Ambient Life - Exploring Stochastic DRL-Powered NPC Behavior in Gaming: A Player-Centric Investigation

John Tember

August 2024

Abstract

This thesis investigates the impact of deep reinforcement learning (DRL) on non-playable character (NPC) behavior in video games, with a focus on how stochastic elements influence player experience. By developing a simulated environment and training NPCs using Proximal Policy Optimization (PPO), the study explores how unpredictable NPC behaviors affect player engagement and interaction strategies. Through player testing and thematic analysis, key aspects such as influence, relationship-building, and independent behaviors were identified as significant factors in enhancing immersion and emotional connection. The findings highlight the potential of DRL to create more dynamic and realistic game environments, offering valuable insights for future game design and development.

Contents

1 Introduction

This section provides the background and motivations for this thesis, focusing on the use of deep reinforcement learning (DRL) in video game development. It outlines the research objectives, emphasizing the importance of understanding how stochastic DRL-based NPC behaviors influence player experiences. Finally, the research questions guiding this study are introduced.

1.1 Background

Artificial intelligence (AI) has long been a cornerstone of video game development. Among the various subsets of AI, machine learning (ML) is gaining significant attention for its potential to streamline development processes and enhance in-game content. In the realm of non-playable characters (NPCs), ML has the potential to complexify behaviors while reducing developers' workloads. This approach could lower development costs and time, and provide players with richer and more dynamic interactions with NPCs—qualities highly valued in the gaming industry.

The integration of ML in the development of NPC behavior marks a significant departure from traditional methods. Many ML techniques incorporate stochastic elements, which, when applied to NPC behavior, introduce dynamic and less predictable interactions. In contrast, traditional methods, such as finite state machines (FSM) [27, 7], often rely on deterministic systems, operating on strict rules to ensure consistent behaviors. While it is possible to add stochastic elements to FSMs, this has not been common practice.

Repetitive and predictable interactions are commonly associated with negative experiences, such as boredom, lack of motivation, and a sense of purposelessness [15]. The dynamic nature of ML-driven NPCs can mitigate these issues by introducing stochastic behavior, providing a more engaging and stimulating gaming experience.

Among various ML techniques, reinforcement learning (RL) stands out for its ability to allow computers to learn behaviors through trial and error without explicit instructions. This method, which reduces the need for developers to manually script actions [8], enables NPCs to learn and potentially adapt their behaviors in real-time based on player interactions. However, scalability issues often confine these successes to simpler problems. The rise of deep learning (DL) has offered novel strategies to address these challenges, leading to the specialized field of DRL. DRL extends RL applications to complex decision-making scenarios previously deemed unmanageable. Consequently, DRL can further enhance the dynamic nature of NPC interactions, while simultaneously reducing the cost of development.

However, these advancements come with complexities. The predictability offered by traditional methods is crucial for game aspects tied to specific events. When events must unfold in a precise manner, stochastic DRL methods may not suffice, due to their inherent unpredictability. This might also be one of the reasons why these methods are rarely used today. For ambient life—NPCs

that populate the background—DRL can introduce realism and dynamism while reducing the development workload. Since ambient NPCs are meant to enhance the game's realism without the need for strict control, allowing them more freedom in their behaviors might be beneficial.

The integration of DRL in video games, particularly for creating autonomous, adaptive ambient life, presents a promising frontier for complex behaviors that enhance realism and immersion at lower development costs. While the potential benefits of DRL in delivering more dynamic and engaging gameplay are acknowledged, the actual impact on players remains relatively unexplored. As NPCs begin to exhibit behaviors not explicitly programmed by developers, it is crucial to understand how players perceive and interact with these autonomous characters. Balancing the unpredictability and dynamism of DRL-driven NPCs with the need for a coherent and enjoyable player experience poses a significant challenge. This evolving field calls for deeper investigation into how these advancements influence the core of gaming: the player's experience. This thesis aims to address this gap by examining the effects of stochastic DRL-driven NPC behaviors on player experience.

1.2 The Rise of DRL

Creating autonomous agents capable of learning and adapting to their environments is a key goal in RL [26]. RL has achieved notable successes in optimizing human-computer dialogue systems and playing board games [9, 16, 24, 28]. However, scalability issues often confine these successes to simpler problems, limiting agents' ability to handle more complex scenarios. These challenges, rooted in memory management complexity, computational demands, and the need for extensive datasets, are widely recognized in research [25].

One significant issue is the curse of dimensionality, which refers to the exponential increase in computational complexity and the amount of data needed to model and analyze high-dimensional spaces. As the number of dimensions increases, the volume of the space grows rapidly, making the available data sparse. This sparsity complicates the algorithms' ability to find patterns and make accurate predictions [3]. This problem is particularly significant in video games, where NPCs inhabit large, complex virtual worlds that are challenging to represent with simple functions, requiring sophisticated data structures instead.

The rise of deep learning (DL) has offered novel strategies to address these challenges. Deep neural networks (DNNs) are key components in this context, capable of approximating complex functions and identifying intricate patterns. While traditional neural networks (NNs) typically consist of one hidden layer, DNNs are characterized by their use of multiple hidden layers, allowing them to model more complex relationships in data. DL has significantly enhanced various ML domains, including object detection, speech recognition, and language translation [12]. The critical advantage of DNNs lies in their ability to distill and represent high-dimensional data within compact, low-dimensional feature spaces. By embedding inductive biases, particularly hierarchical representations, into neural architectures, researchers have effectively countered the curse of dimensionality [2].

These advancements have propelled RL forward, giving rise to the specialized field of DRL. DRL extends RL applications to complex decision-making scenarios previously deemed unmanageable due to their high-dimensional nature. Notably, DRL has achieved remarkable results, such as developing algorithms that master various Atari 2600 video games at superhuman levels using only pixel-based visual input [14]. This breakthrough demonstrated the potential for training agents on unprocessed, high-dimensional sensory data using reward signals alone.

Another significant achievement is AlphaGo, a hybrid DRL system that defeated a human Go champion [23]. AlphaGo moved beyond traditional rule-based chess AI by combining DNN-based supervised learning and DRL with heuristic search strategies.

1.3 Aim

This thesis explores the impact of stochastic DRL-based NPC behavior on player experiences, specifically focusing on aspects that players tend to notice and engage with. To achieve this, the study was structured around several key steps:

1. Development of a simulated environment that mirrors a typical video game setting.
2. Creation of NPCs designed to represent ambient life in video games.
3. Training of these NPCs to exhibit stochastic behavior using DRL-techniques.
4. Assessment of the effects of the stochastic DRL-based behavior on player interaction.
5. Conducting quantitative interviews to gather players' insights.

1.4 Research questions

The following research questions were designed to capture the goal of this thesis:

1. How does stochastic DRL-based behavior in ambient NPCs influence player experience in video games?
   (a) Which aspects of the behavior are recognized by players?
   (b) To what extent does the behavior affect players' interaction strategies?
2. How do the results of this study correlate with established literature on player experience?

2 Theoretical Background

This section introduces the theoretical background for the thesis, focusing on the theory that related to the specific RL-algorithm utilized in this study. Additionally, it discusses current literature on player experience and provide insight into the analysis methodology employed for the analysis of the results.

2.1 Reinforcement Learning

The primary objective of RL is for an agent to learn a policy π. A policy is a method of mapping states s to actions a. The state encapsulates the agent's observations of the environment, while the actions denote the agent's responses regarding these observations. Essentially, the goal is for the agent to learn a policy that maps observations to appropriate behaviors; this process is called policy optimization.

RL works through trial-and-error learning. An agent interacts with an environment and receives rewards based on its actions. The agent uses these rewards to adjust its actions with the goal of maximizing the cumulative reward it receives over time, which is the sum of rewards given at each state transition.

At each time step t, the agent observes its environment's state, represented as s_t, and responds by taking an action a_t. This action causes both the agent and the environment to transition to a new state, s_{t+1}, which reflects the consequences of the action. For instance, if the agent consumes resources in state s_t, those resources will no longer be available in state s_{t+1}. When modeled properly, the state s_t contains all the necessary information to make appropriate decisions. Whether the transition is good or bad is determined by a reward function.

With each shift to a new state, the environment provides a reward r_{t+1} to the agent as feedback. The agent strives to learn a policy π that optimizes its expected rewards in the long run. This policy advises the agent on the action to take based on the current state, aiming to achieve the greatest expected rewards. The core challenge in RL is for the agent to deduce the outcomes of its actions through trial and error, leveraging each interaction as an opportunity to enhance its decision-making process [8].

2.1.1 Markov Decision Chain

A key aspect of RL is the Markov property, which posits that the next state transition depends only on the current state and action, not on any previous states. This principle implies that future decisions should be based solely on the present state. The Markov Decision Processes (MDP) consist of:

- A set of states S and a probability distribution for the initial state $p(s_0)$.
- A set of actions A.
- Transition dynamics $T(s_{t+1}|s_t, a_t)$ that determine the probability of moving to the next state given the current state and action at time t.
- A reward function $R(s_t, a_t, s_{t+1})$ that assigns immediate rewards for transitions between states due to actions.
- A discount factor $\gamma \in [0, 1]$, with values closer to 0 emphasizing the importance of immediate rewards over future ones.

The strategy, or policy π, maps each state to a probability distribution of actions: $\pi : S \to p(A = a|S)$. In episodic MDPs, where states reset after an episode of length T, the sequence of states, actions, and rewards creates a path or rollout according to the policy. The sum of rewards collected through a policy rollout, called the return, is $R = \sum_{t=0}^{T-1} \gamma^t r_{t+1}$. RL aims to find the optimal policy that maximizes the expected returns across all states:

$$\pi^* = \arg\max_{\pi} \mathbb{E}[R \mid \pi]. \tag{1}$$

In contrast, non-episodic MDPs, which are not used in this study, have an infinite T. In these scenarios, a discount factor $\gamma < 1$ ensures that the total reward does not reach infinity. Although focusing on entire paths is impractical in non-episodic MDPs, strategies based on a sequence of transitions are still applicable [8].

2.1.2 RL Algorithms

There are two main approaches to solving RL problems: methods based on value functions and methods based on policy search. There is also a hybrid actor-critic approach that employs both value functions and policy search. In this study, the actor-critic approach was used. To provide some context for comparison, this section provides a brief introduction to all of them.

Value Functions Value function methods are based on estimating the return (expected reward) of being in a given state. The state-value function $V^\pi(s)$ is the expected return when starting in state s and following π subsequently:

$$V^\pi(\mathbf{s}) = \mathbb{E}[R|\mathbf{s}, \pi]. \tag{2}$$

The optimal policy, π^*, has a corresponding state-value function $V^*(s)$, and vice versa; the optimal state-value function can be defined as:

$$V^*(\mathbf{s}) = \max_\pi V^\pi(\mathbf{s}) \quad \forall \mathbf{s} \in \mathcal{S}. \tag{3}$$

If $V^*(s)$ is available, the optimal policy could be retrieved by choosing among all actions available at s_t and picking the action a that maximizes $\mathbb{E}_{s_{t+1} \sim T(s_{t+1}|s_t,a)}[V^*(s_{t+1})]$.

In the RL setting, the transition dynamics T are unavailable. Therefore, we construct another function, the state-action value or quality function $Q^\pi(s, a)$, which is similar to V^π, except that the initial action a is provided and π is only followed from the succeeding state onward:

$$Q^\pi(\mathbf{s}, \mathbf{a}) = \mathbb{E}[R|\mathbf{s}, \mathbf{a}, \pi]. \tag{4}$$

The best policy, given $Q^\pi(s, a)$, can be found by choosing a greedily at every state: $\text{argmax}_a Q^\pi(s, a)$. Under this policy, we can also define $V^\pi(s)$ by maximizing $Q^\pi(s, a) : V^\pi(s) = \max_a Q^\pi(s, a)$.

Policy Search. Policy search methods in RL focus on finding the best policy π^*, which dictates the actions an agent should take without the need for a value function model. These methods utilize a policy π_θ parameterized by θ, which is adjusted to maximize the expected return, $\mathbb{E}[R|\theta]$. This optimization can be achieved through either gradient-based or gradient-free techniques. While gradient-free methods, such as evolutionary strategies [5, 11], are useful for exploring low-dimensional parameter spaces and can optimize even nondifferentiable policies, gradient-based training [6, 13, 20, 21, 29, 30] is generally preferred for its efficiency, especially when dealing with policies that have many parameters.

In practice, policy search often involves generating a stochastic policy by defining parameters for probability distributions—Gaussian distributions for continuous actions and multinomial distributions for discrete actions—allowing for direct action sampling. Evolutionary algorithms, a type of gradient-free method, have shown success by evaluating a population of agents, making them suitable for optimizing arbitrary models and encouraging exploration in parameter space [5, 11]. Despite their computational demand, evolutionary algorithms have been pivotal in training large networks, even enabling the first NN to learn an RL task directly from high-dimensional visual inputs [11]. Recent advancements have further solidified the relevance of these methods in RL, offering scalable solutions beyond traditional gradient-based approaches [18].

Policy Gradients. Gradients can provide a strong learning signal as to how to improve a parameterized policy. However, to compute the expected return, we need to average over plausible trajectories induced by the current policy parameterization. This averaging requires either deterministic approximations (e.g., linearization) or stochastic approximations via sampling [4]. Deterministic approximations can only be applied in a model-based setting where a model of the underlying transition dynamics is available. In the more common model-free RL setting, a Monte Carlo estimate of the expected return is determined. For gradient-based learning, this Monte Carlo approximation poses a challenge since gradients cannot pass through these samples of a stochastic function. Therefore, we turn to an estimator of the gradient, known in RL as the REINFORCE rule [30]. Intuitively, gradient ascent using the estimator increases the log probability of the sampled action, weighted by the return. More formally, the REINFORCE rule can be used to compute the gradient of an expectation over a function f of a random variable X with respect to parameters θ:

$$\nabla_\theta \mathbb{E}_X[f(X;\theta)] = \mathbb{E}_X[f(X;\theta)\nabla_\theta \log p(X)]. \tag{7}$$

As this computation relies on the empirical return of a trajectory, the resulting gradients possess a high variance. By introducing unbiased estimates that are less noisy, it is possible to reduce the variance. The general methodology for performing this is to subtract a baseline, which means weighting updates by an advantage rather than the pure return. The simplest baseline is the average return taken over several episodes [30], but there are many more options available [21].

Actor-Critic Methods. Actor-Critic methods represent a sophisticated category of RL algorithms that synergize value functions with a policy's explicit representation. These algorithms feature a dual-component system where the "actor" (policy) adapts based on feedback from the "critic" (value function), as depicted in a figure. This dynamic allows for a balance between minimizing the variance inherent in policy gradients and the bias introduced by employing value function methods [10, 21].

The hallmark of Actor-Critic methods lies in their utilization of the value function as a benchmark for policy gradient adjustments. Distinguishing them from other baseline approaches is their incorporation of a learned value function, positioning Actor-Critic methods as a specialized branch of policy gradient methodologies.

Backpropagation Through Stochastic Functions. Central to RL is backpropagation, a fundamental mechanism for training NNs. Backpropagation allows the network to update its weights based on the error between the predicted and actual outcomes, effectively minimizing the loss function.

One pivotal application of backpropagation in RL is through the REINFORCE rule [30]. The REINFORCE rule enables NNs to develop stochastic policies tailored to specific tasks. For example, in image analysis tasks like object tracking [19] or annotation [31], the network can focus on different regions of the image based on a stochastic policy. In these scenarios, the network samples a small segment of the image to analyze, optimizing computational efficiency by reducing the amount of data processed at each step. This technique is known as hard attention within the deep learning field, where RL is used to make discrete, stochastic decisions about which parts of the input to focus on. It represents an innovative application of basic policy search techniques, extending their utility beyond traditional RL tasks.

2.2 Proximal Policy Optimization

Proximal Policy Optimization (PPO) is a DRL technique designed to address some of the key challenges in training policies for complex environments. At its core, PPO aims to update policies in a way that is both efficient and stable, avoiding the pitfalls of excessively large policy updates that can lead to performance degradation. This is achieved through a clever balance between exploration and exploitation, ensuring that the policy improves steadily without taking overly risky or conservative steps.

The innovation behind PPO lies in its approach to adjusting the policy. Unlike traditional methods that make large, risky updates based on each batch of data, PPO limits the size of policy updates. It does this by optimizing a specially designed objective function that penalizes changes to the policy that move it too far from the previous policy. This mechanism ensures that the policy evolves in a more controlled and gradual manner.

One of the key features of PPO is its simplicity and ease of implementation compared to other advanced RL algorithms. It does not require complex architecture or a sophisticated understanding of the underlying environment model. Instead, PPO can be implemented with standard NN components and optimized using general optimization techniques.

PPO operates by collecting a set of trajectories through interaction with the environment, using the current policy. It then uses these trajectories to estimate the advantage and update the policy. The objective function it optimizes encourages the new policy to achieve higher rewards while keeping the policy update within a predefined range. This is crucial for maintaining the stability of the learning process and preventing the policy from oscillating or diverging.

A significant advantage of PPO is its robustness across a wide range of environments, from classic control tasks to more complex scenarios such as 3D locomotion and robotics. This versatility makes PPO a popular choice among researchers and practitioners looking for a reliable and effective method for training policies in DRL [22].

PPO)s particularly relevant for training non-player characters (NPCs) in video games due to its efficiency, stability, and ease of implementation. These attributes make PPO an ideal choice for the dynamic and complex environments found in video games, where NPCs must exhibit adaptive, realistic behavior. PPO's ability to balance exploration and exploitation ensures that NPCs can learn varied behaviors without the risk of significant policy degradation, which is crucial for maintaining an engaging gaming experience. Moreover, the simplicity of PPO allows for rapid iteration and testing, essential in game development where time and resources are often limited. The robustness of PPO across different scenarios means it can be applied to a wide range of game genres, from strategy and simulation to action and adventure, making it a versatile tool for enhancing NPC behavior in video games.

3 Methodology - Development

This section outlines the developmental phase of the methodology, detailing the development of the *Environment*, which served as the test-bed for this study.

The *Environment* was composed of three components: the *Scene*, the *Human*, and the *Agents*. Any future references to the *Environment* refer to all three components collectively. For readers new to RL, it's important to understand that the *Scene* alone is not the entire *Environment*.

For clarity, terms in *italic*, such as "the *Environment*", refer to specific components developed during this study. When referring to general concepts, such as "in game design, a player navigates through various environments", terms are not italicized.

References to "players" specifically denote those who took part in player testing during the evaluation phase of the methodology, which is covered in Section 4.

References to "the training" specifically denote the training of the *Agents*, which is covered in Section 3.3.4

3.1 Scene

The *Scene*, illustrated in Figure 1, served as the virtual location of the *Environment*. It was specifically designed to be appropriate for a video game setting, in order to eliminate experimental confounds. Experimental confounds are variables that interfere with the validity of the study by introducing bias or distraction. For instance, a poorly designed setting could divert players attention away from the objective, thereby affecting the results.

The *Scene* was also designed as a space that the *Agents* could explore and learn from, independently of the *Human*. This independence was crucial for the study, which focused on ambient life. It ensured that the *Agents* could learn behaviors that were not solely centered around the *Human* but were dynamic and realistic in relation to the entire *Environment*. If the *Scene* had been designed as a flat, empty landscape, the *Agents* would have been compelled to focus only on the *Human* due to the lack of other stimuli. Consequently, the *Agents* would not have represented ambient life accurately.

Designed as a swamp-like forest, the *Scene* consisted of various elements such as terrain and vegetation. These elements included obstacles like trees, rocks, hills and dips, which offered potential hiding spots and navigational challenges. Mushrooms and pools, illustrated in Figure 2, provided resources the *Agents* could learn to utilize to regulate their *Needs*, such as for food and water. The concept of *Needs* are introduced in Section 3.3.3.

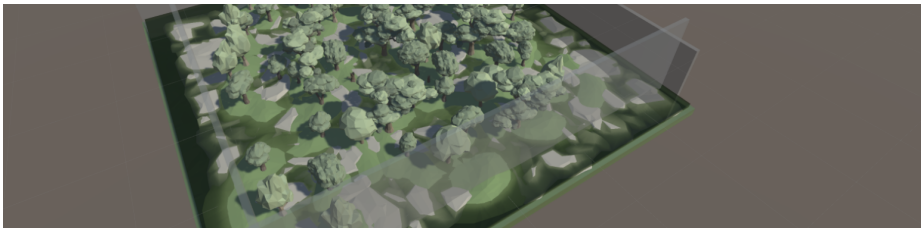

Figure 1: The *Scene*, designed as a swamp-like forest, included trees, rocks, hills and dips, water, and mushrooms.

Figure 2: Mushrooms that grew in the *Scene*, serving as a source of food for the *Agents*. Beside them, a pool of water, from which the *Agents* could drink.

3.2 Human

When training an agent with DRL, or RL in general, the simulation is often accelerated to save time. Consequently, it is not feasible to train an agent based on real-time player interactions, as a player cannot accurately participate in an accelerated simulation. Additionally, the simulation is often replicated into multiple instances so that the agent can learn from them simultaneously, to further same time. This approach would require the player to participate in several simulations at once, which also is not feasible. To counter this problem, the *Human*, illustrated in Figure 3, was developed.

The *Human* was developed to simulate the presence of the players during training of the *Agents* and to serve as the character that the players could control during the player testing. Therefore, two control settings were developed: manual, via keyboard and mouse, and autonomous, via FSM.

The autonomous control was designed to avoid overfitting, which occurs when an agent learns to perform very well in a specific environment but fails to generalize to different or slightly varied environments. If the *Human* performed only repetitive behaviors during training, the *Agents* would learn that it would never do anything else. Consequently, during play-testing, if players did not perform these specific behaviors, the *Agents* would not understand what to do, and potentially behave in ways that would make players uncomfortable.

The *Reward Function* that guided each *Agent's* learning process included one variable directly affected by the *Human*: the distance between the *Human* and each *Agent*. The *Agents* needed to experience variations in this distance to understand its relationship to the rewards they received. To avoid overfitting, the *Human* needed to move around in non-repetitive patterns. Consequently, the FSM was given two states: Follow and Wander. In the Follow state, the *Human* was randomly assigned an *Agent* to follow. In the Wander state, it consistently moved towards random locations in the *Scene*.

The *Reward Function* also included the *Agents* opinions of the *Human*. During training, these opinions were randomly assigned, and consequently, the FSM did not have to include any behavior that influenced them. However, during manual control, the opinions could be influenced through two tools provided the *Human*: a stick and a tomato. Players could use these tools to interact with the *Agents*; the stick could be used to slap them, while the tomato could be used to feed them. Additionally, players could move around freely in the *Scene*.

Figure 3: The *Human* could be controlled either manually or automatically.

3.3 Agents

The *Agents* were designed as raccoons, illustrated in Figure 5. Their purpose was to learn a behavior that the players could interact with. The behavior was based on a system of three components: the *Policy*, the *Translator*, and the *Memory*. The system, illustrated in Figure 4, operated as follows: each *Agent* observed the *Environment* and passed its observations to the *Policy*, which generated actions based on these observations. The actions were then passed to the *Translator*, which processed them into *Instructions*. These *Instructions* were then stored in the *Memory*. The *Agent* continuously read its *Instructions* from its *Memory* to execute its behavior.

Each *Agent* observed and operated independently, but used the same *Policy* and *Translator*. They each had their own *Memory* and produced their own set of *Instructions* based on their independent observations. This approach ensured that each *Agent* could operate independently but base their operation in the behavior, as they all followed the same underlying system.

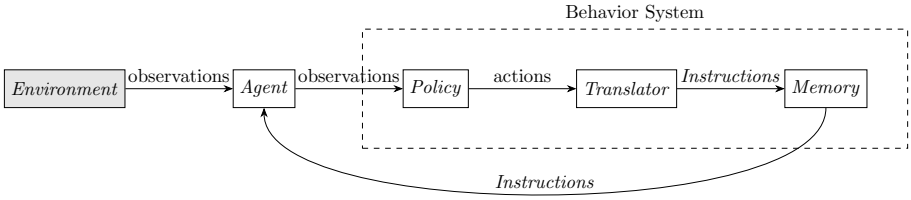

Figure 4: Flow of observations from the *Environment* to *Instructions* in the *Agents* behavior system.

In typical scenarios, actions produced by a policy are directly used by an agent to execute a behavior. However, the *Translator* took the actions produced by the *Policy* and converted them into *Instructions*. *Instructions* were essentially the same as things as actions, but renamed to distinguish their role in the system. To clarify, actions were produced by the *Policy*, and *Instructions* were produced by the *Translator*. *Instructions* were thereby the final actions used by the *Agents* to execute their behavior. This approach was used to add a layer of complexity and control to the system.

Figure 5: The agents that inhabited the forest.

3.3.1 Policy

A policy is a method of mapping observations to actions, enabling an agent to behave optimally in its environment. For instance, if an agent observes food and has a high level of hunger, a good policy should map these observations to actions that make the agent to eat. To clarify, all pieces of information an agent experiences are considered observations; thus, in this situation, the level of hunger is also an observation. An optimal policy ensures that an agent behaves in a way that maximizes the reward it receives from the reward function. Initially, the agent does not know the optimal policy; it needs to be learned.

In practice, learning an optimal policy is extremely challenging and often impossible due to the complexity of environments. Furthermore, even if the optimal policy can be found, it's challenging to determine whether or not it actually is the optimal policy. For the purpose of this study, it was not necessary for the *Agents* to behave optimally, but behave in a way that felt appropriate to the players. The quality of the *Policy* was therefore determined by the *Agents* ability to sustain their *Needs*, and react appropriately to the *Human*, as the *Human* represented the player in the *Environmnet*.

Structure. A policy can be constructed in various ways. In this project, the *Policy*, illustrated in Figure 6, was constructed using three components: the *Deep Neural Network (DNN)*, a softmax function, and probability distributions. The *Policy* operated within a discrete action space, meaning it could only produce whole numbers. However, this did not limit it from processing decimal numbers as input. This choice was motivated by the nature of the *Agents* behavior, which was inherently discrete, such as walking or running, rather than specifying the exact velocity of movement. If the *Agents* were to determine precise velocities, a continuous action space would have been more suitable.

As a brief overview of the *Policy*, for each *Agent*: the *DNN* processed the observations of the *Agent* to produce logits, which represented non-normalized probabilities that indicated how likely the *Agent* was to execute certain actions. These logits were normalized by a softmax function and stored in probability distributions, which represented the normalized likelihood of executing certain actions. Randomly generated values were then used to sample actions from the distributions and pass them to the *Translator*. Therefore, the selected actions depended on both the probabilities in the distributions and the random values. This approach allowed the *Policy* to produce different actions even when processing identical observations, which is the core of stochastic behavior.

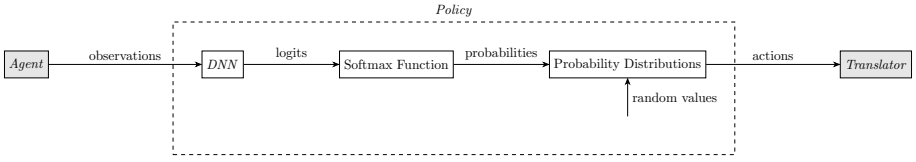

Figure 6: Structure of the *Policy*.

Observations. Full observability means that an agent can observe every bit of information in an environment. This is mostly a theoretical concept, as observational capabilities are limited. Practically, it's often impossible to achieve full observability unless the environment is simple and small. Therefore, a subset of observations needs to be chosen. The chosen observations significantly impact an agent's ability to optimize a policy. If observations are useless or unnecessary, the agent must learn to disregard them, which is a waste of time and effort.

The identification of useful observations was determined through an extensive testing phase. During this phase, the *Agents* were given different observations of the environment, and the selected observations were based on the *Agents* ability to optimize the *Policy*.

The *Agents* were allowed to observe information about the *Human* and their respective closest mushroom. These observations were all of a spatial nature, such as x, y, and z positions, relative positions between each *Agent* and the objects, dot-products, and distances. Essentially, information that helped the *Agents* locate these objects.

The *Agents* were also allowed to observe information about themselves. They could observe their own positions, the satisfaction of their *Needs*, and opinions of the *Human*. However, they did not observe their own relative positions, dot-products, or distances, as these variables require a reference object for measurement. Although it would have been possible to measure these variables against the center of the *Scene*, it was determined that their own positions were already sufficient for them to learn their location within the *Scene*. Additionally, they observed a property indicating whether their front feet were in a pool. This property was crucial for determining if they could bend down and drink at that moment. Spatial observations about the location of pools were not provided, as the pools were scattered widely around the *Scene*, and the *Agents* did not have difficulty learning to find them.

All spatial observations were normalized between -1 and 1, simplifying the learning process of the *Policy*. Non-normalized observations are generally not recommended, as they can easily diverge or become very large, making it harder to optimize a policy. The satisfaction of each *Need*, opinion of the *Human*, and the property indicating if the *Agents* were standing in water were already naturally normalized. The chosen observations, and their numerical type, are detailed in Table 1 on the following page.

Target	Observation	Type
Human	x-position	float
	y-position	float
	z-position	float
	relative x-position	float
	relative y-position	float
	relative z-position	float
	dot-forward	float
	dot-right	float
	distance	float
Closest Mushroom	x-position	float
	y-position	float
	z-position	float
	relative x-position	float
	relative y-position	float
	relative z-position	float
	dot-forward	float
	dot-right	float
	distance	float
Agent (Itself)	x-position	float
	y-position	float
	z-position	float
	hunger	float
	thirst	float
	opinion	integer (-1, 0, or 1)
	standing in water	boolean (0 or 1)

Table 1: Table of observations made by the *Agent* paired with their numerical type.

Deep Neural Network. DNNs are mathematical functions that can be used to map observations to actionable outputs. A policy does not necessarily have to use a DNN for this task; any mathematical function could be used, as long as it is complex enough to model the relationship between observations and actions. However, DNNs are a common choice due to their proven ability to approximate these relationships.

The DNN is distinct from the policy itself. While the policy produces the final, definite actions, the DNN is a step in the process. The output of the DNN can represent different things depending on the structure of the policy, but it usually relates to the actions in some manner.

In this study, the *DNN* produced logits, which are non-normalized probabilities indicating how likely an *Agent* was to select specific actions. For instance, if three logits were produced with numerical values 4, 7, and 3, they indicated that the *Agent* had three options to select from and was most likely to select the second action, as it had the highest probability.

The selection process is more complicated than this simple example. It involves normalizing the logits and grouping them into probability distributions. This process is further explained later in this section.

DNNs can have diverse structures but consist of artificial neurons connected by edges. These neurons are typically organized into layers: an input layer, hidden layers, and an output layer. The input layer receives the observations, while the output layer provides the output based on these observations. As observations pass through a DNN, they are transformed by mathematical operations into signals. When the signals reach the output layer, they represent the decisions made by the DNN.

Each neuron holds a numerical value called a bias b. Each edge holds a numerical value called a weight w. When a signal passes from neuron i through its edge to a neuron j, it is multiplied by the weight w_{ij} of the edge, and the receiving neuron's bias b_j is added. If a neuron receives signals from multiple neurons, all signals are summed into a resulting signal s_j. This mathematical operation is illustrated in Equation 2.

$$s_j = \sum_i w_{ij} x_i + b_j \tag{2}$$

The signal is then processed by an activation function, which determines whether the signal should be passed on, modified, or stopped. In this project, the activation function was a Rectified Linear Unit (ReLU) function, illustrated in Equation 3. ReLU is designed to stop signals if they have a negative value.

$$\text{ReLU}(s) = \max(0, s) \tag{3}$$

The *DNN*, illustrated in Figure 7, was a fully connected feed-forward network (FFN) with two hidden layers. In a FFN, neurons are structured into layers, where each neuron connects only to neurons in the next layer. Fully connected means each neuron is connected to every neuron in the following layer.

The input layer had 25 neurons, one for each observation. The output layer had 9 neurons, one for each *Sub-Behavior*, which are introduced in the following section. Each hidden layer contained 128 neurons, a configuration chosen based on common practice and established reliability.

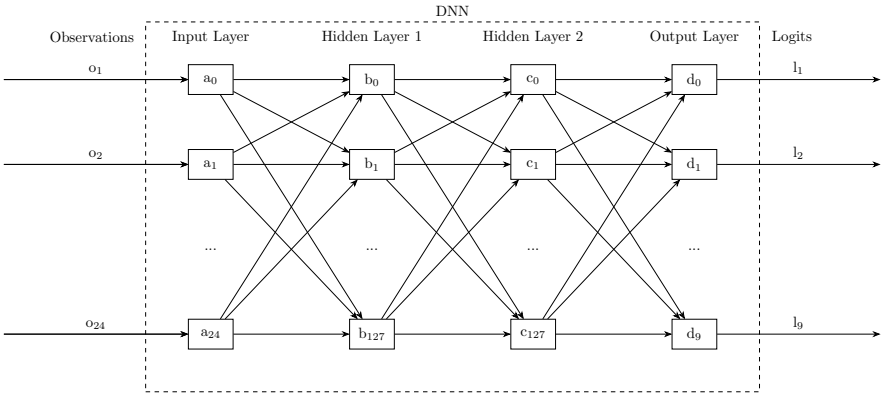

Figure 7: Structure of the *DNN*.

Probability Distributions. To fully grasp the concept of this chapter, it is recommended to read the introduction to the following section, as it provides a structural overview of the *Translator*, which is discussed in this section.

Probability distributions provide a set of probabilities that indicate the likelihood of an agent selecting specific actions. In contrast to logits, these probabilities are normalized to sum to 1. Normalization facilitates the selection process, as non-normalized values are harder to sample. A distribution containing three probabilities provides three options for the agent to choose from. For instance, a distribution of $(0.1, 0.6, 0.3)$ indicates that the agent is most likely to select the second action, as it has the highest probability of 60%.

Each time the *Policy* was called, the *Agent* needed to pass one action to each *Branch* in the *Translator*. Since the *Translator* contained three *Branches*, the *Agent* needed to select three actions. This, in turn, required three separate distributions, as only one action can be sampled from each distribution. Furthermore, since each *Branch* contained three *Sub-Behaviors* to choose from, each distribution had to contain three probabilities, representing three options.

To normalize the logits into probabilities, a softmax function, illustrated in Equation 4, was used, where l represents a vector of three logits:

$$\text{softmax}(l) = \left(\frac{e^{l_0}}{e^{l_0} + e^{l_1} + e^{l_2}}, \frac{e^{l_1}}{e^{l_0} + e^{l_1} + e^{l_2}}, \frac{e^{l_2}}{e^{l_0} + e^{l_1} + e^{l_2}} \right) \tag{4}$$

By normalizing three of the nine logits produced by the *NN* at a time, it ensured that the logits associated with one *Branch* did not influence the probabilities associated with another. The result was three distributions, each containing three probabilities, corresponding to the three *Branches* in the *Translator*.

As an example, if the first three logits associated with the first *Branch* had the numerical values 4, 7, and 3, they would be normalized into the distribution $(0.047, 0.936, 0.017)$. This distribution indicates that the second *Sub-Behavior* within this *Branch* is most likely to be chosen, with a probability of 93.6%.

The final action was then selected from the distributions by generating a random sampling value r between 0 and 1 and sampling the action for which the condition in Equation 5 was true, where i represents the selected action, and p_j represents the probability of action j in each distribution.

$$i \quad \text{if} \quad \sum_{j=0}^{i-1} p_j \leq r < \sum_{j=0}^{i} p_j \tag{5}$$

In the case of the previous example, where the distribution $(0.047, 0.936, 0.017)$ was associated with the first *Branch*, if r was randomized to 0.99, the third action would be sampled. This is because the sum of the previous probabilities, $0.047 + 0.936$, is less than 0.99, but the sum including the third action, $0.047 + 0.936 + 0.017$, exceeds 0.99. The selected action, in this case represented by the numerical value 2, was then passed to the *Translator*, which activated the third *Sub-Behavior* in the first *Branch*, as it was also represented to the numerical value 2.

3.3.2 Translator

The purpose of the *Translator* was to convert the actions produced by the *Policy* into *Instructions* that the *Agent* could use to execute its behavior. On one side of the *Translator*, actions came in, and on the other side, *Instructions* came out.

The *Translator* was structured into three hierarchical layers, illustrated in Table 2: *Branches*, *Sub-Behaviors*, and *Instructions*. An *Instruction* was always paired with a state. This structure resembled a branching tree: each *Branch* contained a collection of *Sub-Behaviors*, which in turn contained one or more *Instructions*. When the *Policy* produced actions, each *Branch* received one of them, and activated one of its *Sub-Behaviors* based on this action. The activated *Sub-Behavior* then stored its *Instructions* in the *Memory*, which is explained later in this section. In the end, the *Memory* contained all *Instructions* from all activated *Sub-Behaviors*. The *Memory* was continuously read by the *Agent* to perform its behavior. Figure 8 illustrates an example of the process, which is further explained in this section.

Branch	**Sub-behavior**	**Instruction**	**State**
Movement	Walk	move	positive
	Run	move	positive
		sprint	active
	Empty		
Turning	Turn Right	turn	positive
	Turn Left	turn	negative
	Empty		
Special	Eat	eat	active
	Drink	drink	active
	Empty		

Table 2: Structure of the Translator.

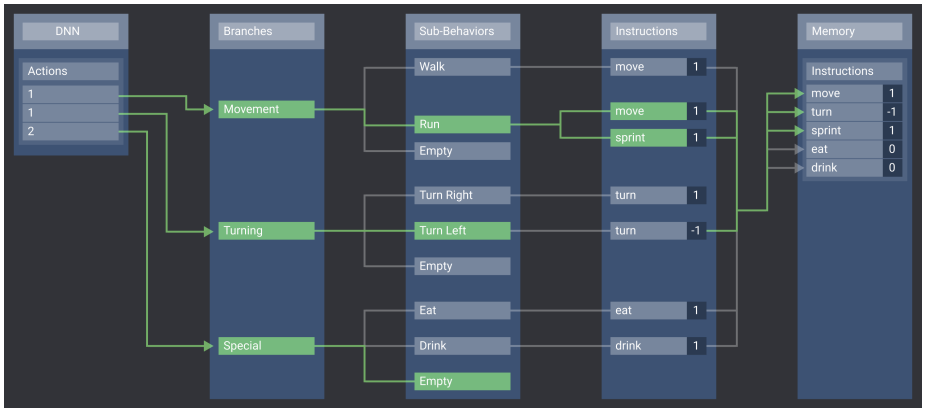

Figure 8: An example of the *Translator*, illustrating the process of receiving actions from the *DNN*, producing *Instructions*, and storing them in the *Memory*.

Branches. *Branches*, the first layer of the *Translator*, was each responsible for different parts of the behavior. Each time the *Policy* produced actions, they each received one of these actions. There was always one action produced for each *Branch*; as there were three of them, there were three actions. Each *Branch* used its received action to activate one of its *Sub-Behaviors*. For instance, if a *Branch* contained two *Sub-Behaviors*, *Walk* and *Run*, it would receive an action with the numerical value 0 or 1 from the *Policy*. In this case, if the action had the numerical value 1, it would activate *Run*, as it was the second *Sub-Behavior*. A *Branch* could only have one active *Sub-behavior* at any given time.

This structure prevented the *Agent* from activating conflicting *Sub-Behaviors* simultaneously. For instance, it ensured that *Walk* and *Run* could not occur at the same time, as they were part of the same *Branch*. However, *Walk* and *Turn Left* could be combined, as they were part of different *Branches*.

In this study, three *Branches* were created and named: *Movement*, *Turning*, and *Special*. *Movement* contained the *Sub-Behaviors Walk* and *Run*. Turning contained *Turn Left* and *Turn Right*. Special contained *Eat* and *Drink*. Each *Branch* also contained an empty *Sub-Behavior*, which, when activated, did nothing. This allowed the *Agents* to stand still and do nothing when necessary.

Sub-behaviors. *Sub-Behaviors* were parts of the overall behavior. For instance, if the *Agent* was running while turning right, the activated *Sub-Behaviors* would have been *Run* and *Turn Right*. Each *Sub-Behaviors* contained one or multiple *Instructions* paired with states. *Walk* contained only one, [move, positive], while *Run* contained two: [move, positive] and [sprint, active]. Walking backward, in this context, would contain [move, negative]. The next chapter further explains the concepts of states, such as negative, positive and active.

Instructions. *Instructions* representing the smallest operations that the *Agent* could perform, such as moving and turning in a certain direction, or engaging in specific activities like eating or drinking. They were categorized into two types: boolean-based and axis-based. Boolean-based *Instructions* had two possible states: active or inactive, while axis-based had three states: negative, neutral or positive. This structure was created to encapsulate different action spaces. For example, turning could be rightwards (positive), leftwards (negative), or not at all (neutral), while eating could either be done (active) or not done (inactive); it was not possible to eat backward, thus no need for a three-dimensional action space. The categorization of the *Instructions* are listed in Table 3.

Type	**Instruction**
Axis-based	movement
	turning
Boolean-based	eat
	drink
	sprint

Table 3: Categorisation of *Instructions*.

Memory. The *Memory* was designed to store the current state of all *Instructions*. Table 4 illustrates the structure of the *Memory* in a state where its associated *Agent* was stationary. This component played a critical role in ensuring that the *Agent* could consistently execute it's behaviors.

It's inefficient and computationally expensive to have a policy produce new action at every possible moment. Moreover, constantly re-evaluating decisions does not align with natural decision-making processes. Therefore, it is standard practice to produce action at specific intervals and sustain the decisions until the next evaluation.

In this study, the *Policy* generated a new actions every 20 milliseconds. Between intervals, the *Agent* continued to perform its behavior according to the most recent decision. The *Memory* facilitates this by maintaining a record of the latest *Instructions*. This approach allows the *Agents* to act consistently, even between decision-making cycles.

Instruction	**State**
move	neutral
turn	neutral
sprint	inactive
eat	inactive
drink	inactive

Table 4: Structure of the Memory, which stores *Instructions* paired with states. In this instance, all *Instructions* are in a neutral or inactive state, which meant the *Agent* would do nothing.

3.3.3 Needs

A reward function is crucial for guiding an agent's learning process. It provides feedback on whether an agent is learning something useful. Designing an effective reward function is a complex task that often involves extensive trial and error. Typically, this process involves creating an initial reward function, training an agent—which can take several hours—and then iteratively modifying and testing the reward function until it effectively promotes the desired behavior. This iterative process is often highly time-consuming, so methods that simplify it are highly beneficial. To address this concern, the *Needs* were developed.

The *Needs* were components designed to be flexible and modifiable. In this sense, they were not just needs the *Agent* needed to tend to, but also a framework for managing needs. Each *Need* was built on two variables: satisfaction, duration, and one component: *Local Reward Function (LRF)*. The *Needs* that were used in the final implementation of the *Agents* were hunger and thirst. The *Agents* could increase the satisfaction of these *Needs* by eating mushrooms, and drinking water from the pools.

Satisfaction was a numerical value ranging from 0 to 1, indicating the current level of fulfillment. A lower value signified deprivation (e.g., being hungry), while a higher value signified saturation (e.g., having overeaten). The satisfaction of a need continuously decreased over time based on its duration, meaning the *Agent* would naturally become hungry or thirsty as time passed.

The duration was important as it determined how quickly the satisfaction of a need decreased. By modifying the duration, it was possible to adjust how the *Agent* prioritized certain *Needs*. For instance, extending the duration of hunger allowed the *Agent* to spend more time on other activities rather than constantly seeking food. Equation 6 illustrates the relationship between satisfaction and duration.

$$s_t = s_{t-1} - \frac{1}{d} \tag{6}$$

where s_t represent the decreased satisfaction, s_{t-1} denote the satisfaction in the previous timestep, and d denotes the duration.

Satisfaction alone did not indicate how well a *Need* was being fulfilled; this was determined by the *LRF*. The *LRF* spanned the same interval as the satisfaction, from 0 to 1, taking satisfaction as input and producing a reward, which is illustrated in Equation 7, where s denotes the satisfaction of a *Need*, and r denotes the reward given to the *Agent*:

$$LRF(s) = r \tag{7}$$

The structure of the *LRF* was initially designed through intuition, but later refined though trial and error by analyzing the *Agents* learning process during training, evaluating what worked well and what didn't. In the next page, the structure of the *LRF* is explained.

Local Reward Function. The *LRF*, illustrated in Equation 8, was specifically designed to ensure the agent received appropriate feedback for maintaining or neglecting its *Needs*. Its shape, resembling a wave, was chosen to create a nuanced response to the *Agents* behavior:

1. **Gradual Penalty** for Neglect: Initially, the reward decreases smoothly, allowing the *Agent* some flexibility without immediate harsh penalties.
2. **Sharp Penalty for Severe Neglect**: As neglect becomes more severe, the reward drops sharply, strongly discouraging the *Agent* from ignoring critical *Needs*.
3. **Balanced Reward for Maintenance**: When *Needs* are adequately maintained, the reward remains relatively stable, providing consistent positive feedback.

$$f(x) = \begin{cases} 0.5 \cdot (s_1 - s_0) \cos\left(\frac{\pi(x-p+r)}{1+p-r}\right) + s_0 & \text{for } -1 < x \leq p - r \\ s_1 & \text{for } p - r < x < p + r \\ 0.5 \cdot (s_1 - s_2) \cos\left(\frac{\pi(x-p-r)}{1-p-r}\right) + s_2 & \text{for } p + r \leq x < 1 \end{cases} \tag{8}$$

Where the following variables are represented as:

1. Satisfaction point p represents the value on the x-axis where max reward is given, examples given in Figure 9.
2. Satisfaction range r represents the range of which the satisfaction can stray from the satisfaction point without the *Agent* receiving less reward, examples given in Figure 10.
3. Deprived reward s_0 represents the reward given when a *Need* is fully neglected, examples given in Figure 12.
4. Satisfied reward s_1 represents reward given when a *Need* is maintained, examples given in Figure 11.
5. Saturated reward s_2 represents the reward given when a *Need* is fully saturated, examples given in Figure 13.

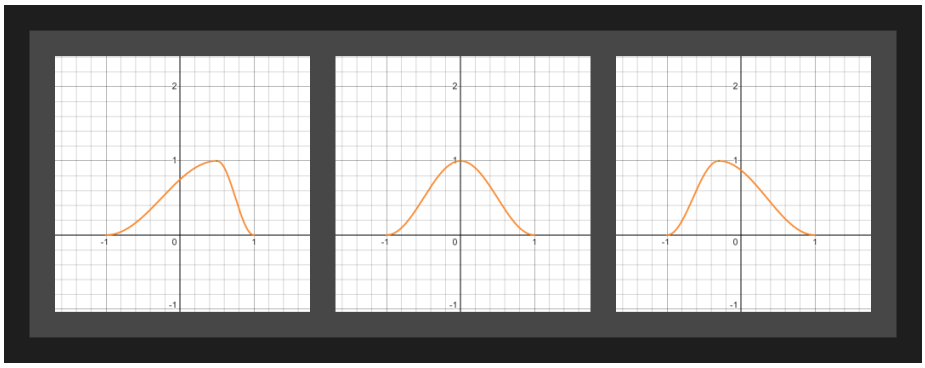

Figure 9: From left to right: satisfactionPoint equals 0.5, 0 and -0.3

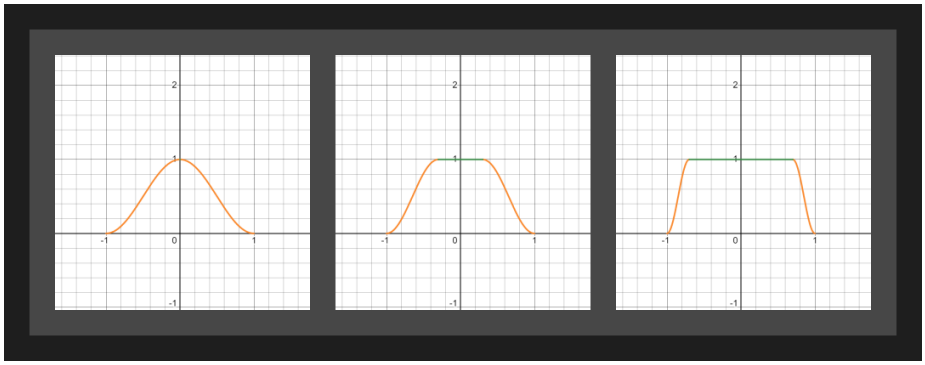

Figure 10: From left to right: satisfactionRange equals 0, 0.3 and 0.7

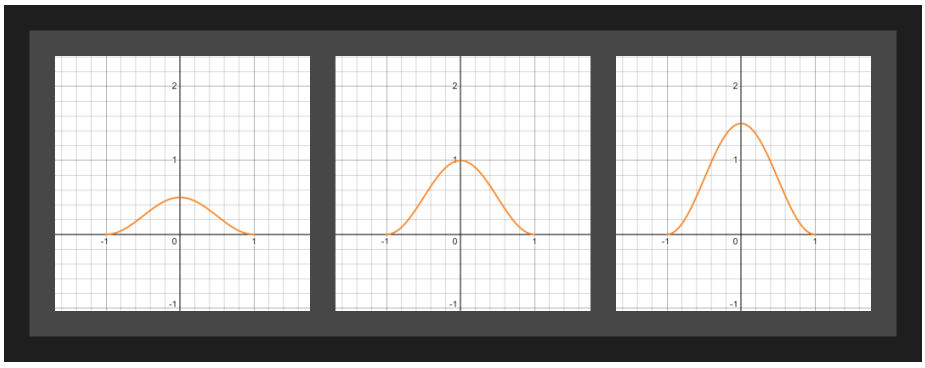

Figure 11: From left to right: satisfiedReward equals 0.5, 1 and 1.5

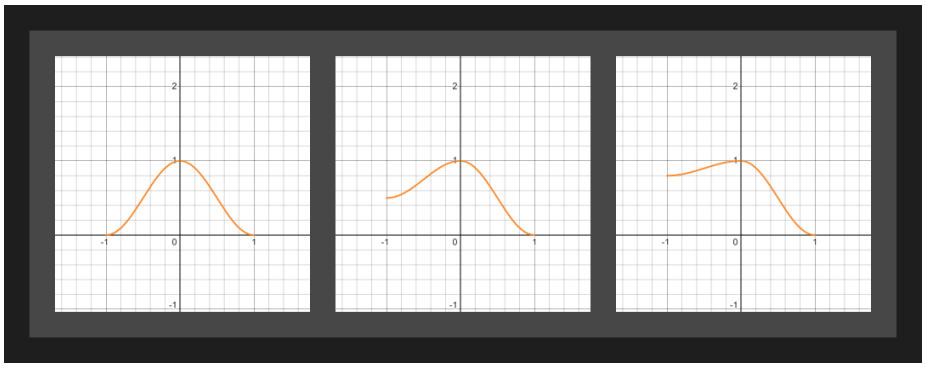

Figure 12: From left to right: deprivedReward equals 0, 0.5 and 0.8

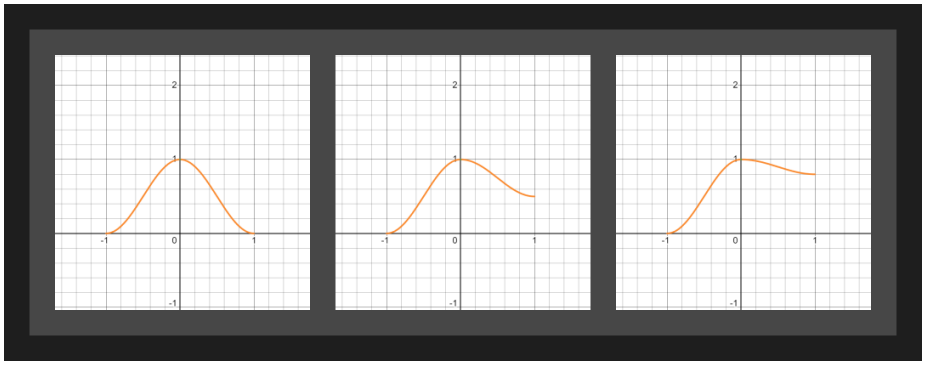

Figure 13: From left to right: saturatedReward equals 0.5, 1 and 1.5

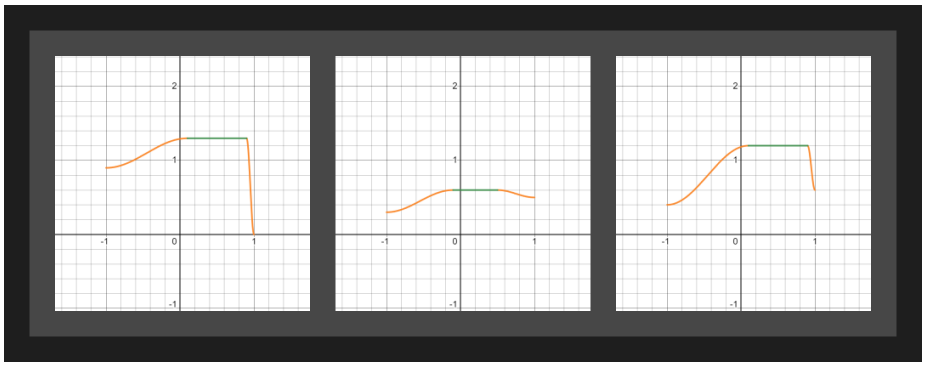

Figure 14: Three examples of random *LRFs*

3.3.4 Training

The training of the *Agents* was done with the DRL algorithm PPO. The hyperparameters that was used for the algorithm are listed in the Appendix A. This section introduces and explains the training, encompassing the *Reward Function* and the structure of the training process.

Reward Function. A reward function is the mechanism that provides rewards to an agent based on its behavior. In this study, the *Reward Function*, illustrated in Equation 9, was based on three variables: the reward from the *LRF* of each *Need*, the distance between an *Agent* and the *Human*, and an *Agent's* opinion of the *Human*.

The *Agents* opinion o of the *Human* had three states: dislike, indifferent, and like. When disliking the *Human*, higher rewards were given when maintaining a larger distance d from the *Human*. Conversely, when liking the *Human*, higher rewards were given when staying close to the *Human*. When indifferent to the *Human*, the reward was not influenced by the distance between them.

This reward was scaled according to half the size of the *Scene*. For instance, if the *Agent* disliked the *Human*, the highest reward was given when the *Agent* was at least half the *Scene* away from the *Human*. This design choose was made to ensure the *Agents* did not consistently have to stay at the opposite side of the *Scene* at all times, in relation to the *Human*.

The resulting variable was then multiplied with all the rewards from the *LRF* of each *Need*, to produce the final reward.

$$R_{total} = k \prod_i r_i \tag{9}$$

where

$$k = \begin{cases} \min(d \cdot 2, 1) & \text{if } o = -1 \\ 1 & \text{if } o = 0 \\ \max(1 - d \cdot 2, 0) & \text{if } o = 1 \end{cases}$$

$$r_i = \textit{LRF} \text{ of } \textit{Need} \text{ i}$$

$$d = \text{distance between } \textit{Agent} \ \& \ \textit{Human}$$

$$o = \textit{Agent's} \text{ opinion of the } \textit{Human}$$

There are two major methods for providing rewards to an agent: dense and sparse rewards. Dense rewards indicate that an agent receives a reward at every timestep based on its overall behavior in the environment. In contrast, rewards are considered sparse if an agent is given rewards exclusively when it achieves a specific goal.

To provide some context, in this study, it would have been considered sparse rewards if the *Agents* received a reward every time they found and consumed food. This was not the case. Instead, the *Agents* were given a reward at each timestep. Consuming food only affected the *Agents* hunger, and the *Agents* continuously received a higher reward if this action balanced their *Needs* effectively.

Training in Episodes. Training in episodes is a common approach in when optimizing a policy. Each episode represents a separate sequence of interactions between an agent and an environment.

In this study, each episode began with the *Agents* placed in random locations in the virtual environment. The *Agents* interacted with the environment by navigating, finding food, water, and responding to the *Human*'s presence. Rewards were given at each timestep based on how well the *Agents* were fulfilling their *Needs*. Interacting with the environment, such as eating or drinking, affected *Needs*, which continuously provided better or worse rewards based on how fulfilled they were.

The episode ended after 5000 timesteps, never by achieving a specific goal. The *DNN* of the *Policy* was updated based on the experiences gathered during the episode, allowing it to improve its behavior in subsequent episodes. The training ended when the *Agent* was not able to optimize the policy any longer, which was determined by the cumulative reward function plateauing, indicating that further training did not result in significant improvements. This was observed after approximately 4 million timesteps.

Varied Experiences and Dynamic Environment. When training an agent, it is important to provide varied experiences to avoid overfitting. Overfitting occurs when the agent learns to perform very well in the specific training environment but fails to generalize to different or slightly varied environments. This happens because the agent experiences the same scenarios repeatedly, leading it to believe that the environment is static and unchanging.

In this study, a dynamic environment meant that the observations made by the *Agent* had to change. Most of the *Agents* observations were spatial, such as the positions of itself, the *Human*, and the nearest mushroom. These observations could be made dynamic by changing the location of the *Agent*. However, for the *Human*, this was not enough, as the *Agent* needed to learn that the *Human* would move around, mimicking the behavior of players. This was done by integrating the *FSM*, as previously explained in Section 3.2.

To add further dynamics to the environment, it was essential to change the environment at the beginning of each episode so that the agent received more varied observations. The variables listed in Table 5 showcase the variables that were randomly modified at the beginning of each episode.

Object	**Variable**
Human	position (x, y)
	state of FSM (Stalk / Wander)
	target *Agent* to stalk (only in Stalk state)
Agents	position (x, y)
	opinion of *Human*
	satisfaction of each need (hunger and thirst)

Table 5: Table of variables that changed at the beginning of each episode, where Object refers to the entity which contained the Variable.

4 Methodology - Evaluation

When the *Environment* was fully developed, player testing was conducted to evaluate the player experience in the *Environment*. All players were men in their mid-20s, active in the video game industry, identifying as gamers, and educated in ML. Before testing, players were briefed on the study's objectives. During testing, they interacted with the *Environment* and were encouraged to verbalize their thoughts so that their experiences could be recorded for further analysis.

Afterwards, interviews were conducted to ground their experiences in established literature on player experience. The purpose of the interviews was to explore connections between the experiences in the *Environment* and the existing literature on player experience. The Player Experience Inventory (PXI), a comprehensive study on measuring player experience, was chosen to represent this literature.

This methodology ensured that data recorded during testing was not influenced by the interviews, as they were conducted afterwards, helping to mitigate interview bias.

4.1 Player Experience Inventory

The Player Experience Inventory (PXI) was developed to quantitatively assess player experiences at both functional and psychosocial levels. Functional levels refer to immediate experiences resulting from game design choices, such as audiovisual appeal and ease of control. Psychosocial levels, on the other hand, refer to deeper emotional experiences like feelings of curiosity or a sense of mastery. The development of the PXI involved consultations with 64 game user research experts, leading to the creation of several scales, grouped into constructs based on their thematic relevance. The subsequent validation of these scales was conducted through five studies across different populations, involving a total of 529 players. The findings from these studies affirm the scales' theoretical structure and demonstrate their discriminant and convergent validity. Additionally, the PXI showed consistent performance across various sample sizes and research designs, evidencing its configural invariance. This robust validation underscores the PXI's reliability as a framework for measuring player experience [1].

The PXI's ability to assess the nuanced relationship between game design choices and player emotional responses makes it a valuable tool for validating player experience. In this study, the PXI was used to ground the results of the player testing within established literature, by evaluating how well the players' experiences within the *Environment* align with the expected player experiences.

4.1.1 Adaptation

The PXI consists of 10 constructs, divided into functional and psychosocial levels. Functional constructs measure immediate experiences resulting from game design choices, such as ease of control, goals, and rules. Psychosocial constructs measure second-order emotional experiences, such as immersion and curiosity. Since this study aimed to evaluate the effects of stochastic behavior on players, rather than game implementation, the functional constructs were excluded.

Originally, the PXI employs quantitative measures by assessing scales. However, due to the small number of players and the need for qualitative insights, the scales were removed, and the definitions of the constructs were used directly. These constructs were slightly redefined to better align with the specific nature of the *Environment*. The constructs are illustrated in Table 6.

Construct	Definition/Redefinition
Autonomy	Autonomy is defined as the feeling of freedom to engage with the game according to preferences. It was redefined as feeling like actions have an impact on the *Agents* behavior.
Curiosity	Curiosity is defined as a feeling of interest or curiosity. It was redefined as feeling curious or interested in any aspect related to the *Agents* behavior.
Meaning	Meaning is defined as a sense of connection with the game. It was redefined as feelings of connection or emotion towards the *Agents*, in response to its behavior.
Mastery	Mastery is defined as a feeling of achievement or a sense of competence. It was redefined as feelings of insight or learning something new about the *Agents* behavior.

Table 6: The psychosocial constructs taken from the PXI, their definition and redefinition.

4.1.2 Testing Method

Briefing. Player testing began with a briefing on the study's objective. players were then asked to complete a form providing informed consent for voice recording during the session and subsequent data analysis. Any questions or concerns raised by the players were addressed at this stage.

Gameplay Session. players were given instructions on how to control the *Human*, navigate the *Environment*, and interact with the *Agents*. They were encouraged to explore all possible interactions, including the tools provided to the *Human*. players were allowed to freely interact with the *Environment* for approximately 5 minutes to familiarize themselves with the controls. After this, they were encouraged to actively interact with the *Agents* for approximately 15 minutes. During this step, players were asked to think out loud, and every phrase they said was collected for later analysis.

Interview. After the gameplay sessions, a structured interview was conducted. The interview questions were designed according to the constructs taken from the PXI, with one question for each construct. The questions were phrased generally, and not specifically about the *Agents* to avoid bias. The questions were as follows:

1. Autonomy: Can you describe a moment during the gameplay where you felt like your actions had an impact?
2. Curiosity: At any point during the gameplay, did anything spark a feeling of interest or curiosity?
3. Meaning: Was there ever a moment during gameplay where you experience any emotions or feelings of connection?
4. Mastery: Was there a moment during the gameplay where you felt you gained insights or learned something new?

Debriefing. At the conclusion of the testing sessions, the players were debriefed and provided with additional information about the study. Any remaining questions or concerns were addressed, and players were thanked for their participation.

4.2 Thematic Analysis

Thematic analysis is a qualitative research method that aims to identify themes within data. The process typically starts with the familiarization phase, where researchers immerse themselves in the data through reading and re-reading. Following this, initial codes are generated by systematically sampling and labeling data points of interest. These codes are then grouped into potential themes, which are reviewed and refined to ensure they accurately represent the data. If a theme only contains a few data points, it might be removed, as it indicates that those data points do not represent the data in a larger context [17].

Given its versatility and depth, thematic analysis is well-suited for validating research findings in studies that delve into the intricacies of player experiences. Player experience is hard to quantify, and the capacity to uncover underlying themes and patterns within qualitative data makes it an excellent tool for exploring how players perceive and are affected by game elements. By applying thematic analysis, it was possible to interpretation the data recorded during player testing, ensuring that the validation of research findings is grounded in a comprehensive understanding of player experiences.

Application. Thematic analysis was used to extract phrases from the recordings of the player testing, while considering the context they were expressed in, to grasp the underlying meaning. These phrases were subsequently tagged with codes reflecting this underlying value.

Phrase Extraction. The initial step in thematic analysis involved identifying valuable phrases within the interview data. Following transcription, each interview was closely reviewed alongside its recording to ensure a comprehensive understanding of the context. Statements by players concerning the *Agents* that were deemed valuable were extracted. For each selected phrase, a descriptive analysis of its underlying meaning was conducted. The next phase involved coding these phrases, as described in the next paragraph.

Coding Process. The extraction of phrases was followed by the coding phase, where each phrase, along with its interpreted meaning, was catalogued in a spreadsheet. For instance, the statement "The agent isn't hungry" was coded under "Emotional Attribution" and "Perceived Insight," indicating the player's attribution of hunger to an *Agent* and the perception of the *Agent*'s lack of hunger. Codes were then subject to a filtering and merging process, elaborated in the following paragraph.

Code Filtering. This stage involved refining the coded phrases by merging or removing similar or singular codes. Codes such as "Emotional Attribution" and "Motivational Attribution," both projecting human-like emotions or motivations onto *Agents*, were merged due to their similarity. Codes representing isolated instances were excluded to focus on recurring patterns within the player

experience, which left some phrases without any assigned codes. These phrases were also removed.

Theme Development. Instead of adopting the conventional approach of generating themes through thematic analysis, it was decided to leverage the interview results to directly connect developed codes with pre-existing theoretical themes on player experience. Four psychological constructs from the PXI had been chosen to serve as these predefined themes, facilitating a targeted exploration of how interview findings align with established concepts in player experience research.

5 Result

Thematic Analysis. The thematic analysis uncovered eight codes in the recorded player testing data, illustrated in Table 7.

Code	**Occurrence**	**Definition**	**Example**
Attribution of Intention	21	Players believe an *Agent* possess particular intentions.	"The raccoon is trying to find water"
Attribution of Motive	17	Players believe an *Agent* possess particular motives.	"The raccoon is sitting still because it is tired'
Independent Behavior	12	Player observes an *Agent* behave autonomously from the player.	"The raccoon is doing something without me"
Influence	12	Player believes they have an impact on the behavior of an *Agent*.	"The raccoon likes me because I gave him a tomato"
Uncertainty	12	Players express uncertainty or doubt.	"I believe the agent likes me"
Attribution of Emotion	9	Players believe an *Agent* possess particular emotions.	"The raccoon is feeling hungry"
Unexpected Behavior	7	Player observes *Agent* behavior that was not anticipated by the player.	"The raccoon stayed with me even if I slapped it"
Relationship	7	Players express having formed a connection with an *Agent*.	"This raccoon likes me"

Table 7: The uncovered codes from the thematic analysis, their occurrences, definition and an example a phrase that was sampled.

During the interview, participants were asked about their experiences to ground the codes in established literature. Table 8 illustrates these relationships.

Autonomy. Players were asked to describe moments during gameplay when they felt their actions had consequences or an impact. They identified three key reasons: having an **Influence** on the *Agents*' behavior and how these influences forged **Relationships** with the *Agents*, and observing the *Agents*' **Independent Activities**, which made them feel that their absence also mattered.

Curiosity. Players were asked to describe moments during gameplay that sparked their curiosity. They identified five key reasons: having an **Influence** on the *Agents*' behavior, forging **Relationships** with the *Agents*, observing the *Agents*' **Independent Activities**, noticing **Unexpected Behavior** from the *Agents*, and experiencing **Uncertainty** about the *Agents*' actions. These aspects piqued their curiosity about the game dynamics.

Meaning. Players were asked to describe moments during gameplay when they felt emotions or a sense of connection. They identified three key reasons: the perception of **Relationships** with the *Agents*, observing **Unexpected Behavior** from the *Agents*, and witnessing the *Agents*' **Independent Activities**, as it made them feel like the *Agents* had their own purpose. These elements made players feel connected to the *Agents* and enhanced their emotional engagement.

Mastery. Players were asked to describe moments during gameplay when they felt they gained insights or learned something new. They identified four key reasons: having an **Influence** on the *Agents*' behavior, building **Relationships** with the *Agents*, observing **Unexpected Behavior** from the *Agents*. These aspects contributed to their learning experience and sense of mastery as they navigated the game.

	Autonomy	**Curiosity**	**Meaning**	**Mastery**
Influence	✓	✓		✓
Relationship	✓	✓	✓	✓
Independent Behavior	✓	✓	✓	✓
Unexpected Behavior		✓	✓	
Uncertainty		✓		

Table 8: Table of codes created from the thematic analysis, showing their relevance to each construct.

Furthermore, five instances occurred where a player expressed an experience specifically positive. Four of them were linked to *Unexpected Behavior*, and four were linked to *Relationship*. Examples are "I like that the raccoons don't do what I want them to sometimes, it makes thing more fun!" and "I love that this raccoon stayed with me even when I slapped his friends".

6 Discussion

Answering Research Questions. This thesis focused on stochastic DRL-based NPC behavior and its effects on player experience. The result contribute by providing insights into the way players are affected by such behavior och how they choose to interact with it. In this section the research questions will be answered and discussed, along with some thought about future work.

1. How does stochastic DRL-based behavior in ambient NPCs influence player experience in video games?
   (a) Which aspects of the behavior are recognized by players?
   (b) To what extent does the behavior affect players' interaction strategies?
2. How do the results of this study correlate with established literature on player experience?

Research Question 1: Introducing stochastic DRL-based behaviors in ambient NPCs has yielded numerous positive outcomes in player experience. The infusion of unpredictability has notably enriched the perceived depth of NPC interactions. Consequently, players often attribute human-like intentions, emotions, and motives to NPCs, particularly when their actions deviate from expectations. This uncertainty prompts players to engage in creative reflection, imagining and projecting narratives that may not be inherently true but contribute to their perception. Despite lacking actual learning capabilities, the stochastic DRL-based NPCs has shown successful in giving such impressions to the player by adapting the behavior based on their interactions. Emotional bonds between players and NPCs are notable, especially when NPCs are perceived to exhibit affectionate behaviors.

Research Question 1.a: Players' engagement goes beyond mere interaction; they are deeply invested in understanding and exploring the dynamic behaviors of the NPCs. The uncertainty of the NPCs behavior left players feeling curious, shifting their focus to a deeper level of engagement. The players wanted to explore how these characters could learn and adapt over time, which adds a layer of depth to the gameplay. Moreover, players are not just interested in the mechanics but also in forming meaningful relationships with NPCs. This desire for connection signifies a shift towards a more immersive and emotionally rich gaming experience, where players are not just players but active contributors to the narrative and evolution of the game world.

The results show that players tend to focus on behaviors that are unexpected, as these spark curiosity in the player. They also focus on human-like characteristics in the NPCs, with a strong tendency to attribute these characteristics to the NPCs. Furthermore, players focus on relationships, feeling a sense of immersion when the NPCs show signs of liking or disliking the player.

Research Question 1.b: The uncertainty introduced by the NPCs has been shown to affect players' strategies. When NPCs display unexpected behaviors, players become curious and adapt their interactions to learn more. As players find meaning in the relationships they form and recognize that their actions impact the NPCs, they tend to change their interaction strategies to shape these relationships.

Research Question 2: Comparing player experiences with to existing research further bolsters the results. In the PXI framework, player experience was described through psychological constructs, four of which were examined in this thesis. All four constructs were found to be connected to various aspects of the player experience outlined in this study. This correlation underscores the alignment of this methodology with previous literature in player experience.

6.1 Future Work

Player Diversity. When conducting research involving user studies, it is essential to ensure demographic diversity. In this study, all players were male, identified as gamers, active in the video game industry, and in their 20s. This lack of diversity likely influenced the results. A more comprehensive approach would involve expanding the demographic scope and understanding how different players engage with DRL-driven NPCs. For instance, cultural backgrounds can shape player expectations and interactions, age can impact adaptability to complex NPC behaviors, and educational backgrounds can correlate with strategies and problem-solving approaches in games. Additionally, investigating the influence of gender and the experiences of individuals who do not identify as gamers would provide valuable insights.

A diverse player pool can help identify unintentional biases that might alienate or disadvantage certain player groups, offering insights into creating more inclusive gaming environments. This also raises important questions about accessibility, emphasizing the need for DRL-driven games to accommodate players with varying abilities and gaming proficiencies.

Explore Long-term Player Engagement. Exploring long-term player engagement would involve examining how interactions with DRL-based NPCs with stochastic behavior evolve, affecting both player satisfaction and retention. Initially, engagement may be driven by the novelty and curiosity surrounding their behavior. However, sustaining interest requires continuously evolving challenges and interactions. Long-term studies could uncover patterns in how players adapt to these behaviors and how these adaptations influence game design. Such research could lead to DRL systems that dynamically adjust to player skill levels and preferences, ensuring the experience remains fresh and engaging.

Additionally, understanding the psychological effects of prolonged exposure to such NPCs—such as attachment or desensitization—could offer valuable insights into designing NPCs that maintain player interest over extended periods. This approach necessitates innovative game design strategies that anticipate and adapt to changing player expectations, ensuring that games remain compelling and rewarding. By focusing on these aspects, developers could create NPCs that not only captivate players initially but also foster sustained engagement and long-term satisfaction.

Enhance Behavioral Complexity. Enhancing behavioral complexity involves developing NPCs capable of exhibiting a wider range of behaviors and responses, simulating more realistic and human-like interactions. By creating NPCs that can learn new strategies and behaviors in real-time, games can offer more personalized and challenging experiences. This approach not only enhances the gameplay experience but also pushes the boundaries of what is possible in game design, fostering continuous innovation in the field of interactive entertainment.

7 Conclusion

The research reveals that DRL-based NPCs significantly affects player experiences by introducing unpredictability and diversity, encouraging players to attribute human-like qualities to NPCs and engage more deeply with the game. This shift towards stochastic behaviors fosters a dynamic player-NPC relationship, surpassing traditional scripted interactions. Players exhibit a keen interest in exploring NPC behaviors and learning mechanisms, striving to form meaningful relationships with them. This inclination indicates a move towards immersive, emotionally rich gameplay where players actively contribute to the narrative.

The study highlights that players often attribute intentions, emotions, and motives to NPCs, especially when their actions deviate from expectations. This anthropomorphism deepens player engagement, leading to a more interactive and emotionally connected experience. The unpredictability of stochastic DRL-driven NPC behaviors prompts players to engage in creative reflection, imagining and projecting narratives that enhance their gameplay experience.

Furthermore, the research underscores the importance of player autonomy, curiosity, meaning, and mastery within the PXI framework. Players' sense of autonomy is bolstered by the NPCs' adaptive behaviors, while their curiosity is piqued by the NPCs' unexpected actions. The ability to form emotional connections with NPCs adds a layer of meaning to the gameplay, and the opportunity to learn from NPC behaviors contributes to a sense of mastery.

The findings suggest that NPCs with stochastic DRL-based behavior can lead to more innovative and engaging game design, offering personalized and evolving challenges that keep the gameplay experience fresh and rewarding. The potential for such NPCs to adapt to individual player preferences and skill levels can result in long-term player engagement and satisfaction.

A PPO Hyperparameters

The following table lists the hyperparameters used for training the PPO algorithm in our experiments:

```
[language=Python, caption={PPO Hyperparameters}]
default:
    trainer: ppo
    batch_size: 1024
    beta: 5.0e-3
    buffer_size: 10240
    epsilon: 0.2
    hidden_units: 128
    lambd: 0.95
    learning_rate: 3.0e-4
    learning_rate_schedule: linear
    max_steps: 5.0e5
    memory_size: 128
    normalize: false
    num_epoch: 3
    num_layers: 2
    time_horizon: 64
    sequence_length: 64
    summary_freq: 10000
    use_recurrent: false
    vis_encode_type: simple
    reward_signals:
        extrinsic:
            strength: 1.0
            gamma: 0.99
```

References

[1] Vero Vanden Abeele, Katta Spiel, Lennart Nacke, Daniel Johnson, and Kathrin Gerling. Development and validation of the player experience inventory: A scale to measure player experiences at the level of functional and psychosocial consequences. *International Journal of Human-Computer Studies*, 135:102370, 2020.

[2] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

[3] Mehmet Caner and Maurizio Daniele. Deep learning with non-linear factor models: Adaptability and avoidance of curse of dimensionality. *arXiv preprint arXiv:2209.04512*, 2022.

[4] Marc Peter Deisenroth, Gerhard Neumann, Jan Peters, et al. A survey on policy search for robotics. *Foundations and Trends® in Robotics*, 2(1–2):1–142, 2013.

[5] Faustino Gomez and Jürgen Schmidhuber. Evolving modular fast-weight networks for control. In *International Conference on Artificial Neural Networks*, pages 383–389. Springer, 2005.

[6] Nicolas Heess, Gregory Wayne, David Silver, Timothy Lillicrap, Tom Erez, and Yuval Tassa. Learning continuous control policies by stochastic value gradients. *Advances in neural information processing systems*, 28, 2015.

[7] Devang Jagdale. Finite state machine in game development. *International Journal of Advanced Research in Science, Communication and Technology*, 10(1), 2021.

[8] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[9] Nate Kohl and Peter Stone. Policy gradient reinforcement learning for fast quadrupedal locomotion. In *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA'04. 2004*, volume 3, pages 2619–2624. IEEE, 2004.

[10] Vijay Konda and John Tsitsiklis. Actor-critic algorithms. *Advances in neural information processing systems*, 12, 1999.

[11] Jan Koutník, Giuseppe Cuccu, Jürgen Schmidhuber, and Faustino Gomez. Evolving large-scale neural networks for vision-based reinforcement learning. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 1061–1068, 2013.

[12] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[13] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research*, 17(1):1334–1373, 2016.

[14] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[15] David M Ndetei, Pascalyne Nyamai, and Victoria Mutiso. Boredom–understanding the emotion and its impact on our lives: an african perspective. *Frontiers in Sociology*, 8:1213190, 2023.

[16] Andrew Y Ng, Adam Coates, Mark Diel, Varun Ganapathi, Jamie Schulte, Ben Tse, Eric Berger, and Eric Liang. Autonomous inverted helicopter flight via reinforcement learning. In *Experimental robotics IX: The 9th international symposium on experimental robotics*, pages 363–372. Springer, 2006.

[17] Gery W Ryan and H Russell Bernard. Techniques to identify themes. *Field methods*, 15(1):85–109, 2003.

[18] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

[19] Juergen Schmidhuber and Rudolf Huber. Learning to generate artificial fovea trajectories for target detection. *International Journal of Neural Systems*, 2(01n02):125–134, 1991.

[20] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.

[21] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[22] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[23] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[24] Satinder Singh, Diane Litman, Michael Kearns, and Marilyn Walker. Optimizing dialogue management with reinforcement learning: Experiments with the njfun system. *Journal of Artificial Intelligence Research*, 16:105–133, 2002.

[25] Alexander L Strehl, Lihong Li, Eric Wiewiora, John Langford, and Michael L Littman. Pac model-free reinforcement learning. In *Proceedings of the 23rd international conference on Machine learning*, pages 881–888, 2006.

[26] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[27] Inworld Team. Ai npcs and the future of gaming. 2023.

[28] Gerald Tesauro et al. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[29] Daan Wierstra, Alexander Förster, Jan Peters, and Jürgen Schmidhuber. Recurrent policy gradients. *Logic Journal of IGPL*, 18(5):620–634, 2010.

[30] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8:229–256, 1992.

[31] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International conference on machine learning*, pages 2048–2057. PMLR, 2015.