UCLouvain

École polytechnique de Louvain

Deep Q-Learning Algorithm: An Experimental Investigation of the Impact of Environmental Configurations on Learning Complexity and Performance in a 2D Video Game

Author: **Corentin VERMEULEN**
Supervisors: **Francois-Xavier STANDAERT, Julien HENDRICKX**
Reader: **Brieuc PINON**
Academic year 2023–2024
Master [120] in Data Science: Information technology

Abstract

Artificial Intelligence (AI), specifically reinforcement learning (RL), has significantly advanced through the development of Deep Q Networks (DQNs). DQNs effectively combine Q-learning with deep neural networks to manage high-dimensional state spaces. Whilemost studies focus on DQN algorithm architectures, this thesis investigates the impact of different environmental configurations on learning complexity and performance.

The research involved training a DQN agent in various environmental settings to assess its learning efficiency and performance. The environments included both random and fixed conditions, varied action spaces, and different levels of uncertainty in action execution. The Flappy Bird game environment, implemented via OpenAI Gym, served as the testbed for these experiments.

The findings indicate that DQN agents trained in random conditions outperformed those trained in fixed conditions, demonstrating better generalization capabilities. Enlarging the action space was observed to slow down learning and reduce performance. Furthermore, introducing uncertainty in action execution had minimal impact on learning velocity when the uncertainty did not alter game dynamics. However, significant changes in game dynamics due to uncertainty drastically reduced learning speed and performance.

The study concludes that environmental configurations substantially influence the learning and performance of DQN agents. Training in varied conditions enhances generalization, while increased action spaces and dynamic uncertainties pose challenges to learning efficiency. These insights can guide the development of more robust reinforcement learning systems capable of adapting to diverse real-world environments.

Acknowledgment

I would like to express my deepest gratitude to my supervisors, *Julien Hendrickx* and *Francois-Xavier Standaert*, for their invaluable guidance and support throughout the preparation of this master's thesis. Despite their demanding schedules, they generously dedicated their time to provide assistance and insightful feedback, contributing significantly to the development of this work.

I am sincerely grateful to *Brieuc Pinon* for taking the time to meet with me, answering my questions, and providing me with an invaluable overview of the general reinforcement learning literature. I am also profoundly thankful for his willingness to serve as the reader for this thesis.

I am also indebted to *Guillaume Fontaine* for their unwavering support and camaraderie during our parallel journeys in tackling similar thesis subjects. Our exchanges, ranging from technical discussions to emotional support, have been immensely beneficial and enriching.

Additionally, I extend my thanks to *Consortium des Équipements de Calcul Intensif (CECI)* for graciously allowing me to utilize their servers for conducting the computational experiments integral to this research. Their generosity has played a pivotal role in the successful execution of this project.

I wish to extend my sincere thanks to all individuals who have contributed, directly or indirectly, to the completion of this thesis. Your support has been instrumental in this endeavor.

Contents

Chapter 1

Introduction

Artificial Intelligence (AI) is the field of computer science dedicated to creating systems capable of performing tasks that typically require human intelligence, such as problem-solving, understanding natural language, and recognizing patterns. Machine Learning (ML), a subset of AI, focuses on developing algorithms that enable computers to learn from data and make predictions or decisions with minimal human intervention. Among the various branches of ML, reinforcement learning (RL) plays a crucial role due to its unique approach to solving complex decision-making problems.

Reinforcement learning involves training an agent to make decisions in an environment to maximize cumulative rewards received as feedback. It has been used in many fields such as robotics, recommendation systems, finance, and game solving driving innovation and efficiency in these areas.

Various reinforcement learning algorithms have been developed, each employing distinct methodologies. Among these, Q-learning stands out as a widely used approach, notable for its ability to learn from experiences without prior knowledge of the environment. An advancement of Q-learning is Deep Q-Learning (DQN), which utilizes neural networks to approximate the Q-value function, contrasting with the tabular approach of classical Q-learning. DQN has demonstrated effectiveness, particularly in handling large state-action spaces.

Although DQN demonstrates impressive performance across various applications and has undergone numerous modifications to enhance its effectiveness and learning process, it is often used as a black box, with few studies examining its complexity like S. Zhang et al., 2023. The studies aimed at improving its performance typically focus on the algorithm's architecture like Hessel et al., 2017. Until now, there has been a notable absence of studies explicitly investigating the influence of the

environment on the DQN algorithm, particularly concerning learning complexity. Learning complexity is contingent upon various dimensions, such as the size of the state space, the dimension of the action space, and the stability of the environment. Given the lack of prominent theoretical studies in this area, the initial approach involves an experimental inquiry to examine how learning dynamics change when modifying these environmental parameters.

Understanding environment configuration in reinforcement learning is vital as it profoundly affects algorithm performance, efficiency, and generalization. By comprehending the nuances of the environment, developers can design agents capable of adapting to diverse scenarios, effectively managing uncertainty, and performing efficiently. This comprehension is indispensable for creating resilient RL solutions capable of navigating real-world complexities and enhancing overall algorithm design.

Proposition

This thesis aims to explore how diverse environment configurations influence DQN reinforcement learning algorithms. It focuses on two key outcomes: the speed of learning and the quality of learning. The research questions investigate the learning velocity and performance of DQN agents in relation to various configurations. Specifically, the study examines the impact of fixed versus random environment designs, evaluates learning complexity with different action space expansions, and compares performance when uncertainty is introduced in action execution. These inquiries lead to three distinct research questions addressing specific aspects of environment configuration.

- *Context to learn*: How does the choice between a fixed environment configuration, where the agent encounters the same game scenario in each training session, and a random environment configuration, where the agent faces a different scenario each time, impacts the the learning efficiency and the performance of a DQN agent?

- *Action space expansion*: What is the effect on learning complexity in DQN algorithms when the action space is expanded, and how does this expansion influence the agent's ability to make informed decisions within a larger set of possible actions?

- *Uncertain action outcome*: How does introducing uncertainty about the agent's action execution affect the performance of the DQN agent, and how does it adapt to navigate environments with varying levels of action uncertainty?

Specifically to address these research questions, we trained a DQN agent to play a 2D video game as commonly done to benchmark DQN performances (Hessel et al., 2017).
Using a 2D game implemented in Python, we have the flexibility to modify the source code and create the desired environmental conditions for each research question. We tracked the agent training to identify trends and derive answers using specific outcome measures, focusing on learning velocity, agent performance and stability to define the optimal configuration

Summary of results

Concerning best context to learn, both configurations, fixed and random, provided the same training velocity and average training performances and. However, there was a slight difference in late learning performance. In a test environment, the agent trained under fixed conditions had more difficulty generalizing, while the agent trained in a random configuration achieved higher performance.
Introducing more actions to the action space slowed the learning and unexpectedly reduced the agent's performance.
Introducing different levels of uncertainty in action execution led to two distinct behaviours, when the game dynamics do not change, the uncertainty did not significantly impact the agents' performance. On the contrary, the performance dropped massively with increasing uncertainty when game dynamic is modified. In testing environments, we observed a strong transfer of skills from agents trained in highly uncertain configurations to less uncertain environments, but not in the other direction.

Outline

After this introduction, Chapter 2 will provide a background on deep learning, reinforcement learning, and deep Q-learning. Readers familiar with these concepts may skip this section. Chapter 3 will detail the implementation of the game environment and DQN agent, including the development of specific performance measures. Subsequently, the three research questions will be addressed in Chapters 4, 5, and 6 each following a consistent structure comprising methodology, hypotheses, experiment descriptions, and results. Finally, Chapter 7 will integrate the results in the actual knowledge in the field, highlight strengths and limitations of the current work, and outline future work to further advances in the field

Chapter 2

Background

This sections focuses on introducing the foundational concepts that lead to the *Deep Q-Learning* algorithm, which combines *deep learning* and *reinforcement learning*. It then presents current improvements added to the DQN algorithm and introduces a triad causing divergence met in DQN.

Context

Artificial intelligence (AI) is a broad concept with no clear boundaries. *Machine learning* (ML) is a subset of AI that focuses on developing algorithms that enable machines to learn from data and improve their performance on a specific task over time like making prediction and/or decisions (Li, 2018).
In ML, there are three primary paradigms: *Supervised Learning*, which consist of learning from labelled data, *Unsupervised Learning*, which consist of learning with no labelled data and finally *Reinforcement Learning* which is about learning to make decisions from feedback.

2.1 Deep learning

Deep learning (DL) is a sub-field of machine learning that employs artificial neural networks to model and solve complex problems, drawing inspiration from the structure and function of human brain neurons as developed in Rosenblatt, 1962. It can be applied either in supervised or unsupervised applications, but it can also be adapted for reinforcement learning (Figure 2.1).

Deep learning has achieved significant success in various fields characterised by large amounts of data and complex tasks. Its state-of-the-art methods in image recognition, natural language processing, speech recognition, computer vision, self-driving

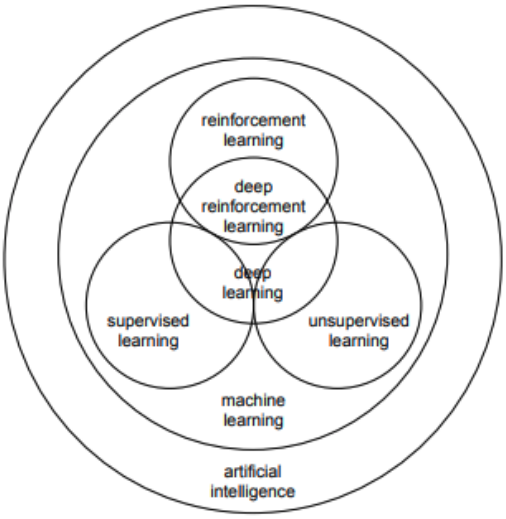

Figure 2.1: AI subdomain picture illustrating the relationships between Machine Learning (ML), Supervised Learning, Unsupervised Learning, Reinforcement Learning, Deep Learning, and Deep Reinforcement Learning. (Li, 2018)

cars, financial fraud detection, and many others demonstrating its versatility and impact across diverse domains. Deep learning has the potential to revolutionise numerous aspects of society and industry.

2.1.1 Perceptron and neuron

The key component of DL is the perceptron, which is the simplest form of an artificial neural network model, consisting of a single layer of neurons, introduced by Rosenblatt, 1962 as a single linear binary classifier. As shown on figure 2.2 it only takes binary input and produces binary output, akin to biological neurons. The perceptron can be expressed mathematically as:

$$y = \begin{cases} \mathbf{1} \text{ if } \sum_i x_i * w_i > b \\ \mathbf{0} \text{ otherwise} \end{cases} \tag{2.1}$$

where y is the binary output, x_i the input i od the input vector, w_i the associated weight and b the threshold to activate the perceptron.

However, it has evolved into the artificial neuron, a more versatile unit. As shown on figure 2.3, an artificial neuron accepts input signals (not necessarily

binary), each associated with a weight, which are then processed through an activation function to generate an output signal. Neurons can be organised into layers to form neural networks with diverse functionalities. The neuron can be expressed mathematically as:

$$z_j = \sum_i x_{ij} * w_{ij} \tag{2.2}$$

$$y_j = \phi(z_j) \tag{2.3}$$

where z_j is the linear combination of input signals, x_{ij} the input i of the signal vector, w_{ij} the associated weight vector, y_j the output vector and ϕ is the activation function.

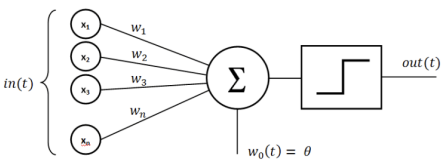

Figure 2.2: Perceptron unit (Kassel, 2021)

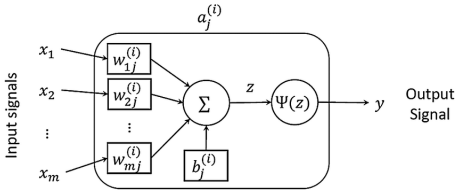

Figure 2.3: Neuron unit (Tai et al., 2021)

Activation functions play a crucial role in neural networks by introducing non-linearity, allowing them to capture complex patterns from the input data. It's essential for activation functions to be differentiable to support backpropagation, a key algorithm discussed later. Below are some of the most popular activation functions:

- **Threshold Function**: At the core of the binary perceptron, the threshold activation function produces a binary output. If the input value is positive, it outputs a binary "1"; otherwise, it outputs a "0". To define a threshold other than zero, simply incorporate the desired threshold value into the bias term within the linear combination computation. Its main applications are in binary classification and implementing logical operations. It can also be used for thresholding in image segmentation.
  It's defined by:
  $$\phi(z_j) = \begin{cases} \mathbf{1} \text{ if } z_j \geq 0 \\ \mathbf{0} \text{ otherwise} \end{cases} \tag{2.4}$$

- **Sigmoid**: The sigmoid function squashes the output of a neuron to the range $[0, 1]$ making it suitable for binary classification tasks where the output represents probabilities. It is commonly employed in the output layer of neural

networks to provide the probabilities of each output value. One advantage of the sigmoid function is its ease of differentiation.
It's defined by:

$$\phi(z_j) = \frac{1}{1 + e^{-z_j}} \tag{2.5}$$

- **Softmax**: The softmax function normalizes the outputs of the output layer into a probability distribution over multiple output nodes, rendering it suitable for multi-class classification tasks.
It's defined by:

$$\phi(z_j) = \frac{e^{z_j}}{\sum_i e^{-z_i}} \text{for } i = 1, 2, n \tag{2.6}$$

- **Hyperbolic tangent (Tanh)**: Tanh squashes the output of a neuron to the range [-1, 1], allowing it to capture both positive and negative values. It is commonly used in hidden layers of neural networks.
It's defined by:

$$\phi(z_j) = \frac{e^{z_j} - e^{-z_j}}{e^{z_j} + e^{-z_j}} \tag{2.7}$$

- **Rectified linear unit (ReLU)**: The ReLU activation function is both simple and effective. It facilitates the smooth flow of gradients during back-propagation, mitigating the vanishing gradient problem (explained later). Consequently, it has become the most commonly used activation function within the deep learning community. It's defined by:

$$\phi(z_j) = max(0, z_j) \tag{2.8}$$

  Several variations of the ReLU activation function have been proposed, but none of them have entirely replaced the classical ReLU due to inconsistencies in the benefits offered by the variants(Ramachandran et al., 2017).

2.1.2 Neural network and Deep Network

We can combine and interconnect multiple neurons organized into layers to form a neural network. Neural networks, also known as shallow networks or feedforward neural networks, typically consist of an input layer, a few hidden layers, and an output layer as depicted by figure 2.4a. They map input values to output values by combining multiple simple functions to capture complex patterns. However, when a network comprises multiple hidden layers, we refer to it as a deep neural network, see 2.4b. This depth enables the network to effectively tackle more complex tasks.

The Universal Approximation Theorem (Hornik et al., 1989) states that any

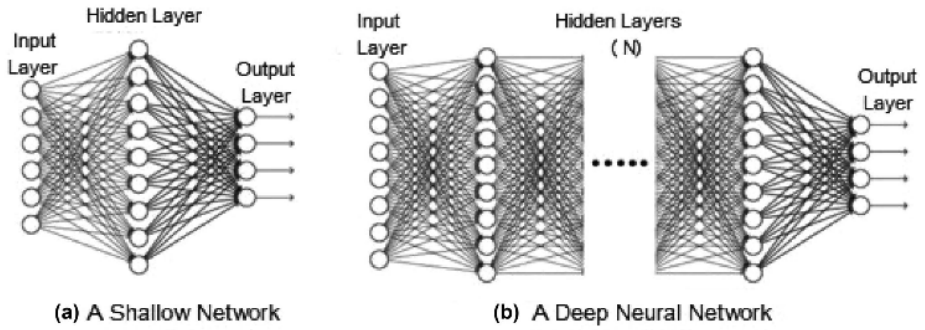

Figure 2.4: Comparison between neural network **a** and deep neural network **b** (Sarker, 2021)

continuous function can be approximated by a neural network with a single hidden layer, provided that the layer contains a sufficiently large number of neurons. However, this may not always be the most efficient approach, as increasing the number of neurons in a single layer can lead to computational inefficiencies. Alternatively, using multiple layers can achieve the same approximation with fewer total neurons, as deeper architectures can capture hierarchical representations of the data.
Designing the number of neurons and layers, which constitute the architecture of the neural network, is essential as it directly impacts the network's capability to learn complex patterns and generalize to unseen examples. Selecting the appropriate number of neurons and layers requires balancing model complexity with computational resources, training time, and data efficiency. Furthermore, finding an architecture that generalizes well to unseen data while avoiding overfitting is crucial for achieving optimal performance.

Neural networks, including deep neural networks, have the capability to map various types of input data to corresponding outputs. For instance, in speech recognition tasks, they can translate audio signals into text. Similarly, for translation tasks, they can convert French sentences into English sentences. In the context of image classification, neural networks map arrays of pixels to specific classes.

2.1.3 Forward propagation

The forward propagation or forward pass is the process where the data flows through the network from the input layers, through the hidden layers if any and finally to the output layers. In Figure 2.3, a_j^i represents the activated value of

neuron j from layer i, which mathematically corresponds to

$$z_j^i = \sum_k a_k^{i-1} * w_k^i \tag{2.9}$$

$$a_j^i = \phi(z_j^i) \tag{2.10}$$

where z_j^i is the linear combination from previous layer, a_k^{i-1} are the activated values from the previous layer or input value if it's the input layer and w_k^{i-1} are the associated weight.
Put simply, at each layer (excluding the input layer), we calculate the weighted sum of the activated values from the preceding layers. This process allows us to derive a representation of the input from the previous layer. The whole layer can be put into a vector as:

$$Z^i = A^{i-1}W^i \tag{2.11}$$

$$A^i = \phi(Z^i) \tag{2.12}$$

Once we traverse the entire network, we obtain the output values at the end. From these, we can compute the loss. The selection of the loss function depends on the task at hand. For regression tasks, Mean Squared Error (MSE) is commonly used. For classification tasks, Cross-entropy loss is often employed. In problems involving probability distribution comparison, such as in generative models, Kullback-Leibler Divergence (KL Divergence) is a common choice (Hall, 1987).

2.1.4 Back propagation

To ensure the network achieves the desired tasks, it is crucial to train and optimise the model. This involves updating the weights W_i of each layer to minimise the chosen loss function.
A common algorithm to update the weights is the backpropagation algorithm proposed by Rumelhart et al., 1986. After the forward propagation, the loss is computed and then the loss derivative can be computed in reverse, from the output layer to the input layer. Subsequently, the weights are updated based on this derivative to minimize the loss.

The back propagation allows to compute weight gradient with respect to the loss function L. To update the weights of layer i it's corresponding gradient are computed with chain rule:

$$\frac{\partial L}{\partial W^i} = \frac{\partial L}{\phi(Z^i)} \frac{\partial \phi(Z^i)}{\partial Z^i} \frac{\partial Z^i}{\partial W^i} \tag{2.13}$$

Weights are then updated with learning rate η as:

$$W^i \leftarrow W^i - \eta \frac{\partial L}{\partial W^i} \tag{2.14}$$

The learning rate controls the size of the step taken during optimization. A small learning rate would require many iterations to reach the optimum solution or may get stuck in a local minimum. Conversely, a too large learning rate could lead to large steps during optimization, potentially overshooting the minimum of the loss function and causing the optimization process to oscillate around it without converging. One solution to mitigate these issues is to adapt the learning rate during training using adaptive learning rate methods. One commonly used adaptive learning rate method is to adjusts the learning rate for each parameter based on the magnitude of past gradients as described by Adam Kingma and Ba, 2017.

Vanishing gradient problem

During backpropagation, weights are updated proportionally to their gradient value. When this gradient is close to 0, the weights are not updated, leading to suboptimal learning. The vanishing gradient problem refers to the phenomenon where the gradients of the loss function with respect to the parameters become extremely small as they are propagated backward through the network's layers. The gradient can become very small when the number of layers increases due to the nature of backpropagation, which is built on the chain rule of derivatives, involving the multiplication of gradients at each layer. Gradients can therefore quickly diminish to near zero.
To mitigate the vanishing gradient problem, various techniques can be employed:

- **ReLU Activation Functions**, ReLU units introduce non-linearity by only activating for positive inputs, allowing gradients to flow more freely during backpropagation compared to saturating activation functions like sigmoid or tanh.
- **Proper Weight Initialization** is crucial to the convergence of a neural network, Glorot and Bengio, 2010 propose an initialization scheme that brings substantially faster convergence.
- **Batch Normalization** allows the use of higher learning rates, requires less careful weight initialization, and makes the optimization landscape smoother, which helps achieve faster learning (Ioffe and Szegedy, 2015).

2.2 Reinforcement learning

Reinforcement learning (RL) is a sub-field of machine learning in which an agent learns to make decisions within an environment. It learns a policy to maximise the cumulative rewards received as feedback from the environment. The agent must learn optimal behaviour not only for immediate rewards but also for potential delayed rewards in the future. The process resembles a trial-and-error search, akin to a child discovering a new world. RL algorithms continuously explore and exploit actions to achieve a balance between learning and achieving high rewards over time.

Nowadays, RL is used in robotics, where new robots learn to accomplish complex tasks such as manipulating objects and navigating through complex environments (Lobbezoo et al., 2021). It is also used in recommendation systems, adapting recommendations based on user behavior (Zhao et al., 2021). Moreover, it finds application in finance for the development of automated trading strategies (Yang et al., 2021), controlling nuclear fusion plasma in a tokamak (Degrave et al., 2022), playing complex games at an equal human level or even super-human level such as Go with AlphaGo agent (Silver et al., 2016).

2.2.1 Key components of Reinforcement learning

Reinforcement learning relies on several fundamental elements that govern its functioning. These components form the backbone of the RL framework, shaping how agents interact with their environments and make decisions.

- **Agent**: This component is responsible for making decisions and interacting with the environment. It is the entity that learns from its actions and experiences.
- **Environment**: This refers to the external system or surroundings in which the agent operates. It includes everything that the agent can perceive and interact with.
- **State**: A state $s \in \mathcal{S}$ represents the current situation or condition of the environment. It encapsulates all relevant information needed for the agent to make decisions.
- **Action**: Actions $a \in \mathcal{A}$ are the choices or decisions that the agent can take in a specific state. These actions lead to transitions from one state to another.

- **Reward**: Rewards $R \in [-1, 1]$ are feedback signals that the agent receives for its actions in a given state. They indicate the desirability or utility of the action taken by the agent.
- **Policy**: A policy $\pi(s) : \mathcal{S} \to \mathcal{A}$ is the strategy or set of rules that the agent uses to select actions based on its current state and the perceived environment. It guides the decision-making process of the agent.

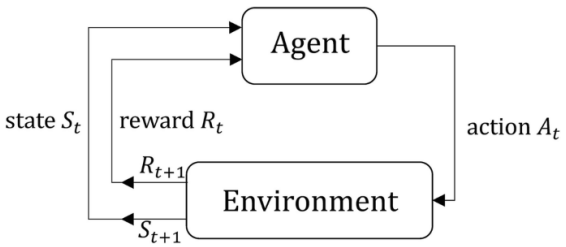

Figure 2.5: Environment and agent interactions

The figure 2.5 depicts the classical loop in RL, illustrating the interaction between the environment and the agent. As the agent selects and executes an action A_t based on the current state S_t, the environment is thus modified and the state transitions from S_t to S_{t+1}, yielding a feedback or reward R_{t+1}. Usually when the action is qualified as good, the agent receive a positive reward. The sequence of states, actions, rewards is called a *trajectory*: $\tau = (S_0, A_0, R_0, S_1, A_1, R_1, ...)$. It record the interactions between the agent and the environment.

Markov decision process

A Markov decision process (MDP) is a mathematical framework for modelling sequential decision-making by an agent. It extends Markov chains by incorporating the concepts of actions and rewards alongside states and transitions.

In the context of reinforcement learning, MDPs serve as the foundation for formulating and solving decision-making problems. Reinforcement learning algorithms utilise MDPs to learn optimal policies aiming to maximise the expected return, also known as cumulative reward, over time. By exploring various actions and observing the resulting rewards, the agent learns to make decisions that lead to the most favourable outcomes within the given environment.

Therefore, MDPs are fundamental to understanding and implementing reinforcement learning algorithms, as they provide a structured framework for modelling and solving sequential decision-making problems in dynamic environments.

Expected Returns

The expected return is the sum of the total discounted rewards the agent will gather throughout its trajectory. The goal of the agent is to maximise the expected return by taking the best actions, adopting the optimal policy. Usually, we use a discount factor $\gamma \in [0, 1[$ to control the impact of future rewards. The closer γ is to 1, the more weight we give to future rewards.
The expected return of trajectory τ from timestamp t is defined as:

$$\begin{aligned} G(\tau_t) &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... \\ &= R_{t+1} + \gamma G(\tau_{t+1}) \\ &= \sum_{k=0}^{\infty} \gamma^k R_{k+t+1} \end{aligned} \tag{2.15}$$

Value function

The value function measures the quality of a state by estimating the expected returns starting from that state, based on the policy the agent is following. It takes a state as input and provides an evaluation of its goodness or desirability. The value function is mathematically written as follows in Li, 2018:

$$\begin{aligned} V_\pi(s) &= \mathbb{E}_{\tau\sim\pi}[G(\tau)|S_0 = s] \\ &= \mathbb{E}_{A_t\sim\pi(.|S_t)}[\sum_{t=0}^{\infty} \gamma^t R(S_t, A_t)|S_0 = s] \end{aligned} \tag{2.16}$$

where τ is the trajectory chosen under policy π, $G(\tau)$ is the expected return from trajectory τ, $A_t \sim \pi(.|S_t)$ is an action under state S_t sampled under policy π, γ is the discounting factor and $R(S_t, A_t)$ is the immediate reward of choosing action A_t in state S_t.

Similarly we can define the action-value function, which estimates the expected return from a state and chosen action

$$\begin{aligned} Q_\pi(s, a) &= \mathbb{E}_{\tau\sim\pi}[G(\tau)|S_0 = s, A_0 = a] \\ &= \mathbb{E}_{A_t\sim\pi(.|S_t)}[\sum_{t=0}^{\infty} \gamma^t R(S_t, A_t)|S_0 = s, A_0 = a] \end{aligned} \tag{2.17}$$

The objective of reinforcement learning is to find the optimal policy leading to the optimal value function:

$$v^*(s) = \max_\pi v_\pi(s) \quad , \forall s \in S \tag{2.18}$$

Similarly we obtain the optimal action-value function

$$q^*(s,a) = \max_{\pi} q_\pi(s,a) \quad , \forall s \in S, a \in A \tag{2.19}$$

We have the following relationship:

$$v^*(s) = \max_{a \sim A} q^*(s,a) \tag{2.20}$$

The Bellman equation is a recursive way to determine the optimal path through a sequence of decisions. The Bellman equation for optimal value function is

$$V^*(s) = \max_{a} \mathbb{E}_{s' \sim (.|s,a)}[R(s,a) + \gamma V^*(s')] \tag{2.21}$$

and the Bellman equation for optimal action-value function is

$$Q^*(s,a) = \max_{a} \mathbb{E}_{s' \sim (.|s,a)}[R(s,a) + \gamma \max_{a'} Q^*(s',a')] \tag{2.22}$$

where $R(s,a)$ is the immediate reward from state s by performing action a.
The optimal policy for state s is

$$\pi^*(s) = \underset{a \sim A}{argmax} \{R(s,a) + \gamma v^*(s')\} \tag{2.23}$$

Exploration-exploitation trade-off

Reinforcement learning is an expansive field comprising various types of algorithms. One common challenge shared among these algorithms is the *exploration-exploitation trade-off*.
In order to maximise cumulative rewards, agents must find a balance between exploiting known strategies and exploring new possibilities. Exploitation involves leveraging existing knowledge to select actions that appear optimal based on current information. However, relying on exploitation only may limit the agent's ability to discover potentially better actions.

On the other hand, exploration consist of testing unknown territory to uncover alternative actions that may yield higher rewards. By taking calculated risks and experimenting with new strategies, agents can expand their understanding of the environment and potentially identify better strategies.
Therefore, achieving an optimal balance between exploitation and exploration is crucial for effective decision-making in reinforcement learning. While exploitation ensures the immediate utilisation of known strategies, exploration drives innovation and facilitates the discovery of potentially more rewarding options.

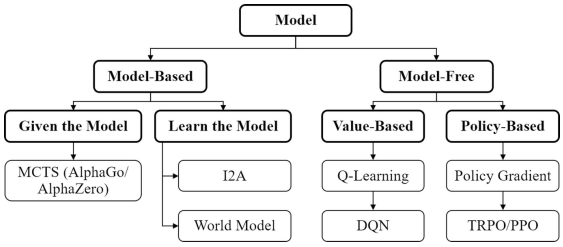

Figure 2.6: Model based vs Model free methods H. Zhang and Yu, 2020

Model based vs Model free

In Reinforcement Learning, the term "model" refers to the understanding of the environment.

In model-based approaches, the model can either be given or the agent will learn it. When the model is known, we have information about aspects such as the resulting state and the reward of an action before executing it. For example, an agent playing chess may know the rules of chess and therefore the consequences of each move. As illustrated in Figure 2.6, among model-based approaches, we have the "given the model" approach, where the agent has access to all information about transition probabilities between states and the rewards of each action. Conversely, we have the "learn the model" approach, where the agent attempts to estimate these transition probabilities and rewards.

On the other hand, in a model-free approach, explicit knowledge about the environment's behaviour is lacking, and the process does not involve constructing a model. Instead of relying on a model of the environment, the approach directly focuses on finding the best policy.

To clarify the distinction between model-based "learn the model" and model-free approaches, "learn the model" implies acquiring knowledge about the environment to make decisions (the agent obtains an estimated representation of the environment and its transition probabilities and rewards), while model-free methods directly learn the policy based on past experience without an explicit representation of the model.

Policy based vs Value-based

Value-based and policy-based methods are two categories of reinforcement learning algorithms that differ in how they optimise the policy.
Value-based optimisation algorithms focus on optimising the action-value function

$Q*_{\pi}(s, a)$, which estimates the value of taking a particular action in a given state. These algorithms aim to learn the optimal action-selection strategy indirectly by maximising the expected cumulative reward over time.

As explained in Ding et al., 2020, the advantages of value-based methods include low variance in value function estimation, high sample efficiency, and reduced risk of getting trapped in local minima. However, these methods typically struggle with continuous action spaces and may overestimate the value function when using ϵ-greedy and max operator. Additionally, they may encounter convergence and stability issues.

On the other hand, policy-based optimisation algorithms directly optimise the policy, which specifies the agent's behaviour in terms of the actions it selects in different states. By directly adjusting the parameters of the policy, these algorithms seek to find the policy that maximises the expected cumulative reward without explicitly estimating the action values.

Consequently, policy-based methods are more stable and capable of handling continuous action spaces. They also offer advantages such as simpler policy parameterisation and improved convergence (Ding et al., 2020).

Basically, value-based methods optimise the action-value function to indirectly find the optimal policy, while policy-based methods directly adjust the policy parameters to find the best action-selection strategy.

On policy vs Off policy

The difference between on-policy and off-policy methods lies in how they manage exploration and exploitation within their policies.

On-policy methods involve the agent using the same policy for both exploration and exploitation. This means that the decisions made by the agent are based on the policy it is currently refining. Consequently, these methods update their policy based on experiences generated by their current policy. While this approach is typically more stable, it may lack in exploration.

Conversely, off-policy methods can refine the policy using experiences generated from another policy. This enables a clear distinction between exploration (utilising a policy other than the one being refined) and exploitation (utilising the policy being refined). For instance, imagine that for ϵ ($\epsilon \in [0, 1]$) of the time, the agent follows a random policy (exploration, different from the one being refined), while

for $1 - \epsilon$ of the time, it exploits its policy. This approach can be more efficient for exploration and, therefore, achieve good performance. However, they may lack stability.

2.2.2 Q-Learning

Since this thesis utilises *Deep Q-Learning*, we will develop its foundational version: *Q-Learning*.
Q-Learning is a model-free, value-based, off-policy reinforcement learning algorithm. In Q-Learning, the agent learns to make decisions without explicitly modelling the environment dynamics and directly estimates the value of taking a particular action in a given state.
To facilitate this, Q-Learning maintains an action-value function represented in a Q-table, which stores the expected rewards for each state-action pair. This table is updated iteratively based on the rewards received during interactions with the environment.
For each state-action pair in the Q-table, the Q-Learning algorithm updates the Q-values according to equation 2.22:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R(S_t, A_t) + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{2.24}$$

where α denotes the learning rate, $R(S_t, A_t) + \gamma \, max \, aQ(St + 1, a)$ represents the target Q-value, and γ is the discount factor controlling the influence of future rewards. The difference between the target Q-value and the current Q-value is referred to as the temporal difference, serving as an error term guiding the update process.

During the inference phase, the agent selects actions based on the learned Q-values. It chooses the action that maximises the expected reward for the given state according to the Q-table.
While theoretically sound, Q-Learning faces practical challenges, especially in environments with a large number of states. As the number of states increases, the size of the Q-table grows exponentially, making it computationally prohibitive and inefficient. This limitation motivates the development of more scalable and efficient algorithms, such as Deep Q-Learning, which utilises neural networks to approximate the Q-function and replace the Q-table, enabling effective learning in high-dimensional state spaces.

2.3 Deep Reinforcement Learning

Deep reinforcement learning is a particular type of reinforcement learning that leverages deep neural networks for state representation and/or function approximation for value function, policy, transition model, or reward function (Li, 2018). This approach combines the power of reinforcement learning algorithms with the capacity of deep learning models to handle complex and high-dimensional data. By utilising deep neural networks, deep reinforcement learning enables more sophisticated and nuanced decision-making in dynamic environments, where traditional reinforcement learning methods may struggle to generalise effectively. Moreover, the flexibility and adaptability of deep neural networks make them well-suited for learning complex mappings between states, actions, and rewards, thereby enhancing the performance and efficiency of reinforcement learning algorithms in a wide range of applications.

2.3.1 Deep Q-Learning

As stated in Section 2.2.2, Q-Learning becomes impractical when the number of possible state-action pairs becomes too large, as the Q-table grows excessively. Deep Q-Learning (DQN) by Mnih et al., 2015 addresses this limitation by replacing the Q-table used in traditional Q-Learning with a deep neural network. This network estimates the expected reward for each action based on the current state. Additionally, DQN incorporates various techniques to enhance training stability, distinguishing it from standard Q-Learning.

Deep Q-Learning is a model-free, off-policy value-based algorithm. It is considered model-free because the agent learns without prior knowledge of the environment's rules, relying instead on trial and error. The algorithm is off-policy because the behaviour policy is not always the greedy policy. Specifically, the agent selects a random action with probability ϵ as part of exploration, and it selects the greedy policy action with probability $1-\epsilon$ for exploitation. The behaviour policy, therefore, is defined as $\epsilon \cdot \text{random}(action) + (1 - \epsilon) \cdot \pi(s, a)$. DQN is value-based since it revolves around estimating the Q values directly without explicitly modelling the policy.

2.3.2 DQN algorithm

Before we integrate all the elements, let's break down the components of the DQN algorithm.

Deep Neural Network

The DQN algorithm employs a deep neural network that takes the state observation as input and outputs an estimated Q-value for each possible action. This neural network is commonly referred to as the "policy network." We denote the output of the policy network for state s and action a as $\Pi(s)_a$.

The observation can be either a list of values or a pixel array representing the game state image. In the latter case, convolutional layers are integrated into the neural network architecture. This enhances performance and generalisation capabilities, allowing the network to effectively process visual information and learn representations that are more generalisable across different environments.

Experience replay buffer

Also known as memory replay, the DQN algorithm utilises an experience replay buffer to store all previous experiences as transitions $\mathcal{T} = (s_t, a_t, R, s_{t+1}, is\_done)$ observed during previous interactions with the environment. This memory replay mechanism enables more efficient learning by breaking down temporal correlations in the data and improving sample efficiency. Experience replay allows the algorithm to learn from a diverse set of experiences, reducing the risk of over-fitting to recent experiences and improving the stability of the learning process. By randomly sampling experiences from the replay buffer during training, the algorithm can explore a broader range of state-action pairs and learn more effectively from past interactions.

Target network

When only using one (policy) network for both the predicted Q-value and the target Q-value, we encounter a non-stationary target problem. This issue arises because during the training process, as the policy network updates its parameters to better approximate the Q-values, the target Q-values also change. Consequently, the target values used for training become moving targets, leading to instability and slow convergence in learning.
To address this problem, as proposed in the paper by Mnih et al., 2015, a second network, called the target network, is introduced. This target network is essentially a clone of the policy network but remains frozen during training and is updated periodically every $\mathcal{C}$ steps. By using the target network to generate target Q-values, the non-stationarity of the target values is mitigated. This approach stabilises the training process by providing more consistent and less volatile target values, resulting in more reliable updates to the policy network and faster convergence towards optimal policies.

Algorithm

The DQN algorithm with experience replay is described as followed in Mnih et al., 2015:

Algorithm 1: Deep Q-Learning with memory replay

Initialisation
- initialise memory D with capacity N
- initialise policy network Π with random weight θ
- initialise target network Π' with weight $\theta' = \theta$

for *episode=1, M* **do**
- initialise environment s_0
- **for** *timestamp t=1, T* **do**
  - **Select action**
    - Random action a_t with probability ϵ
    - Greedy action $a_t = argmax_a \, \Pi(s_t)_a$
  - Execute action a_t and observe next state s_{t+1} and reward r_t;
  - Store transition $T_t = \{s_t, a_t, r_t, s_{t+1}\}$ in D
  - Sample random mini-batch of transitions T_i from D
  - **Optimise model**
    - Compute $y_j = r_j + \gamma \, max_{a'} \, \Pi'(s_{j+1})_{a'}$
    - Compute loss between y_j and $\Pi(s_j)_{a_j}$
    - Perform gradient descent step on loss
  - Every $\mathcal{C}$ step, synchronise target network as $\Pi' = \Pi$

Action selection

As commonly used in reinforcement learning, action selection in the algorithm follows an epsilon-greedy policy. This policy allows the agent to balance exploration and exploitation, crucial for effective learning.
In Algorithm 1, the agent selects a random action with probability ϵ and chooses the best action according to its policy network for the remaining time. The value of ϵ decays over time according to equation 2.25, typically decreasing as the agent gains more experience.

$$\epsilon \leftarrow \epsilon_\infty + (\epsilon_0 - \epsilon_\infty) * e^{-\frac{t}{\epsilon_{decay}}} \tag{2.25}$$

Figure 2.7 depicts the value of epsilon in function of the steps done during the training for different values of epsilon decay. A higher value of slows the decay allowing a longer exploration phase.

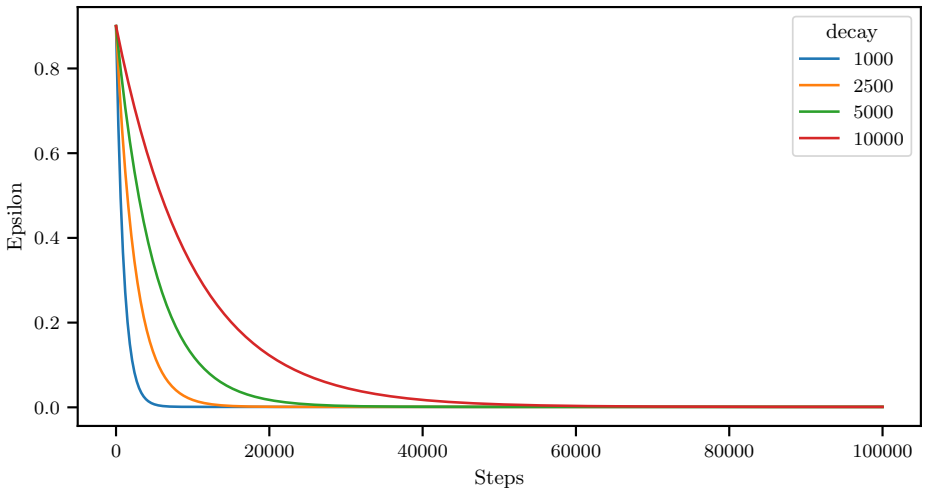

Figure 2.7: Epsilon in function of the steps t with different values of ϵ_{decay}

Initially, ϵ is set to a high value (EPS_Start), encouraging exploration during the early stages of learning. This high exploration rate enables the agent to gather diverse experiences and learn about the environment. As learning progresses, ϵ gradually decreases, shifting the agent's focus towards exploitation. Eventually, the agent primarily exploits its learned knowledge to maximise rewards. For example, ϵ_∞ may be set to a small value like 0.001, indicating that the agent continues to explore with a low probability even after extensive learning, ensuring some level of exploration is maintained throughout the training process.

Model Optimisation

The optimisation step of the DQN algorithm involves computing the loss between the predicted Q-value $\Pi(s_j)_{a_j}$ and the target Q-value $r_j + \gamma\, max\, a' \Pi'(s_{j+1})_{a'}$. If the state s_j is a terminal state (indicating the end of an episode, such as when the game is over), there is no subsequent state $sj + 1$, and the target Q-value is simply the immediate reward r_j.
Typically, stochastic gradient descent (SGD) or its variants like Adam are used as optimisation algorithms. These algorithms update the parameters of the policy network based on the gradients of the loss function with respect to the network's parameters. Through repeated iterations of optimisation, the policy network gradually learns to better estimate the Q-values and improve its performance in the reinforcement learning task.
Every $\mathcal{C}$ step, the target network is synchronised according to a soft update.

$$\theta' = \tau * \theta + (1 - \tau) * \theta' \tag{2.26}$$

2.3.3 DQN Improvements

The DQN Algorithm from Mnih et al., 2015 is a good initial algorithm, but many extensions have been proposed to enhance its performance, both in terms of final outcomes and data efficiency. This section first lists some popular extensions and explains the problems they aim to solve. Understanding these limitations of classical DQN can provide insight into the need for these extensions. Then we will present Rainbow DQN from Hessel et al., 2017, which proposes a symbiotic combination of some extensions to increase performances.

This thesis was conducted using the original version of DQN from Mnih et al., 2015, but it can readily be extended to more advanced versions of DQN.

Double DQN

Q-Learning can sometimes perform poorly due to an overestimation of the value function, stemming from the max function in the action-value function (Equation 2.24). To address this issue, Hasselt, 2010 proposed the use of two Q-functions to decouple the evaluation of an action and the selection of the action.

Algorithm 2: Double Q-Learning (Hasselt, 2010)

Initialise Q^A, Q^B, s_t
for *timestamp t=1, T* **do**
  Choose actions a_t based on Q^A and Q^A
  Execute action a_t and observe s_{t+1} and r_t
  Choose randomly either UPGRADE(A) or UPGRADE(B)
  if UPGRADE(A):
    Define $a* = argmax_a Q^A(s_{t+1}, r_t)$
    $Q^A(s_t, a) \leftarrow Q^A(s_t, a) + \alpha(r_t + \gamma Q^B(s_t, a*) - Q^A(s_t, a)$
  else UPGRADE(B):
    Define $b* = argmax_a Q^B(s_{t+1}, r_t)$
    $Q^B(s_t, a) \leftarrow Q^B(s_t, a) + \alpha(r_t + \gamma Q^A(s_t, b*) - Q^B(s_t, a)$
  $s_{t+1} \leftarrow s_t$

This can easily be adapted to Deep Q-Learning, named Double DQN. This approach differs from classical DQN, which also employs two networks but assigns them different roles. In classical DQN, the "policy" network handles both action selection and value evaluation, while the "target" network is periodically updated to provide stable target values during training. In contrast, Double DQN separates the action selection and value evaluation processes entirely, utilizing two distinct

Q-functions to mitigate overestimation bias. This has been demonstrated to reduce the overestimation present in DQN (Hasselt, 2010).

Prioritized experience replay

Classical DQN samples transitions uniformly from the replay memory. Schaul et al., 2016 propose to sample experiences with a probability proportional to their significance. This aims at replaying important transitions more frequently and increasing learning efficiency. The probability is proportional to the temporal difference $r_j + \gamma \, max_{a'} \, \Pi'(s_{j+1})_{a'} - \Pi(s_j)_{aj}$ from Algorithm 1. It has been shown that DQN with prioritized experience replay outperforms classical DQN most of the time.

Dueling network architecture

Wang et al., 2016 present a new network architecture. They implement two separate networks, one for estimating the state value function and the other for estimating the advantages of each action. The dueling network architecture improves learning efficiency by enabling the network to generalize better across actions and states, especially in environments where the values of different actions vary significantly depending on the state.

The outputs of the two networks are combined to obtain the Q-value for each action in the state:

$$Q(s,a) = V(s) + A(s,a) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s,a') \tag{2.27}$$

where $V(s)$ is the network estimating the state value, $A(s,a)$ estimates the advantage of taking action a in state s, $\mathcal{A}$ is the action space. Both network V and A share a convolutional encoder.

Multi-step bootstrap targets

Classical DQN employs single-step learning, where the agent updates its policy based on the immediate rewards, or in other words, the rewards received after each action. In multi-step learning, however, the agent considers the rewards obtained over n steps to update its policy. The n-step return is calculated by summing the rewards obtained over the n-step trajectory, discounted by the appropriate discount factor for each time step.

$$R_t^{(n)} = \sum_{k=0}^{n-1} \gamma^k R_{t+k+1} \tag{2.28}$$

Multi-step learning reduce the variance of the estimated returns, improve sample efficiency by allowing the agent to learn from fewer transitions, provides a more stable learning and often lead to faster learning (R. Sutton and Barto, 1998).

Distributional Q-Learning

When a state-action pair is passed to classical DQN, only a single value is returned for $Q(s, a)$, which corresponds to the mean of the value function prediction, failing to capture the uncertainty and variability of the returns. To address this problem, Dabney et al., 2017 propose to estimate a distribution of the returns instead of only estimating the mean, using quantile regression or other distributional approaches. The agent can make better decisions when it knows both the mean and the uncertainty (represented by the spread) of the estimated Q-value of each action. It has been shown that this approach improves learning stability and performance over classical DQN.

Noisy DQN

Noisy networks, as introduced by Fortunato et al., 2019, facilitate efficient exploration. They are straightforward to implement and slightly increase computational costs.
Classical DQN can encounter limitations with ϵ-greedy policies, particularly when many actions are required before the first reward is collected. Noisy networks propose to address this issue by injecting noise directly into the parameters of neural networks, thereby combining a classical deterministic stream with a learnable noisy stream. Initially, the network learns to utilize the noisy stream to improve exploration, but over time, it tends to ignore it. The noise can also be annealed gradually to encourage exploration during the early stages of learning and exploitation in later stages.
Enhanced exploration during the early phase of the learning process can lead to improved performance later on.

Rainbow DQN

Hessel et al., 2017 integrates all of the listed extensions of DQN to create a highly efficient DQN agent called Rainbow. The specific implementation details can be found in their paper. Rainbow DQN achieves state-of-the-art performance on a wide range of reinforcement learning tasks, particularly in the Atari 2600 benchmark suite. It significantly outperforms the original DQN algorithm and its individual extensions, demonstrating the effectiveness of combining multiple advanced techniques in deep reinforcement learning.

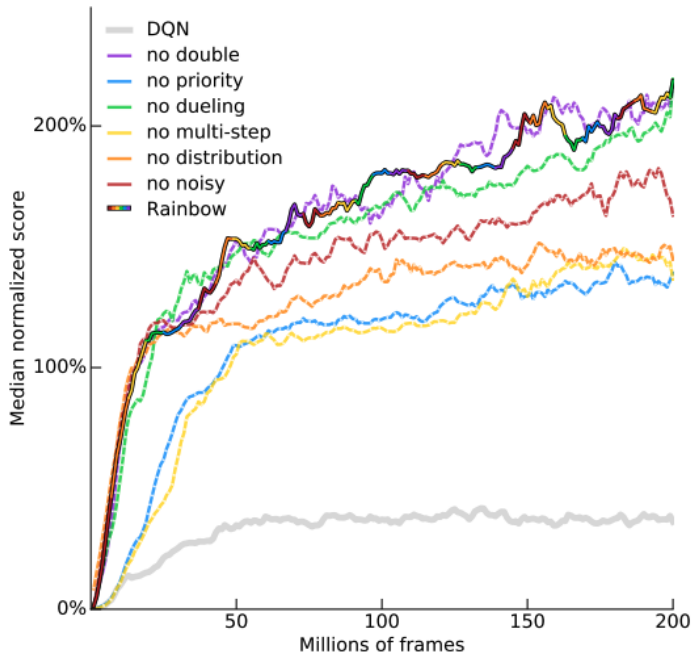

Figure 2.8: Median human-normalized performance across 57 Atari games, as a function of time. Comparing Rainbow agent to DQN and six different ablation. (Hessel et al., 2017)

Figure 2.8 shows the median normalized performance of different agents across Atari games. We first observe that classical DQN (grey) is not performing well compared to the Rainbow agent (rainbow). The other lines represent the Rainbow agent with the ablation of one extension. We see that the most important extensions to Rainbow are *Priority* (blue), *Multi-step* (yellow), and *Distributional* (orange) since their ablation causes a major drop in performance. The moderately important extensions appear to be *Noisy* (red) and *Dueling* (green), while *Double* (purple) doesn't seem to significantly affect the Rainbow performance.

While Rainbow DQN shows state-of-the-art performance, it's important to take into consideration the computational cost required to train these models and find appropriate hyper parameters. Ceron and Castro, 2021 examines the computational costs of the paper by Hessel et al., 2017 and revisits it with reduced computational power while presenting new insights into the Rainbow agent.

2.3.4 Deadly triad

In reinforcement learning, instability and divergence can arise during training depending on the algorithm used, a phenomenon termed the "Deadly Triad" R. S. Sutton and Barto, 2020. DQN is an algorithm that can suffer from the Deadly Triad as it satisfies all of its conditions. The Deadly Triad occurs when three elements converge in an algorithm:

- **Function Approximation:** The agent generalizes the value function when the state space is too large to be stored in memory.
- **Bootstrapping:** The agent is updated based on existing estimates (such as temporal difference methods) rather than actual rewards. Bootstrapping methods are usually faster to learn.
- **Off-Policy Training:** The agent optimises a policy while behaving with another policy, for example, to balance exploration and exploitation.

The concept of the triad is developed in R. Sutton and Barto, 1998. According to Li, 2018, these three elements are crucial: function approximation for handling complex problem with large state-action space and making generalisations, bootstrapping for efficiency in computations and data usage, and off-policy learning for allowing flexibility in behaviour policies compared to target policies.

The DQN algorithm is an off-policy algorithm that uses temporal difference methods, thereby employing bootstrapping, and utilises a neural network to approximate the value function, rendering it susceptible to the Deadly Triad. However, DQN can still produce good performances. van Hasselt et al., 2018 investigate in practice the impact of the Deadly Triad.

Chapter 3

Implementation and measures of performances

Before delving into the experiment details, it's important to outline the chosen game for the experiments and the implementation process of both the game and DQN agent. This section will introduce the game, describe its implementation, and detail the implementation of the DQN agent. Following that, various performance measures used in the analysis of the experiment results will be discussed.

3.1 Implementation

3.1.1 Environment

FlappyBird

The game selected for the thesis is *Flappy Bird*. This choice was made due to its simplicity and relevance to reinforcement learning tasks. Flappy Bird gained widespread popularity a few years ago, thanks to its minimalistic yet addictive design. Despite its simple graphics and mechanics, the game presents a considerable challenge to players, requiring precise timing and quick reflexes to succeed. This simplicity, coupled with its high difficulty level, contributed significantly to its viral success and enduring popularity among gamers.

Additionally, Flappy Bird's straightforward gameplay makes it an ideal testbed for studying reinforcement learning algorithms, allowing researchers to focus on the core principles of the algorithm without being overshadowed by complex game mechanics. Flappy Bird is a 2D game with discrete action spaces, enabling straightforward implementation of algorithms such as Deep Q-Networks (DQN). The game also features several adjustable parameters, making it conducive to experimentation

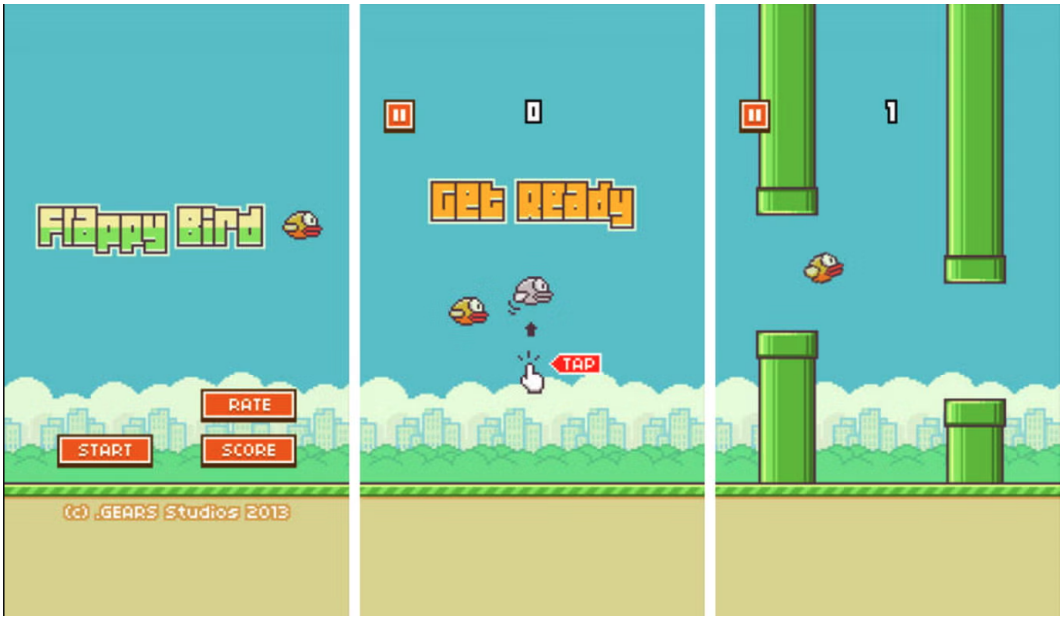

Figure 3.1: Flappy Bird gameplay (Vincent, 2014)

and the introduction of randomness.

In Flappy Bird, players control a bird that must navigate through a series of obstacles, represented by pairs of pipes with gaps in between (Figure 3.1). The bird has two actions: "flap," which causes it to ascend, and "idle" doing nothing, causing it to descend due to gravity. Players must time their flaps carefully to guide the bird through the openings in the pipes and maintain its flight for as long as possible. Colliding with a pipe or touching the ground results in the game ending.

OpenAI Gym

OpenAI Gym is a toolkit for developing and testing reinforcement learning agents. It provides complete out-of-the-box game environments like Atari games, as well as simpler toy environments for testing simpler tasks. It provides a standardised interface for interacting with environments, promoting ease of use and interoperability across different algorithms and environments. Also, these environments are not limited to games; they span various problem domains, including robotics, control systems, and optimisation tasks.
Moreover OpenAI Gym serves as a bench-marking platform, enabling researchers to compare the performance of different algorithms on standardised tasks.

OpenAI provides a standardized interface for every environment, with the key components being the following:

- *Action space*: Each environment has an action space defining the possible actions the agent will choose from. There are discrete action space (ex: jumping or do nothing) or continuous actions space (ex: the angle to turn a car on a circuit).
- *Observation space*: Each environment has an observation space defining what the agent can receive as input to make its decisions. The observation space provides information about the current state of the environment, enabling the agent to understand its surroundings and take appropriate actions. can range from simple lists of values representing important measures to more complex structures such as images in arrays of pixels or a stack of array of pixels, the latter gives temporal information.
- *Step function*: This function applies an action from the action space, thereby modifying the state of the environment. It returns the observation of the resulting state, the reward associated with this action, whether the game is completed or not, and any additional specific information about the environment.
- *Reset function*: This function simply resets the game to its initial state and returns its observation, allowing for the start of a new game.

Thesis environment implementation

The environment[1] was built on top of an OpenAI Gym wrapper around a *Flappy Bird* game developed in Python by Nogueira, 2024. The wrapper ensures that all core functions and specifications of an OpenAI Gym environment are present.
Some modifications were needed to tailor it to the tasks, such as determining which values were returned as observations for a simple agent. The version used in the thesis returns the following variables:

- player_x: x position of the bird on the screen
- player_y: y position of the bird on the screen
- upper_pipe_y: y position of the next upper pipe coming
- lower_pipe_y: y position of the next lower pipe coming
- pipe_center_y: y position of the center between the two pipes

[1]The source code can be found on GitHub at https://github.com/CorentinVermeulen/epl_flappybird.

- pipe_center_x: x position of the center between the two pipes
- v_dist: vertical distance between the center of the pipes and the player
- h_dist: horizontal distance between the center of the pipes and the player,
- player_vel_y: vertical velocity of the player

Some values may seem redundant due to the presence of others, like 'h_dist' since h_dist = pipe_center_x - player_x. However, after local tests, it was found that the agent performed more efficiently with all variables rather than just a few.

Additionally, it's possible to provide an image of the game state as an array of pixels as observation instead of the list of variables, which is a more general approach but computationally intensive. Initially, we opted to test with a simpler approach using only a list of variables.

Concerning the rewards, it's a good practice to clip them between -1 and 1. Therefore, we found that assigning rewards of $A : +0.1$ when the bird was kept alive, $P : +1$ for passing a pipe, and $GO : -1$ for a game-over performed well. We tried other combinations such as $[A : 0, P : +1, GO : -1]$, $[A : +1, P : +0, GO : -1]$, $[A : +0.1, P : +0, GO : -1]$, and $[A : +1, P : +1, GO : -1]$, but none outperformed the first one in this application.

To measure the progress of the game, we could either use the original game score corresponding to the number of pipes passed or the duration of the game in time frames, which is a continuous version of the score and corresponds to the number of time the agent had to make a decision. We decided to use the duration as it was more precise. To prevent infinite game-play when the agent successfully passes all pipes in succession, we've opted to conclude the game after 20 pipes, which corresponds to a duration of 1517 time-frames.

3.1.2 DQN Agent

The DQN algorithm implementation is highly inspired by the tutorial provided by Paszke and Towers, 2017 on the PyTorch documentation website.
As mentioned earlier, initially, computations were conducted on observations comprising only a list of relevant parameters, which is faster and more straightforward. We did not choose to use the RGB version, which involves providing the game state image as input, necessitating the addition of a CNN before the policy network, because of excessive computation needs.

Before launching experiments, we needed to determine the optimal set of hyper-parameters to ensure the effectiveness of the algorithm and facilitate the isolation of the impact of subsequent game context modifications.
The hyper-parameters with the most significant impact include the network architecture, the learning rate, γ the attention given to future rewards, and τ the portion of the policy network that is copied to update the target network.
The batch size is also an important hyper-parameter. The best learning occurred with high values of batch size, but this also led to the most time and computation-intensive learning. Therefore, we opted for a value that strikes a balance between effectiveness and computational cost.

Network architecture

Many different network architectures were tested, from shallow networks to deep networks with many large layers. The optimal network architecture comprised either 4 hidden layers, each with a size of 256 (256, 256, 256, 256) and a total of 200,706 trainable parameters, or 6 hidden layers with sizes (64, 128, 256, 512, 256, 128), totalling 338,114 trainable parameters. Despite their parameter differences, both architectures delivered similar performance levels. To mitigate computational costs, we opted for the architecture with fewer parameters which is the 4 hidden layers of size 256.
Figure 3.2 illustrates the average duration for each layer size with a 95% confidence interval. We observe that the two previously mentioned architectures truly stand out from the group.

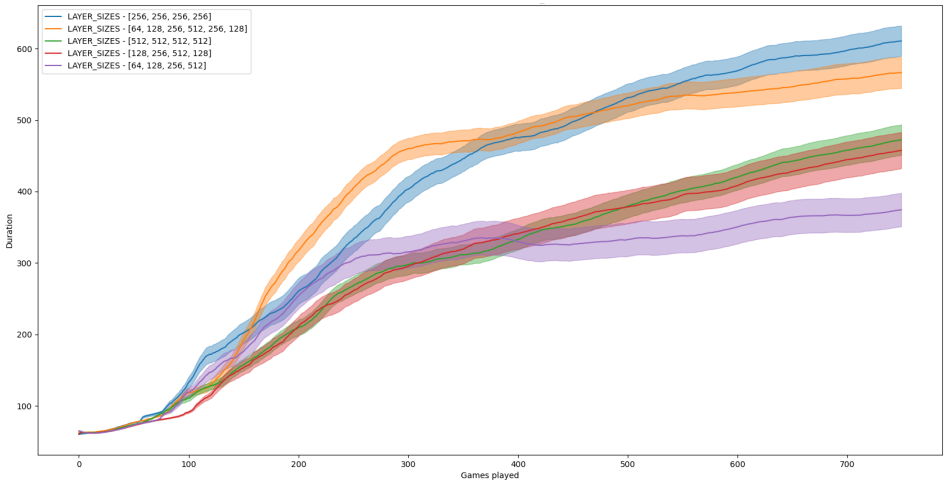

Figure 3.2: Average duration with 95% CI (n=12)

Gamma γ - Learning rate lr - Tau τ

Once the layer size was fixed, a grid-search-like experiment was initiated to determine the optimal hyper-parameter set for gamma (γ), which represents the weight given to future rewards as explained in equations 2.15 and 2.24; the learning rate for the optimiser (Adam optimiser presented by Kingma and Ba, 2017); and tau, which corresponds to the portion of the policy network that is copied to the target network when the latter is updated, as depicted in equation 2.26.

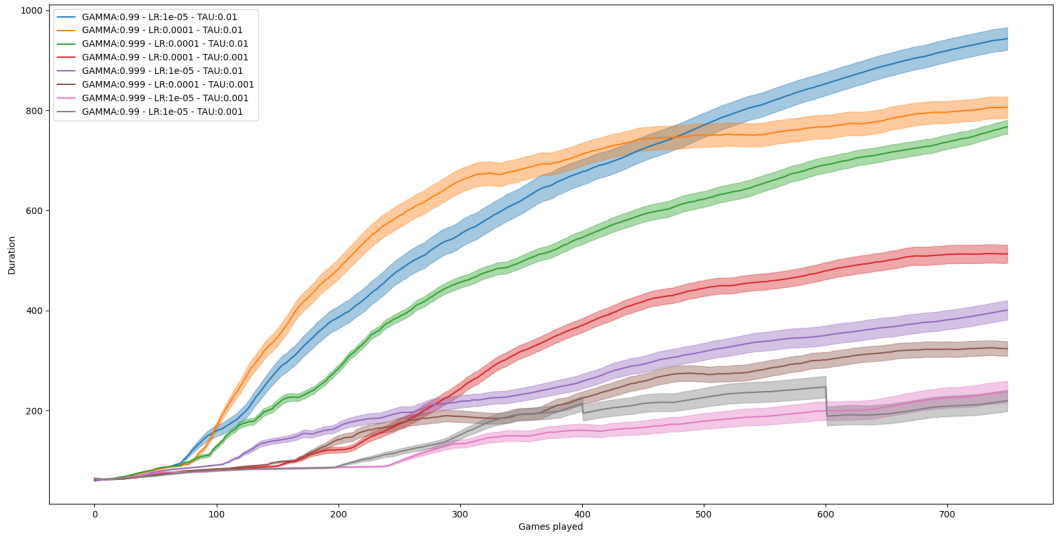

Figure 3.3: Average duration with 95% CI (n=5)

Figure 3.3 illustrates that three sets of hyper-parameters stand out: $\{\gamma = 0.99, \tau = 0.01, lr = 1e-5\}$, $\{\gamma = 0.99, \tau = 0.01, lr = 1e-4\}$, and $\{\gamma = 0.999, \tau = 0.01, lr = 1e-4\}$. The choice of τ value is straightforward, as only 0.01 stands out. Concerning γ, we opted for 0.99. Regarding the learning rate, certain parameter configurations yielded favourable results with both $1e-4$ and $1e-5$. However, in some cases, we obtained more stable outcomes with $1e-5$, thus we will retain this value.

3.2 Measures of performance

It is essential to define performance measures to establish an objective framework for assessing learning outcomes accurately. The primary goal is to characterise the influence of changes in game parameters on both learning complexity and quality. Additionally, defining these measures will facilitate the comparison of various learning iterations and enhance the communication of results.

By identifying the optimal hyper-parameter set for the baseline learning model and applying them consistently in subsequent experiments, we aim to isolate the effects of changes in game configuration. This approach will help us pinpoint the specific impact of these variations on the learning process.

3.2.1 Average duration plot and running mean

The duration, defined as the number of frames until the end of the game, reflects the success of the agent's decision-making. The main plot displayed during the training is the average duration plot, as depicted in figure 3.4. The average duration plot is characterised by the following equation:

$$y_i = \frac{\sum_{j=0}^{i} \text{duration}_j}{i} \tag{3.1}$$

This equation signifies that the value for the i^{th} game played represents the mean duration from the start until that game. To gain further insight into the learning behaviour across epochs (games played), we might additionally present the running mean curve with a specific window width. This curve provides a more expressive representation of the learning behaviour. For instance, when the agent performs poorly for several epochs, the running mean will be directly impacted, whereas the average curve can remain more stable.

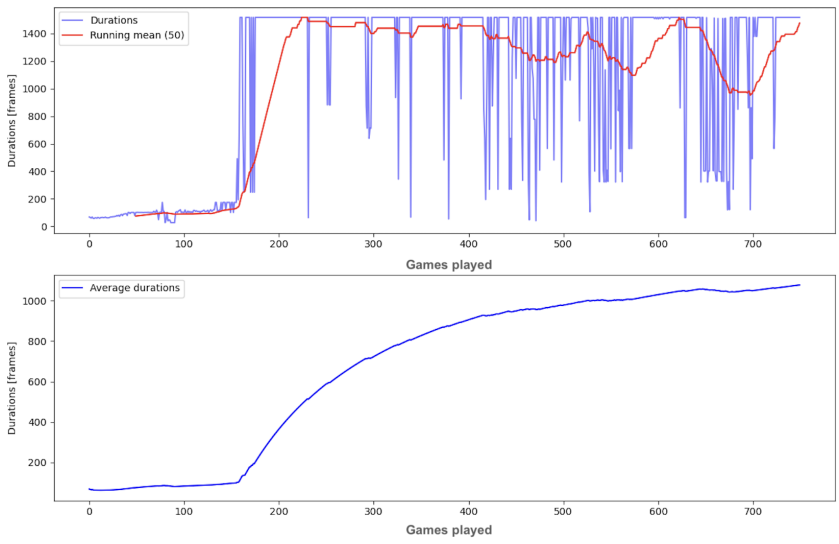

Figure 3.4: upper: Learning duration with a running mean on window width = 50; bottom: Average learning duration

In the upper plot of figure 3.4, the blue line depicts the duration for each game played during the learning process, while the red line represents a running mean with a window width of 50 games. The bottom figure illustrates the average duration plot.

3.2.2 Learning Velocity Index (LVI)

The first indicator characterizes the velocity of learning, it is supposed to indicate how many training games were needed to reach a certain level of performance. More precisely, it identifies when the running mean curve begins to ascend until it surpasses a certain value, expressed as the X% threshold of the maximum duration. Its value is a real number between 0 and the total number of games played.
For example, on figure 3.5 we added the $LVI50$, the Learning Velocity Index to 50% of maximum duration. The maximum duration is 1517 timeframes so LVI correspond to the number of game the agent had to play until the running mean of learning duration reached a value of $0.5 \times 1517 = 758.5$ which is about 125 games.

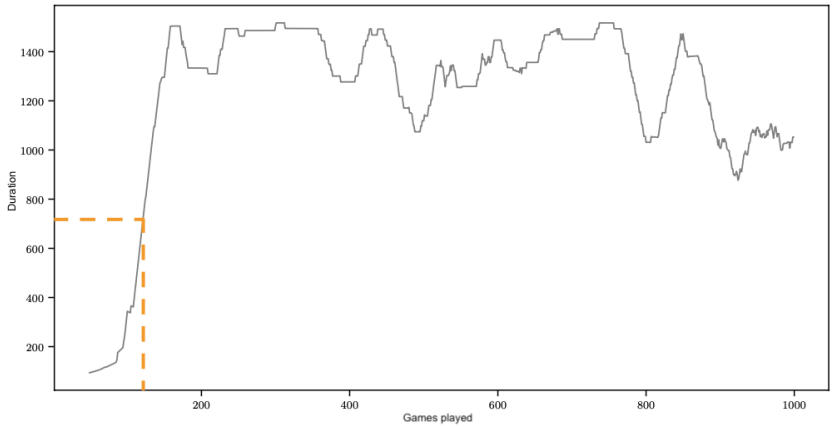

Figure 3.5: Running Mean plot with LVI50

The values of the Learning Velocity Index (LVI) are computed based on the maximum attainable duration. In cases where multiple configurations are compared and performance varies, for example on figure 3.6, the LVI curve for the less performant configuration will be lower than that of the more performant configuration. This discrepancy may occur even if both configurations exhibit the same learning velocity, as the lower performance diminishes the LVI. To address this, we can compute a normalized LVI where the threshold X is based on the final average duration value rather than the maximum attainable duration.

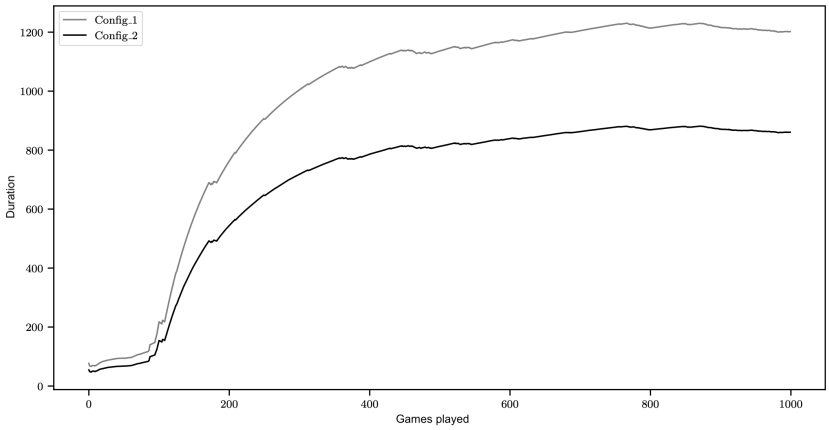

Figure 3.6: Average duration for two different configurations

Figure 3.7 depicts the LVI for two illustrative configurations across different threshold values. At first glance, it appears that Configuration 2 (black) learns more slowly, as it requires more games to reach each threshold than Configuration 1 (grey). However, when the LVI is normalized as on figure 3.8 by the mean duration , corresponding to the asymptotic value shown in Figure 3.6 (the last value on the right), we observe that both configurations exhibit the exact same LVI. This is expected, as the duration values of Configuration 2 where generated according to $\text{config\_1} * 0.7 \pm \text{small\_noise}$.

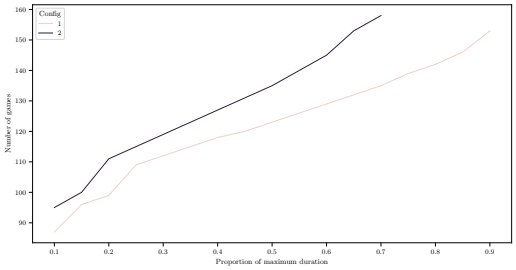

Figure 3.7: LVI

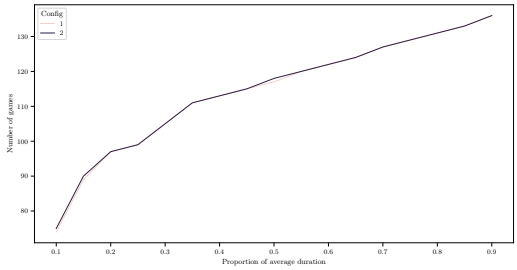

Figure 3.8: Normalized LVI

Of course, this value is influenced by the exploration phase of the DQN learning algorithm. If the value of ϵ decreases rapidly, we will have less exploration and a quicker exploitation. Conversely, if the exploration phase was adequate, the average learning rate will increase faster as the agent begins to exploit its learned knowledge and no longer plays randomly. To maintain consistency in our comparisons, we will always ensure that experiments are conducted with the same decay speed of epsilon. This approach ensures that the results remain comparable across experiments.
Additionally, we can envisage conducting the same experiment with different values

of epsilon decay and observe its effects. This would provide further insights into the impact of epsilon decay on learning performance.

3.2.3 Duration Surpass Proportion (DSP)

We can separate the training into two phases. The first phase, characterized by the LVI (Learning Velocity Indicator), occurs during early training when the agent makes significant progress in learning to play the game. The second phase is the late learning period, where the agent already understands the game rules sufficiently but needs to perfect its policy slightly. The Duration Surpass Proportion (DSP) indicator is meant to quantify the performance and stability during the late learning period.

DSP measures the proportion of games in which the agent surpasses a certain duration, represented as an $X\%$ threshold of the maximum duration. Its value ranges from 0 to 1, as it is a ratio between two number of games. The ratio is computed based on the period after the running mean surpasses the threshold, to discard the influence of the initial learning velocity.

For example, the $DSP50$ indicator measures the number of games achieving a duration of $0.5 \times 1517 = 758.5$ time-frames, divided by the number of games the agent played during the training session after that the running mean curve surpasses this threshold value (after game number 185 on figure 3.9).

It offers a straightforward measure of the model's performance consistency without taking into account the learning velocity part, which is represented by the LVI indicator. It allows us to assess the model's reliability and determine its ability to consistently meet desired performance levels. By tracking how often the learning process exceeds this duration threshold, we gain insights into the model's robustness and its capacity to maintain performance over time.

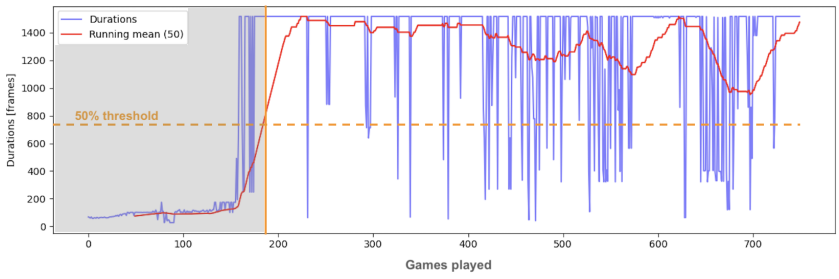

Figure 3.9: Average duration plot with a 50% duration threshold

Figure 3.9 shows the average duration plot with a 50% duration threshold and the vertical line where the running mean surpasses this threshold. The $DSP50$ is the number of games above this threshold line divided by the total number of games in the non-shaded area. The shaded area corresponds to the LVI area.

Chapter 4

Best context to learn

Previous sections have detailed the implementation of both the game and the agent following a DQN algorithm, along with the introduction of two specific performance measures to quantify learning. We can now proceed to the first experiment, which concerns determining the best context for learning.
This section will outline the methodology, state the hypotheses, and present the results aimed at evaluating under which conditions—random or fixed—a DQN agent learns to play Flappy Bird most effectively, both in terms of learning speed and overall performance.
In the next chapters, once the optimal learning context is determined, we will proceed with the following two experiments, focusing on the impact of increased action space and the introduction of uncertainty in action execution.

An advantage of Flappy Bird is the ability to create an infinite number of different scenarios by adjusting the vertical position of the pipe holes through which the bird must pass without touching the pipes. The vertical spacing of two consecutive pipes can affect the game's difficulty, making it easier if they are close together or more challenging if they are farther apart, requiring greater anticipation.

4.1 Methodology

When focusing on the vertical position of the pipes, there are two possible configurations:

- The fixed configuration where the vertical positions of the pipes remain the same for all games. This means the agent will always encounter the same sequence of pipe positions in each game. However, within a single game, not all pipes are at the same position.

- The random configuration where the vertical positions are random, meaning the agent faces different sequences of pipe positions each time it plays.

Hypothesis

Exploration and generalization: When the pipes' positions are random, the agent's exploration is enhanced by continually facing new situations, forcing it to generalize its learned strategies. Therefore, we hypothesize that the agent will learn faster and perform better due to increased exploration and generalization compared to when the pipes' positions are fixed.

Learning Velocity: The agent in a fixed condition only needs to learn a single trajectory and does not need to generalize. This might suggest that the agent will learn faster initially. However, when the agent in fixed conditions surpasses its previous best score, it encounters new, unknown situations, similar to the agent in a random environment. Because the agent in fixed conditions is less exposed to new situations, it is less likely to generalize, we hypothesize that he may not learn as quickly as the random agent.

Performances on Random Games: To assess the generalization ability of the agents, both configurations will be tested in random games. We hypothesize that the agent trained in the random configuration will outperform the agent trained in the fixed configuration when tested on random games, demonstrating superior generalization capabilities.

Experiments

We will conduct experiments to assess the performance of the DQN agent under fixed and random pipe position conditions. Both configurations will employ identical agent hyper-parameters (Table 2) and game settings, except for the variability in pipe positions. Given the stochastic nature of the DQN algorithm during exploration and its variance in learning performance, we will execute 10 training sessions for each configuration. For each training session, we will save the model parameters of the latest best-performing epoch.
We will evaluate general performance in terms of mean duration and the number of solved games. Additionally, we will assess learning velocity by comparing the Learning Velocity Index (LVI) and evaluate performance by examining the Duration Surpass Proportion (DSP).

To determine if the agent under fixed conditions has learned a single trajectory or generalized to some extent, we will test both configurations on random games and

evaluate their performance in terms of duration. For each trained agent, we will test its score on 100 random games.

Finally, since our objective is to select the best agent for solving Flappy Bird, we will compare the best agent from each configuration by testing them in random games.

4.2 Results

Figure 4.1 shows the average duration over the 10 learning sessions for both configurations. While the results suggest that the random configuration may perform slightly better, this difference is not statistically significant. An ANOVA test on the equality of the means for both configurations confirms this, as the p-value is 0.118, indicating no significant difference.
Similarly, in Figure 4.2, there appears to be a slight indication of better performance in the random configuration. However, an ANOVA test on the number of solved games also fails to show a significant difference between the two configurations (p-value = 0.118).

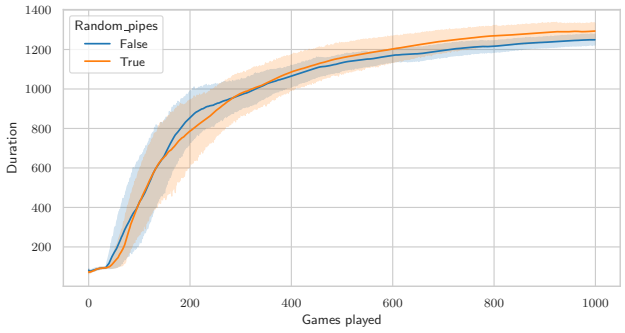

Figure 4.1: Average duration by configuration

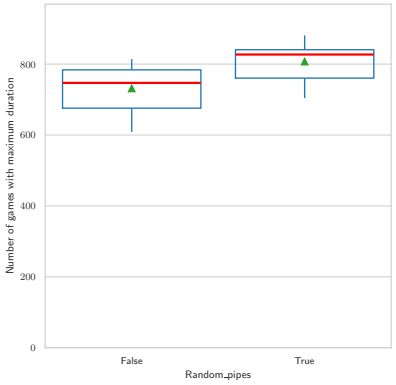

Figure 4.2: Number of solved games

Concerning the LVI and normalized LVI, figure 4.3 and 4.4, it's clear that both configuration learn at the same speed and there is no benefit in term of learning velocity to train an agent in random conditions.

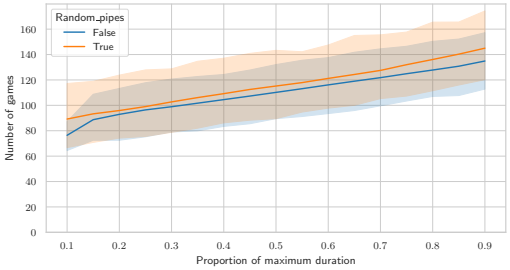

Figure 4.3: LVI

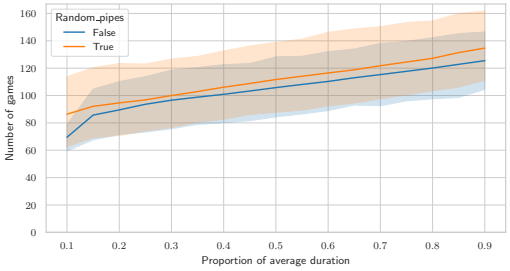

Figure 4.4: Normalized LVI

The DSP plot in Figure 4.5 shows a difference in performance for high threshold values. The agents trained under fixed conditions experience some drops at specific thresholds, which may correspond to challenging sequences of pipes in the games. Conversely, the agent trained under random conditions has a higher DSP, indicating superior performance.

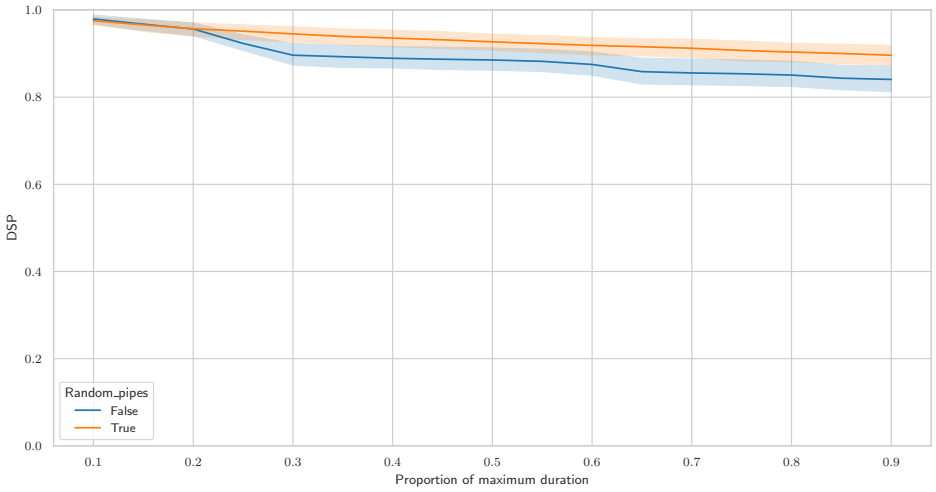

Figure 4.5: DSP

After testing each trained agent under random conditions over 100 games, it became evident on figure 4.6 that the agents trained under fixed conditions did not generalize well and were unable to perform effectively in different game scenarios. In contrast, the agents trained under random conditions demonstrated strong generalization capabilities, successfully solving almost every game.

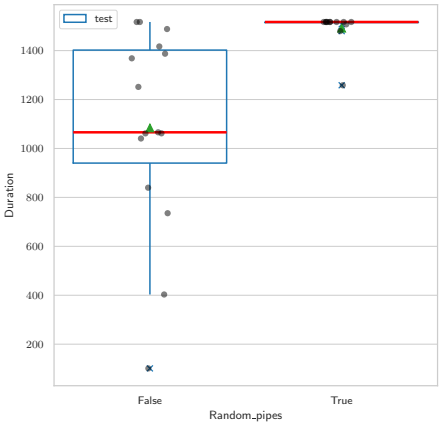

Figure 4.6: Test durations in random games by configuration

In the case where we aim to retain a single agent capable of solving every FlappyBird game, we compare the best-performing agents from both configurations. By "best," we refer to the agent that solved the most games during its training sessions and exhibited a high mean training duration.
As shown in Figure 4.7, the best agent trained under random conditions demonstrates superior performance, consistent with the insights gained from the entire population. Figure 4.8 further illustrates that while the agent trained under fixed conditions generally performs well, it occasionally fails to find a good trajectory and struggles to adapt to every game scenario. In contrast, the agent trained under random conditions exhibits better adaptability and consistently performs well across different games.

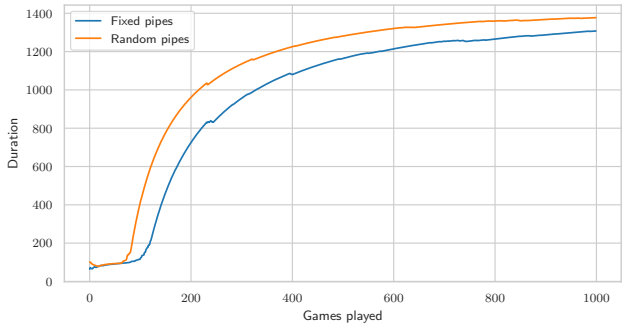

Figure 4.7: Average learning duration for best agents

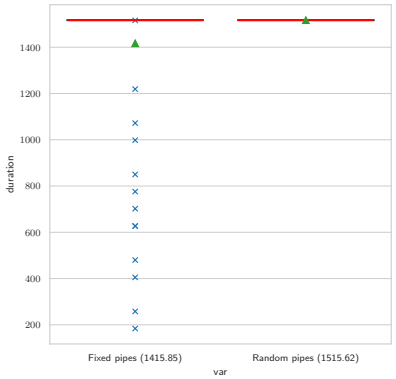

Figure 4.8: Testing durations in random games for best agents

4.3 Conclusion

We can conclude that there is no significant difference in terms of average duration during training sessions between the random and fixed configurations, although the data suggests that the random configuration may perform slightly better.

There is no difference in terms of learning velocity (LVI) between the two configurations, but there is a difference in terms of performance (DSP), where the random configuration performs slightly better for high threshold values.

Agents trained under fixed conditions did not learn at a slower rate; however, they exhibited a lack of generalization in test games where the pipe positions differed from their training environment. On the other hand, the agents trained under random conditions performed very well in the test games, as the testing conditions closely matched their training environment.

In a scenario where we need to retrain an agent to play FlappyBird with high performance across multiple games, we would train our agent under random conditions. Given the variance in performance, we would run the training multiple times and select the best-performing agent.

Chapter 5

Impact of increasing action space

Previous chapter studies the best context to learn when following a DQN algorithm. This chapter presents the study of increasing complexity by expanding the number of actions available to the agent while the next chapter will focus on the introduction of uncertainty in action execution.

In the classic Flappy Bird game, the player has only two possible actions: Flap or Idle, which is the simplest configuration. We could modify the game to include three actions: Idle, Small Flap, and Big Flap. A Big Flap would be used when a significant change in direction is needed, while a Small Flap would be used for minor trajectory adjustments. What if the agent has the choice between 4 or 8 different flap forces? We will discuss the methodology used in this study and present the results of how increasing the number of possible actions impacts the agent's performance and learning velocity.

5.1 Methodology

Multiple learning sessions will be conducted with different numbers of possible actions. For the baseline configuration, we will make the game slightly more challenging by increasing the jump force value from 5 to 9. The value of 9 corresponds to the same value found in the original gameplay.

When we increase the number of actions, the jump force associated with each action has to be different. For the idle action, no jump force is applied. Then the jump force for all other actions is computed using an interpolation based on the total number of possible actions (equation 5.1).

$$J(a) = \frac{J_0 \times i_a}{|\mathcal{A}| - 1} \tag{5.1}$$

Where $J(a)$ is the jump force associated with action a, $\mathcal{A}$ is the action space, and i_a is the index of the action. The index 0 corresponds to idle, and the index $|\mathcal{A}| - 1$ corresponds to the maximum jump force J_0, which is 9 in this case.

Hypothesis

Increased LVI: With more actions to choose from, the decision-making process becomes more complex. The agent needs to learn the optimal action from a larger set of possibilities, which can make the learning process slower. We expect to have a increasing LVI as the number of actions increases.

Increased accuracy: With a refined grid of the possible jump forces, it allows for more precise control over the agent's behavior, potentially leading to better performance once the agent has learned the optimal policy. We expect to have even or increasing performances (DSP) as the number of possible actions increases.

Increased generalization: Larger action space might help the agent generalize better across different states, leading to improved performance in random test environments.

Experiments

We will run 10 training sessions for each number of possible actions in $|\mathcal{A}| = [2, 3, 4, 8, 16]$. As explained earlier, the initial jump force is 9. Since Section 4 demonstrates that training under random pipe conditions is more effective, we will also incorporate random pipe conditions in these sessions. From these experiments, the mean duration, number of solved games, DSP (Duration Surpass Proportion), and LVI (Learning Velocity Index) will be computed and analyzed.

To evaluate the generalization impact, we will run 100 test games with random pipes and a maximum jump force of 9 for each trained agent and report their performances.

5.2 Results

The experiments were initially launched similarly to other experiments with 1000 training games, but this proved insufficient to capture the trends for a high number of actions. Therefore, we relaunched the experiments with 2500 games for number

of possible actions $|\mathcal{A}| \in [3, 4, 8, 16]$. We did not rerun the experiment for $|\mathcal{A}| = 2$ as it had already reached its converged value and would have required significant computational time[1]. For $|\mathcal{A}| = 2$, we reported the asymptotic values from the last 1500 epochs with dashed lines instead of recomputing them on figure 5.2.
While Figure 5.1 gives a first overview of the average duration, it shows that the training has reached its asymptotic values since the mean duration does almost not increase anymore. This is even better illustrated in Figure 5.2, depicting the running mean with a window width of 100 games. We can see that there is a clear drop in terms of performance between $|\mathcal{A}| = 2$ and the other values, which was not expected. We also see that the number of games needed to reach the asymptotic duration varies with $|\mathcal{A}|$.

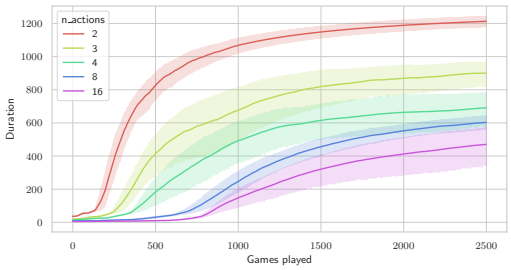

Figure 5.1: Average duration by configuration

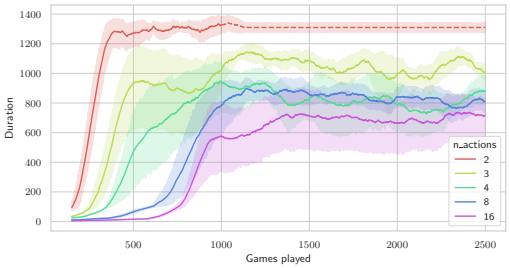

Figure 5.2: Duration running mean (w=100)

As briefly explained before, the learning velocity is clearly impacted by the number of possible actions. This trend is observable in both the classical LVI shown in Figure 5.3 and the normalized LVI in Figure 5.4. As $|\mathcal{A}|$ increases, the number of games needed to reach a certain duration threshold also increases.
This behavior was expected, as the learning velocity is slower when the agent has a larger action space. A larger action space means that the agent has more potential actions to evaluate and choose from at each decision point. This increases the complexity of the learning task, requiring the agent to spend more time exploring and understanding the effects of each action.

[1]To provide an idea of the computation time: an average run duration for $|\mathcal{A}| = 4$ on 2500 games is around 200 minutes, and approximately 110 minutes for $|\mathcal{A}| = 16$.

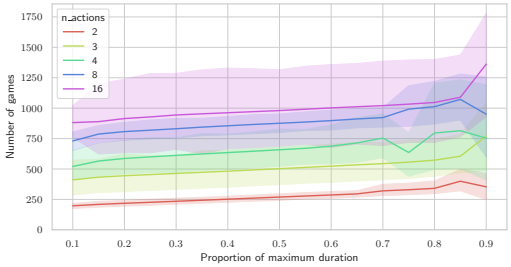

Figure 5.3: LVI

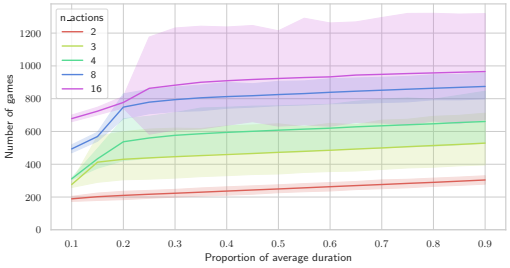

Figure 5.4: Normalized LVI

Figure 5.5 shows the DSP for all values of $|\mathcal{A}|$. Surprisingly, the performance drops when $|\mathcal{A}|$ increases. This was not expected since having more possible actions with various jump forces should allow the agent to take more precise trajectories and therefore have equal or higher performance.
Additionally, we observe that the slope of the curve for high $|\mathcal{A}|$ values is steeper, indicating that agents have difficulties reaching the end of the game. In contrast, when $|\mathcal{A}| = 2$, the differences between DSP at $x = 0.1$ and $x = 0.9$ are smaller, meaning that many of the games where the first pipe is passed, the bird solves the game.
One possible explanation for the drop in performance is the lack of sufficient exploration. With a larger action space, the agent needs more time to explore and find the optimal policy. In these experiments, the epsilon decay value was fixed and constant across all the experiments. If the epsilon decay had been adjusted to account for the larger action space, it might have allowed for more effective exploration, which could have mitigated the drop in performance. However, keeping the epsilon decay constant was necessary to ensure that the LVI indicator remained meaningful.

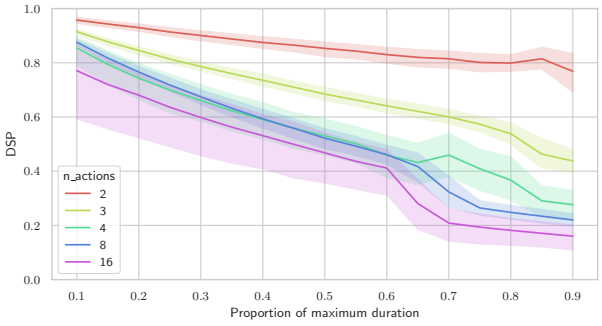

Figure 5.5: DSP by configuration

When testing the best agents from each training session in random games, the

performances align with the training performances in figure 5.6. The mean duration drops as the number of possible actions increases.

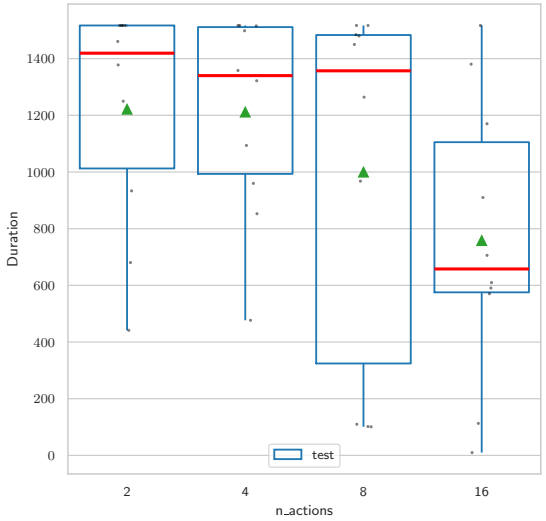

Figure 5.6: Test duration in random games by configuration

5.3 Conclusion

Increasing the number of possible actions for the agent makes the training more complex and results in slower progress. Surprisingly, this increased complexity does not lead to improved performances; instead, it decreases them. This outcome may be explained by a too-short exploration phase, which leads to the discovery of a sub-optimal policy. Ultimately, increasing the number of possible actions does not improve the generalization of the agent. However, this outcome may also be impacted by the sub-optimal policy discovered during training.

Chapter 6

Impact of uncertainty in action execution

The previous chapter focused on how the expansion of the action space affects learning velocity. This section will focus on performance when introducing uncertainty into the game. To introduce uncertainty, we will add randomness to the agent's actions. This randomness is intended to prevent the agent from accurately predicting the next state, as transitions depend on a degree of unpredictability. We achieve this by adding random noise to the jump force of the flap action. The amplitude of the noise will vary, allowing us to control the impact of randomness.

This approach differs from the first experiment (Section 4) where randomness only impacted the positions of the pipes. Here, the actions of the bird are also affected by randomness. In the first experiment, the agent could predict the next state following an action since the rules were fixed. However, in this case, the varying jump force makes it impossible to predict the exact next state.

6.1 Methodology

Multiple learning sessions will be conducted with a random jump force of different amplitudes. The initial jump force is denoted by $J = 5$, corresponding to the jump force in an easy game without randomness. Uniform noise with a specific amplitude will be added each time the agent jumps. The effective jump force J_{rand} is expressed as:

$$J_{rand} = J + (J \times k \times x) = J \times (1 + kx) \tag{6.1}$$

Where x is a random variable following a uniform distribution between -1 and $+1$, k is the proportion factor that determines the weight of the random component, and J represents the original fixed value for the jump force.

A run with a k value of 0 is equivalent to a run with no randomness. These experiments explore the scenario where randomness arises from the action itself. Specifically, when performing the action $a = \text{jump}$, the next state is affected by a uniform distribution.

Hypothesis

Impact on Performances:The value of the factor k significantly impacts the mean duration and the number of solved games as it impacts the DSP indicator. The slope of the DSP curve is expected to be steeper for higher values of k, as it becomes more difficult to reach the end of the game.

Impact on Learning Velocity: The learning velocity is expected to be negatively impacted by the factor k. The rationale is that the agent will be less able to predict the correct next state, leading to more errors and requiring additional time to optimize its network.

Transfer of Performance Under Testing Conditions: An agent trained in highly random conditions (with a high value of k) will perform well under less random conditions (with smaller values of k). Conversely, an agent trained under less random conditions will perform worse compared to an agent trained under highly random conditions.

Experiments

The first experiment will involve running 10 training sessions for different values of $k = [0, 0.5, 1, 1.5, 2, 2.5]$. Again, since Section 4 demonstrates that training under random pipe conditions is more effective, we will also incorporate random pipe conditions in these sessions. The number of possible actions remains 2 as in the original game since previous section 5 showed it was the most optimal. From these experiments, the mean duration, number of solved games, DSP (Duration Surpass Proportion), and LVI (Learning Velocity Index) will be computed and analyzed.

To evaluate the transfer of performance, we will test each trained agent 100 times in three different testing configurations:

- *Baseline configuration*: The jump force is not random ($k = 0$).
  We expect that every agent will perform well, regardless of their training

configuration.

- *Moderate random configuration*: The jump force will be random with a k value of 1. This value is chosen to avoid negative jump force values (if $k > 1$, the J_{rand} can become negative).
  We expect good performance from all agents except those trained under the configuration with $k < 1$.

- *Highly random configuration*: The jump force will be random with a k value of 2.5, which is the highest value of k tested during training.
  We expect to see gradually increasing performance corresponding to the increasing training values of k.

6.2 Results

Figure 6.1 depicts the average duration during the training sessions for each configuration while figure 6.2 the number of solved games. We observe two types of curves: first, the group with k in $[0, 0.5, 1]$ which shows approximately the same performance. An ANOVA test with a significance level of $\alpha = 0.05$ was performed to ascertain whether there were significant variations in mean durations and the number of solved games for these three configurations. The test failed to reject the equality of means (p-value = 0.316), indicating no significant differences in mean duration and mean number of solved games.
The second group consists of configurations with k values higher than 1, where the performance significantly drops as k increases.
The reason behind this separation is that when k is lower than 1, the jump force value is always positive. However, when k surpasses 1, it can lead to negative jump force values, which is contrary to the intended direction of a jump. When the jump force is negative, instead of propelling the agent upwards, it would push the agent downwards. This counterproductive action makes it much more difficult for the agent to navigate through the game, significantly impacting its performance.

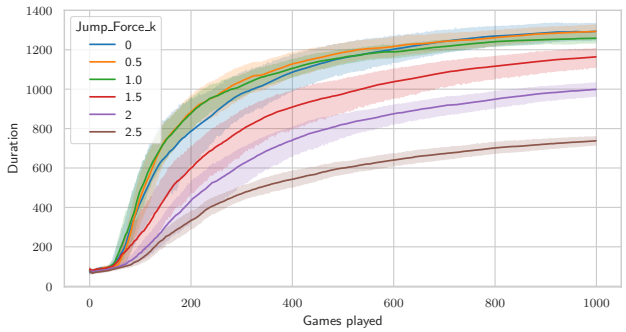

Figure 6.1: Average duration by configuration

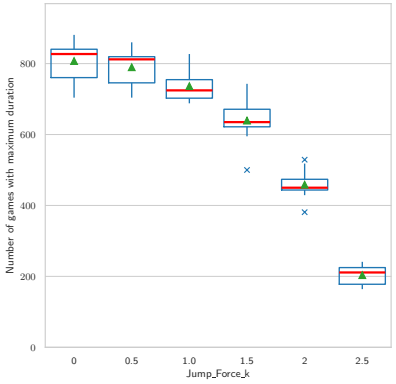

Figure 6.2: Number of solved games

Figure 6.3 also shows two distinct behaviors in the curves. The first group, composed of $k = [0, 0.5, 1]$, exhibits the same learning velocity index (LVI), indicating that k has no influence on the learning velocity for these configurations. The second group, with $k = [1.5, 2, 2.5]$, shows a higher LVI as the threshold increases. This is expected due to the lower performance of these configurations. Therefore, it is essential to examine the normalized LVI as shown in Figure 6.4. We can observe that the normalized LVI is higher for the second group than for the first group, indicating that having k values that produce inverse jumps slows the learning process. However, among those groups, there is no significant relationship between the normalized LVI and the value of k.

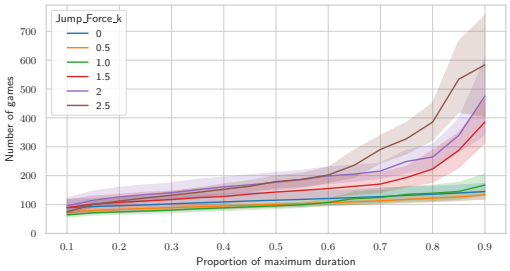

Figure 6.3: LVI

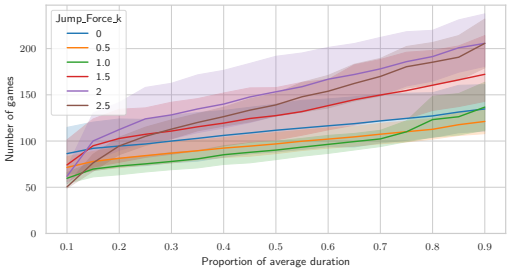

Figure 6.4: Normalized LVI

Concerning the Duration Surpass Proportion (DSP), which indicates the performances after the learning phase, we clearly see that configurations with high values of k are difficult to solve, likely due to the high chance of encountering "impossible jumps".
For lower values of k, there is no significant difference in terms of performance, although the DSP for high threshold values decreases slightly as k increases.

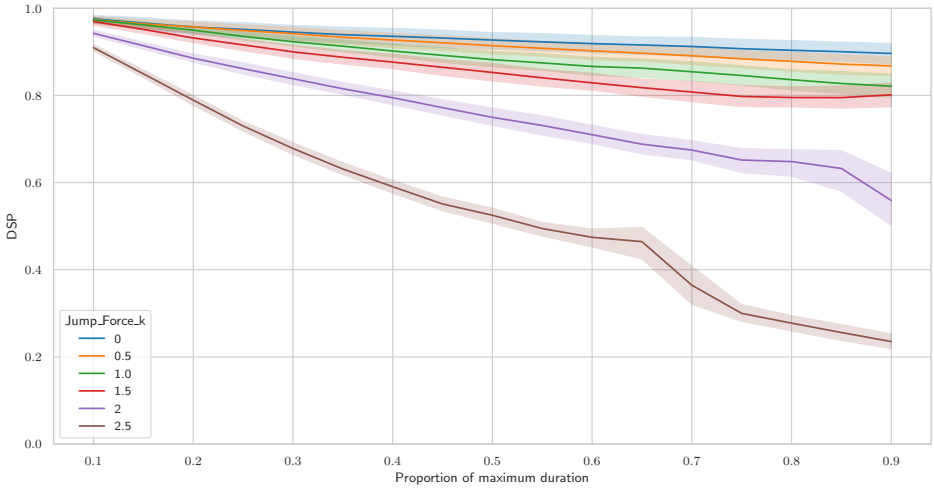

Figure 6.5: DSP

Concerning the transfer of performance, figures 6.6, 6.7, and 6.8 depict the three different configurations with increasing random test factor k from 0 to 2.5.

In fixed conditions (Figure 6.6), we observe that all trained agents show good performance regardless of the training k value. Some disparities are found at $k = 1$ and $k = 2.5$; however, the medians of each configuration are equivalent indicating robustness in equal or less random environnement than their training environnement.

When increasing the test k factor to 1 (figure 6.7), we observe that test performance of agents trained under conditions with $k \geq 1$ are higher than for those with $k < 1$, as expected. This hypothesis is further confirmed for the test values $k = 2.5$, where agents trained with low k values perform poorly. Since agents trained with high k values perform well regardless of the test k value, we can say that there is a strong transfer of learning from high variability to low variability environments. In contrast, agents trained with low k values struggle when tested with higher k values, highlighting that training in less variable environments does not prepare agents well for more variable conditions.
This indicates that exposure to higher variability during training enhances the agent's ability to generalize and perform well in more diverse situations.

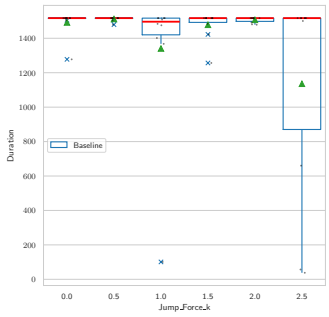

Figure 6.6: Mean duration in fixed conditions ($k = 0$)

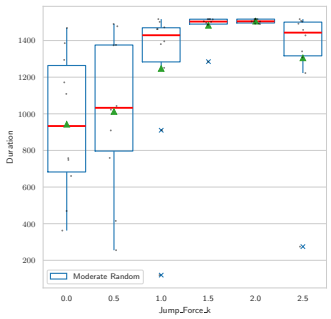

Figure 6.7: Mean duration in moderate random conditions ($k = 1$)

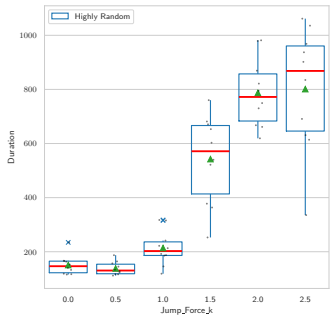

Figure 6.8: Mean duration in highly random conditions ($k = 2.5$)

6.3 Conclusion

For most of the results, we observed two distinct groups separated by $k = 1$, each exhibiting different behaviors. The factor k does not significantly impact the mean duration for the first group ($k < 1$), whereas it has a significant effect for the second group ($k > 1$).
As for the performances, the Learning Velocity Index (LVI) and the Duration Surpass Proportion (DSP) of the first group are not impacted by the value of k, while the second group is negatively affected.
The separation of the two behaviors is explained by the possibility of negative jumps when $k > 1$, which may lead to unsolvable games. The higher the value of k when $k > 1$, the higher the chance of encountering an unsolvable game, resulting in poorer performances. One could evaluate the probability of encountering an unsolvable game given the value of k and assess if the agent performs accordingly to this probability or not.
We can thus conclude that as long as the uncertainty (represented by k) does not change the game dynamics, it has a minimal effect on performance and learning velocity. After testing agents in different configurations from their training environments, we found that there exists a strong transfer from high variability environments to less random environments. In other words, training agents in more variable conditions (higher k values) enhances their generalization capabilities and performance across different environments. This suggests that incorporating randomness during training can lead to more robust and adaptable agents.

Chapter 7

General conclusion

After training a Deep Q learning algorithm in different environment configurations, it was found that training an agent with varied environment at each epoch, rather than the same environment, improves the agent's performance and ability to generalize without affecting its learning efficiency. It was found that increasing the complexity of the environment by adding possible actions negatively impacted both the training speed and performance. Finally, increasing the uncertainty of action execution decreased the training performance but enhanced the agent's generalization capabilities and performance across different environments.

Increasing the action space negatively impacted the performances, it was unexpected since increasing the number of possible actions should increases the precision of the agent trajectory resulting in better performances. One possible explanation is a too short exploration phase. One could make a new experiment with longer exploration phase.

A first limitation of this work concerns the optimality of the training hyperparameter set. Although we attempted to find the best set, it is not guaranteed that we achieved the optimal configuration for each training setup.

The experiments were conducted on a simple version of a DQN agent due to computational power constraints. Future work could involve using a more sophisticated DQN agent, such as the Rainbow DQN from Hessel et al., 2017, or using an RGB state representation for more generalized results.

Additionally, this thesis lacks a characterization of the probability of solving the game given the uncertainty level discussed in Section 6. Comparing the actual performances with the theoretical performances could provide insights into whether the agent is performing optimally or if there is potential for improved performance.

More generally, this research was limited to an experimental approach due to the lack of established theory in the literature. Consequently, the results may be highly specific to the game used in the experiment. Based on these experimental results, future research could involve developing a theoretical framework independent of a specific game, focusing on learning complexity based on the size of the input and output spaces.

An important perspective for future research based on this work is the dual focus on evaluating the impact of both the DQN architecture and the environmental conditions and their interactions. While most studies focus on the algorithm's architecture, it is still relevant to investigate how different environmental configurations influence the training and performance of reinforcement learning algorithms, as well as the role of randomness in these algorithms.

Appendix

The source code can be found on GitHub at https://github.com/CorentinVermeulen/epl_flappybird

Parameter	**Value**	**Description**
Epochs	1000[1]	Number of training games
Memory Size	100 000	Size of the replay buffer
EPS_Start	0.9	Probability of doing a random action at the start
EPS_End	0.001	Minimal probability of doing a random action
EPS_Decay	2500	Speed at which the probability decreases
TAU	0.01	Proportion of policy network copied to target network
Layers_Sizes	[256,256,256,256]	Network sizes of the DQN
Gamma	0.99	Attention given to future rewards
Batch_size	256	
LR	$1e-4$	Adam learning rate

Table 1: Hyper parameters for experiences 1 to 3

Parameter	**Value**	**Description**
Gravity	1	Force pulling the bord to the ground
Jump Force (easy)	5	Force making the bird jump (in easy game)
Jump Force (hard)	9	Force making the bird jump (in hard game)
Random pipes	[True, False]	Vetical positions of the pipes
Jumpforce_k	[0, 0.5, 1, 1.5, 2, 2.5]	Amplitude of the randomness in the jump force
n_actions	[2, 3, 4, 87, 16]	Number of possible actions

Table 2: Game parameters for FlappyBird environment

Bibliography

Ceron, J. S. O., & Castro, P. S. (2021). Revisiting Rainbow: Promoting more insightful and inclusive deep reinforcement learning research [ISSN: 2640-3498]. *Proceedings of the 38th International Conference on Machine Learning*, 1373–1383. Retrieved May 9, 2024, from https://proceedings.mlr.press/v139/ceron21a.html

Dabney, W., Rowland, M., Bellemare, M. G., & Munos, R. (2017, October). Distributional Reinforcement Learning with Quantile Regression [arXiv:1710.10044 [cs, stat]]. https://doi.org/10.48550/arXiv.1710.10044

Degrave, J., Felici, F., Buchli, J., Neunert, M., Tracey, B., Carpanese, F., Ewalds, T., Hafner, R., Abdolmaleki, A., de las Casas, D., Donner, C., Fritz, L., Galperti, C., Huber, A., Keeling, J., Tsimpoukelli, M., Kay, J., Merle, A., Moret, J.-M., ... Riedmiller, M. (2022). Magnetic control of tokamak plasmas through deep reinforcement learning [Publisher: Nature Publishing Group]. *Nature*, *602*(7897), 414–419. https://doi.org/10.1038/s41586-021-04301-9

Ding, Z., Huang, Y., Yuan, H., & Dong, H. (2020). Introduction to Reinforcement Learning. In H. Dong, Z. Ding, & S. Zhang (Eds.), *Deep Reinforcement Learning: Fundamentals, Research and Applications* (pp. 47–123). Springer. https://doi.org/10.1007/978-981-15-4095-0_2

Fortunato, M., Azar, M. G., Piot, B., Menick, J., Osband, I., Graves, A., Mnih, V., Munos, R., Hassabis, D., Pietquin, O., Blundell, C., & Legg, S. (2019, July). Noisy Networks for Exploration [arXiv:1706.10295 [cs, stat]]. https://doi.org/10.48550/arXiv.1706.10295
Comment: ICLR 2018.

Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks [ISSN: 1938-7228]. *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, 249–256. Retrieved May 10, 2024, from https://proceedings.mlr.press/v9/glorot10a.html

Hall, P. (1987). On Kullback-Leibler Loss and Density Estimation [Publisher: Institute of Mathematical Statistics]. *The Annals of Statistics*, *15*(4), 1491–1519. Retrieved April 4, 2024, from https://www.jstor.org/stable/2241687

Hasselt, H. (2010). Double Q-learning. *Advances in Neural Information Processing Systems*, *23*. Retrieved May 8, 2024, from https://papers.nips.cc/paper_files/paper/2010/hash/091d584fced301b442654dd8c23b3fc9-Abstract.html

Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., & Silver, D. (2017, October). Rainbow: Combining Improvements in Deep Reinforcement Learning [arXiv:1710.02298 [cs]]. https://doi.org/10.48550/arXiv.1710.02298
Comment: Under review as a conference paper at AAAI 2018.

Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, *2*(5), 359–366. https://doi.org/10.1016/0893-6080(89)90020-8

Ioffe, S., & Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift [ISSN: 1938-7228]. *Proceedings of the 32nd International Conference on Machine Learning*, 448–456. Retrieved June 2, 2024, from https://proceedings.mlr.press/v37/ioffe15.html

Kassel, R. (2021, April). Perceptron : Qu'est-ce que c'est et à quoi ça sert ? Retrieved April 4, 2024, from https://datascientest.com/perceptron

Kingma, D. P., & Ba, J. (2017, January). Adam: A Method for Stochastic Optimization [arXiv:1412.6980 [cs]]. https://doi.org/10.48550/arXiv.1412.6980
Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

Li, Y. (2018, October). Deep Reinforcement Learning [arXiv:1810.06339 [cs, stat]]. https://doi.org/10.48550/arXiv.1810.06339
Comment: Under review for Morgan & Claypool: Synthesis Lectures in Artificial Intelligence and Machine Learning.

Lobbezoo, A., Qian, Y., & Kwon, H.-J. (2021). Reinforcement Learning for Pick and Place Operations in Robotics: A Survey [Number: 3 Publisher: Multidisciplinary Digital Publishing Institute]. *Robotics*, *10*(3), 105. https://doi.org/10.3390/robotics10030105

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., & Hassabis, D. (2015). Human-level control through deep reinforcement learning [Publisher: Nature Publishing Group]. *Nature*, *518*(7540), 529–533. https://doi.org/10.1038/nature14236

Nogueira, G. (2024, April). Talendar/flappy-bird-gym [original-date: 2021-02-07T00:05:42Z]. Retrieved April 15, 2024, from https://github.com/Talendar/flappy-bird-gym

Paszke, A., & Towers, M. (2017, March). Reinforcement Learning (DQN) Tutorial — PyTorch Tutorials 2.2.2+cu121 documentation. Retrieved April 16, 2024,

from https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

Ramachandran, P., Zoph, B., & Le, Q. V. (2017, October). Searching for Activation Functions [arXiv:1710.05941 [cs]]. Retrieved May 10, 2024, from http://arxiv.org/abs/1710.05941
Comment: Updated version of "Swish: a Self-Gated Activation Function".

Rosenblatt, F. (1962). *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms* [Google-Books-ID: 7FhRAAAAMAAJ]. Spartan Books.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors [Publisher: Nature Publishing Group]. *Nature*, *323*(6088), 533–536. https://doi.org/10.1038/323533a0

Sarker, I. H. (2021). Deep Learning: A Comprehensive Overview on Techniques, Taxonomy, Applications and Research Directions. *SN Computer Science*, *2*(6), 420. https://doi.org/10.1007/s42979-021-00815-1
img nn vs dnn.

Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2016, February). Prioritized Experience Replay [arXiv:1511.05952 [cs]]. https://doi.org/10.48550/arXiv.1511.05952
Comment: Published at ICLR 2016.

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 [Publisher: Nature Publishing Group]. *Nature*, *529*(7587), 484–489. https://doi.org/10.1038/nature16961

Sutton, R. S., & Barto, A. (2020). *Reinforcement learning: An introduction* (Second edition). The MIT Press.

Sutton, R., & Barto, A. (1998). Reinforcement Learning: An Introduction [Conference Name: IEEE Transactions on Neural Networks]. *IEEE Transactions on Neural Networks*, *9*(5), 1054–1054. https://doi.org/10.1109/TNN.1998.712192

Tai, V., Tan, Y., Abd Rahman, N. F., Chia, C., Zhakiya, M., & Saw, L. (2021). A Novel Power Curve Prediction Method for Horizontal-Axis Wind Turbines Using Artificial Neural Networks. *Energy Engineering*, *118*, 507–516. https://doi.org/10.32604/EE.2021.014868

van Hasselt, H., Doron, Y., Strub, F., Hessel, M., Sonnerat, N., & Modayil, J. (2018, December). Deep Reinforcement Learning and the Deadly Triad [arXiv:1812.02648 [cs]]. Retrieved April 3, 2024, from http://arxiv.org/abs/1812.02648

Vincent, J. (2014). Flappy Bird's Dong Nguyen: 'My games are designed for offline competition' | The Independent. *The Independent*. Retrieved May 7, 2024, from https://www.independent.co.uk/tech/flappy-bird-the-horror-and-despair-of-the-internet-s-most-viral-video-game-9110132.html

Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M., & de Freitas, N. (2016, April). Dueling Network Architectures for Deep Reinforcement Learning [arXiv:1511.06581 [cs]]. https://doi.org/10.48550/arXiv.1511.06581 Comment: 15 pages, 5 figures, and 5 tables.

Yang, H., Liu, X.-Y., Zhong, S., & Walid, A. (2021). Deep reinforcement learning for automated stock trading: An ensemble strategy. *Proceedings of the First ACM International Conference on AI in Finance*, 1–8. https://doi.org/10.1145/3383455.3422540

Zhang, H., & Yu, T. (2020). Taxonomy of Reinforcement Learning Algorithms. In H. Dong, Z. Ding, & S. Zhang (Eds.), *Deep Reinforcement Learning: Fundamentals, Research and Applications* (pp. 125–133). Springer. https://doi.org/10.1007/978-981-15-4095-0_3

Zhang, S., Li, H., Wang, M., Liu, M., Chen, P.-Y., Lu, S., Liu, S., Murugesan, K., & Chaudhury, S. (2023). On the Convergence and Sample Complexity Analysis of Deep Q-Networks with ϵ-Greedy Exploration. *Advances in Neural Information Processing Systems*, *36*, 13064–13102. Retrieved May 27, 2024, from https://proceedings.neurips.cc/paper_files/paper/2023/hash/2a91de02871011d0090e662ffd6f2328-Abstract-Conference.html

Zhao, X., Gu, C., Zhang, H., Yang, X., Liu, X., Tang, J., & Liu, H. (2021). DEAR: Deep Reinforcement Learning for Online Advertising Impression in Recommender Systems [Number: 1]. *Proceedings of the AAAI Conference on Artificial Intelligence*, *35*(1), 750–758. https://doi.org/10.1609/aaai.v35i1.16156

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**