al., 2021). If the compatibility score falls below the threshold, the genotype is added to the relevant species. If it cannot be placed in any existing species, a new species is created, and the genotype becomes its representative. NEAT also employs explicit fitness sharing, where individuals within a species share their fitness scores. This is

achieved by dividing each individual's fitness by the number of individuals of the same species before the selection process. Additionally, NEAT also boosts the fitness of individuals of newly formed species and applies a penalty to those in older species. This approach is designed to promote diversity within the population and mitigate premature elimination of new genotypes (Buckland, 2002).

$$C = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 W \quad (51)$$

The equation (51) calculates the compatibility distance C between two genotypes. N is the size of the larger genome, E and D are the number of excess and disjoint genes respectively, W is the difference in connection weights between matched genes. c_1, c_2, and c_3 are coefficients used to fine-tune the relative significance of the three factors (Papavasileiou et al., 2021; K. Stanley & Miikkulainen, 2002).

2.4.5 – Novelty Search

Designing an effective fitness function is a pivotal aspect of any Neuroevolution algorithm. Typically, fitness functions employed in Neuroevolution are tied to the mechanics of the environment they aim to solve. These types of fitness functions are commonly referred to as goal-oriented or objective functions (K. O. Stanley et al., 2019). However, in deceptive environments, where the path to the goal might not be obvious, such objective functions often steer training agents towards converging on local optima. To counter this, Neuroevolution algorithms depend on their mutation operators to generate distinct phenotypes that result in fresh and superior solutions, enabling future generations to break free from local optima (Omelianenko, 2019). Nevertheless, this approach may not be effective enough in highly deceptive environments. In such cases, Neuroevolution can pivot towards optimizing a different type of fitness function, known as Novelty Search (NS).

NS takes a departure from the traditional approach by disregarding the intricacies of the environment dynamics. Instead, it rewards agents for generating novel behaviors that have not been previously encountered throughout the evolution process. As a result, NS motivates agents to explore their environment extensively, ultimately leading them to optimal solutions. By implementing NS alongside a basic GA model, researchers (Such et al., 2018) have

demonstrated superior performance compared to widely used DRL techniques and other Neuroevolution algorithms in solving the challenging Image Hard Maze problem.

NS uses a novelty metric to understand the uniqueness of an individual's behavior. More precisely, this metric can quantify the dissimilarity between an individual's behavior with that of any other behavior. The calculation of this difference may vary depending on the domain, but in the case of the Image Hard Maze problem, the original authors (Such et al., 2018) utilized the final position of an individual in the maze to define its behavior. Hence, in this situation, the novelty metric is calculated as the Euclidian distance between the final maze positions of two individuals.

In NS, as depicted in equation (52), the complete novelty score of a behavior is determined by computing the mean novelty metric across the k-nearest neighbors, which are sorted based on their behavioral distance from that specific behavior (Lehman & Stanley, 2011). A high novelty score signifies that a behavior stands out from its neighboring behaviors, while a lower novelty score indicates a greater similarity between a behavior and its neighbors.

$$\rho(x) = \frac{1}{k}\sum_{i=0}^{k} dist(x, \mu_i) \quad (52)$$

In equation (52), ρ is the novelty score for behavior x, k is the number of nearest neighbors, the function $dist$ is the novelty metric formula used, and μ_i is the ith-nearest neighbor of x with respect to the distance/novelty metric (Lehman & Stanley, 2011).

When evaluating the novelty of a behavior, the selected k-nearest-neighbors may come from the existing population of behaviors and/or from the most innovative behaviors discovered thus far through the evolutionary process. With each generation, NS determines whether any of the current population's behaviors qualify for inclusion in an archive of the most original behaviors found to date (Omelianenko, 2019). This determination is based on the behavior's novelty score, which must surpass a predefined threshold when compared to the behaviors stored in the archive.

Although research by Uber AI Labs (Such et al., 2018) has demonstrated the effectiveness of NS in learning continuous control tasks, several other studies (Cuccu & Gomez, 2011; Lehman & Stanley, 2010; Segal & Sipper, 2022) have suggested that NS can be further enhanced by integrating it with a goal-oriented fitness function. While these works employ different

methodologies for combining NS with standard fitness functions, their results consistently indicate improvements compared to relying solely on NS or goal-oriented fitness alone. For this reason, a hybrid selection mechanism for Neuroevolution algorithms is worth considering.

For this project, the method used was the hybrid NS approach, introduced by (Cuccu & Gomez, 2011), where the authors combine the normalized fitness score with the normalized novelty score linearly, as seen in equation (53). The hyperparameter denoted as 'γ' (ranging between 0 and 1) serves to govern the impact of the fitness and novelty in the algorithm. It's worth noting that the authors carry out normalization separately for both the fitness and novelty scores, scaling them to fall within the range of 0 to 1.

$$score(i) = (1 - \gamma) * \overline{fit}(i) + \gamma * \overline{nov}(i) \quad (53)$$

2.5 – Deep Learning in Games

While the usage of DL research in video games has exploded in the past decade, most articles on the subject explore how DL can be used to learn a game environment and play it at a superhuman level. The more famous examples of agents that perform at a superhuman level (Mnih et al., 2015; Silver et al., 2016) have already been mentioned in the introduction to this section. More recent examples include achieving a 100%-win rate in a Pokémon battling simulator (Simoes et al., 2020), and developing an AI agent that can defeat esports players in a popular MOBA game (Ye et al., 2020). These are but a few examples in a much broader field of research (Justesen et al., 2017).

Although successfully using DL to solve a specific game or game genre provides more insight into the game generalization capabilities of these algorithms, it does not provide much in terms of their validity for usage in game development-related problems. Luckily, there have also been research developments in the DL field that contemplate these problems. Most of the research on DL for video game development-related problems falls into five categories:

- Procedural content generation (PCG) – PCG is the use of algorithms to generate game content, such as levels, textures, and characters. DL has been used to create PCG algorithms that can generate a wide variety of content that is diverse, interesting, and of high quality (Togelius et al., 2024; Yannakakis & Togelius, 2018). For example, researchers have used DRL to automatically generate three-dimensional virtual environments that users can interact with (López et al., 2020),

another example consists of using a DRL model to restructure a level's layout, so it better fits the player's skill and ability (Huber et al., 2021). This can make game development more efficient by automating the creation of game content and making it possible to create a larger variety of content which would not be feasible while using traditional methods.

- Creating more believable nonplayer characters (NPCs) – Another area where DL is being used in game development is in creating NPCs that are more believable and engaging. Researchers have used DRL to train NPCs to make decisions based on their environment and the player's actions, which can make them more responsive and believable, for example, combining DRL with human guidance to create believable and diverse agents in serious games (Dobrovsky et al., 2017), this can make the game more challenging and engaging, as the NPCs are better able to detect and respond to the player's actions. Another example uses DRL for video game pathfinding (Dowling, 2022), as it can make the game more realistic and believable as NPCs move through environments more naturally.
- Using DL to model players' experiences – Another common point of research is using DL to help model players' experiences, researchers have proposed DRL models that balance the game experience in relation to players' skill level (Reis et al., 2021), adjusting game difficulty so players remain interested while playing. Another example is using DRL to use automated playtesting for player modeling (Roohi et al., 2021), this can help automatize game development in many ways, such as saving development time when doing game level balancing.
- DL agents trained to cooperate – Another area of research used DL to train multiple agents. This area of research is called multi-agent reinforcement learning (MARL). MARL is a subfield of reinforcement learning that deals with the problem of training multiple agents to cooperate or compete in an environment. Specifically for video games, MARL research focuses on agent cooperation, for example, in (Y. Wang et al., 2020) DRL agents learn to cooperate so they can effectively catch a superior evader agent. Another example shows how RL agents can be trained to play the mode capture the flag in "Quake III Arena" at a human level (Jaderberg et al., 2019). These examples show another way of making NPCs that are more interesting for players to interact with.

DL is an active and rapidly evolving area of research, and new developments and applications are emerging all the time. Applications of DL in video games are not just limited to what has been mentioned in this section. It is expected that the usage of DL in video games and game development will continue to grow in the future.

As mentioned before, this thesis seeks to apply DL methodologies to stealth games. Unfortunately, research on this topic is currently very limited. During the process of gathering research for this thesis, only two research articles on the topic were found, one uses a combination of behavior trees and RL to train agents in a stealth environment so that those agents can be used to help developers understand the impact of gameplay changes in levels (Gutiérrez-Sánchez et al., 2021). The second approach trains an RL algorithm to learn a stealth game with the help of a genetic algorithm that shapes the agent reward function (de Mendonça et al., 2016).

It's worth noting that stealth games are a specific genre, and the use of DL in stealth games is a niche area of research and development. Therefore, despite promising results, more research is needed to fully realize the potential of DL in this area.

3 – Testbed Environment

This chapter focuses on explaining the various implementation details that go into the application and creation of the thesis project: the stealth game levels and mechanics, the algorithms created, and the tools developed to measure the algorithm's performance.

3.1 – Stealth Game

As previously mentioned in Section 1.2, this thesis seeks to apply DRL algorithms to a game genre that has seen little research in the field of DL. As such, a custom stealth game was developed using the Unity game engine, specifically utilizing Unity version 2020.3.

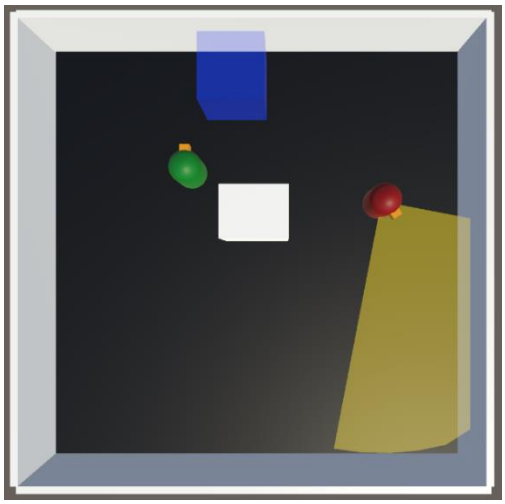
The primary purpose of the created stealth game is to serve as a dedicated testing environment for DL models. Consequently, certain features commonly found in games intended for human players, such as sound and visual effects, were not prioritized. Including these elements would have required additional work that deviates from the game's core objective as a testing ground. Instead, the focus was on implementing simple yet robust stealth game mechanics that are easy and intuitive to comprehend for users.

The implemented stealth game falls into the category of hide-and-seek puzzles, where the 'player' must find the exit of each level while evading being detected by enemies. Although the game can be played by humans, the primary intent is for the player character to be controlled by a DL model.

The game adopts a 3D top-down perspective, allowing the player to fully view the game environment from a fixed camera position, as illustrated in Figure 17. The player's available actions are limited to movement and "assassinating" enemies. Movement can be executed in eight directions: left, right, forward, back, left-forward, left-back, right-forward, and right-back, simulating keyboard movement. The "assassinating" action can only be performed when the player character is undetected near an enemy. By assassinating an enemy agent, the player can eliminate that agent from the level while remaining hidden. Additionally, the player character has the option to choose not to perform any action, resulting in a total of ten possible actions.

Figure 17

Stealth game level example, the player character in green, the enemy agent in red, the enemy cone of vision in yellow, the level exit/goal in blue, and walls/obstacles in white.



As mentioned earlier, completing a level requires the player to reach the end, represented by a blue cube. To achieve this, the player must navigate the level within a specified time limit while remaining undetected by enemy characters. If the player is detected by an enemy or the time limit expires, the level is considered a failure. Both level completion and failure represent terminal/final states.

Enemy agents are controlled by a simple scripted AI. They possess a cone of vision that enables them to detect the player if it falls within the agent's line of sight range, with no obstructing objects between them. Enemy agents can have different types of movement behavior. They can either remain stationary at a designated position or follow a predetermined path set by the developer.

All the implemented stealth game features described above were developed following standard Unity game development practices. Similar implementations of these features can be found in the Unity community forums, videos, and other resources.

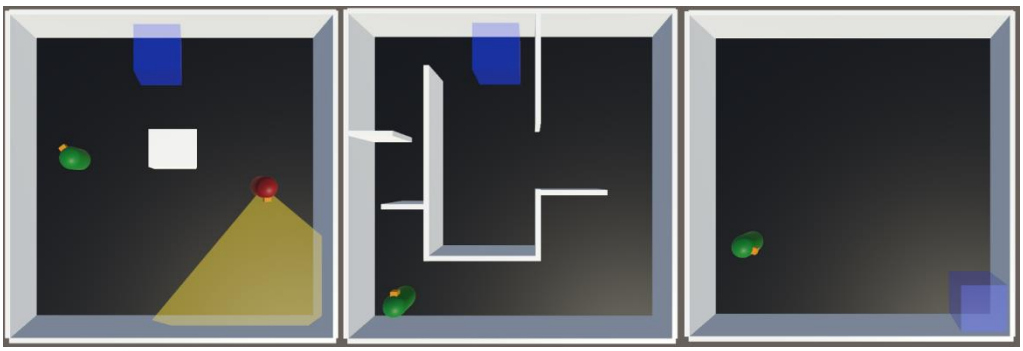
The game is made up of four distinct levels, shown in Figure 18:

- The first level (a) has a simple level structure with a fixed exit/goal, accompanied by an enemy agent that patrols along a predetermined path with a large cone of vision. The focus of this level is to observe the strategies employed by an agent in reaching the goal while avoiding detection by the patrolling enemy.

- The second level (b) contains no enemy agents but presents the most intricate maze structure among all levels, featuring a fixed exit or goal. The primary objective of this level is to assess an agent's capabilities in navigating complex level layouts and observing its perception of the shortest path to the goal.
- The third level (c) exhibits a basic level layout without any enemy agents. However, each time the level starts, the goal's position and the player's starting position are randomized within a set of predefined positions. The primary intention of this level is to evaluate an agent's ability to generalize the goal position rather than solely relying on memorizing the layout of the level.
- The fourth level (d) is the most complex and challenging level. It features a complex maze structure containing two patrolling enemies with a small cone of vision. The purpose of this level is to combine the main characteristics of levels one (a) and two (b) to provide the hardest challenge featured in the implemented stealth game.

Figure 18

Stealth game levels, the first level (a), the second level (b), the third level (c) and the fourth level (d).



(a) (b) (c)

(d)

The inclusion of levels featuring diverse challenges is intended to evaluate the capabilities of the implemented algorithms. By tailoring the levels to encompass different yet common stealth game scenarios, it becomes straightforward to assess the performance of each algorithm in a specific problem domain.

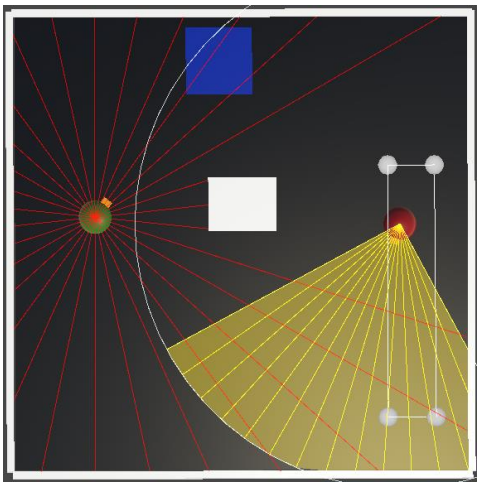
All DL models require a dataset to allow training and decision-making. To address this requirement, the implemented game captures a variety of game variables during each frame of a gameplay session. These variables are subsequently exposed to a DL model, enabling it to utilize them as needed. The recorded game features encompass the following:

1. The x and z position of the level exit or goal.
2. The x and z position of the player character.
3. An array of player-to-wall x and z positions captured at 30 incremented 12-degree y angles, centred and around the player.
4. The x and z positions of enemy characters.
5. The x and z positions that represent the end of an enemy line of sight.

Together the combination of these features makes up the state of the level. While features 1, 2, and 3 are consistently present, features 4 and 5 are only applicable when enemies are present within the level. Once an enemy agent is "assassinated," their position and line of sight values are set to 0 until the level is reset. Figure 19 shows some of the game features used to represent the environment state.

Figure 19

Stealth game environment features, red lines give the array of player-to-wall positions, yellow lines give the positions of the enemy line of sight. Grey circles connected by lines represent the enemy patrol path.



Despite the differences among the DL algorithms featured in this thesis, they all rely on an evaluation metric to be able to learn. In this project, environmental rewards serve as the evaluation metric. Whenever the player character executes an action within the environment, the player receives a reward, which can be positive or negative, indicating the quality of the action. The reward value associated with an action depends on the environment's current state. The following reward scenarios are considered:

- If an action successfully achieves the goal, the player receives the highest possible reward.
- If an action results in the player being caught by an enemy, the player receives the lowest possible reward.
- If an action leads to the assassination of an enemy, the player receives a positive reward.
- For all other environment states resulting from an action, the player receives a (usually very small) negative reward.

The specific values assigned to these four rewards are discussed in Chapter 4.

Given that some of the implemented algorithms benefit from having multiple player agents collecting data at the same time, a distributed version of the environment was implemented. This version uses the Unity Job functionality to take advantage of multithreading, this way any algorithm can control several player characters at the same time. When running this environment all player characters are “blind” to each other existence, as such they do not interfere with one another state. Apart from supporting multiple player characters at the same time, all the previously described environment features and rules are the same as the version that only supports one player character.

As previously mentioned, the primary objective of the implemented stealth game is to serve as a testing ground for DL algorithms. Therefore, all the aforementioned functionalities are abstracted away to ensure easy interaction with any algorithm. The design of this structure draws heavy inspiration from the OpenAI Gym library, which at the time of writing is maintained by the Gymnasium library (Gymnasium Documentation, 2023). Like that library, the implemented game provides the ability to reset the environment through a reset function and interact with it via a step function. The step function enables the algorithm to pass an action to the player character returning the resulting environment state of having performed that action, the associated reward, and a "done" flag indicating whether the returned state is a terminal state or not.

3.2 – Deep Learning Algorithms & Libraries

For this thesis, all the code pertaining to the deep learning algorithms has been developed entirely from scratch. This decision was motivated by several key factors:

- While Unity does have a plug-in called ML-Agents (Unity Technologies, 2017), which contains DRL models, it requires several external libraries to work, making it cumbersome to use and set up, but most importantly, ML-Agents only implements two DRL algorithms, proximal policy optimization (PPO) and Soft Actor-Critic (SAC). This limitation makes it impractical to use since the objective of this project is to evaluate multiple DRL and Neuroevolution algorithms.
- Building these algorithms from scratch directly within the game engine offers a more accurate representation of their computational cost within the context of game development. This approach allows us to understand the computational demands imposed by these algorithms on game performance.
- It is widely acknowledged that custom solutions tend to run faster than generic ones. Implementing every algorithm from scratch makes iterating and testing models faster and less cumbersome. This approach also minimizes the complexities associated with utilizing pre-existing frameworks.
- By eliminating external dependencies, the project becomes lightweight and more accessible. It can be effortlessly deployed on any machine that supports the Unity game engine, without the need for additional software installations or compatibility issues.

3.2.1 – Artificial Neural Network Library

Given that DL is a primary focus of this thesis, an ANN library was developed within the Unity game engine. This library contains essential functionalities commonly found in popular ANN libraries such as Keras and TensorFlow. The features provided by this custom ANN library include:

- A modular and parameterizable ANN architecture, that allows for easy creation of new ANN models with customizable configurations.
- The most used activation functions, namely ReLU, tanh, and softmax.

- Implementation of standard loss functions, including MSE and categorical cross-entropy loss function.
- Incorporation of the widely utilized ANN optimizer, ADAM.
- Manual calculation of gradients as the library does not incorporate automatic differentiation capabilities.

It is worth noting that many popular ANN libraries leverage Graphics Processing Units (GPUs) to significantly accelerate processing speeds. To compete with these established libraries, the ANN library developed in this project also harnesses the power of the GPU, utilizing Unity compute shaders to perform ANN operations. This means that most implementations of the methods described in Chapter 2 utilize compute shaders, this includes the vast majority of matrix operations such as, the forward pass with activation functions, back propagation with optimization, and other algorithm specific matrix operations.

3.2.2 – Deep Learning Algorithms

The second most important library implements all DL algorithms used in the project. It includes Neuroevolution and value-based model-free DRL algorithms.

The library implements the following DRL value-based algorithms:

- DQN – The implementation of this algorithm follows a standard/common approach.
- Double DQN – The implementation of this algorithm also follows a standard/common approach.
- N-step DQN – The implementation of this algorithm adheres to a standard/common methodology.
- DQN with PER – The implementation of this algorithm aligns with the standard approach. It utilizes the Sum-Tree data structure to sample experiences, providing computational efficiency.
- Noisy DQN – The implementation of this algorithm follows a standard/common approach. The calculation of weight noise follows the suggestion provided in the original paper (Fortunato et al., 2017), which recommends using the Factorized Gaussian Noise technique. Another technique used is to sample noise into a very large buffer at the start when initialing the algorithm, and randomly sampling from that buffer

whenever noise is needed, which improves the computation time of the algorithm (Salimans et al., 2017).

- Dueling DQN – The implementation of this algorithm largely aligns with the standard approach. The key distinction lies in the selection of the action with the highest $Q(s, a)$ value. Due to the nature of the Dueling Q-value equation (42), only the advantage network needs to be run to determine the action with the highest Q-value, as the advantage values are sufficient.
- Categorical DQN – The implementation of this algorithm follows a standard/common approach.
- Rainbow DQN – The implementation of this algorithm follows the standard approach.

The library implements the following Neuroevolution algorithms:

- RS – This algorithm's selection and mutation methods follow the methodology described in section 2.4 for Random Search. Noise creation and sampling methods are the same methods used by CMA-ES.
- CMA-ES – The implementation of this algorithm follows the approach described in section 2.4 for CMA-ES, with a few exceptions. The calculation of weight noise uses the Factorized Gaussian Noise technique used by the Noisy DQN algorithm (Fortunato et al., 2017). This implementation uses the ADAM optimizer for gradient ascent instead of the standard update function.
- GA – For the thesis project this algorithm implements all the methods described in section 2.4 for genetic algorithms. It only deviates slightly in the crossover method, where it performs single-point crossover per network layer, instead of having only crossover one point for the entire network parameter space.
- NEAT – The implementation of this algorithm mostly adheres to the standard/common approach. The main differences in this implementation are: no inter-species mating, no activation function mutation, and no bias neurons.
- NS – The implementation of this algorithm follows the methodology described in section 2.4 for Novelty Search. It should be noted that this algorithm is not intended to be used independently, but instead in conjunction with the Neuroevolution algorithms previously mentioned. To distinguish a Neuroevolution algorithm that incorporates NS from its base implementation, the suffix 'NS' is appended to the base name, for example, 'RS-NS,' 'CMA-ES-NS,' 'GA-NS,' and 'NEAT-NS'.

3.2.3 – Tools and Frameworks

Since Unity primarily focuses on game development, it lacks certain essential tools and libraries commonly used in data science and machine learning. Consequently, all required support tools and libraries were developed within the Unity project:

- A custom graph library was created to visualize the progress of implemented algorithms during training as well as other related tests that necessitate graph representations. This library enables the direct display of results within the Unity game engine. It draws inspiration from popular graph libraries like Matplotlib.
- To facilitate large-scale testing, an automated testing library was developed. This library enables the execution of numerous tests for any algorithm, automatically recording and saving relevant testing data into easily understandable JSON files. This feature simplifies the process of conducting extensive tests, ensuring efficient data collection.

4 – Experimental Setup and Results

This chapter seeks to describe the methods used to train and test the implemented algorithms. It explains how game data is processed before being used by any algorithm, and details how the search for hyperparameter tuning was performed, giving an understanding of how the different hyperparameters affect the performance of an algorithm.

4.1 – Data collection & game environment setup

To solve any game environment, DL algorithms need to receive a set of environment features so that they can learn and act. The features used to represent the state of the game environment have already been thoroughly described in section 3.1, but in short, these are: the goal x and z position, the player x and z position, an array of player-to-wall x and z positions, enemies x and z position and the x and z positions of their line of site. Given that the implemented game was built from scratch, any set of features could have been used.

In the realm of DL research, particularly in the context of DRL research, many games use images to depict the game environment state. Convolutional Neural Networks (CNNs) are typically employed to enable DL models to learn from such image data effectively. However, significant time and resources would be required to implement CNNs due to the nature of developing each algorithm from scratch. Consequently, the decision was made to allocate these resources toward the development of other algorithms more pertinent to the objectives of this project.

Existing research shows that games not relying on image-based state representations make use of domain knowledge specific to the implemented game itself. The downside of this approach is the lack of generalizability to games of different genres. Unfortunately, the available research on state representation in the context of stealth games is limited. However, in Gutiérrez-Sánchez et al., (2021) the authors present a similar game to the one implemented in this project and propose a set of features that can effectively represent the game state, although this project uses different features to represent the environment state.

It is important to mention that another state feature representation was tested, where the array of player-to-wall was represented by the distances instead of positions. This approach proved to be considerably inferior on levels one and two but had superior results on level three, results

can be seen in Table 9 in the Appendix A section. Since levels one and two better represent the dynamics of a simple stealth game, the choice was made to represent the array of player-to-wall as positions. The next section gives a more in-depth description of how these tests were made.

Given that all state features correspond to 2D positions within the game's level, a normalization process is applied using equation (54). This formula ensures that each axis value is scaled to fall within the range of -1 and 1, based on the minimum and maximum values for that axis. In the context of this project, the minimum and maximum values for the x and z axes denote the dimensions of the level. As all levels share the same size, these values remain constant: -7 for both x and z min, and 7 for both x and z max.

$$2 * \frac{value - min}{range} - 1 \quad (54)$$

All the algorithms implemented within this project rely on evaluation metrics, specifically in the form of rewards, to comprehend the consequences of their actions. By analyzing these rewards, the algorithms can learn and progressively make decisions that result in improved performance. As thoroughly detailed in section 3.1, the stealth game encompasses four distinct types of rewards: reaching the goal, assassinating an enemy, being detected by an enemy, and executing actions that do not fall into the aforementioned categories. The corresponding reward values assigned to these outcomes are 50, 15, -20, and -0.01, respectively. Although unintentional, this reward structure bears some resemblance to the one employed in (Gutiérrez-Sánchez et al., 2021). This reward design tries to encourage a learning agent to complete the level as swiftly as possible while avoiding steering the agent toward any specific playstyle.

Reward engineering plays a significant role in RL and Neuroevolution. Given the lack of research on reward engineering for stealth games, determining the types of rewards that yield superior performance can be a time-consuming process. Thus, this work conducted initial straightforward tests until satisfactory performance was achieved with the DQN algorithm, which was the first algorithm developed within this project. All subsequent tests utilized this reward structure.

4.2 – Initial experiments

To fulfil the core objectives of this thesis, every implemented algorithm must be trained across the various levels within the stealth game. This enables a comparative analysis of their overall performance and provides insights into their potential as QA tools.

To effectively train DL algorithms, it is customary to conduct hyperparameter tuning. This process involves adjusting the model's parameters in pursuit of optimal performance. There exist multiple techniques for hyperparameter tuning, but for this project, a manual approach was adopted. This entailed iteratively tweaking the model parameters, evaluating the outcomes, and repeating the process until no significant changes in the results were observed. Given that each implemented DL algorithm contains an extensive number of potential hyperparameter combinations, manual tuning approaches typically fall short of exhaustively exploring all possibilities. However, due to the researcher's active involvement and guidance throughout the tuning process, it is often possible to discover hyperparameter configurations that yield satisfactory performance (within the constraints of available time and resources), even if they may not be the absolute best parameters.

Considering the range of algorithms implemented in this project, employing identical methods for hyperparameter tuning across all algorithms is impractical. For instance, DQN and GA, despite both being utilized to train NNs, share very few hyperparameters and use entirely distinct approaches to training. DQN utilizes gradient descent methods, whereas GA leverages natural selection methods. Consequently, each algorithm "family" necessitates a unique procedure for hyperparameter tuning.

Nevertheless, there are certain commonalities observed in the tests that facilitate a fair comparison of algorithms, even in cases where they differ significantly. These commonalities encompass the following aspects:

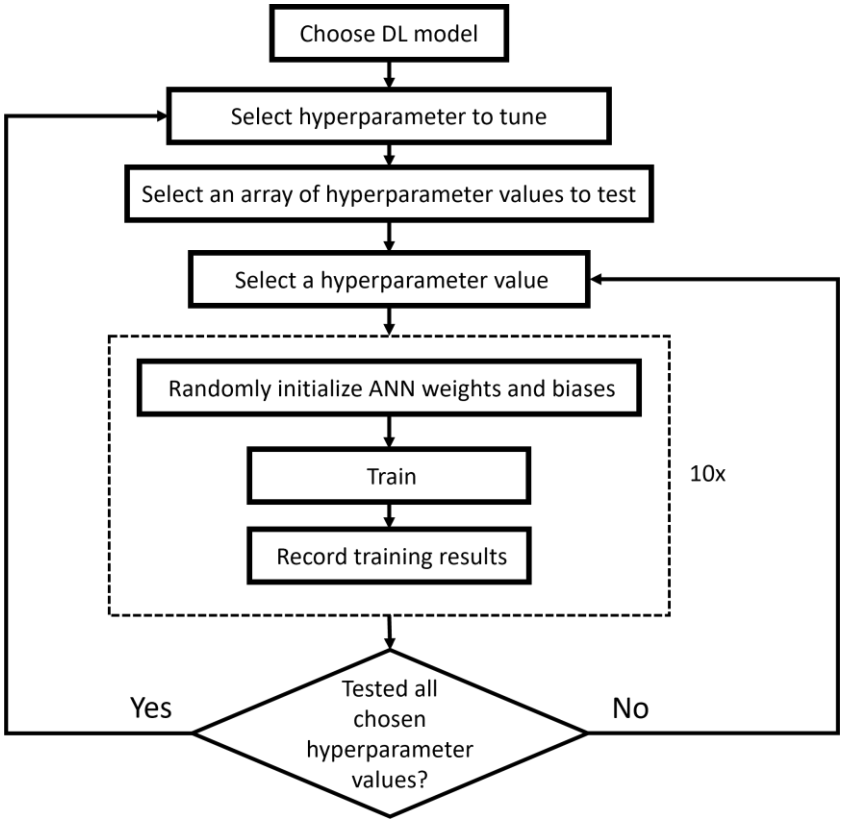
1. Each episode on any of the tested levels has a maximum duration of 700 steps. If this threshold is exceeded, the level is reset.
2. A training session is deemed complete once 300 episodes have been accomplished.
3. All algorithms initiate training from scratch, implying that they start with a network containing untrained values.
4. Measurements indicating algorithm performance are taken once a training session is complete.

5. To evaluate an algorithm's performance on a specific level, it undergoes ten complete training sessions on that level. The average of performance metrics obtained from these sessions forms the baseline for assessing the algorithm's effectiveness on the level.
6. Every algorithm is thoroughly tested on levels one (a), two (b), and three (c), this gives insight into the capabilities of an algorithm in solving different tasks that are common in stealth games.

These procedures establish the baseline for evaluating the performance of the developed algorithms, aiding in algorithm comparisons and hyperparameter tuning. A summarized version of the hyperparameter tuning prosses utilized in this work can be seen in Figure 20.

Figure 20

Diagram summarizing the hyperparameter tuning prosses conducted in this research.



It is important to mention that before initiating the hyperparameter tuning process, all algorithms underwent preliminary testing to identify potential hyperparameter combinations

capable of achieving satisfactory performance on levels one (a), two (b), and three (c). These initial tests were conducted more superficially, primarily relying on intuition and limited data recording. Although the specific test data and parameter values from this preliminary phase were not meticulously documented, the hyperparameters derived through this initial exploration served as a starting point for the subsequent comprehensive and in-depth hyperparameter tuning described in this chapter.

4.2.1 – DQN & extensions Hyperparameter Tuning

The first group of algorithms tested belonged to the Q-Learning family, which contains, DQN, Double DQN, Dueling DQN, N-step DQN, DQN with prioritized experience replay, Noisy DQN, Categorical DQN, Rainbow DQN. Since DQN serves as the foundation for all the other algorithms within this group, many of the hyperparameters remain consistent across them. For this reason, the approach used to find optimal hyperparameters for each algorithm involved initially conducting hyperparameter tuning for the DQN algorithm. The resulting best-performing hyperparameter combinations were subsequently used as the base for tuning the other algorithms within the group.

The DQN algorithm has a large range of different hyperparameters, with some primarily associated with the NN structure, while others are more related to the Q-Learning aspects of the model. The following hyperparameters were considered for tuning:

1. NN parameters initialization function
2. Initial NN parameters standard deviation (if applicable)
3. Number of NN layers
4. Number of neurons per hidden layer
5. ADAM optimizer learning rate and decay values
6. Sample/batch size
7. Gamma/discount rate
8. Target network copy period

Each hyperparameter configuration was tested twice, once using the ReLu activation function and another using the Tanh activation function. This approach aimed to assess the impact of different activation functions on the performance of the DQN algorithm.

The initial focus of hyperparameter testing examined the impact of different NN initialization functions, considering varying values for the standard deviation when applicable. This group of functions included Xavier, Xavier normalized (Glorot & Bengio, 2010), He (He et al., 2015), random normal, and random uniform. The results obtained from these tests revealed notable disparities among the NN initialization functions, with Xavier, random normal, and random uniform demonstrating superior performance compared to the others. The results achieved using these three methods were very similar, making it challenging to identify the best initialization function. For subsequent tests, the random uniform function with a standard deviation of 0.005 was selected due to its lower computational requirements. A detailed summary of the results from these experiments can be found in Appendix A Table 10.

The next set of parameters to be tested involved the number of layers in conjunction with the number of neurons per layer. Specifically, the number of layers examined was two, three, and four, while the neuron sizes per layer were 32, 64, 128, 256, and 512. The results obtained from these tests revealed that a four-layer NN performed worse than the two- and three-layer counterparts for the tested neuron sizes. Additionally, the results indicated that employing either 32 or 512 neurons led to weaker performance when compared to the other quantities. On the other hand, using two or three layers with neuron sizes of 64, 128, and 256 provided similar results. Among these options, the configuration of three layers with a neuron size of 128 provided overall stronger results. Consequently, subsequent tests utilized this NN structure. A detailed summary of the results from these experiments can be found in Appendix A Table 11.

The third group of tests focused on the hyperparameters of the ADAM optimizer, specifically the learning rate and the decay values. This group of tests was more exhaustive than the previous two, as it involved exploring numerous combinations of Lr and decay values, results can be seen in Appendix A Table 12. While many combinations produced strong results, the following emerged as the overall best-performing ones:

1. Lr: 5e-3, Decay: 1e-3
2. Lr: 1e-3, Decay: 1e-4
3. Lr: 5e-4, Decay: 1e-5
4. Lr: 1e-4, Decay: 1e-7

For the subsequent DQN tests only the last two combinations were utilized. However, for the other algorithms within the Q-Learning family, all four combinations were employed during testing.

The next hyperparameter tested was the target network update/copy period. This hyperparameter determines how many learning steps need to occur before the target network is updated with the online network values, this can be seen in algorithm pseudocode in Table 2. Several values for this parameter were used, but they did not seem to impact algorithm performance by much, as results were similar across different update/copy values, as can be seen in Appendix A Table 13. However, 50 steps between every update provided the best results, as such, subsequent tests use 50 as their target network update/copy period.

The next tested hyperparameter was the batch size. According to the literature, it is standard to use a batch size of 32 for the DQN algorithm and its extensions (Stooke & Abbeel, 2018). For this reason, the range of values explored for batch size ranged from 16 to 96. According to the obtained results, the chosen range of batch sizes did not lead to significant variations in algorithm performance. All tested batch sizes produced similar outcomes, as can be seen in Appendix A Table 14. Consequently, for subsequent tests, a batch size of 32 was adopted. This selection was based on the observation that it offered strong performance while remaining one of the computationally less demanding options.

The final hyperparameter tested for the DQN algorithm was the gamma value, also known as the discount rate. A range of gamma values, varying from 0.9 to 0.99, were explored during testing. The results from these experiments, as displayed in Appendix A Table 15, demonstrated a consistent decrease in algorithm performance as the gamma value decreased. As such, 0.99 was the selected gamma value for subsequent tests since it produced the best results.

Table 3 provides the best-found hyperparameters for the DQN algorithm. But it is worth noting that other tested combinations of the Lr and decay values also provided very strong results.

Table 3

DQN hyperparameters found that offer the best overall performance, for both Tanh and ReLu activation functions

Initialization function	**Layers**	**Neuron number**	**Lr & decay**	**Update target network**	**Batch size**	**Gamma**
Random Uniform (std 0.005)	3	128	0.0001; 1e-7	50	32	0.99

After obtaining data on the potential best hyperparameter combinations for the DQN algorithm, the hyperparameter tuning process continued to the DQN extensions.

For both Double DQN and Dueling DQN, it was noted that these algorithms share the same hyperparameters as the DQN algorithm. Hence, there was no need for separate hyperparameter tuning for both. Instead, the best-performing hyperparameter combinations from the DQN tests were directly applied to the Double DQN and Dueling DQN algorithms to assess their performance, as can be seen in Appendix A Tables 16 and 19, respectively. The results revealed that, on average, the Double DQN algorithm exhibited slightly stronger performance compared to that of the DQN algorithm.

The subsequent DQN extension subjected to testing was the N-step DQN algorithm, which introduces a new hyperparameter known as "n-step." To tune this algorithm, a range of values between 2 and 10 was explored for the n-step parameter, in conjunction with the best hyperparameter combinations found for the DQN algorithm. As indicated in Appendix A Table 17, results show that the N-step DQN algorithm exhibited robust performance across all tested n-step values. Furthermore, the outcomes revealed only marginal differences in performance, making it challenging to definitively identify the n-step value that yields the absolute best performance. Nevertheless, an n-step value of 3 emerged as one of the strongest performers, while also being computationally efficient. Consequently, for subsequent tests and evaluations, an n-step value of 3 was selected as the optimal choice.

The subsequent extension under examination was Noisy DQN, which introduced a new hyperparameter known as "sigma." This parameter is part of the initialization function of the sigma weights and sigma bias values of a noisy network layer. In the original research paper (Fortunato et al., 2017), the authors recommend utilizing a sigma value of 0.5. However, the tests conducted in this thesis involved exploring sigma values close to 0.5. Upon analysing the final results, presented in Appendix A Table 18, the sigma value of 0.05 led to the best results. Notably, the sigma value of 0.01 showcased almost equal performance. However, to gain a deeper understanding of the optimal sigma value, additional tests would be required.

The next DQN extension tested was the Distributional DQN. This extension introduces the concept of a support vector, consisting of a fixed size (number of atoms) and a range represented by minimum and maximum values, which in turn reflects the potential range of achievable rewards. In the original paper (Bellemare et al., 2017), the authors suggest an atom size of 51 and recommend using -10 and 10 as the min and max atom values, respectively. However, the experiments conducted in this project explored combinations of min and max values that are more aligned with the possible environment rewards within the stealth game. Upon analysing

the results obtained from testing various combinations of min and max values, as displayed in Appendix A Table 20, it is evident that differences in performance were not substantial, as the results appeared to be relatively similar. Nonetheless, it was observed that the recommended range of -10 to 10 performed the worst among the tested combinations, while the combination of -25 to 63 demonstrated the best performance, even if only marginally so.

The next extension tested was PER, which introduces two new tunable hyperparameters, α and β. According to the original paper (Schaul et al., 2015), α controls the degree of aggressiveness in prioritizing experience sampling, while β controls the level of bias correction applied during training. The authors suggest using $\alpha = 0.6$ and $\beta = 0.4$ to achieve optimal results. Appendix A Table 21 shows the results from testing a range of combinations for α and β. These results revealed that lower values for α consistently yielded the best performance outcomes, while β values around 0.5 showed better performance compared to both low and high values. Overall, the combination that provided the strongest overall results was found to be $\alpha = 0.2$ and $\beta = 0.6$. Nevertheless, it is worth noting that other β values, when paired with an α value of 0.2, also delivered comparable outcomes.

The last extension tested was the Rainbow DQN algorithm. This algorithm consists of joining all the previous DQN extensions into one cohesive algorithm, consequently, the Rainbow DQN algorithm itself doesn't introduce any new hyperparameters. For this reason, the Rainbow DQN was tested utilizing the best combinations of hyperparameter values derived from the previous DQN extension tests. The exception from this approach pertains to the utilization of noisy layers within the Rainbow DQN network architecture. The results from these experiments, as seen in Appendix A Table 22, show that using noisy layers only on the network's Value layers provides the best overall results. However, it is worth noting that the other tested combinations of noisy layers also delivered similar outcomes.

Overall, the best extension-specific hyperparameters found were the following:

- N-Step – step of value 3
- Noisy DQN – sigma value of 0.05
- Distributional DQN – atom min and max value of -23 and 63 respectively
- PER – alpha and beta values of 0.2 and 0.6 respectively
- Rainbow DQN – Noisy Value network layers

4.2.2 – Neuroevolution algorithms Hyperparameter Tunning

The second group of algorithms tested belonged to the Neuroevolution family, which contains, RS, CMA-ES, GA, and NEAT, as well as their NS counterparts, RS-NS, CMA-ES-NS, GA-NS, and NEAT-NS. While these algorithms share many tunable hyperparameters, it is important to recognize that optimal values for one algorithm may not necessarily yield the best results for another, as each approach is unique. Therefore, shared hyperparameters are thoroughly tested individually by each algorithm, which contrasts with the approach taken in the previous group of tests examining the DQN algorithm and its extensions.

Despite the differences between Neuroevolution algorithms and the DRL algorithms implemented, the best ANN initialization parameters identified for the DQN algorithm (as outlined in Table 3) were also applied to the implemented Neuroevolution algorithms. This decision was motivated by the desire to streamline the hyperparameter tuning process for the Neuroevolution algorithms and, notably, due to promising preliminary results achieved using the same ANN initialization parameters.

Similar to the previous group of tests and for the same practical reasons, each hyperparameter configuration was tested twice, once employing the ReLu activation function and once using the Tanh activation function.

It is important to note that the methodology employed for hyperparameter tuning in this project may not effectively evaluate the ability of Neuroevolution algorithms to generalize environments that change for each training cycle or generation. Consequently, some Neuroevolution algorithms were not assessed on level three, and even for those that were, their performance should not be seen as their true ability to generalize the level. A comprehensive evaluation of their generalization capabilities would require testing under a different format.

The RS algorithm was the first to undergo hyperparameter tuning, with the following hyperparameters being considered for tuning:

- Noise standard deviation.
- Number of ANN layers.
- Number of neurons per hidden layer.
- Population size.

The initial tests focused on measuring the impact of the standard deviation (std) of the noise sampled for use in the mutation step. The range of tested values for the noise std spanned from 0.005 to 0.03. The results obtained showed that lower values of noise std tend to show overall worse results than higher values. According to Appendix A Table 23, a noise std of 0.03 produced the best results, with slightly lower values also producing comparable outcomes. However, as the highest tested value was also the best, it remains unclear if further increases in noise std would lead to better results.

The next parameter to be tested was the number of ANN layers. Specifically, the number of ANN layers examined were two, three, and four. Based on the data presented in Appendix A Table 24, using three layers produced the overall best results, with the other configurations displaying only marginally inferior performance. As such, further RS tests utilized ANNs with three layers.

The third hyperparameter tested was the number of neurons per layer. A range of 32 to 256 was used for these tests. The results obtained showed that lower neuron numbers tended to produce marginally better results than higher counts, as seen in Appendix A Table 25. Consequently, a neuron count of 32, representing the lowest tested size, provided the best results.

The final RS parameter tested was the population size. Given that even small increases in the population size can have a large impact on computational performance, the range for this parameter was kept low, as such the tested range was between 26 and 76. The results show that increasing the population size leads to better outcomes, albeit only slightly within the tested range, as evidenced by the data presented in Appendix A Table 26.

The finalized set of optimal hyperparameters found for the RS algorithm is summarized in Table 4.

Table 4

RS hyperparameters found that offer the best overall performance, for both Tanh and ReLu activation functions

Initialization function	Layers	Neuron number	Noise std	Population size
Random Uniform (std 0.005)	3	32	0.03	76

The next Neuroevolution algorithm considered for hyperparameter testing was the CMA-ES algorithm. In addition to all the hyperparameters shared with RS, the CMA-ES algorithm implemented in this project incorporates two additional hyperparameters: learning rate and decay. These additions are necessitated by the utilization of the ADAM optimizer.

Although the CMA-ES algorithm utilized in this project adheres to the methodology outlined in section 2.4 for CMA-ES, the hyperparameter tuning version did not incorporate any fitness normalization techniques, including the Rank normalization approach described in the chapter. The reasoning behind this decision is that fitness normalization could be deemed a form of hyperparameter tuning, therefore Rank Normalization was regarded as an aspect of hyperparameter tuning and was subject to specific tests designed to compare its performance against non-normalized fitness scores.

The first group of tests focused on the ADAM optimizer hyperparameters, specifically the learning rate and decay values. Testing these parameters involved evaluating various combinations of them. According to the results obtained, seen in Appendix A Table 27, many combinations produced good results. For this reason, the following four combinations of Lr and decay values were used for subsequent tests:

1. Lr: 1e-2, Decay: 1e-3
2. Lr: 1e-2, Decay: 1e-5
3. Lr: 5e-3, Decay: 1e-4
4. Lr: 5e-3, Decay: 1e-6

The next hyperparameter under analysis was the Noise std value. The tested values for the Noise std ranged from 0.01 to 0.025. The results shown in Appendix A Table 28, suggest that lower Noise std values tend to lead to better outcomes. Overall, the result that provided the best results was a std of 0.015, although the value 0.01 provided comparable results.

The third hyperparameter tested was the number of ANN layers. Specifically, the number of ANN layers examined were two, three, and four. Based on the data presented in Appendix A Table 29, using three layers produced the best results, with the other configurations displaying significantly inferior performance. As such, further CMA-ES tests utilized ANNs with three layers.

The next hyperparameter under scrutiny was the number of neurons used per hidden layer, with a range spanning from 32 to 256. The outcomes from this evaluation show that utilizing 32

neurons per layer resulted in notably inferior results compared to the other values, whereas employing 256 neurons per layer yielded the best outcomes. Interestingly, the remaining tested values, except for 32, demonstrated similar performance levels to using 256, as depicted in Appendix A Table 30.

To determine the effectiveness of implementing the Rank rewards technique versus the current method of not normalizing fitness scores, a test was conducted. Appendix A Table 31 demonstrates that the current approach consistently outperformed the utilization of Rank rewards. Therefore, the Ranked rewards technique was not used on subsequent tests.

The last CMA-ES parameter tested was the population size. The data presented in Appendix A Table 32 shows that a population of size 50 yielded the overall best results. Because the tested range is short, it is difficult to ascertain if CMA-ES truly benefits from smaller population sizes.

The finalized set of optimal hyperparameters found for the CMA-ES algorithm is summarized in Table 5. While this table provides the best-found hyperparameters for the CMA-ES algorithm, it is worth noting that other tested combinations of the Lr and decay values also provided very strong results.

Table 5

CMA-ES hyperparameters found that offer the best overall performance, for both Tanh and ReLu activation functions

Initialization function	Layers	Neuron number	Lr & decay	Noise std	Population size
Random Uniform (std 0.005)	3	256	0.01; 1e-5	0.015	50

The GA algorithm was the next Neuroevolution algorithm to be hyperparameter-tuned. This algorithm uses all the RS parameters while introducing four brand new ones: tournament size, elitism number, minimum mutation, and maximum mutation. Only the first two were considered for tuning while the other two remained constant throughout testing. The tournament size defines the sample size used to do the tournament selection method, while the elitism number defines the number of best-performing individuals that advance to the next generation. The minimum and maximum mutation parameters were introduced via the adaptive mutation technique. These parameters define the mutation rate of low and high-quality individuals, and

an offspring's overall mutation is defined by the sum of its parent's mutation rates, the minimum and maximum mutation values were 0.02 and 0.1 respectively.

Like with the RS hyperparameter search, the first tunable parameter considered was the noise std sampled for use in the mutation step. The range of tested values for this parameter spanned from 0.005 to 0.03. The obtained results show that noise std values below 0.015 have weaker performance compared to higher values, while noise std values of 0.015 and above had comparable performance, as can be seen in Appendix A Table 33. Overall, a noise std of 0.015 produced the best results and was subsequently used for the next tests.

The next parameter under examination was the tournament size, the range tested was between 2 and 10. The results obtained show that the GA performance is not significantly impacted by small differences in tournament size, with the exception of a size of 2 (the minimum possible value) where it demonstrated a clear drop in performance, as can be seen in Appendix A Table 34. Overall, the best performance was achieved with a tournament size of 10.

The third parameter under evaluation was the elitism number. The tested range for this parameter fell between 1 and 7. The gathered results demonstrate that the GA algorithm is robust to small changes in elitism number, as all the tested values showed comparable results, as seen in Appendix A Table 35. Overall, the elitism value that offered the best performance was the value 3.

The next parameter to be tested was the number of ANN layers. Specifically, the number of ANN layers examined were two, three, and four. Based on the data presented in Appendix A Table 36, using three layers produced the overall best results, with the other configurations displaying only marginally inferior performance.

The subsequent hyperparameter under evaluation was the number of neurons per layer. A range of 32 to 256 was used for these tests. The results obtained in Appendix A Table 37 showed that none of the neuron numbers tested showed significant differences in performance, none stood out as better or worse. Overall, the best results were achieved using a neuron count of 64.

The final GA evaluated hyperparameter was the population size. The data presented in Appendix A Table 38 shows that increasing the population size corresponds to better algorithm performance. For this reason, the highest population size tested (76) produced the best results.

The finalized set of optimal hyperparameters found for the GA algorithm is summarized in Table 6.

Table 6

GA hyperparameters found that offer the best overall performance, for both Tanh and ReLu activation functions

Initialization function	Layers	Neuron number	Minimum Mutation	Maximum Mutation	Noise std	Tournament size	Elitism size	Population size
Random Uniform (std 0.005)	3	64	0.02	0.1	0.015	10	3	76

The final Neuroevolution algorithm tested was the NEAT algorithm. Interestingly, it only shares three parameters with the other Neuroevolution algorithms, namely the population size, noise std, and the initialization function for the ANN parameters. In addition to these parameters, NEAT introduces fifteen new hyperparameters, which unfortunately could not be thoroughly tuned due to the time constraints for this project. A list of these parameters can be seen in Appendix A Table 39. However, the performance of the shared hyperparameters was still evaluated.

The initial hyperparameter subjected to evaluation was the noise std value, with the tested range spanning from 0.005 to 0.03. The results obtained indicated that within this range, only a noise value of 0.005 consistently yielded weaker overall results. Conversely, the other tested values demonstrated comparable performance, as illustrated in Table 40. Ultimately, the noise std value that produced the best overall results was determined to be 0.015.

The second and final parameter tested was the population size. Given the tested range of population size values, a size of 50 provided the best performance, as seen in Appendix A Table 41.

The finalized set of the shared optimal hyperparameters found for the NEAT algorithm is summarized in Table 7.

Table 7

NEAT shared hyperparameters found that offer the best overall performance, for both Tanh and ReLu activation functions

Initialization function	Noise std	Population size
Random Uniform (std 0.005)	0.015	50

Results from all Neuroevolution algorithms show that under the current testing methodology, they are unable to solve level three (c). As a result, a new methodology was developed with the specific goal of achieving better results on level three. The primary objective of this new methodology was to encourage better generalization. To do this, Neuroevolution algorithms were designed to create new generations only after completing ten episodes, instead of one episode as in the previous approach. Previously, the total number of episodes per test was 300, but with the new methodology, this number was increased to 3000, allowing for the same number of generations to be created as with the previous methodology.

With this new methodology, all Neuroevolution algorithms tested manage to significantly increase their performance on level three, as can be seen in Appendix A Table 42. Notably, the GA and NEAT algorithms exhibited superior performance compared to other Neuroevolution algorithms, while displaying comparable performance to each other.

Despite the superior results displayed by this new approach, further tests conducted reverted back to the previous approach, as utilizing 3000 episodes per test was not possible given the time constraints of this project.

After completing the hyperparameter tuning process for all Neuroevolution algorithms, the tests proceeded to them in combination with the NS technique. The resulting hybrid algorithms are identified by appending 'NS' to their original names, resulting in RS-NS, CMA-ES-NS, GA-MS, and NEAT-NS. These modified algorithms retained their original parameters while introducing two new ones related to the added NS technique. The first parameter, known as 'novelty relevance,' governs the degree to which the novelty metric influences the algorithm's overall fitness. The second parameter, 'K-nearest neighbors,' controls the number of neighbors used when calculating the novelty score for an individual. To evaluate the effect of NS, the best-found hyperparameter combinations for non-NS parameters were employed, with a focus on tuning the novelty relevance parameter while keeping the k-nearest neighbors value constant at 10 throughout the experiments. The range of tested novelty relevance values ranged from 0.2 to 0.8.

The first hybrid algorithm tested was RS-NS. The results from Appendix A Table 43 show that RS-NS yields the best results with a novelty relevance value of 0.2, with an observable decrease in algorithm performance as the novelty relevance value increases. Overall results demonstrate that RS-NS exhibits stronger performance than RS without NS on level two (b), while performance on other levels remains comparable between the two.

The second NS algorithm under examination was CMA-ES-NS. The results indicate that a novelty relevance of 0.8 produces inferior outcomes compared to the other tested values. Novelty relevance values of 0.2 and 0.5 exhibit similar performance, with a slight advantage observed for the latter, as can be seen in Appendix A Table 43. A comparison of the best results between CMA-ES with NS and without NS reveals their similarity, making it challenging to determine the superior approach.

GA-NS was the third algorithm under evaluation. The results indicate that a novelty relevance of 0.8 produces inferior outcomes compared to the other tested values. Novelty relevance values of 0.2 and 0.5 exhibit similar performance, with a slight advantage observed for the latter, as can be seen in Appendix A Table 44. The best results obtained from the GA and GA-NS show that using NS yields marginally better performance.

The last NS hybrid algorithm tested was NEAT-NS. The results from Appendix A Table 45 show that NEAT-NS yields marginally better results with a novelty relevance value of 0.2, with a slight decrease in algorithm performance as the novelty relevance value increases. Overall results demonstrate that NEAT-NS exhibits slightly stronger performance than NEAT without NS on level one (a), and significantly worse on level two (b).

Overall, the best novelty relevance values per algorithm were the following:

- RS-NS – 0.2
- CMA-ES-NS – 0.5
- GA-NS – 0.5
- NEAT-NS – 0.2

4.3 – Final Experiments

To gain deeper insights into the learning capabilities of DL algorithms in the context of stealth games, it is essential to subject them to increasingly complex stealth scenarios and challenges, until they closely resemble those found in commercial stealth video games. The previous section presented tests on levels one (a), two (b), and three (c), each offering distinct challenges, yet all relatively straightforward for most human players. Considering this, this thesis conducted similar tests on level four (d), which represents a significantly more challenging

scenario that could pose initial difficulties even for some human players. As such, the main purpose of this section is to report the results achieved from these tests.

Due to the heightened complexity of level four (d), two adjustments were made to the baseline procedures outlined in section 4.2. Notably, the maximum number of steps per episode was extended from 700 to 1400, and the number of episodes required for a full training session was doubled from 300 to 600.

Although the purpose of the tests conducted in this section was not hyperparameter tuning, multiple combinations of hyperparameters were tested for most algorithms. This approach is more robust and allows for a greater understanding of an algorithm's capabilities to learn level four (d).

4.3.1 – DQN and extensions final experiments

Upon evaluating the first algorithm, DQN, it became evident that it faced challenges in solving level four (d). The findings presented in Appendix A Table 46, reveal that the DQN algorithm, on average, only learned to avoid enemies. However, it's worth noting that there were outlier training sessions where DQN managed to consistently assassinate one enemy or successfully reach the level goal.

The second algorithm under evaluation was Double DQN. The results provided in Appendix A Table 47, show that Double DQN behaves very similarly to the DQN algorithm, although on average Double DQN results were slightly superior.

The third algorithm tested on level four (d) was Duelling DQN. The results obtained by Appendix A Table 48, show that Duelling DQN shows comparable performance to the DQN algorithm. On average it only learned to avoid enemies, with the exception of a few training sessions.

The N-step DQN algorithm was the fourth algorithm tested. The results presented in Appendix A Table 49 reveal that while some hyperparameter combinations exhibited similar performances to the DQN algorithm, others showed better results by consistently reaching the level four (d) goal. Notably, the combination of an ADAM learning rate and decay values of 5e-4 and 1e-5, respectively, demonstrated superior performance compared to all other tested combinations.

PER DQN was the next DQN extension under evaluation. The test results for PER DQN on level four (d), seen in Appendix A Table 50, show that, on average, it exhibits behavior similar to the N-step DQN algorithm, albeit with slightly inferior results.

The sixth algorithm tested was Noisy DQN. The results presented in Appendix A Table 51 show that much like the DQN algorithm, on average Noisy DQN only managed to learn to avoid enemies with overall comparable results.

The seventh algorithm tested was Distributional DQN. The findings, as presented in Appendix A Table 52, indicate that overall performance is superior when utilizing the Tanh activation function compared to the ReLu activation function. While previous algorithms, namely N-step DQN and PER DQN, exhibited some variance in performance according to the activation function used, the discrepancy was not as significant as with Distributional DQN. Specifically, results obtained using the Tanh activation function with Distributional DQN demonstrate that the algorithm was able to consistently reach the goal or assassinate an enemy. Conversely, when the ReLu activation function was used, the algorithm only learned to avoid enemies.

The eighth algorithm tested was Rainbow DQN, which represents the last DQN extension in this evaluation. The results indicate that Rainbow DQN exhibits similar behavior to that of the Distributional DQN algorithm but with better overall performance. Moreover, Rainbow DQN displays a more pronounced variation in results when different activation functions are used, as can be seen in Appendix A Table 53.

4.3.2 – Neuroevolution final experiments

After testing all the implemented DRL algorithms, attention was shifted towards assessing the group of Neuroevolution algorithms on level four.

The initial algorithm evaluated was RS, with the outcomes of these tests presented in Appendix A Table 54. The results indicate that, on average, RS succeeds in reaching the level goal early in training and eventually consistently assassinates an enemy before reaching the goal.

The second Neuroevolution algorithm tested was RS-NS. This algorithm also routinely found the level goal, but it failed to consistently assassinate an enemy before reaching the goal, as can be seen from the gathered results from Appendix A Table 55.

CMA-ES was the third Neuroevolution algorithm to be evaluated on level four (d). The results gathered from Appendix A Table 56 show that this algorithm was only able to learn to avoid enemies and nothing else. Further analysis revealed that CMA-ES would on occasion reach the level goal at the beginning of training, but it did not make much difference further on.

The fourth algorithm evaluated was the CMA-ES-NS algorithm. Much like the RS-NS algorithm, CMA-ES-NS routinely found the level goal, but it failed to consistently assassinate an enemy before reaching the goal, as can be seen from the gathered results from Appendix A Table 57.

GA was the next Neuroevolution algorithm under evaluation. The results indicate that, on average, as seen in Appendix A Table 58, GA succeeds in reaching the level goal early in training and eventually consistently assassinates an enemy before reaching the goal.

Next, the GA-NS algorithm was evaluated. The results provided in Appendix A Table 59, show that GA-NS behaves very similarly to the GA algorithm, with overall stronger population performance but slightly weaker performance for the best individual's average.

The seventh Neuroevolution algorithm tested was the NEAT algorithm. According to Appendix A Table 60, NEAT consistently managed to reach the goal but was inconsistent at assassinating enemies.

The last algorithm tested was NEAT-NS. Appendix A Table 61 shows that the NEAT-NS algorithm had similar results to the RS algorithm.

4.4 – Computational Results

This section presents the results obtained from assessing the computational cost of each implemented algorithm and explains the methodology employed to achieve these results.

Across all implemented algorithms, the component that places the greatest demand on computational resources is the algorithm update loop. While these algorithms employ different methodologies, the update loop serves the same fundamental functions for each of them. These include selecting an action for each agent, performing the chosen actions, storing the resulting data, and updating the algorithm. Notably, there are some key differences between algorithms, such as DQN and its extensions, which update their ANNs with each step, whereas

Neuroevolution algorithms only update their ANNs at the conclusion of an episode. Regardless, since the update loop represents the algorithms' most substantial computational requirement, a fair and effective methodology was devised to measure it.

Because many videogame-related functionality also costs computational resources, a new basic level was created. This level removed all possible physics objects, and enemies, and placed the goal in an unreachable position. This way an algorithm's computational performance would be less affected by external factors.

For all experiments performed in this section, each algorithm was executed with its optimal hyperparameter configurations determined during the tuning experiments, with a few differences. The Neuroevolution algorithms were evaluated with a uniform ANN architecture consisting of three layers, each with 128 neurons for every hidden layer, as this is the most frequent structure employed by DQN and its extensions. However, NEAT and NEAT-NS, which lack fixed topologies, were exceptions to this rule. Additionally, all Neuroevolution algorithms operated with a consistent population size of 50, as this was the most frequently employed value during the hyperparameter tuning processes.

Finally, the tests measure the average Update time performed in 15,000 total loops (150 episodes with 100 steps each).

Table 8 presents the findings derived from these experiments. Notably, it reveals that the implemented Neuroevolution algorithms demand fewer computational resources when compared to the implemented DRL algorithms, with Rainbow DQN registering the slowest execution time and NEAT performing the fastest.

Table 8

Average time in milliseconds for a single run of the Update function for each algorithm. These results were achieved on a CPU 'AMD Ryzen 9 7900x', and GPU 'NVIDIA GeForce RTX 3070'

Algorithm	Average Update Time ms
DQN	4.78
Double DQN	5.27
Dueling DQN	6.06
N-step DQN	4.71
PER DQN	4.85
Noisy DQN	5.16
Distributional DQN	5.81
Rainbow DQN	6.94
RS	3.88
CMA-ES	3.86
GA	3.94
NEAT	0.07
RS-NS	3.88
CMA-ES-NS	3.83
GA-NS	4.04
NEAT-NS	0.08

It is worth noting that while considerable attention was devoted to optimizing these algorithms for computational efficiency, certain algorithms, specifically GA, GA-NS, NEAT, NEAT-NS, and Rainbow DQN, could have benefited from additional improvements. However, given the time constraints of this project, it was not possible to make further enhancements. The mentioned Neuroevolution algorithms might have benefitted from a crossover method that leveraged CPU multithreading, as the current implementation relies on a single CPU thread for this process. Additionally, while the Rainbow DQN implementation benefits from optimizations made in DQN and extensions, further analysis could be conducted to optimize it even further. With these considerations in mind, the performance of these algorithms could be improved.

5 – Results Analysis and Discussion

This chapter's main purpose is to make sense of the results gathered in Chapter 4 and relate them to the proposed objectives described in Chapter 1.

5.1 – **Create a suitable learning stealth game environments**

The first objective of this thesis was to create several diverse stealth game environments/levels that could serve as a useful testbed for training DL algorithms. For this, four stealth levels were created that required different skills to beat. For these levels to be trained on by DL models a set of uniform features that represented any level state were designed. Also, for DL models to be able to learn the created levels a reward structure was created. The results obtained in Chapter 4 indicate that for the most part, this objective was successful. While utilizing their best-found hyperparameter combinations all algorithms were able to learn levels one (a) and two (b), while levels three (c) and four (d) had more nuanced results.

All algorithms were able to learn level one (a), by successfully avoiding the enemy and reaching the goal, notably only a few algorithms consistently solved for the biggest reward by consistently assassinating the enemy before reaching the goal.

Level two (b) was also solved by all algorithms, that were able to explore enough of the maze and reach the goal without a reward structure that indicated how close an agent was to the goal.

Level three (c) was solved by most DRL algorithms by being able to generalize searching and achieve stochastic control, the only DRL algorithm that failed in this task was the Noisy DQN algorithm with its best training session only reaching an average reward of 11.5. Unfortunately, the training methods utilized by Neuroevolution are not well suited to learning stochastic environments, demonstrating very poor results, as shown by the results of Appendix A Table 32. Applying a training methodology that incentivizes generalization, allows the Neuroevolution algorithms to achieve very high scores. Despite this, they were not able to generalize the goal's position, instead, they learned that the goal always spawns near the edges of the map, so they developed a behavior where they constantly circle around the level in the hope of finding the goal.

Level four (d) proved more challenging for most DRL algorithms while being relatively straightforward for the Neuroevolution algorithms. Most DRL algorithms struggled to able to consistently reach the goal on level four (d), the only DRL algorithms that averaged positive results were N-step DQN, Distributional DQN, and Rainbow DQN. Despite this, all DRL algorithms always had at least one training session where they managed to average positive results by consistently reaching the goal. In the case of Neuroevolution algorithms, only CMA-ES was unable to produce good results, analyzing this further suggests that the current reward structure might not allow CMA-ES to successfully explore the level, as the algorithm can occasionally reach the goal early in training, but forgoes going there instead preferring avoiding all enemies. This is more evident when looking at the results from CMA-ES-NS, which are mostly positive, reinforcing the notion that the current reward structure does not allow CMA-ES to learn level four (d).

Overall, results indicate that the environment developed serves as a successful testbed for DL algorithms, but with a few shortcomings, indicating that further improvements can be made and should be considered.

5.2 – Create a suitable methodology for optimizing DL algorithms

The second objective of this thesis was to devise a successful methodology to optimize DL algorithms for the development of game levels. As Chapter four reports on the hyperparameter tunning sections, all algorithms were able to record satisfactory performance on the tested levels, one (a), two (b), and three (c). Utilizing the best-found hyperparameter combinations on level four (d) had mixed results, as explained in the previous section. This indicates that even if a certain hyperparameter combination works well on levels one (a), two (b), and three (c), there might still be the need to perform hyperparameter tuning, as that combination might perform poorly on other levels. Despite this, the best hyperparameter combinations found for the Neuroevolution algorithms mostly performed successfully on all levels.

The results from this hyperparameter tuning process showed that all implemented DRL algorithms were sensitive to parameter adjustments. Although some parameters such as batch size, and target network copy period were less susceptible to changes, most other parameters provided considerably different results even with minor modifications. Notably, variations in

the ADAM optimizer learning rate and decay values led to vastly different outcomes for the implemented DRL algorithms.

In addition to these findings, it was observed that Neuroevolution algorithms behaved very differently than DRL algorithms in this context, being especially robust to hyperparameter changes. Chapter Four's results demonstrate that minor adjustments to Neuroevolution algorithms' hyperparameters have little to no effect on their performance, except for the ADAM optimizer's learning rate and decay values used by CMA-ES and CMA-ES-NS. These outcomes suggest that DQN-based algorithms may require a more robust hyperparameter tuning approach to fully understand their problem-solving capabilities, while Neuroevolution algorithms appear to offer a good understanding of their potential even with minimal hyperparameter tuning.

It is worth noting that the methodology used for hyperparameter tuning was very laborious and required a large amount of time to perform, as it roughly took three months on a CPU ‘AMD Ryzen 9 7900x’, and GPU ‘NVIDIA GeForce RTX 3070’ to conduct all the reported tests.

While the methodology employed did find several useful hyperparameter combinations for every algorithm, the success of this approach is debatable, given the time requirements and, in some cases, the lack of transferability to other levels.

5.3 – Final Algorithms Analyses

For this thesis, sixteen different algorithms were created, as mentioned before these algorithms can be separated into two different families namely DRL and Neuroevolution.

For the most part, the DRL algorithms managed to consistently learn levels one (a), two (b), and three (c). The following is a list of DRL algorithms that provided the best overall performance on the previously mentioned levels:

1º Rainbow DQN
2º N-step DQN
3º Double DQN
4º Distributional DQN
5º DQN
6º Dueling DQN
7º PER DQN

8º Noisy DQN

The order of the previous list changes when evaluating the results on level four (d) becoming:

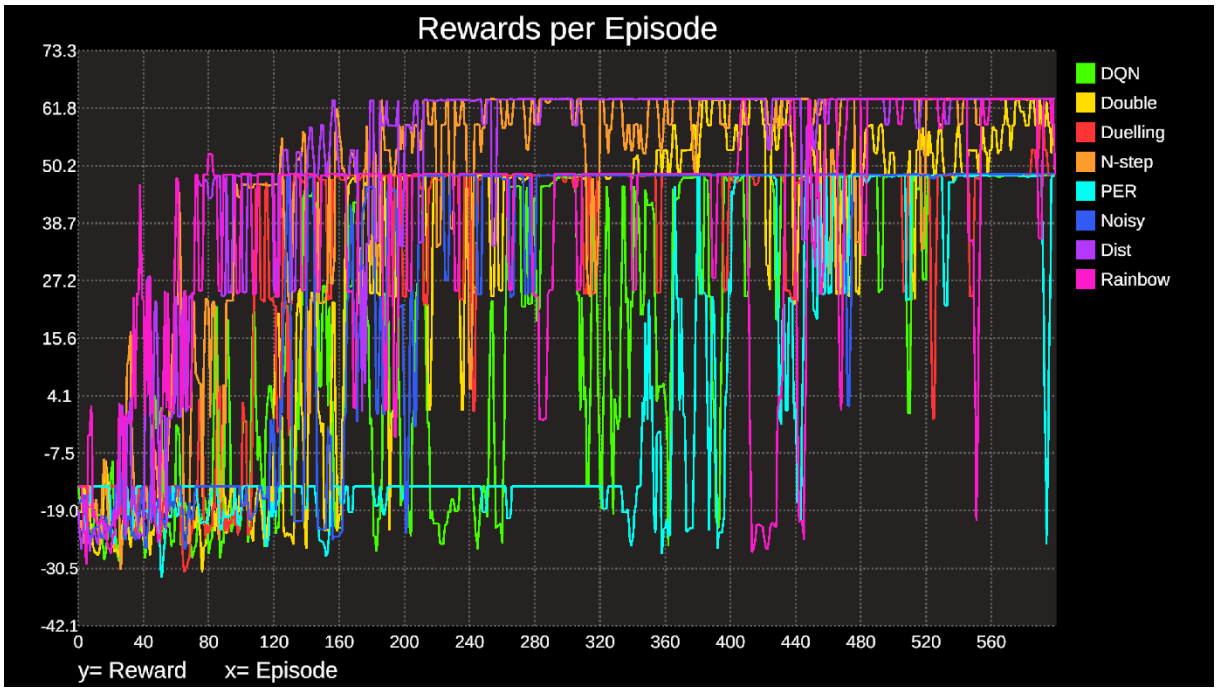
1º Rainbow DQN
2º Distributional DQN
3º N-step DQN
4º PER DQN
5º Dueling DQN
6º Double DQN
7º DQN
8º Noisy DQN

Figure 21 shows the best training session level four (d) for each of the implemented DRL algorithms.

Results demonstrate that Rainbow DQN consistently yields the best performance, which is consistent with current research (Hessel et al., 2018), while the Noisy DQN algorithm is consistently the worst algorithm, which is not consistent with available research (Fortunato et al., 2017). It is worth noting that the N-step DQN algorithm consistently provided some of the best results from the DRL family, which is somewhat surprising as it is the least explored DRL algorithm from this list in the research field, and it was also the DRL algorithm with the fastest update loop.

Figure 21

Rewards over each episode for DQN and its extensions algorithms on level four (d). This graphic represents a smoothed version using the rolling average window size of three. The detailed version of this results that includes all algorithm information can be found at: https://github.com/rorix14/Thesis-Project/tree/main/Graphs



The algorithms from the Neuroevolution family consistently performed well in all levels, including level three (c) when a few changes to the training methodology were applied. Overall, the best performers on learn levels one (a), two (b), and three (c) were:

1º NEAT

2º GA-NS

3º GA

4º RS-NS

5º NEAT-NS

6º CMA-ES-NS

7º CMA-ES

8º RS

The order of the previous list changes when evaluating the results on level four (d) becoming:

1º GA

2º GA-NS

3º NEAT-NS

4º RS

5º NEAT

6º RS-NS

7º CMA-ES-NS

8º CMA-ES

Figure 22 shows the best training session level four (d) for each of the implemented Neuroevolution algorithms.

Analyzing the two lists shows that the GA and GA-NS algorithms consistently yield the best results. Further analysis shows that the gradient-dependent algorithms (CMA-ES and CMA-ES-NS) tend to have worse performance than the algorithms that only rely on natural selection mechanisms.

Figure 22

Best individual fitness over each generation for all Neuroevolution algorithms on level four (d). This graphic represents a smoothed version using the rolling average window size of three. The detailed version of this results that includes all algorithm information can be found at: https://github.com/rorix14/Thesis-Project/tree/main/Graphs

Analyzing both algorithms' families against each other, it is clear that Neuroevolution algorithms exhibit, on average, stronger performance than the DRL algorithms on all the implemented levels, with the exception of level three (c). Not only do they exhibit stronger results they are also computationally less demanding, provided the used environment can make use of multithreading operations.

While it seems that Neuroevolution algorithms have the best performance results in this comparison, it is worth noting that they were not able to truly generalize for level three (c) and most DRL algorithms did have success in that task. It is also important to point out that when observing these algorithms during training, particularly on levels one (a) and four (d), the highest performing DRL algorithms seem to actually learn the level's mechanics, while the Neuroevolution algorithms seem to be mostly brute forcing a solution by trying to multiple different actions and seeing what works best.

6 – Conclusion

This thesis sought to investigate the potential applicability of DL algorithms in game development, with a particular focus on automated testing in stealth games. To evaluate this possibility, different approaches were developed, namely the creation of diverse stealth game environments, the implementation and optimization of state-of-the-art DL algorithms for learning these environments, a comparison of the implemented algorithms' performance, and an analysis of their computational costs.

To develop the different stealth game levels, four different environments, each with distinct but common stealth game problems, were designed to serve as testbeds for evaluating the performance of DL models. These environments were successfully used to train and test the implemented DL algorithms, providing insights into their capabilities in solving the given stealth game challenges. The methodology used for representing the states of these environments allowed most implemented algorithms to solve the challenges presented, indicating their understanding of the environments' dynamics. However, a few exceptions surfaced, indicating that there is still room for improvement in the representation of stealth environments' states utilized in this research. Despite these exceptions and the limited scope of this research compared to the overall existing gaming genres, the overall positive results in solving different stealth game problems suggest that the implemented DL algorithms are capable of learning and finding solutions to different kinds of environments, possibly in other game genres as well.

This project also tried to establish a methodology for optimizing state-of-the-art DL algorithms through a hyperparameter tuning methodology. By doing so, this research identified the differences in hyperparameter tuning for each algorithm and demonstrated the impact of individual hyperparameters on an algorithm's overall performance. This tuning process showed the potential and effectiveness of each implemented DL algorithm in learning the different stealth game environments. However, due to time and computational resource limitations, the hyperparameter tuning methodology used could not be fully explored, with some DL algorithms possibly still benefiting from further tuning experiments. Nevertheless, the results gathered provide valuable insights for future development in the context of hyperparameter tuning.

The comparative analysis of implemented state-of-the-art DL algorithms done in the results and discussion chapters highlighted the strengths and weaknesses of these algorithms when applied

to the task of learning the created stealth game environments, giving insight into their differences in the ability to solve different stealth game common tasks. Furthermore, this analysis also compared the overall performance of each algorithm within their respective family (DRL and Neuroevolution), identifying the Rainbow-DQN algorithm as the top-performing algorithm within the DRL family and GA as the best-performer within the Neuroevolution family. This information also contributes to the ongoing research of testing and comparison of the utilized DL algorithms (Hausknecht et al., 2014; Hessel et al., 2018; Mnih et al., 2015; Risi & Togelius, 2017; Such et al., 2018).

In addition, the computational costs of the implemented DL algorithms were evaluated in the context of game development. Although the scale of these experiments was limited, they effectively assessed the performance of the algorithms' update loops, a crucial aspect of integrating DL algorithms into game development. The results from these experiments revealed that algorithms from the Neuroevolution family generally outperformed those from the DRL family in terms of update loop completion time, with NEAT being the fastest and Rainbow-DQN the slowest. Additionally, sharing this thesis project through a publicly accessible repository provides the research community with an opportunity to examine the optimization methodologies employed to achieve these computational results.

While the results achieved in this thesis successfully fulfill its proposed objectives, they fall short of providing concrete answers on how DL algorithms can be effectively integrated into game development. To better comprehend the potential of DL in game development, future research needs to build upon the findings presented here.

Although the stealth environments implemented offer different challenges, the variety presented in this project is very small when compared to the realm of possible stealth game scenarios. For this reason, possible future work could involve creating more unique stealth challenges and subsequently training the DL algorithms implemented in this project in those environments. Likewise, as mentioned previously, some algorithms could still benefit from further optimizations to reduce computational costs.

Other possible future research could focus on exploring other widely used DL techniques not employed in this project. For instance, while this research explored value-based DRL algorithms, it did not implement any policy-based DRL algorithms that have been proven to have state-of-the-art performance, even surpassing value-based DRL algorithms on some problems.

Another possibility for future work could be to explore different ANN structures. Recurrent neural networks and long short-term memory networks have proven to have strong results when applied to some of the algorithms presented in this thesis.

Nevertheless, the results presented in this thesis still make progress towards understanding the potential applicability of DL in game development, particularly in the context of stealth games. The insights gained from this project also contribute to a better overall understanding of the possibilities that DL techniques may offer to the gaming industry. As challenges and opportunities continue to evolve in this dynamic field, this work serves as a step towards the integration of DL techniques into game development practices, creating opportunities for more engaging and immersive gaming experiences in the future.

Bibliography

A Gentle Introduction to the Rectified Linear Unit (ReLU) - MachineLearningMastery.com. (n.d.). Retrieved October 5, 2023, from https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/

Aggarwal, C. (2018). Neural Networks and Deep Learning: A Textbook. In *Neural Networks and Deep Learning* (1st ed.). Springer.

AI winter - Integrasi Komputer | Wiki eduNitas.com. (n.d.). Retrieved October 5, 2023, from https://wiki.edunitas.com/IT/114-10/AI-winter_5401_eduNitas.html#cite_note-W-1

Bauer, B. (2018). *Common Stealth Level Design Mistakes - Bauer Design Solutions - YouTube*. Youtube. https://www.youtube.com/watch?v=ysXTQgHP-NY&ab_channel=BauerDesignSolutions

Bellemare, M. G., Dabney, W., & Munos, R. (2017). A Distributional Perspective on Reinforcement Learning. *Proceedings of the 34th International Conference on Machine Learning*, *70*, 449–458. https://proceedings.mlr.press/v70/bellemare17a.html

Bottou, L. (2012). Stochastic gradient descent tricks. *Lecture Notes in Computer Science*, *7700*, 421–436. https://doi.org/10.1007/978-3-642-35289-8_25

Brown, B., & Zai, A. (2020). Deep Reinforcement Learning in Action. In *Deep Reinforcement Learning in Action* (First). Manning Publications.

Buckland, M. (2002). *AI Techniques for Game Programming*. Cengage Learning PTR.

Choi, D., Shallue, C. J., Nado, Z., Lee, J., Maddison, C. J., & Dahl, G. E. (2020). On Empirical Comparisons of Optimizers for Deep Learning. *ArXiv*, *abs/1910.05446*. https://www.tensorflow.org/

Chollet, F. (2021). Deep Learning with Python. In *Deep Learning with Python* (Second). Manning Publications.

Cieślak, K. (2022). *How to design stealth games? - Try Evidence*. Try_evidence. https://tryevidence.com/blog/how-to-design-stealth-games/

Croak, M., & Dean, J. (2021). *A decade in deep learning, and what's next*. The Keyword. https://blog.google/technology/ai/decade-deep-learning-and-whats-next/

Cuccu, G., & Gomez, F. (2011). When novelty is not enough. *Lecture Notes in Computer Science*, *6624*, 234–243. https://doi.org/10.1007/978-3-642-20525-5_24/COVER

de Mendonça, M. R. F., Soares Bernardino, H., & Fonseca Neto, R. (2016). Evolution of Reward Functions for Reinforcement Learning applied to Stealth Games. In *UNIVERSIDADE FEDERAL DE JUIZ DE FORA*. Universidade Federal de Juiz de Fora.

Dobrovsky, A., Borghoff, U. M., & Hofmann, M. (2017). Applying and augmenting deep reinforcement learning in serious games through interaction. *Periodica Polytechnica*

Electrical Engineering and Computer Science, *61*(2), 198–208. https://doi.org/10.3311/PPee.10313

Dowling, A. (2022). Pathfinding in Random Partially Observable Environments with Vision-Informed Deep Reinforcement Learning. *ArXiv*, *abs/2209.04801*. http://arxiv.org/abs/2209.04801

Elsayed, S. M., Sarker, R. A., & Essam, D. L. (2010). A comparative study of different variants of genetic algorithms for constrained optimization. *Lecture Notes in Computer Science*, *6457*, 177–186. https://doi.org/10.1007/978-3-642-17298-4_18

Elster, A. (n.d.). *Deep Learning (DL) - Questions and Answers in MRI*. Retrieved October 5, 2023, from https://mriquestions.com/what-is-a-neural-network.html

Enyinna Nwankpa, C., Ijomah, W., Gachagan, A., & Marshall, S. (2021). Activation Functions: Comparison of trends in Practice and Research for Deep Learning. *2nd International Conference on Computational Sciences and Technology*, , 124–133. https://arxiv.org/abs/1811.03378v1

Fortunato, M., Gheshlaghi Azar, M., Piot, B., Menick, J., Hessel, M., Osband, I., Graves, A., Mnih, V., Munos, R., Hassabis, D., Pietquin, O., Blundell, C., & Legg, S. (2017). NOISY NETWORKS FOR EXPLORATION. *ArXiv*, *abs/1706.10295*.

Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. *International Conference on Artificial Intelligence and Statistics*, *9*, 249–256. https://proceedings.mlr.press/v9/glorot10a.html

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. The MIT Press.

Gordillo, C., Bergdahl, J., Tollmar, K., & Gisslén, L. (2021). Improving Playtesting Coverage via Curiosity Driven Reinforcement Learning Agents. *2021 IEEE Conference on Games (CoG)*, 1–8. https://doi.org/10.1109/CoG52621.2021.9619048

Gutiérrez-Sánchez, P., Gómez-Martín, M. A., González-Calero, P. A., & Gómez-Martín, P. P. (2021). Reinforcement Learning Methods to Evaluate the Impact of AI Changes in Game Design. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, *17*(1), 10–17. https://doi.org/https://doi.org/10.1609/aiide.v17i1.18885

Gymnasium Documentation. (2023). Farama Foundation. https://gymnasium.farama.org/

Hammel, B. (n.d.). *What learning rate should I use?* Retrieved October 5, 2023, from http://www.bdhammel.com/learning-rates/

Hausknecht, M., Lehman, J., Miikkulainen, R., & Stone, P. (2014). A neuroevolution approach to general atari game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, *6*(4), 355–366. https://doi.org/10.1109/TCIAIG.2013.2294713

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *2015 IEEE International Conference on Computer Vision (ICCV)*, 1026–1034. https://doi.org/10.1109/ICCV.2015.123

Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., & Silver, D. (2018). Rainbow: Combining Improvements in Deep Reinforcement Learning. *The Thirty-Second AAAI Conference on Artificial Intelligence* , 3215–3222. www.aaai.org

Hinton, G., Srivastava, N., & Swersky, K. (2020). *Neural Networks for Machine Learning, Lecture 6a Overview of Mini-Batch Gradient Descent.*

Huber, T., Mertes, S., Rangelova, S., Flutura, S., & Andre, E. (2021). Dynamic Difficulty Adjustment in Virtual Reality Exergames through Experience-driven Procedural Content Generation. *Proceedings of the 2021 IEEE Symposium Series on ComputationalIntelligence (SSCI)*, 1–8. https://doi.org/10.1109/SSCI50451.2021.9660086

Hyperbolic tangent - MATLAB tanh. (n.d.). The MathWorks. Retrieved October 5, 2023, from https://www.mathworks.com/help/matlab/ref/tanh.html

Igiri, C., Uzoma, A., & Silas, A. (2015). Effect of Learning Rate on Artificial Neural Network in Machine Learning. *International Journal of Engineering Research & Technology*, *4*(2). www.ijert.org

Jaderberg, M., Czarnecki, W. M., Dunning, I., Marris, L., Lever, G., Castañeda, A. G., Beattie, C., Rabinowitz, N. C., Morcos, A. S., Ruderman, A., Sonnerat, N., Green, T., Deason, L., Leibo, J. Z., Silver, D., Hassabis, D., Kavukcuoglu, K., & Graepel, T. (2019). Human-level performance in 3D multiplayer games with population-based reinforcement learning. *Science*, *364*, 859–865. https://doi.org/10.1126/science.aau6249

Jenkins, A., Gupta, V., Myrick, A., & Lenoir, M. (2019). Variations of Genetic Algorithms. *ArXiv*, *abs/1911.00490*. http://arxiv.org/abs/1911.00490

Justesen, N., Bontrager, P., Togelius, J., & Risi, S. (2017). Deep Learning for Video Game Playing. *IEEE Transactions on Games*, *12*, 1–20. http://arxiv.org/abs/1708.07902

Katoch, S., Chauhan, S., & Kumar, V. (2021). A review on genetic algorithm: past, present, and future. *Multimedia Tools and Applications*, *80*, 8091–8126. https://doi.org/10.1007/s11042-020-10139-6

Kim, J. H., & Wu, R. (2021). *Leveraging Machine Learning for Game Development – Google AI Blog*. Google Research. https://ai.googleblog.com/2021/03/leveraging-machine-learning-for-game.html

Kingma, D. P., & Lei Ba, J. (2014). ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION. *CoRR*, *abs/1412.6980*. https://doi.org/https://doi.org/10.48550/arXiv.1412.6980

Kinsley, H., & Kukieła, D. (2020). *Neural Networks From Scratch in Python*. Sentdex.

Lanham, M. (2020). *Hands-On Reinforcement Learning for Games: Implementing self-learning agents in games using artificial intelligence techniques* . Packt Publishing.

Lapan, M. (2020). *Deep Reinforcement Learning Hands-On* (Second). Packt Publishing.

Lehman, J., & Stanley, K. O. (2010). Revising the Evolutionary Computation Abstraction: Minimal Criteria Novelty Search. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation: Vol. GECCO '10* (pp. 103–110). Association for Computing Machinery.

Lehman, J., & Stanley, K. O. (2011). Abandoning Objectives: Evolution through the Search for Novelty Alone. *Evolutionary Computation Journal*, *19*(2), 189–223.

Lonza, A. (2019). *Reinforcement Learning Algorithms with Python: Learn, understand, and develop smart algorithms for addressing AI challenges*. Packt Publishing.

López, C. E., Cunningham, J., Ashour, O., & Tucker, C. S. (2020). Deep reinforcement learning for procedural content generation of 3D virtual environments. *Journal of Computing and Information Science in Engineering*, *20*(5), 1–33. https://doi.org/10.1115/1.4046293/1074423

Marsili-Libelli, S., & Alba, P. (2000). Adaptive mutation in genetic algorithms. *Soft Computing*, *4*, 76–80. https://doi.org/10.1007/s005000000042

Marta, D. L. (2016). *Deep Learning Methods for Reinforcement Learning* [Master Thesis]. Instituto Superior Técnico, Universidade de Lisboa.

Mehrpouyan, H., Fridman, L., Mallik, R. K., Amiri, R., Nallanathan, A., & Matolak, D. (2018). A Machine Learning Approach for Power Allocation in HetNets Considering QoS. *2018 IEEE International Conference on Communications (ICC)*, 1–7. https://www.researchgate.net/publication/323867253

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2015). Playing Atari with Deep Reinforcement Learning. *Nature*, *519*, 529–533. https://doi.org/https://doi.org/10.1038/nature14236

Nakamura, K., Derbel, B., Won, K.-J., & Hong, B.-W. (2021). Learning-Rate Annealing Methods for Deep Neural Networks. *Electronics*, *10*(16). https://doi.org/10.3390/electronics10162029

Nielsen, M. (2019). *Neural Networks and Deep Learning*. Determination Press.

Omelianenko, I. (2019). Hands-On Neuroevolution with Python. In *Hands-On Neuroevolution with Python*. Packt Publishing.

Palanisamy, P. (2018). *Hands-On Intelligent Agents with OpenAI Gym: Your guide to developing AI agents using deep reinforcement learning*. Packt Publishing.

Papavasileiou, E., Cornelis, J., & Jansen, B. (2021). A Systematic Literature Review of the Successors of "NeuroEvolution of Augmenting Topologies." *Evolutionary Computation*, *29*(1), 1–73. https://doi.org/10.1162/evco_a_00282

Part 2: Kinds of RL Algorithms — Spinning Up documentation. (n.d.). OpenAI. Retrieved October 5, 2023, from https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html

Reis, S., Reis, L. P., & Lau, N. (2021). Game Adaptation by Using Reinforcement Learning Over Meta Games. *Group Decision and Negotiation*, *30*(2), 321–340. https://doi.org/10.1007/s10726-020-09652-8

Risi, S., & Togelius, J. (2017). Neuroevolution in Games: State of the Art and Open Challenges. *IEEE Transactions on Computational Intelligence and AI in Games*, *9*(1), 25–41. https://doi.org/10.1109/TCIAIG.2015.2494596

Roohi, S., Guckelsberger, C., Relas, A., Heiskanen, H., Takatalo, J., & Hämäläinen, P. (2021). Predicting Game Difficulty and Engagement Using AI Players. *Proceedings of the ACM on Human-Computer Interaction*, *5*(CHI PLAY), 1–17. https://doi.org/10.1145/3474658

Salimans, T., Ho, J., Chen, X., Sidor, S., & Sutskever, I. (2017). Evolution Strategies as a Scalable Alternative to Reinforcement Learning. *ArXiv*, *abs/1703.03864*. http://arxiv.org/abs/1703.03864

Salloum, Z. (2019). *Exploration in Reinforcement Learning*. https://towardsdatascience.com/exploration-in-reinforcement-learning-e59ec7eeaa75

Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2015). PRIORITIZED EXPERIENCE REPLAY. *CoRR*, *abs/1511.05952*.

Segal, E., & Sipper, M. (2022). Adaptive Combination of a Genetic Algorithm and Novelty Search for Deep Neuroevolution. *CoRR*, *abs/2209.03618*. http://arxiv.org/abs/2209.03618

Shao, K., Tang, Z., Zhu, Y., Li, N., & Zhao, D. (2019). A Survey of Deep Reinforcement Learning in Video Games. *ArXiv*, *abs/1912.10944*. http://arxiv.org/abs/1912.10944

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., & Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, *529*, 484–489. https://doi.org/10.1038/nature16961

Simoes, D., Reis, S., Lau, N., & Reis, L. P. (2020). Competitive deep reinforcement learning over a pokémon battling simulator. *2020 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, 40–45. https://doi.org/10.1109/ICARSC49921.2020.9096092

Smith, R. (2006). *GDC Vault - Level Building for Stealth Gameplay*. GDC Vault. https://www.gdcvault.com/play/1013211/Level-Building-for-Stealth

Stanley, K., & Miikkulainen, R. (2002). Evolving Neural Networks through Augmenting Topologies. *Evolutionary Computation*, *10*(2), 99–127.

Stanley, K. O., Clune, J., Lehman, J., & Miikkulainen, R. (2019). Designing neural networks through neuroevolution. *Nature Machine Intelligence*, *1*(1), 24–35. https://doi.org/10.1038/S42256-018-0006-Z

Stooke, A., & Abbeel, P. (2018). Accelerated Methods for Deep Reinforcement Learning. *ArXiv*, *abs/1803.02811*.

Such, F. P., Madhavan, V., Conti, E., Lehman, J., Stanley, K. O., & Clune, J. (2018). Deep Neuroevolution: Genetic Algorithms are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning. *ArXiv*, *abs/1712.06567*. http://bit.ly/

Sutton, R., & Barto, A. (2018). Reinforcement Learning. In *Reinforcement Learning: An Introduction* (Second Edition). Bradford Books.

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, *3*(1), 9–44. https://doi.org/10.1007/BF00115009

Telikani, A., Tahmassebi, A., Banzhaf, W., & Gandomi, A. H. (2022). Evolutionary Machine Learning: A Survey. *ACM Computing Surveys*, *54*(8), 1–35. https://doi.org/10.1145/3467477

Togelius, J., Khalifa, A., Earle, S., Green, M. C., & Soros, L. (2024). Evolutionary Machine Learning and Games. In *Handbook of Evolutionary Machine Learning* (pp. 715–737). https://doi.org/10.1007/978-981-99-3814-8_25

Unity Technologies. (2017). *Unity ML-Agents Toolkit*. Github. https://github.com/Unity-Technologies/ml-agents

Van Hasselt, H., Guez, A., & Silver, D. (2016). Deep Reinforcement Learning with Double Q-Learning. *Oceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, *30*(1), 2094–2100. https://doi.org/10.1609/aaai.v30i1.10295

Vani, S., & Rao, T. V. M. (2019). An experimental approach towards the performance assessment of various optimizers on convolutional neural network. *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI)*, 331–336. https://doi.org/10.1109/ICOEI.2019.8862686

Wang, Q., Ma, Y., Zhao, K., & Tian, Y. (2022). A Comprehensive Survey of Loss Functions in Machine Learning. *Annals of Data Science*, *9*(2), 187–212. https://doi.org/10.1007/S40745-020-00253-5

Wang, Y., Dong, L., & Sun, C. (2020). Cooperative control for multi-player pursuit-evasion games with reinforcement learning. *Neurocomputing*, *412*, 101–114. https://doi.org/10.1016/j.neucom.2020.06.031

Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M., & de Freitas, N. (2016). Dueling Network Architectures for Deep Reinforcement Learning. *International Conference on Machine Learning*, 1995–2003. https://doi.org/10.48550/arXiv.1511.06581

Wirsansky, E. (2020). *Hands-On Genetic Algorithms with Python*. Packt Publishing.

Wong, K. (2014). *Stealth Game Design*. Game Developer. https://www.gamedeveloper.com/design/stealth-game-design

Yannakakis, G. N., & Togelius, J. (2018). *Artificial Intelligence and Games*. Springer International Publishing. https://doi.org/10.1007/978-3-319-63519-4

Ye, D., Chen, G., Zhang, W., Chen, S., Yuan, B., Liu, B., Chen, J., Liu, Z., Qiu, F., Yu, H., Yin, Y., Shi, B., Wang, L., Shi, T., Fu, Q., Yang, W., Huang, L., & Liu, W. (2020, November 25). Towards Playing Full MOBA Games with Deep Reinforcement Learning. *Proceedings of the 34th International Conference on Neural Information Processing Systems*. http://arxiv.org/abs/2011.12692

Zhang, W. J., Yang, G., Lin, Y., Ji, C., & Gupta, M. M. (2018). On Definition of Deep Learning. *2018 World Automation Congress (WAC)*, 1–5. https://doi.org/10.23919/WAC.2018.8430387

Appendix A

Table 9

DQN performance using distances instead of positions. The performance metrics of the DQN algorithm when using distances instead of positions for the player-to-wall array feature set. The hyperparameters used were: batch size 32; gamma 0.99; update target network period 50; network layers 3; hidden neuron size 128; Gaussian (std 0.01) parameter initialization function; LR 0.005; LR decay 0.001.

	Level One		**Level Two**		**Level Three**	
	Reward	Loss	Reward	Loss	Reward	Loss
Tanh	22.31	0.15	6.54	0.08	25.52	0.13
Relu	14.1	0.39	-2.6	0.02	7.78	0.09

Table 10

DQN performance using different parameter initialization methods. The hyperparameters used were: batch size 32; gamma 0.99; update target network period 50; network layers 3; hidden neuron size 128; LR 0.005; LR decay 0.001.

		Level One		**Level Two**		**Level Three**	
		Reward	Loss	Reward	Loss	Reward	Loss
Uniform (std 0.05)	Tanh	22.4	0.12	-3.68	0.02	8.28	0.06
	ReLu	4.71	1.81	6.77	0.01	6.43	0.05
Uniform (std 0.01)	Tanh	25.40	0.14	30.47	0.03	8.76	0.03
	ReLu	25.7	0.13	31.58	0.02	3.99	0.06
Uniform (std 0.005)	Tanh	28.23	0.13	15.73	0.02	13.01	0.07
	ReLu	32.12	0.06	37.47	0.01	3.78	0.07
Uniform (std 0.001)	Tanh	20.91	0.15	27.89	0.05	8.70	0.06
	ReLu	-12.10	0.0	25.69	0.03	0.4	0.04
Gaussian (std 0.05)	Tanh	26.88	0.12	26.79	0.03	15.77	0.09
	ReLu	18.82	2.87	29.9	0.01	4.5	0.04
Gaussian (std 0.01)	Tanh	23.91	0.14	31.1	0.3	8.49	0.07
	ReLu	30.42	0.66	37.15	0.02	2.94	0.06
Gaussian (std 0.005)	Tanh	28.83	0.11	30.83	0.02	8.35	0.06
	ReLu	5.11	0.05	22.44	0.02	3.43	0.07
Gaussian (std 0.001)	Tanh	19.84	0.14	27.01	0.03	10.23	0.07
	ReLu	-1.35	0.1	29.32	0.03	0.89	0.05
Xavier	Tanh	31.06	0.13	20.01	0.03	13.75	0.07
	ReLu	-1.39	2.12	36.22	0.02	4.15	0.06
Xavier Normalized	Tanh	24.32	0.13	1.44	0.02	5.07	0.05
	ReLu	8.35	1.52	17.87	0.01	5.96	0.09
He	Tanh	13.65	0.1	-5.44	0.01	3.44	0.05

	ReLu	2.2	2.89	-5.65	0.01	8.63	0.08

Table 11

DQN performance using different layers and neuron sizes. The hyperparameters used were: batch size 32; gamma 0.99; update target network period 50; Uniform (std 0.005) parameter initialization function; LR 0.005; LR decay 0.001.

Layers	Neurons		**Level One**		**Level Two**		**Level Three**	
			Reward	Loss	Reward	Loss	Reward	Loss
2	32	Tanh	21.5	0.25	21.14	0.13	2.81	0.06
		ReLu	24.68	0.19	9.49	0.04	1.59	0.06
	64	Tanh	28.77	0.17	26.4	0.05	2.4	0.05
		ReLu	28.12	0.19	10.32	0.03	2.73	0.08
	128	Tanh	22.02	0.12	30.19	0.02	4.5	0.07
		ReLu	---	---	17.5	0.03	2.38	0.04
	256	Tanh	30.61	0.09	33.67	0.02	9.8	0.1
		ReLu	30.38	0.12	23.99	0.04	2.52	0.06
	512	Tanh	28.21	0.07	31.86	0.02	7.7	0.07
		ReLu	23.39	0.08	23.38	0.03	4.97	0.09
3	32	Tanh	23.36	0.25	18.66	0.12	3.16	0.06
		ReLu	-5.1	0.12	20.11	0.03	0.52	0.06
	64	Tanh	22.5	0.21	26.39	0.07	9.12	0.08
		ReLu	13.78	0.05	22.24	0.03	2.03	0.07
	128	Tanh	28.23	0.13	15.73	0.02	13.01	0.07
		ReLu	32.12	0.06	37.47	0.01	3.78	0.07
	256	Tanh	19.38	0.1	27.99	0.02	14.56	0.08
		ReLu	30.04	0.19	33.66	0.02	6.51	0.09
	512	Tanh	18.91	0.1	13.79	0.01	10.9	0.07
		ReLu	7.53	4.23	36.93	0.01	7.01	0.06
4	32	Tanh	21.22	0.29	15.3	0.11	5.61	0.06
		ReLu	-12.52	0.06	-6.37	0	1.27	0.04
	64	Tanh	29.01	0.21	24.22	0.07	5.35	0.07
		ReLu	-11.4	0.07	-6.75	0.01	0.97	0.03
	128	Tanh	24.4	0.14	23.55	0.04	6.57	0.07
		ReLu	-10.96	0.06	1.97	0	0.72	0.06
	256	Tanh	25.01	0.12	18.18	0.02	1.38	0.07
		ReLu	-11.93	0.08	34.14	0.01	1.15	0.06
	512	Tanh	14.21	0.14	6.56	0	6.14	0.06
		ReLu	7.17	2.54	36.84	0.01	9.03	0.06

Table 12

DQN performance using different learning rates and decay values for the ADAM optimizer. The hyperparameters used were: batch size 32; gamma 0.99; update target network period 50;

network layers 3; hidden neuron size 128; Uniform (std 0.005) parameter initialization function.

Lr	Decay		Level One		Level Two		Level Three	
			Reward	Loss	Reward	Loss	Reward	Loss
0.1	1e-2	Tanh	2.82	0.27	-6.91	0.02	6.44	0.07
		ReLu	19.4	0.24	-6.47	1.73	3.76	0.05
	1e-3	Tanh	-19.05	NAN	-7	0.09	3.53	0.16
		ReLu	6.64	0.2	-4.91	3444	4.36	0.06
	1e-4	Tanh	-24.12	NAN	-7.02	NAN	-6.62	NAN
		ReLu	5.38	0.34	-6.37	3453	6.39	0.09
0.01	1e-2	Tanh	19.73	0.3	23.89	0.19	2.44	0.06
		ReLu	14.36	0.22	26.27	0.06	2.5	0.08
	1e-3	Tanh	27.91	0.1	28.87	0.02	9.74	0.07
		ReLu	9.11	0.18	37.27	0.02	4.25	0.09
	1e-4	Tanh	-4.85	0.34	-5.6	0.02	2.91	0.08
		ReLu	1.6	0.25	16.45	0.02	1.38	0.05
0.005	1e-2	Tanh	21.86	0.35	25.16	0.23	1.33	0.05
		ReLu	16.18	0.32	10.77	0.08	0.78	0.06
	1e-3	Tanh	28.23	0.13	15.73	0.02	13.01	0.07
		ReLu	32.12	0.06	37.47	0.01	3.78	0.07
	1e-4	Tanh	10.66	0.11	-3.82	0	5.6	0.07
		ReLu	23.82	0.12	18.25	0.01	4.32	0.05
0.001	1e-2	Tanh	12.57	0.4	5.29	0.14	0.3	0.03
		ReLu	1.02	0.31	6.97	0.22	0.26	0.03
	1e-3	Tanh	22.99	0.29	12.89	0.13	1.78	0.05
		ReLu	28.09	0.11	21.06	0.08	0.16	0.05
	1e-4	Tanh	31.29	0.06	25.51	0.02	23	0.08
		ReLu	28.77	0.09	27.95	0.02	6.69	0.08
	1e-5	Tanh	24.7	0.07	0.55	0.01	15.36	0.07
		ReLu	12.4	2.17	30.8	0.01	16.64	0.08
0.0005	1e-2	Tanh	6.25	0.23	-3.78	0.02	0.29	0.03
		ReLu	8.73	0.33	2.79	0.17	1.26	0.06
	1e-3	Tanh	27.7	0.39	14.47	0.24	2.81	0.08
		ReLu	13.94	0.2	22.15	0.1	0.95	0.06
	1e-4	Tanh	33.95	0.11	20.45	0.04	11.04	0.08
		ReLu	30.07	0.13	35.38	0.02	3.85	0.07
	1e-5	Tanh	35.16	0.05	26.07	0.01	27.06	0.06
		ReLu	25.3	0.01	37.1	0.01	12.86	0.07
	1e-6	Tanh	32.78	0.07	2.8	0.01	14.12	0.06
		ReLu	20.94	0.64	38.27	0.01	12.96	0.06
	1e-7	Tanh	36.27	0.06	12.27	0.01	15.59	0.07
		ReLu	18.4	0.6	38.97	0.01	16.02	0.07
	1e-8	Tanh	34.65	0.05	9.8	0.02	19.25	0.07
		ReLu	15.15	0.68	37.81	0.01	17.33	0.08
0.0001	1e-2	Tanh	-2.01	0.3	-3.12	0.03	1.76	0.06
		ReLu	-0.55	0.3	4.51	0.16	1.81	0.04
	1e-3	Tanh	19.39	0.43	3.55	0.14	0.84	0.07
		ReLu	6.9	0.32	6.07	0.23	1.66	0.07

	1e-4	Tanh	22.18	0.32	21.77	0.21	2.67	0.04
		ReLu	16.6	0.12	29.48	0.06	1.74	0.05
	1e-5	Tanh	27.52	0.2	27.3	0.05	8.33	0.09
		ReLu	35.02	0.07	36.31	0.01	6.72	0.08
	1e-6	Tanh	33.34	0.16	32.66	0.05	12.84	0.09
		ReLu	31.87	0.05	36.29	0.01	8.16	0.05
	1e-7	Tanh	30.37	0.16	31.4	0.03	16.57	0.1
		ReLu	37.07	0.05	41.01	0.01	9.42	0.08
	1e-8	Tanh	30.77	0.17	25.19	0.03	13.89	0.09
		ReLu	37.77	0.06	39.19	0.01	7.99	0.07
	1e-9	Tanh	31.38	0.15	20.95	0.04	15.47	0.08
		ReLu	25.59	0.05	37.11	0.01	11.37	0.06
0.00005	1e-3	Tanh	8.57	0.27	-0.93	0.05	0.74	0.06
		ReLu	5.28	0.49	-3.2	0.04	0.17	0.07
	1e-4	Tanh	24.6	0.51	12.42	0.19	0.72	0.04
		ReLu	24.62	0.23	17.92	0.12	0.58	0.04
	1e-5	Tanh	25.02	0.32	20.14	0.13	6.31	0.09
		ReLu	32.58	0.09	30.57	0.05	6.28	0.07
	1e-6	Tanh	29.51	0.29	25.15	0.1	6.21	0.07
		ReLu	25.6	0.08	35.7	0.02	6.99	0.08
	1e-7	Tanh	26.26	0.26	14.70	0.07	9.7	0.1
		ReLu	31.19	0.08	36.11	0.02	5.16	0.07
	1e-8	Tanh	27.29	0.29	20.04	0.09	6.28	0.07
		ReLu	33.8	0.09	37.35	0.01	6.84	0.07
	1e-9	Tanh	29.32	0.26	17.26	0.05	7.51	0.07
		ReLu	28.91	0.07	36.65	0.02	5.41	0.06
0.00001	1e-3	Tanh	-9.12	0.19	-1.88	0.03	1.68	0.06
		ReLu	-12.8	0.25	2.42	0.19	1.3	0.07
	1e-4	Tanh	9.62	0.34	4.73	0.17	1.16	0.06
		ReLu	7.69	0.47	10.54	0.31	1.49	0.07
	1e-5	Tanh	17.12	0.41	11.5	0.18	0.28	0.07
		ReLu	18.88	0.34	13.41	0.16	1.41	0.06
	1e-6	Tanh	18.87	0.43	26.95	0.35	0.78	0.07
		ReLu	21.54	0.27	15.36	0.1	3.33	0.08
	1e-7	Tanh	21.45	0.46	16.16	0.2	1.79	0.07
		ReLu	19.13	0.24	17.96	0.12	2.79	0.08
	1e-8	Tanh	11.32	0.24	17.13	0.24	2.77	0.05
		ReLu	19.92	0.23	13.24	0.09	3.26	0.07
	1e-9	Tanh	17.98	0.39	16.46	0.21	1.43	0.21
		ReLu	21.39	0.23	14.5	0.07	2.65	0.05

Table 13

DQN performance using different target network update/copy period values, with different ADAM learning rate and decay values. The hyperparameters used were: batch size 32; gamma

0.99; network layers 3; hidden neuron size 128; Uniform (std 0.005) parameter initialization function.

copy period	Lr & decay		Level One		Level Two		Level Three	
			Reward	Loss	Reward	Loss	Reward	Loss
25	0.0005;	Tanh	32.13	0.06	23.43	0.01	24.79	0.05
	1e-5	ReLu	18.37	1.4	33.98	0.02	18.30	0.08
	0.0001;	Tanh	34.21	0.18	16.21	0.03	15.68	0.08
	1e-7	ReLu	36.53	0.06	39.25	0.01	11.08	0.09
50	0.0005;	Tanh	35.16	0.05	26.07	0.01	27.06	0.06
	1e-5	ReLu	25.3	0.01	37.1	0.01	12.86	0.07
	0.0001;	Tanh	30.37	0.16	31.4	0.03	16.57	0.1
	1e-7	ReLu	37.07	0.05	41.01	0.01	9.42	0.08
75	0.0005;	Tanh	28.71	0.06	25.42	0.01	24.41	0.06
	1e-5	ReLu	25.21	0.52	37.74	0.01	15.39	0.07
	0.0001;	Tanh	33.62	0.17	22.4	0.04	13.01	0.1
	1e-7	ReLu	29.54	0.05	39.89	0.01	10.55	0.06
100	0.0005;	Tanh	32.21	0.06	9.39	0.01	16.07	0.07
	1e-5	ReLu	22.73	0.23	33.27	0.01	15.54	0.07
	0.0001;	Tanh	35.22	0.15	25.87	0.05	11	0.07
	1e-7	ReLu	33.73	0.05	32.05	0.02	10.69	0.08
125	0.0005;	Tanh	32.68	0.07	2.79	0	14.95	0.07
	1e-5	ReLu	33.31	0.06	34.78	0.01	13.14	0.07
	0.0001;	Tanh	35.11	0.16	19.45	0.04	13.94	0.11
	1e-7	ReLu	21.95	0.04	37.43	0.01	11.13	0.08
150	0.0005;	Tanh	35.25	0.06	6.78	0.01	15.65	0.06
	1e-5	ReLu	30.1	0.1	32.07	0.01	11.6	0.09
	0.0001;	Tanh	32.8	0.14	13.15	0.02	14.19	0.1
	1e-7	ReLu	25.19	0.07	34.53	0.01	11.39	0.07
175	0.0005;	Tanh	27.71	0.05	6.3	0	17.66	0.07
	1e-5	ReLu	25.3	0.08	31.6	0.01	13.35	0.09
	0.0001;	Tanh	36.01	0.16	11.82	0.02	12.89	0.11
	1e-7	ReLu	31.47	0.07	33.38	0.01	10.29	0.08
200	0.0005;	Tanh	31.1	0.06	13.67	0.01	16.31	0.07
	1e-5	ReLu	30.36	0.11	33.53	0.01	10.8	0.1
	0.0001;	Tanh	37.14	0.14	18.54	0.02	11.26	0.1
	1e-7	ReLu	32.99	---	34.64	0.01	11.59	0.08

Table 14

DQN performance using different batch sizes, with different ADAM learning rates and decay values. The hyperparameters used were: gamma 0.99; update target network period 50; network layers 3; hidden neuron size 128; Uniform (std 0.005) parameter initialization function.

Batch Size	Lr & Decay		Level One		Level Two		Level Three	
			Reward	Loss	Reward	Loss	Reward	Loss
16	0.0005;	Tanh	28.73	0.08	20	0.02	19.93	0.07
	1e-5	ReLu	17.89	0.22	32.48	0.02	14.41	0.1
	0.0001;	Tanh	24.26	0.23	22.85	0.05	13.39	0.09
	1e-7	ReLu	30.55	0.07	39.53	0.01	9.22	0.08
32	0.0005;	Tanh	35.16	0.05	26.07	0.01	27.06	0.06
	1e-5	ReLu	25.3	0.01	37.1	0.01	12.86	0.07
	0.0001;	Tanh	30.37	0.16	31.4	0.03	16.57	0.1
	1e-7	ReLu	37.07	0.05	41.01	0.01	9.42	0.08
48	0.0005;	Tanh	37.48	0.04	20.77	0.01	21.7	0.04
	1e-5	ReLu	22.04	0.6	39.31	0.01	15.67	0.07
	0.0001;	Tanh	35.39	0.13	20.14	0.03	13.71	0.09
	1e-7	ReLu	34.59	0.05	38.88	0.02	15.67	0.07
64	0.0005;	Tanh	35.17	0.09	18.59	0.01	18.31	0.05
	1e-5	ReLu	11.86	0.43	37.2	0.01	12.76	0.06
	0.0001;	Tanh	34.17	0.12	19.89	0.02	14.69	0.08
	1e-7	ReLu	36.4	0.05	40.7	0.01	10.02	0.07
80	0.0005;	Tanh	35.03	0.04	24.23	0.01	25.02	0.05
	1e-5	ReLu	5.68	0.04	32.9	0.01	15.88	0.07
	0.0001;	Tanh	40.83	0.1	28.08	0.02	11.5	0.08
	1e-7	ReLu	37.13	0.07	40.41	0	10.04	0.06
96	0.0005;	Tanh	38.33	0.03	21.26	0.01	17.19	0.05
	1e-5	ReLu	-1.2	8.57	34.85	0.41	10.92	0.07
	0.0001;	Tanh	36.41	0.11	28.38	0.02	11.9	0.06
	1e-7	ReLu	36.21	0.04	39.25	0.01	10.9	0.06

Table 15

DQN performance using different gammas values, with different ADAM learning rates and decay values. The hyperparameters used were: batch size 32; update target network period 50; network layers 3; hidden neuron size 128; Uniform (std 0.005) parameter initialization function.

Gamma	Lr & Decay		Level One		Level Two		Level Three	
			Reward	Loss	Reward	Loss	Reward	Loss
0.99	0.0005;	Tanh	35.16	0.05	26.07	0.01	27.06	0.06
	1e-5	ReLu	25.3	0.01	37.1	0.01	12.86	0.07
	0.0001;	Tanh	30.37	0.16	31.4	0.03	16.57	0.1
	1e-7	ReLu	37.07	0.05	41.01	0.01	9.42	0.08
0.98	0.0005;	Tanh	34.53	0.07	14.55	0	20.70	0.04
	1e-5	ReLu	33.82	0.08	37.31	0.01	15.16	0.09
	0.0001;	Tanh	35.4	0.16	10.55	0.03	14.47	0.12
	1e-7	ReLu	33.86	0.09	35.05	0.01	9.25	0.09
0.96	0.0005;	Tanh	29.79	0.07	-5.46	0	12.03	0.04
	1e-5	ReLu	33.41	0.07	17.16	0	10.46	0.1

	0.0001;	Tanh	28.3	0.19	-4.08	0.01	8.86	0.1
	1e-7	ReLu	26.38	0.08	17.35	0.01	8.11	0.09
0.94	0.0005;	Tanh	18.06	0.06	-6.32	0	11.87	0.06
	1e-5	ReLu	22.38	0.15	6.71	0.01	5.54	0.06
	0.0001;	Tanh	18.38	0.16	-5.14	0	7.19	0.1
	1e-7	ReLu	20.86	0.09	-4.31	0	5.75	0.09
0.92	0.0005;	Tanh	10.06	0.05	-6.51	0	9.25	0.06
	1e-5	ReLu	11.83	0.06	-2.24	0.01	5.76	0.07
	0.0001;	Tanh	13.07	0.19	-5.31	0	10.17	0.13
	1e-7	ReLu	10.56	0.1	-4.86	0	4.74	0.09
0.9	0.0005;	Tanh	3.65	0.06	-6.74	0	7.7	0.08
	1e-5	ReLu	0.06	0.08	-4.31	0	5.67	0.07
	0.0001;	Tanh	4.22	0.14	-6.45	0	7.45	0.1
	1e-7	ReLu	-0.97	0.09	-5.64	0	4.62	0.06

Table 16

Double DQN performance, with different ADAM learning rates and decay values. The hyperparameters used were: batch size 32; gamma 0.99; update target network period 50; network layers 3; hidden neuron size 128; Uniform (std 0.005) parameter initialization function.

Lr & decay		**Level One**		**Level Two**		**Level Three**	
		Reward	Loss	Reward	Loss	Reward	Loss
0.0005; 1e-5	Tanh	40.97	0.05	30.22	0.01	24.5	0.06
	ReLu	36.61	0.07	29.11	0.01	17.8	0.1
0.0001; 1e-7	Tanh	37.49	0.18	32.16	0.04	17.95	0.11
	ReLu	38.3	0.06	38.91	0.01	11.8	0.09
0.001; 1e-4	Tanh	38.8	0.04	26.7	0.01	17.57	0.07
	ReLu	31.31	0.17	32.15	0.01	9.09	0.07
0.005; 1e-3	Tanh	38.98	0.06	28.48	0.03	10.45	0.07
	ReLu	11.67	0.1	31.79	0.02	1.75	0.05

Table 17

N-step DQN performance using different n-step values, with different ADAM learning rates and decay values. The hyperparameters used were: batch size 32; gamma 0.99; update target network period 50; network layers 3; hidden neuron size 128; Uniform (std 0.005) parameter initialization function.

N-step	**Lr & Decay**		**Level One**		**Level Two**		**Level Three**	
			Reward	Loss	Reward	Loss	Reward	Loss
2	0.0005;	Tanh	40.05	0.08	38.84	0.01	26.58	0.07
	1e-5	ReLu	36.17	0.11	38.97	0.01	16.43	0.1
		Tanh	36.99	0.24	34.35	0.05	14.15	0.13

	0.0001; 1e-7	ReLu	42.79	0.1	39.22	0.01	14.84	0.13
	0.001;	Tanh	34.86	0.09	34.18	0.03	15.89	0.08
	1e-4	ReLu	32.88	0.38	41.12	0.01	11.48	0.13
	0.005;	Tanh	32.7	0.13	39.75	0.03	19.37	0.12
	1e-3	ReLu	28.77	0.11	36.76	0.03	5.06	0.11
3	0.0005;	Tanh	40.17	0.08	41.36	0.02	29.26	0.09
	1e-5	ReLu	36.94	0.16	40.24	0.01	17.83	0.13
	0.0001;	Tanh	38.96	0.3	36.49	0.06	19.32	0.21
	1e-7	ReLu	38.51	0.11	38.04	0.02	13.32	0.17
	0.001;	Tanh	42.73	0.09	37.62	0.02	26.22	0.09
	1e-4	ReLu	36.42	0.14	41.17	0.02	14.43	0.17
	0.005;	Tanh	42.09	0.09	34.74	0.05	15.72	0.12
	1e-3	ReLu	40.51	0.14	37.07	0.02	2.94	0.14
4	0.0005;	Tanh	42.30	0.11	37.76	0.02	32.06	0.1
	1e-5	ReLu	41.89	0.37	37.03	0.04	18.12	0.17
	0.0001;	Tanh	39.95	0.33	37.37	0.09	16.94	0.22
	1e-7	ReLu	39.84	0.16	40.32	0.02	13	0.21
	0.001;	Tanh	38.52	0.12	35.33	0.03	22.02	0.09
	1e-4	ReLu	25.36	0.95	35.66	0.03	14.49	0.2
	0.005;	Tanh	39.48	0.15	38.78	0.58	20.23	0.17
	1e-3	ReLu	38.17	0.13	30.87	0.03	5.11	0.23
5	0.0005;	Tanh	39.53	0.12	40.4	0.03	34.67	0.14
	1e-5	ReLu	40.94	0.11	39.71	0.03	21.46	0.17
	0.0001;	Tanh	39.37	0.38	34.83	0.07	20.1	0.31
	1e-7	ReLu	43	0.24	38.45	0.04	11.53	0.23
	0.001;	Tanh	40.96	0.13	37.75	0.03	25.68	0.12
	1e-4	ReLu	39.14	0.3	38.75	0.03	11.74	0.2
	0.005;	Tanh	35.48	0.2	40.43	0.05	20.05	0.17
	1e-3	ReLu	33.13	0.22	31.15	0.03	5.95	0.24
6	0.0005;	Tanh	43.01	0.13	39.25	0.02	28.63	0.13
	1e-5	ReLu	44.39	0.17	38.16	0.04	20.21	0.22
	0.0001;	Tanh	42.65	0.55	33.72	0.09	18.56	0.33
	1e-7	ReLu	43.12	0.21	37.28	0.05	16.69	0.39
	0.001;	Tanh	40.25	0.15	39.89	0.05	29.52	0.18
	1e-4	ReLu	44.46	0.23	36.26	0.06	12.14	0.27
	0.005;	Tanh	36.12	0.25	38.5	0.06	21.53	0.16
	1e-3	ReLu	36.83	0.22	26.6	0.05	6.61	0.29
7	0.0005;	Tanh	43.16	0.16	37.9	0.04	31.83	0.17
	1e-5	ReLu	43.1	0.12	35.27	0.07	19.06	0.21
	0.0001;	Tanh	40.73	0.46	37.96	0.1	14.79	0.3
	1e-7	ReLu	42.61	0.26	38.66	0.05	16.45	0.39
	0.001;	Tanh	34.65	0.15	35.12	0.06	24.15	0.16
	1e-4	ReLu	39.41	0.53	34.64	0.05	14.63	0.3
	0.005;	Tanh	37.92	0.24	37.7	0.06	22.18	0.23
	1e-3	ReLu	37.86	0.31	23.69	0.06	8.33	0.34
8	0.0005;	Tanh	39.46	0.19	37.93	0.04	29.07	0.18
	1e-5	ReLu	42.75	0.14	37.77	0.05	24.79	0.27
	0.0001;	Tanh	40.52	0.59	33.97	0.11	16.02	0.36
	1e-7	ReLu	35.26	0.28	27.71	0.05	15.37	0.43

	0.001;	Tanh	35.04	0.24	36.88	0.05	28.5	0.19
	1e-4	ReLu	44.43	0.21	31.59	0.05	15.5	0.39
	0.005;	Tanh	40.43	0.2	41.2	0.06	19.33	0.23
	1e-3	ReLu	34.68	0.2	34.69	0.66	7.87	0.42
9	0.0005;	Tanh	43.89	0.21	36.92	0.06	30.02	0.17
	1e-5	ReLu	39.22	0.19	35.42	0.06	21.68	0.32
	0.0001;	Tanh	42.14	0.54	38.98	0.12	16.2	0.33
	1e-7	ReLu	39.72	0.33	35.17	0.06	15.13	0.41
	0.001;	Tanh	40.09	0.19	33.53	0.07	20.98	0.19
	1e-4	ReLu	36.58	---	35.86	0.06	16.73	0.35
	0.005;	Tanh	41.02	0.23	32.05	0.09	20.7	0.26
	1e-3	ReLu	44.21	0.21	28.5	0.07	6.83	0.36
10	0.0005;	Tanh	39.8	0.16	36.98	0.06	28.94	0.21
	1e-5	ReLu	37.52	0.18	34.79	0.07	18.27	0.27
	0.0001;	Tanh	42.02	0.73	36.27	0.15	13.18	0.34
	1e-7	ReLu	38.42	0.36	35.63	0.07	15.2	0.46
	0.001;	Tanh	37.61	0.22	39.59	0.07	23.06	0.22
	1e-4	ReLu	38.23	0.3	35.82	0.08	16.33	0.36
	0.005;	Tanh	39.27	0.3	23.57	0.17	21.56	0.2
	1e-3	ReLu	41.72	0.33	18.56	0.04	6.15	0.33

Table 18

Noisy DQN performance using different sigma values, with different ADAM learning rates and decay values. The hyperparameters used were: batch size 32; gamma 0.99; update target network period 50; network layers 3, the first layer is linear while the other two are noisy; hidden neuron size 128; Uniform (std 0.005) parameter initialization function for the first layer linear layer, and Xavier initialization function for the noisy layers, as per original implementation (Fortunato et al., 2019).

Sigma	**Lr & Decay**		**Level One**		**Level Two**		**Level Three**	
			Reward	Loss	Reward	Loss	Reward	Loss
0.5	0.0005;	Tanh	0.32	0.1	-1.12	0	2.55	0.06
	1e-5	ReLu	16.51	0.2	7.55	0.12	1.64	0.05
	0.0001;	Tanh	18.57	0.13	-4.94	0	5.47	0.07
	1e-7	ReLu	20.78	0.09	14.51	0.02	0.29	0.05
	0.001;	Tanh	7.71	0.08	-3.55	0	5.92	0.07
	1e-4	ReLu	25.85	0.23	19	0.08	3.9	0.06
	0.005;	Tanh	6.83	0.14	-6.89	0	4.45	0.06
	1e-3	ReLu	10.58	0.32	3.02	0.05	2.7	0.08
0.1	0.0005;	Tanh	12.82	0.09	-7.02	0	5.13	0.06
	1e-5	ReLu	35.97	0.09	11.56	0.07	3.27	0.05
	0.0001;	Tanh	23.01	0.11	4	0.01	7.79	0.07
	1e-7	ReLu	32.12	0.07	23.95	0.01	1.86	0.05
	0.001;	Tanh	16.34	0.07	-4.65	0	7.92	0.06
	1e-4	ReLu	6.43	0.19	17.78	0.03	3.72	0.04

	0.005;	Tanh	-0.41	0.11	-2.66	0	3.78	0.07
	1e-3	ReLu	15.21	0.14	8.42	0.13	1.69	0.06
0.05	0.0005;	Tanh	13.46	0.08	-7.02	0	8.14	0
	1e-5	ReLu	35.56	0.01	7.29	0.03	4.39	0.05
	0.0001;	Tanh	31.47	0.09	4.58	0.02	4.86	0.06
	1e-7	ReLu	30.59	0.04	30.78	0.02	4.4	0.06
	0.001;	Tanh	20.27	0.11	-5.31	0	5.92	0.05
	1e-4	ReLu	21.49	0.09	20.04	0.04	2.23	0.06
	0.005;	Tanh	15.67	0.17	-7	0	3.03	0.07
	1e-3	ReLu	13.5	0.4	23.61	0.03	2.45	0.06
0.01	0.0005;	Tanh	20.9	0.06	-7.02	0	6.71	0.06
	1e-5	ReLu	16.58	0.09	17.36	0.09	5.25	0.06
	0.0001;	Tanh	25.72	0.09	-1.91	0.01	6.4	0.06
	1e-7	ReLu	36.3	0.05	28.44	0.02	4.95	0.05
	0.001;	Tanh	23.33	0.08	-6.7	0	5.04	0.05
	1e-4	ReLu	30.98	0.12	19.24	0.07	3.03	0.06
	0.005;	Tanh	18.57	0.15	-7	0	2.92	0.05
	1e-3	ReLu	28.4	0.12	10.59	0.13	2.26	0.06
0.005	0.0005;	Tanh	16.53	0.08	6.07	0	6.32	0.06
	1e-5	ReLu	18.17	0.08	12.43	0.07	5.69	0.06
	0.0001;	Tanh	28.3	0.09	6.32	0.02	5.79	0.05
	1e-7	ReLu	19.7	0.04	27.15	0.02	3.36	0.05
	0.001;	Tanh	10.05	0.07	-7	0	6.87	0.04
	1e-4	ReLu	24.35	0.05	14.05	0.02	1.52	0.04
	0.005;	Tanh	13.78	0.11	-0.74	0.01	4.19	0.07
	1e-3	ReLu	22.58	0.22	13.83	0.16	2.3	0.04
0.001	0.0005;	Tanh	13.44	0.07	-7	0	6.12	0.08
	1e-5	ReLu	15.75	0.11	16.44	0.05	6.03	0.07
	0.0001;	Tanh	19.73	0.08	-6.44	0	6.16	0.07
	1e-7	ReLu	37.56	0.05	30.97	0.01	3.35	0.07
	0.001;	Tanh	18.99	0.06	-6.67	0	5.76	0.04
	1e-4	ReLu	30.39	0.13	20.94	0.04	4.05	0.08
	0.005;	Tanh	18.32	0.14	-6.98	0	3.65	0.06
	1e-3	ReLu	17.81	0.09	26.88	0.04	2.21	0.1

Table 19

Dueling DQN performance, with different ADAM learning rates and decay values. The hyperparameters used were: batch size 32; gamma 0.99; update target network period 50; 1 input layer, 2 Value network layers, 2 Advantage network Layers; hidden neuron size 128; Uniform (std 0.005) parameter initialization function.

Lr & decay		**Level One**		**Level Two**		**Level Three**	
		Reward	Loss	Reward	Loss	Reward	Loss
0.0005; 1e-5	Tanh	29.18	0.04	5.53	0.01	17.62	0.06
	ReLu	10.01	28.04	37.22	0.01	12.51	0.05
0.0001; 1e-7	Tanh	31.36	0.08	27.6	0.03	12.12	0.09
	ReLu	37.82	0.05	37.25	0.01	7.4	0.05

0.001; 1e-4	Tanh	25.47	0.03	35.58	0.01	17.24	0.07
	ReLu	25.73	0.17	34.47	0.02	4.69	0.06
0.005; 1e-3	Tanh	32.52	0.04	27.89	0.01	11.32	0.06
	ReLu	32.3	0.04	30.48	0.01	1.47	0.04

Table 20

Distributional DQN performance, with different atoms min/max values, ADAM learning rates, and decay values. The hyperparameters used were: batch size 32; gamma 0.99; update target network period 50; network layers 3; hidden neuron size 128; Uniform (std 0.005) parameter initialization function.

Atoms min & max	**Lr & Decay**		**Level One**		**Level Two**		**Level Three**	
			Reward	Loss	Reward	Loss	Reward	Loss
-10; 10	0.0005;	Tanh	36.94	0.04	36.12	0.04	17.95	0.05
	1e-5	ReLu	26.94	0.04	32.8	0.03	22.54	0.05
	0.0001;	Tanh	32.7	0.04	19.6	0.03	9.42	0.05
	1e-7	ReLu	22.01	0.04	23.93	0.03	9.25	0.04
	0.001;	Tanh	30.43	0.04	31.7	0.04	16.69	0.05
	1e-4	ReLu	28.2	0.04	26.12	0.03	8.49	0.05
	0.005;	Tanh	38.94	0.04	33.9	0.04	14.23	0.05
	1e-3	ReLu	9.43	0.04	23.74	0.03	2.54	0.04
-7; 50	0.0005;	Tanh	33.49	0.04	30.76	0.04	13.6	0.05
	1e-5	ReLu	28.76	0.05	26.24	0.04	24.72	0.05
	0.0001;	Tanh	31.56	0.05	34.36	0.04	19.33	0.05
	1e-7	ReLu	33.12	0.05	28.95	0.04	9.89	0.05
	0.001;	Tanh	36.3	0.05	25.63	0.04	11.63	0.05
	1e-4	ReLu	32.13	0.05	32.44	0.04	13.02	0.05
	0.005;	Tanh	31.7	0.05	32.64	0.04	17.22	0.06
	1e-3	ReLu	24.82	0.05	30.66	0.04	2.83	0.04
-25; 50	0.0005;	Tanh	38.79	0.04	20.63	0.03	18.48	0.05
	1e-5	ReLu	32.47	0.04	25.15	0.03	20.61	0.05
	0.0001;	Tanh	33.19	0.05	15.21	0.03	10.35	0.05
	1e-7	ReLu	34.56	0.04	26.74	0.03	6.47	0.05
	0.001;	Tanh	33.97	0.04	23.55	0.03	14.96	0.05
	1e-4	ReLu	25.75	0.04	26.87	0.03	13.18	0.05
	0.005;	Tanh	36.74	0.04	16.6	0.03	16.78	0.05
	1e-3	ReLu	24.22	0.04	17.27	0.03	4.64	0.04
-25; 63	0.0005;	Tanh	32.48	0.04	32.48	0.04	19.98	0.05
	1e-5	ReLu	31.56	0.04	31.33	0.04	24.36	0.05
	0.0001;	Tanh	32.71	0.05	12.22	0.03	10.01	0.05
	1e-7	ReLu	32.43	0.04	21.87	0.03	9.8	0.05
	0.001;	Tanh	33.49	0.04	27.82	0.03	19.79	0.05
	1e-4	ReLu	35.54	0.04	19.82	0.03	13.47	0.05
	0.005;	Tanh	35.56	0.04	33.05	0.04	12.49	0.05
	1e-3	ReLu	24.81	0.04	19.19	0.03	5.26	0.04

Table 21

DQN with PER performance, with different combinations of beta and alpha values, different ADAM learning rates, and decay values. The hyperparameters used were: batch size 32; gamma 0.99; update target network period 50; network layers 3; hidden neuron size 128; Uniform (std 0.005) parameter initialization function.

Beta	**Alpha**	**Lr & Decay**		**Level One**		**Level Two**		**Level Three**	
				Reward	Loss	Reward	Loss	Reward	Loss
0.2	0.2	0.0005;	Tanh	33.93	0.02	30.69	0.01	18.36	0.02
		1e-5	ReLu	8.59	0.26	31.49	0.01	12.93	0.02
		0.0001;	Tanh	39.06	0.03	14.49	0.01	19.64	0.02
		1e-7	ReLu	32	0.02	39.06	0.06	7.76	0.01
		0.001;	Tanh	34.92	0.02	23.14	0.01	10.31	0.01
		1e-4	ReLu	2.91	0.53	36.52	0.01	11.43	0.01
		0.005;	Tanh	23.54	0.03	16.3	0.01	10.65	0.01
		1e-3	ReLu	26.69	0.06	29.85	0.01	6.19	0.01
	0.4	0.0005;	Tanh	35.2	0.03	16.3	0.01	10.93	0.04
		1e-5	ReLu	-12.19	1.12	23.55	0.02	9.09	0.05
		0.0001;	Tanh	36.74	0.06	30.05	0.02	12.78	0.06
		1e-7	ReLu	31.54	0.04	32.75	0.02	9.23	0.06
		0.001;	Tanh	36.66	0.04	13.95	0.02	8.84	0.04
		1e-4	ReLu	-7.8	0.86	30.14	0.02	5.48	0.05
		0.005;	Tanh	34.17	0.05	19.91	0.02	5.16	0.04
		1e-3	ReLu	15.41	0.05	22.14	0.02	3.81	0.06
	0.6	0.0005;	Tanh	36.99	0.06	8.51	0.12	3.63	0.11
		1e-5	ReLu	-12.32	2.94	18.87	0.1	9.13	0.21
		0.0001;	Tanh	35.2	0.12	7.04	0.08	6.54	0.2
		1e-7	ReLu	33.09	0.1	27.32	0.07	6.78	0.26
		0.001;	Tanh	36.06	0.07	20.39	0.1	2.59	0.11
		1e-4	ReLu	-10.29	2.48	16.67	0.14	4.79	0.16
		0.005;	Tanh	29.04	0.15	12.91	0.06	4.83	0.13
		1e-3	ReLu	2.37	4.35	8.98	0.05	4.05	0.35
	0.8	0.0005;	Tanh	28.69	0.18	5.38	0.16	1.4	0.36
		1e-5	ReLu	-14.76	6.91	7.3	0.26	4.46	0.56
		0.0001;	Tanh	32.46	0.19	6.39	0.08	6.46	0.72
		1e-7	ReLu	26.78	0.19	26.07	0.13	8.15	0.91
		0.001;	Tanh	24.58	0.23	12.75	0.16	0	0.32
		1e-4	ReLu	-14.6	8.2	14.49	0.65	1.68	0.44
		0.005;	Tanh	13.71	0.71	12.37	0.4	0.79	0.39
		1e-3	ReLu	1.52	4.46	-2.38	0.12	2.19	0.79
0.4	0.2	0.0005;	Tanh	33.3	0.02	20.82	0	18.49	0.02
		1e-5	ReLu	8.72	0.14	32.85	0.01	12.98	0.01
		0.0001;	Tanh	37.62	0.03	17.34	0.01	12.96	0.01
		1e-7	ReLu	36.14	0.02	42.67	0.01	12.18	0.01

		0.001;	Tanh	33.12	0.02	22.53	0.01	15.14	0.02
		1e-4	ReLu	7.27	0.26	33.54	0.01	6.08	0.01
		0.005;	Tanh	28.05	0.03	30.23	0.01	15.23	0.02
		1e-3	ReLu	11.37	0.05	35.46	0.02	5.15	0.01
	0.4	0.0005;	Tanh	34.97	0.02	16.1	0.01	20.01	0.05
		1e-5	ReLu	-3.18	0.57	30.29	0.02	10.06	0.05
		0.0001;	Tanh	39.21	0.06	14.82	0.02	12.59	0.06
		1e-7	ReLu	34.44	0.05	36.84	0.02	8.95	0.06
		0.001;	Tanh	35.23	0.04	16	0.01	12.62	0.05
		1e-4	ReLu	-10.46	1.84	33.28	0.02	7.65	0.05
		0.005;	Tanh	24.77	0.05	12.77	0.01	5.67	0.04
		1e-3	ReLu	4.12	0.39	16.12	0.02	6.22	0.07
	0.6	0.0005;	Tanh	36.94	0.07	9.68	0.06	8.04	0.14
		1e-5	ReLu	-10.18	7.15	26.01	0.07	8.86	0.28
		0.0001;	Tanh	36.54	0.17	8.44	0.06	7.74	0.3
		1e-7	ReLu	32.01	0.1	35.29	0.12	7.2	0.12
		0.001;	Tanh	25.7	0.11	27.86	0.1	6.41	0.16
		1e-4	ReLu	-6.31	3.71	29.33	0.09	4.33	0.21
		0.005;	Tanh	27.11	0.12	9.94	0.08	6.48	0.16
		1e-3	ReLu	1.69	1.13	16.67	0.05	4.86	0.16
	0.8	0.0005;	Tanh	30.07	0.29	4.77	0.15	8.69	0.54
		1e-5	ReLu	-12.15	34.15	17.91	0.51	7.05	1.17
		0.0001;	Tanh	33.65	0.33	2.69	0.23	8.51	1.13
		1e-7	ReLu	7.4	0.4	33.79	0.31	8.2	1.49
		0.001;	Tanh	22.74	0.31	17.44	0.68	3.84	0.59
		1e-4	ReLu	-12.32	14.37	18.74	0.12	4.65	0.88
		0.005;	Tanh	24.45	0.5	12.15	0.25	4.77	0.6
		1e-3	ReLu	-11.33	4.53	14.99	0.14	3.35	1.3
0.6	0.2	0.0005;	Tanh	40.24	0.02	12.31	0	19.63	0.02
		1e-5	ReLu	7.53	0.16	34.4	0.01	14.77	0.02
		0.0001;	Tanh	39.6	0.04	24.27	0.01	15.77	0.01
		1e-7	ReLu	31.8	0.03	38.29	0.01	12.73	0.01
		0.001;	Tanh	32.86	0.02	25.63	0.01	22.40	0.02
		1e-4	ReLu	6.23	0.22	30.29	0.01	8.11	0.01
		0.005;	Tanh	28.95	0.03	27.34	0.01	27.61	0.02
		1e-3	ReLu	25.21	0.07	35.15	0.02	3.84	0.01
	0.4	0.0005;	Tanh	35.39	0.04	17.47	0.02	22.66	0.06
		1e-5	ReLu	-7.97	0.81	35.86	0.02	9.87	0.05
		0.0001;	Tanh	33.78	0.07	19.58	0.03	10.48	0.07
		1e-7	ReLu	30.18	0.05	41.5	0.02	12.1	0.07
		0.001;	Tanh	38.3	0.04	23.99	0.02	14.16	0.06
		1e-4	ReLu	1.6	0.6	23.87	0.02	8.2	0.06
		0.005;	Tanh	33.68	0.06	23.29	0.03	20.8	0.06
		1e-3	ReLu	14.7	0.41	22.94	0.02	6.79	0.1
	0.6	0.0005;	Tanh	28.82	0.15	10.42	0.18	17.31	0.37
		1e-5	ReLu	-7.12	2.15	26.88	0.1	7.72	0.39
		0.0001;	Tanh	35.11	0.26	16.78	0.18	6.76	0.66
		1e-7	ReLu	35.6	0.11	38.67	0.1	8.77	0.57
		0.001;	Tanh	10.85	0.22	26.79	0.08	15.26	0.47
		1e-4	ReLu	-4.35	0.99	24.67	0.12	7.56	0.49

		0.005;	Tanh	22.09	0.23	9.43	0.14	13.05	0.49
		1e-3	ReLu	-5.14	2.23	16.52	0.12	3.32	0.44
	0.8	0.0005;	Tanh	4.69	0.95	4.34	0.2	5.25	1.3
		1e-5	ReLu	-8.34	6.19	8.75	0.64	8.02	7.93
		0.0001;	Tanh	37.66	0.82	5.65	0.3	7.9	2.28
		1e-7	ReLu	32.02	0.58	35.65	0.43	8.67	2.19
		0.001;	Tanh	6.11	0.69	8.74	0.42	5.34	2.11
		1e-4	ReLu	-10.16	6.65	23.48	0.74	7.66	2.01
		0.005;	Tanh	15.73	0.9	13.26	0.45	6	1.73
		1e-3	ReLu	3.61	0.92	22.68	0.63	2.25	1.83
0.8	0.2	0.0005;	Tanh	38.46	0.02	32.48	0.01	21.51	0.01
		1e-5	ReLu	8.35	0.24	37.69	0.01	11.66	0.01
		0.0001;	Tanh	38.67	0.04	34.15	0.01	14.34	0.01
		1e-7	ReLu	33.5	0.02	41.09	0.01	11.92	0.01
		0.001;	Tanh	31.23	0.02	21.98	0.01	21.01	0.02
		1e-4	ReLu	-1.1	0.26	33.79	0.01	7.46	0.01
		0.005;	Tanh	31.06	0.03	27.41	0.01	24.41	0.02
		1e-3	ReLu	10.16	0.11	33.24	0.02	1.99	0
	0.4	0.0005;	Tanh	30.76	0.06	25.18	0.02	24.8	0.08
		1e-5	ReLu	6.28	0.93	27.66	0.02	11.27	0.06
		0.0001;	Tanh	38.72	0.1	15.24	0.03	9.55	0.08
		1e-7	ReLu	29.85	0.07	39.92	0.02	9.11	0.07
		0.001;	Tanh	30.65	1.88	17.88	0.02	21.36	0.1
		1e-4	ReLu	1.88	0.67	36.37	0.03	9.13	0.07
		0.005;	Tanh	28.44	0.08	23.38	0.03	10.3	0.08
		1e-3	ReLu	7.23	0.46	29.61	0.03	6.12	0.1
	0.6	0.0005;	Tanh	27.77	0.36	18.44	0.28	8.07	0.9
		1e-5	ReLu	-7.38	1.06	13.58	0.24	9.53	0.65
		0.0001;	Tanh	37.6	0.6	-6.86	0.01	7.57	1.08
		1e-7	ReLu	30.11	0.25	26.35	0.07	10.21	0.91
		0.001;	Tanh	26.99	0.41	1.98	0.16	7.33	1.12
		1e-4	ReLu	8.18	1.19	24.6	0.33	7.57	0.76
		0.005;	Tanh	12.88	0.53	-1.97	0.16	7.26	1.06
		1e-3	ReLu	-5.72	0.78	5.53	0.23	4.4	0.84
	0.8	0.0005;	Tanh	1.42	1.57	-6.62	0.15	6.19	2.55
		1e-5	ReLu	-4.42	1.56	-5.08	0.29	5.09	2.49
		0.0001;	Tanh	19.23	2.38	-6.89	0.05	5.46	2.52
		1e-7	ReLu	24.52	2.27	8.71	0.87	6.64	2.66
		0.001;	Tanh	0.97	1.8	-5.7	0.23	6.71	2.54
		1e-4	ReLu	-2.38	1.63	1.13	0.69	5.83	2.62
		0.005;	Tanh	-15.15	1.29	-6.29	0.21	5.83	2.48
		1e-3	ReLu	5.58	1.42	0.6	0.77	3.38	2.37

Table 22

Rainbow DQN performance using different combinations of noisy and regular network layers, with different ADAM learning rates and decay values. The hyperparameters used were: batch size 32; gamma 0.99; update target network period 50; 1 input layer, 2 Value network layers,

2 Advantage network Layers; hidden neuron size 128; Uniform (std 0.005) parameter initialization function for regular layers, and Xavier initialization function for the noisy layers; N-step 3; support min and max values –25 and 63 respectively; Beta 0.6; Alpha 0.2; Sigma 0.05.

	Lr & Decay		**Level One**		**Level Two**		**Level Three**	
			Reward	Loss	Reward	Loss	Reward	Loss
Noisy	0.0005;	Tanh	40.26	0.01	29.73	0	32.78	0.01
Value &	1e-5	ReLu	41.03	0.01	40.86	0	30.3	0.01
Advantage	0.0001;	Tanh	42.99	0.01	37.81	0.01	20.11	0.01
layers	1e-7	ReLu	44.28	0.01	38.06	0	21.8	0.01
	0.001;	Tanh	41.81	0.01	36.95	0.01	29.5	0.01
	1e-4	ReLu	42.89	0.01	38.12	0	15.6	0.01
	0.005;	Tanh	14.58	0.01	0.14	0	7.07	0.01
	1e-3	ReLu	32.41	0.01	34.82	0	7.06	0.01
Noisy	0.0005;	Tanh	44.96	0.01	43.69	0.01	31.77	0.01
Value	1e-5	ReLu	46.16	0.01	42.22	0.01	29.75	0.01
layers	0.0001;	Tanh	44.35	0.01	42.5	0.01	18.52	0.01
	1e-7	ReLu	41.48	0.01	43.8	0.01	24.72	0.01
	0.001;	Tanh	41.35	0.01	36.18	0.01	29.33	0.01
	1e-4	ReLu	36.69	0.01	34.9	0	18.96	0.01
	0.005;	Tanh	40.2	0.01	26.74	0.01	9.32	0.01
	1e-3	ReLu	1.76	0.01	17.87	0.01	5.21	0.01
Noisy	0.0005;	Tanh	40.15	0.01	35.39	0	29.93	0.01
Advantage	1e-5	ReLu	40.3	0.01	41.28	0	22.87	0.01
layers	0.0001;	Tanh	41.15	0.01	31.66	0.01	16.76	0.01
	1e-7	ReLu	39.9	0.01	43.27	0	16.11	0.01
	0.001;	Tanh	37.76	0.01	39.66	0	30.8	0.01
	1e-4	ReLu	45	0.01	38.52	0	15.49	0.01
	0.005;	Tanh	36.48	0.01	28.39	0	19.46	0.01
	1e-3	ReLu	41.11	0.01	35.5	0	3.16	0.01
No Noisy	0.0005;	Tanh	39.05	0.01	39.59	0.01	25.64	0.01
layers	1e-5	ReLu	43.85	0.01	42.25	0	27.04	0.01
	0.0001;	Tanh	42.21	0.01	35.47	0.01	17.51	0.01
	1e-7	ReLu	37.93	0.01	39.39	0	19.57	0.01
	0.001;	Tanh	41.96	0.01	35.7	0	23.54	0.01
	1e-4	ReLu	43.73	0.01	23.47	0	16.83	0.01
	0.005;	Tanh	41.76	0.01	43.07	0.0	31.23	0.01
	1e-3	ReLu	39.44	0.01	19.94	0	3.05	0.01

Table 23

RS performance, with different Noise std values. The hyperparameters used were: population size 50; network layers 3; hidden neuron size 128; Uniform (std 0.005) parameter initialization function.

Noise std		Level One		Level Two	
		Population Reward	Best Reward	Population Reward	Best Reward
0.005	Tanh	15.12	46.21	-4.33	-1.69
	ReLu	12.05	46.02	-2.69	4.41
0.01	Tanh	18.59	48.63	0.6	12.14
	ReLu	15.92	46.52	-0.43	12.14
0.015	Tanh	13.3	50.89	0.14	13
	ReLu	14.75	48.71	-2.23	8.54
0.02	Tanh	21.05	49.01	0.12	21.11
	ReLu	19	48.93	-1.93	8.74
0.025	Tanh	17.99	49.8	0.15	21.79
	ReLu	17.65	48.83	-3.51	7.06
0.03	Tanh	23.41	49.73	-1.93	15.38
	ReLu	17.69	50.30	-0.11	15.26

Table 24

RS performance, with different number of layers. The hyperparameters used were: population size 50; noise std 0.03; hidden neuron size 128; Uniform (std 0.005) parameter initialization function.

Layers		Level One		Level Two	
		Population Reward	Best Reward	Population Reward	Best Reward
2	Tanh	23.24	49.08	2.25	6.78
	ReLu	20.94	49.54	2.17	15.92
3	Tanh	23.41	49.73	-1.93	15.38
	ReLu	17.69	50.30	-0.11	15.26
4	Tanh	21.59	49.04	-2.88	20.1
	ReLu	14.47	48.73	-4.44	4.16

Table 25

RS performance, with different neuron sizes. The hyperparameters used were: population size 50; noise std 0.03; network layers 3; Uniform (std 0.005) parameter initialization function.

Neurons		Level One		Level Two	
		Population Reward	Best Reward	Population Reward	Best Reward
32	Tanh	17.92	49.78	0.54	21.92
	ReLu	21.11	50.32	0.31	15.03
64	Tanh	23.99	49.1	0.36	18.1
	ReLu	16.32	49.88	0.48	15.17
128	Tanh	20.94	49.54	2.17	15.92
	ReLu	23.41	49.73	-1.93	15.38
256	Tanh	20.37	49.51	-0.05	23.31

	ReLu	16.98	49.18	-2.08	7.81

Table 26

RS performance, with different population sizes. The hyperparameters used were: noise std 0.03; network layers 3; hidden neuron size 128; Uniform (std 0.005) parameter initialization function.

Population		**Level One**		**Level Two**	
		Population Reward	Best Reward	Population Reward	Best Reward
26	Tanh	20.35	48.8	0.25	17.34
	ReLu	19.62	48.46	0.31	12.53
50	Tanh	17.92	49.78	0.54	21.92
	ReLu	21.11	50.32	0.31	15.03
76	Tanh	19.19	50.5	-1.01	16.1
	ReLu	11.82	49.91	0.73	20.55

Table 27

CMA-ES performance using different learning rates and decay values for the ADAM optimizer. The hyperparameters used were: population size 50; noise std 0.01; network layers 3; hidden neuron size 128; Uniform (std 0.005) parameter initialization function.

Lr	**Decay**		**Level One**		**Level Two**		**Level Three**	
			Population Reward	Best Reward	Population Reward	Best Reward	Population Reward	Best Reward
0.01	1e-2	Tanh	33.74	37.23	25.07	27.16	8.43	31.7
		ReLu	36	39.76	20.48	22.85	5.29	22.58
	1e-3	Tanh	43.95	47.53	25.06	27.39	9.67	25.65
		ReLu	37.16	40.39	24.32	26.55	4.27	21.5
	1e-4	Tanh	43.55	47.67	25.26	27.53	7.32	25.77
		ReLu	36.62	39.61	16.71	18.52	3.81	16.46
	1e-5	Tanh	43.85	47.67	25.2	27.74	7.82	26.46
		ReLu	36.97	40.07	16.52	18.53	2.5	11.84
	1e-6	Tanh	43.64	47.26	25.36	27.71	8.37	24.64
		ReLu	36.93	40.05	16.5	18.53	3.81	14.95
0.005	1e-2	Tanh	29.86	37.71	26.65	33.12	5.48	38.11
		ReLu	20.41	26.3	10.42	13.29	3.83	31.79
	1e-4	Tanh	38.04	43.26	27.11	30.79	8.46	33.78
		ReLu	24.73	29.55	12.77	15.27	4.31	30.69
	1e-5	Tanh	37.01	42.1	27.36	30.74	6.17	36.26
		ReLu	22.8	27.28	12.54	15.3	3.58	27.48
	1e-6	Tanh	36.87	42.08	27.32	30.73	6.6	34.79
		ReLu	25.55	30.21	12.61	15.29	3.65	28.95
0.05	1e-2	Tanh	25.44	26.62	-5.56	-5.45	2.98	7.9

		ReLu	24.25	25.72	4.54	5.09	1.72	5.56
	1e-3	Tanh	25.06	26.28	-6.08	5.93	0.15	5.38
		ReLu	14.15	15.41	2.75	3.3	1.55	5.04
0.1	1e-4	Tanh	19.6	20.94	-7.01	-6.92	-2.18	1.51
		ReLu	13.53	14.25	-6.43	-6.19	0.13	1.12

Table 28

CMA-ES performance using different Noise std values, with different ADAM learning rates and decay values. The hyperparameters used were: population size 50; network layers 3; hidden neuron size 128; Uniform (std 0.005) parameter initialization function.

Noise std	**Lr & Decay**		**Level One**		**Level Two**		**Level Three**	
			Population Reward	Best Reward	Population Reward	Best Reward	Population Reward	Best Reward
0.01	0.01;	Tanh	43.95	47.53	25.06	27.39	9.67	25.65
	1e-3	ReLu	37.16	40.39	24.32	26.55	4.27	21.5
	0.01;	Tanh	43.85	47.67	25.2	27.74	7.82	26.46
	1e-5	ReLu	36.97	40.07	16.52	18.53	2.5	11.84
	0.005;	Tanh	38.04	43.26	27.11	30.79	8.46	33.78
	1e-4	ReLu	24.73	29.55	12.77	15.27	4.31	30.69
	0.005;	Tanh	36.87	42.08	27.32	30.73	6.6	34.79
	1e-6	ReLu	25.55	30.21	12.61	15.29	3.65	28.95
0.015	0.01;	Tanh	42.72	47.59	19.02	21.64	6.3	30.12
	1e-3	ReLu	32.23	36.82	9.5	12.14	4.74	30.61
	0.01;	Tanh	39.33	44.24	20.74	23.27	6.58	31.42
	1e-5	ReLu	35.01	40.56	9.63	12.23	6.41	26.45
	0.005;	Tanh	39.09	47.26	29.4	35.52	5.85	42.02
	1e-4	ReLu	24.86	33.51	17.69	22.1	3.68	36.95
	0.005;	Tanh	38.22	46.61	29.54	35.52	7.05	41.25
	1e-6	ReLu	25.51	33.78	17.87	22.17	3.36	35.39
0.02	0.01;	Tanh	41.09	48.07	19.17	22.26	6.64	36.31
	1e-3	ReLu	36.21	43.84	6.19	9.14	4.21	32.71
	0.01;	Tanh	40.26	47.35	17.8	20.57	9.66	39.84
	1e-5	ReLu	34.62	41.28	4.16	6.67	4.08	29.95
	0.005;	Tanh	34.95	47.45	23.44	30.54	3.16	43.63
	1e-4	ReLu	21.28	34.36	2.47	6.12	3.85	40.43
	0.005;	Tanh	34.74	47.36	23.63	30.72	3.85	40.43
	1e-6	ReLu	22.28	35.28	3.48	6.22	3.56	40.68
0.025	0.01;	Tanh	39.63	48.32	23.08	27.61	7.12	41.83
	1e-3	ReLu	27.57	35.8	6.02	8.93	4.43	37.03
	0.01;	Tanh	39.8	48.07	22.59	26.72	6.93	40.8
	1e-5	ReLu	30.99	39.05	4.75	8.66	4.47	35.04
	0.005;	Tanh	30.06	47.49	14.5	21.39	2.83	45.41
	1e-4	ReLu	22.45	40.62	8.57	14.38	3.08	43.24
	0.005;	Tanh	30.1	47.47	14.55	21.82	3.08	45.22
	1e-6	ReLu	24.11	42.29	8.46	14.36	3.01	42.68

Table 29

CMA-ES performance using different number of layers, with different ADAM learning rates and decay values. The hyperparameters used were: population size 50; noise std 0.015; hidden neuron size 128; Uniform (std 0.005) parameter initialization function.

Layers	Lr & Decay		Level One		Level Two		Level Three	
			Population Reward	Best Reward	Population Reward	Best Reward	Population Reward	Best Reward
2	0.01;	Tanh	43.89	48.66	2.1	2.81	9.97	32.37
	1e-3	ReLu	42.78	47.65	7.28	8.57	8.68	33.09
	0.01;	Tanh	43.9	48.67	2.18	2.81	6.23	25.68
	1e-5	ReLu	42.63	47.99	10.14	11.65	6.64	27.31
	0.005;	Tanh	38.91	46.71	10.24	13.05	7.46	37.49
	1e-4	ReLu	38.14	46.54	11.69	14.32	5.45	34.67
	0.005;	Tanh	38.57	46.8	10.22	13.08	7.04	39.95
	1e-6	ReLu	36.45	46.45	11.83	14.32	6.04	36.43
3	0.01;	Tanh	42.72	47.59	19.02	21.64	6.3	30.12
	1e-3	ReLu	32.23	36.82	9.5	12.14	4.74	30.61
	0.01;	Tanh	39.33	44.24	20.74	23.27	6.58	31.42
	1e-5	ReLu	35.01	40.56	9.63	12.23	6.41	26.45
	0.005;	Tanh	39.09	47.26	29.4	35.52	5.85	42.02
	1e-4	ReLu	24.86	33.51	17.69	22.1	3.68	36.95
	0.005;	Tanh	38.22	46.61	29.54	35.52	7.05	41.25
	1e-6	ReLu	25.51	33.78	17.87	22.17	3.36	35.39
4	0.01;	Tanh	42.15	47.57	6.06	7.59	7.48	34.83
	1e-3	ReLu	19.93	23.97	8.01	10.56	5.31	30.85
	0.01;	Tanh	41.84	47.6	15.14	17.78	5.45	35.28
	1e-5	ReLu	18.56	22.12	8.81	10.84	4.63	29.02
	0.005;	Tanh	32.54	42.68	17.81	23.23	4.17	41.28
	1e-4	ReLu	2.43	7.02	7.71	11.8	3.98	33.93
	0.005;	Tanh	31.58	41.77	17.53	22.59	4.44	41.73
	1e-6	ReLu	3.16	8.04	7.74	11.38	3.93	35.48

Table 30

CMA-ES performance using different neuron sizes, with different ADAM learning rates and decay values. The hyperparameters used were: population size 50; noise std 0.015; network layers 3; Uniform (std 0.005) parameter initialization function.

Neurons	Lr & Decay	Level One		Level Two		Level Three	
		Population Reward	Best Reward	Population Reward	Best Reward	Population Reward	Best Reward

32	0.01;	Tanh	32.77	37.45	12.08	14.32	6.11	26.81
	1e-3	ReLu	30.04	34.37	4.04	5.03	5.1	24.65
	0.01;	Tanh	32.19	36.62	12.78	14.74	6.4	27.36
	1e-5	ReLu	34.62	39.16	-0.45	0.53	5.25	23.03
	0.005;	Tanh	29.99	38.72	13.61	17.7	4.85	35.46
	1e-4	ReLu	13.67	20.76	9.9	12.93	3.87	30.26
	0.005;	Tanh	30.37	38.64	15.97	20.35	5.15	35.55
	1e-6	ReLu	12.59	20.06	9.74	12.93	4.04	28.9
64	0.01;	Tanh	38.08	42.52	20.24	22.91	8.55	31.86
	1e-3	ReLu	27.58	31.94	29.43	33.21	4.79	25.93
	0.01;	Tanh	42.8	47.28	20.59	22.91	8.28	31.43
	1e-5	ReLu	26.43	30.34	25.14	28.51	5.96	26.63
	0.005;	Tanh	32.74	43.11	23.22	28.37	5.18	39.63
	1e-4	ReLu	24.12	32.29	17.85	22.29	4.54	33.91
	0.005;	Tanh	34.71	44.37	23.3	28.38	5.83	39.21
	1e-6	ReLu	22.56	30.19	17.97	22.35	4.22	34.31
128	0.01;	Tanh	42.72	47.59	19.02	21.64	6.3	30.12
	1e-3	ReLu	32.23	36.82	9.5	12.14	4.74	30.61
	0.01;	Tanh	39.33	44.24	20.74	23.27	6.58	31.42
	1e-5	ReLu	35.01	40.56	9.63	12.23	6.41	26.45
	0.005;	Tanh	39.09	47.26	29.4	35.52	5.85	42.02
	1e-4	ReLu	24.86	33.51	17.69	22.1	3.68	36.95
	0.005;	Tanh	38.22	46.61	29.54	35.52	7.05	41.25
	1e-6	ReLu	25.51	33.78	17.87	22.17	3.36	35.39
256	0.01;	Tanh	43.33	48.61	21.25	23.81	11.08	35.56
	1e-3	ReLu	32.19	36.86	15.22	17.49	4.83	28.71
	0.01;	Tanh	42.68	48.14	22.63	25.16	7.35	30.21
	1e-5	ReLu	32.89	37.52	17.41	19.3	4.84	28.02
	0.005;	Tanh	37.12	47.64	23.63	30.28	5.41	41.48
	1e-4	ReLu	34.81	46.3	19.4	24.22	4.75	37.56
	0.005;	Tanh	37.13	47.3	23.45	30.49	5.83	42.47
	1e-6	ReLu	34.39	46	19.42	24.18	5.13	38.49

Table 31

CMA-ES performance using Rank reward normalization, with different ADAM learning rates and decay values. The hyperparameters used were: population size 50; noise std 0.015; network layers 3; hidden neuron size 256; Uniform (std 0.005) parameter initialization function.

Lr & decay		**Level One**		**Level Two**		**Level Three**	
		Population Reward	Best Reward	Population Reward	Best Reward	Population Reward	Best Reward
0.01; 1e-3	Tanh	39.88	47.32	14.63	18.91	5.71	35.32
	ReLu	31.75	39.07	9.1	11.26	4.83	29.6
0.01; 1e-5	Tanh	38.15	46.32	14.7	19.05	5.7	33.75
	ReLu	23.72	30.18	10.19	13.3	4.41	28.6
0.005; 1e-4	Tanh	33.81	46.17	11.26	16.52	5.92	44.05

	ReLu	11.7	21.31	12.05	16.2	4.62	40.84
0.005; 1e-6	Tanh	33.23	46.05	13.08	17.82	5.1	43.98
	ReLu	13.18	24.35	11.5	16.19	4.07	39.22

Table 32

CMA-ES performance using different population sizes, with different ADAM learning rates and decay values. The hyperparameters used were: noise std 0.015; network layers 3; hidden neuron size 256; Uniform (std 0.005) parameter initialization function.

Population	**Lr & Decay**		**Level One**		**Level Two**		**Level Three**	
			Population Reward	Best Reward	Population Reward	Best Reward	Population Reward	Best Reward
26	0.01;	Tanh	36.53	43.36	13.33	16.57	6.89	29.6
	1e-3	ReLu	26.45	32.59	7.82	10.14	4.6	27.01
	0.01;	Tanh	37.62	43.81	12.2	15.03	4.82	28.28
	1e-5	ReLu	28.05	33.28	9.73	11.99	6.11	27.6
	0.005;	Tanh	26.84	36.53	9.72	13.95	4.24	37.2
	1e-4	ReLu	19.1	28.71	9.49	14.05	3.83	34.37
	0.005;	Tanh	26.15	35.74	9.85	13.99	3.74	35.59
	1e-6	ReLu	19.26	28.54	9.81	14.24	3.87	32.77
50	0.01;	Tanh	43.33	48.61	21.25	23.81	11.08	35.56
	1e-3	ReLu	32.19	36.86	15.22	17.49	4.83	28.71
	0.01;	Tanh	42.68	48.14	22.63	25.16	7.35	30.21
	1e-5	ReLu	32.89	37.52	17.41	19.3	4.84	28.02
	0.005;	Tanh	37.12	47.64	23.63	30.28	5.41	41.48
	1e-4	ReLu	34.81	46.3	19.4	24.22	4.75	37.56
	0.005;	Tanh	37.13	47.3	23.45	30.49	5.83	42.47
	1e-6	ReLu	34.39	46	19.42	24.18	5.13	38.49
76	0.01;	Tanh	43.41	48.64	20.88	23.64	9.46	37.43
	1e-3	ReLu	38.22	43.01	11.78	13.79	6.04	31.66
	0.01;	Tanh	43.53	48.91	24.84	27.83	8.77	35.46
	1e-5	ReLu	38.28	43.01	11.78	13.84	8	32.64
	0.005;	Tanh	39.73	48.74	26.68	32.48	7.74	45.06
	1e-4	ReLu	33.69	43.89	9.02	13.56	4.09	42.31
	0.005;	Tanh	39.95	48.75	26.84	32.84	8.89	45.77
	1e-6	ReLu	35.64	45.66	8.73	13.45	4.76	42.21

Table 33

GA performance, with different Noise std values. The hyperparameters used were: population size 50; elitism size 3; tournament size 6; minimum/maximum mutation rate 0.02/0.1; network layers 3; hidden neuron size 128; Uniform (std 0.005) parameter initialization function.

Noise std	**Level One**	**Level Two**	**Level Three**

		Population Reward	Best Reward	Population Reward	Best Reward	Population Reward	Best Reward
0.005	Tanh	26.71	50.25	4.8	17.97	3.95	37.64
	ReLu	24.54	48.6	0.91	13.72	4.48	34.93
0.01	Tanh	21.76	48.79	11.04	30.61	5.34	40.01
	ReLu	19.56	48.87	5.12	23.62	4.38	37.44
0.015	Tanh	19.38	49.05	13.08	43.42	3.91	40.25
	ReLu	15.54	48.87	13.51	39.82	3.97	37.49
0.02	Tanh	12.83	50.09	12.24	44.74	2.59	39.66
	ReLu	15.83	48.92	9.07	34.37	3.3	35.39
0.025	Tanh	8.12	48.96	16.76	43.65	2.5	40.63
	ReLu	9.34	48.69	10.39	39.36	4.03	36.58
0.03	Tanh	8.12	48.96	16.76	43.65	2.5	40.63
	ReLu	9.34	48.69	10.39	39.36	4.03	36.58

Table 34

GA performance, with different Tournament sizes. The hyperparameters used were: population size 50; elitism size 3; noise std 0.015; minimum/maximum mutation rate 0.02/0.1; network layers 3; hidden neuron size 128; Uniform (std 0.005) parameter initialization function.

Tournament size		**Level One**		**Level Two**	
		Population Reward	Best Reward	Population Reward	Best Reward
2	Tanh	-11.99	48.96	-3.23	41.23
	ReLu	-11.68	48.77	-4.31	30.82
6	Tanh	19.38	49.05	13.08	43.42
	ReLu	15.54	48.87	13.51	39.82
8	Tanh	23.42	48.95	11.92	33.77
	ReLu	18.61	50.21	0.67	31.03
10	Tanh	20.58	49.01	10.67	43.35
	ReLu	19.24	49.99	11.25	40.35

Table 35

GA performance, with different Elitism sizes. The hyperparameters used were: population size 50; tournament size 10; noise std 0.015; minimum/maximum mutation rate 0.02/0.1; network layers 3; hidden neuron size 128; Uniform (std 0.005) parameter initialization function.

Elitism size		**Level One**		**Level Two**	
		Population Reward	Best Reward	Population Reward	Best Reward
1	Tanh	31.97	50.53	15.91	40.72
	ReLu	32.43	48.84	15.94	37.6
3	Tanh	20.58	49.01	10.67	43.35

	ReLu	19.24	49.99	11.25	40.35
5	Tanh	9.39	48.96	13.79	43.67
	ReLu	13.01	48.44	7.45	36.98
7	Tanh	11.7	50.34	12.48	43.23
	ReLu	9.68	48.49	9.2	39.84

Table 36

GA performance, with different number of Layers. The hyperparameters used were: population size 50; elitism size 3; tournament size 6; noise std 0.015; minimum/maximum mutation rate 0.02/0.1; hidden neuron size 128; Uniform (std 0.005) parameter initialization function.

Layers		**Level One**		**Level Two**	
		Population Reward	Best Reward	Population Reward	Best Reward
2	Tanh	29.86	49.24	16.52	37.88
	ReLu	28.84	49.21	16.91	37.05
3	Tanh	20.58	49.01	10.67	43.35
	ReLu	19.24	49.99	11.25	40.35
4	Tanh	9.14	48.39	9.5	41.02
	ReLu	9.08	48.23	8.57	37.4

Table 37

GA performance, with different neuron sizes. The hyperparameters used were: population size 50; elitism size 3; tournament size 6; noise std 0.015; minimum/maximum mutation rate 0.02/0.1; network layers 3; Uniform (std 0.005) parameter initialization function.

Neurons		**Level One**		**Level Two**	
		Population Reward	Best Reward	Population Reward	Best Reward
32	Tanh	18.48	48.5	18.64	44.37
	ReLu	23.57	48.3	17.07	44.56
64	Tanh	18.07	48.83	15.49	44.2
	ReLu	16.23	49.8	17.48	38.9
128	Tanh	16.2	50.12	15.53	39.02
	ReLu	26.46	49.21	16.83	36.83
256	Tanh	21.67	49.3	9.01	29.3
	ReLu	18.72	49.02	8.62	30.2

Table 38

GA performance, with different population sizes. The hyperparameters used were: elitism size 3; tournament size 6; noise std 0.015; minimum/maximum mutation rate 0.02/0.1; network layers 3; hidden neuron size 64; Uniform (std 0.005) parameter initialization function.

Population		Level One		Level Two	
		Population Reward	Best Reward	Population Reward	Best Reward
26	Tanh	21.67	49.3	9.01	29.3
	ReLu	18.72	49.02	8.61	30.2
50	Tanh	16.2	50.12	15.53	39.02
	ReLu	26.46	49.21	16.83	36.83
76	Tanh	34.04	49.25	19.94	45.54
	ReLu	34.97	49.28	18.89	41.73

Table 39

List of NEAT hyperparameters not tuned in this project. The values for these parameters follow the original work (REF), with small modifications being made based on simple tests that were not recorded.

Parameter	Value
Add Link Rate	0.15
Add Recurrent Link Rate	0.05
Add Neuron Rate	0.1
Weight Mutation Rate	0.15
Weight Replace Rate	0.1
Crossover Rate	0.8
Species Old Threshold	50
Species Old Penalty	0.3
Species Young Threshold	10
Species Young Bonus	0.3
Species no Improvement Threshold	75
Species Compatibility Threshold	0.26
Compatibility Disjoint Weight	11
Compatibility Excess Weight	11
Compatibility Matched Weight	5

Table 40

NEAT performance, with different Noise std values. The hyperparameters used were: population size 50; Uniform (std 0.005) parameter initialization function; the remaining hyperparameters can be seen in Table 37.

Noise std		Level One		Level Two		Level Three	
		Population Reward	Best Reward	Population Reward	Best Reward	Population Reward	Best Reward
0.005	Tanh	35.39	47.17	23.49	32.42	1.08	34.37
	ReLu	35.31	49.20	24	32.44	1.5	34.48
0.01	Tanh	34.56	49.92	30.98	41.43	2.59	38.86
	ReLu	33.96	49.19	30.82	41.42	1.01	37.67
0.015	Tanh	32.77	49.16	32.58	43.45	1.85	38.89
	ReLu	32.73	50.57	32.51	43.43	1.36	38.88
0.02	Tanh	32.87	49.17	32.28	43.68	1.33	40.05
	ReLu	34.06	49.15	32.62	43.69	1.53	40.31
0.025	Tanh	32.6	49.15	33.06	44.23	1.32	39.11
	ReLu	32.74	49.59	33.24	44.22	1.11	39.06
0.03	Tanh	34.78	50.58	31.36	42.13	1.28	39.38
	ReLu	33.94	49.16	31.46	42.1	15.37	40.28

Table 41

NEAT performance, with different population sizes. The hyperparameters used were: noise std 0.015; Uniform (std 0.005) parameter initialization function; the remaining hyperparameters can be seen in Table 37.

Population		Level One		Level Two	
		Population Reward	Best Reward	Population Reward	Best Reward
26	Tanh	31.36	49.14	28.77	30.67
	ReLu	29.24	49.41	24.01	31.94
50	Tanh	32.77	49.16	32.58	43.45
	ReLu	32.73	50.57	32.51	43.43
76	Tanh	35.62	49.25	30.66	40.3
	ReLu	35.63	49.25	30.51	40.34

Table 42

Neuroevolution algorithm performance. RS hyperparameters used can be seen in Table 4, CMA-ES hyperparameters used can be seen in Table 5, GA hyperparameters used can be seen in Table 6, NEAT hyperparameters used can be seen in Table 7

Algorithm		Level Three	
		Population Reward	Best Reward
RS	Tanh	7.80	20.11
	ReLu	22.04	39.29
CMA-ES	Tanh	17.46	25,31
	ReLu	8.78	16.65
GA	Tanh	32.24	44.88

	ReLu	27.06	42.37
NEAT	Tanh	32.81	48.28
	ReLu	10.26	24.99

Table 43

RS-NS performance, with different Novelty Relevance values. The hyperparameters used were: population size 76; noise std 0.03; network layers 3; hidden neuron size 32; Uniform (std 0.005) parameter initialization function.

Novelty Relevance		**Level One**		**Level Two**		**Level Three**	
		Population Reward	Best Reward	Population Reward	Best Reward	Population Reward	Best Reward
0.2	Tanh	19.61	48.74	8.74	37.05	0.84	23.63
	ReLu	24.51	49.01	15.68	45.38	0.87	22.14
0.5	Tanh	3.94	46.21	5.04	32.68	0.47	26.16
	ReLu	7.21	46.13	10.49	40.67	1.42	24.78
0.8	Tanh	-7.56	40.31	-4.2	11.68	0.52	28.75
	ReLu	-4.36	39.07	-4.34	8.96	1.12	28.88

Table 44

CMA-ES-NS performance, using different Novelty Relevance values, with different ADAM learning rates and decay values. The hyperparameters used were: population size 50; noise std 0.015; network layers 3; hidden neuron size 256; Uniform (std 0.005) parameter initialization function.

Novelty Relevance	**Lr & Decay**		**Level One**		**Level Two**		**Level Three**	
			Population Reward	Best Reward	Population Reward	Best Reward	Population Reward	Best Reward
0.2	0.01;	Tanh	43.39	48.63	21.73	28.93	7.6	40.81
	1e-5	ReLu	42.57	48.3	13.92	19.62	7.65	42.06
	0.005;	Tanh	38.41	48.12	8.06	19.96	4.5	46.02
	1e-6	ReLu	36.88	48.09	16.89	29.33	3.62	44.12
0.5	0.01;	Tanh	39.69	48.53	16.95	25.2	8.03	44.84
	1e-5	ReLu	38.43	48.35	17.19	24.69	7.43	44.3
	0.005;	Tanh	34.6	48.46	3.64	18.01	4.51	47.39
	1e-6	ReLu	30.6	47.53	17.71	31.47	2.96	47.23
0.8	0.01;	Tanh	-5.91	46.99	-3.06	23.74	0.83	43.22
	1e-5	ReLu	-5.86	46.84	-2.24	19.37	1.08	43.61
	0.005;	Tanh	-5.92	47.57	-5.92	12.05	0.47	46.89
	1e-6	ReLu	-5.73	47.46	-4.39	15.72	0.44	46.96

Table 45

GA-NS performance, with different Novelty Relevance values. The hyperparameters used were: population size 76; noise std 0.015; elitism size 3; tournament size 10; network layers 3; hidden neuron size 64; Uniform (std 0.005) parameter initialization function.

Novelty Relevance		Level One		Level Two		Level Three	
		Population Reward	Best Reward	Population Reward	Best Reward	Population Reward	Best Reward
0.2	Tanh	32.7	49.06	18.33	47.02	4.79	48.83
	ReLu	29.42	51.3	15.01	46.57	4.45	48.32
0.5	Tanh	26.58	49.49	18.46	47.02	6.48	48
	ReLu	26.14	48.7	17.36	46.73	4.7	48.4
0.8	Tanh	-7.18	48.5	-3.93	41.72	0.86	49.14
	ReLu	-6.76	48.62	-3.51	43.32	0.89	49.29

Table 46

NEAT-NS performance, with different Novelty Relevance values. The hyperparameters used were: population size 50; noise std 0.015; Uniform (std 0.005) parameter initialization function; the remaining hyperparameters can be seen in Table 37

Novelty Relevance		Level One		Level Two		Level Three	
		Population Reward	Best Reward	Population Reward	Best Reward	Population Reward	Best Reward
0.2	Tanh	20.37	52.25	6.1	32.41	1.57	44.71
	ReLu	21.94	53.13	6.68	29.56	1.42	44.54
0.5	Tanh	13.44	49.47	0.49	35.65	1.18	47.2
	ReLu	11.91	49.1	-3.16	29.51	1.02	46.81
0.8	Tanh	-7.68	47.4	-4.14	34.36	0.29	46.96
	ReLu	-9.19	46.88	-4.04	27.6	0.3	46.96

Table 47

DQN performance, using different ADAM learning rates and decay values. The hyperparameters used were: batch size 32; gamma 0.99; update target network period 50; network layers 3; hidden neuron size 128; Uniform (std 0.005) parameter initialization function

Lr & Decay		Level Four	
		Reward	Loss
0.0005; 1e-5	Tanh	-13.16	0.02
	ReLu	-15.92	0.15
0.0001; 1e-7	Tanh	-11.69	0.01

	ReLu	-10.34	0.02
0.001; 1e-4	Tanh	-8.72	0.01
	ReLu	-15.61	0
0.005; 1e-3	Tanh	-15.34	0.01
	ReLu	-14.93	0

Table 48

Double DQN performance, using different ADAM learning rates and decay values. The hyperparameters used were: batch size 32; gamma 0.99; update target network period 50; network layers 3; hidden neuron size 128; Uniform (std 0.005) parameter initialization function

Lr & Decay		**Level Four**	
		Reward	Loss
0.0005; 1e-5	Tanh	-10.83	0.02
	ReLu	-14.26	0.04
0.0001; 1e-7	Tanh	-12	0.01
	ReLu	-7.58	0.03
0.001; 1e-4	Tanh	-9.9	0.01
	ReLu	-15.05	0.01
0.005; 1e-3	Tanh	-14.1	0
	ReLu	-15.27	0

Table 49

Dueling DQN performance, using different ADAM learning rates and decay values. The hyperparameters used were: batch size 32; gamma 0.99; update target network period 50; network layers 3; hidden neuron size 128; Uniform (std 0.005) parameter initialization function

Lr & Decay		**Level Four**	
		Reward	Loss
0.0005; 1e-5	Tanh	-7.69	0.03
	ReLu	-15.72	0.02
0.0001; 1e-7	Tanh	-11.44	0.01
	ReLu	-13.88	0.02
0.001; 1e-4	Tanh	-9.24	0
	ReLu	-14.85	0
0.005; 1e-3	Tanh	-13.73	0
	ReLu	-14.53	0

Table 50

N-Step performance, using different ADAM learning rates and decay values. The hyperparameters used were: batch size 32; gamma 0.99; update target network period 50;

network layers 3; hidden neuron size 128; Uniform (std 0.005) parameter initialization function; N-step 3

Lr & Decay		Level Four	
		Reward	Loss
0.0005; 1e-5	Tanh	1.7	0.04
	ReLu	3.1	0.11
0.0001; 1e-7	Tanh	-4.96	0.02
	ReLu	-2.49	0.05
0.001; 1e-4	Tanh	-11.15	0.02
	ReLu	-10.94	0.03
0.005; 1e-3	Tanh	-14.53	0.01
	ReLu	-14.54	0

Table 51

DQN with PER performance, using different ADAM learning rates and decay values. The hyperparameters used were: batch size 32; gamma 0.99; update target network period 50; network layers 3; hidden neuron size 128; Uniform (std 0.005) parameter initialization function; Beta 0.6; Alpha 0.2

Lr & Decay		Level Four	
		Reward	Loss
0.0005; 1e-5	Tanh	-12.03	0.02
	ReLu	-11.07	0.02
0.0001; 1e-7	Tanh	-7.2	0.01
	ReLu	-0.73	0.02
0.001; 1e-4	Tanh	-9.45	0.01
	ReLu	-5.2	0.01
0.005; 1e-3	Tanh	-4.41	0.01
	ReLu	-14.94	0

Table 52

Noisy DQN performance, using different ADAM learning rates and decay values. The hyperparameters used were: batch size 32; gamma 0.99; update target network period 50; network layers 3; hidden neuron size 128; Uniform (std 0.005) parameter initialization function; Sigma 0.05

Lr & Decay		Level Four	
		Reward	Loss
0.0005; 1e-5	Tanh	-15.13	0.01
	ReLu	-12.74	0.04
0.0001; 1e-7	Tanh	-14.94	0

	ReLu	-14.48	0.01
0.001; 1e-4	Tanh	-15.46	0
	ReLu	-10.89	0.03
0.005; 1e-3	Tanh	-12.69	0.01
	ReLu	-15.87	0.01

Table 53

Distributional DQN performance, using ADAM learning rates, and decay values. The hyperparameters used were: batch size 32; gamma 0.99; update target network period 50; network layers 3; hidden neuron size 128; Uniform (std 0.005) parameter initialization function; support min and max values –25 and 63 respectively

Lr & Decay		**Level Four**	
		Reward	Loss
0.0005; 1e-5	Tanh	-2.45	0.04
	ReLu	-7.3	0.03
0.0001; 1e-7	Tanh	-4.12	0.02
	ReLu	-15.19	0.03
0.001; 1e-4	Tanh	9.1	0.04
	ReLu	-12.62	0.02
0.005; 1e-3	Tanh	1.84	0.04
	ReLu	-14.19	0.02

Table 54

Rainbow DQN performance using different ADAM learning rates and decay values. The hyperparameters used were: batch size 32; gamma 0.99; update target network period 50; 1 input layer, 2 Value network layers, 2 Advantage network Layers; hidden neuron size 128; Uniform (std 0.005) parameter initialization function for regular layers, and Xavier initialization function for the noisy layers; N-step 3; support min and max values –25 and 63 respectively; Beta 0.6; Alpha 0.2; Sigma 0.05; Noisy Value network layer

Lr & Decay		**Level Four**	
		Reward	Loss
0.0005; 1e-5	Tanh	15.85	0.01
	ReLu	-9.33	0.01
0.0001; 1e-7	Tanh	22.2	0.01
	ReLu	-9	0
0.001; 1e-4	Tanh	3.59	0
	ReLu	-14.3	0
0.005; 1e-3	Tanh	-2.89	0
	ReLu	-8.72	0.01

Table 55

RS performance. The hyperparameters used were: population size 76; noise std 0.03; Uniform (std 0.005) parameter initialization function

	Level Four	
	Population Reward	Best Reward
Tanh	-1.57	55.48
ReLu	1.46	53.22

Table 56

RS-NS performance. The hyperparameters used were: population size 76; noise std 0.03; Uniform (std 0.005) parameter initialization function; novelty relevance 0.2

	Level Four	
	Population Reward	Best Reward
Tanh	-5.4	43.36
ReLu	0.44	39.97

Table 57

CMA-ES performance, using different ADAM learning rates and decay values. The hyperparameters used were: population size 50; noise std 0.015; network layers 3; hidden neuron size 256; Uniform (std 0.005) parameter initialization function.

Lr & Decay		Level Four	
		Population Reward	Best Reward
0.01; 1e-3	Tanh	-14.06	-13.91
	ReLu	-14.04	-13.89
0.01; 1e-5	Tanh	-14.05	-13.9
	ReLu	-14.04	-13.89
0.005; 1e-4	Tanh	-14.08	-13.7
	ReLu	-14.04	-13.94
0.005; 1e-6	Tanh	-14.04	-13.99
	ReLu	-14.04	-13.94

Table 58

CMA-ES-NS performance, using different ADAM learning rates and decay values. The hyperparameters used were: population size 50; noise std 0.015; network layers 3; hidden neuron size 256; Uniform (std 0.005) parameter initialization function; novelty relevance 0.5

Lr & Decay		**Level Four**	
		Population Reward	Best Reward
0.01; 1e-3	Tanh	21.81	37.13
	ReLu	8.31	22.81
0.01; 1e-5	Tanh	14.62	27.7
	ReLu	15.33	30.55
0.005; 1e-4	Tanh	6.92	27.43
	ReLu	10.62	27.68
0.005; 1e-6	Tanh	19.07	37.01
	ReLu	9.93	30.19

Table 59

GA performance. The hyperparameters used were: population size 76; noise std 0.015; elitism size 3; tournament size 10; Uniform (std 0.005) parameter initialization function

	Level Four	
	Population Reward	Best Reward
Tanh	12.79	58.09
ReLu	14.1	56.78

Table 60

GA-NS performance. The hyperparameters used were: population size 76; noise std 0.015; elitism size 3; tournament size 10; Uniform (std 0.005) parameter initialization function; novelty relevance 0.5

	Level Four	
	Population Reward	Best Reward
Tanh	21.46	56.91
ReLu	19.8	51.89

Table 61

NEAT performance. The hyperparameters used were: population size 50; noise std 0.015; Uniform (std 0.005) parameter initialization function; the remaining hyperparameters can be seen in Table 37.

	Level Four	
	Population Reward	Best Reward
Tanh	26.62	48.85
ReLu	27.19	49.5

Table 62

NEAT-NS performance. The hyperparameters used were: population size 50; noise std 0.015; Uniform (std 0.005) parameter initialization function; novelty relevance 0.5; the remaining hyperparameters can be seen in Table 37

	Level Four	
	Population Reward	Best Reward
Tanh	-8.31	55.87
ReLu	-3.08	57.27

Appendix B

Figure 23

Average population fitness over each generation for all Neuroevolution algorithms on level one (a). This graphic represents a smoothed version using the rolling average window size of three. The detailed version of this results that includes all algorithm information can be found at: https://github.com/rorix14/Thesis-Project/tree/main/Graphs

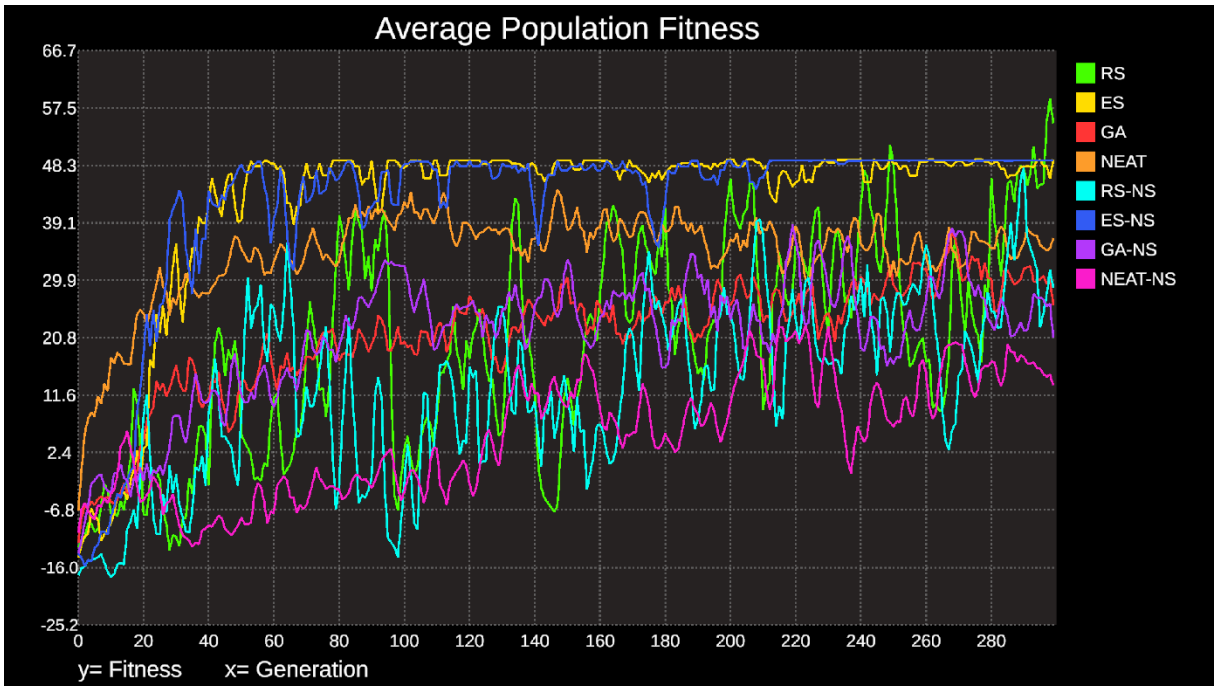


Figure 24

Average population fitness over each generation for all Neuroevolution algorithms on level two (b). This graphic represents a smoothed version using the rolling average window size of three. The detailed version of this results that includes all algorithm information can be found at:

https://github.com/rorix14/Thesis-Project/tree/main/Graphs

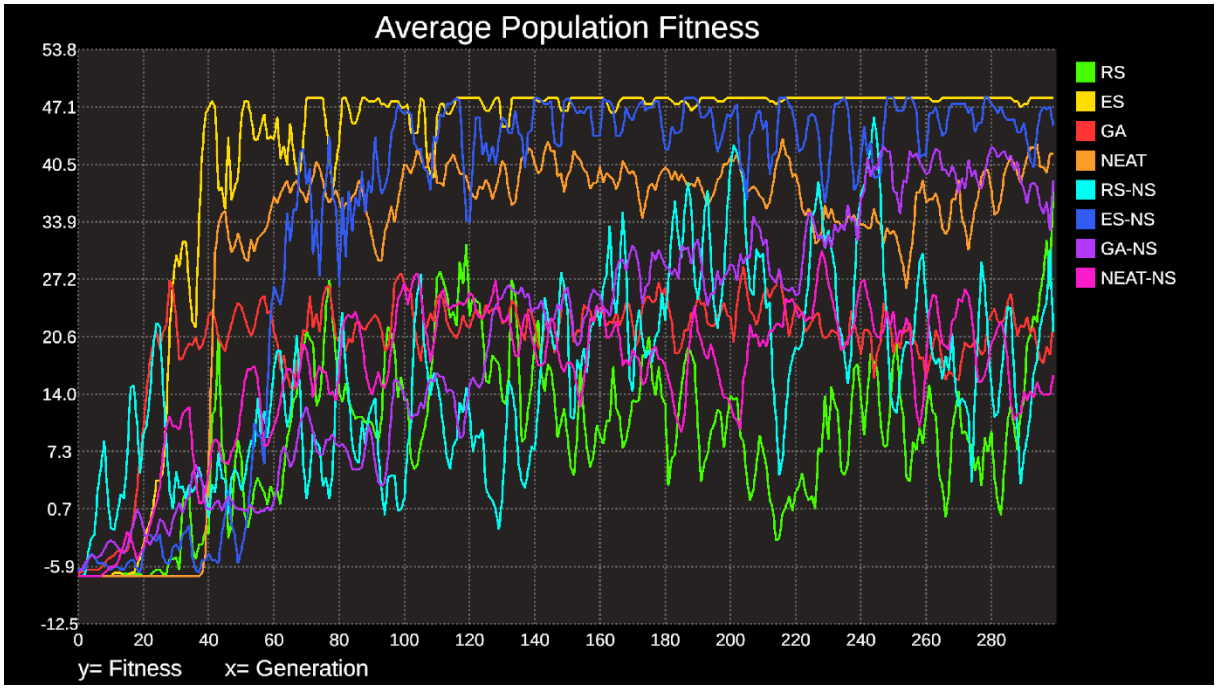


Figure 25

Average population fitness over each generation for all Neuroevolution algorithms on level three (c). This graphic represents a smoothed version using the rolling average window size of three. The detailed version of this results that includes all algorithm information can be found at: https://github.com/rorix14/Thesis-Project/tree/main/Graphs

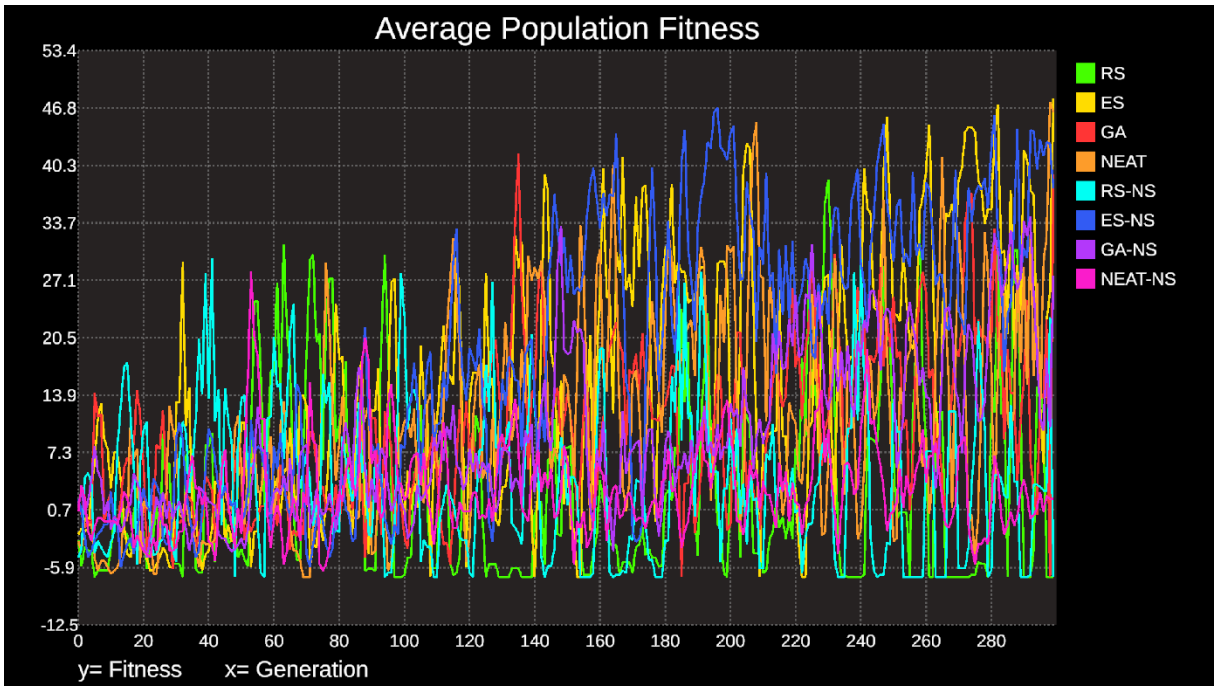


Figure 26

Average population fitness over each generation for all Neuroevolution algorithms on level four (d). This graphic represents a smoothed version using the rolling average window size of three. The detailed version of this results that includes all algorithm information can be found at: https://github.com/rorix14/Thesis-Project/tree/main/Graphs

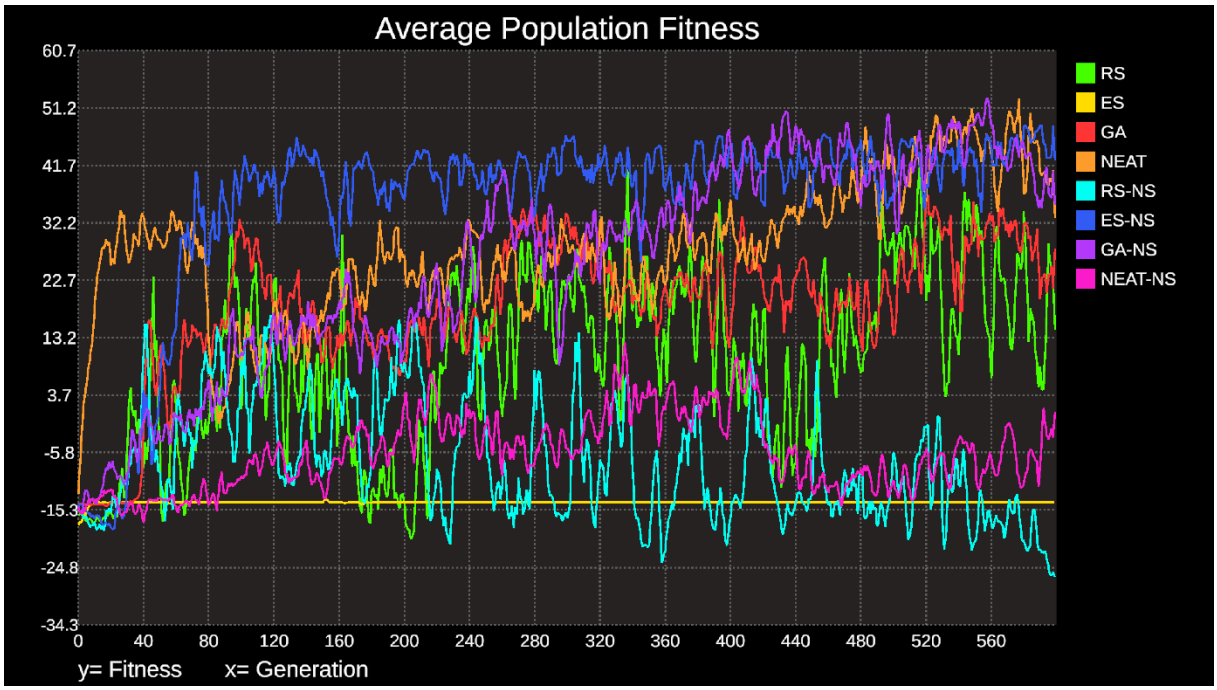


Figure 27

Best individual fitness over each generation for all Neuroevolution algorithms on level one (a). This graphic represents a smoothed version using the rolling average window size of three. The detailed version of this results that includes all algorithm information can be found at: https://github.com/rorix14/Thesis-Project/tree/main/Graphs

Figure 28

Best individual fitness over each generation for all Neuroevolution algorithms on level two (b). This graphic represents a smoothed version using the rolling average window size of three. The detailed version of this results that includes all algorithm information can be found at: https://github.com/rorix14/Thesis-Project/tree/main/Graphs

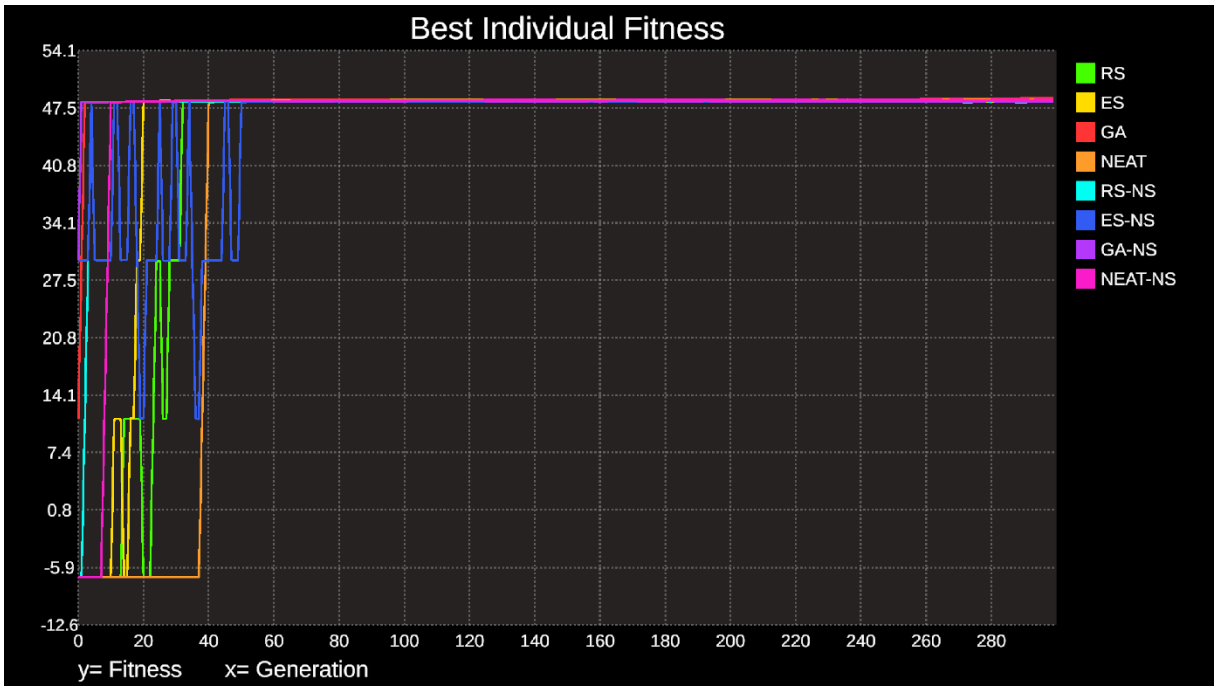


Figure 29

Best individual fitness over each generation for all Neuroevolution algorithms on level three (c). This graphic represents a smoothed version using the rolling average window size of three. The detailed version of this results that includes all algorithm information can be found at: https://github.com/rorix14/Thesis-Project/tree/main/Graphs

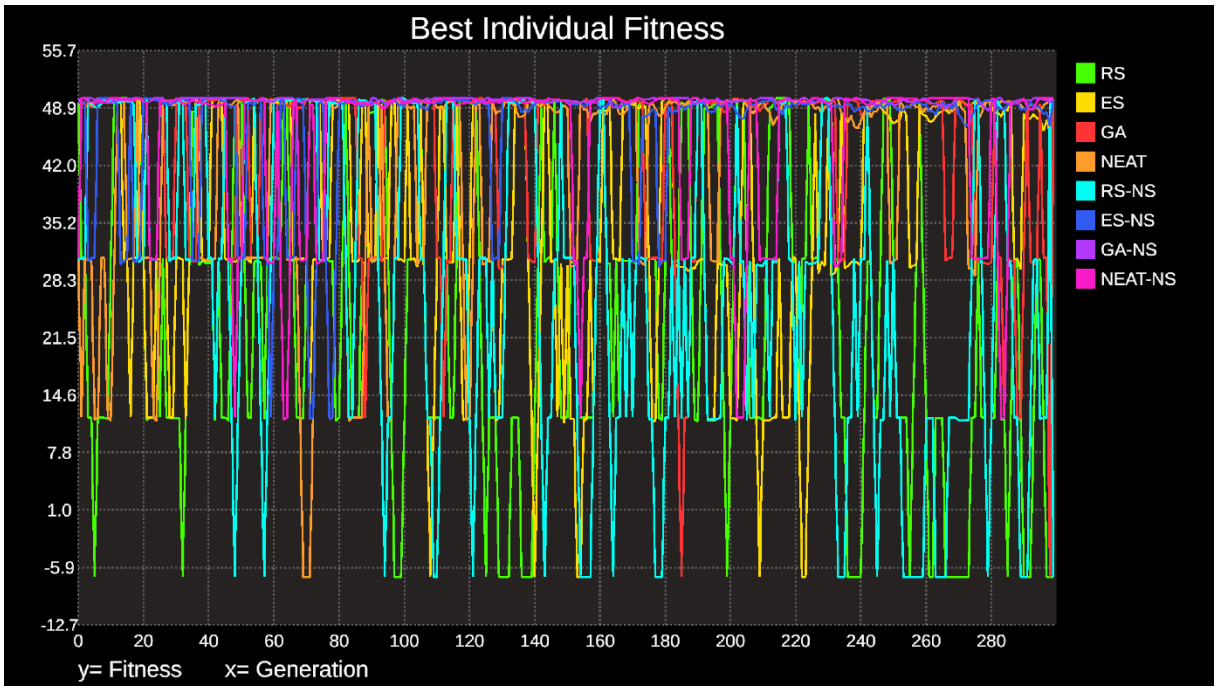


Figure 30

Best individual fitness over each generation for all Neuroevolution algorithms on level four (d). This graphic represents a smoothed version using the rolling average window size of three. The detailed version of this results that includes all algorithm information can be found at: https://github.com/rorix14/Thesis-Project/tree/main/Graphs

Figure 31

Rewards over each episode for DQN and its' extensions algorithms on level one (a). This graphic represents a smoothed version using the rolling average window size of three. The detailed version of this results that includes all algorithm information can be found at: https://github.com/rorix14/Thesis-Project/tree/main/Graphs

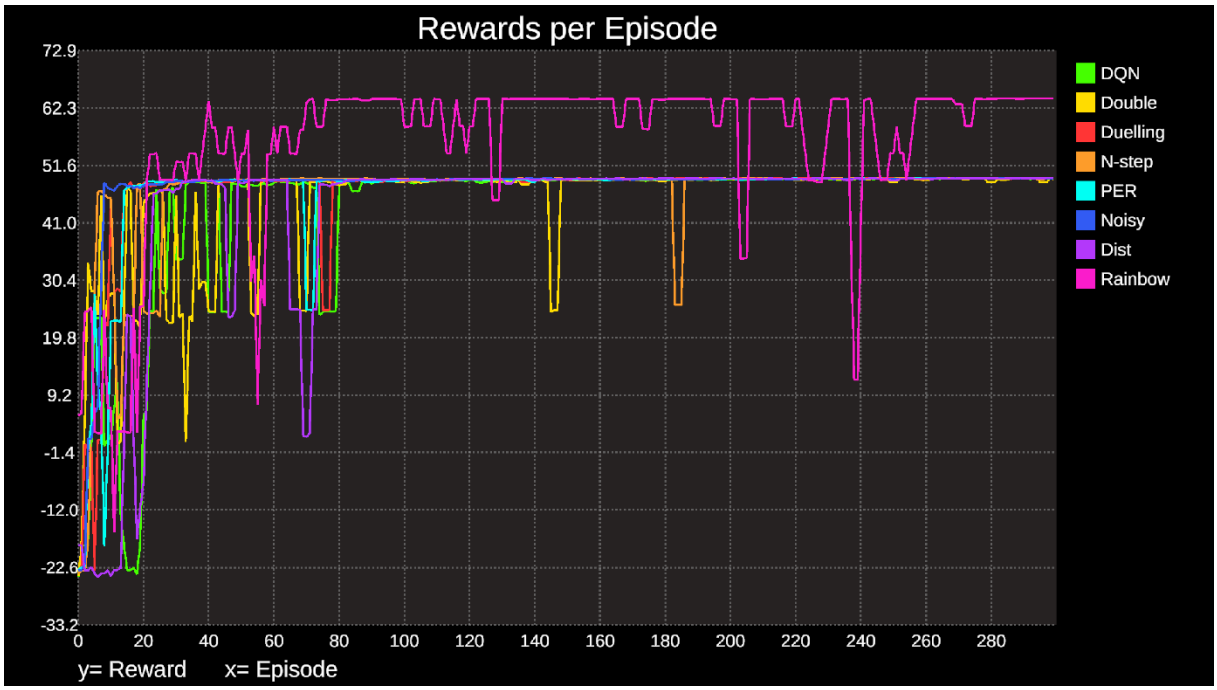


Figure 32

Rewards over each episode for DQN and its' extensions algorithms on level two (b). This graphic represents a smoothed version using the rolling average window size of three. The detailed version of this results that includes all algorithm information can be found at: https://github.com/rorix14/Thesis-Project/tree/main/Graphs

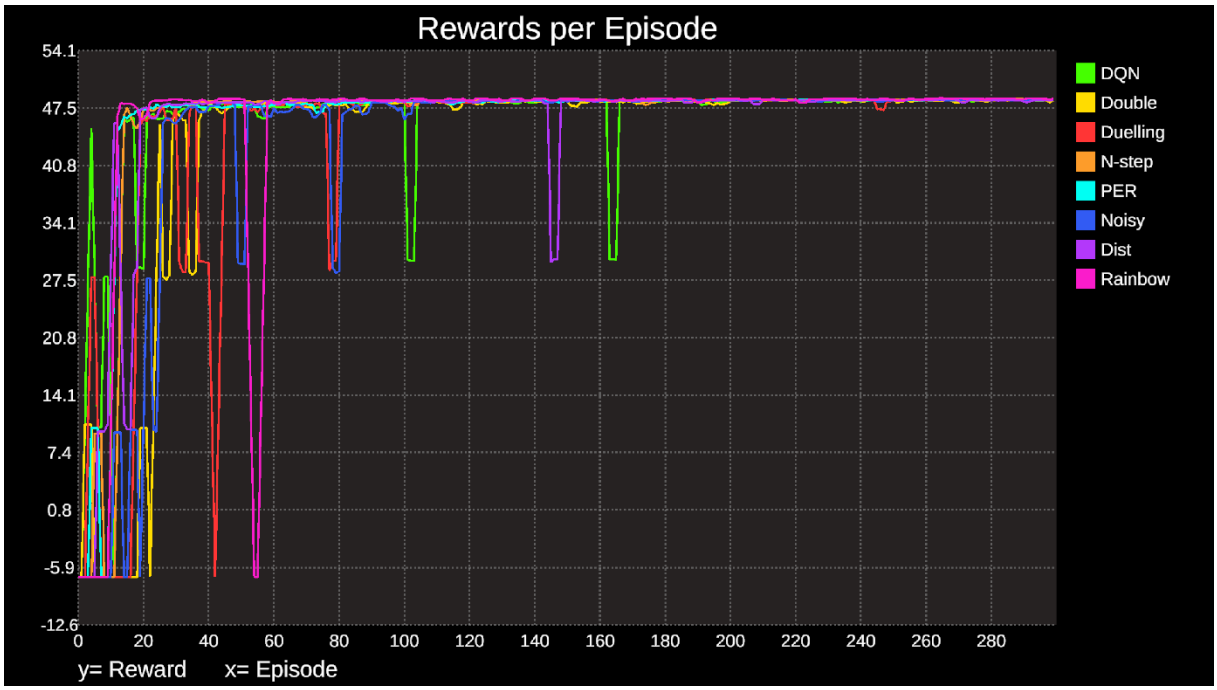


Figure 33

Rewards over each episode for DQN and its' extensions algorithms on level three (c). This graphic represents a smoothed version using the rolling average window size of three. The detailed version of this results that includes all algorithm information can be found at: https://github.com/rorix14/Thesis-Project/tree/main/Graphs

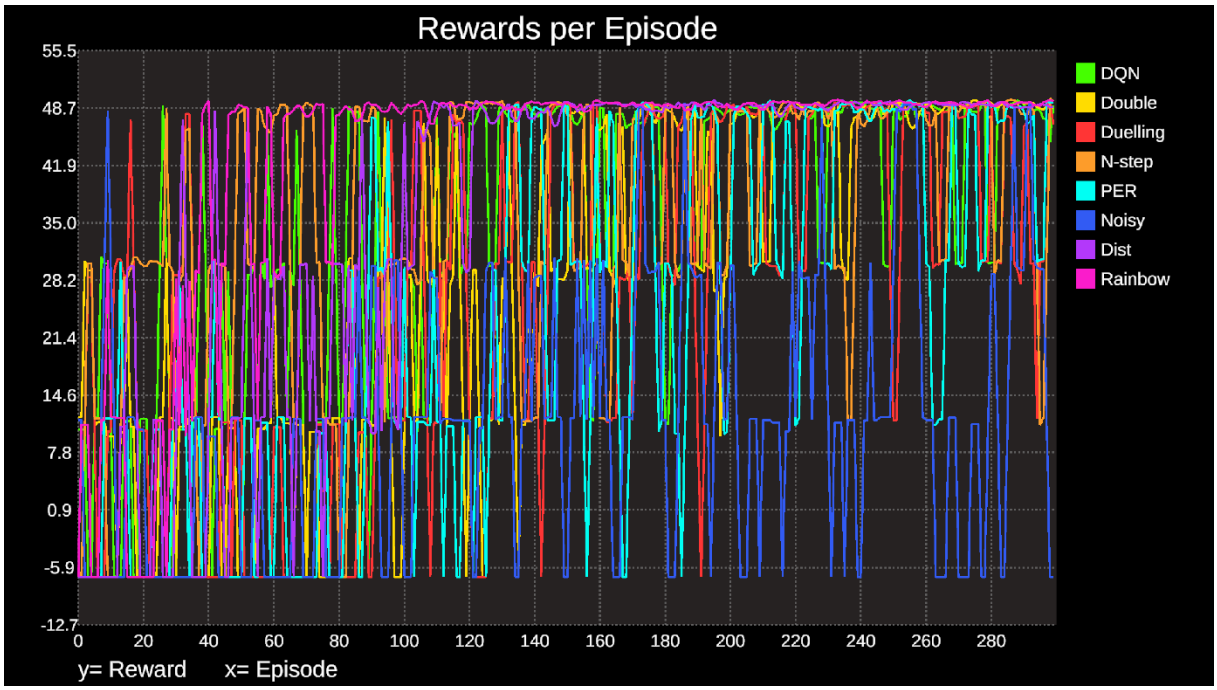


Figure 34

Rewards over each episode for DQN and its' extensions algorithms on level four (d). This graphic represents a smoothed version using the rolling average window size of three. The detailed version of this results that includes all algorithm information can be found at: https://github.com/rorix14/Thesis-Project/tree/main/Graphs

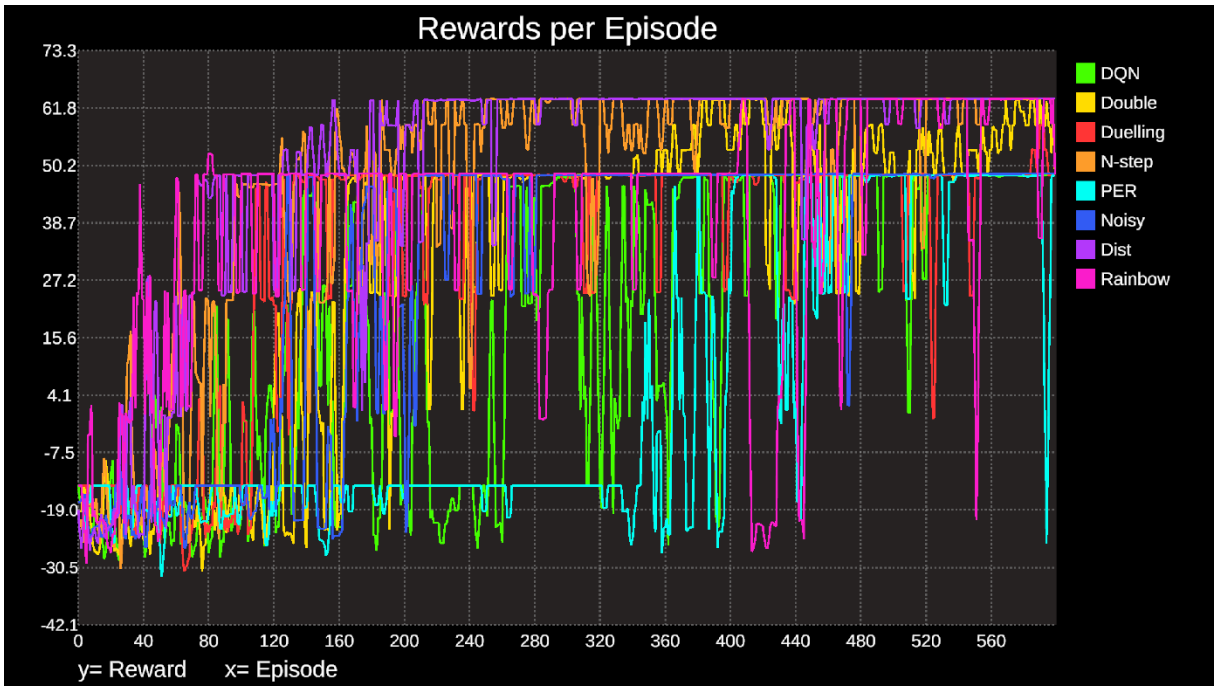


Figure 35

Loss over each episode for DQN and its' extensions algorithms on level one (a). This graphic represents a smoothed version using the rolling average window size of three. The detailed version of this results that includes all algorithm information can be found at: https://github.com/rorix14/Thesis-Project/tree/main/Graphs

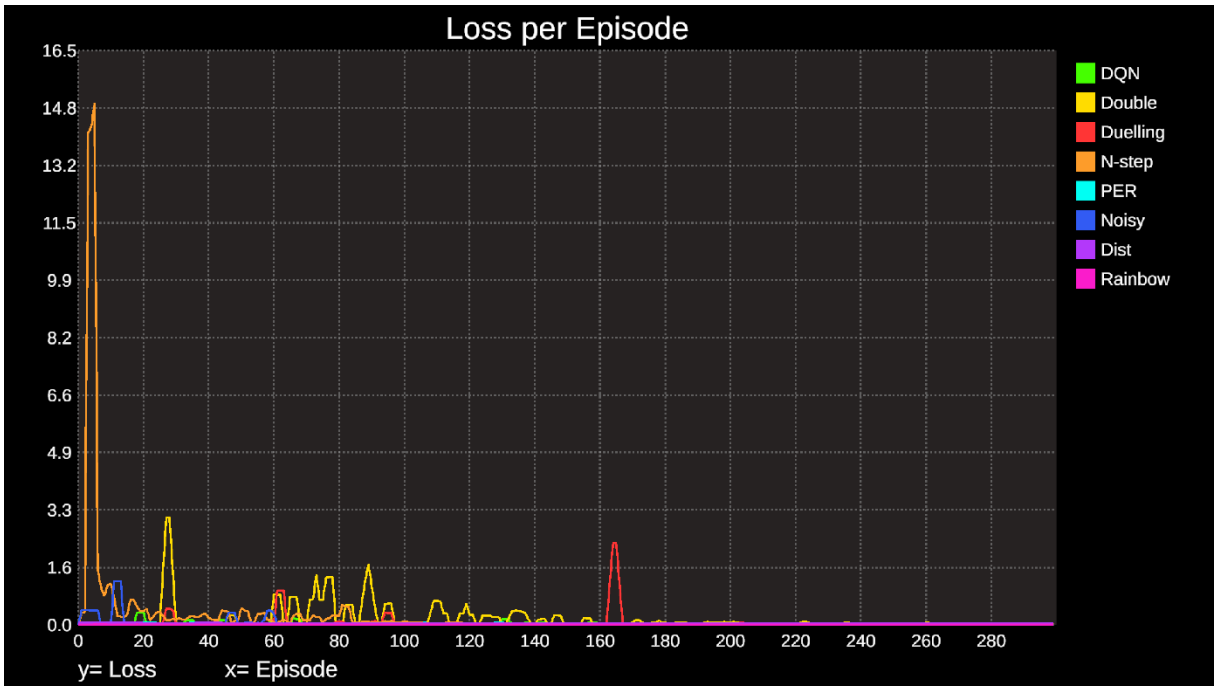


Figure 36

Loss over each episode for DQN and its' extensions algorithms on level two (b). This graphic represents a smoothed version using the rolling average window size of three. The detailed version of this results that includes all algorithm information can be found at: https://github.com/rorix14/Thesis-Project/tree/main/Graphs

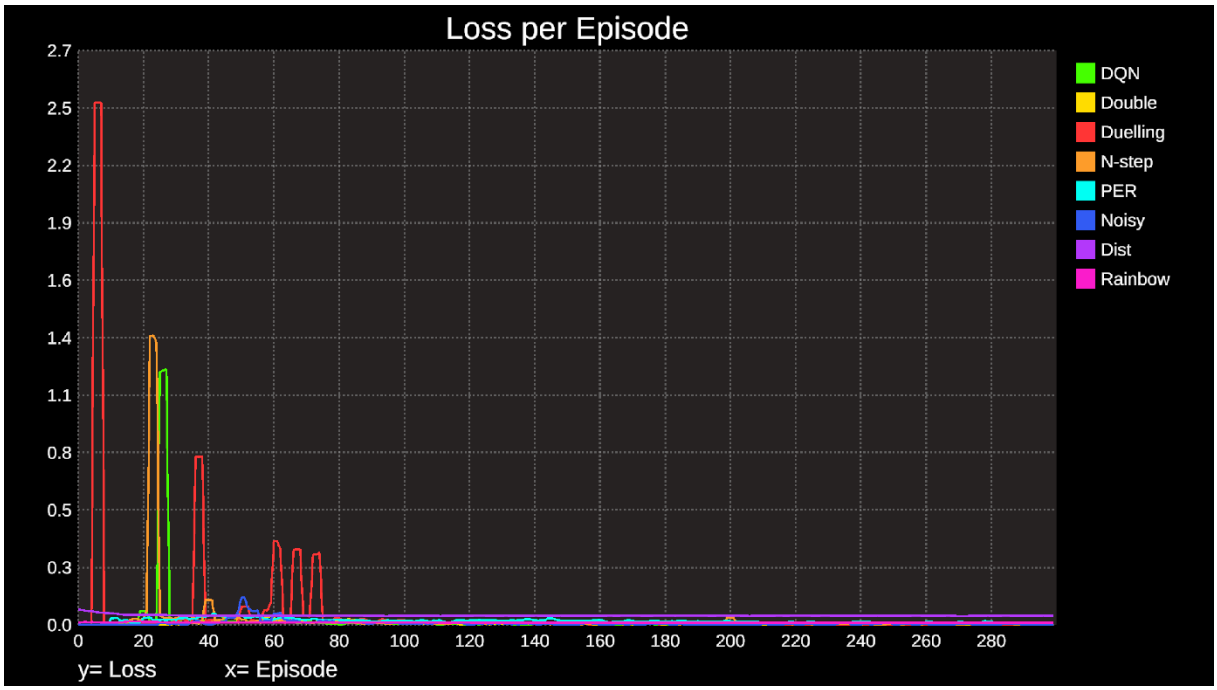


Figure 37

Loss over each episode for DQN and its' extensions algorithms on level three (c). This graphic represents a smoothed version using the rolling average window size of three. The detailed version of this results that includes all algorithm information can be found at: https://github.com/rorix14/Thesis-Project/tree/main/Graphs

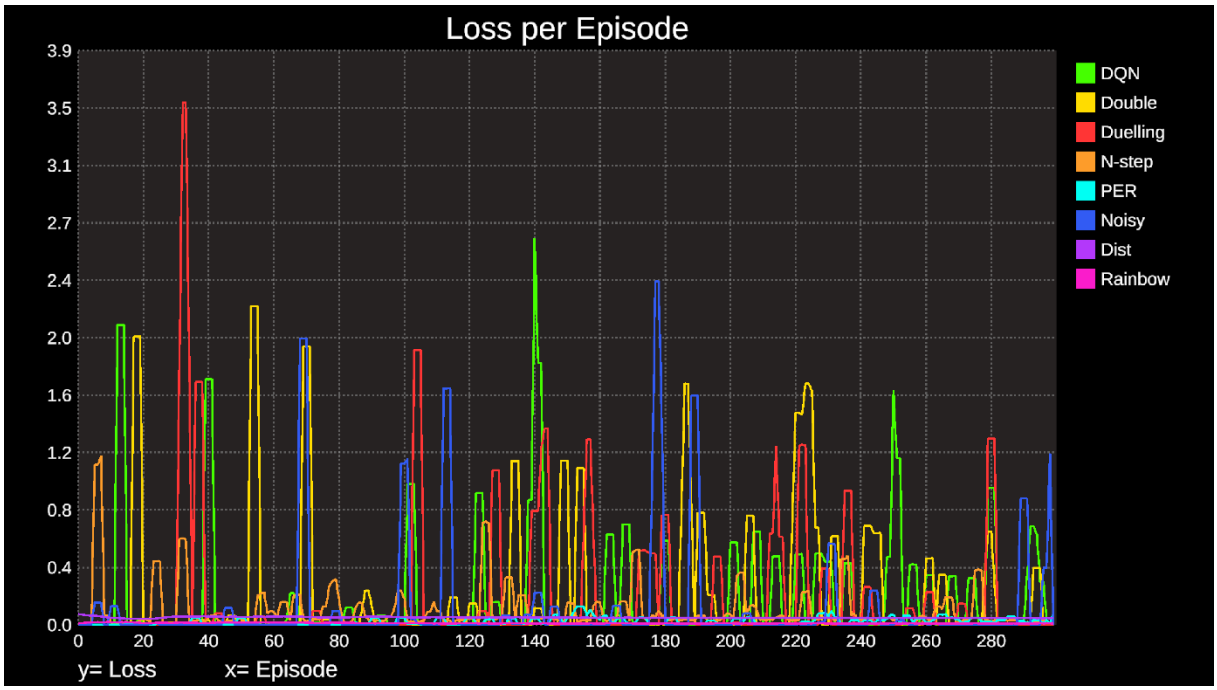


Figure 38

Loss over each episode for DQN and its' extensions algorithms on level four (d). This graphic represents a smoothed version using the rolling average window size of three. The detailed version of this results that includes all algorithm information can be found at: https://github.com/rorix14/Thesis-Project/tree/main/Graphs