École polytechnique de Louvain

Impact of Game Variability on Reinforcement Learning Complexity in Video Games

A Case Study of Geometry Dash

Author: **Guillaume FONTAINE**
Supervisors: **Julien HENDRICKX, Francois-Xavier STANDAERT**
Reader: **Laurent JACQUES**
Academic year 2023–2024
Master [120] in Mathematical Engineering

Abstract

Most advancements in the Reinforcement Learning (RL) field consist of increasing performances and benchmarks. This thesis aims to better understand the RL complexity and innovates by investigating the impact of reinforcement learning complexity within variable video game environments. We consider, as our game model, the popular 2D platformer, *Geometry Dash*, to probe the dynamics of RL under varying conditions. The study examines how different degrees of freedom in the game, game parameter settings, and the introduction of random disturbances influence learning processes. This research utilizes a deep Q-learning (DQL) algorithm combined with a Convolutional Neural Network (CNN) architecture, proven effective in Atari 2600 games. Employing an experimental methodology, the findings provide detailed insights into the strengths and weaknesses of artificial intelligence (AI) learning mechanisms. Ultimately, this research aims to enhance our understanding of RL complexity, paving the way for extended investigations with new algorithms and further experiments. This could hopefully lead to a theoretical framework for understanding how RL complexity evolves with game variability.

Acknowledgements

This master's thesis was supervised by Julien Hendrickx, Professor at the Department of Mathematical Engineering, and Francois-Xavier Standaert, Professor at the Department of Electrical Engineering, both at UCLouvain. Their keen interest in understanding the complexity of reinforcement learning in video games initiated this work. I extend my deepest gratitude to them for their invaluable support and insightful feedback.

I am also grateful to Corentin Vermeulen, a fellow master's student at UCLouvain, who shared the same thesis topic. His discussions, advice, and support were instrumental throughout the thesis period.

I appreciate the group of students from the local a.025 at the Euler building. Their studious yet warm and supportive environment provided a productive framework that greatly aided my work.

Thank you to CECI clusters, as computational resources have been provided by the Consortium des Équipements de Calcul Intensif (CÉCI), funded by the Fonds de la Recherche Scientifique de Belgique (F.R.S.-FNRS) under Grant No. 2.5020.11 and by the Walloon Region.

Special thanks to John Aoga, data scientist and researcher at UCLouvain, who assisted me with all the procedures and issues encountered on the CECI clusters.

Finally, I want to express my heartfelt appreciation to all my friends and family. Their unwavering belief in me and the quality of my work has been a constant source of motivation throughout this journey.

Contents

1 Introduction 1

2 Technical Background 3
2.1 Reinforcement learning . . . 3
2.1.1 Value Iteration . . . 4
2.1.2 Policy Iteration . . . 5
2.1.3 Q-learning . . . 5
2.1.4 Deep Q-learning . . . 6
2.1.5 Policy Gradient Algorithm . . . 6
2.2 Deep Neural Networks . . . 7
2.2.1 Neuron Structure . . . 7
2.2.2 Neural Networks and Deep Neural Networks . . . 8
2.2.3 Convolutional Networks . . . 10

3 Methodology 13
3.1 Geometry Dash Environment . . . 13
3.2 Game Model . . . 13
3.3 Baseline AI design . . . 16
3.3.1 Deep Q-Learning Algorithm . . . 16
3.3.2 State Representation and Transition . . . 17
3.3.3 Neural Network Architectures . . . 18
3.3.4 Training Levels Baseline . . . 21
3.3.5 Reward Function Design and Hyperparameter Tuning . . . 21
3.4 Computational Resources . . . 28
3.5 Metrics for Complexity . . . 29
3.6 Experimental Objectives and Tracks . . . 30

4 Results **31**

4.1 Preliminary Definitions and Computations . . . 32

4.2 Deterministic Levels with Only Spike Obstacles Regularly Spaced . . . 34

4.3 Adding Random Disturbances to the Previous Deterministic Levels . . . 36

4.3.1 Adding Distance Disturbances Between Spikes . . . 36

4.3.2 Adding Jumping Disturbances . . . 40

4.3.3 Adding Gravity Disturbances . . . 45

4.3.4 Adding Player Horizontal Speed Disturbances . . . 46

4.3.5 Adding Moving Spike Disturbances . . . 47

4.4 Increasing the Action Space Cardinality . . . 52

4.4.1 Increasing Action Space Cardinality in a Deterministic Setting . . . 52

4.4.2 Combining Increasing Action Space Cardinality with Jump Disturbances . . . 54

4.5 Adding New Objects to the Game . . . 57

4.5.1 Adding Platform and Orb Objects to the Game . . . 57

4.5.2 Adding Coins to the Game . . . 62

5 Conclusion and Future Work **65**

A Baseline Training Level Examples **69**

Chapter 1

Introduction

In modern society, despite significant advancements in science, understanding how the brain functions and how humans learn remains a challenge. Interestingly, the mechanisms of reward and punishment, effective from a young age, have inspired the development of reinforcement learning (RL). In this domain, an agent interacts with a dynamic environment and receives rewards based on the effectiveness of its actions. The ultimate aim of RL is to develop policies that maximize cumulative rewards over time, guiding the agent towards optimal behavior through trial and error [1, 2].

This approach demands substantial computational resources. However, the development of computational resources has facilitated significant growth in RL in recent years, propelled by advancements in both theoretical frameworks and practical applications. A notable example is mastering the game of Go [3]. Another critical area is video games, where RL has proven pivotal in navigating complex decision-making environments and serves as a benchmark for progress in the field. A landmark in this domain was the creation of the Deep Q-Network (DQN), which significantly advanced the integration of deep learning with reinforcement learning. Introduced in 2013, this approach enabled RL agents to surpass human performance in various Atari 2600 games, setting a new standard for AI in gaming [4]. This success demonstrated the potential of RL agents to effectively manage discrete action spaces.

Building upon these foundational advances, recent work by Schwarzer et al. introduced a value-based RL agent named BBF, which achieves super-human performance in the Atari 100K benchmark with remarkable sample efficiency [5]. The BBF model enhances the scalability of neural networks used for value estimation through several innovative design choices that facilitate this scaling in a resource-efficient manner.

The scope of RL applications has significantly broadened, exemplified by projects such as OpenAI Five, which tackled the strategic complexities of Dota 2. This project demonstrated that RL could achieve sophisticated coordination and strategic gameplay at a professional level [6]. Building on these achievements, Proximal Policy Optimization (PPO) was introduced in 2017. This method provided a crucial enhancement by facilitating stable and efficient learning in environments with continuous action spaces—something not supported by DQN. PPO's ability to manage continuous action spaces without the need for discretizing actions and state spaces marked a significant advancement, broadening its applicability in diverse gaming scenarios [7].

While significant efforts have been directed towards enhancing the performance of reinforcement learning (RL) agents and benchmarking their capabilities, a deep comprehension of the complexity inherent in reinforcement learning remains underexplored. Questions such as what is the learning complexity in deep RL, its dependencies, and the feasibility of learning certain games using these techniques are critical. The concept of learning complexity is broad and we define it using several metrics in Section 3.5. These include the average number of training games required to achieve game mastery, as well as the critical exploration rates during the learning phase. In order to understand what affects the learning complexity and its limits, this thesis explores how RL algorithm performances evolve under game variability, such as modifications to game parameters or the introduction of random disturbances. These modifications can profoundly influence the agent's learning trajectory and the strategies it develops, thus impacting the overall effectiveness and efficiency of the algorithm.

Given the lack of comprehensive studies that address these questions, this thesis adopts an innovative approach by experimentally quantifying the impact of game variability on learning complexity metrics. We prefer an experimental approach to a theoretical one because theoretical frameworks for understanding learning complexity in reinforcement learning (RL) are still underdeveloped. In the realm of optimization and probabilistic learning, multi-armed bandits (MABs) have been effectively utilized to theorize about decision-making processes by balancing the exploration-exploitation trade-off in static settings [8, 9]. However, the dynamic nature of environments such as video games—where both the state and potential actions continuously evolve—presents significant challenges. Applying MAB theories directly to establish complexity bounds in such dynamic settings, especially in complex games, is particularly challenging.

In this study, we use the game *Geometry Dash* as Reinforcement Learning (RL) testing environment [10]. Geometry Dash is a simple 2D platformer with straightforward mechanics, yet it offers the potential for intricate complexity. The game's levels are highly customizable, allowing for the introduction of varying complexities and rules that test the adaptability and efficacy of RL algorithms. Since we did not find any publicly available and well-functioning AI that uses RL for *Geometry Dash*, we develop our own AI using Deep Q-learning, similar to approaches used in Atari research [4]. We first show the impact on learning complexity when random disturbances are added to the game. Additionally, we analyze how increasing the cardinality of the action space affects learning complexity. We finally assess the effects of introducing new objects or obstacles on learning complexity. This study aims to broaden our understanding of reinforcement learning complexity under game variability, by generalizing the results within the context of game *Geometry Dash* to other video games. It also seeks to pave the way for further research in this area.

The technical background chapter of this thesis provides a deeper understanding of RL, discussing some of the main existing RL algorithms, and explaining the concepts of deep neural networks and convolutional neural networks. The AI design and game rules for *Geometry Dash* are then detailed in the methodology chapter. Finally, the different experiments and their results are presented in the Results chapter.

Chapter 2

Technical Background

This chapter offers a comprehensive technical background on the application of reinforcement learning (RL) in video games. It outlines the fundamental concepts of RL, reviews key algorithms that have influenced the field, and examines the integration of RL with deep learning techniques, particularly Convolutional Neural Networks (CNNs).

2.1 Reinforcement learning

Reinforcement Learning (RL) is a type of machine learning paradigm where an agent learns to make decisions by interacting with an environment. The goal of the agent is to take actions in such a way that it maximizes a cumulative reward signal over time[1]. Unlike supervised learning, where the algorithm is trained on a labeled dataset, RL involves learning from trial and error.

In RL, the agent observes the current state s_t of the environment at time t, takes an action a_t, and receives feedback in the form of a reward r_t and the next state s_{t+1}, see figure 2.1. The agent's objective is to learn a policy, which is a mapping from states to actions, that maximizes the expected cumulative reward over time. The learning process often involves exploring different actions, learning from their outcomes, and adjusting the policy accordingly.

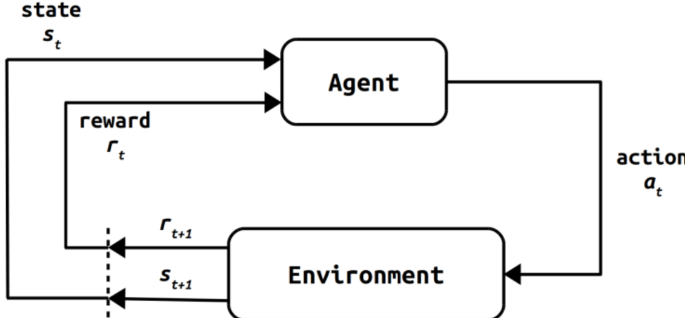


Figure 2.1: Reinforcement learning scheme : Interaction between the agent and the environment

The key components of a reinforcement learning system thus include:

- **Agent:** The decision-maker or learner that interacts with the environment.
- **Environment:** The external system with which the agent interacts. The environment provides feedback to the agent based on its actions.
- **State:** A representation of the current situation or configuration of the environment.
- **Action:** The decision or move made by the agent in response to a given state.
- **Reward:** A numerical signal indicating the immediate benefit or cost of an action taken by the agent.

The learning process in RL is often modeled as a Markov Decision Process (MDP), which is a mathematical framework that formalizes the RL problem. It first defines

- A transition probability $P(s_{t+1}|s_t, a_t)$, representing the probability of transitioning to state s_{t+1} given that the current state is s_t and action a_t is taken.
- A reward function $R(s_t, a_t, s_{t+1})$ representing the immediate reward obtained after taking action a_t in state s_t and transitioning to state s_{t+1}.

The agent then follows a policy $\pi(a_t|s_t)$ representing the probability of taking action a_t in state s_t. The agent's objective is to find an optimal policy $\pi^\star$ that maximizes the expected sum of rewards over time

$$\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right] \tag{2.1}$$

where $0 < \gamma < 1$ is a discount factor and r_t is the immediate reward at time t. This can be achieved through various algorithms. The main ones are presented below. For each one, we call $V(s_t)$ the value associated to state s_t. The bigger the value, the better the state s_t should be. Here as follows is a non-exhaustive list of popular reinforcement learning algorithms.

2.1.1 Value Iteration

The Value Iteration algorithm is a dynamic programming algorithm used in reinforcement learning to compute the optimal value function and policy for a Markov Decision Process (MDP). It was first presented by [11]. In the value iteration method, we would like to obtain the value of a state $V(s_t)$ at time t satisfying the Bellman's equation

$$V(s_t) = \max_{\pi} \mathbb{E}\left[r_t + \gamma V(s_{t+1})\right] \tag{2.2}$$

where $0 < \gamma < 1$ is a discount factor. In practice, we initialize the values V for each state arbitrary and then iterate over a state experience s_t as

$$V^{\text{new}}(s_t) = \max_{a_t} \sum_{s'} P(s'|s_t, a_t)\left(R(s_t, a_t, s') + \gamma V^{\text{old}}(s')\right) \tag{2.3}$$

which will converge to $V^\star$ satisfying the Bellman's recursive equation over time. The optimal policy is then given by

$$\pi^\star(s_t) = \text{argmax}_{a_t} \sum_{s'} P(s'|s_t, a_t) \left(R(s_t, a_t, s') + \gamma V^\star(s')\right) \tag{2.4}$$

The policy is thus deterministic, we only choose the action maximizing the expression above. Value iteration converges to the optimal value function and is straightforward to understand and implement. However, it is computationally expensive for large state spaces and assumes complete knowledge of the environment dynamics.

2.1.2 Policy Iteration

Policy Iteration is an iterative algorithm for solving Markov Decision Processes (MDPs) and finding the optimal policy. It was also presented by [11]. The algorithm alternates between two main steps: policy evaluation and policy improvement. The goal is to find a policy that is both stable and optimal.

Step 1: Policy Evaluation

The first step is to evaluate the current policy's value function. The value function is denoted as $V^\pi(s_t)$, representing the expected cumulative reward starting from state s_t at time t and following policy π, which deterministically gives the action to take.

$$V^\pi(s_t) = \sum_{s'} P(s'|s_t, \pi(s_t)) \left(R(s_t, \pi(s_t), s') + \gamma V^\pi(s')\right) \tag{2.5}$$

where $0 < \gamma < 1$ is the discount factor.

Step 2: Policy Improvement

The second step is to improve the policy based on the current value function. The policy is denoted as $\pi(s)$ and is updated by selecting the action that maximizes the expected cumulative reward in each state:

$$\pi^{\text{new}}(s_t) = \text{argmax}_{a_t} \sum_{s'} P(s'|s_t, a_t) \left(R(s_t, a_t, s') + \gamma V^{\pi^{\text{old}}}(s')\right) \tag{2.6}$$

This algorithm guarantees convergence to the optimal policy and this convergence is typically faster than the value iteration. However, it has the same disadvantages as value iteration. It is computationally challenging and assumes again the complete knowledge of the environment dynamics.

2.1.3 Q-learning

In Q-learning, we define $Q(s_t, a_t)$ the quality of a state/action pair at time t, defined by

$$\begin{aligned} Q(s_t, a_t) &= \mathbb{E}\left(R(s_t, a_t, s_{t+1}) + \gamma V(s_{t+1})\right) \\ &= \sum_{s'} P(s'|s_t, a_t) \left(R(s_t, a_t, s') + \gamma V(s')\right) \end{aligned} \tag{2.7}$$

The value associated to a state s_t is then given by

$$V(s_t) = \max_{a_t} Q(s_t, a_t) \tag{2.8}$$

which corresponds to the quality of that state paired with the action maximizing that quality. In the same way, we define the policy as

$$\pi(a_t|s_t) = \text{argmax}_{a_t} Q(s_t, a_t) \tag{2.9}$$

We compute that quality $Q(s_t, a_t)$ state/action pair iteratively using game experience. When at time t we encounter state s_t and we decide to take action a_t, the quality function is updated as

$$Q^{\text{new}}(s_t, a_t) = Q^{\text{old}}(s_t, a_t) + \alpha \left(r_t + \gamma \max_{a_t} Q^{\text{old}}(s_{t+1}, a_t) - Q^{\text{old}}(s_t, a_t) \right) \tag{2.10}$$

where $\alpha > 0$ is a chosen learning rate. Q-learning is a model-free approach, suitable for environments with unknown dynamics. It can handle large state and action spaces. Nevertheless, the algorithm is sensitive to hyperparameter choices and may converge slowly, especially in noisy environments.

2.1.4 Deep Q-learning

Deep Q-learning is essentially the same as Q-learning, where the quality function is computed using a deep neural network that takes as input an instance of the state s_t at time t and outputs $Q(s_t, a_t)$ for all available actions a_t. Calling θ the set of parameters related to the deep neural network, we then write $Q(s_t, a_t) \to Q(s_t, a_t, \theta)$. The neural network loss for instance associated to the mean square error becomes

$$\mathcal{L} = \mathbb{E}\left[\left(r_t + \gamma \max_{a_t} Q(s_{t+1}, a_t, \theta) - Q(s_t, a_t, \theta) \right)^2 \right] \tag{2.11}$$

We can then apply a stochastic gradient descent on the parameters θ using back-propagation on a set of state/action pair (s_t, a_t) obtained from game experience. This algorithm extends Q-learning to handle high-dimensional state spaces using neural networks. However, it requires careful tuning of hyperparameters to avoid overestimation bias. Silver et al. demonstrated the power of deep neural networks in reinforcement learning by applying them to master the game of Go [3].

2.1.5 Policy Gradient Algorithm

Policy Gradient Methods are a class of reinforcement learning algorithms that directly optimize the policy, which is a mapping from states to actions. It was introduced by [12]. The key idea is to parameterize the policy and then adjust the parameters to maximize the expected cumulative reward. A basic version of it would be to apply the five next steps :

1) <u>Initialize :</u>

Initialize the policy parameters θ randomly or with some pre-defined values. Set the learning rate α, and other hyperparameters.

2) Collect Trajectories :

Generate a set of trajectories by interacting with the environment using the current policy. A trajectory is a sequence of states, actions, and rewards: $\tau = \{(s_0, a_0, r_0), (s_1, a_1, r_1), ..., (s_T, a_T, r_T)\}$.

3) Compute Policy Gradient :

Compute the policy gradient, which is the gradient of the expected cumulative reward with respect to the policy parameters:

$$\nabla_\theta J(\theta) = \mathbb{E}_\tau \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t).R(\tau) \right] \tag{2.12}$$

where $J(\theta)$ is the expected cumulative reward, $\pi_\theta(a_t|s_t)$ is the probability of taking action a_t in state s_t under policy π_θ, and $R(\tau)$ is the total reward of trajectory τ.

4) Update Policy Parameters :

Update the policy parameters using gradient ascent:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta) \tag{2.13}$$

5) Repeat :

Repeat steps 2-4 for multiple iterations or until convergence.

This kind of algorithm directly optimizes the policy without needing value functions, and it can handle high-dimensional action spaces. Still, high variance in gradient estimates might be faced. Also, it can be computationally expensive like value and policy iteration methods.

2.2 Deep Neural Networks

Deep Neural Networks (DNNs) are a class of machine learning models inspired by the structure and functioning of the human brain. They are designed to learn hierarchical representations of data through multiple layers of interconnected neurons. All fundamental related notions are well covered by [13].

2.2.1 Neuron Structure

A neuron, in the context of artificial neural networks and inspired by biological neurons, is the basic building block of these networks. Neurons are the fundamental processing units that receive and transmit information. The fundamental operation of a neuron involves receiving inputs, applying weights, summing them up, and passing the result through an activation function. Mathematically, the output y of a neuron is given by:

$$y = f(w_1 \cdot x_1 + w_2 \cdot x_2 + \ldots + w_n \cdot x_n + b) \tag{2.14}$$

Here, $x_1, x_2, \ldots, x_n$ are the input values, $w_1, w_2, \ldots, w_n$ are the corresponding weights, b is the bias term, and f is the activation function[1]. The general structure of a neuron is illustrated on figure 2.2. Note on the figure the bias b is assigned a weight w_0, we will consider it in most cases to be 1.

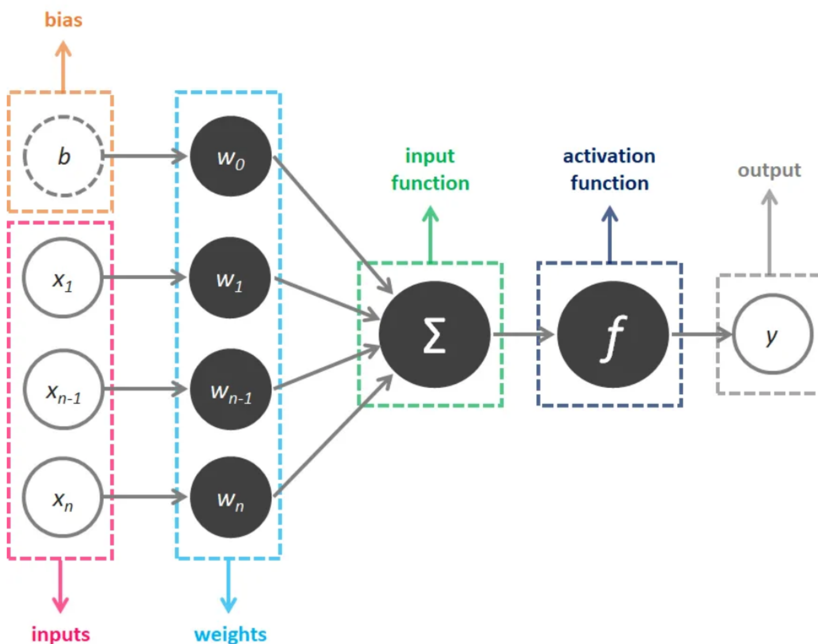


Figure 2.2: General neuron structure

2.2.2 Neural Networks and Deep Neural Networks

A basic neural network consists of layers of interconnected nodes, or neurons, see figure 2.3. The input layer receives the initial data, and the output layer produces the final prediction. Hidden layers between the input and output layers allow the network to capture intricate features. A Deep Neural Network extends the concept of a basic neural network by introducing multiple hidden layers. The depth of the network allows it to learn complex and hierarchical features from the data.

Feedforward Operation

In a deep neural network, we call $a_j^{(l)}$ the output of activation from j^{th} neuron in layer number l. We denote $a^{(l)}$ the output vector of layer l. In a similar way, we define $b^{(l)}$ the bias vector of layer l. Finally, the weight matrix associated to layer is denoted by $W^{(l)}$, where $W_{j,i}^{(l)}$ is the weight associated to the link between the ith neuron in layer $l-1$ and the jth neuron in layer l.

The feedforward operation in a deep neural network involves propagating the input through the layers to produce an output. For each layer l, the output $a^{(l)}$ is calculated as:

[1] Popular activation functions are : sigmoid, ReLU, tanh, Leaky ReLU, PreLU and ELU.

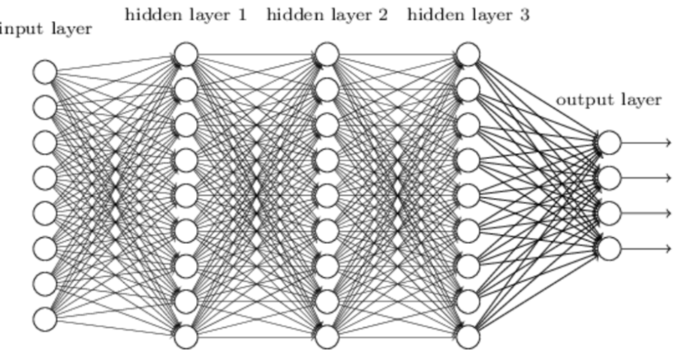


Figure 2.3: Deep Neural Network structure with 3 hidden layers

$$a^{(l)} = f^{(l)}(W^{(l)}a^{(l-1)} + b^{(l)}) = f^{(l)}(z^{(l)}) \tag{2.15}$$

with $f^{(l)}$ the activation function for layer l and $z^{(l)}$ is the weighted input vector to layer l.

Backpropagation and Training

The feedforward operation described above is the process by which input data is passed through the neural network to generate predictions. However, for a neural network to learn and improve its performance, it must undergo a training process. Backpropagation is a crucial algorithm used for training deep neural networks.

The primary goal of training is to minimize the difference between the predicted output and the actual target values. This difference is quantified by a loss function, denoted as $\mathcal{L}$, which measures the disparity between the predicted output $a^{(L)}$ and the true output, L being the last layer index. The training process involves adjusting the weights and biases in the network to minimize this loss.

The backpropagation algorithm consists of two main steps: forward pass and backward pass. First, a forward pass is realized to compute $a^{(l)}$ and $z^{(l)}$ for all layers l. In the backward pass, the gradient of the loss function with respect to the weights and biases is computed. This gradient represents the sensitivity of the loss to changes in the weights and biases. The chain rule of calculus is employed to calculate these gradients layer by layer, starting from the output layer and moving backward through the network.

Let $\delta^{(l)}$ represent the error term for layer l. The error term is computed recursively and used to compute the gradients:

$$\delta^{(l-1)} = \frac{\partial \mathcal{L}}{\partial z^{(l-1)}} = \left[W^{(l)}\right]^T \cdot \delta^{(l)} * f'^{(l-1)}(z^{(l-1)}) \tag{2.16}$$

$$\frac{\partial \mathcal{L}}{\partial W^{(l)}} = \delta^{(l)} \cdot \left[a^{(l-1)}\right]^T \tag{2.17}$$

$$\frac{\partial \mathcal{L}}{\partial b^{(l)}} = \delta^{(l)} \tag{2.18}$$

where $f'^{(l)}$ is the derivative of the activation function.

This backward pass let the weight and bias updates according to the gradient descent algorithm:

$$W^{(l)} \leftarrow W^{(l)} - \alpha \frac{\partial L}{\partial W^{(l)}} \tag{2.19}$$

$$b^{(l)} \leftarrow b^{(l)} - \alpha \frac{\partial L}{\partial b^{(l)}} \tag{2.20}$$

where α is the learning rate, controlling the size of the steps taken during optimization.

This iterative process of forward pass (feedforward operation) and backward pass (backpropagation) continues until the neural network reaches a state where the loss is minimized, and it has learned to make accurate predictions on the training data.

2.2.3 Convolutional Networks

Convolutional Neural Networks (CNNs) are a class of deep neural networks designed for processing structured grid data, such as images. It was partly introduced by [14] in the context of document recognition. CNNs have proven to be very effective in tasks related to computer vision, including image classification, object detection, and segmentation. In the context of video game AI, CNNs can be employed for various purposes, such as recognizing objects, detecting obstacles, or even generating realistic game environments.

Convolutional Layer

The fundamental building block of CNNs is the convolutional layer. In this layer, small filters (also known as kernels) slide or convolve across the input data to produce feature maps. Each filter captures local patterns, allowing the network to recognize hierarchical features.

The convolution operation is represented by the equation:

$$(I * K)(i, j) = \sum_m \sum_n I(m, n) \cdot K(i - m, j - n) \tag{2.21}$$

where I is the input image of size $m \times n$ and K is the filter. Here we consider the input and output channels c_{in}, c_{out} to be one. An illustration of convolution is depicted on figure 2.4. In general case, we use c_{out} sets of kernel composed each of c_{in} kernel, one for each input channel.

The results of each kernel in a single set computed with the formula above are then sum up. We can choose the size of kernels, choose the stride between kernel convolutions and add pooling to the image for size convenience. This convolution is then followed by an activation function, for instance ReLU.

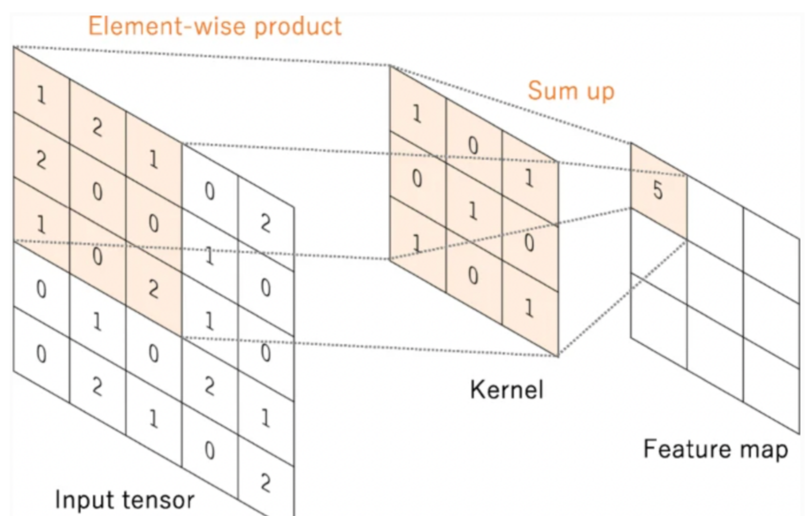


Figure 2.4: Kernel convolution example for a single input channel

Pooling Layer

Pooling layers are often inserted after convolutional layers to reduce the spatial dimensions of the feature maps while retaining important information. Max pooling is a common pooling operation where the maximum value in a local region is retained, helping to capture the most salient features. An example illustrated on figure 2.5.

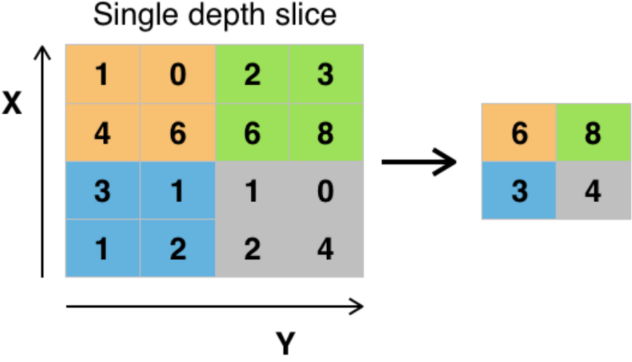


Figure 2.5: 2×2 max pooling on a 8×8 image

Fully Connected Layer

After several convolutional and pooling (subsampling) layers, CNNs typically include one or more fully connected layers to perform high-level reasoning on the learned features. These layers connect

every neuron to every neuron in the previous and subsequent layers, see the above section on deep neural network. A classical CNN structure is illustrated on figure 2.6.

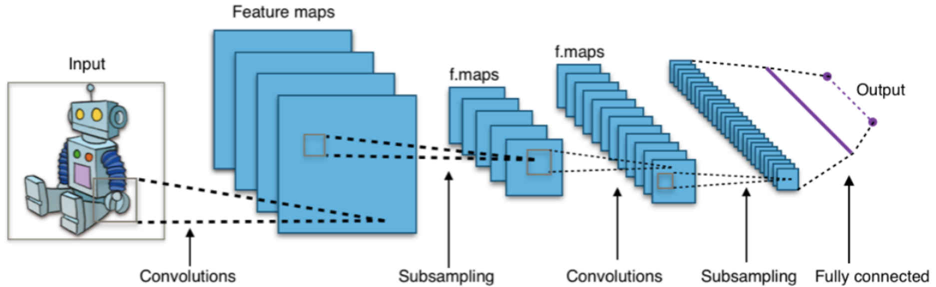


Figure 2.6: Classical CNN structure with two convolutional layers

Training CNNs

The training of CNNs involves a similar process to that of fully connected networks but with the added complexity of handling three-dimensional input data. Backpropagation combined with stochastic gradient descent are also used to update the weights of the network, as in classical deep neural networks.

Chapter 3

Methodology

3.1 Geometry Dash Environment

Due to the limited accessibility of game variables in the official version of *Geometry Dash* for testing Reinforcement Learning (RL) algorithms, we developed a custom version using Python and the Pygame library, inspired by the modifications found in [15]. In our adaptation, we omitted elements such as the rocket ship and eliminated the rotation of the player's character during jumps to simplify the visuals. Despite these simplifications, the essential complexity of the game is preserved, making it an appropriate testbed for evaluating RL algorithms.

3.2 Game Model

Geometry Dash is a popular 2D platformer game known for its challenging levels and rhythm-based gameplay [16]. We selected this game because it is fundamentally deterministic, yet it can be customized from easy to extremely complex levels. Additionally, we have the capability to introduce disturbances and new game mechanics, as we have implemented the game in Python. Therefore, this setup is ideal for all our experiments in the context of game variability. This section details the fundamental game mechanics and rules that we consider by default in this thesis.

Objective:

The primary goal in *Geometry Dash* is to navigate a square-shaped character, often called the "cube" or "icon," through levels laden with obstacles while collecting objects. The player's objective is to reach the end of each level without crashing into any obstacles. The various game objects are depicted in Figure 3.1.

Action Space:

The cube moves automatically from left to right at a constant speed. The player does not control the speed but can direct the cube to jump. The action space is binary, allowing the player at each

Figure 3.1: *Geometry Dash* object description

frame to choose either to jump or not to jump. Jumping is prohibited if the cube is already in the air.

Obstacles:

Levels are strewn with various obstacles, such as spikes and platforms, which must be avoided to avoid failure. Contact with a spike or the side of a platform results in instant failure, necessitating precise jump timing to avoid these obstacles. Figure 3.2 shows a typical scenario where the player must quickly navigate two platforms and then leap over spikes.

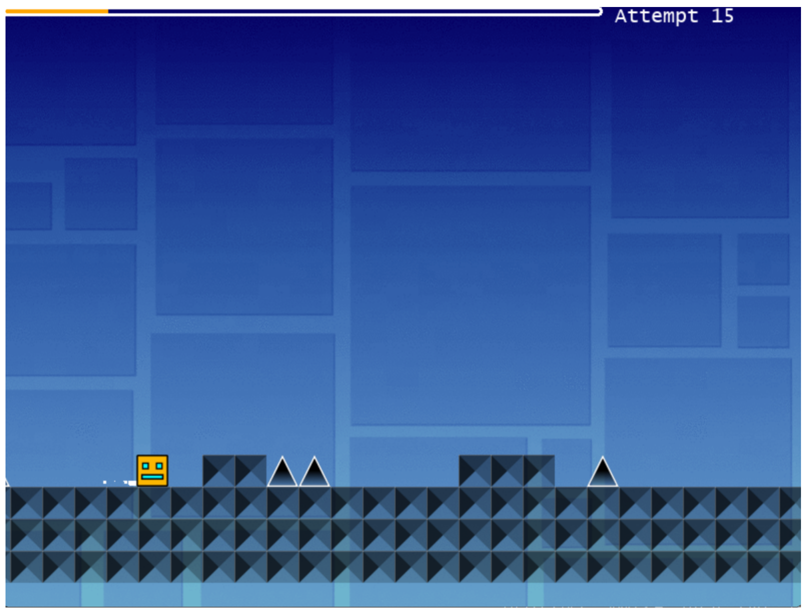


Figure 3.2: A challenging scenario involving double platforms followed by spikes.

Coins:

Coins are strategically placed to challenge the player further, often near obstacles to increase the risk. Collecting these coins contributes to achieving higher scores. In Figure 3.3, a coin is strategically located requiring the player to navigate a lower path to collect it, adding to the game's strategic complexity.

Figure 3.3: Strategically placed coin near obstacles to increase gameplay challenge.

Special Blocks (Orbs):

Orbs within the game act as special blocks that enable a "double jump" when contacted, adding a layer of strategic depth and necessitating precise timing for successful navigation through complex sections. Figure 3.3 illustrates how interacting with an orb can be pivotal for surpassing obstacles and accessing various level parts.

Level Editor and Customization:

Our Python-based platform includes an enhanced level editor that allows for the creation of levels ranging from very easy to nearly impossible. This feature enables us to systematically adjust the level difficulty and evaluate its impact on learning complexity.

Overall, *Geometry Dash* challenges players to maneuver a cube through intricately designed obstacles, collect items, and complete levels. The introduction of special blocks like orbs and the inclusion of a level editor enrich the gameplay, adding variety and depth to the player's experience.

3.3 Baseline AI design

In this subsection, we explain the chosen methodology for designing a *Geometry Dash* AI based on reinforcement learning. Regarding the reinforcement learning algorithms described in subsection 2.1, we decided to apply deep Q-learning for the following reasons

- Deep Q-learning is model free, we can design any deep neural network we like to learn the game.
- Deep Q-learning can learn high dimensional state spaces, which is typical for video games. We could for instance give as input the game image directly if desired.
- Deep Q-learning does not require to know the environment dynamics like the reward or transition probability functions[1].

Deep Q-Learning has proven to be remarkably successful in mastering a variety of video games, as exemplified by its notable application to Atari games [4, 17].

3.3.1 Deep Q-Learning Algorithm

As outlined earlier, our training approach employs a deep Q-learning algorithm. During gameplay, the agent undergoes a sequence of actions, transitioning from state s_t to s_{t+1} with an associated reward r_t at each time step t. The states are pre-processed using the function ϕ, resulting in $\phi(s_t) = \phi_t$ and $\phi(s_{t+1}) = \phi_{t+1}$.

To facilitate learning, these experiences are systematically stored in the player's memory, denoted as $\mathcal{D} = \{e_1, e_2, ..., e_k\}$, accumulated over multiple game episodes into a replay memory. A key innovation in deep Q-learning, known as experience replay[18], is employed. After each step, determined by an ϵ-greedy algorithm, the experience is archived in memory. Periodically, the model is trained on a mini-batch sampled uniformly from the memory $\mathcal{D}$. This strategic utilization of experience replay prevents the agent from fixating on recent experiences and aids in preserving a diverse set of encounters.

The ϵ-greedy strategy is integrated to balance exploration and exploitation. A diminishing function ϵ during training dictates the likelihood of selecting a random move. The epsilon-decaying function for the epsilon-greedy algorithm is given by:

$$\epsilon = \epsilon_\infty + (\epsilon_0 - \epsilon_\infty) \cdot \exp\left(-\frac{n}{\epsilon_\text{d}}\right) \tag{3.1}$$

where n is the number of actions performed so far in the training. This let a smooth transition between exploration and on-policy decision making. This randomness contributes to exploring new configurations, crucial for a comprehensive learning experience.

[1] In such a deterministic game, no transition probability function is needed. Still, not knowing the reward function is problematic for algorithms like value or policy iteration

This approach bears advantages; firstly, it leverages each experience in multiple weight updates, enhancing data efficiency. Additionally, by randomizing the sampling process, correlations between consecutive samples are broken, mitigating variance in updates. Thirdly, in on-policy learning, a greedy exploration approach is crucial to navigate unforeseen moves, preventing the agent from becoming stuck in local minima.

To facilitate smooth weight updates in our Deep Q-Learning algorithm, we employ two neural networks: a policy network, denoted as Q_{pol}, and a target network, denoted as Q_{tar}. During training, using gradient descent with learning rate α, the updates are applied to the policy network. Subsequently, we update the target weights, denoted as θ_{tar}, using a weighted average of the policy weights, θ_{pol}.

The update of the target weights is performed using the formula:

$$\theta_{\text{tar}} = \tau\theta_{\text{pol}} + (1 - \tau)\theta_{\text{tar}} \tag{3.2}$$

where τ is a hyperparameter controlling the degree of update. This mechanism, known as target network soft update, helps stabilize the training process by providing a more consistent and slowly evolving target for the Q-values. This approach mitigates the risk of rapid and potentially destabilizing changes in the target network, contributing to more reliable and effective training in Deep Q-Learning.

The algorithm pseudo-code is illustrated below.

Algorithm 1: Deep Q-learning with Experience Replay

Data: Initialize replay memory $\mathcal{D}$ to capacity N
Data: Initialize target and policy network weights $\theta_{\text{tar}} = \theta_{\text{pol}}$ randomly
for *episode = 1 to* M **do**
  Initialize state s_1 and preprocessed state $\phi_1 = \phi(s_1)$;
  for t *= 1 to* T **do**
    With probability ϵ, select a random action a_t;
    Otherwise, select $a_t = \arg\max_a Q(\phi(s_t), a; \theta_{\text{pol}})$;
    Execute action a_t in emulator and observe reward r_t and state s_{t+1};
    Preprocess $\phi_{t+1} = \phi(s_{t+1})$;
    Store experience $e_t = (\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$;
    Sample random minibatch of experiences $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$;
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_a Q(\phi_{j+1}, a; \theta_{\text{tar}}) & \text{for non-terminal } \phi_{j+1} \end{cases}$;
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta_{\text{pol}}))^2$;
    Update target weights $\theta_{\text{tar}} = \tau\theta_{\text{pol}} + (1 - \tau)\theta_{\text{tar}}$
  end
end

3.3.2 State Representation and Transition

As *Geometry Dash* unfolds continuously over time, we utilize a discrete representation by dividing gameplay into frames, each representing a distinct game state. In the standard version of *Geometry*

Dash, where the horizontal speed is constant and objects are stationary relative to the ground, simplifying the state representation might be feasible. Specifically, frames where the player is airborne and not interacting with orbs could potentially be omitted, as no actions are required in these segments, and the object positions upon landing are predictable due to the constant speed.

However, as mentioned in the introduction, we intend to introduce complexities such as randomly moving spikes in later stages of the game. These additions necessitate a detailed frame-by-frame analysis to accurately capture the dynamics of the game. Therefore, we will maintain a discrete representation where each frame corresponds to a unique game state, ensuring our model is robust enough to handle the increased complexity brought by new game elements.

3.3.3 Neural Network Architectures

In this section, we explore two distinct neural network architecture approaches for our application. The second approach employs a Convolutional Neural Network (CNN) architecture, similar to those proven effective in Atari game AIs. However, this approach offers limited transparency regarding which patterns the neural network recognizes within the image, making it difficult to interpret.

Alternatively, the first approach, which we consider innovative, utilizes sensor grids around the player to detect obstacles and objects around the player. While this sensor-based approach has no guaranteed efficacy a priori, its simplicity could lead to a more interpretable system that still yields respectable results.

Sensor Approach

Our exploratory model involves a straightforward deep neural network using sensors arranged around the player. Each sensor activates to 1 upon contact with a specific object type—spikes, platforms, coins, or orbs—and remains at 0 otherwise. The sensors are organized in a grid layout, with a 10 by 7 array around the player, each cell hosting sensors for all four object types.

This setup results in an input space of $4 \times 10 \times 7 = 280$ binary entries for a single frame. To adequately capture game dynamics, we stack these entries from four consecutive frames, similar to the approach used in Atari AI systems, resulting in an input vector of $4 \times 280 = 1120$ binary entries for each frame time. The preprocessing function ϕ corresponds to the transformation of the last four game frames into this binary vector representation.

The network comprises two dense layers designed to predict the jumping actions of the player. The detailed architecture is visually represented in Table 3.1, and an illustration of the spike sensor positions is shown in Figure 3.4. The complete architecture scheme is illustrated in Figure 3.5.

Table 3.1: Deep Q-Network architecture for Sensor approach

Layer	Output Shape	Param #	Activation
Input (4x4x10x7)	-	-	-
Linear (150)	150	168150	ReLU
Linear (150)	150	22650	ReLU
Linear (2)	2	302	-

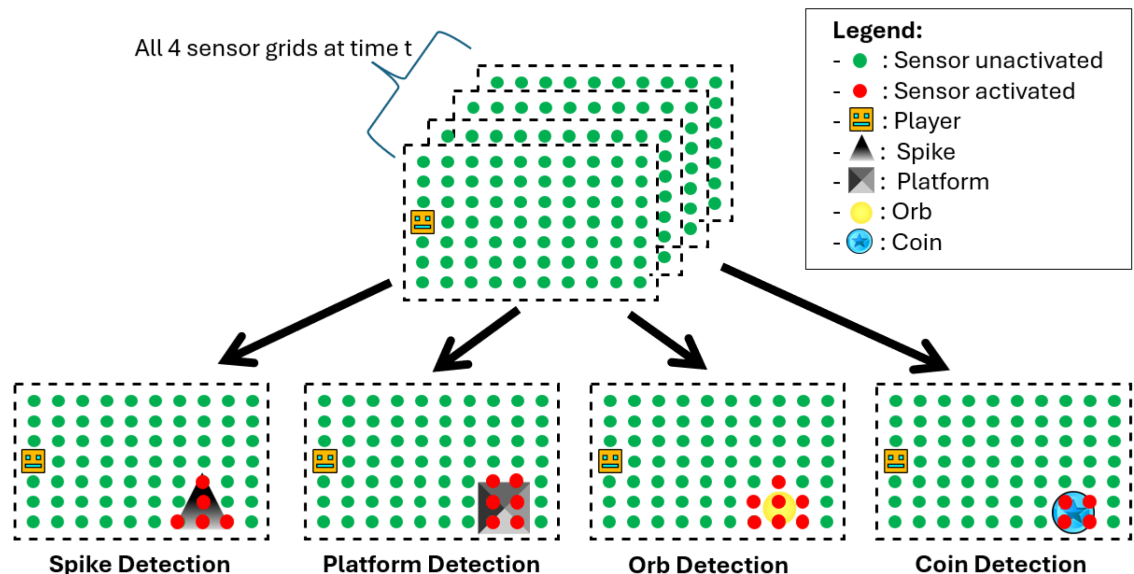


Figure 3.4: Visual explanation of stacked sensor grids for one single frame at time t

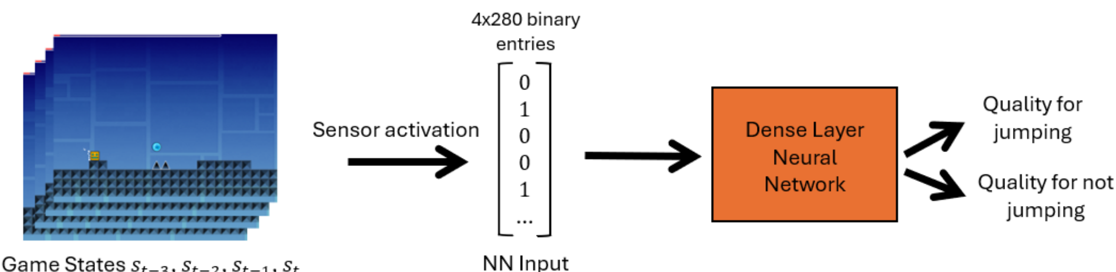


Figure 3.5: Scheme of Sensor architecture method

Convolutional Network Approach

This approach, inspired by the Atari Deep Q-Learning research, preprocesses each game frame by converting it to a downscaled grayscale image. We use a sequence of four consecutive frames at each time step t, namely $s_t, s_{t-1}, s_{t-2}, s_{t-3}$, as input to the network, facilitating the agent's understanding of the dynamic environment. Each frame, originally sized at 800x600, is first cropped to a 600x600 square by removing portions behind the player that do not influence the decision-making process. The images are then downsampled to 100x100 pixels and converted from RGB to grayscale, effectively reducing the color channels from three to one. The preprocessing function ϕ ultimately produces a stack of four 100x100 grayscale images, representing the current neural network input. An example of such a preprocessed image is shown in Figure 3.6.

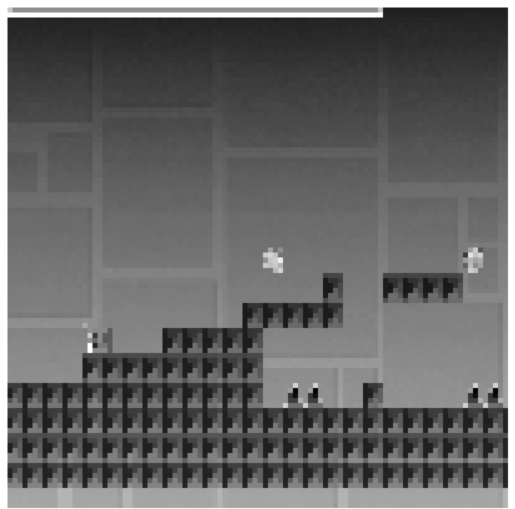


Figure 3.6: Example of one out of four preprocessed images given as input to neural network

The network architecture includes three convolutional layers designed to extract significant feature maps from the input images, following the architecture used in Atari game studies. These feature maps are subsequently fed into a single dense layer that computes the qualitative values necessary to decide whether to make the cube jump. The specifics of the CNN architecture employed are detailed in Table 3.2.

Table 3.2: DQN Neural Network Architecture

Layer	Output Shape	Param #	Activation
Input (4x100x100)	-	-	-
Conv2d (32, kernel=8, stride=4)	(32, 24, 24)	32080	ReLU
Conv2d (64, kernel=4, stride=2)	(64, 11, 11)	32832	ReLU
Conv2d (64, kernel=3, stride=1)	(64, 9, 9)	36928	ReLU
Flatten	5184	-	-
Linear (2)	2	15,552	-

The CNN Method Scheme is illustrated in figure 3.7.

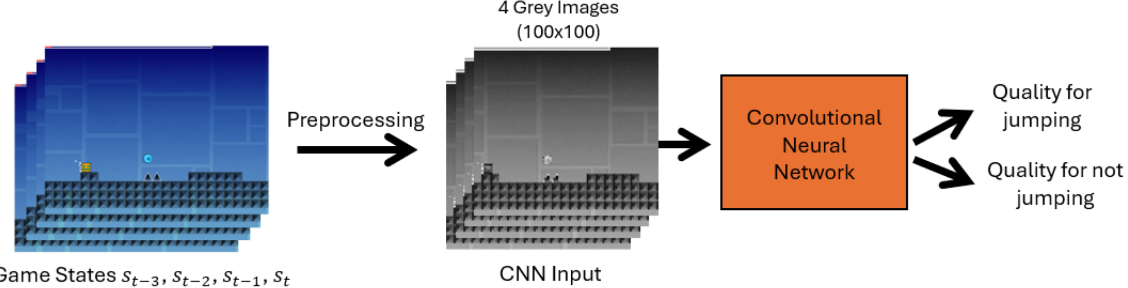


Figure 3.7: Scheme of CNN architecture method

3.3.4 Training Levels Baseline

The objective of this phase is to establish a baseline AI model using one or both of the discussed network architectures. This model should be capable of learning essential game mechanics across a variety of obstacle configurations. To facilitate this, we designed 12 training levels, each featuring diverse obstacles that require precise jump timing. The training regimen involves the AI agent selecting one of these 12 levels at random each time it either completes a level or fails. This approach ensures that the AI experiences a broad range of challenges, which is crucial for learning robust and adaptable strategies. Details and examples of these obstacles are provided in Appendix A to illustrate the varying levels of difficulty. The ultimate goal is for the AI to master all 12 levels, at which point it will be considered a baseline model suitable for further experimentation.

3.3.5 Reward Function Design and Hyperparameter Tuning

Initial Reward and Hyperparameter Choices

An effective reinforcement learning algorithm depends critically on a well-designed reward function $R(s_t, a_t, s_{t+1})$ to guide the agent's transitions from state s_t to s_{t+1} through action a_t. Given that our power resources are limited, we will calibrate the reward function to ensure it is explicit enough for the agent to master the game within a reasonable timeframe. Moreover, algorithm hyperparameters need careful tuning to achieve optimal learning outcomes. The initial reward function and its rationale are as follows:

- $R(s_t, a_t, s_{t+1}) = -1$ when the player dies after state s_t, discouraging failure.
- $R(s_t, a_t, s_{t+1}) = 1$ when the player successfully completes a level, encouraging level completion.
- $R(s_t, a_t, s_{t+1}) = 1$ when the player collects a coin, incentivizing score improvement.
- $R(s_t, a_t, s_{t+1}) = 1$ when the player successfully jumps over a spike, rewarding obstacle avoidance.

Hyperparameter values were initially set based on guidelines from the PyTorch documentation, insights from Atari AI research, and personal experimentation. The selected values are listed in Table 3.3.

Algorithm Parameter	Value
Learning Rate (α)	1×10^{-4}
Discount Factor (γ)	0.95
Soft Update Parameter (τ)	0.01
Number of Games (N)	3000
Batch Size	400
Memory Size	100000
Epsilon Start (ϵ_0)	1.0
Epsilon End (ϵ_∞)	0.01
Epsilon Decay (ϵ_d)	1000

Table 3.3: Initial algorithm parameter values for training the deep Q-networks

The effectiveness of this reward structure and hyperparameter setting is analyzed by observing the training performance, as shown in Figure 3.8. We averaged the episode duration (in % of maximum duration) over 20 runs to identify general learning trends. Initially, both network architectures exhibit increased learning during the exploration phase but plateau subsequently, suggesting potential areas for adjustment.

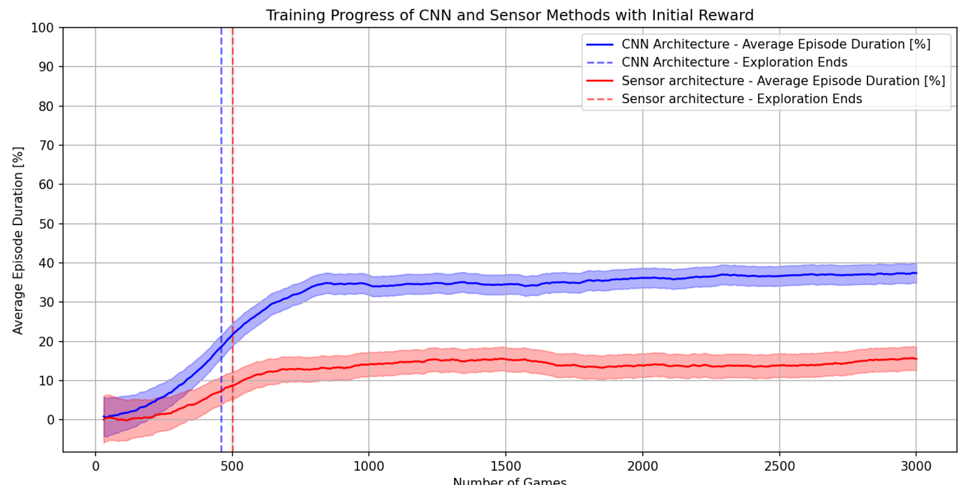


Figure 3.8: Evolution of average episode duration for Sensor and CNN methods with initial algorithm parameters and reward function, showing 95% confidence intervals for both averages based on 20 runs.

A notable issue is the AI's excessive jumping, which often positions it poorly for successfully navigating upcoming obstacles. This behavior appears to stall further learning progress, indicating a need for further tuning or adjustment in the reward function or game mechanics.

Fixing the Jumping Agent Issue

Previously identified issues were addressed by adjusting the reward function to discourage unnecessary jumping. Specifically, the AI was rewarded for staying grounded when it touched the floor. This incentive was designed to be less than the reward for successfully navigating over a spike, to avoid a conflict between staying grounded and jumping to evade obstacles. We experimented with three different "No-Jump" reward values: 0.1, 0.5, and 0.9. The outcomes, depicted in Figures 3.9 and 3.10 for Sensor and CNN methods respectively, showed improvements under all settings.

Despite the varying reward values, both methods benefited from the new reward structure, though the CNN method consistently outperformed the Sensor method, which did not appear to converge within a 3000-game timeframe. The reward value of 0.5 resulted in better convergence for both methods; therefore, this value was selected for ongoing experiments.

Figures 3.11 and 3.12 demonstrate a significant reduction in jump frequency, correlating with enhanced performance when a 0.5 reward for staying on the ground was used. However, neither method has yet mastered all 12 training levels. Further analysis involved examining the average episode duration after 3000 games for each training level, as shown in Figures 3.13 and 3.14. Notably, the CNN method achieved a 100% average episode duration on five levels, none of which contained orbs—special blocks that enable a double jump. This suggests that the AI's failure in other levels predominantly occurs because it does not utilize orbs essential for clearing lethal obstacles.

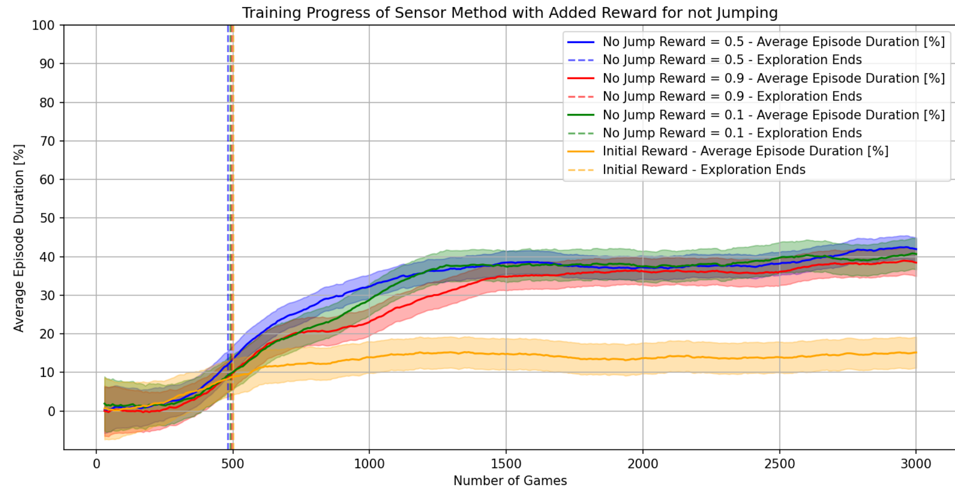


Figure 3.9: Evolution of the average episode duration for the Sensor method, showing initial parameters and the effect of different "No Jump" reward values. Includes 95% confidence intervals based on 20 runs.

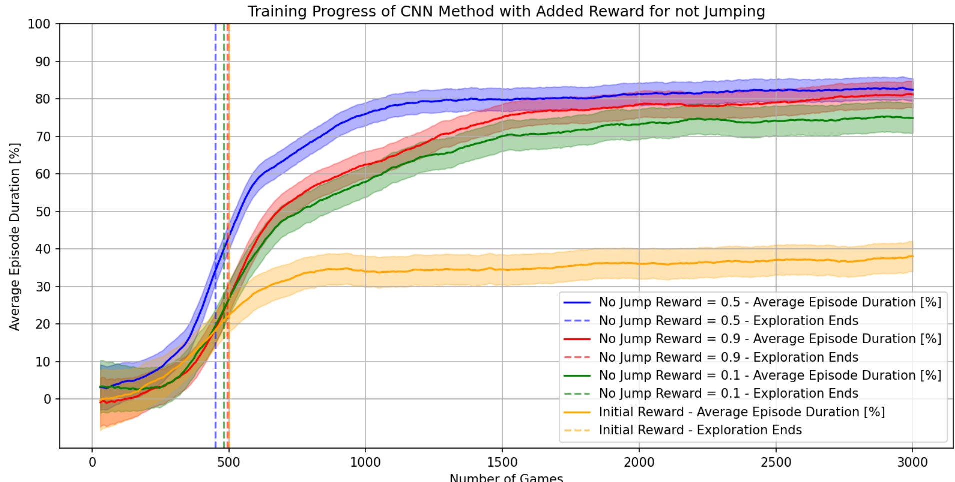


Figure 3.10: Evolution of the average episode duration for the CNN method, showing initial parameters and the effect of different "No Jump" reward values. Includes 95% confidence intervals based on 20 runs.

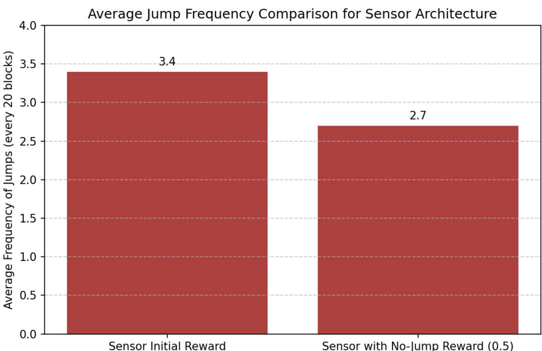


Figure 3.11: Comparison of jump frequency reduction for the Sensor method over 3000 games, with and without a "No-Jump" reward of 0.5.

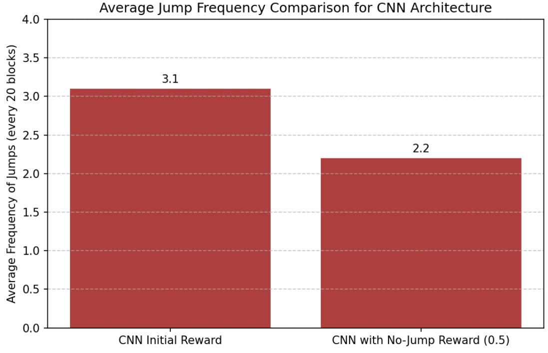


Figure 3.12: Comparison of jump frequency reduction for the CNN method over 3000 games, with and without a "No-Jump" reward of 0.5.

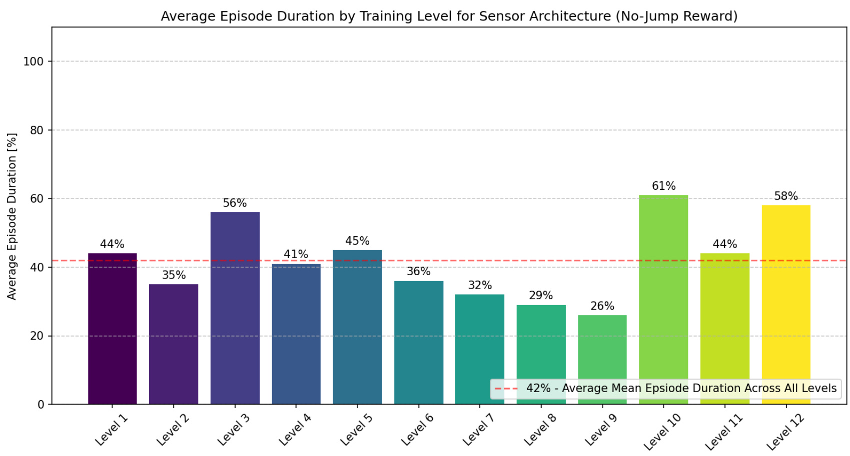


Figure 3.13: Distribution of average episode duration across 12 training levels using the Sensor method, using a "No-Jump" reward of 0.5.

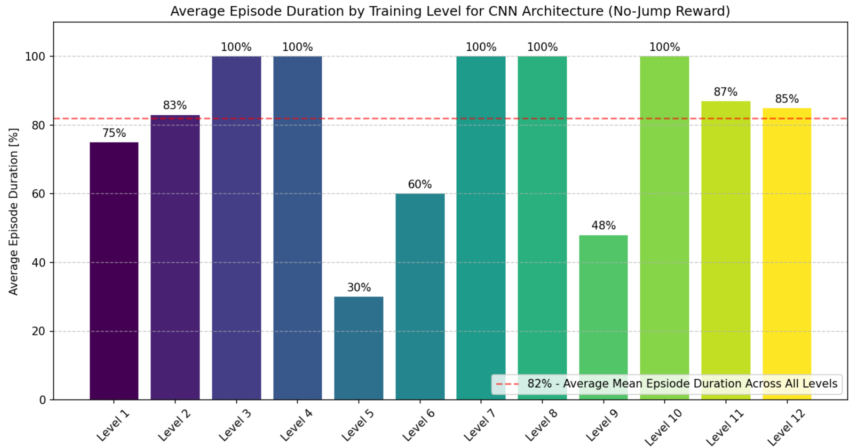


Figure 3.14: Distribution of average episode duration across 12 training levels using the CNN method, using a "No-Jump" reward of 0.5.

Fixing the Orb Issue

To promote the utilization of orbs, which facilitate avoiding obstacles, we introduced an additional reward of 1 for each orb use. This new incentive aims to reinforce the agent's understanding that orbs are beneficial. The impact of this updated reward system on training is depicted in Figure 3.15, which shows the training curves for both the Sensor and CNN methods compared with the initial reward setup. Notably, the CNN method successfully mastered all 12 training levels in fewer than 2400 games with the new reward function.

Conversely, the Sensor method struggled to achieve similar mastery within the same number of games. This difficulty may stem from the limitations of the binary input vector, which may not provide a sufficiently detailed representation of the game state to effectively train the agent.

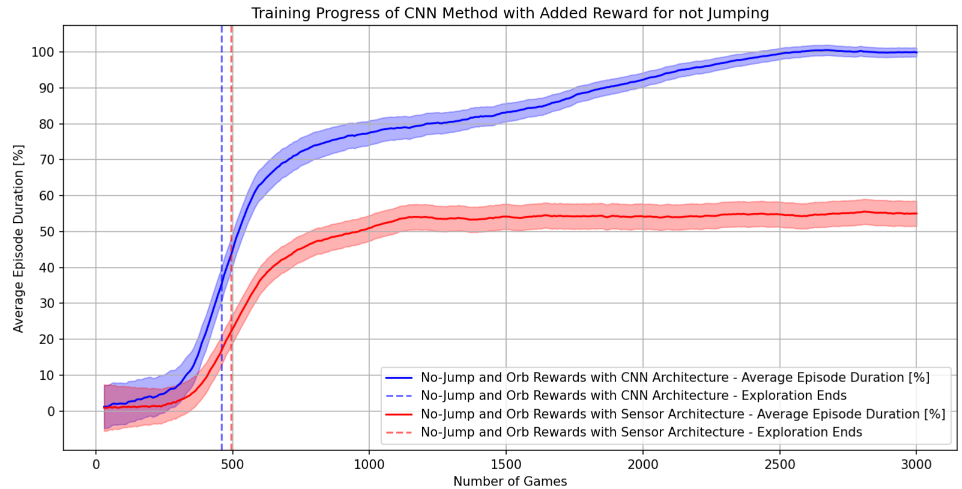


Figure 3.15: Training progress for the Sensor and CNN methods showing the evolution of average episode duration with updated "No-Jump" and "Orb" rewards. Includes 95% confidence intervals for both methods based on 20 runs.

Investigating the Sensor Method Issues

The Sensor method appears to underperform compared to the CNN method within a 3000-game evaluation period. We hypothesize two main reasons for this: the network might lack sufficient expressivity due to inadequate input handling or a too simplistic network architecture, or the network may be too large, leading to unstable training, in which case we should not pick this method for further experiments. To address the first concerns, we experimented with adding more layers (3 or 4 layers) and separately increasing the width of the original layers. The outcomes are shown in Figures 3.16 and 3.17.

Despite efforts to enhance the network by making it wider and deeper, the learning performance did not improve, suggesting that the method may be either too complex to train effectively or inherently flawed due to sensor activations that fail to capture critical game information. Consequently, we have decided to discontinue this approach and will focus exclusively on the CNN method for the remainder of the study.

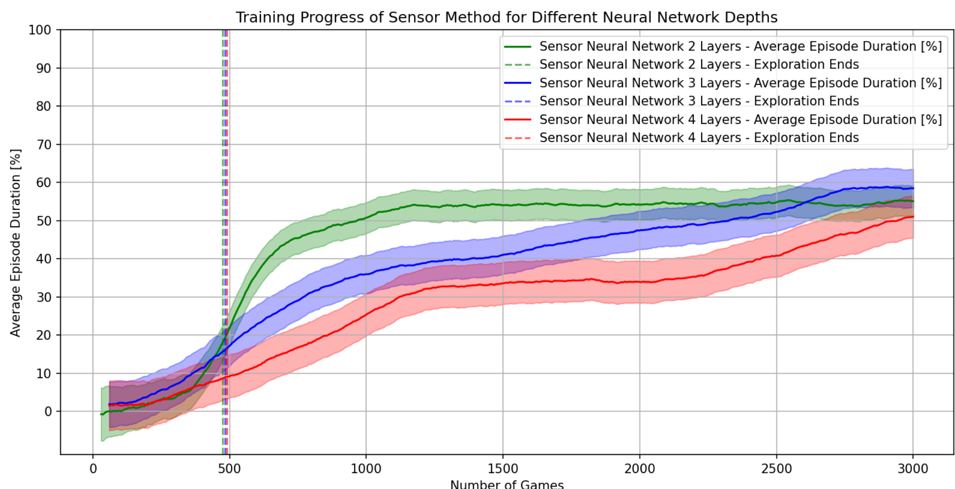


Figure 3.16: Analysis of average episode duration for the Sensor method with three network depth configurations, each layer containing 150 nodes. Includes 95% confidence intervals based on 20 runs.

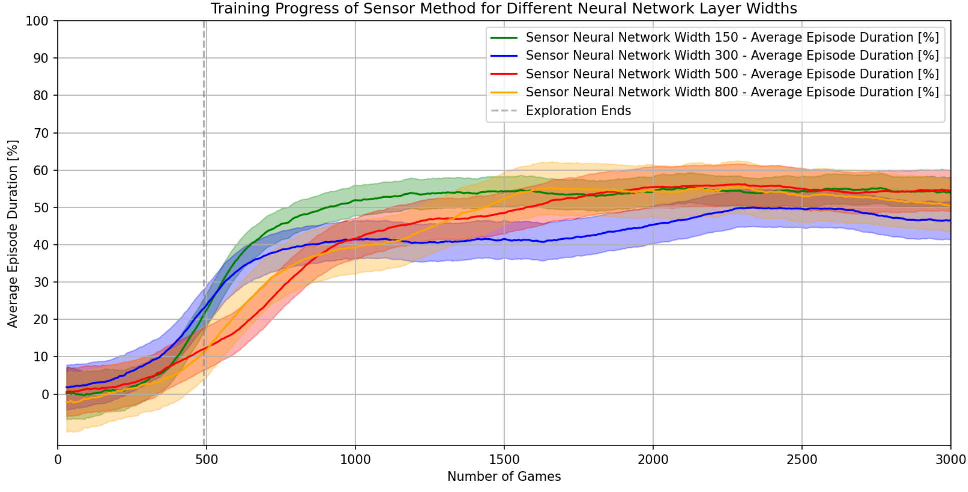


Figure 3.17: Evaluation of average episode duration changes for the Sensor method by varying the width of layers to 150 nodes. Includes 95% confidence intervals based on 20 runs.

Final Reward Function and Algorithm Hyperparameters

Having established that the CNN method is the preferred architecture, we have finalized the reward function as follows:

- $R(s_t, a_t, s_{t+1}) = -1$ for player failure after state s_t.
- $R(s_t, a_t, s_{t+1}) = 1$ for successfully completing the level from state s_t.
- $R(s_t, a_t, s_{t+1}) = 0.5$ for abstaining from jumping while grounded.
- $R(s_t, a_t, s_{t+1}) = 1$ for collecting a coin.
- $R(s_t, a_t, s_{t+1}) = 1$ for executing a jump using an orb.
- $R(s_t, a_t, s_{t+1}) = 1$ for successfully navigating over a spike.

This revised reward function introduces additional rewards for abstaining from unnecessary jumps and using orbs effectively. We maintain the initial algorithm hyperparameters, which are detailed in Table 3.4.

Algorithm Parameter	Value
Learning Rate (α)	1×10^{-4}
Discount Factor (γ)	0.95
Soft Update Parameter (τ)	0.01
Number of Games (N)	3000
Batch Size	400
Memory Size	100000
Epsilon Start (ϵ_0)	1.0
Epsilon End (ϵ_∞)	0.00
Epsilon Decay (ϵ_{d})	1000

Table 3.4: Default algorithm parameters for training the deep CNN Q-network

3.4 Computational Resources

Reinforcement learning simulations, especially those involving large deep neural networks, can be highly computationally intensive. A single training run on a local CPU may take over six hours, making the use of GPUs essential to enhance efficiency and manage the computational load. Without GPU acceleration, generating plots that require 20 runs for computing confidence intervals would take more than five days on a standard laptop.

To address these challenges, we utilize the CECI servers, which allow for substantial computational savings. A comparison of the accumulated training time on local CPU versus CECI's GPU setups, shown in Figure 3.18, indicates that using a GPU yields a fivefold reduction in processing time with such experimental set up, even without parallelization of the runs. Based on these results, we will use CECI clusters for all computations in this study.

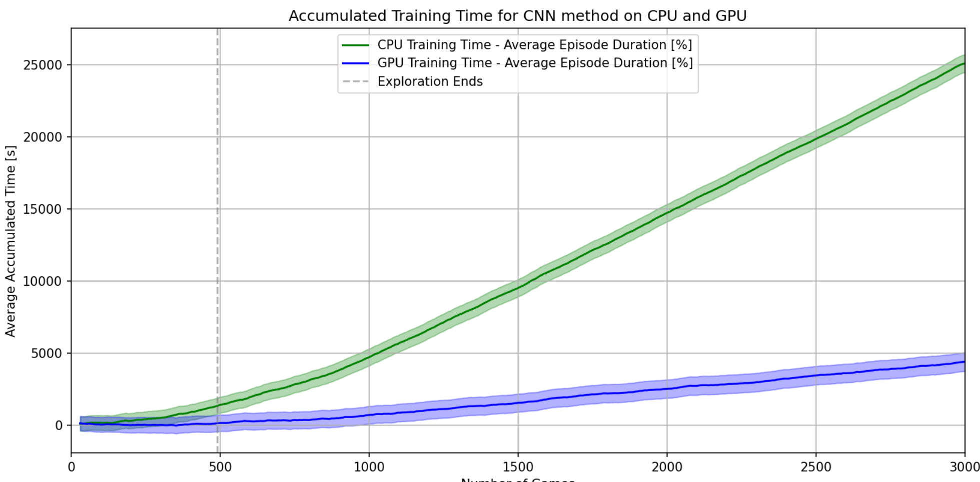


Figure 3.18: Comparison of accumulated time based on 20 runs between CPU and GPU on CECI clusters, highlighting the significant time savings achieved with GPU use. Presence of 95% confidence intervals for both curves.

3.5 Metrics for Complexity

To effectively characterize and compare the learning complexity in this thesis, it is essential to define specific metrics. These metrics are tailored to particular settings and are calculated based on the outcomes of 20 runs to ensure robustness in case an average measurement is required.

- **Average Episode Duration [%]**: This metric represents the percentage of the maximum possible episode duration. It is computed as an average over 20 runs and is continuously tracked throughout the learning process.
- **Average Number** $N(\alpha\%)$: This denotes the average number of training games required for the agent to achieve an average episode duration of at least $\alpha\%$ of the maximum episode duration. For instance, $N(100\%)$ indicates the average number of games needed to master the game fully.
- **Critical Epsilon Decay** $\epsilon_{d,crit}$: This marks the exploration ϵ_d threshold value below which the learning progress stagnates, preventing the achievement of 100% episode duration in at least one of the 20 runs.
- **Average Coin Ratio**: Measures the average number of coins collected per episode. Considering that each level contains three coins, this ratio varies between 0 and 3, providing insight into the agent's efficiency at collecting these game-specific rewards.

These metrics collectively furnish a detailed perspective on the learning complexity and performance dynamics throughout the training process.

3.6 Experimental Objectives and Tracks

With the baseline AI model now capable of mastering sophisticated levels featuring diverse obstacles and precise jump timings, we turn to the primary objectives of this thesis by exploring the following experimental tracks:

- **Random Disturbances:** We will characterize the impact of random disturbances on learning complexity. This includes investigating both static disturbances that remain constant throughout an episode and dynamic disturbances that evolve during gameplay.
- **Action Space Cardinality:** This track explores how increasing the number of available action choices in *Geometry Dash* affects learning complexity.
- **Game Object and Obstacle Configurations:** We will analyze how adding new game objects or altering obstacle configurations influences the learning performance of the AI system.

The *Geometry Dash* environment provides several game parameters for manipulation during experiments, detailed as follows:

- **Gravity:** Adjusting gravity affects how quickly the player character falls post-jump. Varying this parameter requires the AI to adapt jump timings to navigate obstacles effectively.
- **Speed:** By modifying the horizontal speed of the player character, we alter the available reaction time for decision-making, challenging the AI to adapt its strategy under different speeds.
- **Jump Amount:** Changing the vertical speed boost obtained from jumps allows us to test the AI's ability to manage short versus long jumps, crucial for overcoming obstacles and reaching platforms.
- **Object Positions:** The strategic placement and potential random movement of objects add complexity to the levels. This variability enables us to systematically increase the difficulty, testing the robustness of the AI's learning capabilities.

These experiments are designed to thoroughly assess the adaptability and scalability of AI learning strategies within a controlled yet increasingly complex gaming environment.

Chapter 4

Results

In this chapter, we present the experiments and their outcomes. Initially, we introduce preliminary definitions and computations that are referenced throughout the chapter. The chapter is then organized into four main parts.

The first part evaluates the agent's performance on easy deterministic levels composed solely of spike obstacles placed at regular intervals. We examine how the agent masters these levels and identify the limits of its learning capabilities. Additionally, we present some counter-intuitive results that demonstrate the challenges of theorizing learning complexity, even in such simple scenarios.

The second part revisits the same deterministic levels, introducing random disturbances to game parameters such as jump heights, obstacle distances, gravity, and the speeds of the player and obstacles. Our objective is to assess the impact of these disturbances on the learning process as their amplitudes or variances increase. We also explore the agent's learning behaviors under conditions where disturbances are large enough to potentially render the game unplayable.

In the third part, we return to deterministic levels with only spikes, where we expand the action space by offering multiple possible jump heights. We investigate how learning complexity metrics evolve with the increased cardinality of the action space and highlight the crucial role of the exploration phase. We also examine how an increase in action space cardinality, combined with random disturbances, affects the learning process.

Lastly, we revisit the deterministic levels from the first part, this time introducing more advanced obstacles to explore how they impact the learning process, focusing on both the number and types of obstacles introduced. We also analyze the effects of the order in which these obstacles are introduced on learning complexity. Finally, we examine the implications for learning complexity when adding coins to the game (extra quests) that are not essential for survival.

4.1 Preliminary Definitions and Computations

Recall that in *Geometry Dash*, the player moves at a constant horizontal speed and must overcome gravity by jumping at precise moments to avoid obstacles. We define the following parameters for the game:

- $g = 0.86$: The default gravity in the game.
- $J = 10$: The jump amount, representing the vertical speed boost given by jumping.
- $h_s = 6$: The constant horizontal speed of the player.

The ballistic trajectory of the player after a jump at time t_0 from position (x_0, y_0) is described by the equations:

$$x(t) = x_0 + h_s(t - t_0), \tag{4.1}$$

$$y(t) = y_0 + Jt - \frac{g(t - t_0)^2}{2}. \tag{4.2}$$

To calculate the jump distance, we solve for t when $y(t) = y_0$, obtaining:

$$0 = Jt - \frac{g(t - t_0)^2}{2}, \tag{4.3}$$

$$\Rightarrow t - t_0 = \frac{2J}{g}. \tag{4.4}$$

Using the fact that the $x-$movement is uniform, this results in a jump distance of $\Delta = \frac{2h_sJ}{g} \approx 140$. Additionally, we define the width and height of a spike as $l_s = 32$. The critical distances from the center of a spike, below and above which a jump is impossible, d_{crit} and D_{crit}, are calculated by solving:

$$l_s = Jt - \frac{g(t - t_0)^2}{2}, \tag{4.5}$$

$$\Rightarrow t - t_0 = \frac{J \pm \sqrt{J^2 - 2l_sg}}{g}. \tag{4.6}$$

This yields $d_{\text{crit}} = \frac{h_s}{g}(J - \sqrt{J^2 - 2l_sg}) \approx 23$ and $D_{\text{crit}} = \frac{h_s}{g}(J + \sqrt{J^2 - 2l_sg}) \approx 116$. Note the default distance d between two obstacles is set to $\bar{d} = 160$.

A scheme of spike distance d, spike height/width l_s, and critical distances d_{crit} and D_{crit} is depicted on figure 4.1. As the player is not punctual, we will do all the calculations with respect to the bottom-right player corner as it will systematically be the collision point with any obstacle. This point is represented by point A on the figure.

All game parameters are summarized in Table 4.1 and will be referenced in subsequent sections.

Figure 4.1: Scheme representing an example of spike distance d, spike height/width l_s, and critical jump distances d_{crit} and D_{crit}.

Game Parameter	Symbol	Default Value
Gravity	g	$\bar{g} = 0.86$
Jump amount	J	$\bar{J} = 10$
Player's horizontal speed	h_s	$\bar{h}_s = 6$
Spike height	l_s	$\bar{l}_s = 32$
Jump distance	Δ	$\bar{\Delta} = 140$
Short critical jumping distance	d_{crit}	$\bar{d}_{crit} = 23$
Long critical jumping distance	D_{crit}	$\bar{D}_{crit} = 116$
Obstacle distance	d	$\bar{d} = 160$

Table 4.1: Game default parameter values

4.2 Deterministic Levels with Only Spike Obstacles Regularly Spaced

In this section, we consider simple deterministic levels composed solely of spikes, set at a constant distance $d = \bar{d} = 160$. A game setting illustration is represented on figure 4.2. The objective of this section is to investigate the following questions.

- How fast does the agent learn on such simple levels ?
- What are the limits of learning process on those simple levels ?
- Are the RL complexity metrics easy to predict even for such simple level configuration ?

In this configuration, the level is feasible given $d \geq \bar{\Delta} = 140$. If the jump distance Δ was larger than the spike distance d, the level would be impossible for sufficiently many spike obstacles.

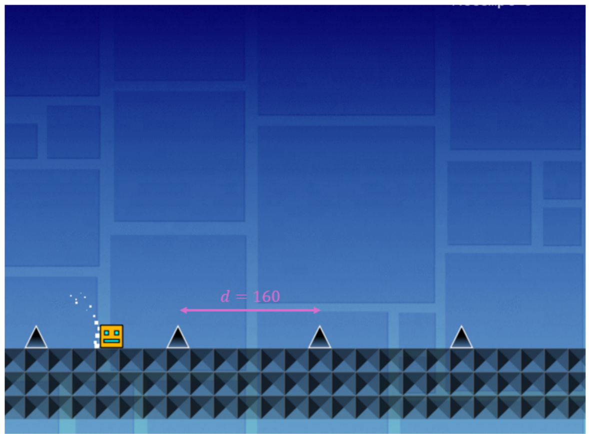


Figure 4.2: Illustration of deterministic level with only spike obstacles with constant distance $d = 160$.

The learning complexity in this scenario is significantly influenced by the exploration phase. We analyzed the learning process across various ϵ_d values, with the learning curves shown in Figure 4.3. For ϵ_d values of 150, 200, and 250, mastery of the game occurs shortly after the exploration phase concludes. Prior to this, random moves are selected for exploratory purposes. Conversely, for $\epsilon_d = 100$, learning stagnates at approximately 5% of average episode duration, indicating insufficient exploration. This shows that insufficient exploration clearly limits the learning process, even on simple levels. However, if this exploration is enough, the agent will master the game shortly after the exploration phase is over.

We then investigate the critical exploration rate values. A grid search identifies the critical decay rate $\epsilon_{d,crit} = 120$, with an absolute error of 2, as the threshold below which learning may not progress to 100% episode duration due to inadequate exploration. Figure 4.4 explores how variations in spike distance d influence $\epsilon_{d,crit}$. Contrary to expectations, the relationship is not strictly increasing; a minor decrease is observed for smaller distances near the jump distance $\Delta = 140$.

This decrease suggests that more precise jump timings are required for small d values, potentially increasing the game's difficulty and the need for exploration. However, as d increases, making obstacles less frequent, the trend reverses and the curve rises, indicating that successful generalization of jump strategies over spikes becomes more challenging, requiring more exploration.

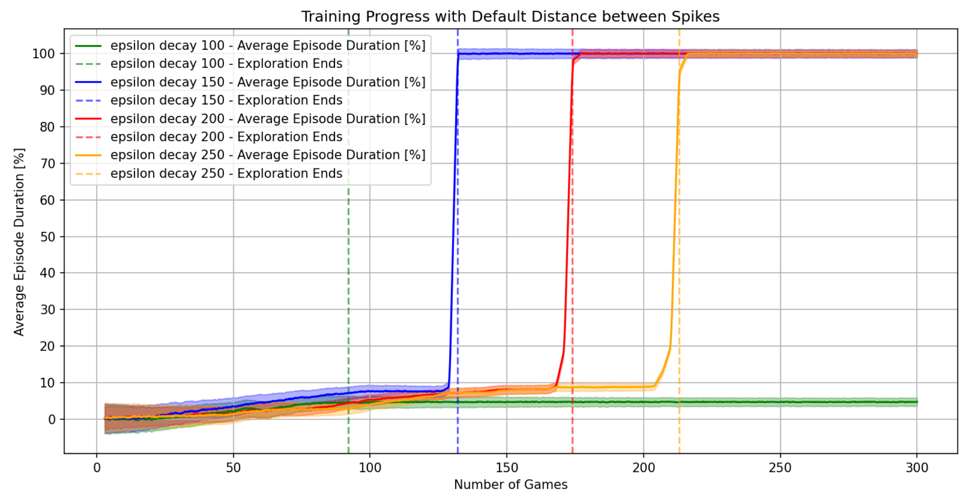


Figure 4.3: Evolution of average episode duration for different ϵ_d values over 300 games. Each curve represents an average, supported by 95% confidence intervals from 20 runs.

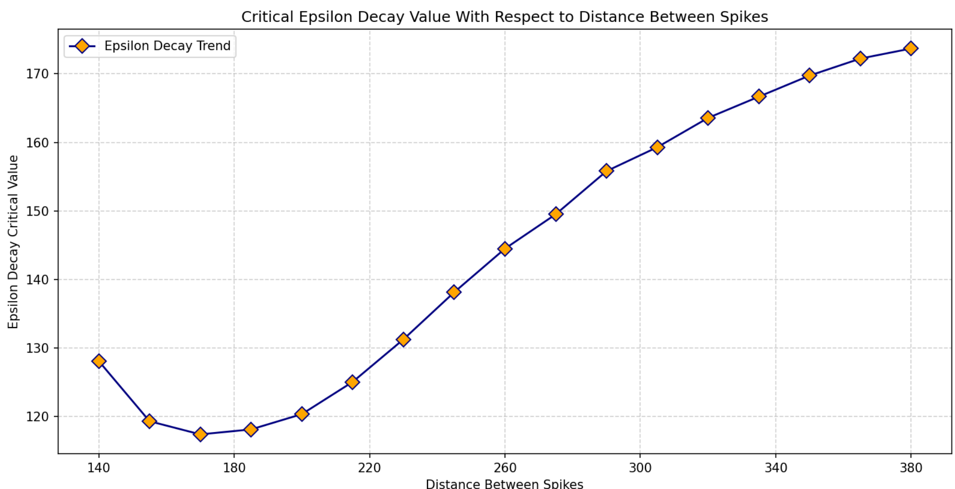


Figure 4.4: Estimated $\epsilon_{d,crit}$ values as a function of the distance between spikes, derived from a grid search with an absolute error of 2.

These observations underscore that even in deterministic settings with such simple levels, the complexity of learning dynamics can be counterintuitive and difficult to predict. Complexity is influenced by multiple factors: the chosen algorithm, its hyperparameters, the neural network architecture, the reward function, and the game environment itself. This reinforces the value of an experimental approach to both justify and elucidate observed trends.

4.3 Adding Random Disturbances to the Previous Deterministic Levels

In this section, we explore the impact of introducing random disturbances to the previously deterministic spike levels on learning complexity. We examine uniform disturbances applied to the following game parameters: spike distance, jump amount, gravity, and player speeds. Additionally, we consider dynamic disturbances applied to the speeds of obstacles that are static in the default game setting. The objectives of this analysis are to answer the following questions :

- How does learning complexity evolve with the variance or amplitude of disturbances ?
- Can the agent consistently master the game when the level remains possible ?
- How is the learning process affected if the disturbances increase to the point where the level may become impossible ?

4.3.1 Adding Distance Disturbances Between Spikes

We first modify the deterministic level by introducing stochastic variability in the distance d between consecutive spikes. The distance is now modeled as a uniformly distributed random variable $d \sim U(\bar{d} - \delta, \bar{d} + \delta)$, where δ represents the variability and is proportional to the standard deviation of d. The distance between each pair of consecutive spikes is independent.

Training simulations were conducted for different δ values, using an exploration decay rate of $\epsilon_d = 250$ to ensure adequate exploration. The results, depicted in Figure 4.5, indicate that the number of training games required to master the level, denoted as $N(100\%)$, increase with δ. Notably, very high δ values, such as $\delta = 40$, show limited convergence, barely mastering 9% of the level on average.

Further analysis involved plotting the relationship between $N(100\%)$ and δ, as shown in Figure 4.6. This relationship generally displays a linear increase with small δ values, transitioning to a steeper ascent as δ approaches 20, beyond which the curve asymptotes. This pattern suggests that levels become increasingly challenging as δ increases, potentially reaching a point of impossibility.

This impossibility arises because as δ increases, the probability that the distance d between some spikes is less than the necessary jump distance $\Delta = \bar{\Delta} = 140$ becomes non-negligible. This arises for $\delta \geq 20$ as the minimal spike distance becomes then $\bar{d} - \delta = 160 - \delta \leq 140$. If enough consecutive spikes feature $d < \Delta$, the level may effectively become impassable, as even optimal jumping at every opportunity would not suffice to avoid obstacles.

Actually, despite the possibility of levels becoming non-viable with $\delta \geq 20$, we can still use the Monte Carlo method to estimate the average episode duration of an effective policy under varying δ values. The optimal policy seems hard to define in close form here, we will thus define an effective policy based on heuristics.

Here, the heuristic policy can be succinctly described as jumping at the greatest possible distance from a spike, ensuring that the jump amount $J = \bar{J} = 10$ is sufficient to clear the next spike. This distance corresponds to $D_{crit} = 116$, which was derived from equation (4.6). If it is not

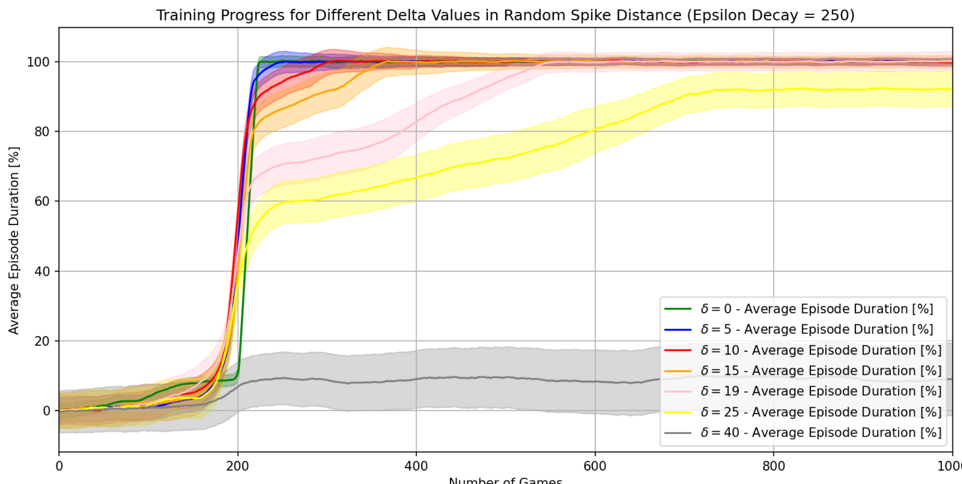


Figure 4.5: Evolution of average episode duration across different δ values in a random spike distance setting, showing increased variability in level difficulty. Each curve is supported by a 95% confidence interval from 20 runs.

possible to adhere to the first rule as the player is still in the air, the next best action is to jump immediately upon landing. The heuristic policy can thus be summarized as:

- Avoid jumping if the distance to the next spike's center is greater than $D_{\text{crit}} = 116$.
- If within D_{crit}, jump at the first opportunity upon landing.

Using this policy framework, we implemented Monte Carlo simulations to derive accurate estimates for the average episode duration across different δ values. The results, depicted in Figure 4.7, show a decline in episode duration from 100% to approximately 20% as δ increases, illustrating the diminishing effectiveness of the heuristic policy with greater level randomness.

Note from Figure 4.5 we computed the average episode duration for $\delta = 25$ and $\delta = 40$. For each δ value, we trained 20 agents and allowed them to play 1000 games post-training to assess their average episode duration. The results for both δ values are shown in Figures 4.8 and 4.9. We observed that for $\delta = 25$, the average episode durations varied significantly among runs, with some runs achieving performances near the ones of heuristic policy, while others lagged behind. The heuristic policy yields approximately 95% average episode duration, with the actual average across runs being around 91.2%, showing a deficit of less than 4 percentage points.

For $\delta = 40$, which significantly increases the game's difficulty and likely renders it impossible, the overall average episode duration across 20 runs was approximately 8.6%, which is about 30 percentage points below the performance achieved by the heuristic policy. To provide a better comparison, we introduced a dummy agent programmed to play randomly at the same δ value. After assessing its performance over 1000 games, the dummy agent surprisingly outperformed the trained agents, achieving an average episode duration of 13.2%. This outcome is depicted in Figure 4.10 and suggests instability in the training process. The likely cause of this instability is the excessive difficulty introduced by $\delta = 40$, which may be too great for the agents to develop effective strategies through conventional training methods.

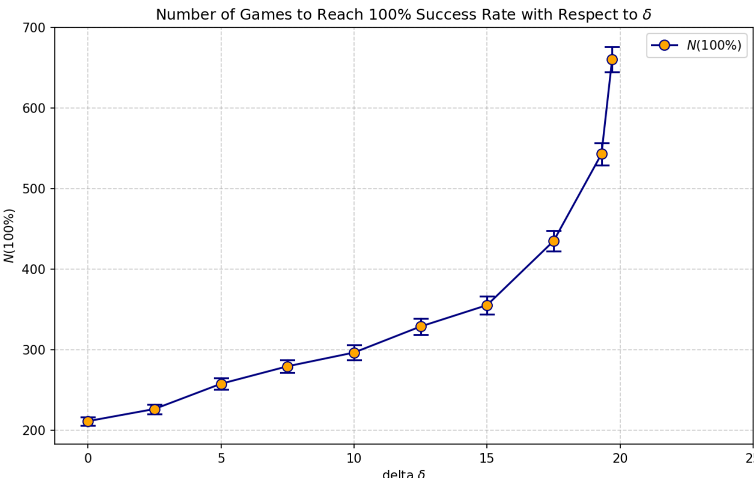


Figure 4.6: Evolution of $N(100\%)$ with respect to δ value, in random spike distance setting. The curve is supported by 95% confident intervals.

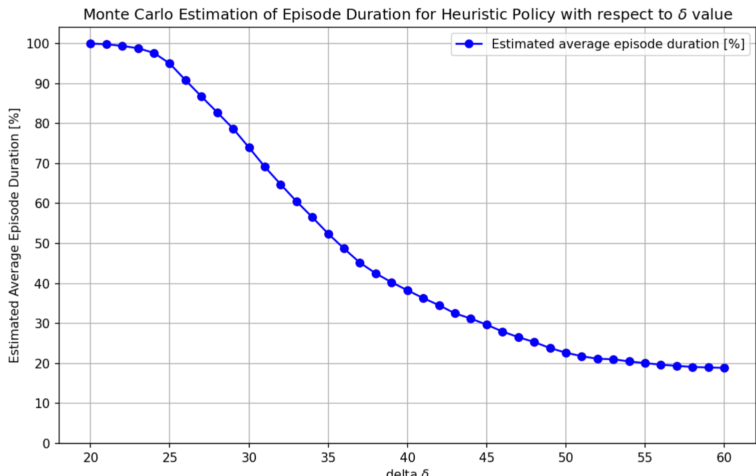


Figure 4.7: Monte Carlo estimation of average episode duration for different δ values with heuristic policy. The experiment occurs in random spike distance setting using 1000 simulations, such that the confidence intervals are not visible on figure.

In conclusion, the agent consistently masters the game as long as the level remains possible. The relationship between the number of training games $N(100\%)$ required for game mastery and the disturbance standard deviation proportional to δ appears linear for small deviations. However, it increases asymptotically as the game approaches impossibility. When the level is impossible, the agent's learning remains consistent for bounded deviations δ. As these deviations become too large, the agent's learning performance declines, eventually becoming worse than that of a dummy agent playing the game randomly.

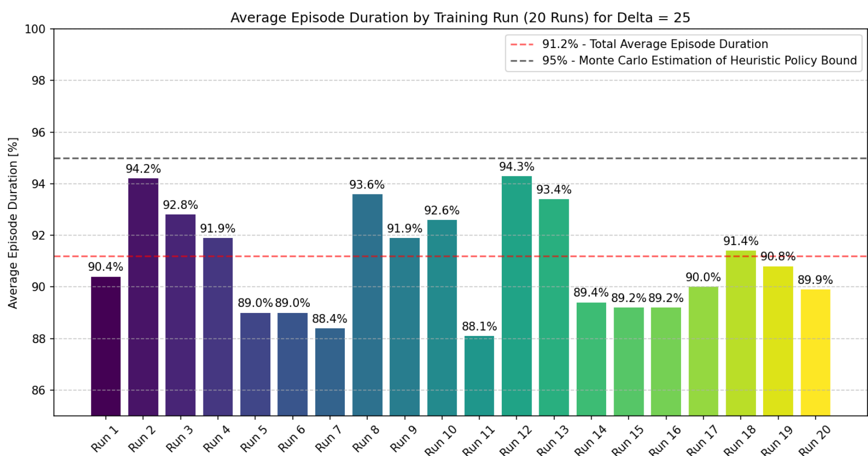


Figure 4.8: Distribution of average episode durations across 20 trained agents over 1000 games for $\delta = 25$. The graph includes a comparison with the Monte Carlo estimated episode duration achievable under the heuristic policy.

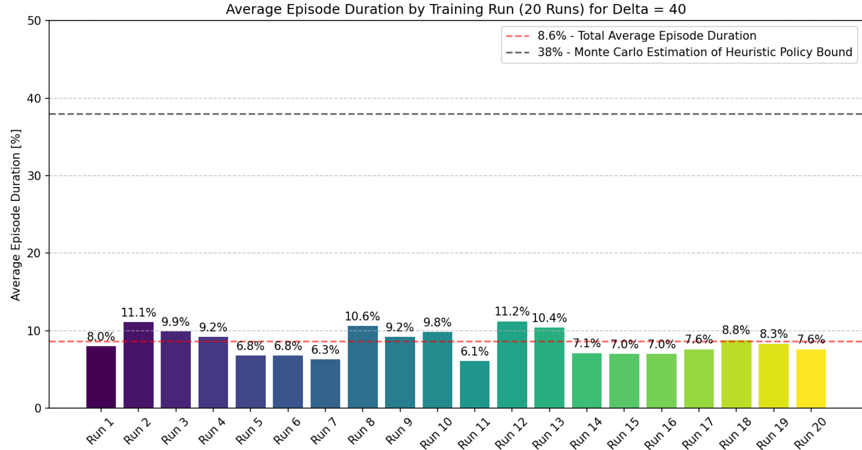


Figure 4.9: Distribution of average episode durations across 20 trained agents over 1000 games for $\delta = 40$. The graph includes a comparison with the Monte Carlo estimated episode duration achievable under the heuristic policy.

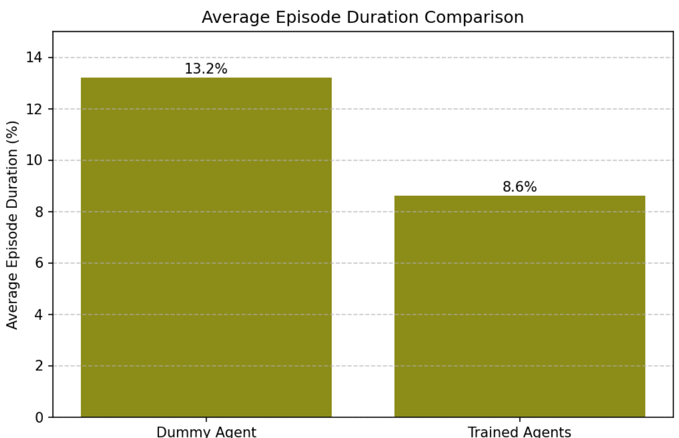


Figure 4.10: Comparison of average episode duration between dummy agent and trained agents in random distance setting with $\delta = 40$.

4.3.2 Adding Jumping Disturbances

We just showed the impact of uniform disturbance applied on spike distance. The idea is to explore the impact of disturbance applied to another game parameter. In this scenario, we return to a level composed solely of spike obstacles with a deterministic distance d. However, we introduce now disturbances on the jump amount J, which is now stochastic and uniformly distributed as $J \sim U(\bar{J} - \delta, \bar{J} + \delta)$, where δ represents the variability proportional to the standard deviation of J. Each jump amount is determined independently for every jump action.

We conducted training simulations across various δ values, and the results are depicted in Figure 4.11 for selected δ values. We set $\epsilon_d = 250$ as the exploration decay rate to ensure a sufficient exploration margin during the learning process. The findings indicate that as δ increases, so does the learning complexity, along with the number of training games $N(100\%)$ required to master the game. For example, a δ value of 6.0 results in mastering only about 6% of the game on average, indicating that very high δ values do not converge to 100%.

As in the previous subsection, we explored how $N(100\%)$—the number of training games required to master the game—changes with variations in δ. The results are depicted in Figure 4.12. It is evident that $N(100\%)$ generally increases with δ, following a polynomial trend, similar to a quadratic curve, even for small δ values. When δ exceeds 2.5, mastering the game appears to be impossible.

The feasibility of jumps at different δ values can be understood through the ballistic equation (4.2). The maximal trajectory height achieved with a given jump amount J is $\frac{J^2}{2g}$. In the worst-case scenario where $J = \bar{J} - \delta$, the jump must be sufficient to clear the spike height l_s, leading to the inequality:

$$\frac{(\bar{J} - \delta)^2}{2g} \geq l_s \quad \Rightarrow \delta \leq \bar{J} - \sqrt{2 l_s g} \approx 2.58. \tag{4.7}$$

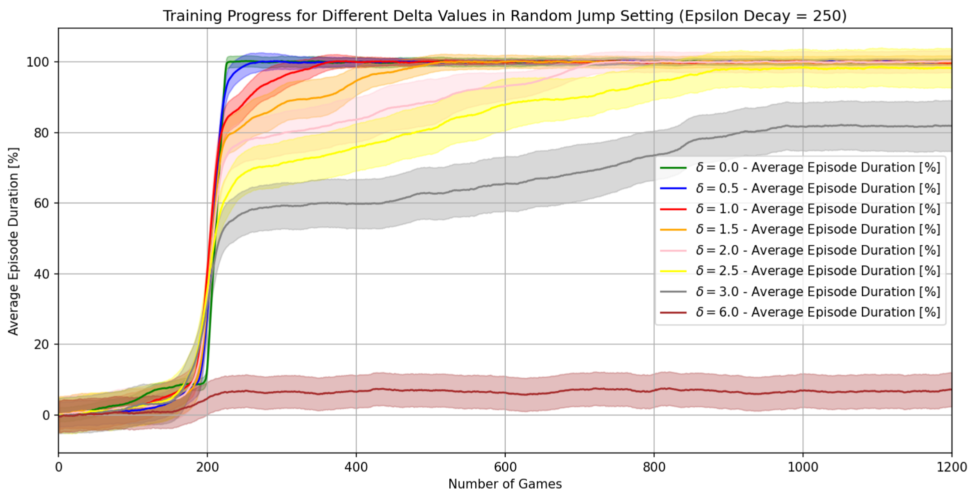


Figure 4.11: Evolution of average episode duration for different jump amount distributions. Each curve is supported by a 95% confidence interval based on 20 runs, illustrating the increased difficulty as δ increases.

For $\delta = 2.58$, the maximum jump distance, when $J = \bar{J} + \delta$, is calculated as:

$$\frac{h_s}{g}(\bar{J} + \delta + \sqrt{(\bar{J} + \delta)^2 - 2l_s g}) \approx 158.64. \tag{4.8}$$

This jump distance is slightly less than the spike separation distance $d = \bar{d} = 160$, implying that the level remains feasible for $\delta \leq 2.58$.

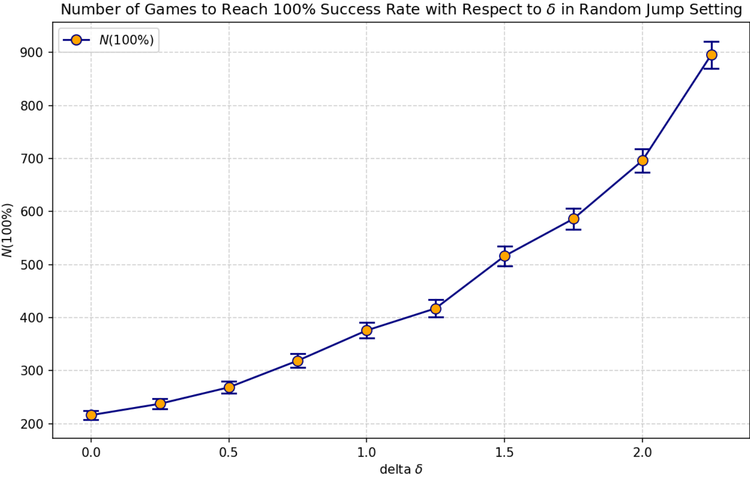


Figure 4.12: Evolution of $N(100\%)$ with respect to δ in a random jump amount setting, showing how game mastery becomes increasingly difficult as δ increases. The curve is supported by 95% confidence intervals.

Despite the probability of levels becoming non-viable with $\delta \geq 2.58$, we can still estimate the average episode duration for an effective heuristic policy using the Monte Carlo method under such conditions with varying δ. Once again, the optimal policy is hard to derive here, we will

thus focus on an effective heuristic. The heuristic policy, expressed in a closed form, involves maximizing the jumping distance from a spike. Specifically, the strategy dictates jumping from as far as possible, ensuring that the reduced jump amount $\bar{J} - \delta$ still allows the player to clear the spike. If it is not feasible to maintain this distance, the policy requires jumping immediately upon landing to attempt avoidance of the next spike.

The critical jumping distance from a spike's center, calculated using Equation (4.2), is given by:

$$\frac{h_s}{g}(\bar{J} - \delta + \sqrt{(\bar{J} - \delta)^2 - 2l_s g})$$

The heuristic policy can thus be summarized as follows:

- Avoid jumping unless you are closer than $\frac{h_s}{g}(\bar{J} - \delta + \sqrt{(\bar{J} - \delta)^2 - 2l_s g})$ from the next spike center.
- Once within this critical distance, jump at the first opportunity after touching the ground.

We implemented the heuristic policy and used the Monte Carlo method to accurately estimate the average episode duration for various δ values. The outcomes are depicted in Figure 4.13, showing a decline in episode duration from 100% to approximately 8% as δ increases.

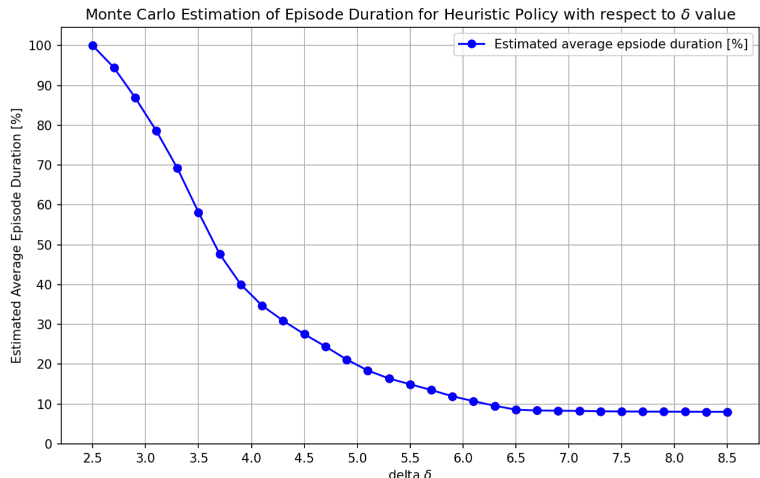


Figure 4.13: Monte Carlo estimation of average episode duration for various δ values under the heuristic policy. This illustrates the impact of increased jump variability on potential game completion rates.

Referring back to Figure 4.11, we computed the average episode durations for $\delta = 3$ and $\delta = 6$. For each δ value, we trained 20 agents and allowed them to play 1000 games post-training to evaluate their performance. The results, depicted in Figures 4.14 and 4.15, are consistent with the observations made in the previous subsection. None of the runs exceeded the performance of the heuristic policy.
With $\delta = 3$, the heuristic policy achieves an average episode duration of 87%, compared to the overall average of 79.6% across all runs. For $\delta = 6$, the heuristic policy achieves only 12%, while the collective average of the runs is about 7%. In contrast, a dummy agent acting randomly

achieves an 8% average episode duration, as illustrated in figure 4.16.

These findings confirm as in previous subsection that while the learning process performs reasonably well with small noise variances ($\delta = 3$), it significantly deteriorates as δ increases, to the extent that it does not even surpass the performance of a dummy agent.

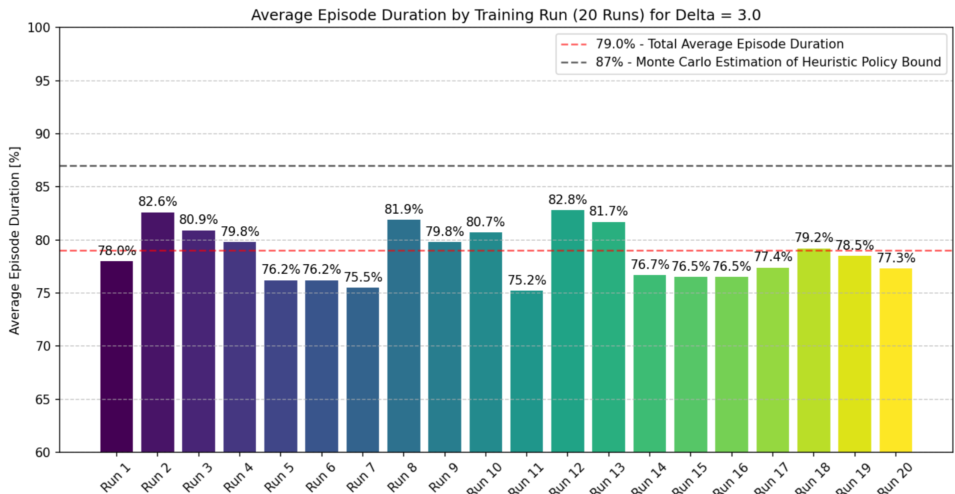


Figure 4.14: Distribution of average episode durations across agents trained for 1000 games at $\delta = 3$ within a randomized jump amount environment. The results are compared to the Monte Carlo estimated heuristic policy performance, highlighting discrepancies between learned and optimal behaviors.

In conclusion, our observations here mirror those found with disturbances applied to spike distance. The agent continues to consistently master the game as long as the level remains possible. The relationship between the number of training games $N(100\%)$ required for game mastery and the disturbance standard deviation proportional to δ is now polynomial (quadratic) for small deviations. However, it increases asymptotically as the game approaches impossibility. When the level is impossible, the agent's learning remains consistent for bounded deviations δ. As these deviations become too large, the agent's learning performance declines, eventually becoming worse than that of a dummy agent playing the game randomly.

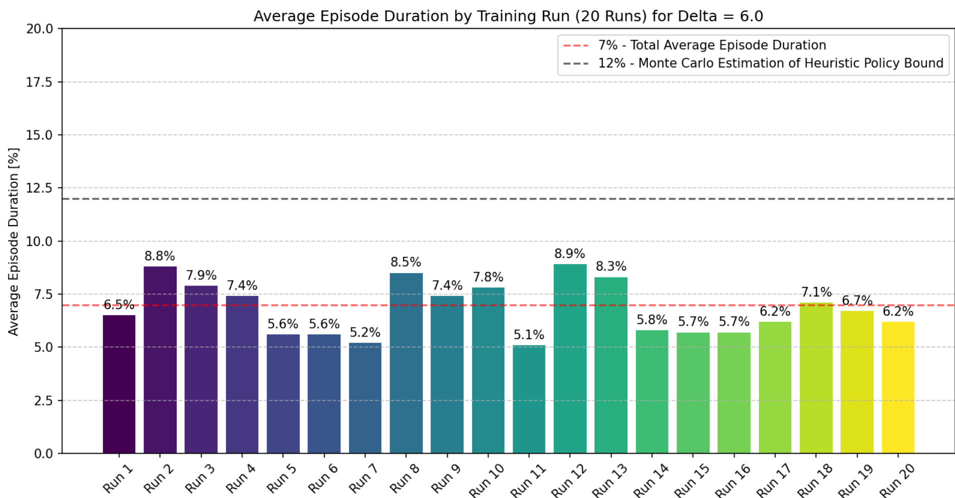


Figure 4.15: Average episode durations for agents trained over 1000 games at $\delta = 6$ in a variable jump amount scenario. This figure contrasts actual gameplay outcomes with those predicted under the heuristic policy from Monte Carlo estimates, demonstrating the challenge of higher δ settings.

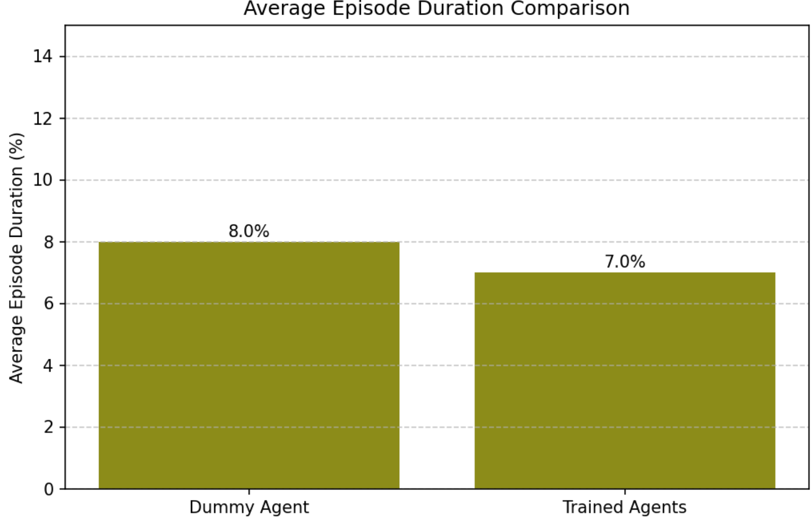


Figure 4.16: Comparison of average episode duration between dummy agent and trained agents in random jump setting with $\delta = 6$.

4.3.3 Adding Gravity Disturbances

We consider back a level with only spike obstacles with a distance $d = \bar{d} = 160$ that is deterministic and a jump amount $J = \bar{J} = 10$ deterministic also, however the gravity before each game will be uniformly distributed as $g \sim U(\bar{g} - \delta, \bar{g} + \delta)$, where δ is proportional to the standard deviation of gravity g. This gravity value is independent between each game.

The training curves and general results are here really similar to both previous subsections. We will thus here only focus on the evolution of $N(100\%)$ with δ. The results are depicted on figure 4.17. We can see that $N(100\%)$ is generally increasing with δ. This increase seems linear for small gravity disturbances. The slope increases a lot for $\delta > 1$ and the game mastery seems impossible for $\delta > 1.2$.
Actually, we can consider the two following extreme cases, either $g = \bar{g} - \delta$ and $g = \bar{g} + \delta$. In the first case, the risk would be that the jump distance Δ is larger than the spike distance $d = \bar{d}$. This gives the following constraint to respect

$$\frac{2h_s J}{\bar{g} - \delta} \leq \bar{d} \quad \Rightarrow \delta \leq \bar{g} - \frac{2h_s J}{\bar{d}} \approx 0.11 \tag{4.9}$$

In the second case $g = \bar{g} + \delta$, the risk would be not to be able to jump above the spike. However, the maximal height in a jump is given by $\frac{J^2}{2(\bar{g}+\delta)}$. This gives a maximal height of 51.5 for $\delta = 0.11$, meaning we still jump above the spike height of $l_s = 32$ with such δ. Therefore, the level becomes indeed impossible for non negative probability with $\delta \geq 0.11$. This supports figure 4.17 trend.

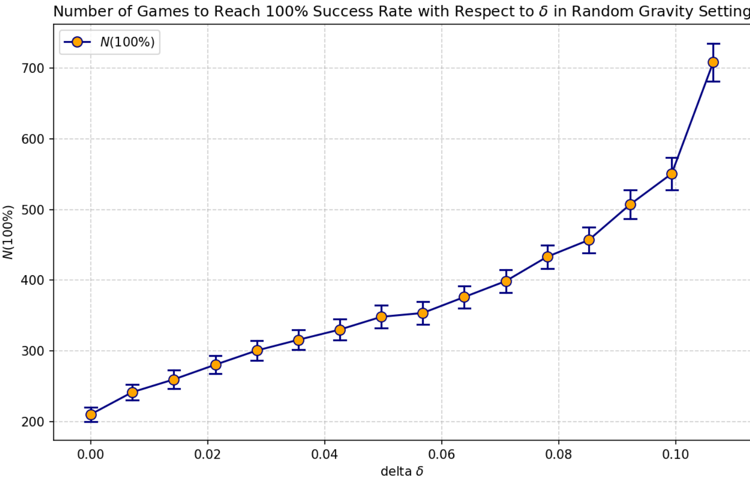


Figure 4.17: Evolution of $N(100\%)$ with respect to δ value, in random gravity setting. The curve is supported by 95% confident intervals.

We could then also derive an effective heuristic policy similarly to previous subsection and compare the learning performances between the heuristic policy and the agent when the level becomes impossible. We would conclude as previously that some training processes find nearly the heuristic policy for small noise variances, while the learning process becomes more unstable with too large noise variances.

The conclusions related to the research questions are identical to both previous subsections.

4.3.4 Adding Player Horizontal Speed Disturbances

We finally apply the disturbance to a fourth game parameter. We revisit a scenario where a level consists solely of spike obstacles spaced at a deterministic distance $d = \bar{d} = 160$, with a fixed jump amount $J = \bar{J} = 10$ and gravity $g = \bar{g} = 0.86$. However, before each game, the horizontal player speed h_s is varied, following a uniform distribution $h_s \sim U(\bar{h}_s - \delta, \bar{h}_s + \delta)$, where δ is proportional to the standard deviation of h_s. Each game's horizontal speed is determined independently.

Similar to previous subsections, the training curves and overall results show consistent patterns. Hence, we focus on the relationship between $N(100\%)$ and δ. The results, illustrated in Figure 4.18, indicate that $N(100\%)$ generally increases with δ. The relationship appears linear for minor perturbations in speed, but the slope moderates and then sharply rises for $\delta > 0.8$. Mastery of the game becomes highly improbable for $\delta > 0.85$.

The primary concern here is that excessive horizontal speed could result in a jump distance exceeding the spike separation $\bar{d} = 160$. The critical case occurs when $h_s = \bar{h}_s + \delta$. From Equation (4.2), we derive the necessary condition for the game to remain feasible:

$$\frac{2(\bar{h}_s + \delta)J}{g} \leq \bar{d} \Rightarrow \delta \leq \frac{\bar{d}g - 2\bar{h}_s J}{2J} = 0.88$$

This suggests that the game could become unplayable if $\delta > 0.88$, corroborating the observations in Figure 4.18.

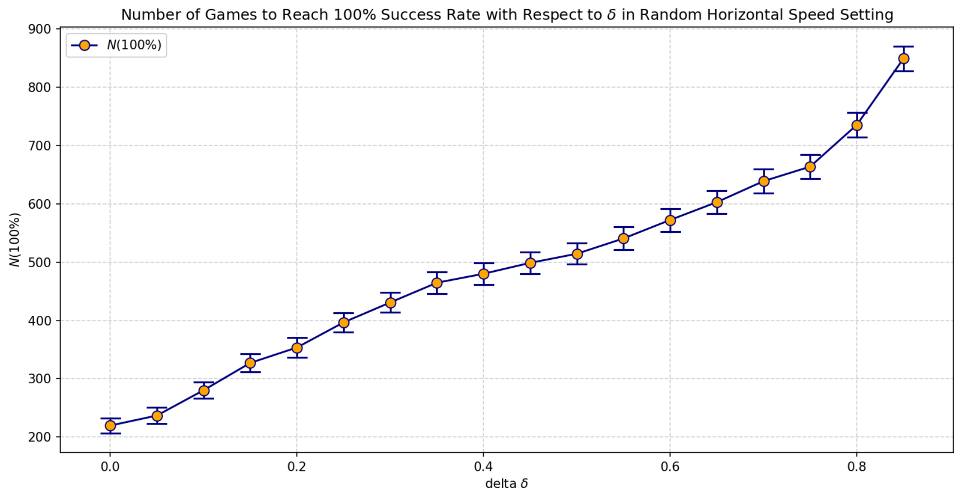


Figure 4.18: Evolution of $N(100\%)$ with respect to δ in a random player horizontal speed setting, highlighted by 95% confidence intervals.

Consistent with earlier findings, we could compare the performance of an effective heuristic policy to the learned policy when the level becomes nearly impossible. Again, we would find that large δ values destabilize the learning process, particularly when the increased noise in horizontal speed makes the level unplayable.

The conclusions related to the research questions are consistent with the findings of the last three subsections. We conclude that uniform disturbances applied to all four game parameters similarly impact the learning complexity, albeit with variations in the relationship between $N(100\%)$ and

the disturbance deviation δ. This relationship is either linear or polynomial for small deviations, depending on the parameter. This difference requires further investigation, as no theoretical background or insights currently explain it.

4.3.5 Adding Moving Spike Disturbances

In the previous subsections, we explored uniformly distributed disturbances such as changes in spike distance, jump amount, gravity, and player horizontal speed, with all objects fixed relative to the ground. Now, we introduce a new dynamic where spikes move horizontally relative to the ground, with their speed varying over time. In this setup, all other game parameters—spike distance, jump amount, gravity, and player speed—are reset to their default, constant values. We consider two dynamics for the moving spikes, a sinusoidal and a random walk distributed horizontal velocity. The objective is to determine how the learning complexity evolves with the variance/amplitude of those dynamic disturbances, and whether or not the agent manage to learn correctly.

Sinusoidal Spike Speed Dynamics

In this scenario, the relative velocity of a spike from the ground $v(t)$ at each frame time t is described by:

$$v(t) = A\sin(\omega t + \phi)$$

Here, A is the amplitude, ω is the angular frequency, and ϕ is a random phase shift uniformly distributed as $\phi \sim U(0, 2\pi)$, independent from one game to another.

The learning curves for angular frequencies $\omega = \frac{2\pi}{24}$ and $\omega = \frac{2\pi}{48}$, across various amplitudes $A = 0, 1, 5, 10$, are illustrated in Figures 4.19 and 4.20. We maintained an exploration decay rate of $\epsilon_d = 250$, consistent with previous subsections. The duration $t \approx 24$ frames corresponds to the time a player spends airborne during a jump.

Analysis of these figures reveals that $N(100\%)$ increases with amplitude A. Additionally, $\omega = \frac{2\pi}{24}$ results in a higher $N(100\%)$ compared to $\omega = \frac{2\pi}{48}$, indicating increased learning complexity with higher sinusoidal frequencies. This outcome is logical as higher frequencies and larger amplitudes generate more unpredictable spike movements, demanding a more robust policy for mastering the game.

In order to discern general trends related to the impact of oscillating amplitude, we computed results for various equidistant amplitudes ranging from $A = 0$ to $A = 10$, for both $\omega = \frac{2\pi}{24}$ and $\omega = \frac{2\pi}{48}$. The outcomes are depicted in Figures 4.21 and 4.22.

The learning curves presented in these figures show that the complexity of learning increases monotonically with A for both ω values. Specifically, the complexity increase tends to be linear for $\omega = \frac{2\pi}{24}$. This observation is logical considering that the critical jump timings coincide with the peak magnitudes of the sinusoidal speed. At these peaks, the window for accurate jump timing narrows linearly with increasing speed amplitude A.

For $\omega = \frac{2\pi}{48}$, the increase also appears linear but with more variation in the slope. This irregularity might be attributed to the complex interplay of sinusoidal oscillations in moving spike speed, which complicates the straightforward linear relationship between $N(100\%)$ and A.

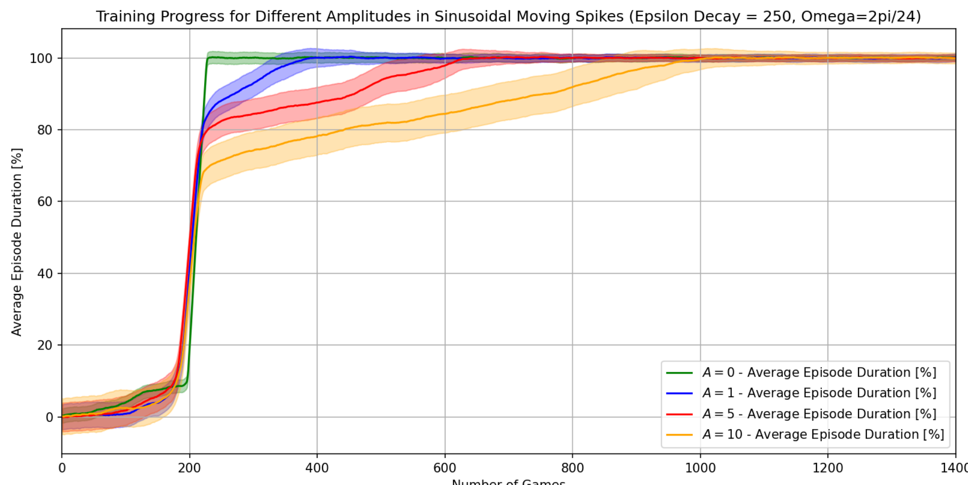


Figure 4.19: Average episode duration for $\omega = \frac{2\pi}{24}$ in a dynamic setting with sinusoidal moving spike speeds. Exploration decay rate $\epsilon_d = 250$ was used. Each curve represents an average, supported by 95% confidence intervals from 20 simulation runs, demonstrating the variability and reliability of the results.

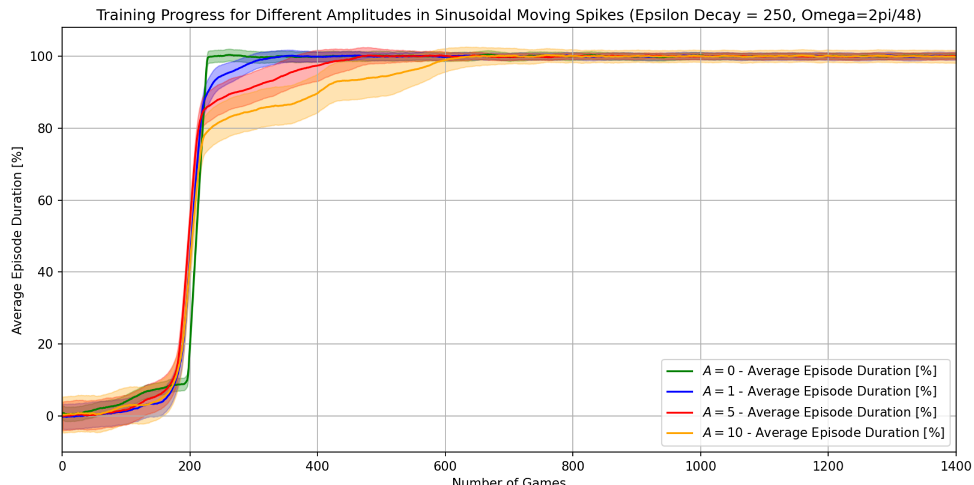


Figure 4.20: Average episode duration for $\omega = \frac{2\pi}{48}$ under conditions of sinusoidal moving spike speeds, with $\epsilon_d = 250$ guiding the exploration process. Each data point is derived from 20 runs, illustrated with 95% confidence intervals to highlight the consistency and trends observed in the data.

We conclude that the chosen values of ω and A are not too extreme for the agent to learn to master the game, given an exploration parameter of $\epsilon_d = 250$. The complexity generally exhibits a linear relationship with A, although the trend is somewhat irregular, possibly influenced by the oscillatory nature of the sinusoidal movements. Notably, the angular frequency ω significantly affects the learning complexity, as higher frequencies necessitate more precise jump timings.

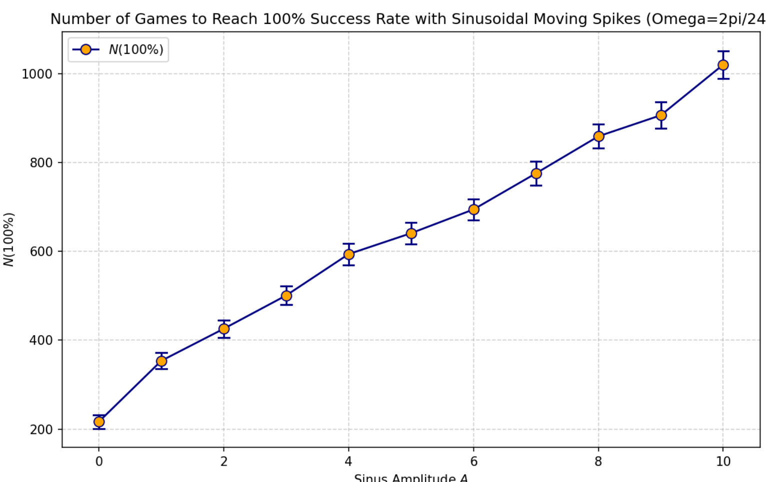


Figure 4.21: Estimated $N(100\%)$ as a function of sinusoidal amplitude A for $\omega = \frac{2\pi}{24}$. The averages are calculated from 20 runs, each accompanied by a 95% confidence interval to highlight variability and trends.

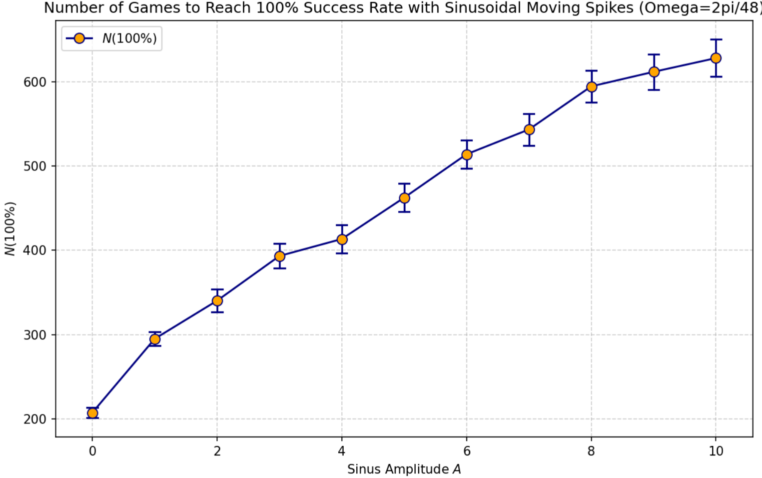


Figure 4.22: Estimated $N(100\%)$ in relation to sinusoidal amplitude A for $\omega = \frac{2\pi}{48}$. Each data point represents an average derived from 20 runs, supported by 95% confidence intervals to illustrate statistical consistency and variation.

Random Walk Spike Speed Dynamics

In this analysis, we explore a scenario where the horizontal velocity of spikes, relative to the ground, follows a Random Walk motion model:

$$v(t) = v(t-1) + \epsilon, \quad \text{where} \quad \epsilon \sim \mathcal{N}(0, \sigma) \tag{4.10}$$

Here, ϵ represents Gaussian noise with standard deviation σ, drawn independently at each time frame t. Consequently, the velocity variance over time is described by:

$$\mathbb{V}[v(t)] = \sigma^2 t \tag{4.11}$$

This model introduces a linearly increasing uncertainty in spike movement as a function of time.

We assessed the learning complexity across various σ values and presented the corresponding learning curves in Figure 4.23, using $\epsilon_d = 250$ as exploration rate. Consistent with theoretical expectations, learning complexity increases with σ, reflecting the proportional rise in velocity variance (σ^2). However, due to the potential for extreme outlier values of ϵ, there is always a non-negligible probability that the level becomes unplayable. Consequently, we shifted our focus to evaluating $N(95\%)$ instead of $N(100\%)$, as the latter would not provide meaningful insights into the achievable performance under such variable conditions.

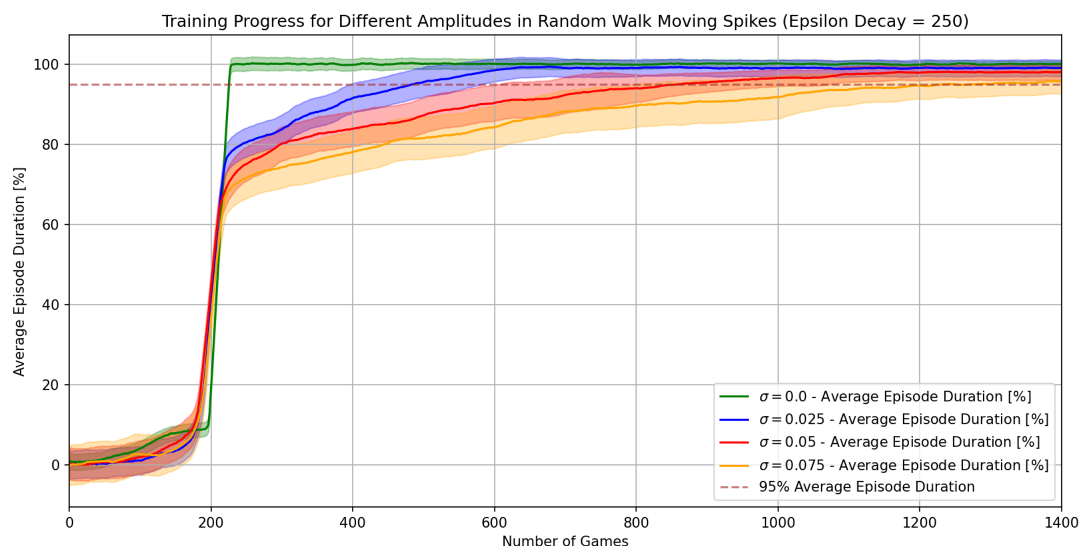


Figure 4.23: Evolution of average episode duration in a setting with Random Walk spike velocity motion for various σ values. Each curve represents an average outcome, bolstered by a 95% confidence interval from 20 runs, highlighting the response to increasing complexity.

The evolution of $N(95\%)$ with varying σ values is illustrated in Figure 4.24. The increase in learning complexity appears to be worse than linear, especially as σ approaches 0.085. Beyond this point, the agent fails to converge to a 95% average episode duration, indicating a significant challenge in adapting to high levels of speed variability. This would probably require a larger exploration rate ϵ_d, or even more stable reinforcement algorithms as PPO (Proximal Policy Optimization) [7], even though it takes as input continuous action spaces instead of discrete ones.

This outcome seems counterintuitive given that the standard deviation of velocity is directly proportional to σ (see Equation (4.11)), suggesting a possible linear trend. This shows once again the intricacies of reinforcement learning complexity in practice, suggesting here worse expected impact of Gaussian noise when it becomes large enough.

Using the Random Walk motion model specified in Equation (4.10), we understand that $v(t)$ follows a normal distribution $N(0, \sqrt{t}\sigma)$. This distribution results from the sum of t independent normal distributions, still yielding a normal distribution. Thus by the Central Limit Theorem [19], a 95% confidence interval ($\mathrm{CI}_{95\%}$) for the spike's horizontal speed at time t can be expressed as:

$$\mathrm{CI}_{95\%} = [-1.96\sigma\sqrt{t}, 1.96\sigma\sqrt{t}]$$

Given the level's total frame time $T = 400$ and player horizontal speed $h_s = 6$, the most extreme case of $\sigma = 0.085$ leads to a 95% probability that:

$$v(t) \in [-3.332, 3.332]$$

This implies that the spike's velocity magnitude is bounded by 55.5% of the player's speed in 95% of cases, as illustrated in Figure 4.24. Above $\sigma = 0.085$, the velocity changes become too extreme for the agent to consistently achieve a 95% average episode duration, illustrating the difficulty in maintaining performance under such variable conditions.

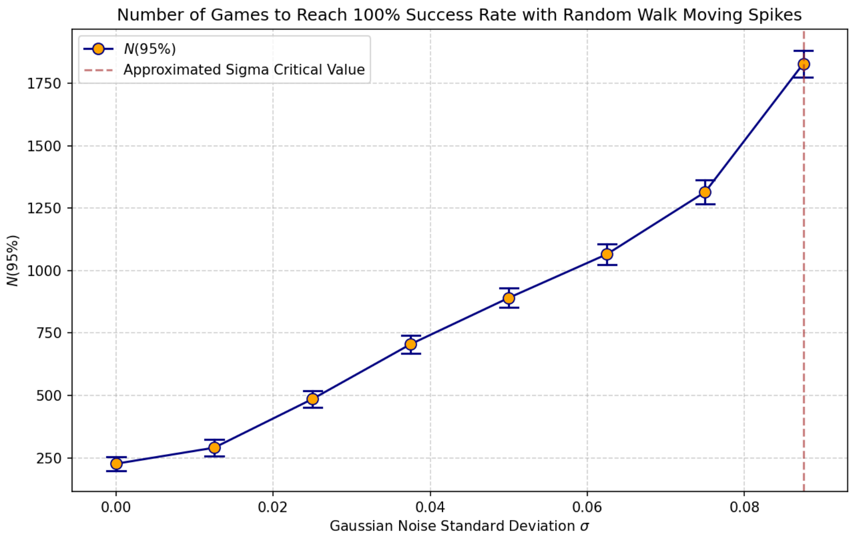


Figure 4.24: Average $N(95\%)$ values for different Random Walk noise standard deviations σ. Each average value is supported by a 95% confidence interval, based on 20 runs, to illustrate variability and certainty in the measurements.

We conclude that the agent almost manages to master the game, achieving a 95% threshold in average episode duration, up to a Gaussian noise deviation of $\sigma = 0.085$. This demonstrates a consistent learning process. Above this standard deviation, the velocity variance becomes too large for the agent to maintain 95% accuracy. It is important to note that game mastery here does not make sense compared to sinusoidal disturbances, as Gaussian noise is not bounded.

4.4 Increasing the Action Space Cardinality

In this section, we explore the effects of expanding the action space cardinality within the game environment, specifically focusing on levels composed solely of spikes. We consider multiple jump heights in our action space and investigate how learning complexity evolves with increases in action space cardinality. Both deterministic and randomly disturbed levels are examined. The objectives of this analysis are to address the following questions:

- How does the learning complexity increase with the action space cardinality?
- What happens to the learning complexity when we combine an increasing action space with random disturbances?
- What is the influence of exploration on the learning process in such settings?

4.4.1 Increasing Action Space Cardinality in a Deterministic Setting

Initially, our focus will be on modifying the action space in a deterministic level setting of *Geometry Dash*, which previously consisted only of two possible actions at each time frame t: jumping or not jumping. We propose to enrich the action space by introducing multiple discrete jump amounts, thereby allowing the agent to select from a set of n distinct jump magnitudes. This adjustment expands the action space cardinality to $n+1$, accounting for the additional option of not jumping.

The level design will remain deterministic, featuring only spikes as obstacles. However, to accommodate the expanded action space, we will introduce n types of obstacles, each consisting of a cluster of i spikes, where i ranges from 1 to n. This setup aims to test the agent's ability to choose optimal jump heights corresponding to different obstacle configurations. An illustrative scheme of this configuration for $n = 3$ is shown in Figure 4.25, depicting a sequence of obstacles with decreasing numbers of spikes. Each obstacle type is consistently separated by a distance of $d = 160$, ensuring uniform challenge spacing.

Each of the n jump amounts is specifically tailored for one type of obstacle, with each obstacle type spaced by a distance $d = 160$. The jump amount i is designed to clear i spikes, by initiating a jump 32 units (spike distance $l_s = 32$) before an obstacle and by landing 32 units past it, representing a jump distance $\Delta_i = l_s(i+2)$. The formula to compute the jump amount required to jump over i spikes is derived from Equation (4.2) and can be expressed as:

$$J_i = \frac{(i+2)gl_s}{2h_s} \approx 2.293(i+2)$$

This model accommodates obstacle configurations ranging from $n = 1$ to $n = 10$. The specific jump amounts for each configuration are detailed in Table 4.2.

We investigated the evolution of average episode duration during learning for various n values, exploring different ϵ_d settings. The results are depicted in Figure 4.26. We observed that $N(100\%)$ increases with the number of jump amounts n, and the increase appears to be polynomial. Notably, while $\epsilon_d = 250$ allows for faster convergence to $N(100\%)$ with smaller n values, the learning complexity escalates more rapidly with increasing n, failing to converge to 100% episode duration

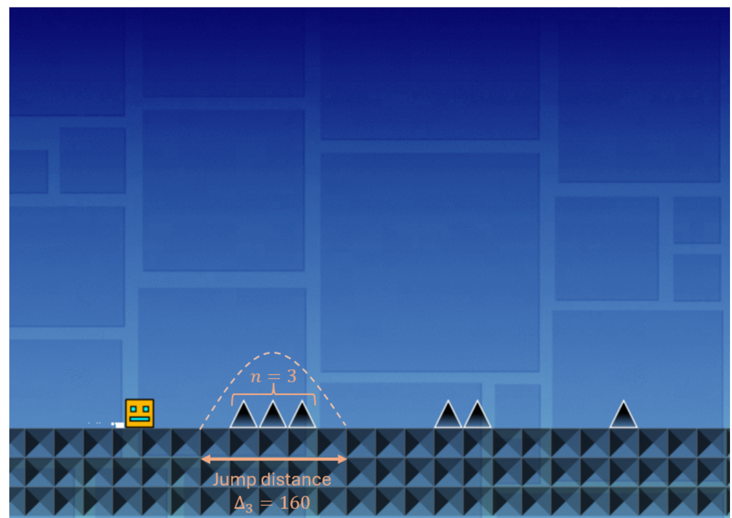


Figure 4.25: Scheme depicting the player in a $n = 3$ obstacle configuration, facing an obstacle with $i = 3$ spikes. This obstacle is followed by two subsequent obstacles consisting of $i = 2$ and $i = 1$ spikes respectively. Each obstacle set is spaced by a distance of $d = 160$.

Number of Spikes i	Jump amount default values $\bar{J}_i$
1	6.88
2	9.17
3	11.47
4	13.76
5	16.05
6	18.35
7	20.64
8	22.93
9	25.23
10	27.51

Table 4.2: Calibrated jump amount value J for each obstacle type composed of n joint spikes

for $n > 6$. Conversely, higher ϵ_d values exhibit a slower increase in complexity, successfully converging to 100% episode duration up to $n = 10$.

This highlights the critical importance of the exploration phase, which is dictated by the ϵ_d value. As the game becomes more complex, an increased level of exploration is necessary. The polynomial growth in complexity associated with the cardinality of the action space $n + 1$ is theoretically supported by findings in the domain of computational learning theory. As indicated by Sutton and Barto (2018), the complexity of reinforcement learning tasks can expand polynomially with the increase in action choices, necessitating a corresponding enhancement in the exploration strategy to effectively learn optimal actions [1]. Their study focuses on non-dynamical systems such as Tic-Tac-Toe. Based on this, we can conjecture that similar expectations might apply to dynamical systems like *Geometry Dash*.

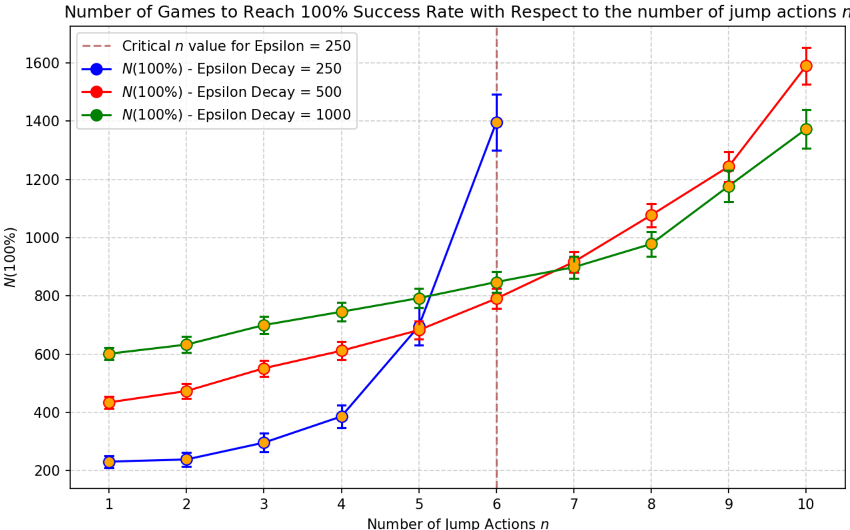


Figure 4.26: Estimation of $N(100\%)$ for varying numbers of jump amounts n, across different ϵ_d values. Each estimated value is reinforced by a 95% confidence interval derived from 20 simulation runs.

4.4.2 Combining Increasing Action Space Cardinality with Jump Disturbances

We now explore the combined effects of increasing the action space cardinality and introducing game disturbances. In this setting, we utilize n distinct jump amounts J_i, where $i \in \{1, 2, ..., n\}$. Each time a jump action i is selected, the corresponding jump amount J_i will vary according to a uniform distribution, $J_i \sim U(\bar{J}_i - \delta, \bar{J}_i + \delta)$, where $\bar{J}_i$ values are detailed in Table 4.2. This variability introduces a stochastic mechanic to each jump, with δ representing the degree of variation.

This experimental setup was tested across three different δ values. The outcomes are illustrated in Figures 4.27, 4.28, and 4.29. It is evident that the complexity of the learning task increases not only with n, the number of jump amounts, but also with δ, the degree of variability in each jump. As expected, both parameters contribute to an increased learning complexity. Interestingly, $N(100\%)$ appears to scale polynomially with n regardless of the δ value, indicating a consistent pattern of complexity increase across different levels of jump variability.

An interesting observation emerges concerning the critical value for n under different settings of ϵ_d and δ. For $\epsilon_d = 250$, the critical n value at $\delta = 0$ remains 6, as noted in the previous section. However, with a slight increase in randomness ($\delta = 0.5$), the critical n rises to 7, suggesting that some level of randomness can enhance exploration of the game mechanics. Conversely, when δ increases to 2.0, the critical n reverts to 6, reflecting the heightened game complexity at higher δ values. This delineates a delicate balance between the complexity introduced by random disturbances and the facilitative role these disturbances play in exploring the game. This interesting observation would deserve further investigations before confirming it.

For $\epsilon_d = 500$, the critical n values are observed to be 9 and 8 for $\delta = 1.0$ and $\delta = 2.0$, respectively. At a higher exploration rate of $\epsilon_d = 1000$, game mastery is achieved across all n values up to

$n = 10$ for $\delta \leq 2$. This underscores the significance of sufficiently balancing exploration and exploitation to adapt to the increasing game complexity.

It is noteworthy that the impact of enlarging the action space significantly influences the critical ϵ_d values, a phenomenon not observed in prior sections. Previously, the levels were likely simple enough for the agent to effectively learn by exploring with only $\epsilon_d = 250$, potentially aided by random disturbances that prevent the agent from becoming mired in repetitive patterns and assist in encountering diverse game dynamics and jump timings.

In conclusion, the agent continues to learn how to master the game with the addition of random disturbances on jump amounts, combined with an increase in action space cardinality, provided that the exploration rate is sufficient. The relationship between $N(100\%)$ and the cardinality $n+1$ appears polynomial, even with added random disturbances on jump amounts. The importance of a sufficient exploration rate is emphasized in these experiments. For a given exploration rate, the critical value of action space cardinality $n + 1$, above which the level is no longer mastered, generally increases with the disturbance variance. Interestingly, we observed that an increase in variance might actually lower this critical value, potentially aiding in better generalization due to the randomness of the variations. This intriguing result warrants further investigation and observations to confirm its validity.

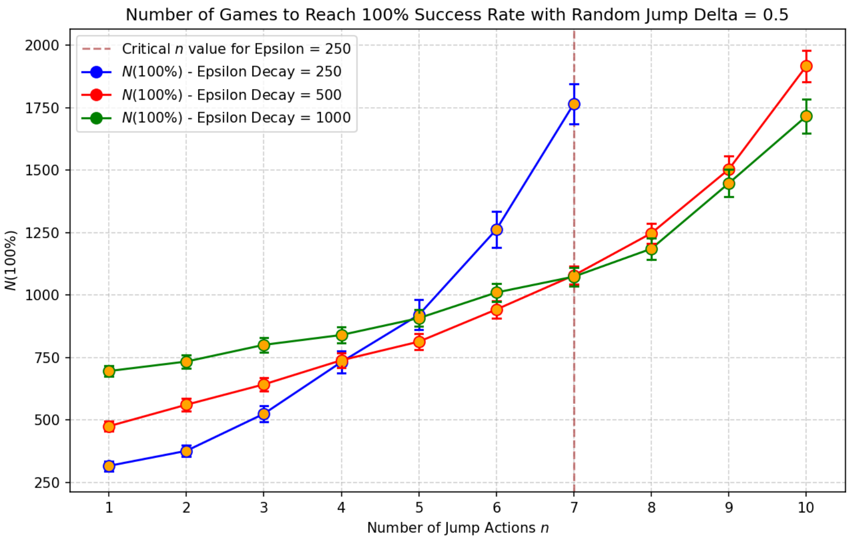


Figure 4.27: Estimation of $N(100\%)$ as a function of the number of jump amounts n, in a setting with random jump variability $\delta = 0.5$. Each value is reinforced by a 95% confidence interval, derived from 20 simulation runs.

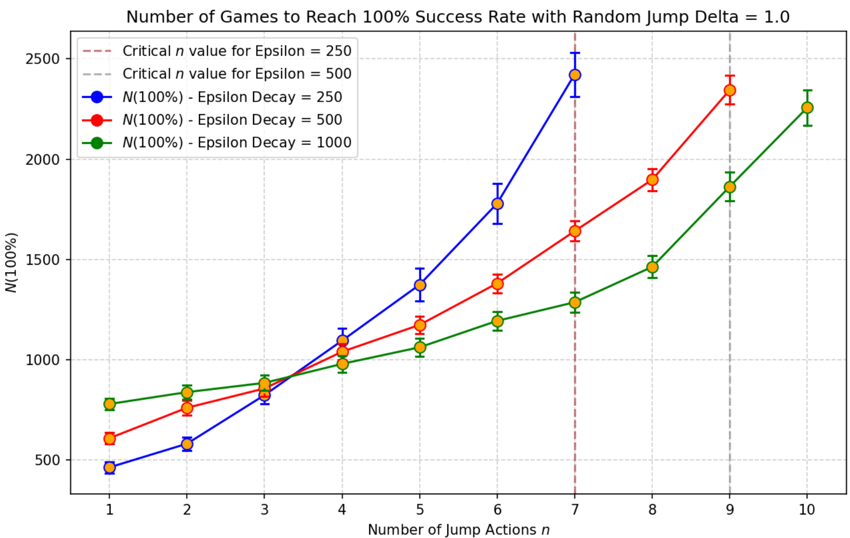


Figure 4.28: Estimation of $N(100\%)$ relative to the number of jump amounts n, under conditions of random jump variability $\delta = 1.0$. Each data point is supported by a 95% confidence interval based on 20 simulation runs.

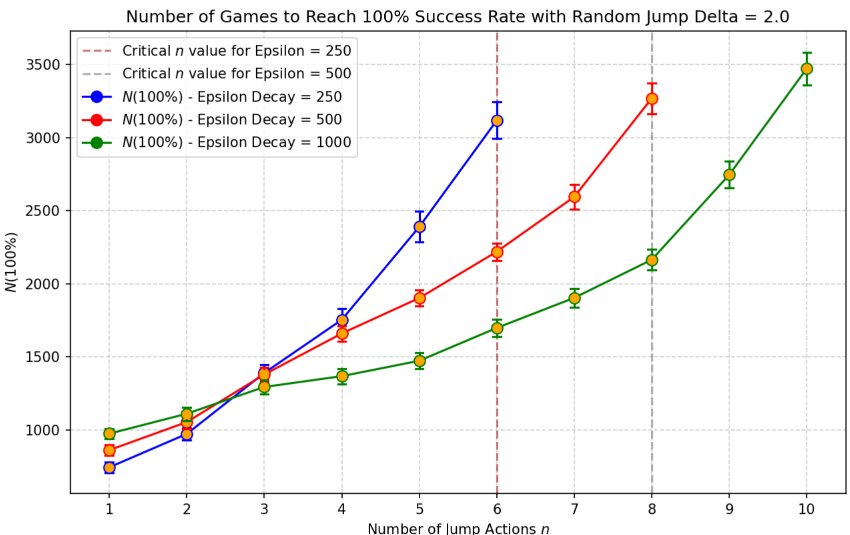


Figure 4.29: Estimation of $N(100\%)$ with varying numbers of jump amounts n, in a scenario with random jump variability $\delta = 2.0$. Confidence in each estimated value is conveyed through a 95% confidence interval from 20 runs.

4.5 Adding New Objects to the Game

In this section, we explore the addition of new objects/obstacles to the game, which until now was solely composed of spike obstacles. We anticipate that the learning complexity will increase with the number of obstacles. However, we will also examine how different types of obstacles impact complexity and investigate the effects of the order in which obstacles are introduced, by considering specific pairs of obstacles. Additionally, we will introduce coins to collect within the game to see how these extra quests affect complexity, noting that collecting a coin is not mandatory for game completion. The objectives of this analysis are to address the following questions:

- How does the learning complexity evolve with the number of obstacles introduced?
- Are there specific obstacles that impact complexity more significantly, and if so, why?
- How does the order in which obstacles are introduced affect the marginal complexity of each new obstacle?
- How does the addition of side quests in the level affect the learning complexity regarding level completion?

4.5.1 Adding Platform and Orb Objects to the Game

In this subsection, we return to a deterministic setting using the default game parameters, but with the addition of new objects and obstacles to enrich gameplay dynamics. Each pair of consecutive obstacles is spaced by $d = 160$ units. In addition to spikes, we incorporate platforms and orbs into the game environment. The variety of obstacles is illustrated in Figure 4.30, which depicts a sorted list of ten obstacle types.

Below is a brief description of each type of obstacle:

1. **Obstacle (1):** A single spike.
2. **Obstacle (2):** A single platform.
3. **Obstacle (3):** Two adjacent spikes.
4. **Obstacle (4):** A sequence of two to five platforms.
5. **Obstacle (5):** Platforms arranged to form stairs, with four platforms per step (up to five steps).
6. **Obstacle (6):** A combination of platforms and spikes that create a ceiling barrier the player must navigate under.
7. **Obstacle (7):** An orb that enables a double jump to clear a trio of adjacent spikes, which cannot be overcome with a regular jump.
8. **Obstacle (8):** A sequence requiring two consecutive orb interactions to leap over six contiguous spikes.

9. **Obstacle (9):** An orb that facilitates a double jump necessary to surpass a deadlock; without this, the player's jump is insufficient.

10. **Obstacle (10):** An orb placed to tempt a double jump that would result in a collision with a wall, serving as a deceptive challenge.

Considering Sequential Obstacle Addition

We first conducted ten experiments, each featuring a distinct level configuration. Configuration i comprises the first i obstacles from our sorted list, where $i \in \{1, 2, \ldots, 10\}$. The training outcomes for all ten configurations are illustrated in Figure 4.31. Notably, these results are presented for $\epsilon_d = 1000$, as training sessions with lower ϵ_d values were unsuccessful. As anticipated, $N(100\%)$ increases with the configuration number, reflecting the escalating complexity of each successive level. The relationship between $N(100\%)$ and the configuration number is further explored in Figure 4.32, which facilitates an easier comparison.

Interestingly, the transitions between configuration pairs such as (1-2), (3-4), and (5-6) demonstrate minimal increases in complexity. This observation suggests that obstacles 2, 4, and 6 introduce negligible additional complexity:

- **Obstacle 2**, which features platforms, does not significantly enhance game complexity as these platforms can be navigated similarly to obstacle 1 spikes and are less lethal than spikes.
- **Obstacle 4** adds minimal complexity despite multiple contiguous platforms; overcoming several platforms positioned together is no more challenging than bypassing a single one.
- **Obstacle 6** appears not to increase complexity significantly. This might be attributed to the game's reward function, which discourages unnecessary jumps. Adhering to this strategic approach allows players to avoid deadly interactions with the ceiling introduced by this obstacle.

Conversely, significant increases in complexity are observed between obstacle configurations (2-3), (4-5), and (6-7), indicating that obstacles 3, 5, and 7 introduce substantial new challenges. Specifically:

- **Obstacle 3** features closely positioned spikes, demanding much more precise timing compared to a single spike. This increase in difficulty stems from the need for more exact jump execution.
- **Obstacle 5** introduces a stair configuration, a novel type of obstacle that alters the player's vertical positioning within the game space. This requires the CNN network to adapt to recognizing objects not only at ground level but also higher up in the image.
- **Obstacle 7** incorporates an orb, introducing a completely new element that necessitates additional training for the agent to effectively understand and use this feature for successful navigation.

We conclude that introducing a variety of obstacle types significantly enhances learning complexity, with a seemingly linear trend between the number of obstacle types and the number of training games for game mastery. However, some obstacles are inherently more challenging and thus demand a greater degree of adaptation and training compared to others.
In this learning process, the complexity required for learning an extra obstacle n depends on all previous $n-1$ introduced obstacles. In order to dive deeper in our exploration, we will next consider relevant pairs of obstacles.

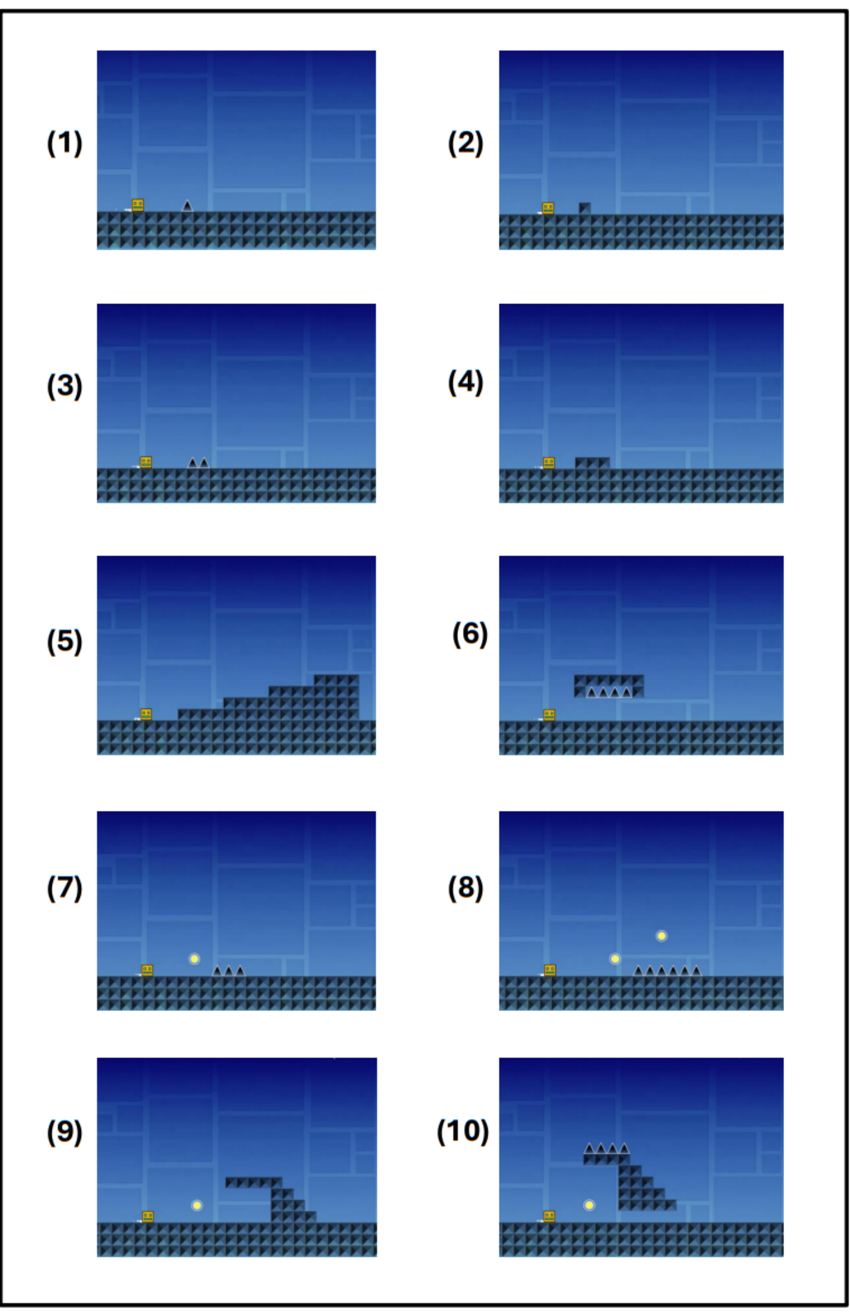


Figure 4.30: Visual representation of all ten obstacles from the sorted list, illustrating diverse game challenges.

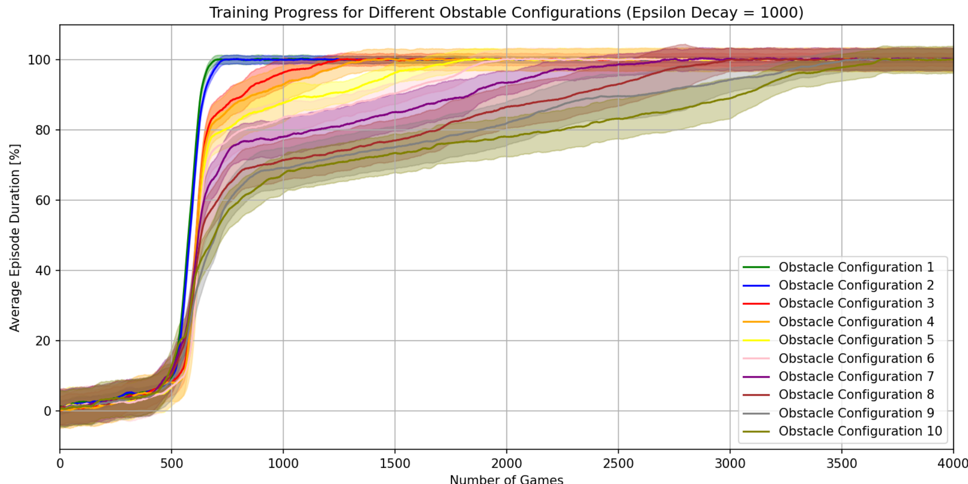


Figure 4.31: Average episode duration for 10 game configurations introducing each new obstacles and objects. Each average curve is supported by a 95% confident interval based on 20 runs.

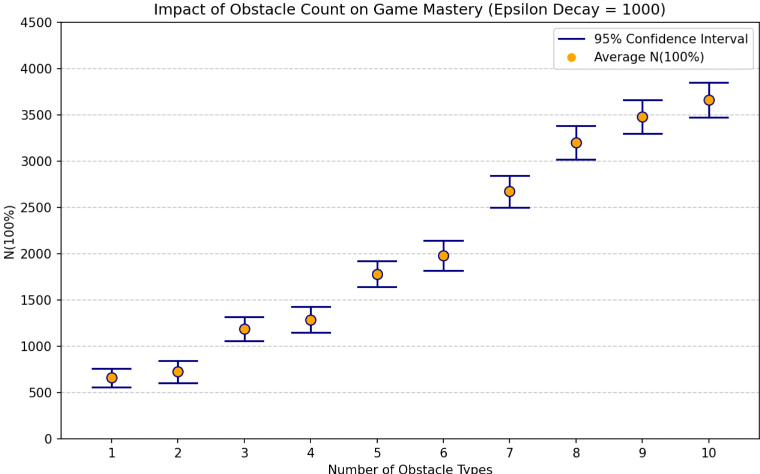


Figure 4.32: Estimated $N(100\%)$ values with respect to the configuration number n. Each data is supported by a 95% confident interval based on 20 runs.

Considering Different Pairs of Obstacles

To better understand the complexity introduced when learning specific pairs of obstacles, this analysis focuses solely on pairs, without the interference from additional obstacles. We consider the same obstacles depicted in Figure 4.30, which lists ten obstacles. We analyze four specific pairs:

1. **Pair (1)–(3)**
2. **Pair (4)–(5)**
3. **Pair (7)–(8)**
4. **Pair (9)–(10)**

Each pair comprises obstacles that are either similar, where one is an extension of the other (the first three pairs), or antagonistic (for the last pair). Indeed for the last pair, one obstacle involves jumping above the structure using an orb (double jump) and the other requires avoiding the orb to pass under the structure.

We conducted the same experiment as in the sequential experiments, setting $\epsilon_d = 1000$ and focusing on all four pairs to assess the complexity of learning each obstacle independently and in combination. The results for all four pairs are depicted in Figure 4.33.

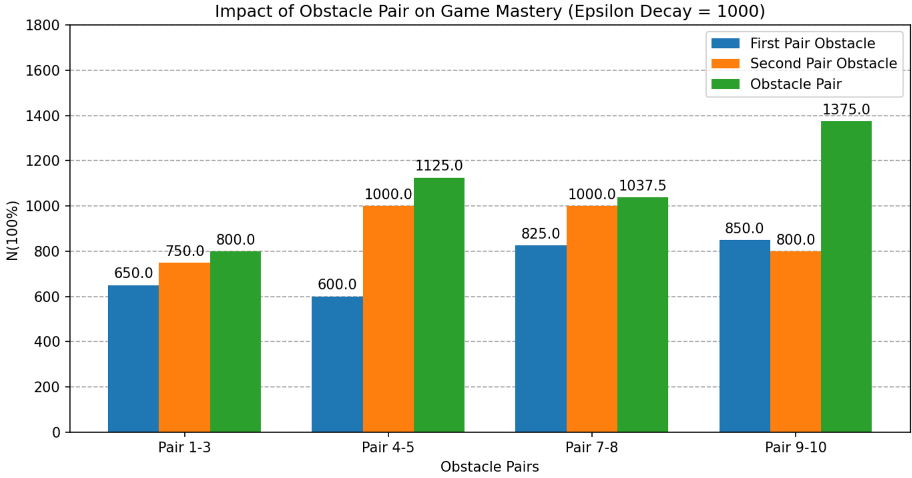


Figure 4.33: Estimated average $N(100\%)$ values for each of the four obstacle pairs, computed based on 20 runs.

For the first three pairs, the complexity of combining both obstacles from the pair is almost equivalent to the complexity of the harder obstacle of the pair (typically the second one). However, in the last pair, while both obstacles can be learned independently with an average of fewer than 850 games, their combination requires an average of 1375 games. This disparity can be attributed to the antagonistic nature of the obstacles: one requires using an orb to jump over the obstacle, while the other demands avoiding the orb to pass under. Unlike humans, who can quickly grasp

these dynamics due to prior experience and cognitive abilities, the AI in this experiment starts from scratch and lacks such pre-existing knowledge.

We observe an interesting trend in Figure 4.32 related to our research questions. As more obstacles are introduced, the increase in complexity becomes more gradual. For instance, the addition of obstacle (10) alongside the first nine results in a marginal increase in the complexity metric $N(100\%)$ by 200 units. In contrast, the previous experiment showed an increase of 525 units when introducing obstacle (10) after obstacle (9), without the preceding eight obstacles. This underscores the importance of the order in which obstacles are introduced. The patterns learned from the initial eight obstacles, such as the strategy taught by obstacle 6—avoiding jumps to survive under a deadly object—may enable the AI to generalize and more effectively master obstacle 10. This capacity for pattern recognition and retention, often taken for granted in humans, is not inherently present in AI in this experimental setup.

4.5.2 Adding Coins to the Game

In this subsection, we explore the effects of incorporating coins into the game configurations discussed previously in the sequential introduction of obstacles. For each of the 10 established configurations, we randomly place three coins in each game level, ensuring that each coin is attainable—either situated on or between obstacles.

The impact of adding coins on the learning complexity is illustrated in Figure 4.34. Our results show a pattern similar to that observed without coins, as documented in Figure 4.32. However, we observe a slight increase in the overall number of training games required to achieve game mastery ($N(100\%)$), with the maximum value reaching approximately 4000 for configuration 10, compared to around 3700 in the same configuration without coins. Despite this increase, the relative complexity between consecutive configurations remains consistent. Ultimately, while the addition of coins introduces a new element to each level, it does not significantly disrupt the learning process in terms of achieving a 100% episode duration.

However, excelling in *Geometry Dash* involves more than just reaching the end of a level; it also requires collecting all available coins. To assess this aspect, we aim to examine the evolution of the average coin ratio per episode during the training process. In our baseline AI model, collecting a coin yields a default reward of $+1$. However, to understand how different reward structures affect learning, we experimented with coin rewards set at $+1$, $+5$, and $+10$.

For this experiment, we focused on configuration 10, which features a diverse range of obstacles. The results for the three coin reward values are presented in terms of average episode duration and average coin ratio, depicted in Figures 4.35 and 4.36, respectively. Interestingly, the evolution of average episode duration shows minimal variation across different coin reward values, though higher rewards slightly decelerate the learning process. This slowdown occurs as the agent prioritizes coin collection over simply surviving and completing the level.

Furthermore, as shown in Figure 4.36, the average coin ratio converges more rapidly to the maximum value of 3 as the coin reward increases. This observation underscores the significant influence of reward settings on learning complexity, particularly in terms of incentivizing coin collection behaviors in the game.

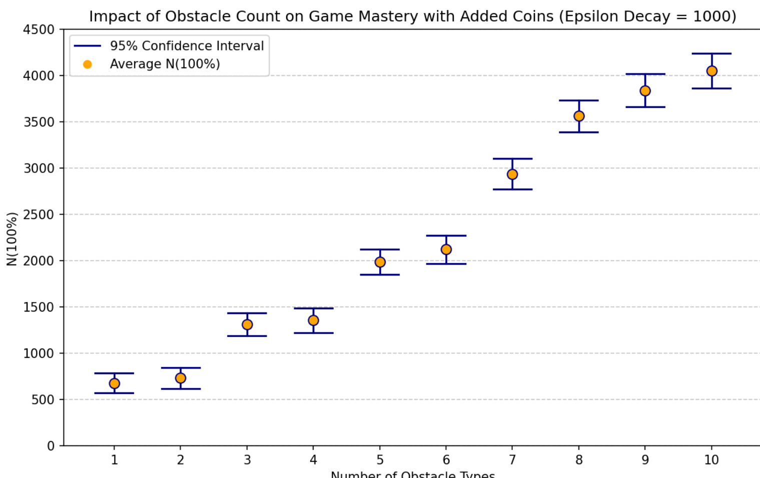


Figure 4.34: Estimated $N(100\%)$ values for each configuration number n, with the addition of three randomly placed coins in each game level. Each estimate is supported by a 95% confidence interval based on 20 runs.

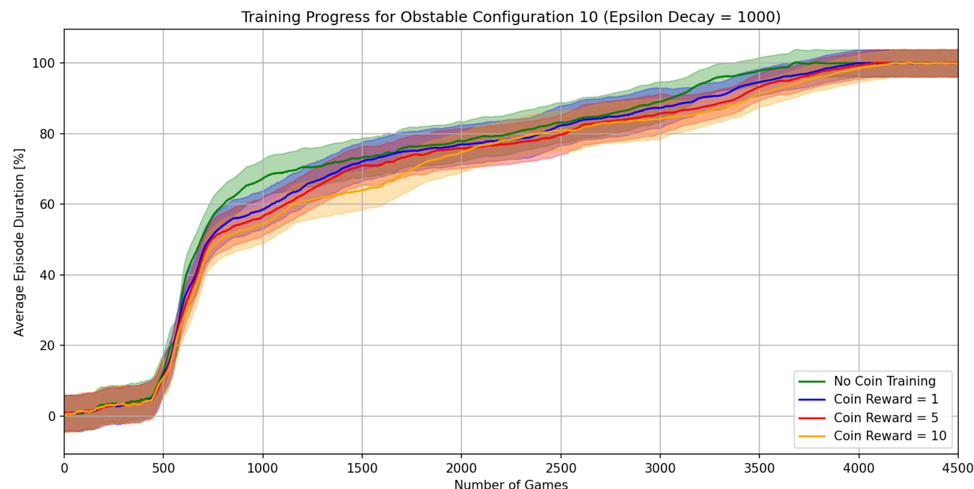


Figure 4.35: Average episode duration for game configuration 10 for three coin reward values. Each average curve is supported by a 95% confident interval based on 20 runs.

In conclusion, the addition of coins to collect in the game does not significantly affect the learning complexity concerning level completion. Collecting coins can be viewed as a side quest, and not collecting them does not impact the skills required for completing the level. However, it is observed that the agent consistently masters the coin collection after playing a sufficient number of games. Furthermore, an increase in the reward associated with coin collection notably enhances the learning related to this task.

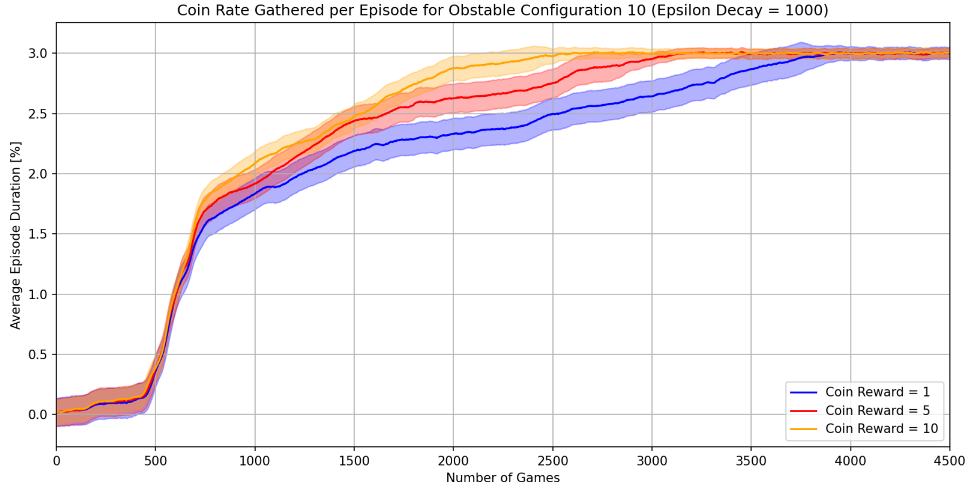


Figure 4.36: Average coin ratio per episode for game configuration 10 for three different coin reward values. Each average curve is supported by a 95% confident interval based on 20 runs.

Chapter 5

Conclusion and Future Work

In this study, we developed a robust *Geometry Dash* AI using deep Q-Learning and Convolutional Neural Networks, akin to the AI designs for Atari games [4]. Additionally, we implemented a customizable *Geometry Dash* environment in Python, based on the repository [15]. Utilizing these implementations, we investigated the evolution of performances and limits of reinforcement learning under game variability.

We first explored the impact on the learning complexity of random disturbances applied to simple levels composed solely of spikes, and investigated the learning limits of the agent when the level becomes impossible due to those disturbances. We then modified the game rules by considering increasing action space cardinality and investigated how the learning complexity evolves with such an increase, and how the learning process behaves when we combine this increase with random disturbances. Finally, we considered adding new obstacles to the game other than spikes, and showed how the types of obstacles and the order in which we introduce them affect the learning complexity.

Our findings generally support the premise that increases in game difficulty—through greater action space cardinality, higher disturbance variance, and a larger number of obstacles—correspond to heightened RL complexity. However, we encountered some counterintuitive results, such as an increase in disturbance variance potentially reducing the critical exploration rate required to achieve game mastery. This observation warrants further investigation and represents an intriguing avenue for future research.

Additionally, we observed that tasks seemingly simple for humans pose greater challenges for AI. For instance, AI learns quite easily two antagonistic obstacles separately. We mean by antagonistic similar obstacles requiring totally different actions to be overcome. However, the AI struggles to learn the combination of both obstacles. This difficulty arises because humans can draw upon existing patterns and experiences in mind to quickly adapt to new tasks, such as playing *Geometry Dash*, whereas AI starts from scratch and must learn these patterns during training.

This study is innovative and among the few to analyze game variability and its impact on RL complexity, particularly employing deep Q-learning in the *Geometry Dash* game. The presented results hopefully will contribute to the theorization of reinforcement learning complexity, which is influenced by numerous factors such as the RL algorithm, network architecture and parameters, exploration phase, reward function, and game environment. The experimental results and observed

trends demonstrate the practical evolutions of this complexity. Nevertheless, the major challenge in theorizing lies in the ability to model game dynamics and obstacles within a framework that can integrate chosen exploration and reward functions, aiming to extend the concepts currently theorized by multi-armed bandits [9] for static decision-making environments.

Our results do not contribute directly to this theorization; however, they increase our understanding of reinforcement learning complexity and open doors to further investigations, possibly exploring other video games. Though, within the confines of *Geometry Dash*, many avenues remain unexplored. A promising direction would involve experimenting with alternative reinforcement learning algorithms known for their stability in learning processes. This approach could address the instabilities encountered with deep Q-learning, especially when disturbance variance is significant.

For instance, the Proximal Policy Optimization (PPO) algorithm [7] has attracted considerable attention for its stability and ease of tuning. Unlike deep Q-learning, which is optimized for discrete action spaces, PPO is specifically designed for environments with continuous action spaces. This makes it potentially more suitable for variants of *Geometry Dash* that feature a continuous spectrum of jump amounts. Comparing the learning performance of deep Q-learning in an expanded discrete action space with that of PPO in environments with increasing ranges of jump amounts could provide valuable insights into how each algorithm performs as the action space size increases, in both discrete and continuous setups.

Another interesting avenue for research could involve expanding the variety of obstacles used in experiments, or even experimenting with randomly designed obstacles, to test the AI's ability to generalize across different types of challenges.

Additionally, combining disturbances to several game parameters simultaneously could reveal how the agent's learning adapts and identify potential limits, even if the level remains technically playable.

Lastly, one particularly relevant idea for convolutional neural network (CNN) architectures involves introducing image noise directly into the game environment or manipulating background patterns and colors to assess their impact on learning complexity. This approach could constitute a focused study on deep learning, building upon existing research in image denoising processes using CNNs [20].

Exploring these directions promises not only to deepen our understanding of the factors that influence reinforcement learning complexity under video game variability but also to potentially lead to novel approaches that enhance learning performance and stability.

Bibliography

[1] Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2018.

[2] Maxim Lapan. *Deep Reinforcement Learning Hands-On*. Packt Publishing Ltd, 2018.

[3] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529.7587 (2016), pp. 484–489.

[4] Volodymyr Mnih et al. "Playing Atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602* (2013).

[5] Max Schwarzer et al. "Bigger, Better, Faster: Human-level Atari with human-level efficiency". In: *Preprint posted on ArXiv* (2021). `https://arxiv.org/abs/2102.06177`.

[6] Christopher Berner et al. "Dota 2 with Large Scale Deep Reinforcement Learning". In: *arXiv preprint arXiv:1912.06680* (2019).

[7] John Schulman et al. "Proximal Policy Optimization Algorithms". In: *arXiv preprint arXiv:1707.06347* (2017).

[8] Aleksandrs Slivkins. "Dynamic Multi-Armed Bandits and Applications in Online Advertising". In: *Review of Network Economics* 9.1 (2010), pp. 1–49.

[9] Clayton Scott. *Multi-Armed Bandits: Theory and Applications to Online Learning in Networks*. Springer, 2019.

[10] Ted Li and Sean Rafferty. *Playing Geometry Dash with Convolutional Neural Networks*. Course Project for CS231N (and CS231A for Sean Rafferty). Available: `https://cs231n.stanford.edu/reports/2017/pdfs/605.pdf`. 2017.

[11] Richard Bellman. "A Mathematical Theory of Communication". In: *Bell System Technical Journal* 38.4 (1957), pp. 623–656.

[12] Richard S. Sutton et al. "Policy Gradient Methods for Reinforcement Learning with Function Approximation". In: *Advances in neural information processing systems*. 2000, pp. 1057–1063.

[13] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323.6088 (1986), pp. 533–536.

[14] Yann LeCun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[15] Yonah Aviv. *Pydash*. `https://github.com/y330/Pydash`. Year.

[16] Geometry Dash Wiki. *Geometry Dash Wiki*. Accessed: June 1, 2024. URL: https://geometry-dash.fandom.com/wiki/Geometry_Dash_Wiki.

[17] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), pp. 529–533.

[18] David Rolnick et al. "Experience Replay for Continual Learning". In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach et al. Vol. 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper_files/paper/2019/file/fa7cdfad1a5aaf8370ebeda47a1ff1c3-Paper.pdf.

[19] John A. Rice. *Mathematical Statistics and Data Analysis*. 3rd. See Chapter 5 for a detailed discussion of the Central Limit Theorem. Duxbury Press, 2006.

[20] Ademola E Ilesanmi and Taiwo O Ilesanmi. "Methods for image denoising using convolutional neural network: a review". In: *Journal of Ambient Intelligence and Humanized Computing* 12 (2021), pp. 1–23.

Appendix A

Baseline Training Level Examples

In this appendix, we present ten excerpts from our baseline training levels, which were instrumental in designing a robust AI capable of playing the game. These examples offer readers insights into the game levels that the AI could master following its design. The ten game screens are depicted in Figure A.1.

We emphasize the diversity of obstacles, which range from various combinations of spikes and platforms to stairs, double jump orbs, and strategies to avoid deadly obstacles such as triple spikes or collecting coins positioned on upper platforms only reachable with an orb. Additionally, some obstacles feature small gaps, challenging the player to navigate through without jumping to avoid collisions.

While these screens do not encompass the full diversity of the game, we hope this appendix provides readers with an understanding of the level difficulty and complexity encountered during AI training.

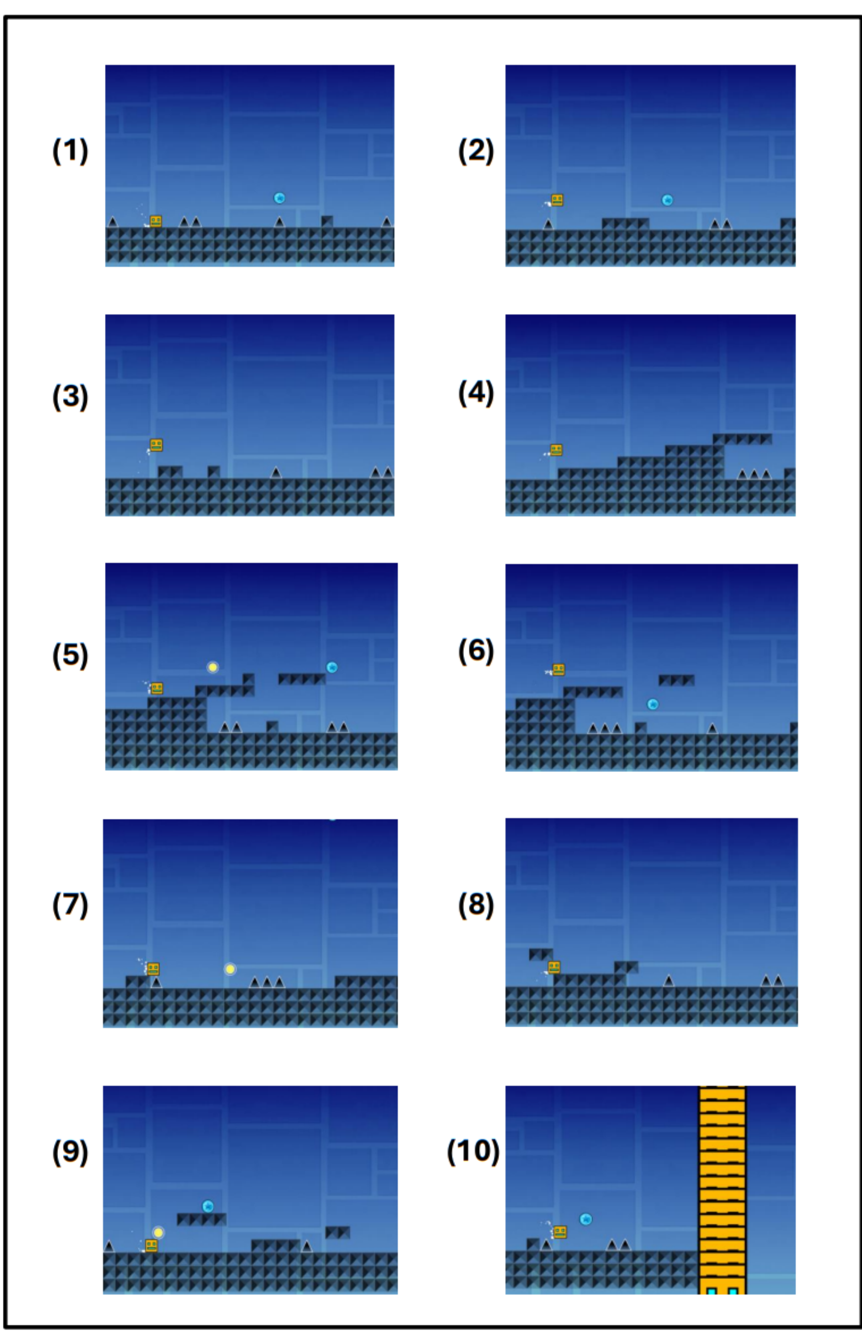
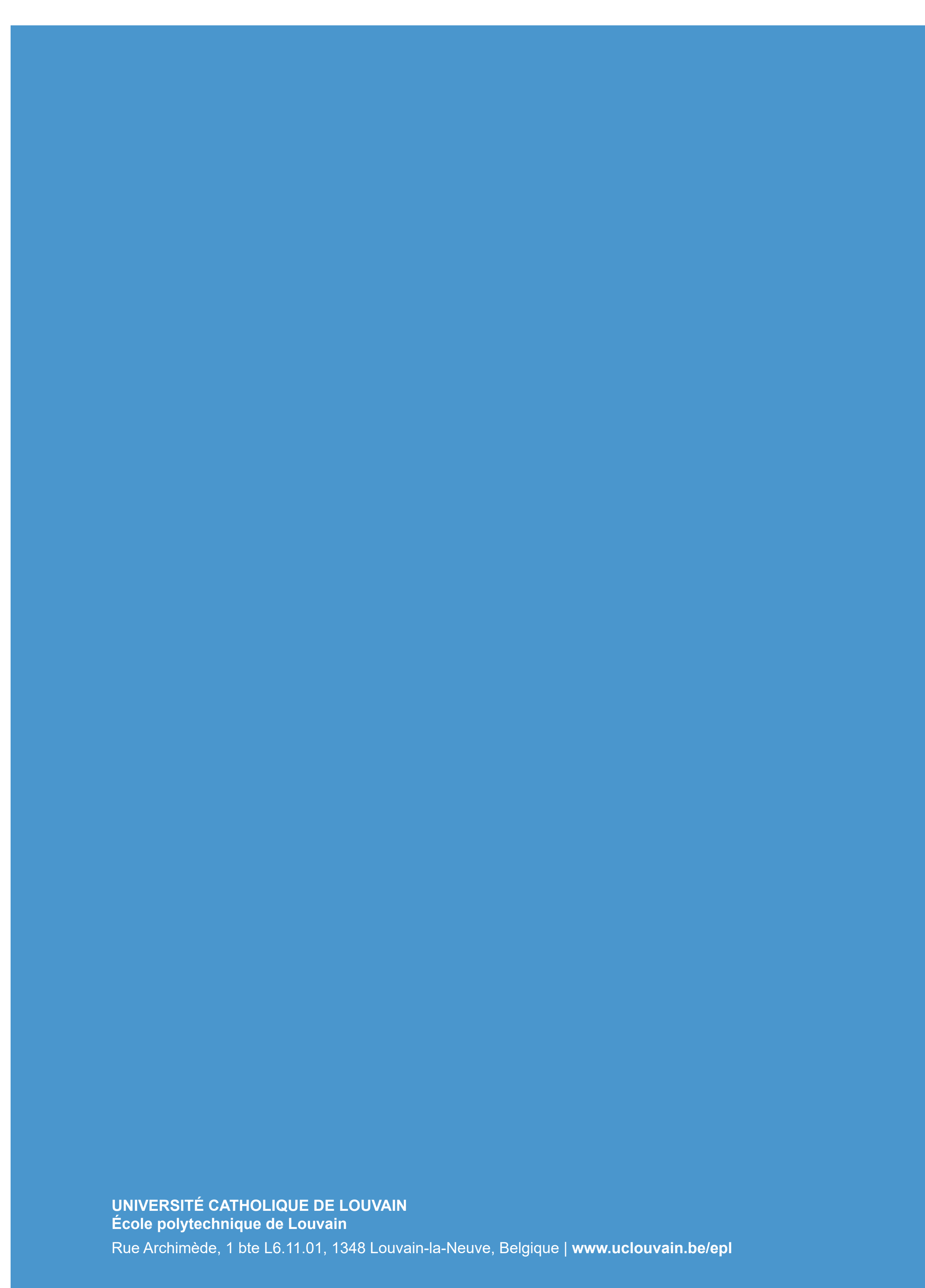


Figure A.1: Example of ten game screens from the baseline training levels used for training the AI.



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