UNIVERSITÀ
DEGLI STUDI
FIRENZE

PhD Program in Smart Computing
Dipartimento di Ingegneria dell'Informazione (DINFO)

Deep Reinforcement Learning for the Design and Validation of Modern Computer Games

Alessandro Sestini

Dissertation presented in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Smart Computing

PhD Program in Smart Computing
University of Florence, University of Pisa, University of Siena

Deep Reinforcement Learning for the Design and Validation of Modern Computer Games

Alessandro Sestini

Advisor:

Prof. Andrew D. Bagdanov

Head of the PhD Program:

Prof. Stefano Berretti

Evaluation Committee:
Prof. Julian Togelius, *New York University*
Dr. Sam Devlin, *Microsoft Research*

XXXV ciclo — October 2023

Abstract

Modern video games are one of the most important forms of entertainment today. Their importance in popular culture is demonstrated by the ever-increasing number of gamers and video games. Today's games are complex environments, with photo-realistic graphics, advanced avatar animations, and they are full of disparate interactions. One of the crucial aspects of the quality of a video game is non-player character behavior: the interactions between players and characters in the game is a vital factor, with the potential to elevate or ruin the player experience. However, the technological progress in game AI techniques has not followed that of other game design aspects - for instance advancements in game graphics. For this reason modern non-player character behaviors suffer from problems inherited from the use of stale techniques. At the same time, recent advances in deep reinforcement learning have had significant impact in training super-human autonomous agents. As a result, now it is possible to train agents that can beat professional players in modern and complex video games. Deep reinforcement learning offers the promise of creating non-player characters that are alive, smart, adaptive and challenging. From a game developer point of view, a fair question that arises looking at these successes is: *could I use these techniques to bring my game characters to life*? We argue that the answer is *no*, and for many reasons.

This thesis discusses the issues that arise from an inappropriate, if direct, application of deep reinforcement learning techniques as game design tools. In order to understand these problems, we first provide an extensive and detailed survey of the state-of-the-art in both game AI and deep reinforcement learning. This analysis defines the foundation on which this thesis builds. Then we propose a list of desiderata that each machine learning system should satisfy in order to create enjoyable non-player characters. Alongside these desiderata, we introduce a new environment and free-to-play game that serves as one of our main testbeds for this domain. Based on the proposed requirements, we design several improvements and novel algorithms that can help video game designers in the use of deep reinforcement learning as an effective design tool. These improvements focus on: the adaptability of trained agents, the problem of faulty reward functions and how to replace them, the low-level usability of current reinforcement learning algorithms, the poor quality of trained behaviors and the problem of model interpretation. For each of the proposed algorithms, we provide a detailed experimental analysis showing that these methods are indeed useful for solving the cited issues. Finally, a list of open challenges illustrates the problems that currently still exist even after the improvements proposed in this manuscript. We strongly believe that solving these challenges will represent a huge leap forward in the creation of better quality video games.

Contents

Chapter 1

Introduction

The video game industry has experienced consistent improvement in the production of quality games. Today it competes economically with the most important multimedia industries and the revenue of the video game companies in 2018 was estimated to be more than twice that of the international film and music industries combined (Statista, 2022). Compared to the not so distant past when gaming consoles and computer gaming were not ubiquitous, video games are no longer a niche but are transversal across ages, genders and devices. Today there are more than 3 billion estimated video gamers worldwide. This is especially thanks to mobile games and more accessible consoles (Finances Online, 2022). The histograms in Figures 1.1 and 1.2 illustrate respectively the revenue of the global video games market and the number of players worldwide in recent years.

However, the arrival of new technologies has increased the complexity of software development and modern games require increasing amounts of manpower and equipment. In particular, the AI systems – e.g. systems that control non-player characters – are still a critical element in the creative process that affects the quality of finished games. These systems play different roles in the development (Figure 1.3) and are one of the most important quality factors for a finished video game. Developers need to answer the demand for complex, expressive non-player character behaviors that are natural while at the same time challenging and enjoyable. The non-player characters must provide the right balance between being beatable and being convincingly competitive. The challenge that these agents offer to the player is the core element in the majority of video games.

On the other hand, AI systems help designers develop better games. A prime example of this is playtesting (Politowski et al., 2022), which plays a crucial role in the production of modern video games. The presence of gameplay issues and bugs can greatly deteriorate the overall player experience and it is therefore crucial they be minimized. However, since modern video games have grown both in size and complexity, thorough coverage is often not feasible using manual human playtesting. For this reason, automated testing approaches have been proposed to mitigate total reliance on human testers by developing AI-based agents to automatically explore large game scenes.

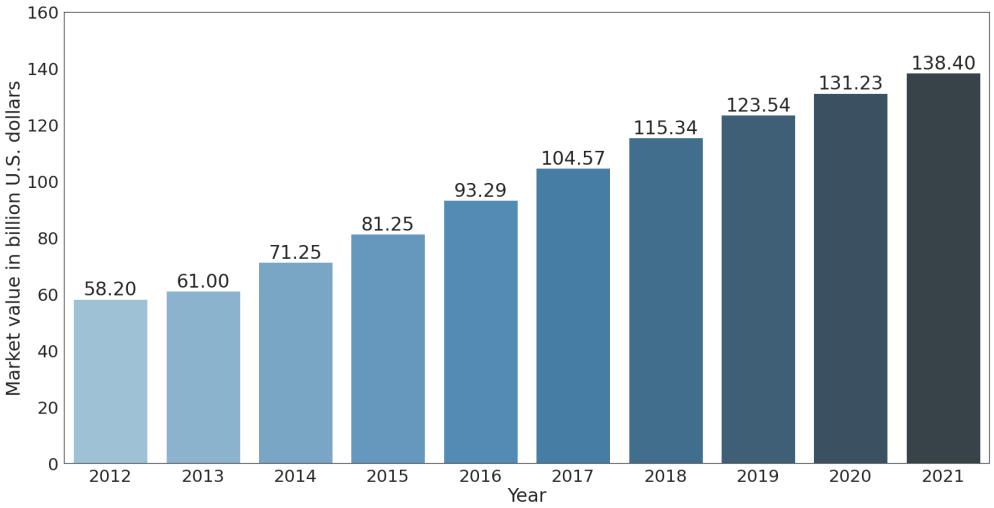


Figure 1.1: Value of the global video games market from 2012 to 2021 in billions of U.S. dollars. Revenues in the gaming industry are based on two major sources, namely hardware, such as consoles, processors, screens, controllers and other accessories, and software – the actual games (Statista, 2022).

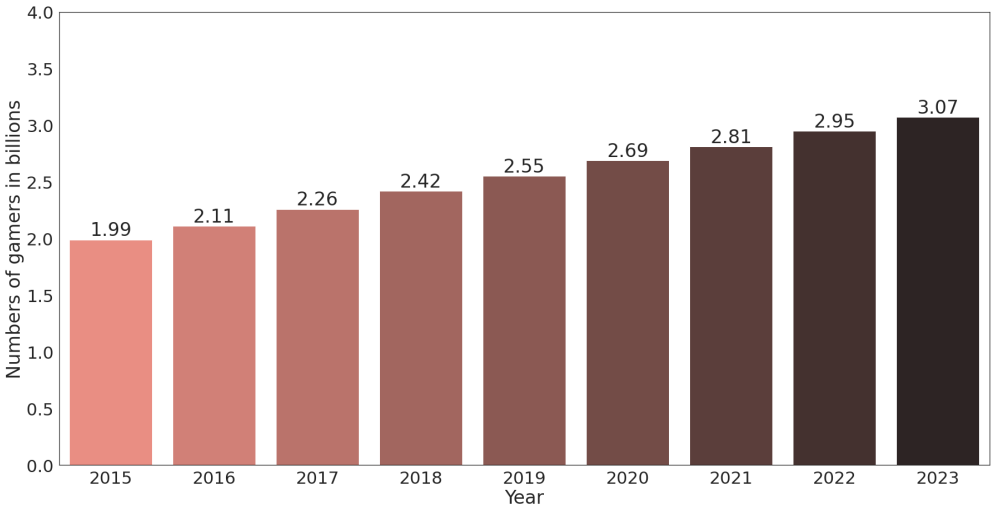


Figure 1.2: Number of active video game players worldwide from 2015 to 2023. There were 2.69 billion video game players worldwide in 2020. The figure will rise to 3.07 billion in 2023 based on a 5.6% year-on-year growth forecast (Finances Online, 2022).

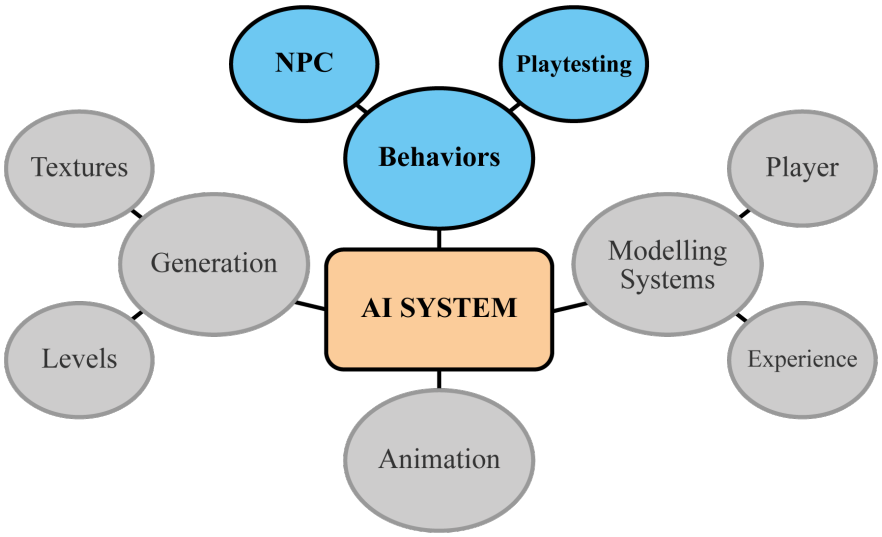


Figure 1.3: Summary of AI applications in video game development. These systems play different roles and have different meanings throughout the development process. In this dissertation we focus on the two main applications: game AI and playtesting.

For both game AI and automated playtesting, many of the current state-of-the-art algorithms are shaped by "classical" systems like finite state machines, behavioral trees, and utility based AI (Yannakakis and Togelius, 2018). These traditional techniques, although tried-and-true, can result in rather predictable and brittle performance in the face of increasing game complexity and changing designs. Moreover, the use of these traditional AI systems typically results in predictable, static and not very convincing behaviors. These techniques are not suitable when faced with the current scope of modern gaming technologies. Scaling these systems to such an open context is still an open problem.

At the same time, recent advances in deep reinforcement learning have shown it is possible to create agents with super-human capabilities for a variety of game environments. The main objective of deep reinforcement learning for games has been training agents to mimic or surpass human players in either classical games like Go (Silver et al., 2016), old-fashioned video games like the Atari 2600 suite (Bellemare et al., 2013), or in modern games like StarCraft II (shown in Figure 1.4), DOTA, and Gran Turismo (Vinyals et al., 2019; OpenAI et al., 2019; Wurman et al., 2022). Recent literature shows it is possible to train agents at super-human levels that can win against professional players. Our ideal objective, however, is not to create new AI systems with super-human capabilities, but rather to create ones that constitute an active part of the game design. We emphasize that our goals are different from those currently covered by most of the literature: for us it is essential to create scalable models that allow agents to have human-like behavior.

Deep reinforcement learning can bring to the creative process of non-player character generation many advantages over classical techniques. It could be possible to have more

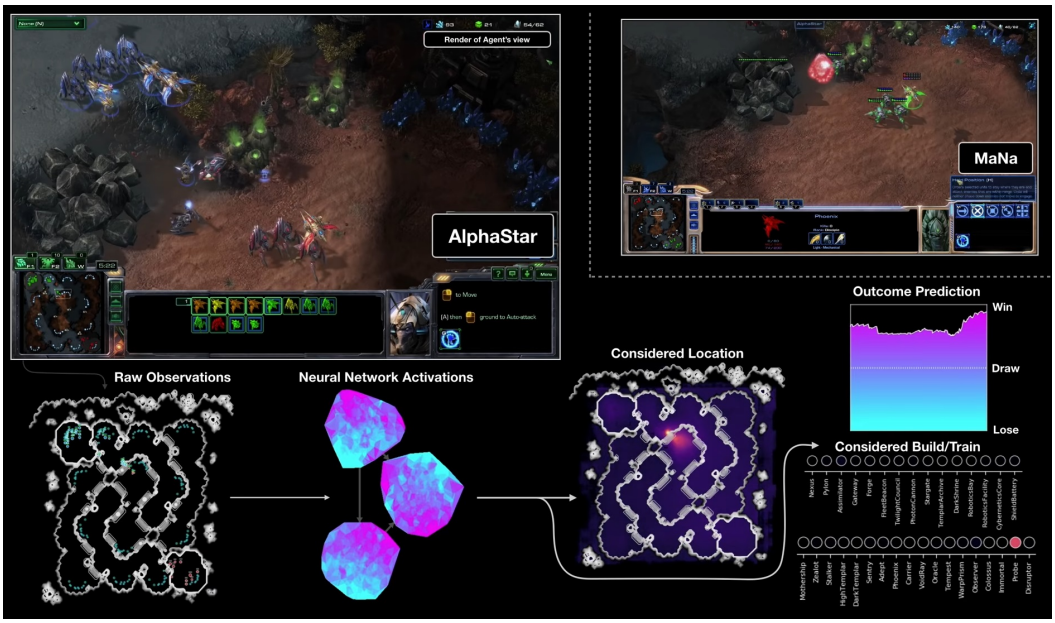


Figure 1.4: A graphical representation of AlphaStar agent processing. This work represents a watershed in the history of deep reinforcement learning applied to video games as it successfully trained super-human agents that could beat professional players at StarCraft II (Vinyals et al., 2019).

various and complex behaviors without the use of hand-crafted agents and unmaintainable scripts that dominate the industry. However, due to many problems yet to be solved, these types of models are far from being widely used in video game production. This dissertation is motivated by the many current problems that exist in order to scale deep reinforcement learning techniques for game design. Our aim is to move from creating super-human agents to developing enjoyable, adaptable, and human-like behaviors. In summary, in this dissertation we will see whether it is possible and how to create active game design tools through deep reinforcement learning, usable not only by machine learning experts but also – and above all – by game designers and developers.

1.1 Dissertation Outline

The rest of the dissertation is organized as follows:

- in Chapter 2 we substantiate the argument that recent deep reinforcement learning techniques are not readily applicable to video game development, although these technologies were created by solving the latter. As part of this, we review and categorize recent literature useful to understand the theory of this dissertation;

- in Chapter 3 we talk about the requirements we identified for creating deep reinforcement learning agents suitable for video game non-player characters. At the same time, we talk about DeepCrawl: one of the most important testbeds used in this dissertation. Most of the experiments and concepts developed in the document are tested and evaluated in this novel open-source environment;
- in Chapter 4 we look at the problem of generalization in deep reinforcement learning. In particular, how we can exploit procedural content generation to train general agents that abstract the rule of the games;
- In Chapter 5 we discuss the problem of engineering a good reward function, and how inverse reinforcement learning can solve the problem by automatically inferring a reward function from expert demonstrations. However, inverse reinforcement learning and procedural content generation do not play well together and we see how to overcome this problem with specific techniques.
- In Chapter 6 we show how to combine different inverse reinforcement learning-based policies. Our aim is to create complex behaviors composed of different sub-policies trained with procedural content generation and inverse reinforcement learning.
- in Chapter 7 we discuss how to combine previous concepts and techniques to obtain autonomous agents for a real video game development use case. We test and validate using new, complex 3D scenario with a novel algorithm that combines imitation learning and curiosity-driven learning.
- in Chapter 8 we show how deep reinforcement learning, and in particular imitation learning, can be used as a real game design tool. We delineate the final requirements that an imitation learning-based tool should have, and support our conclusion with input from professional video game developers. Furthermore, we propose future research directions suitable for maximizing the utility of this dissertation; and
- Finally, in Chapter 9 we summarize our contributions and highlight some higher-level aspects where we see this dissertation contributing to machine learning research applied to video game development.

1.2 Key Contributions

Here we list our key contributions proposed with this dissertation:

- first, we provide an extensive survey and categorisation of problems and their potential solutions related to deep reinforcement learning in video games;
- second, we delineate a list of requirements that both deep reinforcement learning based tools and agents should have to be practically applicable in video game development;

- third, we introduce two different open source environments suitable for anyone wanting to contribute and expand our lines of research;
- fourth, we propose novel algorithms and neural network architectures for solving the cited problems;
- fifth, we conduct extensive and motivated experiments evaluating our algorithms with respect to the current state-of-the-art and we demonstrate their usefulness to video game development; and
- sixth and finally, along with each major contribution we provide links to open source repositories allowing anyone from the research community to replicate the experiments described in this dissertation and to build upon its contributions.

Chapter 2

Background

In Chapter 1 we described the motivations and main contributions of this dissertation. Here we outline the relevant literature and the background theory that will help the reader fully understand our proposed problems, ideas, and solutions. As we have already cited, there exist many different contexts where AI can help video game development, however this dissertation only focuses on agent behaviors. We start by describing basic AI systems commonly used today in the game industry and we then describe the relevant theory used by the algorithms developed in this dissertation, theory related to techniques like deep reinforcement learning, imitation learning, inverse reinforcement learning, and others. Along with the description, we also highlight problems that arise in the classical literature and why these problems inhibit the adoption of such techniques in video games development.

2.1 Definitions

Before getting into details, several terms are used extensively in this dissertation and need to be defined. In this section we describe a list of the most used terms.

First of all, by *Non-Player Character* (NPC) we mean an entity with a behavior that interacts with the game environment – and eventually, but not necessarily, with the player – that is not controlled by humans, but has some form of artificial control. Typically an NPC has the same action space of the player. In any case, the quality of the behavior of the NPC system has the potential to break or elevate the quality of a finished video game.

By *behavior* or *policy* we mean the way in which an NPC behaves in response to a particular situation or stimulus. An NPC policy takes as input a representation of the current observable state of the game environment, and outputs an action based on what it sees.

Finally, by *agent* we mean an NPC that is trained via machine learning, which can be either through deep reinforcement learning, imitation learning, inverse reinforcement learning, or any other techniques not necessarily covered by this dissertation.

2.2 Ad Hoc and Classical Game AI Systems

In this section we briefly describe the most commonly used ad hoc techniques for NPC behaviors. This class contains, among others, finite state machines, behavior trees, and utility-based AI. Most of the subsequent sections are a summary of work from Yannakakis and Togelius (2018).

Finite State Machines

Finite State Machines (FSMs), and their variations, are one of the most commonly used AI techniques for AI behaviors. They represent graph-based algorithms, and are formally defined by three entities:

- A set S of *states*, that represent the information about the task. These are the nodes of the graph. All states in the graph are connected to at least one state and can be reached from at least one other state.
- A set T of *transitions* between nodes that define the conditions in which an NPC changes from one state to another.
- A set A of *actions* that must be executed when the NPC is in a particular state.

We illustrate an example FSM architecture in Figure 2.1. FSMs are nowadays the most used ad hoc systems in video games mainly due to their simplicity to design, implement, and debug. But at the same time FSMs can be extremely complex to implement when faced with large scale video games. Moreover they provide limited flexibility and adaptability – a limitation shared with other ad hoc methods. After they are implemented and debugged for a specific use-case, if not purposely designed they usually require major revisions to be adapted to different games. As a result, FSMs end up implementing very predictable behaviors.

Behavior Trees

Behavior Trees (BTs) are the second most used AI system in the video games industry. BTs make it possible to control NPCs and to define hierarchies of decisions and actions. A BT is formally defined as a directed tree including a set of nodes and edges. The root of a BT is a node without parents. On the other hand, nodes without children are the leaves of this tree. In a basic BT, a non-leaf node can be one of two types:

- *Selector node* - selectors are used when we aim to find and execute the first possible child that can run without failure. A selector node succeeds once any child is performed successfully.
- *Sequence node* - with a sequence node, all children nodes are evaluated sequentially. A sequence node succeeds only if all children are performed successfully.

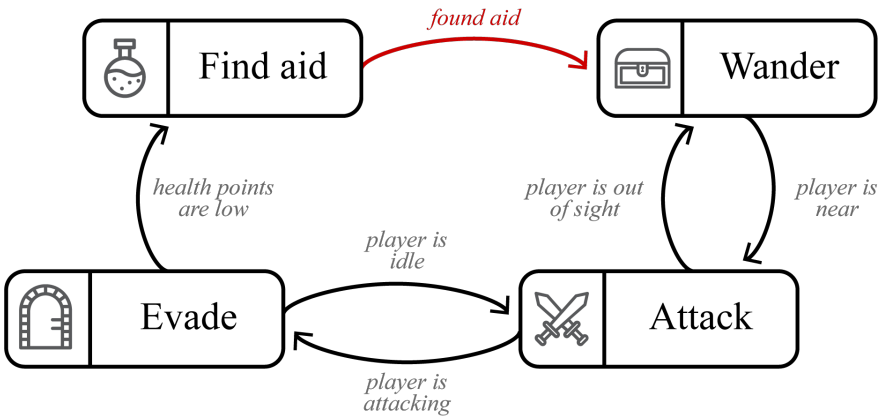


Figure 2.1: Example of finite state machine for a fictional video game.

At the same time, a leaf node can be either an action or a condition node:

- *Action node* - examples of action nodes include playing an animation, changing the state of a character, or any activity that changes the state of the game.
- *Condition node* - a condition node is generally used to test some values. Tests for proximity, testing the state of a character and testing the line of sight are examples of condition nodes. A condition returns success if the condition is met; otherwise, it returns failure.

In addition to the basic nodes, BTs can be extended with decorator and parallel nodes:

- *Decorator node* - a decorator has a single child task. It can be used to increase the conditions for which a child node is executed. A decorator node can be used for filtering which makes a decision to allow a child behavior to run "until fail". A decorator node may also be used to limit the number of runs.
- *Parallel node* - parallel nodes provide concurrency in BTs. A parallel node is used when there are actions that must be run concurrently with others.

An important aspect of BTs is data sharing. To enable complex behaviors, BTs need to share data between each other. The best approach is to decouple execution from data and use an external *blackboard*. A blackboard can store data that can be queried by any task node. Once a BT is instantiated, the root of the tree is ticked at each timestep. Every time a node is executed, it can return one of three result: *success*, *failure*, or *run* if it is still active.

An example BT for a fictional game is shown in Figure 2.2. BTs are in general simpler to design and implement than FSMs, however they suffer from the same drawbacks. Their dynamicity and adaptability is rather low given that they are static knowledge representations.

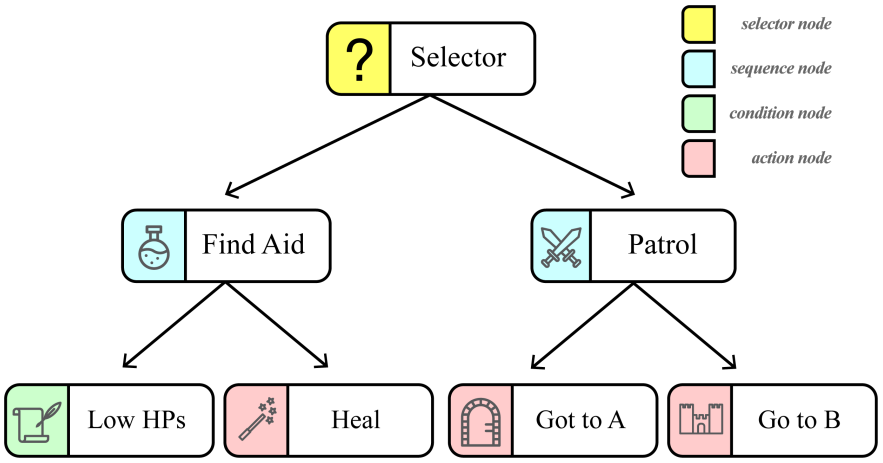


Figure 2.2: An example behavior tree for a fictional video game.

Utility-based AI

Utility-based AI (UBA) is another commonly used approach in the video games industry. It provides a combination of authorial control, reactivity, and believability that can be difficult to match using other ad hoc architectures.

UBAs refer to a class of techniques in which decisions are made on the basis of heuristic functions that represent the relative value of each option under consideration. Thus, utility-based approaches typically have three general steps:

1. build a list of options, which are the choices from which we will get the behaviors;
2. evaluate each option and calculate one or more values that describe how attractive the option is given the current situation; and
3. select an option (or set of options) for execution on the basis of the values calculated in step 2.

A utility can measure anything from observable objective data – e.g. enemy health – to subjective notions such as emotions, mood and threat. The various utilities about possible actions or decisions can be aggregated into linear or non-linear formulas and guide the agent to take decisions based on the aggregated utility. When an NPC must make a decision, the UBA will gather data on the current situation and tell the NPC the best action to take at this exact moment. There are three common approaches to UBA selection:

- *absolute utility* - we evaluate every option and take the one with the highest utility;
- *relative utility* - we select an option at random, using the utility score of each option to define the probability that it will be selected; and

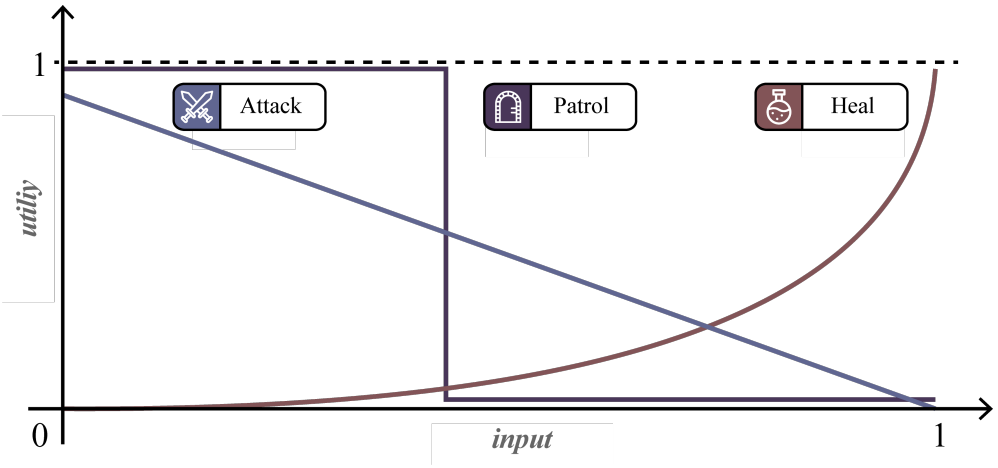


Figure 2.3: Example of utility-based AI for a fictional video game.

- *dual utility reasoner* - we combine both of the previous approaches. It assigns two utility values to each option: a rank (absolute utility) and a weight (relative utility). The rank is used to divide the options into categories, where we only select options that are in the best category. Weight is used to evaluate options within the context of their category.

An example UBA system is shown in Figure 2.3. UBA systems are more dynamic and modular with respect to both FSMs and BTs, and the NPC behaviors are dependent on a number of different factors and thus UBAs allow more complex behaviors. UBA is also extensible as we can easily define new qualitative behaviors and conditions when we need to. It is also more general with respect to FSMs and BTs as the conditions and functions, as well as the entire system, can be used in different use-cases. However, these advantages come at a cost. Defining UBA utility functions requires the designer to express qualitative behaviors and decisions in terms of numbers, which is somewhat unnatural. However, as we will see, this is a problem that exists also with reinforcement learning. Moreover, as for FSMs, UBAs suffer from predictability in their behaviors.

In this section we have briefly detailed the techniques most commonly used to develop NPC behaviors in today's video games, with a focus on the problems that arise using them. In this dissertation we support the hypothesis that DRL can mitigate and overcome problems such as inflexibility, predictability and unsuitability that are intrinsic to the aforementioned algorithms. In the next section we will give an extensive description of the background theory for DRL.

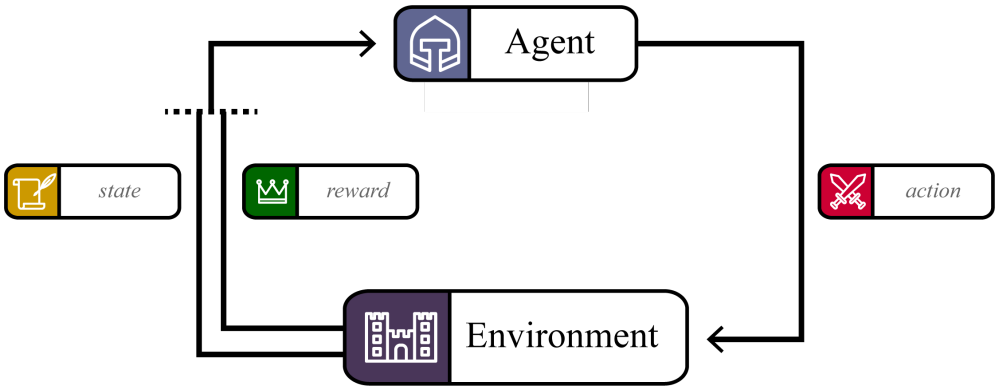


Figure 2.4: A schematic view of a DRL model.

2.3 Survey of Deep Reinforcement Learning Algorithms

In this section we describe the main background theory of deep reinforcement learning and its sub-branches. This section will help the reader better understand all the subsequent algorithms, techniques and experiments we report on in this dissertation. We start by detailing the general deep reinforcement learning problem, and then continue to its sub-branches such as imitation learning, inverse reinforcement learning, and curiosity-driven exploration. The section treats only the theory and literature on DRL useful to fully comprehend the contributions of this dissertation.

Deep Reinforcement Learning

This section is a summary of the work described in Arulkumaran et al. (2017). Before diving into the advent of neural networks to the field, we first introduce the general problem of Reinforcement Learning (RL). The essence of RL is learning through interactions. At each timestep t, an RL agent observes a state s_t and interacts within an environment by taking an action a_t based on s_t. The state is defined as the observable current representation of the environment. The environment then replies to the agent with the next state s_{t+1} which is the consequence of the action made by the agent in the previous timestep, and a reward r_t. The reward evaluates the consequence of the action a_t in the environment. The agent learns its own behavior in response to the reward received. Figure 2.4 gives a schematic view of the agent interaction within the environment.

Formally, RL can be described in terms of a Markov Decision Process (MDP) consisting of:

- a set of states S and a distribution of starting states $p(s_0)$;
- a set of actions A;

- transition dynamics $d(s_{t+1}|s_t, a_t)$ that map a state-action pair at time t onto a distribution of states a time $t+1$;
- a reward function $R : S \times A \times S \to \mathbb{R}$; and
- a discount factor $\gamma \in [0,1]$, where lower values place more emphasis to the more recent rewards.

In addition to this, RL models are based on the use of a policy π:

$$\pi : \mathcal{S} \to p(\mathcal{A} = a|\mathcal{S}), \tag{2.1}$$

which is a function that maps states to a probability distribution over the set $\mathcal{A}$ of actions and determines the behavior of the agent its choice of actions. In the remainder we consider episodic MDP in which the agent interacts with the environment repeatedly in episodes of fixed length T. Each episode is characterized by a sequence of states and actions $\tau = (s_t, a_t)_{t=0}^{T-1}$, usually called a *trajectory* or *rollout*.

The accumulated *return* $G(\tau)$ for a trajectory is given as:

$$G(\tau) = \sum_{t=0}^{T-1} \gamma^t r_{t+1}. \tag{2.2}$$

It is also possible to compute the cumulative return from a particular timestep t – i.e. with $\tau_{s_t \to s_T}$ starting from state s_t to final state s_T, defined as:

$$G_t(\tau_{s_t \to s_T}) = \sum_{k=0}^{T-t-1} \gamma^k r_{k+t+1}. \tag{2.3}$$

The goal of RL is to find the optimal policy π^* that maximizes the expected return from all states:

$$\pi^* = \underset{\pi}{\operatorname{argmax}}\, \mathbb{E}_{s_t \sim d, a_t \sim \pi}[G_t|\pi]. \tag{2.4}$$

A key concept underlying RL is the Markov property, which states that only the current state affects the next one and so any decisions made at step s_t can be based only on s_{t-1}, rather than the entire trajectory $s_0, s_1, ..., s_{t-1}$ up to that point. As Sutton et al. (1998) stated, for the reinforcement learning problem with a reward function R and a set of actions A, the state signal has the Markov property if the environment's response at time $t+1$ is only dependent on the state and action representations at time t. In other words:

$$p(s_t, r_{t-1}|s_{t-1}, a_{t-1}) = p(s_t, r_{t-1}|s_{t-1}, a_{t-1}, ..., s_0, a_0) \tag{2.5}$$

In the next section we describe the two main approaches to solving RL problems: those based on *value functions* and those based on *policy search*.

Value Functions

Value function methods are based on estimating the expected return of being in a given state. The *state-value* function is the expected return when starting in the state s and following the policy π:

$$V^{\pi}(s) = \mathbb{E}_{\pi}[G|s, \pi]. \tag{2.6}$$

The optimal state-value function can be defined as:

$$V^{*}(s) = \max_{\pi} V^{\pi}(s) \quad \forall s \in S. \tag{2.7}$$

Computing the optimal value function requires an environment model. However, in a standard RL setting the set of transition dynamics D are unknown. Therefore, instead of using a state-value function we consider another function called *state-action value function*:

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi}[G|s, a, \pi]. \tag{2.8}$$

The best policy can thus be found choosing action a greedily at every state:

$$\pi(s) = \operatorname*{argmax}_{a} Q^{\pi}(s, a). \tag{2.9}$$

To learn Q^{π} we exploit the Markov property and define the function as a Bellman equation (Bellman, 1954):

$$Q^{\pi}(s_t, a_t) = \mathbb{E}_{s_{t+1}}[r_{t+1} + \gamma Q^{\pi}(s_{t+1}, \pi(s_{t+1}))]. \tag{2.10}$$

This means that we can use the values of our current estimate Q^{π} to improve our estimate:

$$Q^{\pi}(s_t, a_t) \leftarrow Q^{\pi}(s_t, a_t) + \alpha\delta, \tag{2.11}$$

where α is the learning rate and $\delta = Y - Q^{\pi}(s_t, a_t)$ the Temporal Difference (TD) error. This is the foundation of two basic value-based algorithms: Q-learning, which is an *off-policy* algorithm as it updates Q^{π} by transitions not necessarily generated by the current policy; in this case $Y = r_t + \gamma \max_a Q^{\pi}(s_{t+1}, a)$ (Watkins and Dayan, 1992). And State-Action-Reward-State-Action (SARSA), an *on-policy* algorithm that uses transitions generated by the behavioral policy; in this case $Y = r_t + \gamma Q^{\pi}(s_{t+1}, a_{t+1})$ (Rummery and Niranjan, 1994).

Another important value-function based method relies on learning the *advantage* function $A^{\pi}(s, a)$. Unlike producing absolute state-action values, A^{π} instead represents a relative advantage of actions through the simple relationship $A^{\pi} = Q^{\pi} - V^{\pi}$.

Policy Search

Policy search methods directly search for an optimal policy without maintaining a value function model. Typically, a parametrised policy π_{θ} is chosen, whose parameters are updated

to maximise the expected return $G(\tau)$ using gradient-based optimization. The REINFORCE algorithm can be used to compute the gradient of an expectation over a function f of a random variable X with respect to parameters θ (Williams, 1992):

$$\nabla_\theta \mathbb{E}_X[f(X;\theta)] = \mathbb{E}_X[f(X;\theta)\nabla_\theta \log p(X)]. \tag{2.12}$$

As this computation relies on the empirical return of a trajectory, the resulting gradients possess 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:

$$A_t = G_t - b_t, \tag{2.13}$$

where b is the baseline. An example of simple baseline is the average return taken over several episodes.

For the latter problem, it is possible to combine value functions with explicit representation of the policy, resulting in actor-critic methods. The policy represents the actor, that learns by using the feedback of a value function that represents the critic. Actor-critic methods use the value function as a baseline for policy gradients, with the only difference between actor-critic methods and other baseline methods is that actor-critic methods use a learned value function. An example of actor-critic method mostly used in this dissertation is Proximal Policy Optimization.

Proximal Policy Optimization

Proximal Policy Optimization (PPO) is an actor-critic policy search method. The policy is represented by a neural network with parameters θ, updating its weights through backpropagation defining policy gradient loss per timestep as:

$$L^{\text{PG}}(\theta) = \mathbb{E}_t[\log \pi_\theta(a_t|s_t)A_t)], \tag{2.14}$$

where $A_t = G_t - b$ and b is the baseline. PPO is based on clipping the gradient between the current and the previous policy. We define:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}, \tag{2.15}$$

with $r(\theta_{\text{old}}) = 1$. The policy gradient loss for PPO is then defined as:

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t[\min(r_t(\theta)A_t, \text{clip}(r_t(\theta, 1-\epsilon,\ +\epsilon)A_t)], \tag{2.16}$$

where ϵ represents an hyper-parameter of the algorithm. Figure 2.5 shows a single term (i.e., a single timestep t) in L^{CLIP}. During the optimization phase, the method also learns a baseline and maximizes the entropy. We then define the complete joint loss as:

$$L^{\text{PPO}}(\theta) = L^{\text{CLIP}}(\theta) - c_1 L^{\text{VF}} + c_2 S[\pi_{\theta_b}], \tag{2.17}$$

where c_1, c_2 are coefficients, S is an entropy bonus, and L^{VF} is a squared error loss for the value function associated with π_θ.

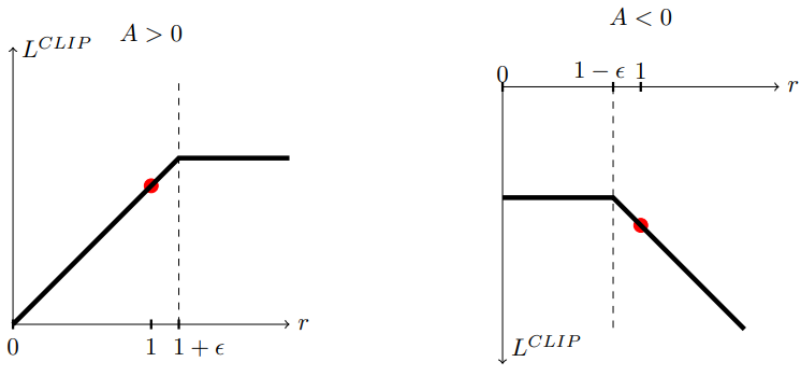


Figure 2.5: Plots showing one term (i.e., a single timestep) of the surrogate function L^{CLIP} as a function of the probability ratio r, for positive advantages (left) and negative advantages (right). The red circle on each plot shows the starting point for the optimization, $r = 1$. The figure is adapted from Schulman et al. (2017).

Imitation and Inverse Reinforcement Learning

One important aspect of DRL is the quality of the reward function. Since the agent goal is to maximize the reward given by the environment, this is one of the most important aspects that determines the final qualitative behavior. However, designing and engineering good hard-coded reward functions is difficult in many domains. In other settings, a badly-designed reward function can lead to agents which receive high rewards in unintended ways. The negative effect of this problem has been demonstrated in works by Randløv and Alstrøm (1998) and Yaeger et al. (1994), and more recent studies such as those by Amodei et al. (2016), Open AI (2016), and Zhang et al. (2021) have further supported these findings.

Inverse Reinforcement Learning (IRL) algorithms attempt to infer a reward function from expert demonstrations (Ng and Russell, 2000). This reward function can then be used to train agents which thus learn to mimic the policy implicitly executed by human experts. IRL offers the promise of solving many of the problems entailed by reward engineering. These approaches have achieved good performance both on continuous control tasks (Fu et al., 2018; Finn et al., 2016b) and on Atari games (Tucker et al., 2018).

Similarly to IRL, Imitation Learning (IL) is a specific case of IRL that aims to directly find a policy that mimics the expert behavior from a dataset of demonstrations, instead of inferring a reward function. Standard approaches are based on Behavioral Cloning that mainly use supervised learning (Bain and Sammut, 1995; Syed and Schapire, 2008; Ross et al., 2011; Reddy et al., 2019; Cai et al., 2019; Knox and Stone, 2009).

Formally speaking, IRL attempts to infer the reward function $R(s,a)$ given a set of demonstrations $E = (\tau_0, \tau_1, ..., \tau_N)$, where τ_i is a trajectory. We assume that E comes from an optimal policy $\pi^*(a|s)$. We can interpret the IRL problem as solving the maximum

likelihood problem:

$$\max_{\theta} \mathbb{E}_{\tau \sim E}[\log p_\theta(\tau)], \tag{2.18}$$

where $p_\theta(\tau) \propto p(s_0) \prod_{t=0}^{T} p(s_{t+1}|s_t, a_t) e^{\gamma^t r_\theta(s_t, a_t)}$ parameterises the reward function $r_\theta(s, a)$ but fixes the dynamics and initial state distribution to that of the MDP. In the case of IL, since the aim is to directly find a policy that mimics the expert behavior, our aim is to update a parameterised policy π_θ to maximize its likelihood based on the expert dataset. Equation 2.18 thus becomes:

$$\max_{\theta} \mathbb{E}_{\tau \sim D}[\log \pi_\theta(\tau)], \tag{2.19}$$

and IRL is reduced to a supervised learning problem.

Generative Adversarial Imitation Learning

Ho and Ermon (2016) propose to cast optimization of Equation 2.19 as a Generative Adversarial Network (GAN) (Goodfellow et al., 2014) optimization problem. They derived an IL algorithm called Generative Adversarial Imitation Learning (GAIL). The objective of the discriminator D is to distinguish between the distribution of data generated by the policy G and the expert distribution. In this case, the policy G is the generator. When D cannot distinguish data generated by G from the true data, it means that G can successfully match the expert behavior. The discriminator D is trained by maximizing:

$$\mathbb{E}_{\pi}[\log(D(s, a))] + \mathbb{E}_{\pi_{\mathrm{E}}}[\log(1 - D(s, a))]. \tag{2.20}$$

The policy is instead optimized maximizing the cost:

$$c(s, a) = \log D(s, a). \tag{2.21}$$

However, GAIL does not impose any special structure on the discriminator. This means that it does not recover a proper reward function, but rather directly recovers a policy that mimics the expert behavior.

Adversarial Inverse Reinforcement Learning

Fu et al. (2018) propose a similar approach to GAIL. They add structure to the discriminator in order to recover a reward function. This IRL approach is called Adversarial Inverse Reinforcement Learning (AIRL). AIRL takes inspiration from GANs by alternating between training a discriminator $D_\theta(s, a)$ to distinguish between policy and expert trajectories and optimizing the trajectory-generating policy $\pi(a|s)$. The AIRL discriminator is given by:

$$D_\theta(s, a) = \frac{\exp\{f_{\theta,\omega}(s, a, s')\}}{\exp\{f_{\theta,\omega}(s, a, s')\} + \pi(a|s)}, \tag{2.22}$$

where $\pi(a|s)$ is the generator policy and $f_{\theta,\omega}(s,a,s') = r_\theta(s,a) + \gamma\phi_\omega(s') - \phi_\omega(s)$ is a potential base reward function which combines a reward function approximator $r(s,a)$ and a reward shaping term ϕ_ω. For deterministic environment dynamics, the AIRL authors show that there is a state-only reward approximator $f^*(s,a,s') = r^*(s) + \gamma V^*(s') - V^*(s) = A^*(s,a)$ which is invariant to transition dynamics and hence *"disentangled"*.

The objective of the discriminator is to minimize the cross-entropy between expert demonstrations $\tau^E = (s_0^E, a_0^E, \dots)$ and generated trajectories $\tau^\pi = (s_0^\pi, a_0^\pi, \dots)$:

$$\begin{aligned}\mathcal{L}(\theta) = &- E_{\tau^E}\left[\sum_{t=0}^{T} \log D_\theta(s_t^E, a_t^E)\right] \\ &- E_{\tau^\pi \sim \pi}\left[\sum_{t=0}^{T} \log\left(1 - D_\theta(s_t^\pi, a_t^\pi)\right)\right].\end{aligned} \tag{2.23}$$

The authors show that, at optimality, $f^*(s,a) = \log \pi^*(a|s) = A^*(s,a)$, which is the advantage function of the optimal policy. The learned reward function is based on the discriminator:

$$\hat{r}(s,a) = \log(D_\theta(s,a)) - \log(1 - D_\theta(s,a)), \tag{2.24}$$

and the generator policy is optimized with respect to a maximum entropy objective (using Equations 2.24 and 2.22):

$$\begin{aligned}J(\pi) &= E_{\tau\sim\pi}\left[\sum_{t=0}^{T} \hat{r}_t(s_t, a_t)\right] \\ &= E_{\tau\sim\pi}\left[\sum_{t=0}^{T} f_\theta(s_t, a_t) - \log(\pi(a_t|s_t))\right].\end{aligned} \tag{2.25}$$

Intrinsic Motivation

One of the greatest difficulties in DRL is the fundamental dilemma between *exploitation vs exploration*. By exploitation we usually mean that the agent exploits the optimal action in order to make progress, while by exploration the agent tries non-optimal actions in order to explore the environment and reduce the risk of getting stuck in a local maximum.

There are several ways to perform exploration in DRL, and most of them rely on intrinsic motivation. Intrinsic motivation aims to encourage agents to explore the states of the environment. Count-based exploration is a natural way to explore, although for high-dimensional state spaces it can be infeasible (Strehl and Littman, 2008; Bellemare et al., 2016).

A classical way to perform count-based exploration is to discretize the state space with a hash function $h : S \to \mathbb{Z}$ and add an exploration bonus to the reward function, defined as:

$$r^+(s) = \frac{\beta}{\sqrt{n(h(s))}}, \tag{2.26}$$

where β is the bonus coefficient. Initially, the counts $n(s)$ are set to zero for the whole range of h. For every state s_t encountered at timestep t, $n(h(s_t))$ is increased by one. An alternative version is to use a state-action count $n(h(s), a)$ instead of using only the state $n(h(s))$.

Several other exploration algorithms have been proposed to manage complex and high-dimensional state spaces. Techniques like curiosity and random network distillation rely on errors in predicting dynamics to push agents to explore never or less-encountered states during training (Pathak et al., 2017; Burda et al., 2018). Stadie et al. (2015) proposed to assign exploration bonuses from a concurrently learned model of the system dynamics, while Sun et al. (2011) showed that the optimal exploration strategy can be effectively approximated by solving a sequence of dynamic programming problems. The interested reader should consult Aubret et al. (2019) work for a detailed survey of the state-of-the-art in intrinsic motivation. An intrinsic motivation algorithm used in this dissertation is Random Network Distillation.

Random Network Distillation

The already cited count-based exploration can be extremely difficult to use with high-dimensional state spaces. The Random Network Distillation (RND) algorithm by Burda et al. (2018) offers a solution to this problem. In general, like count-based exploration methods, RND gives an intrinsic reward that is higher for novel and less-encountered states.

The RND algorithm uses two neural networks: a fixed and randomly initialized target network $\hat{\phi}$ which establishes the prediction problem, and a predictor network ϕ trained on data collected by the agent. The predictor network is trained by gradient descent to minimize the expected MSE:

$$\mathcal{L}^{\text{RND}} = (\hat{\phi}(s) - \phi(s))^2, \tag{2.27}$$

with respect to its parameters. This process distills a randomly initialized neural network into a trained one. The reward r_c is the same MSE used to train the network:

$$r_c(s_t) = (\hat{\phi}(s_{t+1}) - \phi(s_{t+1}))^2. \tag{2.28}$$

The more a state is visited by agents, the closer the output of the predictor network will be to that of the target network for that particular state, lowering the prediction error and thus the reward signal for exploration. States encountered many times will produce low reward values, while for states encountered less frequently the predictor will not be able to perfectly replicate the target, increasing the reward signal and guiding agents toward undiscovered paths.

2.4 Limitations of Deep Reinforcement Learning

Up to this point we have mainly focused on the underlying theory upon which the contributions of this dissertation are built. DRL has been used recently to train agents with

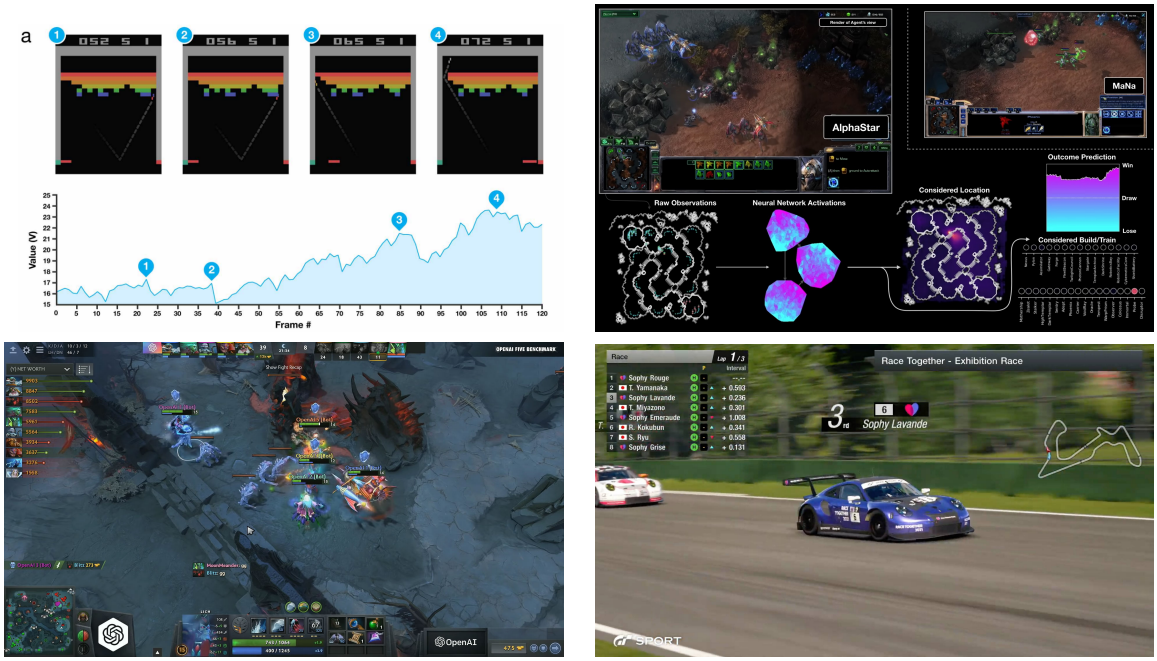


Figure 2.6: Examples of super-human agents train in classical games as well as popular video games. Starting from the upper left image, we have the DQN algorithm (Mnih et al., 2015), AlphaStar (Vinyals et al., 2019), OpenAI Five (OpenAI et al., 2019), and GT Sophy (Wurman et al., 2022).

super-human capabilities in many video games. Starting from the Deep-Q Network (DQN) algorithm used by Mnih et al. (2015) to train agents with super-human capabilities to play games from the Atari 2600 suite (Bellemare et al., 2013), there have been many successful examples of using video games as testbeds for DRL. Vinyals et al. (2019) trained agents with that can beat professional players at StarCraft II by combining DRL and IL, while OpenAI (2019) developed a complex DRL system to train five agents to beat professional players at DOTA 2. More recently, Wurman et al. (2022) trained agents with super-human performance that can play Gran Turismo 7. In Figure 2.6 we summarize these examples.

However, the recent and highly publicized successes of DRL in mimicking or even surpassing human-level play in video games have not yet been translated into effective tools for use in developing game AI. In this dissertation we show that DRL can be used to develop credible agents that are prior-free and that can add variety to gameplay. We believe that DRL can help developers build more varied, complex and useful agents, overcoming the critical aspects of classical ad hoc video game AI techniques. The surprising performance achieved by the research community might make it seem that DRL can solve most of these critical aspects, but we argue that there are still many issues related to the inappropriate application of DRL techniques used as game design tools. Here we highlight some of the main issues preventing broad application of the aforementioned techniques for video game development.

Generalization. Even though DRL algorithms can process high-dimensional state spaces, their ability to generalize is somewhat low with respect to other machine learning algorithms. This is mainly due to the tendency of DRL policies to overfit to the environment on which they are trained. An example is GT Sophy (Wurman et al., 2022), where the authors train a different model for each track of the racing game Gran Turismo. However, this is problematic from a game development perspective. During development, the game, the environments, the assets, and in general all components that define a game can change on a day-to-day basis. Developers can not afford to train new agents every time a design change is made.

Reward Function. As we already cited, engineering a good qualitative reward function is very difficult, even for DRL experts. Translating a qualitative behavior to a mathematical function is somewhat unnatural. The difficulty increases if the user of a plausible DRL tool is a game designer and/or developer, that typically is not an expert in machine learning. IL and IRL can mitigate the problem, but they require very many expert demonstrations. Moreover, the majority of IL and IRL techniques also suffer from overfitting, especially when the environment changes many times.

Behavior Quality and Novel Testbeds. Video games do not need super-human agents. Whether we are talking about game AI or agents for different aspects of video game development like playtesting, having super-human agents is as counterproductive as having random NPCs. NPC agents have different requirements that we must formally define to advance the state-of-the-art in this direction. For the same reason, we need testbeds different from those in the literature that focus only performance.

Interpretability. Whether we are talking about game AI or playtesting, we need agents with meaningful behavior. DRL agents can learn very high-quality strategies, but these models exhibit an important negative characteristic: a performance-transparency trade-off. Understanding the reasoning behind their behavior becomes a necessity when these results drive design decisions.

Usability. Works like AlphaStar and OpenAI Five (Vinyals et al., 2019; OpenAI et al., 2019) are very complex systems consisting of many interrelated components. Such complicated systems require a long time not only to define, but also to run. Both the number of weights in such neural networks and the number of hyperparameters in such systems are huge. This limits the usability of the system for two main reasons: on one hand developers can not wait for days to train agents as they must continue with development, and on the other hand they must ship agents to low power devices such as mobile phones or old-generation consoles.

In Figure 2.7 we give a summary of the aforementioned challenges. In the rest of this dissertation we describe a number of solutions aimed at mitigating specific aspects of these challenges.

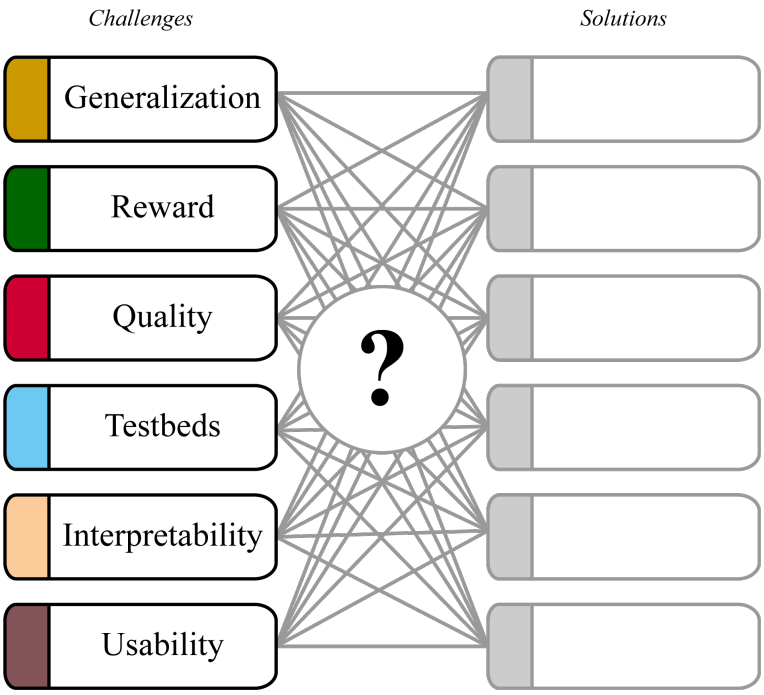


Figure 2.7: List of challenges that interfere with the wide application of DRL in video game development. In this dissertation we try to bridge the gap between challenges and solutions.

Chapter 3

Deep Reinforcement Learning for Turn-based Strategy Games[†]

In Chapter 2 we outlined the relevant literature and background theory upon which the contributions of this dissertation are built. More importantly, we highlighted problems and challenges that limit the wide spread of deep reinforcement learning in video game development. In this chapter we introduce DeepCrawl, a fully-playable Roguelike prototype for iOS and Android in which all agents are controlled by policy networks trained using DRL. Our aim is to understand whether recent advances in such techniques can be used to develop convincing behavioral models for non-player characters in videogames. We begin with an analysis of requirements that such an AI system should satisfy in order to be practically applicable in video game development, and identify the elements of the deep reinforcement learning model used in the DeepCrawl prototype. The successes and limitations of DeepCrawl are documented through a series of playability tests performed on the final game. This chapter offers an initial insight into innovative new avenues for the development of behaviors for non-player characters in video games, as they offer the potential to overcome critical issues with classical approaches.

3.1 Introduction

Technological advances in gaming industry have resulted in the production of increasingly complex and immersive gaming environments. However, the creation of AI systems that control Non-Player Characters (NPCs) is still a critical element in the creative process that affects the quality of finished games. This problem is often due to the use of classical AI techniques that result in predictable, static, and not very convincing NPC strategies. Reinforcement Learning (RL) can help overcome these issues providing an efficient and

[†]Portions of this chapter appeared in: A. Sestini, A. Kuhnle, and A. D. Bagdanov, "DeepCrawl: Deep Reinforcement Learning for Turn-Based Strategy Games", published in the *Experimental AI in Games (EXAG) Workshop at Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 2019.

practical way to define NPC behaviors, but its real application in production processes has issues that can be orthogonal to those considered to date in the academic field: how can we improve the gaming experience? How can we build credible and enjoyable agents? How can RL improve over classical algorithms for game AI? How can we build an efficient ML model that is also usable on all platforms, including mobile systems?

At the same time, recent advances in Deep Reinforcement Learning (DRL) have shown it is possible to train agents with super-human skills able to solve a variety of environments. However the main objective of DRL of this type is training agents to surpass human players in competitive play in classical games like Go (Silver et al., 2016) and video games like DOTA 2 (OpenAI et al., 2019). The resulting, however, agents clearly run the risk of being too strong, of exhibiting artificial behavior, and in the end not being a *fun* gameplay element.

Video games have become an integral part of our entertainment experience, and our goal in this chapter is to demonstrate that DRL techniques can be used as an effective *game design* tool for learning compelling and convincing NPC behaviors that are natural, though not superhuman, while at the same time provide challenging and enjoyable gameplay experience. As a testbed for this work we developed the DeepCrawl Roguelike prototype, which is a turn-based strategy game in which the player must seek to overcome NPC opponents and NPC agents must learn to prevent the player from succeeding. We emphasize that our goals are different than those of AlphaGo and similar DRL systems applied to gameplay: for us it is essential to limit agents so they are beatable, while at the same time training them to be convincingly competitive. In the end, playing against superhuman opponents is not *fun*, and neither is playing against *trivial* ones. This is the balance we try to strike.

With this chapter we propose some requirements that a Machine Learning (ML) system should satisfy in order to be practically applied in videogame production as an active part of game design. We also propose an efficient DRL system that is able to create a variety of NPC behaviors for Roguelike games only by changing some parameters in the training set-up; moreover the reduced complexity of the system and the particular net architecture make it easy to use and to deploy to systems like mobile devices. Finally we introduce a new game prototype, tested with a series of playability tests, that can be a future benchmark for DRL for videogame production.

3.2 Related Work

Here we review the recent works most related to this chapter.

Game AI. Game AI has been a critical element in video game production since the dawn of this industry; agents have to be more and more realistic and intelligent to provide the right challenge and level of enjoyment to the user. However, as stated in Chapter 2, as game environments have grown in complexity over the years, scaling traditional AI solutions like Behavior Trees (BT) and finite state machines (FSM) for such complex contexts is an open problem Yannakakis and Togelius (2018).

Reinforcement Learning. Reinforcement Learning (RL) (Sutton et al., 1998) is directly concerned with the interaction of agents in an environment. RL methods have been widely used in many disciplines, such as robotics and operational research, and games. The breakthrough of applying DRL by DeepMind in 2015 (Mnih et al., 2015) brought techniques from supervised deep learning (such as image classification and convolutional neural networks) to overcome core problems of classical RL. This combination of RL and neural networks has led to successful application in games. In the last few years several researchers have improved upon the results obtained by DeepMind. For instance, OpenAI researchers showed that with an Actor Critic (Konda and Tsitsiklis, 2003) algorithm such as Proximal Policy Optimization (PPO) (Schulman et al., 2017) it is possible to train agents to superhuman levels that can win against professional players in complex and competitive games such as DOTA 2 (OpenAI et al., 2019). For a detailed description of these examples, see Chapter 2.

As already discussed in the introduction, most of the works in DRL aim to build agents replacing human players either in old-fashioned games like Go or chess (Silver et al., 2016; Asperti et al., 2018) or in more recent games such as Doom or new mobiles games (OpenAI et al., 2019; Vinyals et al., 2019; Oh et al., 2019; Kempka et al., 2016; Juliani et al., 2019). Our objective, however, is not to create a new AI system with superhuman capabilities, but rather to create ones that constitute an active part of the game design.

Reinforcement Learning Environments. There has been a significant amount of work in the field of open-source reinforcement learning environments in the past decade. Prior environments, such as the classic OpenAI Gym (Brockman et al., 2016) and the DeepMind Lab (Beattie et al., 2016), have provided valuable platforms for researchers to test and compare the performance of various reinforcement learning algorithms. However, these environments have mainly focused on providing simple, low-dimensional tasks such as balance beam walking and cart-pole balancing. In recent years, there has been a growing interest in using game-based environments to study and train autonomous agents. Popular open-source examples are MineRL (Guss et al., 2019), Nethack (Küttler et al., 2020) and Obstacle Tower (Juliani et al., 2019). However, the focus of these environments is often to replace the human player and create the super-human agent, which can result in NPC agents that behave in unrealistic or "cheaty" way. Moreover, environments like Nethack may not provide the necessary level of realism or control to effectively study NPC behavior.

The environment presented in this paper is a roguelike game specifically designed for studying and training NPC agents via deep reinforcement learning. It offers a realistic and controllable environment for studying NPC behavior, while also avoiding the frustrating experience of playing against a super-human agent. This makes it a unique and valuable tool for research on NPC behavior.

Figure 3.1: Screenshot of the final version of DeepCrawl. Each level of the game consists of one or more rooms, in each of which there is one or more agents. To clear a level the player must fight and win against all the enemies in the dungeon. The game is won if the player completes ten dungeon levels.

3.3 Game Design and Desiderata

In this section we describe the main gameplay mechanics of DeepCrawl and the requirements that the system should satisfy in order to be used in a playable product. The DeepCrawl prototype is a fully playable Roguelike game and can be downloaded for Android and iOS *. In Figure 3.1 we give a screenshot of the final game.

Gameplay mechanics

We decided to build a *Roguelike* as there are several aspects of this type of games that make them an interesting testbed for DRL as a game design tool, such as the *procedurally created environment*, the *turn-based* system and the *non-modal* characteristic that makes available every action for actors regardless the level of the game. In fact, Roguelikes are often used as a proving ground game genre specifically because they involve a limited set of game mechanics, which allows game designers to concentrate on emergent complexity of gameplay as a combination of the relatively simple set of possibilities given to the player.

*Android Play: https://play.google.com/store/apps/details?id=com.miccgame.alessandrosestini&hl=en_CA&gl=US, App Store: https://apps.apple.com/it/app/deepcrawl/id1461452000

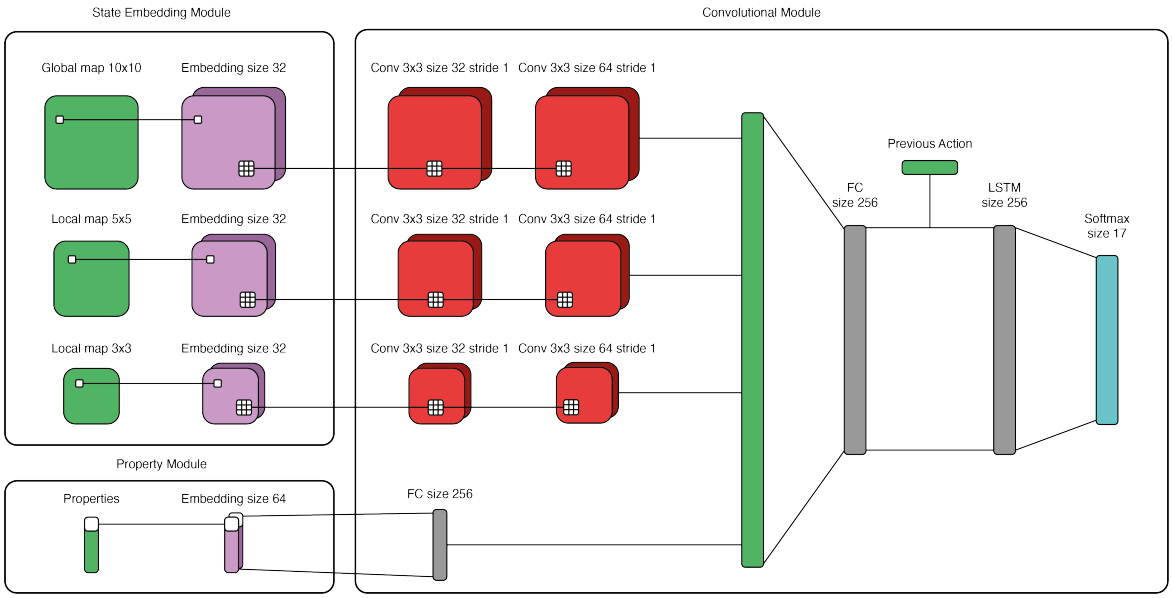


Figure 3.2: The policy network used for NPCs in DeepCrawl. See Section 3.4 for a detailed description.

The primary gameplay mechanics in DeepCrawl are defined in terms of several distinct, yet interrelated elements.

Actors. Success and failure in DeepCrawl is based on direct competition between the player and one or more agents guided by a deep policy network trained using DRL. Player and agents act in procedurally generated rooms, and player and agents have exactly the same characteristics, can perform the same actions, and have access to the same information about the world around them.

Environment. The environment visible at any instant in time is represented by a random grid with maximum size of 10×10 tiles. Each tile can contain either an agent or player, an impassible object, or collectible loot. Loot can be of three types: melee weapons, ranged weapons, or potions. Moreover, player and agent are aware of a fixed number of personal characteristics such as HP, ATK, DEX, and DEF. Agents and player are also aware of their inventory in which loot found on the map is collected. The inventory can contain at most one object per type at a time, and a new collected item replaces the previous one. The whole dungeon is composed of multiple rooms, where in each of them there are one or more enemies. The range of action of each NPC agent is limited to the room where it spawned, while the player is free to move from room to room.

Action space. Each character can perform 17 different discrete actions:

- **8 movement actions** in the horizontal, vertical and diagonal directions; if the movement ends in a tile containing another agent or player, the actor will perform a melee attack:

this type of assault deals random damage based on the melee weapon equipped, the ATK of the attacker, and the DEF of the defender;

- **1 use potion action**, which is the only action that does not end the actor's turn. DeepCrawl has two buff potions available, one that increases ATK and DEF for a fixed number of turns, and heal potion that heals a fixed number of HP; and
- **8 ranged attack actions**, one for each possible direction. If there is another actor in selected direction, a ranged attack is performed using the currently equipped ranged weapon. The attack deals a random amount of damage based on the ranged weapon equipped, the DEX of the attacker, and the DEF of the defender.

Desiderata

As defined above, our goals were to create a playable game, and in order to do this the game must be enjoyable from the player's perspective. Therefore, in the design phase of this work it was fundamental to define the requirements that AI systems controlling NPCs should satisfy in order to be generally applicable in videogame design:

Credibility. NPCs must be *credible*, that is they should act in ways that are predictable and that can be interpreted as intelligent. The agents must offer the right challenge to the player and should not make counterintuitive moves. The user should not notice that he is playing against an AI.

Imperfection. At the same time, the agents must be *imperfect* because a superhuman agent is not suitable in a playable product. In early experiments we realized that it was relatively easy to train *unbeatable* agents that were, frankly, no fun to play against. It is important that the player always have the chance to win the game, and thus agents must be beatable.

Prior-free. Enemy agents must be *prior-free* in that developers do not have to manually specify strategies – neither by hard-coding nor by carefully crafting specific rewards – specific to the game context prior the training. The system should extrapolate strategies independently through the trial-and-error mechanism of DRL. Moreover, this prior-free system should generalize to other Roguelike games sharing the same general gameplay mechanics.

Variety. It is necessary to have a certain level of *variety* in the gameplay dynamics. Thus, it is necessary to support multiple agents during play, each having different behaviors. The system must provide simple techniques to allow agents to extrapolate different strategies in the training phase.

3.4 Proposed Model

Here we describe in detail the main elements of the DRL model that controls the agent in DeepCrawl, with particular attention to the neural network architecture and the reward function.

Policy network and state representation

We used a policy-based method to learn the best strategy for agents controlling NPCs. For these methods, the network must approximate the best policy. The neural network architecture we used to model the policy for NPC behavior is shown in Figure 3.2. The network consists of four input branches:

- the first branch takes as input the whole map of size 10×10, with the discrete map contents encoded as integers:
  - 0 = impassable tile or other agent;
  - 1 = empty tile;
  - 2 = agent;
  - 3 = player; and
  - 4+ = collectible items.

  This input layer is then followed by an embedding layer which transforms the $10 \times 10 \times 1$ integer input array into a continuous representation of size $10 \times 10 \times 32$, a convolutional layer with 32 filters of size 3×3, and another 3×3 convolutional layer with 64 filters.
- The second branch takes as input a local map with size 5×5 centered around the agent's position. The map encoding is the same as for the first branch and an embedding layer is followed by convolutional layers with the same structure as the previous ones.
- The third branch is structured like the second, but with a local map of size 3×3.
- The final branch takes as input an array of 11 discrete values containing information about the agent and the player:
  - agent HP in the range [0,20];
  - the potion currently in the agent's inventory;
  - the melee weapon currently in the agent's inventory;
  - ranged weapon in the agent's inventory;
  - a value indicating whether the agent has an active buff;
  - a value indicating whether the agent can perform a ranged attack and in which direction;
  - player HP in the range [0,20];
  - the potion currently in the player's inventory;
  - the melee weapon in the player's inventory;
  - the ranged weapon in the player's inventory; and
  - a value indicating whether the player has an active buff.

This layer is followed by an embedding layer of size 64 and a Fully-Connected (FC) layer of size 256.

The outputs of all branches are concatenated to form a single vector which is passed through an FC layer of size 256; we add a *one-hot* representation of the action taken at the previous step, and the resulting vector is passed through an LSTM layer. The final output of the net is a probability distribution over the action space (like all policy-based methods such as PPO).

With this model we also propose some novel solutions that have improved the quality of agents behavior, overcoming some of the challenges of DRL in real applications:

- **Global vs local view**: we discovered that the use of both global and local map representations improves the score achieved by the agent and the overall quality of its behavior. The combination of the two representations helps the agent evaluate both the general situation of the environment and the local details close to it; we use only two levels of local maps, but for a more complex situation game developers could potentially use more views at different scales;
- **Embeddings**: the embedding layers make it possible for the network to learn continuous vector representations for the meaning of and differences between integer inputs. Of particular note is the embedding of the last branch of the network, whose inputs have their own ranges distinct from each other, which helps the agent distinguish the contents of two equal but semantically different integer values. For instance:
  - agent HP $\in [0, 20]$;
  - potion $\in [21, 23]$;
  - etc.
- **Sampling the output**: instead of taking the action with the highest probability, we sample the output, thus randomly taking *one of the most probable actions*. This behavior lets the agent make some mistakes during its interaction with the environment, guaranteeing *imperfection* and avoids the agent getting stuck in repetitive loops of the same moves.

Reward function

When defining the reward function for training policy networks, to satisfy the *prior-free* requirement we used an extremely sparse function:

$$R(t) = -0.01 + \begin{cases} -0.1 & \text{for an impossible move} \\ +10.0 * \text{HP} & \text{for the win} \end{cases}, \tag{3.1}$$

where HP refers to the normalized agent HPs remaining at the moment of defeating an opponent. This factor helps the system to learn as fast as possible the importance of HP: winning with as many HP as possible is the implicit goal of a general Roguelike game.

Network and training complexity

All training was done on an NVIDIA 1050ti GPU with 4GB of RAM. On this modest GPU configuration, complete training of one agent takes about two days. However, the reduced size of our policy networks (only about 5.5M parameters in the policy and baseline networks combined) allowed us to train multiple agents in parallel. Finally, the trained system needs about 12MB to be stored. We remind though that more agents of the same type can use the same model: therefore this system does not scale with the number of enemies, but only with the number of different classes.

3.5 Implementation

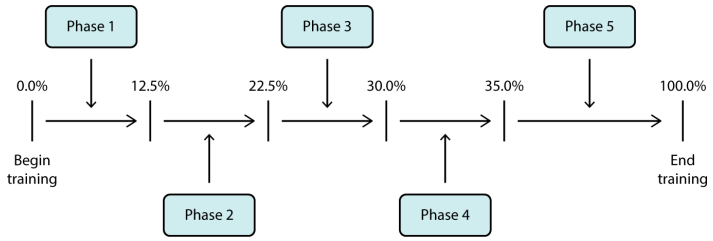
In this chapter we describe how the DeepCrawl policy networks were trained. We used two type of technologies to build both the DRL system and the game:

Tensorforce. Tensorforce (Kuhnle et al., 2017; Schaarschmidt et al., 2018) is an open-source DRL framework built on top of Google's TensorFlow framework, with an emphasis on modular, flexible library design and straightforward usability for applications in research and practice. Tensorforce is agnostic to the application context or simulation, but offers an expressive state- and action-space specification API. In particular, it supports and facilitates working with multiple state components, like our global/local map plus property vector, via a generic network configuration interface which is not restricted to simple sequential architectures only. Moreover, the fact that Tensorforce implements the entire RL logic, including control flow, in portable TensorFlow computation graphs makes it possible to export and deploy the model in other programming languages, like C# as described in the next section.

Unity and Unity ML-Agents. The DeepCrawl prototype was developed with Unity (Unity, 2019) and Unity Machine Learning Agents Toolkit (Unity ML-Agents) (Juliani et al., 2018), that is an open source plugin available for the game engine that enables games and simulations to serve as environments for training intelligent agents. This framework allows external Python libraries to interact with the game code and provides the ability to use pre-trained graph directly within the game build thanks to the TensorFlowSharp plugin (Icaza, 2019).

Training setup

To create agents able to manage all possible situations that can occur when playing against a human player, a certain degress of randomness is required in the procedurally-generated



	agent HPs	enemy HP	loot quantity
Phase 1	20	1	20%
Phase 2	[5, 20]	10	20%
Phase 3	[5, 20]	[10, 20]	20%
Phase 4	[5, 20]	[10, 20]	[10%, 20%]
Phase 5	[5, 20]	[10, 20]	[5%, 20%]

Figure 3.3: Curriculum used for training all agents. Left: a training timeline showing how long each curriculum phase lasts as a percentage of total training steps. Right: the changing generation parameters of all the curriculum phases. The numbers in parentheses refer to a random number in that range; the loot quantity depends on the number of empty tiles in the room (e.g. 20% loot quantity indicates that the number of items in the room is equal to the 20% of empty tiles in that).

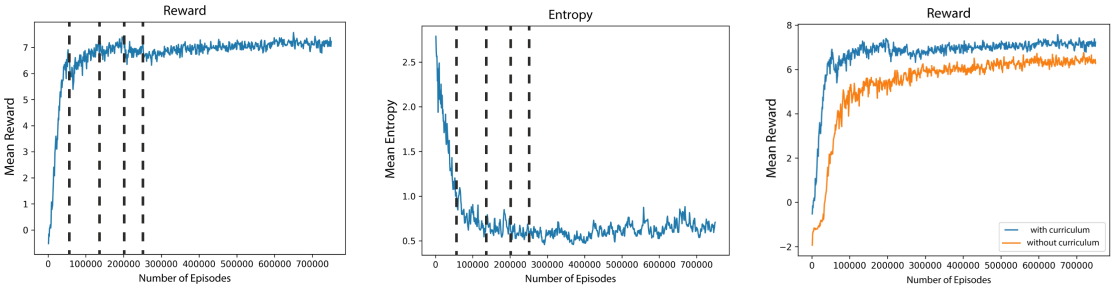
environments: the shape and the orientation of the map, as well as the number of impassable and collectible objects and their positions are random; the initial position of the player and the agent is random; and the initial equipment of both the agent and the player is random.

In preliminary experiments we noticed that agents learned very slowly, and so to aid the training and overcome the problem of the sparse reward function, we use curriculum learning (Bengio et al., 2009) with phases shown in Figure 3.3. This technique lets the agent gradually learn the best moves to obtain victory: for instance, in the first phase it is very easy to win the game, as the enemy has only 1 HP and only one attack is needed to defeat it; in this way the model can learn to reach its objective without worrying too much about other variables. As training proceeds, the environment becomes more and more difficult to solve, and the "greedy" strategy will no longer suffice: the agent HP will vary within a range of values, and the enemy will be more difficult to defeat, so it must learn how to use the collectible items correctly and which attack is the best for every situation. In the final phases loot can be difficult to find and the HP, of both agent enemy, can be within a large range of values: the system must develop a high level of strategy to reach the end of the game with the highest score possible.

Agents opponent. The behavior of the enemies agents are pitted against is *of great importance*. To satisfy requirements defined in Section 3.3, during training the agents fight against an opponent that always makes *random moves*. In this way, the agent sees all the possible actions that a user might perform, and at the same time it can be trained against a limited enemy with respect of human capabilities. This makes the agent beatable in the long run, but still capable of offering a tough and unpredictable challenge to the human player.

Training results

The intrinsic characteristic values for NPCs must be chosen before training. These parameters are not observed by the system, but offer an easy method to create different types of agents.



(a) (b) (c)

Figure 3.4: Plots showing metrics during the training phase for the warrior class as a function of the number of episodes. From left to right: (a) the evolution of the mean reward, (b) the evolution of the entropy, and (c) the difference between the training with and without curriculum. The dashed vertical lines on the plots delineate the different curriculum phases.

Changing the agent's ATK, DEF or DEX obliges that agent to extrapolate the best strategy based on its own characteristics. For DeepCrawl we trained three different combinations:

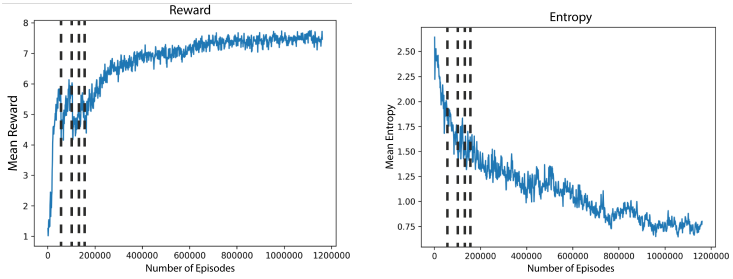
- **Archer**: ATK = 0, DEX = 4 and DEF = 3;
- **Warrior**: ATK = 4, DEX = 0 and DEF = 3; and
- **Ranger**: ATK = 3, DEX = 3 and DEF = 3.

For simplicity, the opponent has always the same characteristics: ATK = 3, DEX = 3 and DEF = 3.

To evaluate training progress and quality, we performed some quantitative analysis of the evolution of agent policies. In Figure 3.4 we show the progression of the mean reward and entropy for the warrior class as a function of the number of training episodes, while in Figure 3.5 and Figure 3.6 we show the other two types of agents following the same trend. In the same figure we show the difference between training with and without curriculum learning. Without curriculum, the agent learns much slower compared to multi-phase curriculum training. With a curriculum the agent achieves a significantly higher average reward at the end of training. While it is generally accepted best practice to repeat experiments multiple times and report the mean and standard deviation of the results, we were unable to do so in this work due to limited computational resources. This limitation should be taken into consideration when interpreting the results of this study. We suggest that future researchers who wish to replicate these results or continue this line of research conduct multiple experimental runs to accurately capture the variability of the measurements.

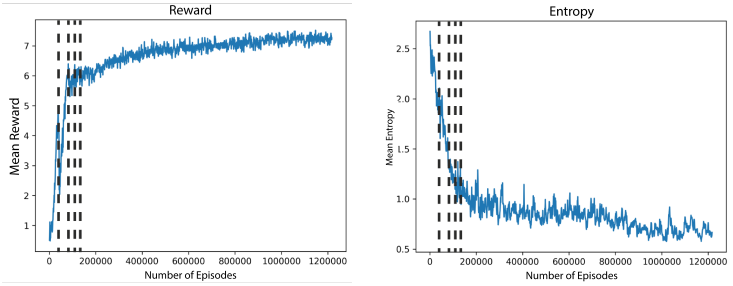
PPO and hyperparameters

To optimize the policy networks we used the PPO algorithm, the detailed description of which is provided in Chapter 2. One agent rollout is made of 10 episodes, each of which



(a) (b)

Figure 3.5: Plots showing metrics during the training phase for the archer class as a function of the number of episodes. From left to right: (a) the evolution of the mean reward and (b) the evolution of the entropy. The dashed vertical lines on the plots delineate the different curriculum phases.



(a) (b)

Figure 3.6: Plots showing metrics during the training phase for the ranger class as a function of the number of episodes. From left to right: (a) the evolution of the mean reward and (b) the evolution of the entropy. The dashed vertical lines on the plots delineate the different curriculum phases.

lasts at most 100 steps, and it may end either achieving success (i.e. agent victory), a failure (i.e. agent death) or reaching the maximum steps limit. At the end of 10 episodes, the system updates its weights with the episodes just experienced. PPO is an Actor-Critic algorithm with two functions that must be learned: the policy and the baseline. The latter has the goal of a normal state value function and, in this case, has the exactly same structure as the policy network show in Figure 3.4.

Most of the remaining hyper-parameters values were chosen after many preliminary experiments made with different configurations: the policy learning rate $lr_p = 10^{-6}$, the baseline learning rate $lr_b = 10^{-4}$, the agent exploration rate $\epsilon = 0.2$, and the discount factor $\gamma = 0.99$.

3.6 Playability Evaluation

To evaluate the DeepCrawl prototype with respect to our desiderata, we conducted playability test as a form of qualitative analysis. The tests were administered to 10 candidates, all passionate videogamers with knowledge of the domain; each played DeepCrawl for sessions lasting about 60 minutes. Then, each player was asked to answer a Single Ease Question (SEQ) questionnaire. All the questions were designed to understand if the requirements laid out in Section 3.3 had been met and to evaluate the general quality of DeepCrawl. Table 3.1 summarizes the results.

Table 3.1: Results of the SEQ questionnaire. Players answered with a value between 1 (strongly disagree) and 7 (strongly agree).

N°	Question	μ	Mo	Md	σ
1	Would you feel able to get to the end of the game?	5.54	6	6	1.03
2	As the level increases, have the enemies seemed too strong?	4.63	4	5	0.67
3	Do you think that the enemies are smart?	5.72	5	6	0.78
4	Do you think that the enemies follow a strategy?	6.18	6	6	0.40
5	Do you think that the enemies do counterintuitive moves?	2.00	2	2	0.63
6	Do the different classes of enemies have the same behavior?	1.27	1	1	0.46
7	Are the meaning of the icons and writing understandable?	5.72	6	6	1.67
8	Are the information given by the User Interface clear and enough?	5.54	6	6	1.21
9	Are the level too big and chaotic?	2.00	1	2	1.34
10	Are the objects in the map clearly visible?	5.81	7	7	1.66
11	Do you think that is useful to read the enemy's characteristics?	6.90	7	7	0.30
12	How much is important to have a good strategy?	6.90	7	7	0.30
13	Give a general value to enemy abilities compared to other Roguelike games	6.00	6	6	0.77
14	Is the game enjoyable and fun?	5.80	6	6	0.87
15	Does the application have bugs?	1.09	1	1	0.30

We reconsider here each of the main requirements we discussed above in Section 3.3 in light of the player responses:

- **Credibility**: as shown by questions 3, 4, and 5, the agents defined with this model offer a tough challenge to players; the testers perceived the enemies as intelligent agents that follow a specific strategy based on their properties.
- **Imperfection**: at the same time, questions 1 and 2 demonstrate that players are confident they can finish the game with the proper attention and time. So, the agents we have trained seem far from being superhuman, but they rather offer the right amount of challenge and result in a fun gameplay experience (question 14).
- **Prior-free**: questions 5 and 12 show that, even with a highly sparse reward, the model is able to learn a strategy without requiring developers to define specific behaviors. Moreover, question 13 indicates that the agents implemented using DRL are comparable to others in other Roguelikes, if not better.

- **Variety**: the testers stated that the differences between the behaviors of the distinct types of agents were very evident, as shown by question 6. This gameplay element was much appreciated as it increased the level of variety and fun of the game, and improved the general quality of DeepCrawl.

3.7 Conclusions

In this chapter we presented a new initial DRL framework for development of NPC agents in video games. To demonstrate the potential of DRL in video game production, we designed and implemented a new Roguelike game called DeepCrawl that uses the model defined in this chapter with excellent results. These versions of the agents work very well, and the model supports numerous agents types only by changing a few parameters before starting training. We feel that DRL brings many advantages to the commonly used techniques like finite state machines or behavior trees.

The DeepCrawl prototype is a step towards the creation of effective game design tools based on DRL. It shows that such techniques can be used to develop credible – yet imperfect – agents that are prior-free and offer variety to gameplay in turn-based strategy games. We feel that DRL, with some more work towards rendering training scalable and flexible, can offer great benefits over the classical, hand-crafted agent design that dominates the industry. One of the most critical aspects we discovered in developing the DeepCrawl framework is the fundamental importance of PCG environments. Our experiments show that training an agent with PCG allows it to avoid memorizing the trajectory to achieve goals, but rather it creates an abstract representation that helps the agent generalize to unseen situations and levels. This will not only lead to more capable agents, but also ones able to handle the many possible situations that can happen when playing against human players. In the next chapter we further develop these concepts via a thorough study of DRL, PCG, and generalization and how these elements can be combined together to improve scalability in DRL.

Chapter 4

Behaviors that Adapt to Changes in Design Parameters[†]

In Chapter 3 we defined requirements for training believable Non-Player Characters (NPCs) and introduced the DeepCrawl environment, an open source testbed that from now on will be a fundamental part of our experiments. Together with these two contributions, we also discovered that Procedural Content Generation (PCG) can greatly increase the quality and generalization ability of trained agents. The ability to manage all the possible situations that can happen when a trained NPC plays against a human player is of fundamental importance.

We also saw that turn-based strategy games like Roguelikes, for example, present unique challenges to DRL. In particular, the categorical nature of their complex game state, composed of many entities with different attributes, requires agents able to learn how to compare and prioritize these entities. Moreover, this complexity often leads to agents that overfit to states seen during training and that are unable to generalize in the face of design changes made during development. In this chapter we propose two network architectures which, when combined with a *procedural loot generation* system, are able to better handle complex categorical state spaces and to mitigate the need for retraining forced by design decisions. The first is based on a dense embedding of the categorical input space that abstracts the discrete observation model and renders trained agents more able to generalize. The second proposed architecture is more general and is based on a Transformer network able to reason relationally about input and input attributes. Our experimental evaluation demonstrates that new agents have better adaptation capacity with respect to a baseline architecture, making this framework more robust to dynamic gameplay changes during development.

[†]Portions of this chapter appeared in: A. Sestini, A. Kuhnle, and A. D. Bagdanov, "Deep policy networks for NPC behaviors that adapt to changing design parameters in Roguelike games", published in the *Reinforcement Learning in Games (RLG) Workshop at Association for the Advancement of Artificial Intelligence (AAAI) conference*, 2020.

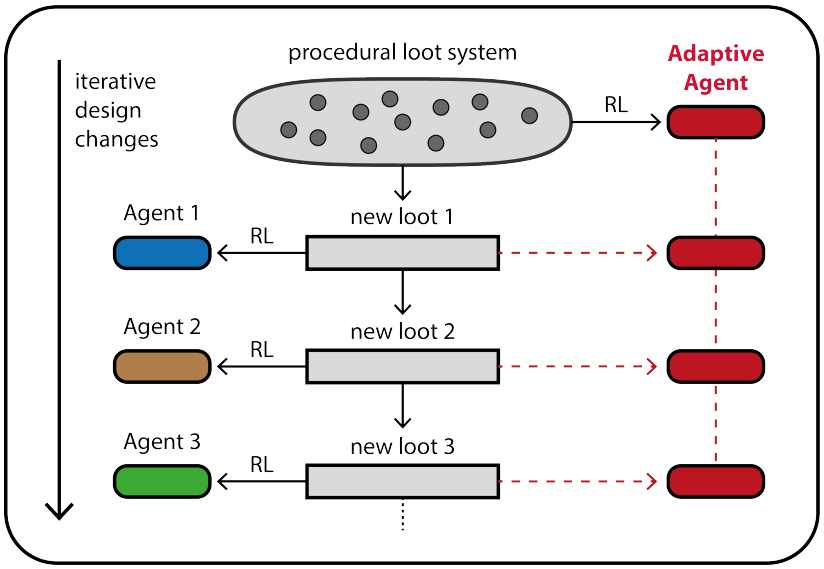


Figure 4.1: A summary of our approach. The left side illustrates the original DeepCrawl framework defined in Chapter 3 in which the agent must be retrained every time the loot distribution changes (e.g. for balancing the overall game). Our approach is shown on the right: with the new adaptive architectures we can create agents which can learn from a new procedural loot system. Agents trained in this way are able to adapt to a changing loot distribution without the need to retrain.

4.1 Introduction

In this chapter we address the challenges of *adaptation* and *scaling* encountered in DeepCrawl. Deep Reinforcement Learning (DRL) algorithms are extremely sensitive to design changes in the environment, since they fundamentally change the way agents "see" the game world around them. Even seemingly minor changes can force a complete retraining of all agents. This is mostly due to the categorical nature of the input state space which makes the network overfit to the specific entities seen during training, leaving it without the capacity to generalize to unseen states. Collectible objects in DeepCrawl and their effect on the game, for example, must be predefined by developers, and are represented by unique integer IDs and not by their effect on the player. This can be an important problem during game development: if developers want to change parameters, for example to balance gameplay, they require agents which can handle these modifications and do not require retraining. This makes it difficult to adapt an existing agent to new scenarios, resulting in inappropriate agent behavior when NPC agents are used in environments for which they were not designed.

Moreover, with the NPC model architecture of the DeepCrawl framework it is not possible to extend the set of available loot or loot types without completely retraining agents from scratch. This is largely due to specific DeepCrawl network architecture: the policy network contains initial *embedding layers* that make it possible for the network to learn a continuous vectorial representation encoding the meaning of and differences between *categorical* inputs.

As mentioned above, in this setting each loot item must be identified by a unique ID in order to be understandable by agents. For this reason, if designers want to add new loot types, for example changing the object definition in order to have a different number of attribute bonuses, it is difficult or impossible to define a unique ID for each object *a priori* – particularly if the attribute bonuses are determined randomly during game play.

To mitigate these problems we implemented a new *procedural loot generation* system and incorporated it into the training protocol: instead of a fixed list of discrete items, in our new system an item is parametrized by a fixed set of attributes, potentially even an extensible set of attributes. These values are drawn from a uniform distribution when generating a new training episode and increase the intrinsic properties of actors when collected. We also propose two alternative policy network architectures that are able to handle the new procedural loot system. As we illustrate in Figure 4.1, the new system combined with procedural loot generation during training renders trained NPCs more adaptive and scalable from a game developer's perspective. This new AI system helps in the design of NPC agents while being robust to iterative design changes across the loot distribution that can happen during video game development. As our experiments show, these new agents are perfectly capable of adapting to loot distributions they have never seen during training, without the need to retrain.

4.2 Related Work

As already mentioned, the potential of DRL in video games has been steadily gaining interest from the research community. Here we review recent works most related to this chapter.

Procedurally Generated Environments. There is a growing interest in DRL algorithms applied in environments with Procedural Content Generation (PCG) systems: Cobbe et al. (2019) demonstrated that diverse environment distributions are essential to adequately train and evaluate RL agents, as they observed that agents can overfit to exceptionally large training sets. On the same page are Risi and Togelius (2019), who stated that often an algorithm will not learn a general policy, but instead a policy that only works for a particular version of a particular task with specific initial parameters. Justesen et al. (2018) explored how procedurally generated levels during training can increase generalization, showing that for some games procedural level generation enables generalization to new levels within the same distribution. Subsequently, the growing need for a PCG environments was also demonstrated by Küttler et al. (2020), Chevalier-Boisvert et al. (2019), and Juliani et al. (2019).

DRL *in* video games. Modern video games are environments with complex dynamics, and these environments are useful testbeds for testing complex DRL algorithms. Some notable examples are: Vinyals et al. (2019) that use a specific deep neural network architecture based on Transformers (Vaswani et al., 2017) able to create super-human agents for StarCraft, and OpenAI (2019) that uses embedding layers similar to the one used in Chapter 3 to manage

the inner attributes of the agent and other heroes in DOTA 2 in order to train agents that outperform human players.

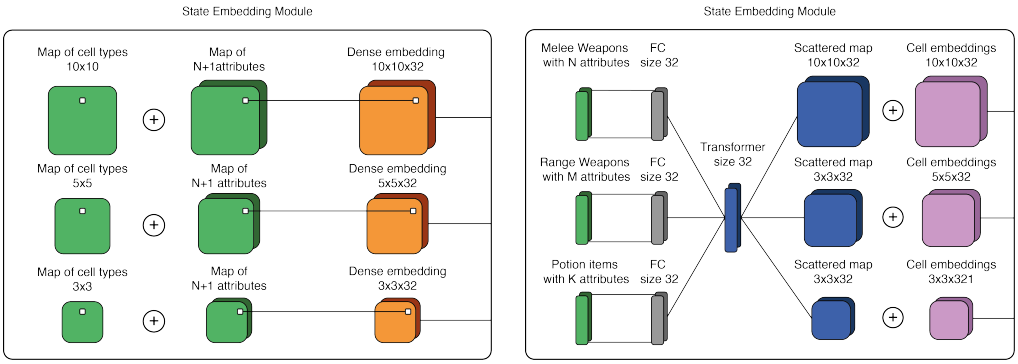
DRL *for* video games. At the same time, there is an increasing interest from the game development community on how to properly use DRL for video game development. Zhao et al. (2020) argued that the industry does not need agents built to "beat the game", but rather to produce human-like behavior to help with game evaluation and balance. Delalleau et al. (2019) dealt with the importance of having an easy-to-train neural network and how it is important to have a framework that enriches the expressiveness of the policy. Pleines et al. (2019) studied different action-space representations in order to create agents that mimic human input, without being super-human. As already discussed, in Chapter 3 we contributed to this aim, defining a DRL framework suited for the production of turn-based strategy games. Our aim is to improve on the latter framework in order to render it more robust to changes to gameplay mechanics during development – i.e. to render DRL agents more *mechanics-free*.

4.3 Proposed Models

Our work builds upon the DeepCrawl framework (Chapter 3). Our overarching goal is to make the system as independent as possible from dynamic changes during the development phase, and we argue that a crucial step in this direction is a *procedural loot generation system* which helps encourage generalizing agent behavior in a fully procedural environment. In particular, in this chapter we define three new desiderata, complementary to those previously defined:

- **Performance.** We desire agents able to properly handle a procedural loot system, so they must understand which object is most useful for defeating the game;
- **Adaptation.** Agents must adapt to changes in gameplay mechanics, in particular changes to the loot generation system during playtesting and rebalancing, without the need of retraining; and
- **Scalability.** We desire a framework that can scale in both the number of possible objects and in the number of attribute bonuses of each object type. Moreover, the framework must have limited complexity to facilitating targeting of systems like mobile devices.

With these three new desiderata in mind, we now describe two architectural solutions that satisfy them. Both are significant modifications of the early, frontend layers of the DeepCrawl network that allow it to better manage our new procedural loot system. We begin with a brief introduction of the original DeepCrawl environment and network, and then continue with the description of two different architectures that address the problems defined above.



(a) Dense Embedding module (b) Transformer Embedding module

Figure 4.2: The two new architectures proposed in this chapter: (a) The Dense Embedding module, and (b) the Transformer Embedding module. These modules replace the *State Embedding Module* in the original architecture shown in Figure 3.2, while the rest of the policy network is left unchanged. See Section 3.2 for a detailed description.

Procedural Loot System

In this chapter we focus mainly on the State Embedding module of the cited architecture, shown in Figure 3.2. For complete details of the DeepCrawl framework and the neural network used by its NPCs, we refer the reader to Chapter 3. As discussed above, the input structure of the DeepCrawl framework defined in Chapter 3 limits the adaptation nature of the trained agents. We overcome this limitation by first defining and implementing a different way to generate collectible items in the environment.

In our proposed parametric loot system, each object has a number of attributes whose values during training are drawn from a uniform distribution when generating an environment for a new training episode. When an actor collects an item, the actor characteristics will increase or decrease according to the attributes of the instance of the looted object. In our current implementation, each object has the same four attributes corresponding to the four characteristics of the actors. This is not a requirement, however, rather it reflects the original design and implementation of categorical loot system in the DeepCrawl game.

This system brings a lot of benefits to DeepCrawl: it makes the game more complex and varied, with the corresponding possibility of creating more convincing NPCs and player/environment interactions. The environment is now fully procedural, which should increase the generalization of the agents. Moreover, during playtesting developers can choose either to use random objects or to define a set of fixed objects with fixed attributes in order to balance the game.

However, to enjoy these benefits the policy networks trained for agent behaviors must be able to accommodate this new procedural loot system. The network described above cannot easily do this due to the categorical nature of its input space. Thus we propose two new solutions.

Dense Embedding Policy Network

Our first model is a straightforward extension of the one used in the previous chapter. We were inspired by the ideas of OpenAI (2019) and DeepCrawl to treat the map of categorical inputs via embedding layers. In contrast to these approaches, however, we use multi-channel maps where each channel represents a different categorical value:

- The first channel represents the type of entity in that position:
  - 0 = impassable object;
  - 1 = empty tile;
  - 2 = agent;
  - 3 = player;
  - 4 = melee weapon;
  - 5 = range weapon; and
  - 6 = potion item.
- The other channels each contain an attribute of the object in that tile, represented by a categorical value. For instance, if a tile contains a melee weapon, its attribute bonuses like health points (HP), attack (ATK), defense (DEF), and dexterity (DEX) are represented by an array of all attributes (plus tile type): `[TYPE, HP, ATK, DEF, DEX]`. If the tile does not contain loot (like an impassable object), this array is filled with the special value `no-attribute`: `[TYPE, NONE, NONE, NONE, NONE]`.

This multi-channel map input, which like the original DeepCrawl network as shown in Figure 3.2, is divided in global and local views and passed through what we refer to as *"dense embedding"* layers: multiple categorical values are combined together and mapped to their corresponding fixed-size continuous representation by a single dense embedding operation. To implement the dense embedding operation, we simply convert each channel into a one-hot representation and apply a 1×1 convolution with stride 1 and *tanh* activation through all channels. In the special case of a single channel, the operation is equivalent to standard embedding layers. The full model architecture is shown in Figure 4.2(a).

This architecture satisfies the requirements we are looking for: the framework is independent from attribute changes to the loot system as long as the types of character attributes remain the same. Moreover, if developers want to change the set of attributes during production, it is no longer necessary to change the entire agent architecture, only the corresponding channels of the dense embedding layer need to be added or removed. As an additional benefit, the network size remains relatively small. A detailed analysis of experimental results for this architecture are given in Section 4.4.

(a) (b) (c)

Figure 4.3: Mean reward during the training phase for all classes as a function of timestep. From left to right: (a) archer, (b) warrior, and (c) ranger. The dashed vertical lines on the plots delineate the different curriculum phases, which are the same as in Chapter 3.

Transformer-based Policy Network

We propose an alternative model based on the recently popular Transformer architecture (Vaswani et al., 2017), and particularly its self-attention layer which has also been successfully applied as state encoder in RL applications (Baker et al., 2019; Vinyals et al., 2019; Zambaldi et al., 2018). This model uses self-attention to iteratively reason about the relations between entities in a scene, and is expected to improve upon the efficiency and generalization capacity over convolutions by more explicitly focusing on entity-entity relations.

Concretely, the self-attention layer takes as input the set of entities e_i for which we want to compute interactions (not including auxiliary `no-attribute` objects), and then computes a multi-head dot-product attention (Vaswani et al., 2017): given N entities, each is projected to a query q_i, a key k_i and a value v_i embedding, and the self-attention values are computed as

$$A = \text{softmax}\left(\frac{QK^t}{\sqrt{d}}\right)V, \tag{4.1}$$

where A, Q, K, and V represent the cumulative interactions, queries, keys and values as matrices, and d is the dimensionality of the key vectors. As in the original paper, we use 4 independent such self-attention heads. Subsequently, the output vectors per head are concatenated and passed on to a fully-connected layer, and finally added to the entity vector e_i via residual connection to yield a fixed-size embedding vector per entity.

The Transformer operation thus produces embeddings which encode relations between loot in the environment. In this case we represent each object by an array of its attribute bonuses, normalized between 0 and 1, which are further processed by fully connected layers with shared weights across loot types. Based on these representations, a Transformer layer is applied to reason about loot-loot relations, resulting in a fixed-size embedding per entity. Following the concept of *spatial encoders* from AlphaStar, all entity representations are then

scattered into a spatial map so that the embedding at a specific location corresponds to the unit/object placed there.

More specifically, we create an empty map and place the embeddings returned by the Transformer at the corresponding positions where the loot is located in the game. We produce such scattered maps for both global and local views which, as before, are concatenated with the embedding map of categorical tile type and then passed on to the remaining convolutional layers. The full network is shown in Figure 4.2(b). This model is, again, independent from changes to the loot generation system, and even if developers change the number of attributes during production, this architecture does not require any adaptation, but can simply be retrained on the new game. The biggest weakness is that this architecture is quite complex and requires more computational resources, which goes against the last desideratum defined in Section 4.3.

4.4 Experimental Results

In this section we report on experiments performed to evaluate differences, advantages, and disadvantages of the two new architectures with respect of the Categorical network. All of our policy networks were implemented using the Tensorforce library (Kuhnle et al., 2017).*

We follow the same training setup of Chapter 3. At the beginning of each episode, the shape and the orientation of the map, as well as the number of impassable and collectible objects and their positions are randomly generated; the initial position of the player and the agent is random, as well as their initial equipment. We also use curriculum learning (Bengio et al., 2009) with the same phases as the previous chapter and during training the agents fight against an opponent that always makes *random moves*. The only difference is the addition of the *loot generation system* described in Section 4.3: each collectible item now has four attributes which correspond to and modify the actor properties (HP, DEX, ATK, DEF). At the beginning of each episode these values are drawn randomly from a uniform distribution for each loot object on the map.

We trained three NPC classes (Archer, Warrior, and Ranger (the same as those from the DeepCrawl framework defined in Chapter 3) using the Transformer, Dense Embedding, and the original Categorical deep policy networks. The NPC classes are distinguished from one another by their character attributes – see Chapter 3 for a complete description of the training procedure. In the following, we assess each of the main requirements discussed above in Section 4.3 in light of our experimental results.

Performance. Figure 4.3 shows the training curves for our two proposed policy networks. The two architectures achieve the same reward, demonstrating that both are able to properly handle the new version of the environment. Table 4.1 shows that, if two agents of the same class fight each other in the testing configuration (where they start with the max amount of HP

*Code available at https://github.com/SestoAle/Adaptive-NPCs-with-procedural-entities

and their initial equipment are neutral weapons, while the loot in the map is still procedural), the Transformer based policy has a slight advantage against the Dense Embedding network.

Table 4.1: Success rates averaged over 100 episodes for pairs of policy networks playing against each other in different testing environments. *Procedural Loot* refers to the environment with fully procedural loot, where each item attribute is drawn randomly at the beginning of an episode. *Uniform loot* refers to the variant with a fixed set of only three types of weapons (low, medium and high power ones). *Skewed Loot* refers to another another such variant, one in which the strongest weapons are far more powerful than the other two. For more details about the experimental setup and loot distributions, see Section 4.4. The proposed network architectures generalize better across distributional loot changes compared to the original categorical architecture.

	Transformer vs Dense Embedding			Dense Embedding vs Categorical			Transformer vs Categorical		
	Procedural loot	Uniform loot	Skewed loot	Procedural loot	Uniform loot	Skewed loot	Procedural loot	Uniform loot	Skewed loot
Archer	55%	56%	52%	62%	58%	**67%**	60%	57%	66%
Warrior	50%	52%	50%	60%	56%	**66%**	60%	58%	66%
Ranger	52%	50%	49%	58%	56%	63%	56%	58%	**64%**

We cannot compare these training curves with those in the original work because of the dynamic nature of the environment introduced by the procedural loot system. The only way to compare with the Categorical network is to train agents with it in the new environment after discretizing the loot attributes into potentially very many unique object IDs. We can, however, have agents of the same class but with different policy fight each other in this new environment. As Table 4.1 shows, the proposed policies have higher average success rate with respect to the Categorical policy network. This demonstrates that these solutions better capture the differences between loot objects.

Adaptation. To demonstrate the improved generalization capacity of our proposed network architectures, we tested them by changing the loot distribution from the fully procedural one used during training to a fixed distribution. This new environment has only three different type of weapons: low, medium and high power (both ranged and melee) that have clear differences between each other – similar to the fixed weapons in the original DeepCrawl. For *high power* weapons we mean loot that gives high value bonuses for all attributes, and so forth for medium and low power ones. Based on the four attributes in DeepCrawl, in this variant a high power sword has attribute bonuses of `[+2, +2, +2, +2]`, a medium power sword has `[+0, +0, +0, +0]`, and a low power sword `[-2, -2, -2, -2]`. We refer to this distribution as the *uniform loot distribution*. We then compare the agents, which have been trained with the full procedural loot, in this testing environment. As Table 4.1 shows, our proposed models have a small advantage compared to the original Categorical framework.

In a subsequent experiment, we change the distribution of fixed loot power: there are still three weapon types, low, medium and high power, but the high power weapons are *far* more powerful than the medium and low power ones, which are comparatively similar. More concretely, a high power sword here has attribute bonuses of `[+5, +5, +5, +5]`, a medium sword `[-2, -2, -2, -2]` and a low power sword `[-3, -3, -3, -3]`. We call this distribution the *skewed loot distribution*. As shown in Table 4.1, with this configuration the success rate for our proposed architectures is much higher, outperforming agents trained with the previous framework and hence showing better adaptation than the Categorical architecture to such a change in the balance of the game.

Scalability. To handle the procedural loot system with the Categorical architecture developers must define a fixed set of class IDs prior to training. This is not a trivial task and can quickly become intractable when the number of attributes for each object increases. Moreover, since the Categorical framework does not generalize (see Table 4.1), if developers want to change loot generation they must define a new set of classes, and that forces retraining of agents. Instead, with our proposed solutions developers simply train their agents with procedurally generated loot and can decide after training whether to use, in the final game, random objects or a fixed set of weapons for balancing the game: our agents will manage both situations without the need of retraining.

Both the proposed frameworks properly handle changes in the number of attributes. In this case retraining is mandatory, but developers need not to worry about changing the network architectures: with the Dense Embedding network they need only to add a new channel for each new attribute, while the Transformer based does not require any changes since it is completely independent of loot parameterization. The Transformer embedding can even handle loot with various number of attributes per type, providing a big advantage with respect to Dense Embedding which requires loot with the same number of attributes.

The biggest drawback of the Transformer network is its complexity. While the Dense and Categorical embedding networks require about 1.3 minutes to train 100 episodes, the Transformer network takes twice as long. The average training times for 100 episodes are 3.31 minutes, 1.36 minutes and 1.10 minutes for the Transformer, Dense Embedding and Categorical networks, respectively. Training was performed on an NVIDIA RTX 2080 SUPER GPU with 8GB of RAM.

In a video game design and development context, this is an important aspect to consider: the continuous changes in the gameplay mechanics require many retrainings, and having a small network is essential. In addition, these frameworks must be implemented to target devices with reduced performance, increasing the need for small and efficient models.

PPO and Hyperparameters

As in Chapter 3, we use the PPO algorithm to optimize the agent model. The agent is trained over the course of multiple episodes, each of which lasts at most 100 steps. An episode

may end with the agent achieving success (i.e. agent victory), failure (i.e. agent death) or reaching the maximum step limit. After every fifth episode, an update of the agent weights is performed based on the previous five episodes. PPO is an Actor-Critic algorithm with two networks to be learned: the actor policy and the critic state-value function. For both the dense embedding and transformer variant, the critic uses the same structure as the policy network.

Hyperparameter values are the same in all experiments and were chosen after a preliminary set of experiments with different configurations: the policy learning rate $lr_p = 5 \cdot 10^{-6}$, the baseline learning rate $lr_b = 5 \cdot 10^{-4}$, the agent exploration rate $\epsilon = 0.2$, and the discount factor $\gamma = 0.99$. For the transformer architecture, we used a two-headed self-attention layer, with queries, keys and values of size 32 and a two-layers MLP with size 128 and 32.

Similar to the results of Chapter 3, this research was limited by insufficient computational resources, thus it was not possible to repeat experiments multiple times and report the mean and standard deviation of the results as recommended in best practice. To accurately interpret the results of this study and to facilitate replication, future researchers should run multiple experiments and measure the variability of the measurements.

4.5 Conclusions

In this chapter, we described several extensions to our DeepCrawl framework. First, we implemented a procedural loot generation system which augments the game with a degree of complexity that makes the game more compelling as benchmark for DRL algorithms, particularly in the context of game development. Moreover, we proposed two neural network architectures, one based on Dense Embeddings and one based on Transformers, which both show substantially improved performance due to their capabilities to reason about loot and attribute bonuses. Overall, our experimental analysis slightly favors the Dense Embedding approach due to its reduced complexity and computational requirements.

The advantages for game development are twofold. On the one hand, Roguelikes such as DeepCrawl may contain a large number of items, or indeed employ a procedural loot generation system, so the ability to effectively learn how to compare and prioritize loot is important for NPCs. On the other hand, this ability makes NPCs robust to modifications to the loot system during development, without the need to retrain the behavioral models from scratch every time. This is important, first, for the balancing process during playtesting which is crucial to final quality; and second, both our proposed architectures can easily be adapted in the face of major changes to the loot system which may occur during production.

There is another major challenge in DRL for video game development: the reward function. The *"prior-free"* desiderata defined in Chapter 3 requires us to non manually specify strategies. Nonetheless, designers may want to have some form of control over the final behavior of agents. This is especially evident when it comes to defining qualitative behaviors, e.g. training an agent to act more *"sneaky"*. However translating a qualitative behavior into a mathematical function is somewhat unnatural for humans. As discussed in

Chapter 2, inverse reinforcement learning and imitation learning can mitigate this problem and still satisfy the "*prior-free*" requirement. In the next chapter we see how we can leverage both PCG and inverse reinforcement learning to create agents that follow designer guidance provided via demonstrations.

Chapter 5

Inverse Reinforcement Learning in PCG Environments[†]

In the previous chapters we saw that deep reinforcement learning achieves very good results in domains where reward functions can be manually engineered. At the same time, there is a growing interest within the community in using games based on Procedurally Content Generation (PCG) as benchmark environments. This type of environment is perfect to study overfitting and generalization of agents under domain shifts. Inverse Reinforcement Learning (IRL) can instead extrapolate reward functions from expert demonstrations, with good results even on high-dimensional problems, however there are no examples in literature on how to apply these techniques to procedurally-generated environments. This is mostly due to the number of demonstrations needed to find a good reward model.

In this chapter we propose a technique based on the state-of-the-art adversarial inverse reinforcement learning algorithm which can significantly decrease the need for expert demonstrations in PCG games. Through the use of an environment with a limited set of initial *seed levels*, plus some modifications to stabilize training, we show that our proposed approach is demonstration-efficient and still able to extrapolate reward functions which generalize to the fully procedural domain. We demonstrate the effectiveness of our technique on two procedural environments, MiniGrid and DeepCrawl, for a variety of tasks.

5.1 Introduction

Despite the cited success of Deep Reinforcement Learning (DRL) in environments where the reward function is known, designing and engineering good hard-coded reward functions is difficult in some domains. In other settings, a badly-designed reward function can lead to agents which receive high rewards in unintended ways (Amodei et al., 2016).

[†]Portions of this chapter appeared in: A. Sestini, A. Kuhnle, and A. D. Bagdanov, "Demonstration-Efficient Inverse Reinforcement Learning in Procedurally Generated Environments", published in the *Conference on Games (CoG)*, 2021.

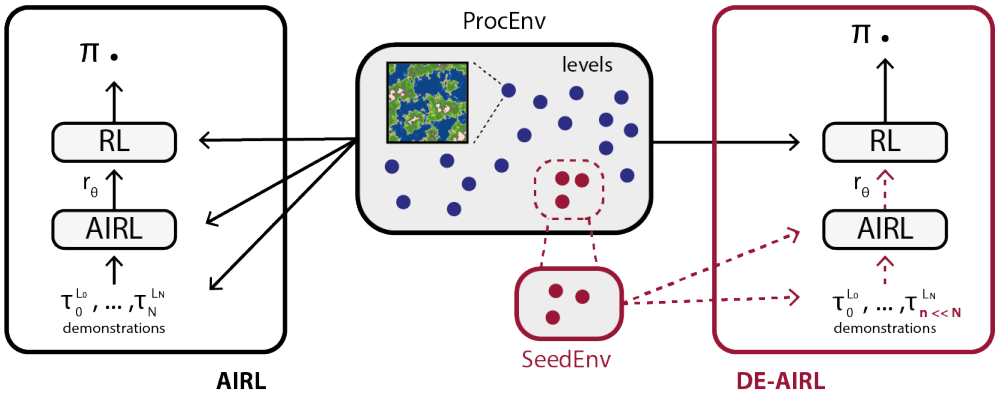


Figure 5.1: Demonstration-efficient AIRL. The left part of the image illustrates the AIRL baseline, which extrapolates a reward function from expert demonstrations directly on the fully procedural environment. This naive application of AIRL requires a large number of expert demonstrations. Our demonstration-efficient AIRL approach is shown in the right part of the image. DE-AIRL extrapolates the reward function on a subset of all possible game levels, referred to as SeedEnv, and is applied in the fully procedural environment, ProcEnv, only after training. This approach enables an RL policy to achieve near-expert performance while requiring only a few expert demonstrations.

Inverse Reinforcement Learning (IRL) algorithms attempt to infer a reward function from expert demonstrations (Ng and Russell, 2000). This reward function can then be used to train agents which thus learn to mimic the policy implicitly executed by human experts. IRL offers the promise of solving many of the problems entailed by reward engineering. These approaches have achieved good performance both in continuous control tasks (Fu et al., 2018; Finn et al., 2016b) and in Atari games (Tucker et al., 2018).

At the same time, there is increasing interest from the DRL community in procedurally-generated environments. In the video game domain, Procedural Content Generation (PCG) refers to the programmatic generation of environments using random processes that result in an unpredictable and near-infinite range of possible states. PCG controls the layout of game levels, the generation of entities and objects, and other game-specific details. Cobbe et al. (2019) noted that in classical benchmarks like the Arcade Learning Environment (ALE) (Bellemare et al., 2013), agents can memorize specific trajectories instead of learning relevant skills, since agents perpetually encounter near-identical states. In Chapter 4 we saw how we can leverage PCG and a set of procedurally-generated rules to increase generalization of trained agents in face of design changes, thus helping the usability of such agents in video game development. Because of this, PCG environments are a promising path towards addressing the need for generalization in RL. For an agent to do well in a PCG environment, it has to learn policies robust to ever-changing levels and a general representation of the state space.

Most IRL benchmarks focus on finding reward functions in simple and static environments

like MuJoCo (Todorov et al., 2012) and comparatively simple video games like Atari (Tucker et al., 2018). None of these RL problems incorporate levels generated randomly at the beginning of each new episode. The main challenges with procedurally-generated games is the dependence of IRL approaches on the number of demonstrations: due to the variability in the distribution of levels, if a not sufficiently large number of demonstrations is provided, the reward function will overfit to the trajectories in the expert dataset. This leads to an unsuitable reward function and consequently poorly performing RL agents. Moreover, in most domains, providing a large number of expert demonstrations is expensive in terms of human effort.

To mitigate the need for very many expert demonstrations in PCG games, we propose a novel inverse reinforcement learning technique for such environments. Our work is based on Adversarial Inverse Reinforcement Learning (AIRL) (Fu et al., 2018) and substantially reduces the required number of expert trajectories (see Figure 5.1). We propose specific changes to AIRL in order to decrease overfitting in the discriminator, to increase training stability, and to help achieve better performance in agents trained using the learned reward. Additionally, instead of using a fully procedural environment for training, we *"under-sample"* the full distribution of levels into a small, fixed set of *seed levels*, and experts need only provide demonstrations for this reduced set of procedurally-generated levels. We show that the disentangled reward functions learned by AIRL generalize enough such that, subsequently, they enable us to find near-expert policy even on the full distribution of all possible levels. We test our approach in two different PCG environments for various tasks.

5.2 Related Work

Here we review the recent works most related to this chapter.

Inverse Reinforcement Learning. Inverse Reinforcement Learning (IRL) refers to techniques that infer a reward function from human demonstrations, which can subsequently be used to train an RL policy. It is often assumed that demonstrations come from an expert who is behaving near-optimally. IRL was first described by Ng and Russell (2000), and one of its first successes was Maximum Entropy IRL (Ziebart et al., 2008), a probabilistic approach based on the principle of maximum entropy favoring rewards that lead to a high-entropy stochastic policy. However, this approach assumes known transition dynamics and a finite state space, and can retrieve only a linear reward function. Guided Cost Learning (Finn et al., 2016b) relaxed these limitations and was one of the first algorithm able to estimate non-linear reward functions over infinite state spaces in environments with unknown dynamics. Recently, Finn et al. (2016a) noticed that GCL is closely related to GAN training, and this idea led to the development of Adversarial Inverse Reinforcement Learning (AIRL) (Fu et al., 2018). This method is able to recover reward functions robust to changes in dynamics and can learn policies even under significant variations in the environment.

Imitation Learning. Similarly to IRL, Imitation Learning (IL) aims to directly find a policy that mimics the expert behavior from a dataset of demonstrations, instead of inferring a reward function which can subsequently be used to train an RL policy. Standard approaches are based on Behavioral Cloning that mainly use supervised learning (Bain and Sammut, 1995; Syed and Schapire, 2008; Ross et al., 2011; Reddy et al., 2019; Cai et al., 2019; Knox and Stone, 2009). Generative Adversarial Imitation Learning (GAIL) (Ho and Ermon, 2016) is a recent IL approach which is based on a generator-discriminator approach similar to AIRL. However, since our goal is to operate in PCG environments, we require IRL methods able to learn a reward function which generalizes to different levels rather than a policy which tends to overfit to levels seen in expert demonstrations.

Other approaches combine IL and IRL (Ibarz et al., 2018): they first do an iteration of Behavioral Cloning, and then apply active preference learning (Christiano et al., 2017) in which they ask humans to choose the best of two trajectories generated by the policy. With these preferences they obtain a reward function, which the policy tries to optimize in an iterative process.

Procedural Content Generation. Procedural Content Generation (PCG) refers to algorithmic generation of level content, such as map layout or entity attributes in video games. A detailed description of the most recent and related papers regarding PCG is given in Chapter 4. Some examples of the growing interest in PCG environments from the DRL community are the works by Risi and Togelius (2019), Justesen et al. (2018), Küttler et al. (2020), Guss et al. (2019), Chevalier-Boisvert et al. (2019) and Juliani et al. (2019). Notably for this chapter, Guss et al. (2019) applied imitation learning in the form of behavioral cloning over a large set of human demonstrations in order to improve the sample efficiency of DRL.

5.3 Modifications to AIRL

In the following we present three extensions to the original AIRL algorithm which increase stability and performance, while decreasing the tendency of the discriminator to overfit to expert demonstrations. For complete details on the AIRL algorithm, we refer the reader to Chapter 2.

- **Reward standardization.** Adversarial training alternates between discriminator training and policy optimization, and the latter is conditioned on the reward which is updated with the discriminator. However, forward RL assumes a stationary reward function, which is not true in adversarial IRL training. Moreover, policy-based DRL algorithms usually learn a value function based on rewards from previous iterations, which consequently may have a different scale from the currently observed rewards due to discriminator updates. Generally, forward RL is very sensitive to reward scale which can affect the stability of training. For these reasons, as suggested by Tucker

et al. (2018) and Ibarz et al. (2018), we standardize the reward to have zero mean and some standard deviation.

- **Policy dataset expansion.** In the original AIRL algorithm, each discriminator training step is followed by only one policy optimization step. The experience collected in this policy step is then used for the subsequent discriminator update. However, a single trajectory may not offer enough data diversity to prevent the discriminator from overfitting. Hence, instead of just one policy step, we perform K iterations of forward RL for every discriminator step as suggested by Tucker et al. (2018).

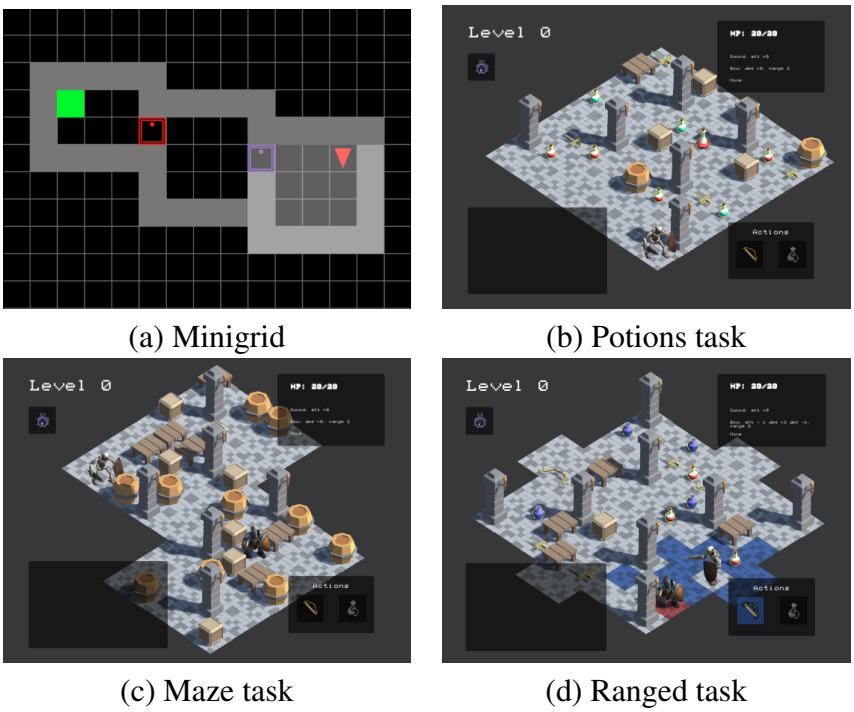
  Moreover, as already noted by Fu et al. (2018) and Reddy et al. (2019), IRL methods tend to learn rewards which explain behavior locally for the current policy, because the reward can forget the signal that it gave to an earlier policy. To mitigate this effect we follow their practice of using experience replay over the previous iterations as policy experience dataset. For the same reason, when we apply the learned reward function, we do not re-use the final, possibly overfitted reward model, but rather one from an earlier training iteration.

- **Fixed number of timesteps.** Many environments have a terminal condition which can be triggered by agent behavior. Christiano et al. (2017) observed that these conditions can encode information about the environment even when the reward function is not observable, thus making the policy task easier. Moreover, since the range of the learned reward model is arbitrary, rewards may be mostly negative in some situations, which encourages the agent to meet the terminal conditions as early as possible to avoid more negative rewards (the so-called *"RL suicide bug"*). For these reasons we do not terminate an episode in a terminal state, but artificially extend it to a fixed number of timesteps by repeating the last timestep.

5.4 Demonstration-efficient AIRL in Procedural Environments

In PCG game environments, the configuration of the level as well as its entities are determined algorithmically. Unless the game is very simplistic, this means it is unlikely to see the exact same level configuration twice. Forward RL benefits from such environmental diversity by increasing the level of generalization and credibility of agent behavior. However, as a consequence of this diversity, many expert demonstrations may be required for IRL to learn useful behavior. This is especially challenging for an adversarial techniques like AIRL as it is known that GANs require many positive examples (Lučić et al., 2019).

In the following, we call the fully procedural environment *ProcEnv*. Levels $L_i \sim$ ProcEnv are sampled from this environment, and sample trajectories $\tau^{L_i} \sim L_i$ from each level, where



(a) Minigrid (b) Potions task

(c) Maze task (d) Ranged task

Figure 5.2: Screenshots of the various environments and tasks.

trajectories $\tau^{L_i} = (s_0, a_0, \ldots, a_{T-1}, s_T)$ are sequences of alternating states and actions. Consequently, if we have two trajectories τ^{L_i} and τ^{L_j}, in most cases (unless $L_i = L_j$) they differ not only in their state-action sequences, i.e. the behavior, but also in their level content L_i vs L_j from which they are sampled.

To illustrate this, suppose we have a simple ProcEnv with two generation parameters: the number of objects $o \in [1, 10]$ and the number of enemies $e \in [1, 6]$, so overall $|\text{ProcEnv}| = 10 \cdot 6 = 60$ level configurations. Sampling expert demonstrations is a two-stage process: first, we sample levels $L_1 = (3, 4), L_2 = (5, 1), L_3 = (7, 2) \sim \text{ProcEnv}$, where (o, e) denotes the number of objects and enemies, respectively, and next we sample corresponding trajectories $\tau_1^{(3,4)}, \tau_2^{(5,1)}, \tau_3^{(7,2)}$, which form our expert dataset. When faced with another trajectory sample based on a random level, say, $\tau^{(1,4)}$, the discriminator can simply distinguish expert and non-expert trajectories by counting objects and enemies in the levels as observed in the states of the trajectories and *ignoring* agent behavior entirely. Sampling more expert trajectories increases the probability of levels being equal (or at least similar), and thus makes it harder to memorize level configurations. However, collecting a large number of demonstrations can be very expensive, and cannot not ultimately solve the problem for rich enough PCG environments.

Our objective is to make AIRL effective and data-efficient when working with PCG environments. Our main idea is to introduce an artificially reduced environment, which we call a *SeedEnv*, that consists of $n \ll N$ levels sampled from the fully procedural ProcEnv.

These levels are then used to obtain n randomly sampled expert demonstrations:

$$\text{SeedEnv}(n) = \{L_1, ..., L_n \mid L_i \sim \text{ProcEnv}\}$$
$$\text{Demos} = \{\tau^{L_i} \mid L_i \in \text{SeedEnv}(n)\}$$

Using the simplified example from before, this would mean that $\text{SeedEnv}(3) = \{L_1, L_2, L_3\}$ and $\text{Demos} = \{\tau_1^{L_1}, \tau_2^{L_2}, \tau_3^{L_3}\}$. In the following, we refer to each $L_i \in \text{SeedEnv(n)}$ as *seed level*.

The reward function is learned via AIRL on the reduced SeedEnv environment instead of the fully-procedural ProcEnv. To distinguish expert from non-expert trajectories, the discriminator thus cannot rely on memorized level characteristics seen in expert demonstrations, but instead must consider the behavior represented by the state-action sequence of the trajectory.

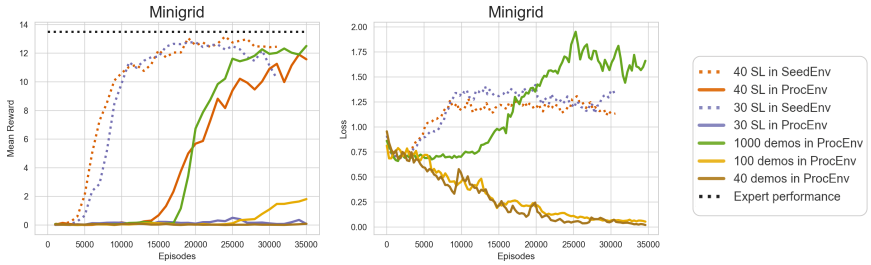
Once the discriminator is trained on SeedEnv, the learned reward function can be used to train a new agent on the full ProcEnv environment. The disentanglement property of AIRL encourages the reward function to be robust to the change of dynamics between different levels, assuming a minimum number of seed levels necessary to generalize across level configurations.

In summary, we observe that there are two sources of discriminative features in expert trajectories: those related to the *level*, and those related to *agent behavior*. If AIRL is applied naively to PCG environments, the discriminator is prone to overfitting to level characteristics seen during expert demonstrations instead of focusing on the expert behavior itself. On the one hand, by reducing discriminator training to the SeedEnv – the set of expert demonstration levels – we force the discriminator to focus on trajectories and to avoid *overfitting to level characteristics*. On the other hand, SeedEnv must contain enough levels to enable the resulting reward function to *generalize beyond levels* in the reduced ProcEnv sample. We show empirically in the next section that the number of levels required to generalize beyond levels sampled in ProcEnv is much smaller than the number required to avoid overfitting, which may be infeasibly large for PCG environments with many configuration options.

5.5 Experimental Results

We evaluate our method on two different PCG environments: Minigrid (Chevalier-Boisvert et al., 2019) and DeepCrawl. For all experiments, we train an agent with PPO algorithm on the ground-truth, hard-coded reward function and then generate trajectories from this trained expert policy to use as demonstrations for IRL. The apprenticeship learning metric is used for IRL evaluation: agent performance is measured based on the ground-truth reward after having been trained on the learned IRL reward model.

We use the state-only AIRL algorithm with all modifications described in Section 5.3 to learn a reward function in all experiments. We also trained policies with state-only GAIL but, as it is not an IRL method, we cannot re-optimize the obtained model, so we instead transfer the learned policy from the SeedEnv to ProcEnv.



(a) (b)

Figure 5.3: Experimental results on MiniGrid. (a) mean reward during training for: our DE-AIRL with different numbers of seed levels on both SeedEnv and ProcEnv, naive AIRL with different numbers of demonstrations on ProcEnv, and expert performance on ProcEnv. (b) discriminator loss during training on either SeedEnv (our approach) or ProcEnv (naive AIRL).

To highlight the importance of our SeedEnv approach for learning good rewards in the context of PCG environments, we perform the following ablations for each task:

- **DE-AIRL (ours)**: We train a reward function on SeedEnv and use it to train a PPO agent on ProcEnv. We show results for a varying number n of seed levels in SeedEnv(n).
- **AIRL without disentanglement**: We train a reward function on SeedEnv, but without the shaping term $\phi_\omega(s)$ which encourages robustness to level variation.
- **Naive AIRL**: We apply AIRL directly on ProcEnv and show results for a varying number n of demonstrations.
- **GAIL**: We train a policy with GAIL on SeedEnv and then evaluate it on ProcEnv.

The code for replicating all experiments is open source and available online*.

Performance on Minigrid

Minigrid is a grid world environment with multiple variants. For our experiments, we use the *MultiRoom* task: a PCG environment consisting of a 15×15 grid, where each tile can contain either the agent, a door, a wall, or the goal. See Figure 5.2 for an example screenshot of the environment. The aim of the agent is to explore the level and arrive at the goal tile by navigating through 2 or 3 rooms connected via doors. The shape and position of the rooms, as well as the position of the goal and the initial location of the agent, are random. Each episode lasts a maximum of 30 steps. The ground-truth reward function gives $+1.0$ for each

*Code available at https://github.com/SestoAle/Demonstration-Efficient-AIRL

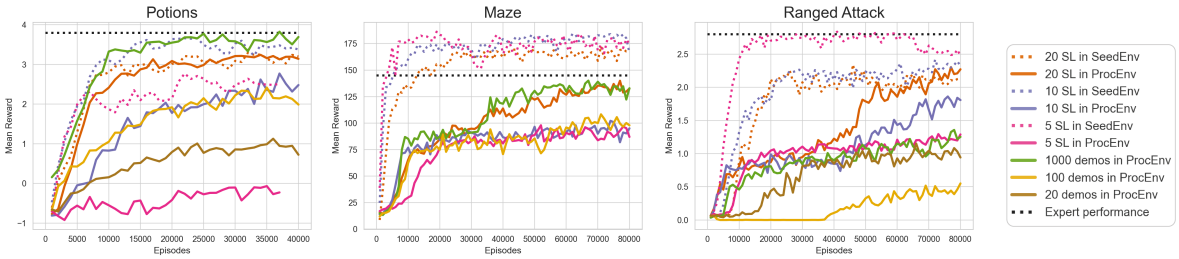


Figure 5.4: Experimental results on DeepCrawl tasks. Mean reward during training for: our DE-AIRL with different numbers of seed levels on both SeedEnv and ProcEnv, naive AIRL with different numbers of demonstrations on ProcEnv, and expert performance on ProcEnv.

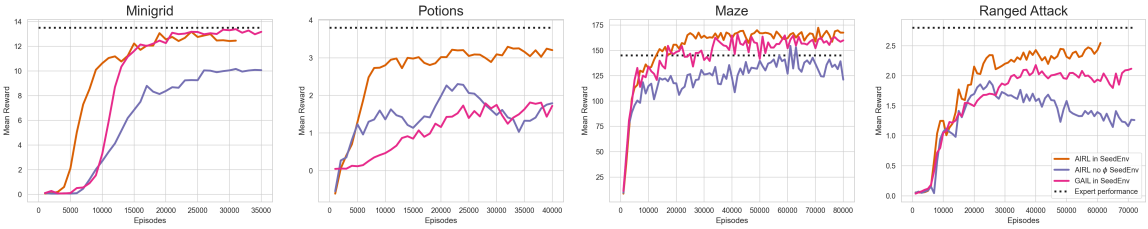


Figure 5.5: Mean reward on SeedEnv throughout training for AIRL without the shaping term, and for GAIL, plus expert performance on ProcEnv for comparison. SeedEnvs consist of 40 levels for Minigrid, and 20 levels for the DeepCrawl tasks.

step the agent stays at the goal location. The action space consist of 4 discrete actions: *move forward*, *turn left*, *turn right* and *open door*.

As the results in Figure 5.3 show, using a SeedEnv with only 40 levels and the associated 40 demonstrations, AIRL is able to extrapolate a good reward function enabling the agent to achieve near-expert performance in ProcEnv. However, if we train a reward model with only 40 demonstrations directly on the full PCG environment, we obtain an inadequate reward function and consequently a poor agent policy. This is also demonstrated by the loss curves: the loss of the discriminator with 40 demonstrations on ProcEnv converges to zero very quickly, indicating the *overfitting to level characteristics* we discussed in Section 5.4. The results also show that 40 is a good number of seed levels for SeedEnv: whereas we find a good policy for SeedEnv with only 30 seed levels, the reward function does not generalize beyond the expert levels to be useful on ProcEnv. Moreover, the plots show that naive AIRL is not successful on ProcEnv with even 100 – so more than twice as many – expert trajectories. Only with 1000 demonstrations does naive AIRL achieve near-expert performance, showing that our DE-AIRL is much more demonstration-efficient.

Figure 5.6: Discriminator loss during training on either SeedEnv (our approach) or ProcEnv (naive AIRL) in the DeepCrawl tasks.

Performance on DeepCrawl

For a complete description of the environment, see Chapter 3. However, for this chapter we do not use the original DeepCrawl game, but three rather different sub-games defined within the game itself (see Figure 5.2):

- **Potions**: The agent must collect red potions while avoiding all other collectible objects. The ground-truth reward function gives $+1.0$ for collecting a red potion and -0.5 for collecting any other item. An episode ends within 20 steps.

- **Maze**: In this variant, the agent must reach a randomly located goal in an environment with many impassable obstacles forming a maze. The goal is a static enemy and there are no collectible objects. The reward function gives $+10.0$ for each step the agent stays in proximity to the goal. Episodes end after 20 timesteps.

- **Ranged Attack**: For this task, the agent has the two attack actions: melee attack and ranged attack. The goal of the agent is to hit a static enemy with only ranged attacks until the enemy is defeated. The ground-truth reward function gives $+1.0$ for each ranged attack made by the agent. The levels are the same as for the Potions task, plus a randomly located enemy. Episodes end after 20 timesteps.

Even for the more complex DeepCrawl tasks, the results in Figure 5.4 show that our demonstration-efficient AIRL approach allows agents to learn a near-expert policy for ProcEnv with few demonstrations: in two of the three tasks only 20 demonstrations are necessary, while for the Ranged Attack task 10 already suffice. Similar to Minigrid, the naive AIRL approach directly applied on ProcEnv does not achieve good performance even with 100 demonstrations – so with more than five times as many demonstrations. With 1000 demonstrations, naive AIRL reaches similar performance on Potions and Maze, but still not on the Ranged Attack task. Figure 5.6 shows the evolution of discriminator losses which behave consistently with what we have observed for the Minigrid environment.

Table 5.1: Average ground-truth episode reward over 100 episodes on ProcEnv. Our AIRL approach trains an agent directly on ProcEnv using the reward model learned on SeedEnv, whereas this is not possible for GAIL, hence the GAIL policy is trained on SeedEnv and then transferred to ProcEnv.

	Minigrid		DeepCrawl			
	Seed levels	MultiRoom	Seed levels	Potions	Maze	Ranged Attack
DE-AIRL (ours)	40	**12.19**	20	**3.78**	**141.87**	**2.30**
GAIL	40	9.00	20	1.71	33.77	1.01
	100	9.21	100	2.38	66.00	1.11

Importance of disentanglement

We claimed above that the use of a disentangling IRL algorithm like AIRL is fundamental for PCG games. We test this experimentally by training an AIRL reward function without the shaping term $\phi(s)$ on a SeedEnv. As the plots in Figure 5.5 show, this modified version does not achieve the same level of performance as the full disentangling AIRL on all tasks. We believe this is due to the variability of levels in SeedEnv: removing $\phi(s)$ takes away the disentanglement property, which results in the reward function no longer being able to generalize, even for the small set of fixed seed levels. Similar results were observed by (Roa-Vicens et al., 2019).

We also train a state-only GAIL model on a SeedEnv. On Minigrid and Maze the policy reaches near-expert performance, while on Potions and Ranged Attack it resembles the performance of AIRL without $\phi(s)$. We believe that this discrepancy is caused by the different degree of *"procedurality"* of these tasks: for Potions and Ranged Attack, there are many different collectible objects with procedural parameters – in fact, all entities and their attributes are chosen randomly at the beginning of each episode. For the other two tasks, the number of procedural choices is smaller, consisting only of the static obstacles and no attributes. The degree of procedurality presumably allows GAIL to achieve good results on SeedEnv for Minigrid and Maze, but not for Potions and Ranged Attack. However, on none of the tasks does GAIL reach the level of performance of our demonstration-efficient AIRL approach when transferring policies from SeedEnv to ProcEnv, as shown in Table 5.1. Note that, as we have mentioned before, GAIL is not an IRL method and hence cannot be re-optimized on the ProcEnv environment, contrary to AIRL, so this shortcoming is not unexpected.

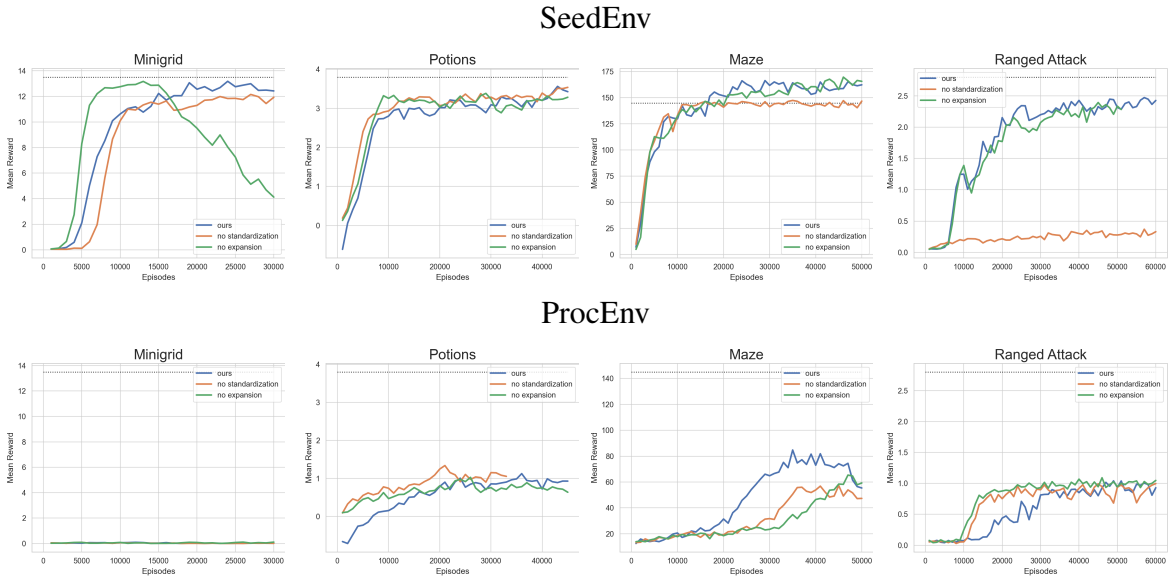


Figure 5.7: Ablation study of the modifications described in section 4 of the main text. The first row represents training in a SeedEnv, while the last row represents training in a ProcEnv. For all the DeepCrawl tasks we used 20 seed levels and 20 demonstrations, while for Minigrid we used 40 seed levels and 40 demonstrations.

Effects of modifications to AIRL

In Figure 5.7 we give an ablation study on both SeedEnv and ProcEnv for the modifications to AIRL proposed in section 4 of the main text. The plots show how the use of both reward standardization and policy dataset expansion yield more stable and better results for the majority of the tasks on both the environment types.

5.6 Implementation Details

In this section we give additional details on the network architectures used for DE-AIRL on the Minigrid and DeepCrawl environments. In Table 5.2 we detail the hyperparameters used for all tasks for both policy and reward optimization.

Similar to Chapter 3 and Chapter 4, limited computational resources prevented this research from repeating experiments multiple times and reporting the mean and standard deviation of the results.

Network Structures

The original authors of AIRL (Fu et al., 2018) use a multilayer perceptron for reward and policy models, however we use Convolutional Neural Networks (CNNs) like (Tucker

et al., 2018). Moreover, we use PPO instead of Trust-Region Policy Optimization (TRPO) (Schulman et al., 2015) as in the original paper.

- **Minigrid.** The policy architecture consists of two branches. The first branch takes the global view of the 15×15 grid, and each tile is represented by a categorical value that describes the type of element in that tile. This input is fed to an embedding layer and then to a convolutional layer with 3×3 filters and 32 channels. The second branch is like the first, but receives as input the 7×7 categorical local view of what the agent sees in front of it. The outputs of the convolutional layers are flattened and concatenated together before being passed through a fully-connected layer of size 256. The last layer is a fully connected layer of size 4 that represents the probability distribution over actions.

  The reward model and the shaping term ϕ_ω have the same architecture. Unlike the policy network, they take only the global categorical map and pass it through an embedding layer, two convolutional layers with 3×3 filters and 32 channels followed by a maxpool, and then two fully-connected layers of size 32 and a final fully-connected layer with a single output. All other layers except the last one use leaky-ReLu activations.

- **Potions and Maze.** The convolutional structure of the policy of the Potions and Maze tasks are the same defined in Chapter 3 without the *"property module"* and the LSTM layer. The reward model takes as input only the global view, then it is followed by a convolutional layer with 1×1 filters and size 32, by two convolutional layers with 3×3 filters and 32 filters, two fully-connected layers of size 32, and a final fully-connected layer with a single output and no activation. The shaping term ϕ_ω shares the same architecture. We used leaky ReLu instead of simple ReLu as used in DCGAN (Radford et al., 2016).

- **Ranged Attacks.** In this case the policy has the complete structure defined in Chapter 3 without LSTM, and the reward model is the same of the previous tasks with the addition of other two input branches that take as input two lists of properties of the agent and the enemy. Both are followed by embedding layers and two fully connected layers of size 32. The resulting outputs are concatenated together with the flattened result of the convolutional layer of the first branch. This vector is then passed to the same 3 fully connected layers of the potion task. The shaping term shares the same architecture.

5.7 Conclusions

In this chapter we have presented an IRL approach, DE-AIRL, which is based on AIRL with a few modifications to stabilize performance, and is able to find a good reward function

Table 5.2: Hyper-parameters for all the tasks. Most of the values were chosen after many preliminary experiments made with different configurations

Parameter	Minigrid	Potions	Maze	Ranged Attack
lr_{policy}	$5e^{-5}$	$5e^{-5}$	$5e^{-6}$	$5e^{-5}$
lr_{reward}	$5e^{-6}$	$5e^{-4}$	$5e^{-4}$	$5e^{-4}$
$\text{lr}_{baseline}$	$5e^{-4}$	$5e^{-4}$	$5e^{-4}$	$5e^{-4}$
entropy coefficient	0.5	0.1	0.1	0.1
exploration rate	0.5	0.2	0.2	0.2
K	3	3	5	3
γ	0.9	0.9	0.9	0.9
max timesteps	30	20	20	20
std_{reward}	0.05	0.05	0.05	0.05

for PCG environments with only few demonstrations. Our method introduces a SeedEnv which consists of only a few levels sampled from the PCG level distribution, and which is used to train the reward model instead of the full fully-procedural environment. In doing so, the learned reward model is able to generalize beyond the SeedEnv levels to the fully-procedural environment, while it simultaneously avoids overfitting to the expert demonstration levels. We have shown that DE-AIRL substantially reduces the number of required expert demonstrations as compared to AIRL when directly applied on the PCG environment. Moreover, the experiments illustrated that the success of our approach derives from the disentanglement property of the reward function extrapolated by AIRL. Finally, we compared to an imitation learning approach, GAIL, and observed that DE-AIRL generalizes better than the GAIL policy when transferring from the expert demonstration levels to the fully-procedural environment.

With DE-AIRL, game designers can use IRL to create qualitative behaviors without engineering reward functions, thus maintaining the "*prior-free*" desiderata and augmenting "*variety*". Moreover, DE-AIRL works specifically with PCG environments and hence we can take advantage of games such as those described in Chapter 4. This greatly improves the general usability of DRL-based tools for non-experts in machine learning. However, in this chapter we demonstrated the effectiveness of DE-AIRL on simple behaviors in simple tasks. Training a complex policy with only IRL is challenging and time consuming. In the next chapter we see how to leverage DE-AIRL to train simple qualitative sub-policies and how to combine them together in order to mitigate these problems while still satisfying our desiderata.

Chapter 6

Policy Fusion Methods[†]

In the last two chapters we first introduced the importance of using procedural content generated environments to avoid retraining in the face of design changes, and subsequently we described a new algorithm that combines inverse reinforcement learning and procedural content generation. The purpose of the latter is to train agents that generalize but at the same time that follow expert guidance without requiring designers to engineer a reward function. However, it has two main drawbacks: on one hand, policies described in Chapter 5 reflect simple behaviors and training on complex, long-time horizons with just inverse reinforcement learning is still a difficult challenge. On the other, in some cases design changes can be significant enough to require an adjustment in the strategy of NPCs, thus requiring some new training.

In this chapter we show how to combine distinct behavioral policies as desired by game designers to obtain a meaningful "fusion" policy which comprises all these behaviors. To this end, we propose four different policy fusion methods for combining pre-trained policies. We further demonstrate how these methods can be used in combination with inverse reinforcement learning in order to create intelligent agents with specific behavioral styles as chosen by game designers, without having to define many and possibly poorly-designed reward functions. Experiments on DeepCrawl and another environment indicate that our proposed entropy-weighted policy fusion significantly outperforms all others. We provide several practical examples and use-cases for how these methods are indeed useful for video game production and designers.

6.1 Introduction

In this chapter we are interested in training Non-Player Characters (NPCs) that exhibit specific behavioral styles or attitudes *chosen by designers*. However, achieving this using

[†]Portions of this chapter appeared in: A. Sestini, A. Kuhnle, and A. D. Bagdanov, "Policy Fusion for Adaptive and Customizable Reinforcement Learning Agents", published in the *Conference on Games (CoG)*, 2021.

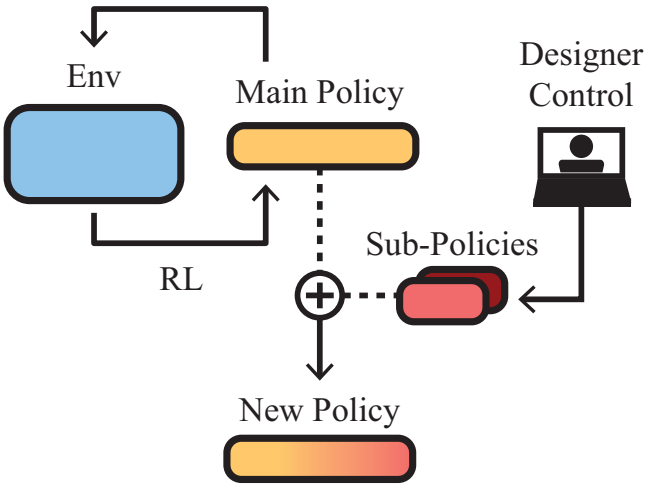


Figure 6.1: Summary of our approach: by combining a previously-trained *main policy* with with one or more sub-policies, we can add local behaviors to the overall agent behavior, adapting it to evolving game design instead of discarding the previously trained agent policy and training a novel one from scratch. In this way, designers can create agents that reflect specific choices in easy and understandable ways.

standard Deep Reinforcement Learning (DRL) training – i.e. via definition and shaping of a complex reward function – can be impractical (Amodei et al., 2016) and expensive since it can require hundreds of thousands of episodes to learn useful policies from scratch. Techniques like Imitation Learning (IL) (Bain and Sammut, 1995) and Inverse Reinforcement Learning (IRL) (Ng and Russell, 2000) can help game designers on the first point, by providing tools to articulate behaviors without requiring handcrafted rewards, however they do not address the sample efficiency problem inherent in DRL training. The policy fusion approaches we propose in this chapter pair well in particular with inverse reinforcement learning in that it is possible to train self-contained, *micro-behaviors* from expert (designer) demonstrations that are then *fused* with the main agent policy to adapt it with the new behavior.

We identify three situations that designers face when training game agents with DRL. In all of the following, we assume the existence of an agent trained with standard DRL which, however, does not reflect designer intent – and which therefore requires adaptation to:

- **Enhance performance**, when the agent trained with standard RL does not meet performance requirements – often due to a sparse or poorly crafted reward function. In standard RL, addressing this usually means tuning hyperparameters, modifying the training setup, adjusting the reward function, and then re-training the agent;

- **Add style**, when the agent does not reflect the qualitative behavior desired by designers. For example, an agent might be supposed to act more "sneaky" or more "aggressive" relative to the current agent behavior. In standard RL, this usually means adapting the reward function accordingly and re-train the agent; and

- **Adapt to new features**, when designers change a detail in the game mechanics such as adding a new object/ability/property/etc. Again, this typically requires re-training a previously trained agent from scratch. Moreover, one cannot rely on the fact that the previous training procedure and hyperparameters will continue to work well in this new version of the environment.

In this chapter we tackle the aforementioned problems and propose methods to combine diversified policies in ways that avoid re-training. As illustrated in Figure 6.1, instead of discarding the previously trained agent policy and training a novel one from scratch, we instead train a sub-policy to handle the intended behaviour aspect. For example, a sub-policy may be trained only to learn how to handle a novel object added by a designer. We then merge the main policy with the sub-policy via policy fusion, which requires no re-training and ideally results in an agent able to properly handle this new object while maintaining the overall skills of the main game policy. We also demonstrate how these approaches can efficiently be used in combination with IRL in order to train agents which reflect the intention of designers without requiring hand-crafted reward shaping. Experiments on two game environments show that our policy fusion approaches outperform fusion methods from the literature and that fused policies achieve the same – in some cases even better – results than re-training the main policy from scratch using engineered reward functions.

6.2 Related Work

Here we review recent work most related to this chapter.

DRL *in* video games. The already cited results from AlphaStar (Vinyals et al., 2019) and OpenAI 5 (OpenAI et al., 2019) demonstrated how DRL can be used to create super-human agents in modern complex video-games, while results such as the work by Ecoffet et al. (2021) show that we can create super-human agents able to surpass human players in the ATARI games of the Arcade Learning Environment (ALE) (Bellemare et al., 2013). However, our motivation is different in that we do not aim to create super-human agents, but rather to facilitate the use of DRL for NPCs as part of game design.

DRL *for* video games. Our goal is to demonstrated how properly use DRL for video game production. As stated in the introduction, Jacob et al. (2020) argued that industry does not need agents build to "beat the game", but rather to produce credible and human-like behaviors. Results such as the work by Delalleau et al. (2019), Alonso et al. (2021), and Pleines et al. (2019) are other notable examples of applying DRL to commercial video games. Within this context, Procedural Content Generation (PCG) has recently gained a lot of attention: in Chapter 4 we noticed that diverse environment distributions are essential to adequately train and evaluate RL agents for video game production, as these kinds of environments enable generalization of agents when faced with design changes.

Inverse Reinforcement Learning. IRL refers to techniques that infer a reward function from human demonstrations, which can subsequently be used to train an RL policy. Adversarial Inverse Reinforcement Learning (AIRL) (Fu et al., 2018) is a state-of-the-art IRL method, which is enhanced by DE-AIRL (Chapter 5) to work with PCG environments. Other IRL methods different from AIRL, which were tested on simple and static environments, are the works by Brown et al. (2019), Christiano et al. (2017), and Ibarz et al. (2018). There are only few examples of applying IRL for video game agents: Tucker et al. (2018) try to use AIRL on ALE without success, while Source of Madness used imitation learning to create diverse game agents in a commercial video game (Carry Castle Studio, 2022).

Ensemble methods for RL. The use of ensemble methods in RL refers to the practice of combining two or more RL algorithms to increase their performance. Wiering and Van Hasselt (2008) survey different methods to combine multiple RL algorithms. Other examples of ensemble methods in RL are the works from Faußer and Schwenker (2011), Hans and Udluft (2010), Peng et al. (2016), Xu et al. (2020), and Peng et al. (2019). However, all of these techniques combine models but with the same goal, instead our aim is to combine policies with different objectives, possibly even orthogonal ones.

Multi-objective learning is motivated by a similar goal, and attempts to combine different reward functions during training to create a complex agent (Vithayathil Varghese and Mahmoud, 2020; Huang and Ontañón, 2020; Brys et al., 2014b,a). Concurrently to our work, Aytemiz et al. (2021) started to tackle a similar problem to ours with a multi-objective approach. These approaches, however, try to combine different objectives during training. Instead, our aim is to combine various policies without re-training of agents.

The *policy fusion* approaches we describe in this chapter are distinct from ensemble methods and multi-objective learning. Our objective is to combine distributions of different policies after training, while ensemble methods combine decisions and multi-objective learning combine reward functions.

6.3 Policy Fusion Methods

This section relies heavily on our DE-AIRL algorithm, and for more details we refer the reader to in Chapter 5.

When game designers want to create game agents with DRL for commercial video games, they must face a large number of challenges. For example, they go through many iterative design choices that change the environment and force agents to adapt with it (Zhao et al., 2020); or they need to adapt the final agent behavior because it does not reflect the designers intention (Jacob et al., 2020); or they need to test new features added to the game under time constraints.

Suppose designers have an agent previously trained with DRL and it must be adapted to a game change. The simplest but most expensive solution is to change something in the training set-up – like reward function or environmental dynamics – and restart the training

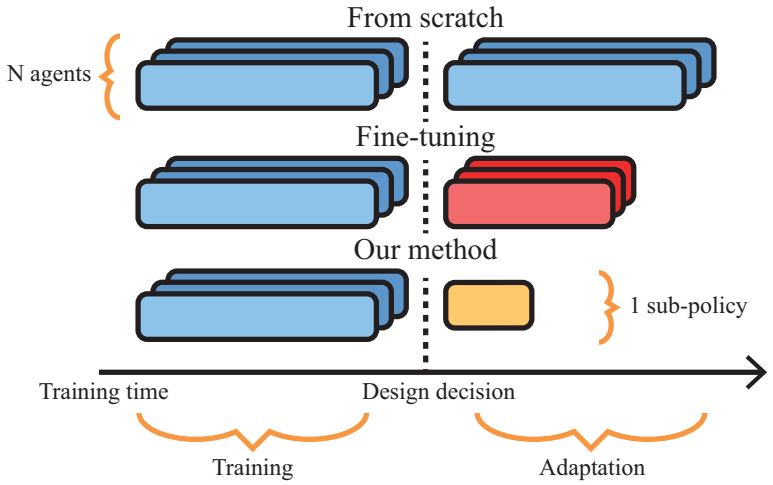


Figure 6.2: Training time for different adaptation methods. The easiest approach is to simply re-train all agents from scratch, but this also requires the most time and assumes the training procedure and hyperparameters remain valid after design changes. Instead, with our policy fusion methods we need to only train a single sub-policy without touching the already-trained agents, requiring about a tenth of the time compared to re-training (see Section 6.4 for more details).

from scratch. Suppose that to train an agent from scratch will take T hours. Usually, a video game does not only have one agent, but multiple ones with different behaviors or characteristics. Hence, if we suppose that there are N agents, training all of them from scratch will take $N \times T$ hours every time a design decision is made.

Another possibility is to fine-tune the agents every time these decisions are made. To fine-tune an agent we need to change something in the training set-up and continue the DRL training of the previously-trained agent. Therefore, we must spend t_{ft} hours, with generally $t_{ft} \leq T$. However, since there are N agents we need to fine-tune all of them, resulting in a total of $N \times t_{ft}$ hours. Moreover, fine-tuning an agent after a design decision is not always trivial. It is known that DRL suffers from overfitting (Justesen et al., 2018) that can render the process of fine-tuning very difficult. However, this can be mitigated with PCG (see Chapter 4. Finally, both fine-tuning and training from scratch often require hyper-parameter tuning which may increase the overall training time.

Our objective is to reduce training time and to avoid the re-training of all agents every time we make a design decision. Our main idea is to train sub-policies that explain locally some type of behavior that designers want to teach to agents. Then, we need a policy fusion method that can combine the main policy with the new sub-policy without losing the skills from its main training. In this way, we get an agent that is now able to adapt to the new behavior. For example, suppose we have a well-trained agent but we add a new usable object to the game, which the agent never saw during training and therefore is unable to use it. Designers could now train a sub-policy which learns only how to best use that object. Then,

they can combine the previously trained policy with the new sub-policy in order to "teach" the agent to properly use the new feature.

Suppose we spend t_{sp} hours to train the sub-policy, then most likely $t_{sp} \leq t_{ft} \leq T$. This method is completely independent from the number N of previously trained agents, because we need to only train 1 sub-policy and combine it with any of the previously trained agents. So, the total training time in this case is just t_{sp}. Figure 6.2 shows an example of the different training times.

Policy Fusion Methods

We propose four different policy fusion methods for DRL to combine different policies. The first two are simple approaches often used for ensemble methods in RL (Wiering and Van Hasselt, 2008; Peng et al., 2019), while the other two are novel techniques proposed by us in this chapter.

Suppose we have a main policy π_0 and a set of sub-policies π_k for $k = 1, \ldots, K$. Each policy takes as input a state s_t and returns a probability distribution over the same discrete action-space. As is common in the reinforcement learning literature, we will write the policy as $\pi_i(a|s_t)$ to indicate that it is a distribution over actions conditioned on state s_t.

To adapt the main policy to include the behaviors of the sub-policies, without the need for retraining, we propose the following fusions methods which result in a fusion policy π_f:

- **Mixture Policy (MP)**: the resulting policy is the average of the main and all sub-policies:

$$\pi_f(a|s_t) = \frac{1}{K+1} \sum_{k=0}^{K} \pi_k(a|s_t). \tag{6.1}$$

- **Product Policy (PP)**: the resulting policy is the product of the main and all sub-policies:

$$\pi_f(a|s_t) = \frac{1}{Z} \prod_{k=0}^{K} \pi_k(a|s_t), \tag{6.2}$$

  where Z is the normalization constant required to make π_f a probability distribution.

- **Entropy-Threshold Policy (ET)**: we compute the entropy of all policies at state s_t and find the sub-policy k^* with minimum entropy:

$$\mathcal{H}_k = -\ln \frac{1}{|A|} \sum_a \pi_k(a|s_t) \ln \pi_k(a|s_t) \tag{6.3}$$

$$k^* = \underset{k=1,\ldots,K}{\operatorname{argmin}} \mathcal{H}_k, \tag{6.4}$$

  Where $|A|$ is the cardinality of the state space. Then, if $\mathcal{H}_{k^*}$ is less then $\mathcal{H}_0$ plus threshold ϵ, we perform the action following the sub-policy, otherwise we perform the

action following the main policy:

$$\pi_f(a|s_t) = \begin{cases} \pi_{k^*}(a|s_t) & \text{if} \quad \mathcal{H}_{k^*} < \mathcal{H}_0 + \epsilon \\ \pi_0(a|s_t) & \text{otherwise} \end{cases} \tag{6.5}$$

- **Entropy-Weighted Mixture Policy (EW)**: the resulting policy is a weighted average of the main policy and the minimum-entropy sub-policy identified using Equation 6.4:

$$\pi_f(a|s_t) = \mathcal{H}_{k^*} \times \pi_0(a|s_t) + (1 - \mathcal{H}_{k^*}) \times \pi_{k^*}(a|s_t). \tag{6.6}$$

6.4 Experimental Results

We performed experiments on two different environments to compare the policy fusion methods. In the first experiment, we combine two independent policies trained with hard-coded reward functions in order to understand the performance of each fusion method. For this, we used the MiniWorld environment, a minimalist 3D interior environment simulator for reinforcement learning and robotics research (Chevalier-Boisvert, 2018). In the second set of experiments, we used the DeepCrawl environment (see Chapter 3 for a complete description) to evaluate how policy fusion methods can be used in combination with Inverse Reinforcement Learning (IRL).

Results on MiniWorld

For our first experiment we use the `PickUpObjs` variant of MiniWorld. In this environment, there is a single large room in which the agent must collect objects of two types: red boxes and green balls. A maximum of 5 objects are spawned in random positions. The observation space of this environment is a single RGB image of size $(80, 60, 3)$.

We train two policies with different, hard-coded reward functions. Our main policy π_0 is trained to collect all the red boxes, with reward:

$$R_0 = +1 \ \text{ for collecting a red box.} \tag{6.7}$$

A single sub-policy π_1 is then learned with the aim of collecting all the green balls:

$$R_1 = +1 \ \text{ for collecting a green ball.} \tag{6.8}$$

The two policies use the same network architecture, consisting of three, stride 2 convolutional layers of size 5, 3 and 3, respectively, and with 32, 32, and 64 channels, respectively. These are followed by two fully connected layers of size 256 and a final fully connected layer followed by a softmax to represent the action distribution. The action space consists of 5 discrete actions: turn left, turn right, move forward, move backward and pick up.

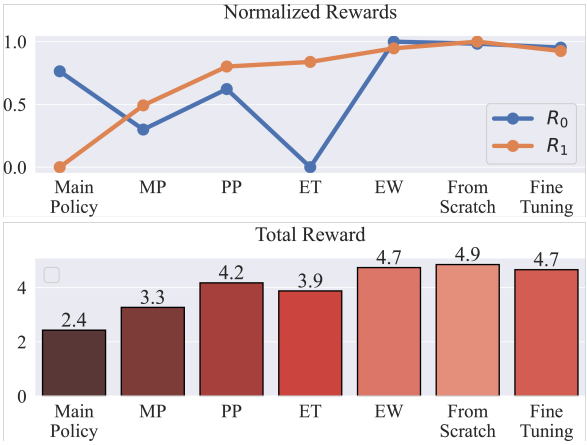


Figure 6.3: Results of the different fusion methods in the MiniWorld environment. The plot shows the normalized rewards R_0 and R_1 using the methods described in Section 6.3. The plot below shows the total reward $R_0 + R_1$ achieved by the respective combined policies. Numbers are averages over 1,000 episodes.

After training the two policies with PPO algorithm, we combine them using the methods proposed in Section 6.3. As baselines, we also train a policy from scratch using the combination of $R_0 + R_1$, as well as a policy fine-tuned for the combined reward $R_0 + R_1$ after being trained first with only R_0. Whereas the fusion methods do not involve further learning, the two baseline policies represent "upper bounds" which are trained to optimize the combined reward. For this first experiment, both the main policy and the sub-policy were trained for 1000 episodes, while the baseline was trained for 3000 episodes.

Figure 6.3 shows the results of these experiments. As the plots illustrate, each policy fusion method improves upon the overall performance of the agent with respect to the combined reward. However, our proposed fusion approach EW outperforms all other methods and achieves the same performance level of the policies trained from scratch or fine-tuned.

This first experiment shows that our proposed policy fusion methods can indeed combine different policies to achieve more complex behavior. The results indicate that, while all yield some improvement, the EW method is by far the best overall, followed by PP. This is a trend we observed in all subsequent experiments.*

Results on DeepCrawl

For a complete description of the environment and the training set-up, refer to the Chapter 3. As before, we used PPO to train agents in this environment, and they all use the Dense Embedding architecture proposed in Chapter 4. We train an agent with the hard-coded reward function in Equation 3.1 as the main policy π_0. The training of this agent reflects the "Ranger" training described in Chapter 3. Subsequently, we train one or more sub-policies and merge

*Code to replicate the experiments is at https://github.com/SestoAle/Policy-Fusion-RL

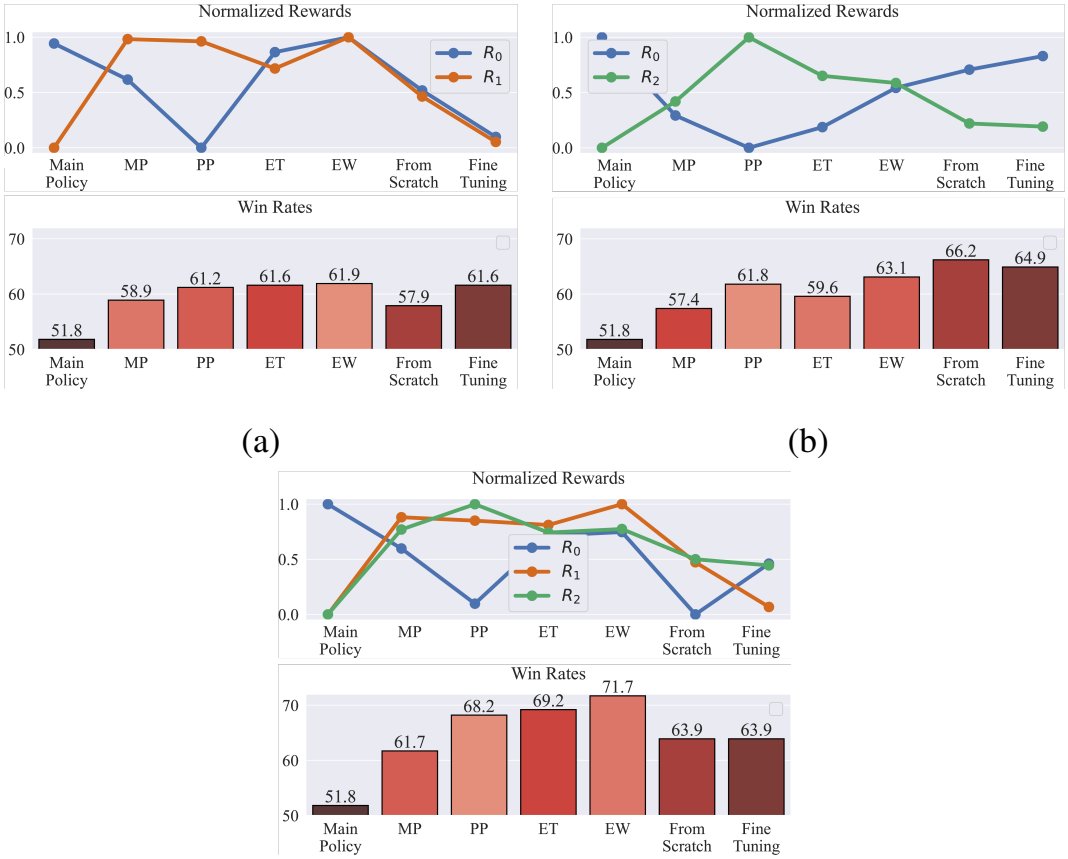


Figure 6.4: Results for the *enhancing performance* use-case. In all figures, the top plot shows the normalized rewards for each fusion method: R_0 refers to the original environment reward, R_1 refers to the reward used for training sub-policy π_1, and R_2 to the reward used for training sub-policy π_2. The bottom plot shows the win rate of the combined agents versus the base agent π_0. (a) results for the combination $\pi_0 + \pi_1$. (b) results for $\pi_0 + \pi_2$. (c) results for $\pi_0 + \pi_1 + \pi_2$. Numbers are averages over 1,000 episodes.

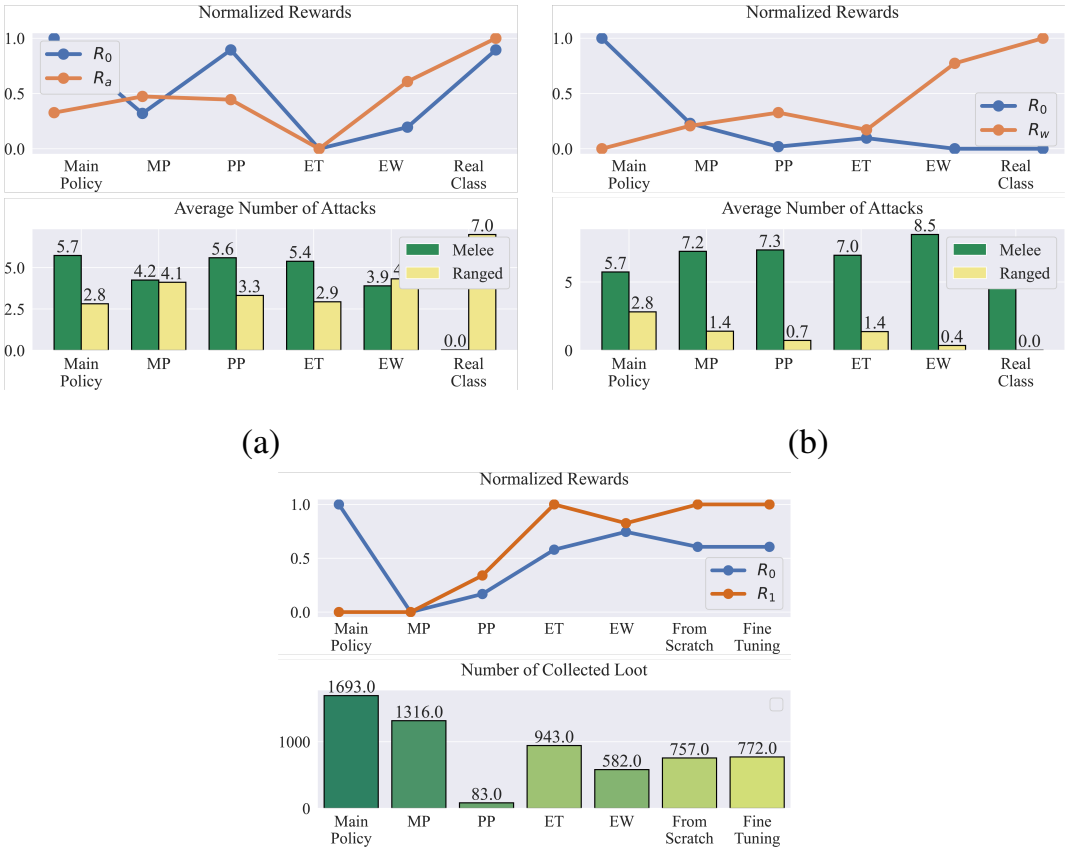


Figure 6.5: Results for the *adding style* use-case. In all figures, the top plot shows the individual normalized rewards for the different fusion methods: R_0 refers to the original environment reward, and R_* refers to the reward used to train sub-policy π_*. The bottom plot shows some qualitative statistics which define the "style" of the agent. (a) combination $\pi_0 + \pi_a$, where π_a is a sub-policy trained to act like an "archer". Numbers are averages over 1,000 episodes. (b) combination $\pi_0 + \pi_w$, where π_w is a sub-policy trained to act like a "warrior". (c) combination $\pi_0 + \pi_1$, where π_1 is a sub-policy trained to avoid loot in the map.

those with the main policy. Each of the following sub-policies are trained with DE-AIRL, using the reward approximators of Chapter 5. The majority of demonstrations for DE-AIRL come from a human expert. In the next section we describe the experiments we conducted, each of which are instances of a use-case outlined in Section 6.1.

Enhancing performance. Suppose that training with the hard-coded reward function does not result in competitive agents. For example, they do not use certain objects in the map which designers believe would improve their win rate. This can be caused by a badly-designed reward function or a sub-optimal training setup.

To explore this use case, we created two different sub-policies π_1 and π_2 for our first experiment. The first is trained with DE-AIRL to collect and use a specific object in the map, while the second is trained from demonstrations to get all loot which increases agent statistics. We then combine $\pi_0 + \pi_1$ and $\pi_0 + \pi_2$, and let the combined agents fight against the base agent π_0. Pitting agents against the base agent allows us to have some quantitative measure about the different policy fusion methods, as our aim here is to create agents that increase their win rate.

As Figure 6.4(a) and 6.4(b) show, combining the main policy with π_1 and π_2 increases the win rate of the main agent for all fusion methods, with EW being the best, followed by PP. Moreover, all methods decrease the original reward function of the environment. This indicates that the reward is probably not optimally designed in order to yield the most competitive agents (in terms of win rate).

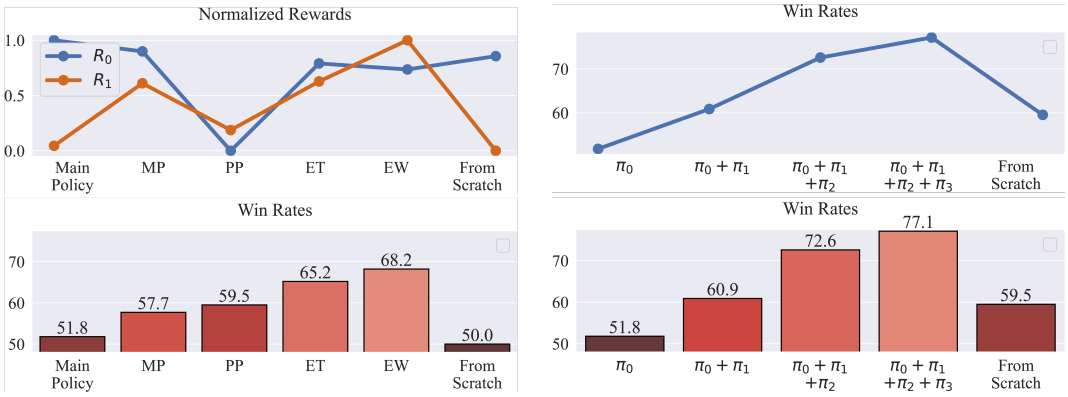
The experiment shows that combining policies can increase the win rate of an agent by more than 10%. Among the methods, EW is the one that decreases the original reward the least while at the same time improving the sub-policy reward. In the π_2 experiment, the EW method even outperforms training from scratch and fine-tuning. This is probably due to the difficulty of combining a hard-coded reward function with a learned one, since these rewards have different scales.

Since both sub-policies individually already increase the competitiveness of the main agent, we also tried to combine them all: $\pi_0 + \pi_1 + \pi_2$. As we expected, Figure 6.4(c) shows that the combination of all three policies using EW results in an even more competitive agent which wins more than 70% of the games against the base policy π_0. Furthermore, EW clearly outperforms the other methods, including training from scratch and fine-tuning.

Adding style. This use case is about training NPCs that exhibit certain behavioral styles specified and controlled by developers. For example, designers may want an agent to act more "sneakily" or more "aggressively". It is very difficult to computationally specify subjective styles, and expect that the main agent π_0 may likely not reflect the qualitative behavior a designer wants. Since we are not interested in the *competitiveness* of the agent here, but rather being able to control the behavioral aesthetics of NPCs, we conduct a qualitative analysis of agent behavior.

We first train a sub-policy π_1 with DE-AIRL which *avoids* loot in the map while fighting against the opponent. This is an interesting experiment for two reason: on the one hand, we add a new style to the main behavior, while, on the other hand, we are trying to limit the agent avoid a behavior it has already learned during training of π_0. Figure 6.5(a) shows that EW method continues to outperform the others, as it decreases the main reward the least while simultaneously augmenting it with the style demonstrated by the designers. Other interesting observations here are that MP does not work at all, while the PP method does add the intended style to the behavior but deviates a lot from the main policy.

For the second experiment in this use-case changed the combat style of π_0 to emulate the style of the other two agent classes in defined in Chapter 3 – Warrior and Archer. These



(a) (b)

Figure 6.6: Results for the *adapting to new features* use-case. (a) the top plot shows normalized rewards for the different policy fusion methods: R_0 refers to the original environment reward and R_1 refers to the reward used for training sub-policy π_1. The bottom plot shows the win rate of the combined agent versus the base agent. All experiments use the E_1 version of the game. (b) both plots show the win rate of the combined agent versus the plain agent with different combinations of sub-policies with π_0. Numbers averages over 1,000 episodes.

two classes are distinguished from each other by the type of attacks they perform in combat: the Warrior relies on melee attacks, while the Archer only on ranged attacks. In contrast, the Ranger, our main policy, performs both melee and ranged attacks. We train Warrior and Archer sub-policies π_w and π_a from demonstrations provided by the pre-trained agents and then combine them with the main policy: $\pi_0 + \pi_w$ and $\pi_0 + \pi_a$. The results are shown in Figures 6.5(b) and 6.5(c). Indeed, with $\pi_0 + \pi_w$ we are able to modify the Ranger behavior act as a Warrior. With $\pi_0 + \pi_a$ we can change the behavioral style of the Ranger *towards* the Archer, i.e., to perform more ranged than melee attacks, but the combination does not perfectly emulate the Archer behavior. Again, in both cases the EW method outperforms all other fusion methods.

Adapting to new features. suppose an agent π_0 was trained on a certain version of the environment, E_0. Later, after design decisions, some aspect of the game has changed. For example, maybe a new usable item in the map was added, to arrive at a new version of the environment E_1. At this point, designers require an agent π_1 which is aware of and able to properly exploit this new feature. They have two choices: re-train from scratch all previously-trained agents, or train a sub-policy in order to teach it how to use only the new object, and then augment π_0 with this sub-policy.

For our first experiment in this use-case, we added a new usable object to the game. We designed this object so that an NPC using it will have an advantage agents that do not. This way, use of the new object will be reflected in the win rate of π_1 over π_0. We train a sub-policy to exploit this feature and combine it with π_0 using policy fusion. This requires

an "adapter" which hides the new object from π_0, so to avoid confusion of encountering a new object never seen during training. As baseline, we train an agent from scratch in E_1 but with the original DeepCrawl reward defined in Equation 3.1. Figure 6.6(a) shows the results: the combined agent is perfectly capable of using the new object, with EW outperforming all other fusion methods. An interesting observation here is that training from scratch does not learn to make use of the new feature: it just ignores the object and reaches the same level of performance as π_0. This demonstrates that training from scratch in the face of design changes is not a trivial task: we may need to tune hyperparameters or even adapt the reward function to train an agent able to incorporate the new feature into its behavior. But, as we mentioned before, engineering a well-designed reward function for very complex behavior is not easy, and training from scratch takes more time than training only a sub-policy. Our approach enables quick and efficient adaptation to new features in a game environments, which facilitates testing without having to wait for agents to re-train.

Our final experiment simulates multiple, iterative design changes: since we know that fusion methods can adapt to a single change in the environment dynamics, what if we add two more features? For this, we first added a new usable object which increases the statistics of agents that collect it, and then we added an instant death tile. Clearly, an agent that adapts to these changes will have an advantage. For evaluation we compare two agents in an environment with both new features present. The first agent π_0 was trained before these features were added. The second agent is π_0 plus all the sub-policies trained how to use the new features. The baseline is an agent re-trained from scratch on the environment with both changes. Since we have established that EW is the best of the considered fusion methods, we only used this technique here. Figure 6.6(b) shows the results for different combinations of sub-policies and π_0. Indeed, we can even combine even all 4 policies and achieve better results than the re-training baseline. We believe that after some hyperparameter tuning, the baseline will likely be able to perform better than our method. However, this would come at a very high cost in term of human effort and time.

Training Times

Taking the last experiment of Section 6.4 as illustrative example, training the main policy takes about 66 hours, while training a single sub-policy requires about 6 hours. Using policy fusion thus requires only an additional 6 hours for each adaptation, whereas it takes another 66 hours per adaptation to re-train main policy from scratch. Our method is about 10 times faster than the standard approach. All training was performed on a NVIDIA RTX 2080 SUPER GPU with 8GB RAM.

6.5 Conclusions

Training intelligent agents in complex environments using deep reinforcement learning is difficult and time-consuming, and moreover requires specialized knowledge of both the domain and state-of-the-art deep learning techniques. In this paper we presented several policy fusion methods that can *combine* policies with the aim of adapting or modifying behavior in the face of game design changes – all without requiring retraining of agents to cope with these changes. Our experiments clearly show that the Entropy-Weighted Mixture (EW) fusion technique significantly outperforms the others methods, and in some cases even surpasses the agent re-trained from scratch or fine-tuned using the combined reward function. Of all methods, EW strikes the best balance of maintaining the original policy behavior while simultaneously augmenting it with the sub-policy's new "style".

Our experiments also showed how these fusion methods can be used in combination with inverse reinforcement learning to create varied and complex behavior without defining new reward functions, which contrasts to the prevailing perception that IRL is difficult to exploit in high-dimensional state spaces (Zhao et al., 2020).

The techniques proposed so far in this dissertastion have the aim of satisfying the desiderata outlined in Chapter 3. In Chapter 4 we augmented the "*credibility*" and "*imperfection*" desiderata with procedural content generation. In Chapter 5 we added human-likeness enhancing "*credibility*" while still satisfying the "*prior-free*" requirement through demonstration-efficient inverse reinforcement learning. In this Chapter we increased "*variety*" via policy fusion still maintaining all the other desiderata. With our improvements we contributed to solve problems defined in Chapter 2, and have made steps forward the creation of effective game design tools based on deep reinforcement learning. What is missing now is the opinion of professional game developers. In the next chapter we see how we can use concepts, algorithms and theories encountered up to now for an important development step: game testing. Moreover, we ask professional game designers how likely they are to use such innovative techniques.

Chapter 7

Curiosity-Conditioned Proximal Trajectories[†]

In the previous chapters we described novel algorithms and benchmarks for training NPCs that can play the game with or against the human player. These agents offer the potential of replacing current game AI systems and their inherited problems. However, as stated in Chapter 1, there is another context where autonomous agents and AI systems are important to game development: playtesting. This chapter combines the concepts and theories encountered so far in this dissertation – such as imitation learning and curiosity driven learning described in Chapter 2, the embedding operations of crucial importance in Chapter 3, and the transformer architecture outlined in Chapter 4 – to train agents that can perform automatic gameplay validation.

In this chapter we propose a novel deep reinforcement learning algorithm to perform automated analysis and detection of gameplay issues in complex 3D navigation environments. The curiosity-conditioned proximal trajectories method combines curiosity and imitation learning to train agents that methodically explore in the proximity of known trajectories derived from expert demonstrations. We show how our new algorithm can explore complex environments, discovering gameplay issues and design oversights in the process, and recognize and highlight them directly to game designers. We also propose a visual analytics interface to aid interpretation of results from the method. This interface transforms information from complex models into interpretable and interactive visual forms. We further demonstrate the effectiveness of the algorithm in a novel 3D navigation environment which reflects the complexity of modern video games. Our results show a higher level of coverage and bug discovery than baseline methods, demonstrating that our method can be a useful tool for game designers to automatically identify design issues. Moreover, our experiments show that the visual explanations provided by the analytics interface result in a significant increase

[†]Portions of this chapter were submitted to: A. Sestini, L. Gisslén, J. Bergdahl, K. Tollmar, and A. D. Bagdanov, "Automated Gameplay Testing and Validation with Curiosity-Conditioned Proximal Trajectories", *Transactions on Games (ToG)*, 2022.

in user trust and acceptance of automated playtesting and increased confidence in the use of machine learning techniques for video game development. A video compilation of the results of this chapter is available online*.

7.1 Introduction

Playtesting plays a crucial role in the production of modern video games. The goal of automated gameplay testing is to free up some of the human resources to allow them to do more meaningful testing such as measuring gameplay balance, difficulty, and potential rate of retention.

Recently, automated testing approaches have been proposed to mitigate total reliance on human testers by training AI-based agents to explore large game scenes (Gordillo et al., 2021). Automated exploration through intrinsic motivation is a step in the right direction, however, we argue that it is not enough. First, we need agents capable of learning the world around them and efficiently understanding the difference between different states. Second, we need agents that recognize the difference between good and bad trajectories in order to recognize which paths are "broken" ones.

Based on these considerations, in this chapter we propose a novel Reinforcement Learning (RL) approach able to train agents which can explore and analyze a large 3D environment composed of complex navigation challenges. We call the approach automated gameplay testing and validation with Curiosity-Conditioned Proximal Trajectories (CCPT). As shown in Figure 7.1, our technique uses a combination of *imitation learning* and *curiosity*. These two concepts work in tandem by guiding the agents to not only imitate recorded expert demonstration trajectories, but to also explore in their proximity. In particular, we propose an *exploration-conditioned intrinsic reward function* leading to agents that do not just learn to be curious, but that learn what it means to behave curiously. The model enables CCPT not only to find bugs and gameplay issues, but to automatically identify, filter and highlight them among the massive quantity of information collected by the agents from their interactions with the environment.

While CCPT is able to point out trajectories containing bugs, these results are expressed as gameplay metrics coming from complex models that are understandable only to experts intimately familiar with the CCPT algorithm and its theoretical foundations. Since this tool is aimed at designers that typically are non-experts in machine learning, the expertise needed to understand the details of said results limits the direct accessibility of CCPT. To address this we propose a purpose-built visual interface designed to convey the massive amount of information yielded by the CCPT algorithm in a concise and easily interpretable way to designers. We show how designers can use the interface to understand not only *which* are the trajectories containing bugs, but also *where* the bug is in the trajectory, *why* the agent has marked it as bug, and *how* the agent was able to exploit it. Graphical representations of

*Video available at: https://tinyurl.com/ccpt-test

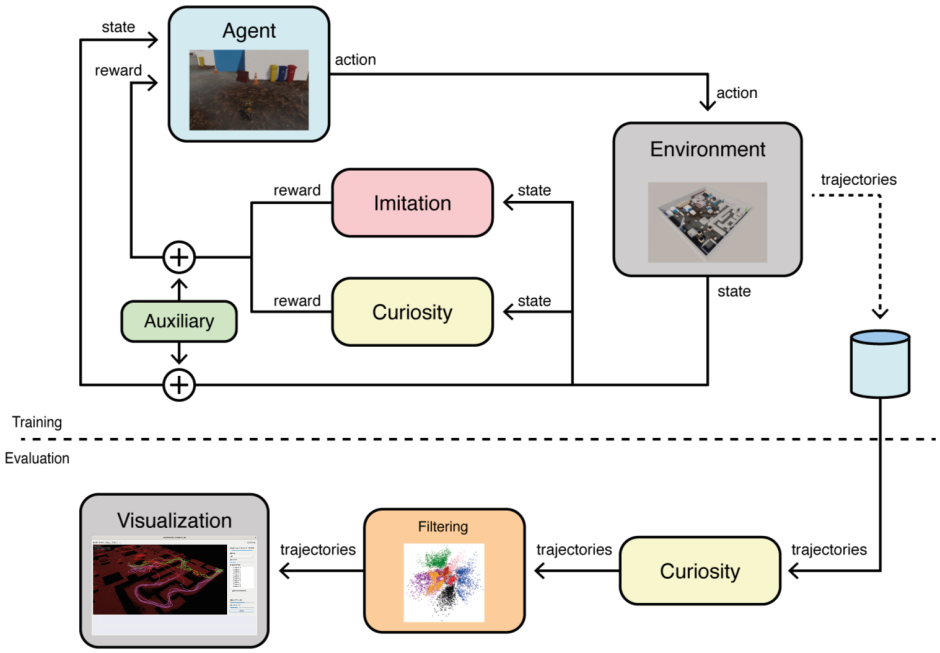


Figure 7.1: Overview of our approach. By combining imitation learning and curiosity, we train agents to playtest a large game scenario. All information gathered during training is saved and eventually filtered upon evaluation through the same curiosity module used when training. Our *exploration-conditioned intrinsic reward function* enables us to filter and highlight trajectories that contain bugs and design oversights like missing collision boxes or gameplay glitches.

automated gameplay test metrics make it easier for designers to make informed decisions regarding future improvements to the game. Moreover, the CCPT visual interface is designed to be completely engine-agnostic which increases the accessibility of the tool by eliminating the typically heavy engineering work needed for re-implementation in different game engines.

The key contributions of this chapter are: a new open source environment for complex 3D navigation challenges, suitable as testbed for training both exploration and navigation agents; a novel neural network architecture for navigation and exploration agents and an empirical demonstration of its effectiveness; a new exploration algorithm which, thanks to its use of expert demonstrations, is able to deliberately explore the proximity of desired trajectories; an interactive tool that uses the CCPT algorithm to improve the interpretability of machine learning models for game validation and testing; and quantitative experimental validation of our approach compared to the state-of-the-art as well as qualitative validation based on a user study involving professional game designers.

7.2 Related Work

Here we review recent work most related to this chapter.

Automated Play Testing. Several recent studies have investigated the use of AI techniques to perform automatic play testing, with a focus on maximizing game state coverage. Many of these recent works heavily rely on classical hand-scripted AI or random exploration (Stahlke et al., 2020; Holmgård et al., 2018). However, when dealing with complex 3D environments with difficult navigation challenges we argue that these techniques are not readily applicable due to the high-dimensional state-space. The Reveal-More algorithm also uses human demonstrations to guide the random exploration, although in simple 2D dungeon levels (Chang et al., 2019). Mugrai et al. (2019) developed an algorithm to mimic human behaviour to get more meaningful gameplay testing, but also to aid in the game design.

At the same time, many other works have used reinforcement learning to perform either automatic play testing or complex navigation exploration. Alonso et al. (2021) trained a reinforcement learning agent to navigate a complex 3D environment toward procedurally-generated goals, while Devlin et al. (2021) trained different agents to perform a Turing test to evaluate the human-likeness of trained bots. Closer to our work, Agarwal et al. (2020) trained reinforcement learning agents to perform automated play testing in 2D side-scrolling games, also providing a set of visualizations for level design analysis, and Gordillo et al. (2021) used intrinsic motivation to train many agents to explore a 3D scenario with the aim of finding issues and oversights.

Although this chapter draws inspiration from Gordillo et al. (2021), they based their approach on count-based exploration that may not be feasible when faced with complex environmental dynamics that, due to the tabular nature of such algorithms, explode in complexity (Strehl and Littman, 2008). Moreover, the use of a purely exploration-based technique can slow down coverage time, especially if designers want to test a particular part of the environment. Finally, even given good visualizations of the results, this approach does not tell where, when, and how the issues are found, but rather leave the burden of recognizing them to the designers.

Imitation Learning. Similar to Chang et al. (2019), we make use of demonstrations to guide the exploration. However, for this aim we use a state-of-the-art imitation learning algorithm. Imitation learning aims to distill a policy mimicking the behavior of an expert demonstrator from a dataset of demonstrations. It is often assumed that demonstrations come from an expert who is behaving near-optimally. Standard approaches are based on Behavioral Cloning (BC) that mainly use supervised learning (Bain and Sammut, 1995; Ross et al., 2011; Knox and Stone, 2009), while more advanced methods are based on adversarial trainings Ho and Ermon (2016); Fu et al. (2018). A relevant method for our discussion is the Adversarial Motion Prior (AMP) algorithm, which is a GAN-based imitation learning method that aims to increase stability of adversarial approaches Peng et al. (2021).

Intrinsic motivation. Intrinsic motivation aims to encourage agents to explore the environment states in the absence of an extrinsic reward. The already mentioned count-based exploration is a natural way to do exploration, although for high-dimensional state spaces it can be infeasible (Strehl and Littman, 2008). Another class of exploration methods rely on

errors in predicting dynamics (Pathak et al., 2017; Burda et al., 2018). These are machine learning techniques for high-dimensional states that aim to push agents to explore never or less-encountered states during training. The interested reader should consult Aubret et al. (2019) for a detailed survey of the state-of-the-art in intrinsic motivation.

Visualization-Based Analysis

Visual analytics tools are important for both deep reinforcement learning approaches and for gameplay validation. Most AI models exhibit an important negative characteristic: a performance versus transparency trade-off. Understanding the reasoning behind their behaviors becomes a necessity when these results drive design decisions (Puiutta and Veith, 2020). Many approaches have been proposed to mitigate the black-box nature of AI models through visualization. Jaunet et al. (2020) created a visual analytics interface for studying the behavior of an agent with memory exploring a level of ViZDoom (Kempka et al., 2016). Druce et al. (2021) proposed a user interface to better understand when the action output of automated agents can be trusted or not.

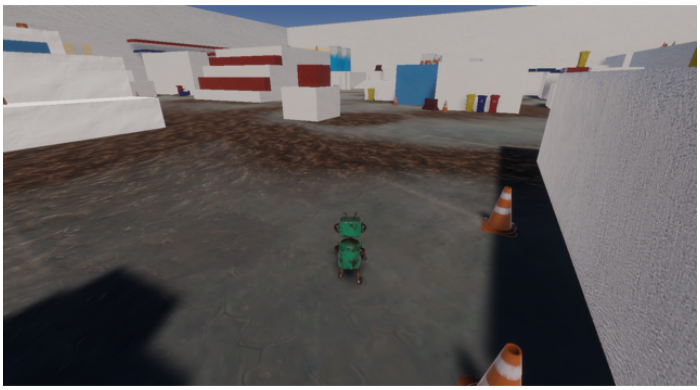
In addition, visual interfaces are already widely used for understanding human tester behaviors and the impact on game design decisions. The survey of visualization-based analysis of gameplay data by Wallner and Kriglstein (2013) gives an overview of the current state of this emerging field of research. Our aim is to combine the usefulness of visual interfaces for game metrics with more complex analytic interfaces for explainable machine learning approaches like our curiosity-conditioned proximal trajectories.

7.3 Curiosity-Conditioned Proximal Trajectories

This section details our approach to train agents guided by expert priors to find bugs and gameplay issues. First, we define the game environment we use for assessing our approach. Second, we detail our RL setup and training algorithm. Finally, we describe how the approach highlights issues among all information it gathers during training.

The Navigation Environment

To validate our approach, and to support continued research, we propose a 3D navigation environment designed to resemble modern game scenarios. A screenshot of the environment is given in Figure 7.2 together with a top-down view of the whole map. The scene is approximately $500\,\mathrm{m} \times 500\,\mathrm{m} \times 60\,\mathrm{m}$ in size and contains a variety of navigation challenges and dynamic elements such as moving platforms and elevators. Agents wishing to explore all secrets contained in the map must learn complex navigation strategies. Our environment is comparable to recent navigation studies (Gordillo et al., 2021; Devlin et al., 2021; Alonso



(a) Screenshot of our environment.

(b) Top-down view.

Figure 7.2: Overview of our proposed environment.

et al., 2021) and is open source and available for anyone who wants to exploit, contribute to, or expand on this research.[†]

Agents spawn in the center of the map and each episode ends after 500 timesteps, with the agent performing 1 action every 10 game frames. There are four goal areas in the environment selected as they are the most difficult spots to reach (indicated by the green arrows in Figure 7.2(b)). The agent has a total of 10 discrete actions: move forward/backward/left/right, move in one of the 4 diagonal directions, jump, and wait. The agent can also perform a double jump while in the air and climb on surfaces of specific elements located around the

[†]Code available at https://github.com/SestoAle/Navigation-Environment.

map. Moreover, since the intent of the chapter is to provide an automated way to detect bugs and glitches in a game scene, we manually introduce such gameplay issues like missing collision boxes and glitches throughout the map.

The state observed by the agent at any instant in time consists of a local perception in the form of a 3D semantic occupancy map, as well as scalar information about physical attributes of the agent (discretized global position, if it is grounded, if it is attached to a climbable surface, if it can perform a double jump, its velocity, and its direction). We show an example 3D semantic occupancy map as inputs to the networks in Figure 7.3. These maps are a categorical discretization of the space and elements around the agent, and each voxel is defined by the semantic integer value of the type of object at that position.

The only extrinsic reward provided is given when an agent reaches an active goal, for which it receives +10 for each timestep it stays inside this area. With such a sparse reward in such a large environment, agents have very little chance of receiving even a single, non-zero reward, thus making training very hard. Moreover, with just the extrinsic reward, even if they learn to reach a goal location, after convergence agents will always follow the same trajectory without exploring for new paths. Instead we need agents able to efficiently arrive to a goal area while continuing to search for undiscovered paths, thus combining rewards for both exploration and imitation.

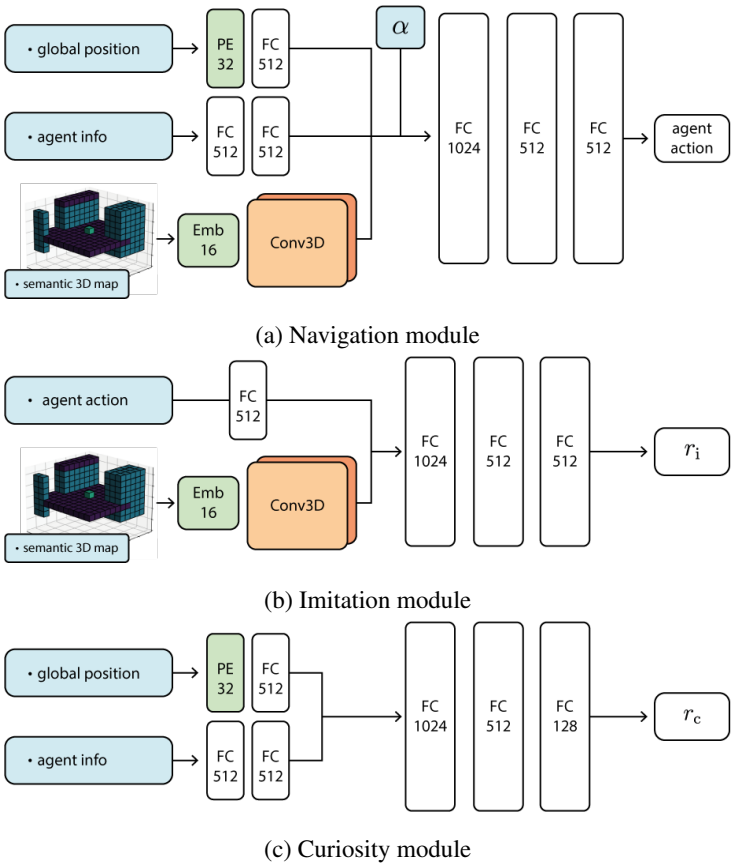
The CCPT Algorithm

In this section we detail the algorithm used to generate agents to perform automated playtesting. We devise our method following a main idea: train agents which explore in the *proximity* of trajectories predefined by an expert in order to automatically identify overlooked issues. Our agents consist of a navigation module plus imitation and exploration sub-modules which both contribute to the reward function used to drive policy learning.

Navigation Module. The navigation module is defined by the policy. As shown in Figure 7.3(a), the policy takes as input both the global information and the local perception defined by the semantic occupancy map. It also takes an auxiliary input that defines the level of exploration followed in a particular episode. Since it is a fundamental part of the reward function, we defer the description of how this affects training to the Reward Function section below. To encode the global 3D position $\mathbf{p} = (x, y, z) \in \mathbb{Z}^3$ of the agent we use positional embeddings similar to the one from Vaswani et al. (2017):

$$\begin{aligned} \mathrm{P}_{i,2j} &= \sin \frac{p_i}{10000^{\frac{2j}{\mathrm{d}}}}, \\ \mathrm{P}_{i,2j+1} &= \cos \frac{p_i}{10000^{\frac{2j}{\mathrm{d}}}}, \end{aligned} \tag{7.1}$$

where $j \in \{0, 1, 2, ..., d/2 - 1\}$, d is the embedding size, p_i the respective components of $\mathbf{p}$ and $\mathrm{P} \in \mathbb{R}^{3 \times d}$ the resulting encoding. We claim that the use of such an embedding is crucial



(a) Navigation module

(b) Imitation module

(c) Curiosity module

Figure 7.3: Overview of the module architectures used in this chapter.

for training agents that navigate and explore these environments. In fact, the general way of encoding such information with normalization or with learned embeddings it is not enough to efficiently understand the difference between different states. In Section 7.5 we support this claim with ablation experiments.

These vectors are concatenated with the other agent information and passed through a feed-forward network. The semantic occupancy map is instead passed through its own 3D convolutional network. All the processed vectors are then concatenated together and passed through an MLP. The policy is trained using the PPO algorithm.

Imitation Module. The imitation module trains the agent to follow the expert trajectories and to guide playtesting toward a particular area. We use the AMP algorithm (Peng et al., 2021), which is built on top of GAIL (Ho and Ermon, 2016). Given a set of expert demonstrations E, the goal is to learn to measure the similarity between the policy and the demonstrations, and to then update the policy via forward-RL. The objective is modeled as a discriminator $D(s,a)$ trained to predict whether a given state-action pair (s,a) is sampled from the demonstration set or generated by running the policy. AMP adopts the loss function

proposed for the least-square GAN (Mao et al., 2017):

$$\mathcal{L}^{\text{AMP}} = \arg\min_D \mathbb{E}_{d^E(s,a)}[(D(s,a)-1)^2] + \mathbb{E}_{d^\pi(s,a)}[(D(s,a)+1)^2], \tag{7.2}$$

where $d^E(s,a)$ and $d^\pi(s,a)$ respectively denote the likelihood of observing a state-action pair in the dataset E or by following the policy π. The reward function for training the policy is then given by:

$$r_{\text{i}}(s_t,a_t) = \max[0, 1-0.25(D(s_t,a_t)-1)^2]. \tag{7.3}$$

To further increase training stability, we apply gradient penalties that penalize nonzero gradients on samples from the dataset (Mescheder et al., 2018).

The state s of the imitation module is described by the local perception of the agent, which is defined by the 3D semantic occupancy map. This is passed through a 3D convolutional network and is concatenated with the action embedding before being fed to a feedforward network, as shown in Figure 7.3(b).

Curiosity Module. The curiosity module is responsible for optimizing coverage of the environment in the neighborhood of expert demonstrations via intrinsic exploration. Instead of using count-based exploration like Gordillo et al. (2021), which can be infeasible for high-dimensional state spaces, we use the Random Network Distillation (RND) algorithm (Burda et al., 2018). The curiosity module gives an intrinsic reward that is higher for novel and less-encountered states. With this reward we can train agents with forward-RL to increase coverage of the environment. for a complete description of the RND algorithm, see Chapter 2.

The more a state is visited by agents, the closer the output of the predictor network will be to that of the target network for that particular state, lowering the prediction error and thus the reward signal for exploration. States encountered following the expert demonstrations will produce low reward values, while for states encountered less frequently the predictor will not be able to perfectly replicate the target, increasing the reward signal and guiding agents toward undiscovered paths.

As shown in Figure 7.3(c), both target and predictor networks take as input the global position, encoded with the positional embedding of Equation 7.1, and the other agent information. These vectors are then concatenated and passed through a feedforward network.

Reward Function. The core of the algorithm lies in the reward function. Our aim is to combine the above modules to derive agents that can explore the proximity of expert trajectories. In this chapter we propose an *exploration-conditioned intrinsic reward function*. Inspired by works like de Woillemont et al. (2021) and Gisslén et al. (2021), or in general by goal-conditioned policies (Andrychowicz et al., 2017), our reward function is:

$$R(s_t,a_t) = \alpha \cdot r_{\text{c}}(s_{t+1}) + (1-\alpha)\cdot r_{\text{i}}(s_t,a_t) + r_{\text{e}}(s_t,a_t), \tag{7.4}$$

where r_c is the reward from the curiosity module, r_i is the reward from the imitation module, r_e is the extrinsic reward from the environment, and $\alpha \in [0,1]$ is a weight hyperparameter that controls the level of exploration versus imitation. The value of α is randomly sampled at the beginning of each episode and remains fixed for all timesteps. If an agent samples a value of $\alpha < 0.5$, the r_i prevails and the reward leads agents to follow the expert demonstrations more closely. When $\alpha = 0$, the reward moves agents toward a perfect replication of expert trajectories. In contrast, if $\alpha > 0.5$ the r_c is dominant and the reward lead the agent to explore more. The greater the α, the farther away from expert priors agents explore. When $\alpha = 1$, the agent completely avoids the expert demonstrations, finding completely new ways to arrive to the goal location. The r_e is needed to make agents arrive in the goal area independently of the sampled α.

In order for the agent to understand in which way it should behave in a particular episode, the sampled α is part of the state space of the agent. Since α controls the reward that the agent receives, we are basically combining curiosity-driven and goal-conditioned reinforcement learning. In this setting we obtain meaningful exploration via curiosity and not just timestep-level randomness, and the result of training is not just a curious agent, but an agent which we can control to behave like the expert or like an explorer just by changing α. Algorithm 1 details the full CCPT agent training procedure.

Algorithm 1 Training with CCPT

input: E dataset of expert demonstrations, ϵ filter threshold
$\pi \leftarrow$ initialize policy
$D \leftarrow$ initialize AMP discriminator
$\hat{\phi} \leftarrow$ initialize RND target network
$\phi \leftarrow$ initialize RND predictor network
$G \leftarrow$ initialize external dataset

while not converged **do** ▷ Training
  $B \leftarrow$ initialize policy experience dataset
  for each episode $i = 1, ..., m$ **do**
    Sample $\alpha_i \sim [0,1]$
    Collect trajectory $\theta_i = (s^0, a^0, r_e^0 .., s^T, a^T, r_e^T)$ by executing π
    for each timestep $t = 1, ..., T \in \theta_i$ **do**
      $r_i^t = \max[0, 1 - 0.25(D(s_t, a_t) - 1)^2]$
      $r_c^t = (\hat{\phi}(s_{t+1}) - \phi(s_{t+1}))^2$
      $R^t = \alpha_i \cdot r_c^t + (1 - \alpha_i) \cdot r_i^t + r_e^t$
      Update reward R^t in θ_i
    end for
    Store θ_i in B
    Store (θ_i, α_i) in G ▷ store all trajectories in the external dataset
  end for
  Update D with $\mathcal{L}^{\text{AMP}}$ with samples from B
  Update ϕ with $\mathcal{L}^{\text{RND}}$ with samples from B
  Update π with $\mathcal{L}^{\text{PPO}}$ with samples from B
end while
return G

Detecting Suspicious Trajectories

The final result of our algorithm is not trained agents, but rather all the information gathered during training. Given the set of all trajectories performed during training, our aim is to find those that evidence game behavior unintended by designers. Since in our case the intended paths are described by the demonstrations, we must find trajectories that arrive at the same goal area defined by experts but are very different from the expected experience.

Thanks to the α component of the reward function in Equation 7.4, for low α values the agent will revert to following the expert demonstrations very closely, thus lowering the rewards output by the curiosity module for states that are very near to those seen in expert demonstrations. In contrast, as α increases agents will explore more and more, and the rewards output by the curiosity module for states explored in this setting will be kept relatively high with respect to those near to the expert demonstrations.

We perform a simple preliminary filtering of trajectories by removing all those gathered with $\alpha < 0.5$. For the remaining trajectories we exploit the values of the curiosity module. Given all trajectories that arrive at the goal location with $\alpha \geq 0.5$, we compute the average curiosity values along the trajectory from the start to the goal location:

$$\hat{r}_{\mathrm{c}}(\theta_i) = \frac{\sum_{t=0}^{T} r_{\mathrm{c}}(s_t)}{T}, \tag{7.5}$$

where $\theta_i = (s_0^i, a_0^i, ..., s_T^i, a_T^i)$ is a trajectory, T is the number of timesteps to arrive to the goal location, and r_{c} is the reward of the curiosity module at the end of the training. We then define the set Θ as:

$$\Theta = \{\theta_i \mid \hat{r}_{\mathrm{c}}(\theta_i) > \epsilon\}, \tag{7.6}$$

where ϵ is a predefined threshold. As we will show in Section 7.5, this set will define the trajectories that are far from the expert demonstrations and likely exhibit unintended game behavior. Algorithm 2 details the full process of filtering and highlighting suspicious trajectories.

Algorithm 2 Filtering with CCPT

input: G dataset of trajectories found during training phase by Algorithm 1
$\Theta \leftarrow$ initialize bugged trajectory set
for trajectory $\theta_i \in G \mid \alpha_i \geq 0.5$ **do** ▷ Trajectory Evaluation
  $\hat{r}_{\mathrm{c}}^i = \frac{\sum_{t=0}^{T} r_{\mathrm{c}}(s_t)}{T}$
  if $\hat{r}_{\mathrm{c}}^i(\theta_i) > \epsilon$ **then**
    Store θ_i in Θ
  end if
end for
return Θ ▷ return set of broken trajectories

Implementation and Hyperparameters

CCPT is implemented in Tensorflow and all hyperparameters and their settings are:

- **Navigation module**: Learning rate $\beta_{\text{nm}} = 7e^{-5}$, discount $\gamma = 0.90$, and entropy coefficient $c_2 = 0.1$ in PPO.
- **Imitation module**: learning rate $\beta_{\text{im}} = 7e^{-5}$, replay buffer size $100,000$, batch size 32, and gradient penalty coefficient 5.0.
- **Curiosity module**: learning rate $\beta_{\text{cm}} = 7e^{-5}$ and batch size 128.

These settings were chosen after a set of preliminary experiments made with different configurations. All training was performed deploying ten agents in parallel on the same machine with an NVIDIA RTX 2080 SUPER GPU with 8GB RAM and a AMD Ryzen 7 3700X 8-Core CPU.

7.4 A Visual Analytics Interface to CCPT

In order to create a better connection between the CCPT algorithm and designers, we propose a visual analytics interface for CCPT. It is essential for game designers to understand the results of automated playtesting in order to take actionable decisions. Visualization can serve as an effective aid to this analysis. CCPT provides useful information, but in in the form of raw numbers such as the trajectories $\theta_i \in \Theta$ defined by low-level states s_i and values like r_{c}. These numbers are invaluable for analysis, but they are many and dense. The specific goal of the visual interface is to facilitate high-level visual analysis of the navigation behavior of agents and to assess whether there are bugs or overlooked issues in the game scene. In Figure 7.4 we give an overview of the visualization interface, that consists of three major components, each described below.

Exploration Overview

Shown in Figure 7.4(a), the Exploration Overview is the main visualization panel. It takes all the data gathered during training and provides an interactive view of CCPT results familiar to game designers. The main aim of this module is to show designers how agents have explored the environment.

To construct the Exploration Overview panel, we first collect the set of *all* trajectories gathered by CCPT:

$$\hat{\Theta} = \{\theta_i \mid i = 1, ..., M\}, \tag{7.7}$$

where M is the total number of episodes within the playtest session. Each state s_t^i of the trajectories $\theta_i \in \hat{\Theta}$ is composed of, among other elements, the (x, y, z) position of the agent. We use this information to compute and display a 3D heatmap of all positions visited by agents during training and exploration. This gives an idea of how agents explored the scene and which trajectories are the most followed – which should be the same as or similar to the expert demonstrations. Users can navigate the view in 3D and zoom in and out to highlight

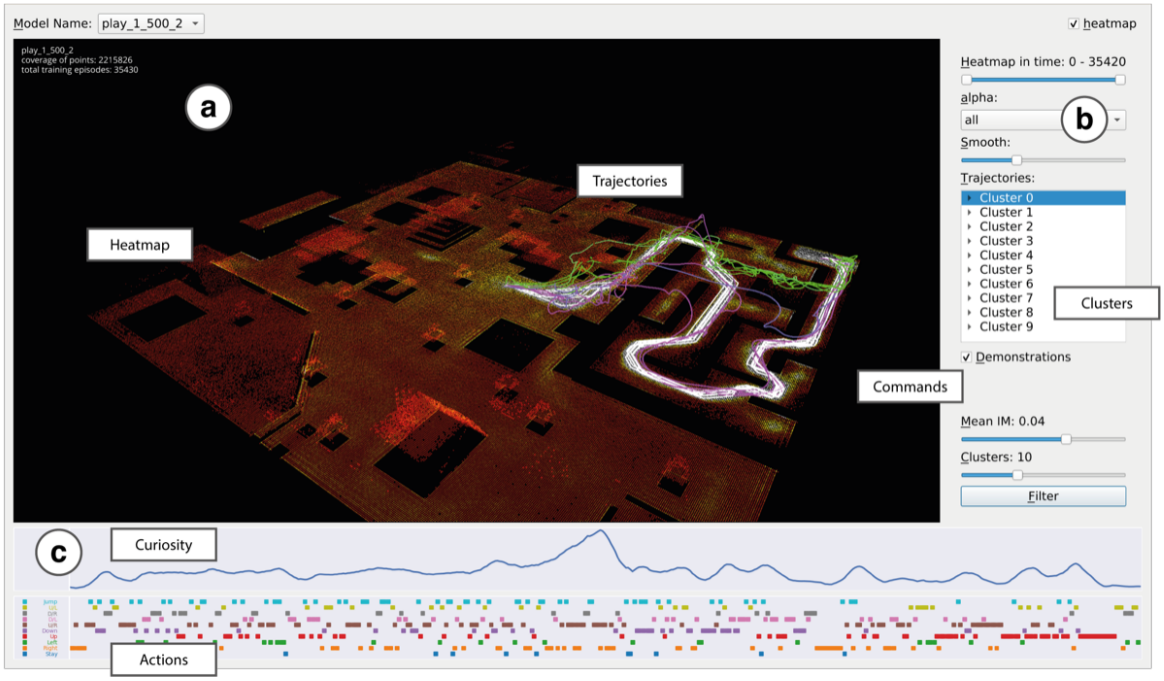


Figure 7.4: Overview of the visual gameplay analytics interface. The interface combines data and metrics coming from the CCPT algorithm and displays an overview summary of the results. The interface is intended to be an accessible and interactive tool to aid understanding of complex models even for non-experts in machine learning. More details about each element of the interface are given in Section 7.4.

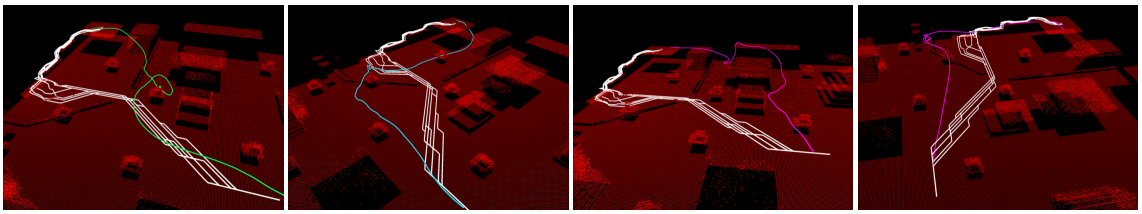
details. We emphasize that for the construction of the 3D heatmap we use *only the states collected by CCPT during training (Algorithm 1)* and that we do not use any additional information from the real game environment.

A filtered subset of suspicious trajectories highlighted by CCPT are overlaid on the Exploration Overview. The way all trajectories identified using Equation 7.6 are filtered is controlled by the user using the Command Interface (described below). The trajectories are drawn taking into account the (x, y, z) position of each $s_t^i \in \theta_i$. Finally, a brief description and statistics from the current experiment are displayed in the top-left corner of the Exploration Overview.

Command Interface

Illustrated in Figure 7.4(b), this module contains tools that enable users to modify the Exploration Overview. Starting from the top-right corner, we have:

- **Heatmap in time**: this slider lets users define a temporal window in which to navigate the heatmap. This aids in understanding how agent exploration evolves over time.



(a) (b) (c) (d)

Figure 7.5: Close-up images illustrating different bugs and gameplay issues found by CCPT. White trajectories indicate expert demonstrations, while the colored ones are those highlighted by CCPT as problematic. The bugs are: (a) missing collision box; (b) exploiting props; (c) using unintended path; and (d) exploiting glitches.

- **Selection of α**: this combo box lets designers filter the heatmap according to the α value. From Equation 7.4 we see that as $\alpha \to 0$ agents should mainly follow expert trajectories, and thus the reconstructed environment will visualize just trajectories in the proximity of the demonstrations. As $\alpha \to 1$, however, agents should have avoided the states that are part of the expert demonstrations, thus exploring a wider range of the scene. This selection is useful for debugging how an imitating agent behaves compared to an exploratory one.

- **Trajectory smoothing slider**: since we are using exploratory agents, the trajectories in Θ can be noisy and less informative in their raw state. Designers can compensate for this with this slider that smooths the overlaid trajectories to reduce noise. The slider defines the intensity of the smoothing. Smoothing is done using Savitzky-Golay filters to preserve salient features (Press and Teukolsky, 1990).

- **Trajectory cluster view**: this tree view contains the list of all the trajectories overlaid in the exploration overview. Since Θ contains many trajectories that are very similar to each other, visualizing all of them would result in a cluttered interface. We therefore cluster them with the K-Medoid algorithm (Zhang and Couloigner, 2005) with similarity determined using the 3D Fréchet distance:

$$D_{\text{Fréchet}}(A,B) = \min_{\mu} \max_{a \in A} d(a, \mu(a)), \tag{7.8}$$

  where A and B are two trajectories, $a = (x,y,z) \in A$ are points in the trajectory, $\mu : A \to B$ is a one-to-one continuous mapping from a point on trajectory A to a point on trajectory B, and $d(u,v)$ is the Euclidean distance between two points. The number of clusters can be manually selected by designers through the slider below the trajectory selection tree. After clustering the exploration overview is rendered using the trajectory in each cluster having the highest $\hat{r}_c(\theta_i)$ (Equation 7.5).

- **Cluster selection**: designers can isolate single trajectories θ_s by selecting them in the tree selection view, as well as view all the other paths that belong to the same cluster, by expanding the tree of θ_s. Designers can change the trajectory color by selecting from three modes: all trajectories in the same color, trajectories colored according to the Fréchet distance between them (Equation 7.8), or all trajectories colored using a gradient based on the normalized $r_c(s_t^i)$ values of each $s_t^i \in \theta_i$. For the latter, the higher the $r_c(s_t^i)$ is the more red the trajectories become. This mode has a particularly useful function: the points highlighted in red are the ones that were less visited by agents during training. They are the most difficult points to reach and very different from the expected behaviors. Thus, with high probability these points highlight a bug or overlooked issues designers seek. In Section 7.5, we show examples of this feature.

Trajectory Details

At the bottom of the interface are two plots displaying details of the selected trajectory θ_s (see Figure 7.4(c)). The first is the evolution of the curiosity value $r_c(s_t^s)$ of each state $s_t^s \in \theta_s$ over time. The second plot displays all actions a_t^s performed by the agent over all timesteps t of θ_s. This plot is useful for understanding how the agent found θ_s and its relative bugs and gives designers the ability to manually replicate them. If users hover the pointer over one of these plots, the corresponding point in the 3D trajectory is highlighted.

Implementation

The visual analytics interface is implemented in Python using VisPy, a high-performance interactive 3D data visualization library, and PyQt5, a python binding for the cross-platform GUI toolkit Qt. The implementation is open-source and available online.[‡] An important design feature is that it aims to be completely engine agnostic. Thanks to the Exploration Overview, designers can distinguish all the main 3D structures of the game scene without the need to visualize results directly in the real game environment. The information visualized depends only on the Θ data gathered during automated playtesting and does not require any additional game-specific data. We believe this to be a huge advantage for designers as it eliminates the burden of using the game engine every time they need to understand where a bug occurs. If needed, they can use the information from the action plot to replicate the problem directly into the game.

7.5 Experimental Results

In this section we detail experiments that showcase the capability of CCPT to perform automated playtesting. We are interested in four primary research questions:

[‡]Code available at https://github.com/SestoAle/CCPT.

	Coverage	**Bugs Found**	**Bugs Detected**
CCPT (ours)	1.96 ± 0.11	$\mathbf{12.80 \pm 1.60}$	$\mathbf{12.80 \pm 1.60}$
Linear Combination	1.24 ± 0.35	7.00 ± 0.00	6.60 ± 0.49
Peng et al. (2021)	0.84 ± 0.33	2.00 ± 0.63	0.00 ± 0.00
Gordillo et al. (2021)	$\mathbf{2.39 \pm 0.34}$	10.40 ± 0.48	4.20 ± 0.75

Table 7.1: CCPT results compared to the state-of-the-art: using a linear combination of curiosity and imitation; using the AMP approach Peng et al. (2021); and using Gordillo et al. (2021) approach. Coverage is expressed in millions of unique states explored during training. The Bugs Found column regards issues found by agents during training, but not necessarily identified as issues, while Bugs Detected are issues found and identified by Algorithm 2 as bugs by our model. The numbers represent the mean and standard deviation over 5 different runs. The bold ones describe the best results.

1. Can CCPT find and highlight bugs and oversights in a complex 3D environment?
2. How does the *exploration-conditioned intrinsic reward function* improve playtesting efficiency?
3. Do our models offer better performance than traditional models used for 3D navigation?
4. Does the visual analytics interface increase explainability and trust in the CCPT algorithm?

For all experiments in this section we used our navigation environment described in Section 7.3. All images in this section were captured in the CCPT visual analytics interface. To further investigate the robustness of CCPT, we report additional experiments using the ViZDoom environment, a real game environment different from our 3D navigation environment (Kempka et al., 2016).

Playtesting Performance

To evaluate the ability of CCPT to find and highlight bugs we tested four different goal areas in the environment. These areas represent four of the most difficult spots to reach, with trajectories that include dynamic elements, climbable surfaces and complex navigation challenges. For the goal areas we provide six expert demonstrations for each one showing the intended way to arrive at each specific goal. We then train ten agents in parallel and show the most relevant trajectories found by the algorithm for each area. In Figure 7.8 we give visualizations of the results of the four goal area experiments.

Table 7.1 summarizes the results found for all four areas. CCPT is able to find and highlight different ways of arriving to the same goal area of the expert demonstrations, however taking different paths and using different elements with respect to the intended

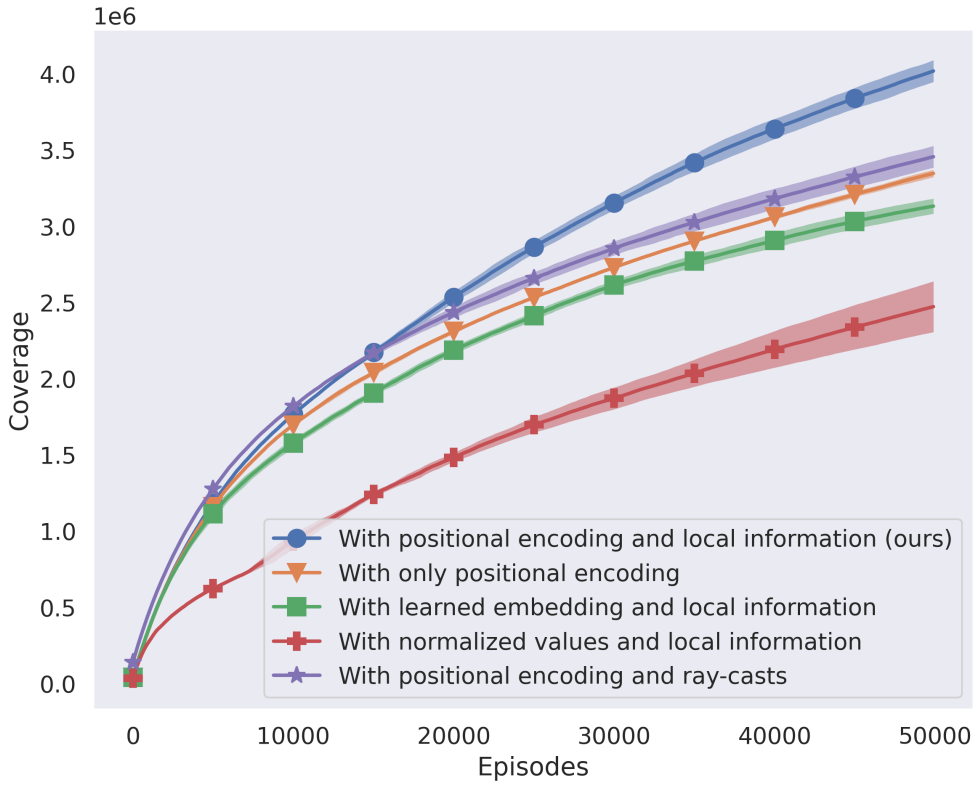
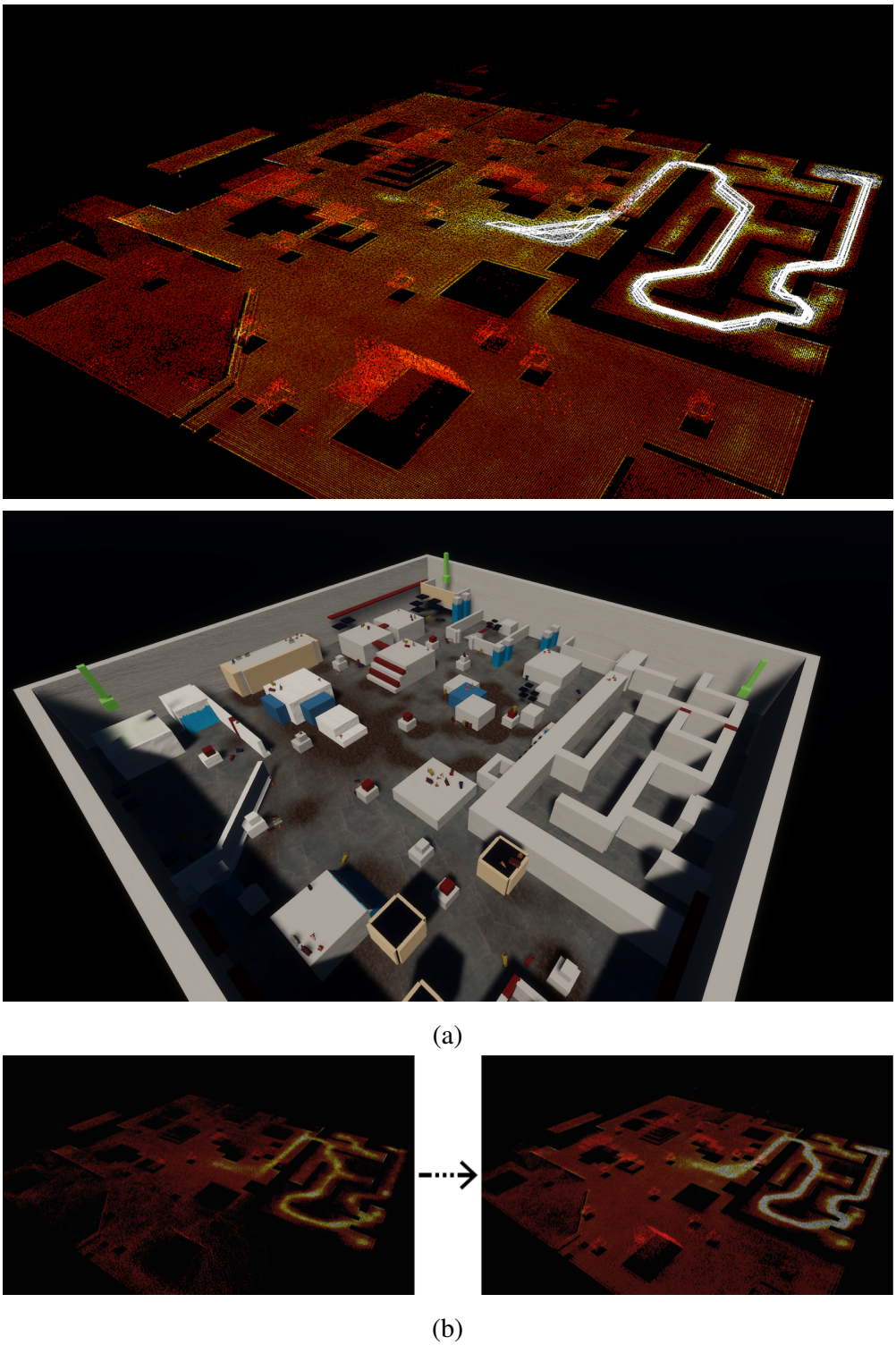


Figure 7.6: Results of ablation study. The lines represent the mean and and the error bars the standard deviation of the coverage of the different approaches over 5 different runs. Details in Sections 7.5.

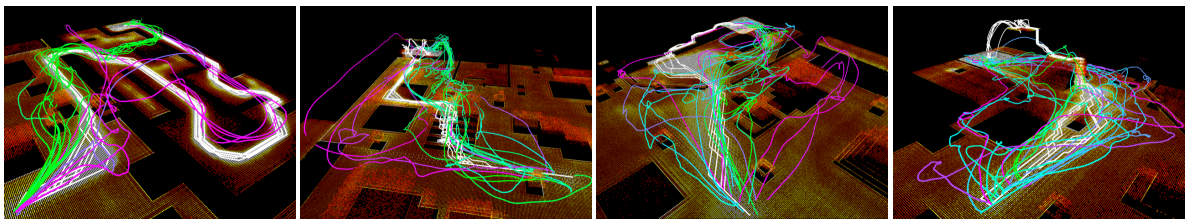
ones. We compare CCPT with three main algorithms from the state-of-the-art: the one from Gordillo et al. (2021) that uses an approach based solely on curiosity; the AMP algorithm (Peng et al., 2021), which only uses imitation learning with the same set of expert demonstrations used for CCPT; and a linear combination of curiosity and imitation learning, without the use of our *exploration-conditioned intrinsic reward function*. We compare three different metrics: "*coverage*" is expressed in millions of unique states explored during training. It is computed by counting the discretized positions gathered by the agent during training; "*bugs found*" regards issues found during training, but not necessarily *identified* as issues; and "*bugs detected*" regards issues found and identified as bugs. The first metric measures the exploratory abilities of the agents, the second one describes the performance of the approaches, while the latter measures the ability to highlight bugs directly to developers with Algorithm 2. We argue that the third metric is the most important one for game designers. Developers usually are not experts in machine learning and thus require methods that automatically identify where and what bugs are. This avoids looking for issues in a huge amount of poorly explainable results coming from mere exploration. Compared to the baselines, even if the approach from Gordillo et al. (2021) seems to perform slightly better in terms of environmental coverage, CCPT clearly outperforms other methods not only in



(a)

(b)

Figure 7.7: Exploration overview. (a) a 3D heatmap created by analyzing the (x, y, z) positions gathered during training compared to the real environment. (b) the evolution of the heatmap over time: to the left is the heatmap after few training episodes, and to the right the heatmap at the end of playtesting.



(a) Area 1 (b) Area 2 (c) Area 3 (d) Area 4

Figure 7.8: Qualitative visualizations on four different tested areas. The white trajectories indicate the expert demonstrations, while the colored ones are those highlighted by CCPT.

finding bugs we manually inserted in the environment, but also in highlighting them directly for designers. We reiterate that the focus of CCPT is finding and detecting issues, even to the detriment of environment coverage.

Figure 7.5 shows close-up examples of some CCPT results. Instead of relying only on expert demonstrations, the agent took many different paths:

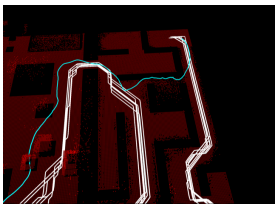
- in Figure 7.5(a) the agent uses a missing collision box in a tiny portion of the wall, thus arriving into the goal area by exploiting a slight slope of the wall of a pillar;
- in Figures 7.5(b) and 7.5(c) the agent exploits two oversights that allows it to jump over the wall and skip the main entrance. The first one uses an unintended prop near the wall while the second one exploits a climbable surface and precise double jumps; and
- in Figure 7.5(d) the agent exploits a glitch that allows it to perform infinite jumps and to skip the wall like in the previous examples. This last issue is quite interesting as the agent has actually learned to exploit the glitch rather than using it at random.

Such examples can be also found in all of the four goal areas tested in this chapter, highlighting the good performance of our algorithm. Figure 7.9 shows other close-up examples of bugs found and highlighted by the algorithm.

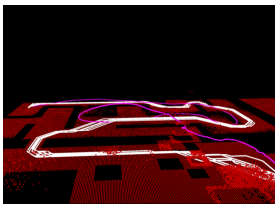
Evaluating the Reward Function

To evaluate the performance of our *exploration-conditioned intrinsic reward function* we performed an in-depth study using different fixed values of α in Equation 7.4: $\alpha = 0.5$ resulting in an average of r_c and r_i, $\alpha = 0.0$ corresponding to only using imitation, and $\alpha = 1.0$ corresponding to only using curiosity.

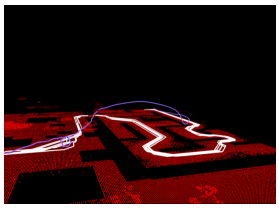
Table 7.1 shows the number of points in space covered for the four methods. As expected, agents trained with only imitation learning cover the smallest part of the environment, while those trained with only curiosity cover the most points. However, note that agents trained



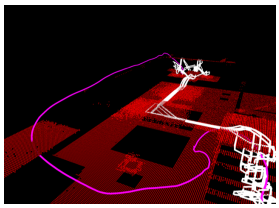
(a) Missing collision box



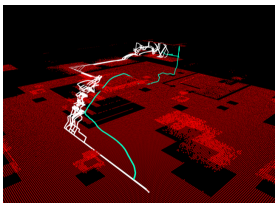
(b) Skip main entrance



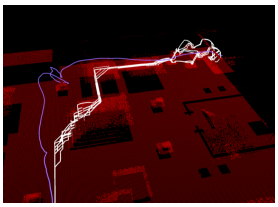
(c) Skip labyrinth



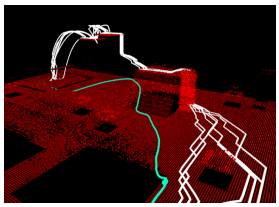
(d) Using unintended path



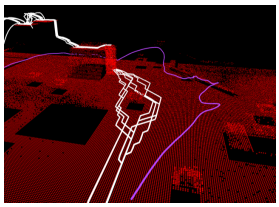
(e) Using glitch



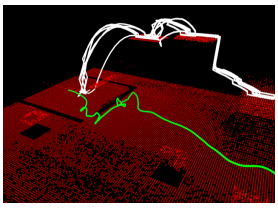
(f) Exploiting props



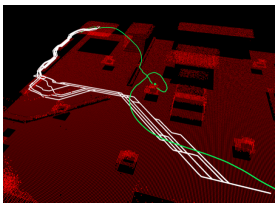
(g) Exploiting slope



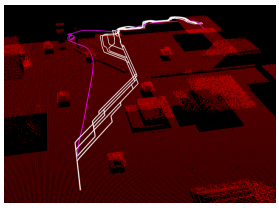
(h) Using glitch



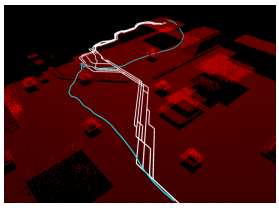
(i) Missing collision box



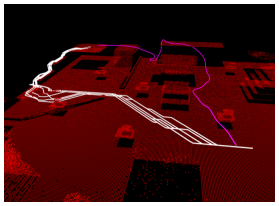
(l) Missing collision box



(m) Using glitch

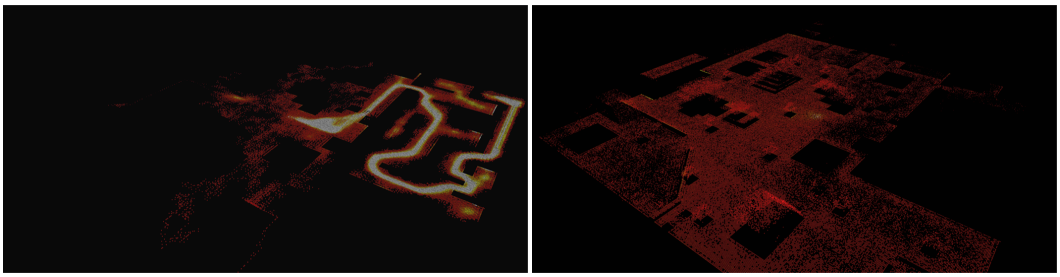


(n) Exploiting props



(o) Using unintended path

Figure 7.9: Close-up images of the four area experiments. Each image represents a different bug or issue found by CCPT. The white trajectories indicate expert demonstrations, while the colored ones are those highlighted by CCPT.



(a) $\alpha \to 0$ (b) $\alpha \to 1$

Figure 7.10: Comparison of heatmaps created by (a) agents imitating experts and by (b) exploratory agents.

Table 7.2: Results of the Likert questionnaire. Designers answered with a value between 1 (strongly disagree) and 7 (strongly agree).

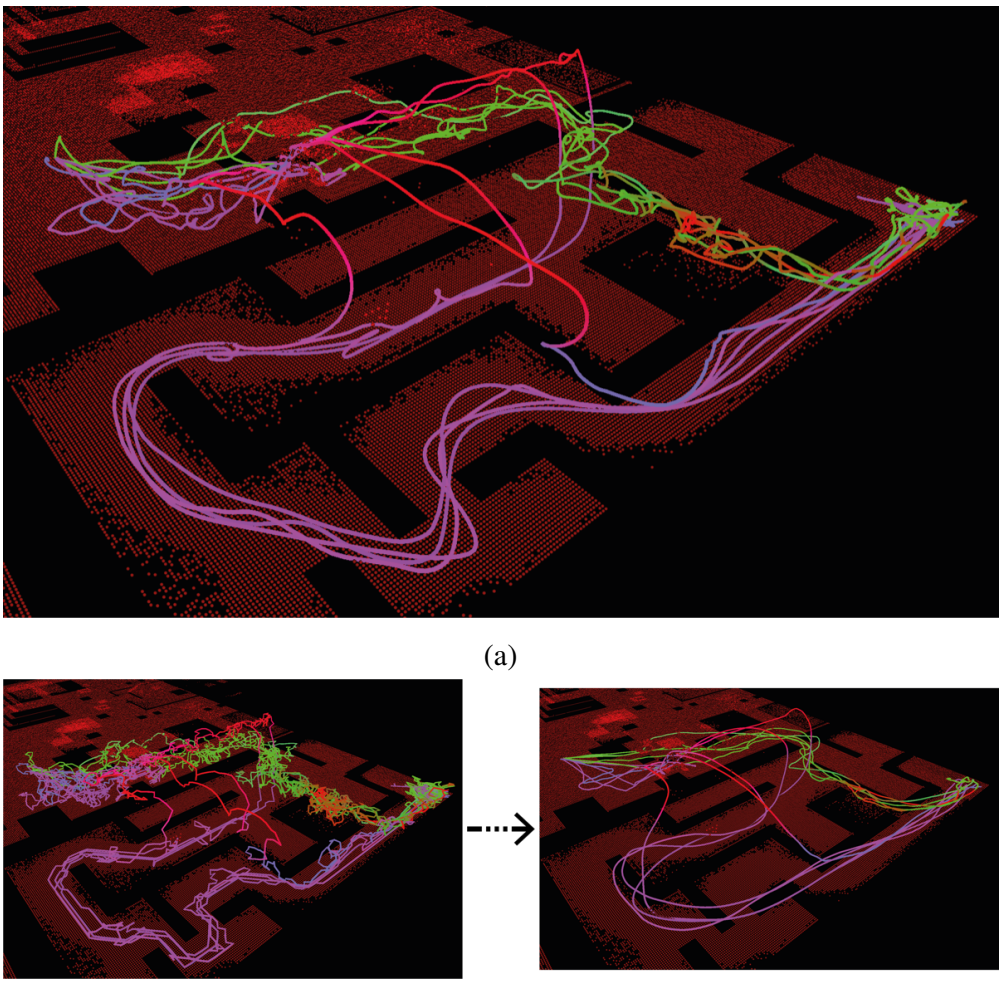
N°	Question	μ	Mo	Md	σ
1	Do you think the automated playtesting based on ML algorithms can be useful for game development?	6.4	7	7	0.89
2	Do you think the visual analytic tool is a useful aid for understanding the results of the algorithm?	6.2	6	6	0.83
3	The interface is easy to use.	5.6	6	6	0.89
4	The visual interface will aid your decision-making process after a playtesting session.	6.0	6	6	0.70
5	I understand how the ML algorithm for automated playtesting works (the theory behind it).	4.8	6	4	2.68
6	I understand how the visual interface works, its usefulness and what it shows.	6.2	6	7	0.83
7	I can make a comparison between the reconstructed environment and the real game scene.	6.2	7	7	1.30
8	Given a bugged trajectory, I can understand where the tool says there might be a bug.	6.6	7	7	0.54
9	It seems I can easily follow the trajectory within the level (without smoothing).	4.6	-	5	2.07
10	It seems I can easily follow the trajectory within the level (with smoothing).	6.0	7	6	1.00
11	The interface seems to give me enough information to recognize the bug.	5.8	6	6	0.83
12	The tool seems to give me enough information to manually replicate the bug.	6.2	6	6	0.83
13	I think a tool like this can increase confidence in ML based systems for video game development.	6.2	6	6	0.83

with our algorithm achieve much better exploration compared to imitation learning alone and an average of imitation and curiosity. This is an interesting finding: while they cover a very large portion of the map, agents trained with our reward are still able to follow the expert demonstrations. This enables us to filter and highlight broken trajectories as described in Section 7.3.

Ablation Study

We performed a series of ablation tests to better understand the performance of our models. In particular, we claim that the combination of the semantic occupancy map and the positional embeddings described in Section 7.3 is an efficient way to encode environment information when compared to standard navigation and exploration architectures.

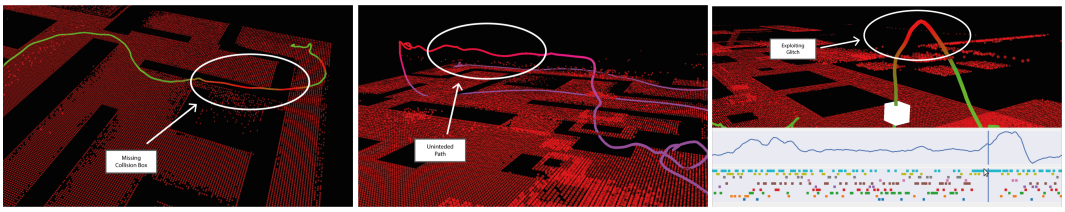
In Figure 7.6 we plot map coverage as a function of training steps for different con-



(a)

(b)

Figure 7.11: Trajectory overview. (a) shows the set of suspicious trajectories represented by the centroids of the clusters in Θ. The red parts of each trajectory represent the inferred location of the bug as indicated by the r_c metric. (b) gives a comparison between noisy and smoothed trajectories.



(a) (b) (c)

Figure 7.12: Close-up images illustrating bugs found by CCPT and highlighted by the interface. In (a) the red part of the broken trajectory highlights the presence of a missing collision box. In (b), the method provides insight into when the agent deviates from the expert trajectories and takes an unintended path using a prop (an exploitable pillar). In (c), an example action plot. Analyzing the action plot in the positions where the interface indicates a bug, developers can understand how the agent exploits that bug using the sequence of actions provided. The light blue squares represent *JUMP* actions by the agent.

figurations of the policy network architecture. From the plot it is clear that the positional embeddings used to encode global positions provide a significant boost to performance compared to using only normalized values as in (Gordillo et al., 2021; Devlin et al., 2021; Alonso et al., 2021) or even to learned embeddings. To the best of our knowledge, this is the first example of using such positional embeddings in deep reinforcement learning. Furthermore, the plot shows that the 3D semantic occupancy map and the relative 3D convolutional network improve the results compared to using only global information and to a ray-casting baseline similar to Gordillo et al. (2021). The ray-casting approach uses 24 rays cast in various directions and at various heights. Each ray provides two values: the collision distance and the semantic value of the collided object. From the plots it is clear that the combination of positional embeddings and semantic occupancy map that defines our full model clearly outperforms all other ablations.

Visualization Experiments

To evaluate the features of the visual analytics interface described in Section 7.4, we present a series of examples for explaining and interpreting the results experiments in Section 7.5.

Environment reconstruction. In Figure 7.7(a) we see a heatmap generated from an experiment in the Navigation Environment along with a screenshot of the actual game scene. Note how the reconstructed environment reflects the original structures and how most 3D elements are distinguishable. These 3D structures are all fairly distinguishable without the need to be traced back to the game engine. This allows the approach to be completely engine-agnostic and game-agnostic. Since it is based only on the positions gathered by the agent during training, the tool can reconstruct any type of environment the user wants to test. The heatmap clearly shows how the agent mainly follows the expert trajectories highlighted

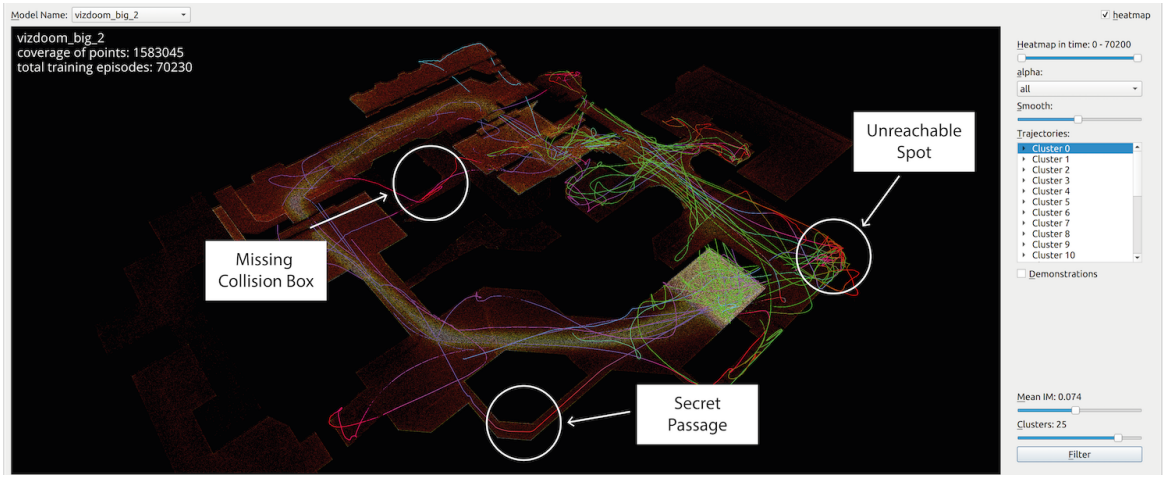


Figure 7.13: Visualization experiment on a ViZDoom level. Indicated are three examples of suspicious trajectories highlighted by the visual analytics interface.

in white. In Figure 7.7(b) we see how the heatmap changes over time, giving insight in how the agent explores the environment during training.

Finally, in Figure 7.10 we see the environment reconstructed by different agents. Agents with $\alpha = 0$, which are purely imitating agents, reconstruct only a small portion of the environment in the proximity of expert demonstration trajectories. On the other hand, agents with $\alpha = 1$ are purely exploratory and reconstruct a large portion of the map, as they are trained to find different paths arriving at the goal location.

Visualizing trajectories. In Figure 7.11(a) we see an example of trajectories overlaid on the environment reconstruction. As described in Section 7.4, only the centroids of the clusters found using K-Medoids are shown. Clustering works well, as each of the trajectories delineate different paths to the goal. We show the smoothing process in Figure 7.11(b). Smoothing removes noise due to the exploratory nature of the agents while maintaining the general shape and important details of the path.

Identifying bugs. We provide some close-up images of bugs highlighted by the visual interface:

- In Figure 7.12(a) we see how the agent exploits a missing collision box to skip parts of the intended path. Using the $r_c(s_t^i)$ metric, the interface directly highlights in which portion of the 3D path the bug likely occurs, as these values indicate trajectory states that are very different from the expert ones. In this case, the red colored part shows exactly where the missing collision box is.

- Figure 7.12(b) illustrates how the agent can use props to take an unintended path to the goal area, diverging from the expected behavior. In this case, the red colored

part exactly points out the elements in the trajectory used by the agent follow this unintended path, giving developers insight in what they should change.

- Figure 7.12(c) we see how the agent uses a glitch to arrive to the goal area. This seemingly meaningless behavior is explained by the plots visualized at the bottom. If we look at the values in the positions highlighted in red by hovering pointer over the highest curiosity values, we see how the agent spammed many *jump* actions that, in this particular spot, allow it to fly over to the goal area. In this case, developers can not only see where the bug is, but also understand how the agent exploits it.

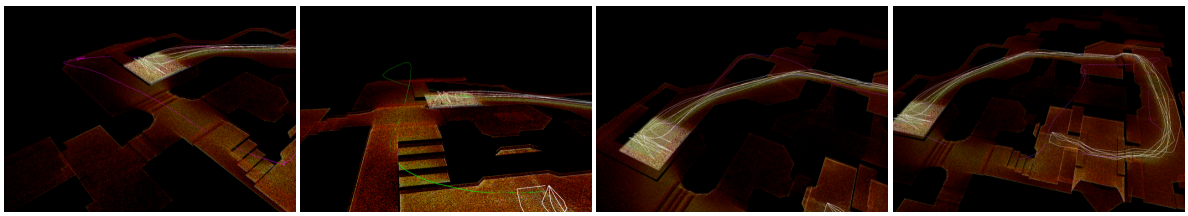
User Study

To gain insight into the effectiveness of such visual analytics interface, we conducted a preliminary user study with 5 experienced, professional level designers. The study consisted of a 20 minute video explanation of the tool showing all features using the two experiments of Section 7.5. After viewing the video, participants were asked to answer a Likert questionnaire, whose results are summarized in Table 7.2. The study aimed to determine the usefulness of the approach to real level designers in terms of increasing their understanding of the results of algorithms like CCPT.

The results of questions 1 and 2 suggest that most participants think this tool *"could help relieve stress put on testing teams to identify problem areas in levels"* and that *"these tools that provide a higher quality experience for players are a great step forward for development"*. Although most testers are not experts in machine learning as the σ of question 5 indicates, the responses to questions 2, 6 and 13 demonstrate that our proposed visual analytics interface substantially increases confidence in the results. Moreover, designers *"won't use machine learning algorithms for their game unless they have this type of tool"*. However, some users claimed that "*the tool looks good and with a bit of training it can be useful*", indicating that it is not completely understandable at first sight.

Questions 7 through 12 show that all the features included in the tool help designers gain insight into the bugs the tool highlights. Most problems are related to the visualization of the trajectories that *"are extremely difficult to follow without isolating down single trajectories"*. Although the smoothing *"definitely helps make the trajectory more readable"*, designers said that *"the issues of size and contrast still make it difficult to follow"*. Instead, features like highlighting the bug and showing the action sequence were much appreciated as participants *"really like the breakdown of the bots actions"* and *"definitely have all the information to know that a bug is occurring"*.

In conclusion, this preliminary user test shows that experienced designers see the potential of both the algorithm and the visual interface and that using the algorithm alone would not be as useful without clearly explainable results provided by a tool like the one we propose. All users also provided suggestions of how to improve the project. For instance, they *"would love a video component to help pinpoint exactly what and where it took place"* and *"what*



(a) (b) (c) (d)

Figure 7.14: Close-up images from the ViZDoom experiment. Each image represents a different bug or issue found by CCPT. The white trajectories describe the expert demonstrations, while the colored ones are those highlighted by CCPT.

about more complex environments? Vertical, rolling terrain, or areas that layer on top of each other?".

ViZDoom Experiments

In this section we investigate the robustness of CCPT applied in a real game different to our 3D navigation environment. To verify that CCPT can be a useful tool for game development, we applied it to the ViZDoom environment, which is a semi-realistic 3D world based on the classic first-person shooter video game Doom (Kempka et al., 2016). The model architectures for all the modules are the same as in Section 7.3, but with the 3D semantic occupancy map and the relative 3D convolutional network replaced with the renderer depth buffer and a 2D convolutional network. We replaced the 3D occupancy map state space because we were unable to access the VizDoom game code and implement our own version of the 3D occupancy map. The depth buffer samples have a single channeled size of 32×32 and the network is composed of two 2D convolutional layers with 32 and 64 channels. Each layer has kernel sizes of 3×3, strides of 2 and ReLU activations. The action space consists of 6 discrete actions: move forward, move left, move right, turn left, turn right and jump.

We tested our algorithm in the level shown in Figure 7.13(a). The scene is approximately $2800\,\mathrm{m} \times 2800\,\mathrm{m} \times 100\,\mathrm{m}$ in size. Since we are interested in coverage testing, we removed all enemies and props.

For the experiment, we recorded expert demonstrations showing the intended path for arriving at a specific goal area. We then let CCPT run and in the end we visualize the most relevant trajectories highlighted by the algorithm. Figure 7.13(b) shows some qualitative results in which we can see how the algorithm is perfectly able to find and highlight different ways of arriving to the same goal area with respect to expert trajectories shown in white. In Figure 7.14 we give some close-up examples of specific findings:

- Figure 7.14(a) shows how the agent can arrive at the goal area using a completely different path than the intended one;

- Figure 7.14(b) shows a trajectory similar to the one before, but using an elevated spot that was not meant to be reachable;
- Figure 7.14(c) shows a slight variation of expert demonstrations using a hidden path; and
- Figure 7.14(d) shows how the agent can use a complex path to skip part of the level and arrive in the goal location using a tiny gap between two walls.

Surprisingly, the CCPT algorithm found many different broken trajectories, even in a scene from a real game like this one. Some of these are just different ways to reach the goal location. However, many of them highlight interesting paths like the ones shown in the figure. Without our interface, it would be difficult to understand *which* the issues are, *where* they are, and *how* agents found them. The amount of data collected in Θ for this experiment was massive, and we argue that the use of such visual analytics interface is essential even for experts in CCPT.

7.6 Conclusions

In this chapter we introduced a novel reinforcement learning approach to automatically playtest complex 3D scenarios. Curiosity-Conditioned Proximal Trajectories (CCPT) enables developers and designers to specify an area to test in the form of expert demonstrations. CCPT uses a combination of imitation learning and curiosity, guided by what we call an *exploration-conditioned intrinsic reward function*, for performing exploration in the proximity of demonstrated trajectories. Our approach is not only able to find glitches and oversights, but can also automatically identify and highlight trajectories containing potential issues. Our results show a high level of coverage and bug discovery in our proposed navigation environment, highlighting how well the particular combination of curiosity and imitation works for this purpose.

Our experiments also show how our proposed visual analytics interface allows designers to not only easily understand *which* trajectories contain bugs, but also *where*, *how*, and *why* these bugs were flagged by CCPT. Deep analysis of the results of automated playtesting is fundamental to designers in order to allow them to make actionable design decisions. The interface directly supports this by connecting the results of complex deep reinforcement learning methods to an interpretable and interactive interface. We believe that algorithms and tools like these will be useful means for AAA game designers to automatically identify issues or potential exploits with less reliance on human testers, ultimately increasing confidence in the use of machine learning techniques in video game production.

In the next chapter we further develop these ideas, asking developers and designers what characteristics they would like to see in a DRL-based tool for video game development, and in particular for the use-case of gameplay validation. This analysis will help us to move from theoretical methods to an effectively useful and, above all, usable tool.

Chapter 8

Imitation Learning as Designer Assistance Tool[†]

In Chapter 7 we described a new algorithm and a new visualization tool that aid automatic gameplay validation. Our qualitative analyses with professional designers demonstrated that algorithms like these will be useful tools for AAA game development. However, most of the developers doubted that these techniques are readily applicable in a real development process due to many factors: the time required to for training, the expertise needed to setup the environment and hyper-parameters, and the poor re-usability of trained agents. That is, we now need to move from looking at the desiderata described in Chapter 3 to meeting the real requirements that could translate these approaches into effective game design tools.

For this reason, in this this chapter we still concentrate on playtesting applications, proposing a new approach to automated game validation and testing, with a clear focus on use-cases. With the input of professional game developers, we argue that reinforcement learning may not be enough to create usable tools. Our method leverages a data-driven imitation learning technique which requires little effort and time, and no knowledge of machine learning or programming that designers can use to efficiently train game testing agents. We investigate the validity of our approach through a user study with industry experts. The survey results show that our method is indeed a promising approach to game validation and that data-driven programming would be a useful aid to reduce time and increase quality of modern playtesting.

Developers also highlighted several open challenges that, in their opinion, still exist in using a machine learning-based tool that goes beyond the desiderata defined in Chapter 3. With the help of the most recent literature, in this chapter we analyze the identified challenges and propose future research directions suitable for supporting and maximizing the utility of the dissertation beyond the specific techniques it proposes.

[†]Portions of this chapter appeared in: A. Sestini, L. Gisslén, J. Bergdahl, K. Tollmar, and A. D. Bagdanov, "Towards Informed Design and Validation Assistance in Computer Games Using Imitation Learning", published in the *Human-in-the-Loop Learning (HiLL) Workshop at Neural Information Processing Systems (NeurIPS)*, 2022.

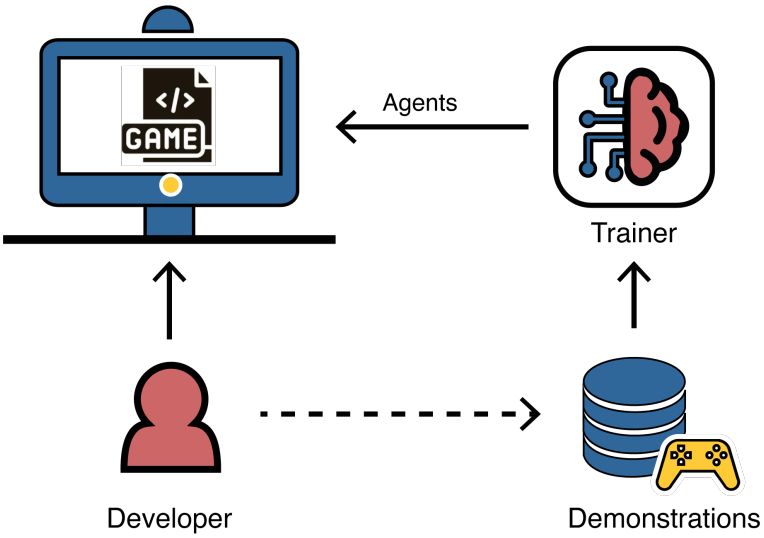


Figure 8.1: Our approach lets designers perform automated game validation directly during the design phase. The training is interactive as the algorithm allows designers to smoothly switch between designing the level, providing gameplay demonstrations, and getting feedback from the trained testing agents.

8.1 Introduction

As presented in Chapter 7, automated playtesting and validation techniques have been proposed to mitigate the need for manual validation in large games. This could be done either by crafting model-based bots and using them to perform automated playtesting (Stahlke et al., 2020), or with reinforcement learning as shown in Chapter 7. However, the study we report on in the previous chapter highlighted a few drawbacks. Model-based agents, typically bots with hand-crafted behaviors (i.e. achieved by scripting), require a certain level of domain knowledge and programming skills that game designers do not necessarily have. Moreover, the lack of generalization in this method might render the agents unusable if changes are made in the game environment.

Reinforcement learning agents, on the other hand, can not only learn to play the game without scripting but can also be easily retrained when the environment changes. However, reinforcement learning is sample inefficient and arguably requires a high level of expertise in machine learning to be effectively used (e.g. for properly crafting a well performing reward function). Reinforcement learning in general provides a low level of controllability as the agents will try to exploit the environment regardless of the intentions of the designer (Open AI, 2016; Jacob et al., 2020). Additionally, making a game compliant with a reinforcement learning training setup is a considerable engineering effort potentially requiring intrusive changes to the game's source code.

In this chapter we propose a data-driven approach for creating agent behaviors for

automated game validation (illustrated in Figure 8.1). Unlike methods where agents are manually programmed or scripted, our method uses imitation learning, in particular the DAgger algorithm (Ross et al., 2011), to clone the behavior of the user. With this, we can leverage developer expertise while at the same time not requiring additional knowledge of programming or machine learning. The feedback loop for the designers can therefore be rapid with a more efficient creation process than having to wait days or sometimes weeks for manual playtests. This study is made to either validate or refute above claims. The research questions we are focusing on are: Can imitation learning be used as an effective game design tool? What improvements and research are needed to maximize its value as a tool? To explore this, we showcase the possible use cases for which level designers could use our approach. Further, we report on results from survey that explores how our method satisfies the requirements previously mentioned and gauges the interest in such a tool by professional developers. Based on the answers from the survey, we propose future research directions for improving game validation with imitation learning.

8.2 Related Work

Here, we review work from the literature most related to this chapter.

Automated Playtesting

For an extensive description of the current literature and studies regarding automated playtesting, we refer the reader to Section 7.2. As stated in Chapter 7, there are two main ways to perform automated gameplay testing. The first one is through classical, hand-scripted AI, of which which works by Stahlke et al. (2020), Holmgård et al. (2018), Chang et al. (2019), Mugrai et al. (2019), Stahlke et al. (2020), and Xiao et al. (2005) are notable examples. In parallel, several other works have used reinforcement learning to perform automated playtesting. In Chapter 7 we described a novel DRL-based algorithm to perform automatic validation and automatic search of gameplay bugs, and other notable examples are Politowski et al. (2022), Zheng et al. (2019), Agarwal et al. (2020), Bergdahl et al. (2020), and Gordillo et al. (2021).

Imitation Learning

Few approaches have applied imitation learning to game testing agents. The method of by Chang et al. (2019) uses demonstrations to guide exploration, although it does not use a learning algorithm. Zhao et al. (2020) used an approach similar to behavioral cloning in which gameplay agents learn behavioral policies from the game designers. Harmer et al. (2018) trained agents through a combination of imitation learning and reinforcement learning with multi-action policies for a first person shooter game. Tucker et al. (2018) used an inverse

reinforcement learning technique for training agents that can play a variety of games in the Atari 2600 game suite of the Arcade Learning Environment (ALE) (Bellemare et al., 2013).

8.3 Proposed Method

We propose an approach based on imitation learning to showcase the potential of data-driven methods for direct gameplay validation to designers. In this section, we first describe why we decide to use IL, then we detail our IL setup and training algorithm.

8.4 Method Comparison

In this section we better delineate the different common approaches to automated testing and justify why we decide to use imitation learning.

Reinforcement Learning. Reinforcement Learning (RL) for game validation is known to be sample inefficient. For example, the work proposed by Gordillo et al. (2021) required an average of two days of training before thorough coverage of a game scene is achieved. Even if sample inefficiency can be mitigated by combining RL with IL similar to what we did in Chapter 7, it is still challenging to practically apply RL in production. Moreover, RL typically grants little control to the user over the final behavior of the policy as the trained agents will inherently exploit the reward function regardless designer intention (Open AI, 2016). We can alleviate this problem with reward shaping, but this requires considerable knowledge of RL and machine learning in general, making it impractical for the general user. Even given these problems, RL still has some advantages. If designers prefer exploitation and exploration over imitation, then RL is preferable; and similar to IL, it does not require any programming knowledge to train.

Scripted Bots. By scripted bots we mean bots modelled with programming methods, i.e. the standard way of creating game AI bots. To program autonomous bots, designers must have relatively high domain knowledge of the level design, but also scripting skills to successfully craft the behaviors of the bots. Moreover, even if they have very fine control over the final behavior of the bots, the scripted agents will quickly become sub-optimal, and even unusable, when faced with design changes in the environment. The setup time also greatly depends on the complexity of the game and game scene. For the reasons listed above, we argue that this approach does not satisfy the requirements listed in Section 8.1.

Imitation learning (IL). IL is more sample-efficient than RL and allows training of agents in minutes or only few hours. With this method we only need to leverage the designer's domain knowledge of the game, thus effectively adding humans-in-the-loop for superior controllability. Not only can we demonstrate the intended behavior, but we can even correct it using new demonstrations with little additional training time. Compared to scripted behavior, IL provides the same level of generalization as RL as the policy model should be able to

generalize to some extent to unseen observations. However, compared to both RL and model-based bots, IL calls for no or limited knowledge of theory or programming skills as designers only have to demonstrate what they want the IL agent to do. With IL we can mitigate some of the drawbacks of RL-trained and scripted agents.

Approach

Here we describe our approach to exploring the potential of data-driven methods for direct gameplay validation to designers.

Algorithm. We use an IL approach based on the DAgger algorithm (Ross et al., 2011), as shown in Figure 8.1. DAgger allows designers to train agents interactively by actually playing the game. The core of the algorithm is to let designers seamlessly move between designing the level, providing game demonstrations and getting feedback from trained testing agents. Our approach requires little training time and is more sample efficient than the baseline methods, as we see in Section 8.5. Designers can also provide corrections to faulty agent behaviors, resulting in a continuous and rapid feedback loop between designers and agents. Providing corrections is as easy as taking the controller back and playing the game one more time. Once they are satisfied with agent behavior, designers can stop providing demonstrations and watch the agent validate the game. Developers can then make any kinds of design changes and wait for agent feedback without recording any new demonstrations.

State Space. For the approach to be effective, it must be as general as possible in order to adapt to the many different game genres and scenarios that designers may construct. Since 3D movement often is a crucial gameplay element of modern video games, we focus on finding the best state representation taking this into consideration. This approach, with minor observation tweaking, will transfer well to similar, but contextually different game modes. For setting up a behavior, developers define a goal position in the game environment. The spatial information of the agent relative to the goal is composed of the $\mathbb{R}^2$ projections of the agent-to-goal vector onto the XY and XZ planes as well as their corresponding lengths normalized to the available gameplay area. We also include information about the agent indicating whether it can jump, whether it is grounded or climbing, as well as any other auxiliary data expected to be relevant to gameplay. The user also specifies a list of entities and game objects that the agent should be aware of, e.g. intermediate goals, dynamic objects, enemies, and other assets that could be useful for achieving the final goal. From these entities, the same relative information is inferred as for the main goal position relative to the agent. Lastly, the agent also has local perception. A semantic map is used similar to the one used by th CCPT algorithm described in CHapter 7 which is general and performant. We illustrate an example of such a semantic 3D occupancy map used as input to the networks in Figure 8.4. This map is a categorical discretization of the space and elements around the agent, and each voxel in the map carries a semantic integer value describing the type of object at the corresponding game world position. The settings of the semantic map can be configured by

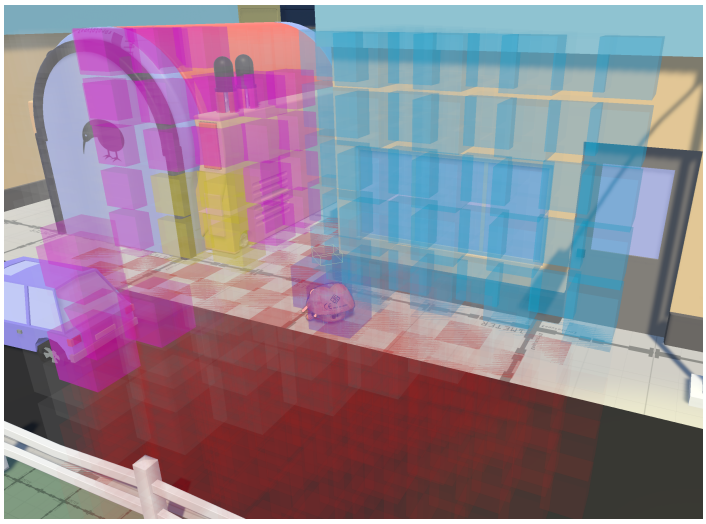


Figure 8.2: Example of a semantic map used by the agent. Each cube describes the semantic integer value of the type of object at the corresponding position relative to the agent highlighted in blue.

the designers.

Neural Network. We build the neural network π_θ with parameters θ with the goal of being as general and reusable as possible. The way we define the structure of the network is to fundamentally allow a higher level of usability by game designers. First, all the information about the agent and goal is passed into a linear layer producing the self-embedding $x_\mathrm{a} \in \mathbb{R}^d$, where d is the embedding size. The list of entities is passed through a separate linear layer with shared weights producing embeddings $x_{\mathrm{e}_i} \in \mathbb{R}^d$, one for each entity e_i in the list. Each of these embedding vectors is concatenated with the self-embedding, producing $x_{\mathrm{ae}_i} = [x_\mathrm{a}, x_{\mathrm{e}_i}]$, with $x_{\mathrm{ae}_i} \in \mathbb{R}^{2d}$. This list of vectors is then passed through a transformer encoder with 4 heads and final average pooling, producing a single vector $x_\mathrm{t} \in \mathbb{R}^{2d}$. In parallel, the semantic occupancy map $M \in \mathbb{R}^{s \times s \times s}$ is first fed into an embedding layer, transforming categorical representations into continuous ones, and then into a 3D convolutional network. The output of this convolutional network is a vector embedding $x_\mathrm{M} \in \mathbb{R}^d$ that is finally concatenated with x_t and passed through a feed forward network, producing an action probability distribution. The complete neural network architecture is shown in Figure 8.4(c).

Given a demonstration dataset $\mathcal{E} = \{\tau_i \mid \tau_i = (s^i_0, a^i_0, ..., s^i_{T_i}, a^i_{T_i}),\ i = 1, .., N\}$ of N trajectories τ_i, each composed of a sequence of state-action pairs (s^i_k, a^i_k), we update the network following the objective:

$$\arg\max_\theta \mathbb{E}_{(s,a)\sim\mathcal{E}}[\log \pi_\theta(a|s)]. \tag{8.1}$$

This objective aims to mimic the expert behavior which is represented by the dataset $\mathcal{E}$.

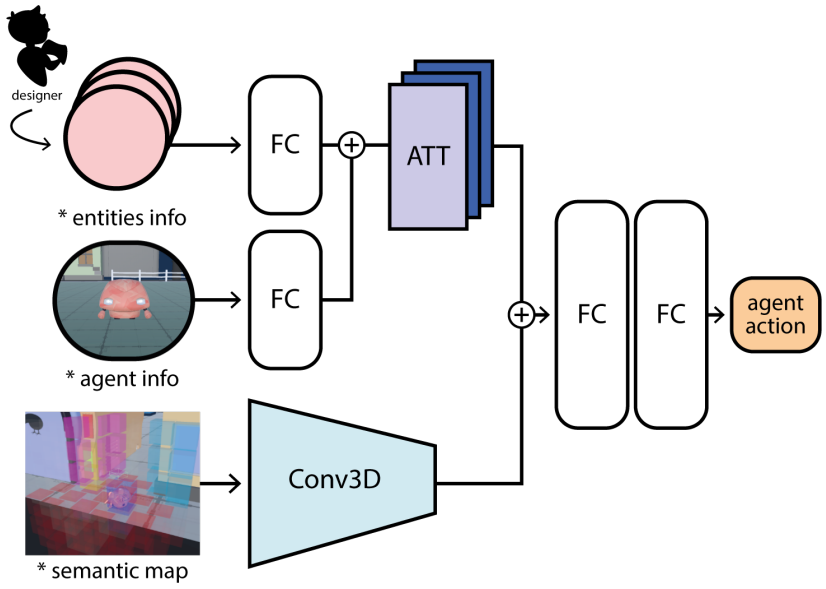


Figure 8.3: Overview of the neural network architecture used in this chapter.

8.5 Experimental Results

In order to evaluate the performance of our approach, we define a set of use cases that resemble typical situations that game designers face in their daily workflow. In Table 8.1, we report quantitative results for each use case, comparing our approach to the RL baselines described below.

Experimental Setup

Here we describe the environment and training setup used in this chapter to test our proposed approach.

Environment. The environment used in chapter is shown in Figure 8.4. It is a 3D navigation environment with procedurally generated elements. The environment is purposefully built like a mutable sandbox which can offer scenarios where a designer's level design workflow can be simulated. Users can add and change the goal location, agent spawn positions, layout of the level, location of intermediate goals and the locations of dynamic elements in the map. In this environment, the agent has a set of 7 discrete actions: move forward, move backward, turn right, turn left, jump, shoot, and do nothing. In addition, the agent can use some interactable objects located around the map.

Training Setup. We compare our approach to two main baselines: Base-RL and Tuned-RL, both trained using the PPO algorithm with identical hyperparameters (Schulman et al., 2017). Base-RL utilizes a naive reward function that gives a positive, progressive reward based on the distance to the goal. For Tuned-RL, a hand-crafted, dense reward function is

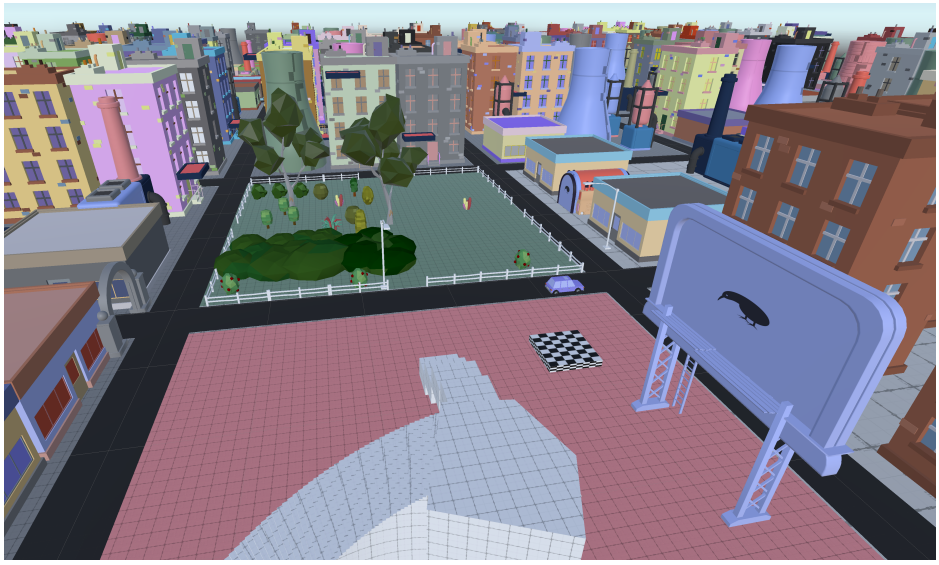


Figure 8.4: Overview of the environment used in this study.

used that is instead designed to lead the agent along a path, reaching multiple sub-goals before the final main goal. We use the same settings for all subsequent experiments. For each of the methods, we are interested in: the success rate of the agent (i.e. how many times it reaches the goal), training time (i.e. the time it takes to reach that success rate), generalization success rate (i.e. the success rate of the same agent in a different version of the environment), and imitation metric (i.e. how close the trajectories made by the agent are to the demonstrations). For the latter, we use the 3D Frechét distance between the agent trajectories and the demonstrator ones. As we showed in Chapter 7, the Frechét distance is a good metric to compare trajectories in 3D spaces as it takes into account the spatial relationships between the points in the curves, rather than just the raw coordinates of the points. Moreover, it is flexible enough to handle different types of curves and it is quite robust to noise. For these reasons is a good measurement of how far a given policy trajectories is with respect to the demonstrator ones. Other distance metrics, such as the Euclidean distance and Hausdorff distance, may not be as robust and do not consider the shape of the curve. While dynamic time warping is another option, it can be computationally expensive for large datasets.

To optimize the policy network we used the DAgger algorithm (Ross et al., 2011). The number of demonstrations used for training the policy depends on the use case: we started with an average of 4 episodes for the first iteration of training, then we correct the behavior of the agent with all the necessary demonstrations. Most of the hyperparameter values were chosen after many preliminary experiments made with different configuration: learning rate $\mathrm{lr} = 0.0005$, batch size $\mathrm{bs} = 512$, and number of epochs $\mathrm{e}_0 = 300$ for the first DAgger iteration, and $\mathrm{e}_i = 100$ for the subsequent ones. We found this set of values to work well for

all use cases. All training was performed on a single machine with an NVIDIA GTX 1080 GPU with 8 GB VRAM, a 6-core Intel Xeon W-2133 CPU, and 64 GB of RAM.

Use-case Evaluation

In order to evaluate the performance of the approach, we define a set of use-cases that resemble typical situations that game designers face in their daily workflow. In Table 8.1, we report quantitative results for each use-case, comparing our approach to the aforementioned RL baselines.

Use-case 1: Validating Design Changes. For the first use case, we investigate whether an IL agent can validate in real-time the layout of a level. The goal in this experiment is to train an agent to follow human demonstrations, then to change the level and see how the agent adapts to these changes. This will give developers a glimpse into the usefulness of our proposed approach when they are still in the design process: they can see how players would adapt accordingly to modifications in the level layout. In Figure 8.5, we give the details of the use case. The agent is tasked to navigate to a goal hidden behind a door that can be opened by interaction with a button. After training the agent, we test it by removing and adding new elements to the level. We argue, with the support of Table 8.1, that IL is more suitable than RL for use cases like this. IL is generally significantly faster than RL, and at the same time it gives the user much more control over the final agent behavior as the agent can be guided via demonstrations. As seen in this table, we could improve controllability and training time of the RL solution (e.g. Tuned-RL), but this requires reward shaping, which is a very difficult task – especially for non-experts in machine learning (Jacob et al., 2020). Moreover, defining a suitable reward function requires many adjustment iterations to the training setup that decreases the overall efficiency of the system. It is evident from the table that IL gives the same level of generalization as RL and it adapts easily to slight variations of the same environment.

Use-case 2: Validating Complex Trajectories. For this use case we train our agent to reproduce a complex trajectory as shown in Figure 8.6. The agent has multiple intermediate goals: use the elevator, interact with the button to create the bridge, destroy a wall by shooting it, and arrive at the goal location. Here we are not interested in generalization, but only in the ability to replicate the expert behavior as quickly and closely as possible. In Table 8.1, we see how our interactive approach is much more suitable for our goals compared to the baselines. As the table shows, it takes a lot of time for the RL agent to train, which is problematic as efficiency is a key requirement. Even the Tuned-RL baseline is much more time consuming than our approach. Moreover, the RL agent will exploit the environment without taking into account the real intention of the designer. An interesting finding here is that the RL agent has found a different way to get to the goal location, which is not desirable for this specific use case but would be very useful for exploit detection.

Use-case 3: Navigation. Navigation is one of the fundamental aspects of many modern

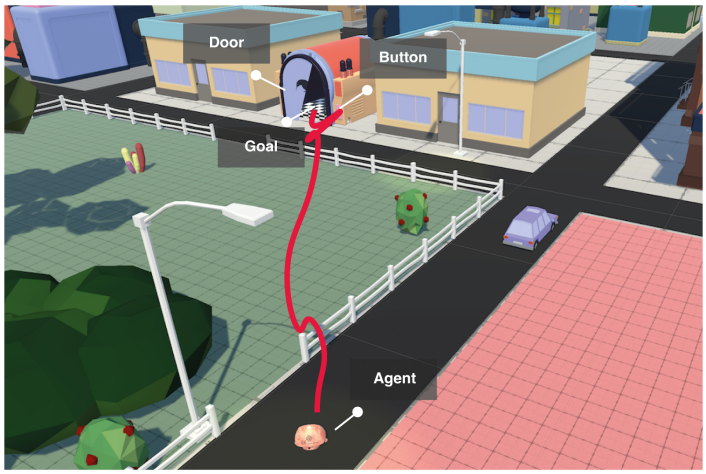


Figure 8.5: Screenshot of the "*Validating Design Changes*" use-case.

video games. The traditional scripted way to handle navigation is to pre-bake and use navigation meshes in combination with classical pathfinding algorithms. However, using a navigation mesh becomes intractable in many practical situations. Navigation meshes are computationally expensive to generate and maintain in complex environments, and have several limitations. They are unable to geometrically represent areas that can only be accessed through jumping, leading to characters getting stuck or behaving unexpectedly. Moreover, a navigation mesh may not be able to accurately represent dynamic or changing environments, such as those with moving platforms or objects that alter the shape of the walkable area. As a result, a trade-off must often be made between navigation cost and quality. This is achieved in practice by either removing some of the navigation abilities or by pruning navigation mesh connections (Alonso et al., 2021). Moreover, every time designers change something in the layout, the navigation mesh needs to be regenerated. For this use case, we consider a complex navigation task where the agent has to navigate through a city that the designers have not completely finished yet. To simulate this, we first record demonstrations in an unfinished city, and then we evaluate the trained agent within a test city that changes every 2 seconds until it reaches the goal location. A screenshot of this use case is given in Figure 8.7. Similar to the second use case, we want the agent to always follow the same path that is demonstrated. Table 8.1 summarizes the results. Since the city environment is large, training agents with RL would take a lot of time compared to guiding them along the correct path. Moreover, even if they enjoy a similar level of generalization, the corresponding RL agents will try to explore the environment finding different ways to arrive at the goal location. The Tuned-RL baseline, however, achieves slightly better results in this case. We must reiterate that this baseline is infeasible to replicate by a designer as it needs a specific reward function and many iterations. Even then, the overall result is not very different from our approach. As shown by the results from these experiments, the combination of these elements is more easily achievable with IL

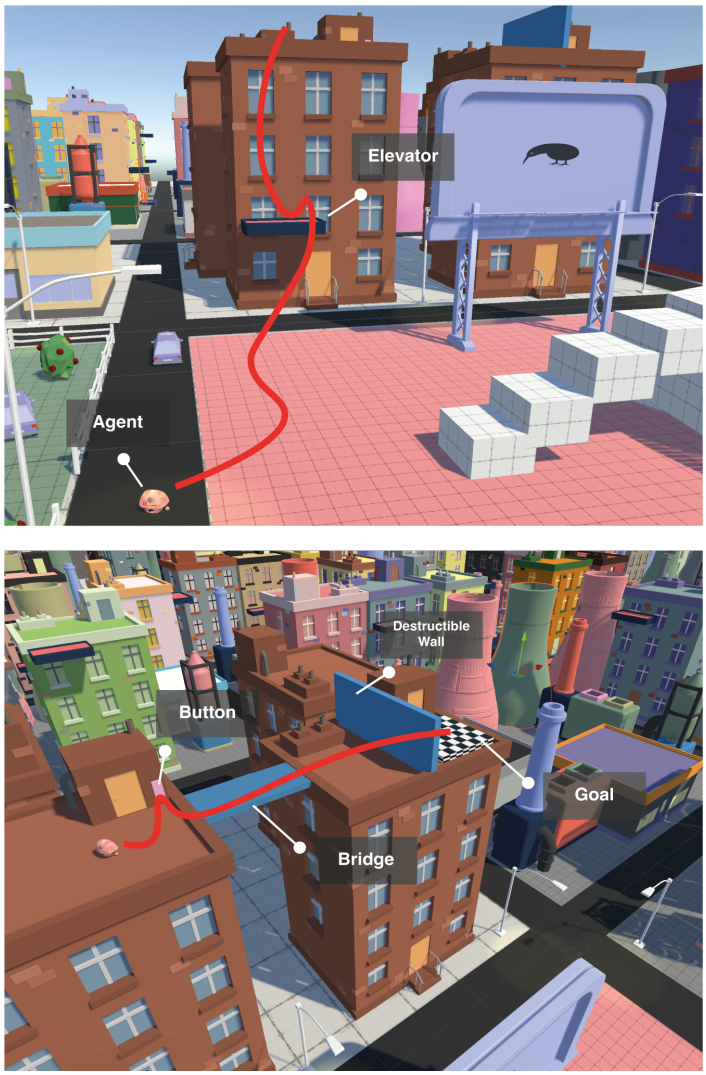


Figure 8.6: Screenshot of the "*Validating a Complex Trajectory*" use-case.

rather than RL.

8.6 User Study

We performed a qualitative user study in the form of an online survey * with professional game and level designers, not only to assess validity and desirability of using our proposed approach but also to identify open opportunities for improving automated game validation.

*A video describing what was shown in the survey is available at: https://vimeo.com/754718818/011e28a122.

Figure 8.7: Screenshot of the "*Navigation*" use-case.

	Use-case 1			
	Success ↑	Time ↓	Generalization ↑	Imitation ↓
Ours	**0.95 ± 0.00**	**0.02** h	**0.96 ± 0.05**	**6.84 ± 0.34**
Simple-RL	0.91 ± 0.02	5.00 h	0.90 ± 0.00	8.37 ± 0.27
Tuned-RL	0.95 ± 0.00	4.48 h	0.90 ± 0.02	8.13 ± 0.20
	Use-case 2			
	Success ↑	Time ↓	Generalization ↑	Imitation ↓
Ours	0.90 ± 0.02	**0.06** h	-	**7.73 ± 1.10**
Simple-RL	0.00 ± 0.00	18.18 h	-	28.11 ± 0.41
Tuned-RL	**0.92 ± 0.03**	13.56 h	-	9.53 ± 1.37
	Use-case 3			
	Success ↑	Time ↓	Generalization ↑	Imitation ↓
Ours	0.81 ± 0.08	**0.22** h	0.78 ± 0.03	46.07 ± 1.03
Simple-RL	0.00 ± 0.00	16.49 h	0.00 ± 0.00	80.22 ± 0.53
Tuned-RL	**0.86 ± 0.05**	4.12 h	**0.87 ± 0.03**	**16.79 ± 2.12**

Table 8.1: Quantitative results of our experiments. We compare our approach with two main baselines: Simple-RL, which uses PPO algorithm with a sparse reward function and Tuned-RL which uses an hand-crafted and very dense reward function. All the numbers refer to the mean and standard deviation of 5 training runs. For a complete description of the use-cases, see Section 8.5. Since in use-case 2 we are not interested in generalization, we do not report the values for that experiment.

ID	Role	Genre(s)	YoE	MLK
P01	Level Designer	FPS	15	Low
P02	Level Designer	FPS	11	None
P03	Level Designer	Racing	3	Very Low
P04	Level Designer	RPG	6	High
P05	Level Designer	Racing	1	Very Low
P06	Level Designer	RPG	4	Very Low
P07	Game Designer	RPG	9	Low
P08	Game Designer	Sport	3	Very Low
P09	Game Designer	Sport	4	Very Low
P10	Game Designer	Match3	1	Very Low
P11	Level Designer	RPG	15	Very Low
P12	Level Designer	FPS	21	Low
P13	Gameplay Designer	RPG	15	Very Low
P14	Game Designer	RPG	22	None
P15	Level Designer	FPS, racing	10	Very Low
P16	Level Designer	RPG	5	Very Low

Table 8.2: Summary of participants. Abbreviations: YoE (years of experience), MLK (machine learning knowledge), FPS (first person shooter), RPG (role-playing game), TPS (third person shooter).

Survey Description

The data for the survey were collected from participants recruited using snowball sampling. Subjects were recruited among different game studios of varying sizes, with different background knowledge and workflows. The survey is composed of Likert questions with some additional open questions. We also let participants provide additional feedback to individual questions. In Table 8.2 we report the details of all survey participants. In Table 8.3 we report all the questions in the user survey.

The survey was divided into four sections: the first asks participants of some background information; the second asks them about their current game validation workflow, in particular if they use manual or automated playtesting; the third, being the main part of the survey, asks participants what they think about our solution, if they would use it in their games, what characteristics an agent/approach like this should have to help them in their game and level design work, if they think IL would help them create better games; and the fourth contains optional questions about possible use cases and future directions they see that we had not considered.

Since one of the main focuses of this chapter is to assess what improvements and research are needed to maximize the value of using our proposed approach, one important question in the third section of the survey asks participants to evaluate various characteristics that an agent should display for automated content evaluation. These characteristics are:

- *Imitation*: the agent can exactly replicate the demonstrations;
- *Generalization*: the agent can adapt to different variations of the same situation;
- *Exploration*: the agent can explore beyond the demonstrations and find bugs and issues;

Table 8.3: All the questions presented to industry experts through the survey. The *Type* column refers to the type of answer allowed for each question: *Open* allowed participants to answer with a free-text response; *Multi* allowed participants to choose from list of answers; *[Yes, No]* allowed participants to answer either yes or no; and "*[1, 7]*" allowed participants to answer with a value between 1 and 7, where 1 generally means "*totally disagree*" and 7 means "*totally agree*".

N°	Question	Type
1	What is your area of expertise?	Open
2	What game or franchise do you work on?	Open
3	What is your level of machine learning knowledge?	Multi
4	How many years of experience do you have in the video game industry?	Open
5	What tools/software do you most commonly use in your everyday work?	Open
6	Is any part of the level design evaluation process automatic in your current workflow?	[Yes, No]
7	Have you used scripted automated method in your daily work?	[Yes, No]
8	How often do you rely on automated rather than manual playtesting?	Multi
9	Were the examples shown realistic enough to resemble AAA level design evaluation?	[1, 7]
10	Generally, do you think automated evaluation and testing of the level directly in the game editor is useful?	[1, 7]
11	How useful do you think the demonstrated agent could be in assisting with content evaluation?	[1, 7]
12	What are the important characteristics that an agent for automated content evaluation should have? Please rank the options below	-
12*	Imitation (the agent can exactly replicate the demonstrations)	[1, 7]
12*	Generalization (the agent can adapt to different variations of the same situation)	[1, 7]
12*	Exploration (the agent can explore beyond the demonstrations and find bugs and issues)	[1, 7]
12*	Personas (the agent can have different types of behaviors)	[1, 7]
12*	Efficiency (the agent must use as few demonstrations as possible)	[1, 7]
12*	Controllability (having control over the agent behavior vs complete autonomy)	[1, 7]
12*	Feedback (the agent gives feedback when the amount of demonstration data is enough)	[1, 7]
12*	Fine-Tuning (behaviors can be fine tuned after initial training)	[1, 7]
12*	Interpretability (the agent can inform when and why it fails)	[1, 7]
13	Any other important characteristics that come to mind?	Open
14	Do you think the method demonstrated can be useful to you for the content you create?	[1, 7]
15	Would the method demonstrated allow you to create more meaningful gameplay experiences more easily?	[1, 7]
16	What are the potential problems you see in using a method like this?	Open
18	How does the demonstrated solution compare with scripted agents?	Multi
19	How useful do you think this method would be in aiding your everyday decision-making processes as designer?	[1, 7]
20	Do you see other use cases for this method?	Open
21	What other features would you like to see in a method like this?	Open
21	Do you have any other feedback, comment or idea for us?	Open

- *Personas*: the agent can have different types of behaviors;
- *Efficiency*: the agent must use as few demonstrations as possible;
- *Controllability*: having control over the agent behavior vs complete autonomy;
- *Feedback*: the agent gives feedback when the amount of demonstration data is enough;
- *Fine tuning*: behaviors can be fine tuned after initial training;
- *Interpretability*: the agent can inform me when and why it fails.

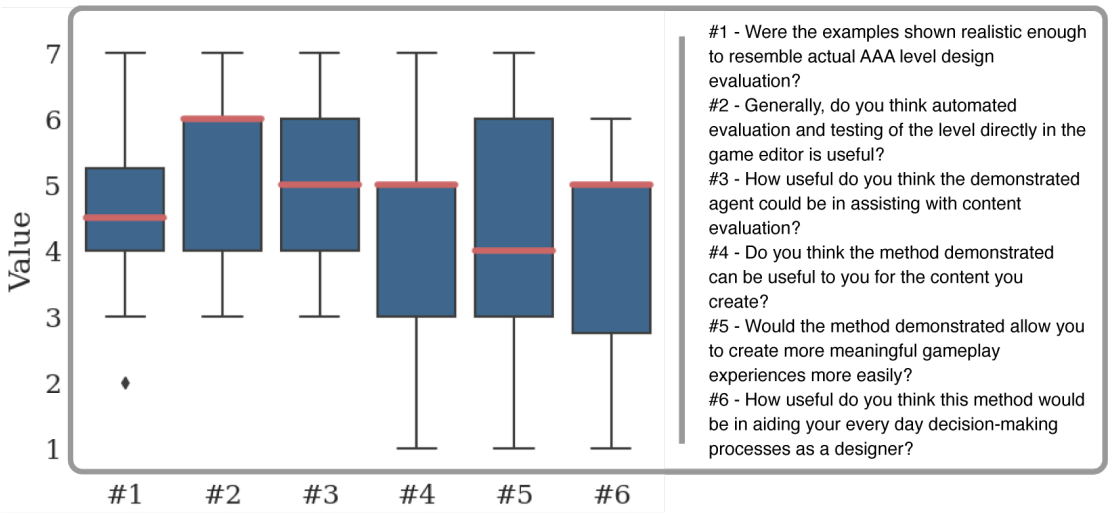


Figure 8.8: Boxplot of answers to some important questions from the survey. The boxes represent the area between the lower and upper quartiles. The red lines represent the median of the values, and the whiskers represent the minimum and maximum of the distributions (except for the outliers that are visualized with black dots).

8.7 Survey Results

We received a total of 16 survey responses. In Figure 8.8 and Figure 8.9 we report some of the most interesting results obtained. The majority (71.4%) of the participants have more than 5 years of experience in level design, with a median of 7.5 years. All the participants are level, game, or gameplay designers. The general machine learning knowledge of the participants range from very low to low. 83.3% of the respondents do not use automatic playtesting and they rely on external people or systems for their validation. 72.2% of designers never rely on automated playtesting and only 38.9% have used at least once a scripted automated method in their daily work. They work on different game projects and game genres and they use different tools in their level design workflow, but all respondents have knowledge of game engine editors. The general feeling is that designers would very likely use a tool like this as *"it definitely would be useful to speed up the typically time consuming iterative design process"* and *"a method like this can definitely speed up iteration, which is one of the main things any designer spends a lot of time"*. Moreover, they can relate to the demonstrated example as "*it is not quite like how designers are building their levels, but it is not far off*" and they think the examples shown are "*realistic for some games such as platformers*".

Participants acknowledge the usefulness and the potential of the demonstrated method, even if it would need the proper adaptations for handling the specific game genre they are working on. Further, 81.2% of the respondents agree that automated validation of the level directly in the game editor is useful and 93.0% of them think the demonstrated agents could

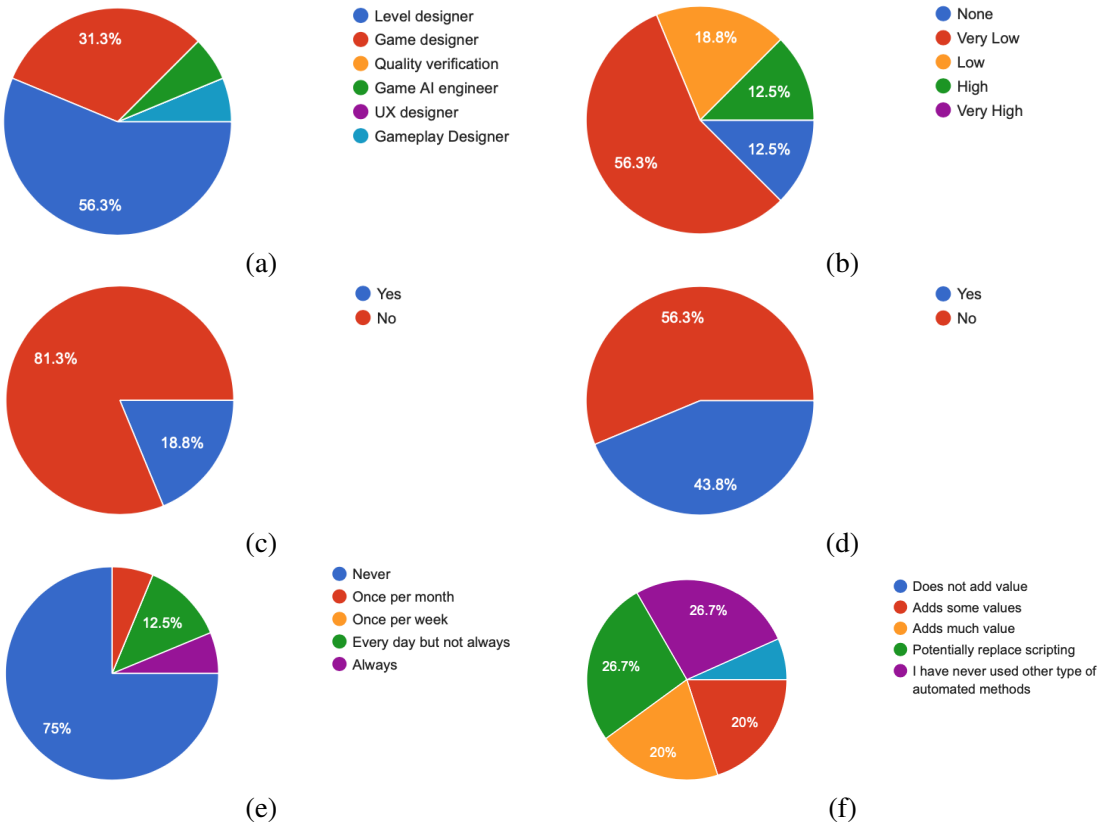


Figure 8.9: Pie charts of answers to some of the most important questions in the survey. (a) The question "*what is your area of expertise?*"; (b) The question "*what is your machine learning knowledge?*"; (c) The question "*is any part of the level design evaluation process automatic in your current workflow?*"; (d) The question "*have you used scripted automated method in your daily work?*"; (e) The question "*how often do you rely in automated rather than manual playtesting?*"; and (f) The question "*how does the demonstrated solution compare with scripted agents?*".

be useful for assisting in validating their content. Many respondents (41.1%) agree that a method like this adds value compared to scripted agents and some of them (23.5%) think it could completely replace scripting.

Some designers are skeptical about the complexity of the examples shown in the survey and they would like to see how well the approach works in more complicated games. One participant says that *"[he is] wondering about how effective this would be in genres that are more complicated mechanically"*. This was expected due to the preliminary nature of this study and that we do not cover every genre but focus on a few. However, a larger group (81.2%) think that complexity was high enough which reinforces the notion that the environment is relevant to a significant part of the community. Designers also describe the challenges of using such a tool in their daily workflow: one argues that "*human players play very differently than bots, can be difficult to only rely on AI when it comes to testing*"

	Mean	Median	Std
Imitation	4.56	4.00	1.63
Generalization	6.50	7.00	0.73
Exploration	6.81	7.00	0.40
Personas	5.87	6.00	1.31
Efficiency	5.50	5.50	1.21
Controllability	4.18	4.00	1.68
Feedback	6.00	6.00	0.96
Fine tuning	5.62	6.00	1.50
Interpretability	6.50	7.00	1.03

Table 8.4: Results of characteristics questions. Participants could give a value between $[1, 7]$ to each of the category, where 1 means "*not so important*" and 7 means "*very important*".

or "*demonstrations take time, which would push busy level designers to not use it. Demonstrating how to solve specific issues – e.g. recover from a mistake – should not be on the level designer*". Not all game creators are fully convinced by the approach because "*the demonstration video showed a process that does not really show a level designer anything new or unexpected*".

After describing their doubts, we asked participants to evaluate each of the characteristics delineated in the previous section to specify which ones are the most important to improve the approach. Table 8.4 summarizes these characteristics ratings. We claim that these results bring much value to our research: not only do we know that our initial hypothesis is supported by professional designers, but we also what they really want and why they are skeptical of using such an approach. This is important because, as we will see in Section 8.8, the values in Table 8.4 form very precise research directions that we encourage all of the game research community to consider in future contributions to this field.

We asked participants to evaluate each of the characteristics delineated in the previous section and which are the most important ones to improve the approach. Table 8.4 refers to the results. We claim that these results bring much value to our research: not only do we know that our initial hypothesis is supported by professional designers, but we also know what they really want and why they are skeptical of using such an approach. This is important because, as we will see in Section 8.8, the values in Table 8.4 form very precise research directions that we encourage all of the game research community to consider in future contributions to the video game industry.

8.8 Conclusions and Future Directions

In this chapter, we first claimed that data-driven programming via imitation learning is a suitable approach for real-time validation of game and level design. We proposed an imitation learning approach and we investigated it focusing on three different design validation use-cases. Our experiments demonstrate how this type of approach can satisfy many of the requirements for being an effective game design tool in comparison to simple reinforcement

learning and model-based scripted behaviors. We also performed a user study with professional game and level designers from different, and in many cases, disparate game studios and game genres. We asked participants to assess the desirability and opportunities of using such an approach in their daily workflow. Moreover, we asked designers what characteristics they would want from a tool for creating autonomous agents that validate their design.

The user study highlights the desire of designers to have an automated way to test and validate their games. Along with our preliminary results we demonstrate that the data-driven approach we proposed is a potential candidate for achieving such objectives. The study also highlights challenges and the gap that exists between techniques from the literature and their actual use in the game industry. In light of the results of our qualitative analyses and experimental results obtained throughout the entire dissertation, we now propose a series of research directions that will help such approaches move beyond an impractical tool to an effective game design tool. With this we want to lower the bar for researcher who would like to contribute and improve this dissertation, as this would help drive the research, and therefore industry, forward.

Generalization. One of the most important requests is to have an agent that not only imitates expert behaviors, but that is more representative of the unpredictable nature of players. With our proposed general purpose neural network and state space we mitigate this problem, but in many cases agents trained under one set of demonstrations are not able to adapt to larger variations (Huang et al., 2017). Generalization as a subject in self-learning agents has already been addressed in the literature, but the state-of-the-art approaches are not readily applicable for our purpose: most of them use either interactions with the game (Ho and Ermon, 2016; Fu et al., 2018; Torabi et al., 2018) or learn the inverse dynamics of the environment (Monteiro et al., 2020). A possible future direction is to try data augmentation techniques used in offline reinforcement learning algorithms such as the work done by Sinha et al. (2022). Offline reinforcement learning has several connections to behavioral cloning (Fujimoto and Gu, 2021). It learns a policy using a pre-defined dataset without direct interactions with the environment, but in contrast with IL that supposes the data comes from an optimal policy, offline reinforcement learning can also leverage sub-optimal examples. This leads to trained agents being more general and allows a higher exploration level. With all this considered, we argue that generalization is far from solved for this use case.

Personas. One recurring feature requested by survey participants is the possibility of training the model for different behavior types, or so called *personas*. The aim is to have various behaviors similar to the multi-modal nature of how humans play, in order to create more meaningful agents. The research community has addressed the problem of creating personas many times, but exclusively in RL contexts. Works from de Woillemont et al. (2021), Roy et al. (2021) have explored how to combine behavior styles with different reward functions which is not applicable to IL. The work by Peng et al. (2021) is an example of how to combine IL and RL to create different animation styles. More research into training for different playstyles with only IL would allow designers to train more meaningful and diverse

testing agents for validating gameplay in different ways.

Exploration. Participants frequently brought up the exploration aspect, i.e. the ability of agent to look beyond expert demonstrations in search for overlooked issues. Exploration is a well known problem in the RL literature (Burda et al., 2018), but exploration in IL is, to our knowledge, largely unexplored. This is mainly due to the conflicting nature of an agent that must both learn to, more or less, precisely follow the expert demonstrations as well as explore beyond the expert behavior. Moreover, we would not use exploration to improve agents in the validation context, but we would like to exploit exploration to find bugs. In Chapter 7 we provided a good example of how to leverage both IL and RL to train agents to both follow demonstrated behaviors but also to explore in search for issues. However, this type of solution is still too sample inefficient to be used as an active game design tool.

Usability. The usability aspect is one of the most important for participants. This includes both providing useful information to designers about the models they are training, and the ease-of-use of the tool. Recent techniques from the explainable RL (Druce et al., 2021) research community could be used to address the challenge of interpreting behaviors of the agent. To improve upon this, one can use techniques from game analytics research and gameplay visualization (Wallner and Kriglstein, 2013). Additionally, a sentiment found in the survey is that for a tool like this to be usable, there is a need to minimize the effort imposed on the end-user. To address the latter, few-shot IL is an active topic that can potentially help to reduce the number of samples required. Works by Duan et al. (2017) and Hakhamaneshi et al. (2021) specifically addressed this problem within IL, mainly exploiting meta-learning techniques (Finn et al., 2017). However, these approaches require many preliminary training iterations and it is unclear how this could be applied to a game in development that is unstable and not yet finished. One way to both improve the agent's quality and the usability of the tool is to leverage the already cited offline reinforcement learning techniques (Kumar et al., 2020; Chen et al., 2021; Sinha et al., 2022). We can leverage the recording process to not only store demonstrations, but all other interactions the agent performs while testing the level.

Multitiple Agents. Many designers noted that most modern video games are multi-agent systems, and in order to thoroughly test these environments we need to train multiple interacting agents. Most of the current research in IL focuses on single agents learning from a single teacher. There are a few examples of multi-agent IL. Harmer et al. (2018) used IL in a multi-agent game, but they do not really address the problem of a multi-agent system, while Le et al. (2017) proposed a joint approach that simultaneously learns a latent coordination model along with the individual policies. We believe there is still a long way to go for meaningful multi-agent IL, especially for this use case, and we encourage researchers to follow this research direction.

Chapter 9

Conclusions

Video games are becoming more and more popular as a form of entertainment. The number of persons playing video games is constantly increasing, and with the rise of portable gaming devices people are playing more video games more often than ever before. The demand for better games is increasing, as people desire games that are more realistic, have better graphics, and are more challenging.

At its core, this dissertation is about the potential of deep reinforcement learning for video game development. In it we propose powerful and flexible tools for creating artificial intelligence agents that can learn to play games in realistic and challenging ways. We believe that this technology will continue to be developed and improved, and that the work described in this dissertation represents a significant step forward. We have shown that deep reinforcement learning can be used to train agents that can play a variety of different games. The agents that we have developed are:

- *credible*: they act in ways that are predictable, that can be interpreted as intelligent, and that offer the right challenge to the player;
- *imperfect*: they are imperfect and not superhuman, as they behave like humans;
- *prior-free*: they learn to play the game without any prior knowledge of the rules and without manual specification of strategies; and
- *various*: they exhibit different behaviors in different situations, like real human players.

To conclude, we put the content of this dissertation into context with respect to the challenges highlighted in Chapter 2 which we think are the most relevant for its contribution to future research in deep reinforcement learning applied to video: the quest for more believable and appealing NPCs. In Figure 9.1 we illustrate a summary of the contributions of this dissertation to each of these challenges.

Generalization. In Chapter 3 and especially in Chapter 4 we showed how to exploit procedural content generation and procedural game rules to train agents that do not memorize

trajectories to achieve goals, but that rather learn an abstract representation of the state-space and of the environment that enables them to generalize and manage all the possible situations that can happen both in playing with human players and during the development process. Moreover, in Chapter 6 we proposed new policy fusion methods able to easily adapt a trained agent to changes in strategy. Since during development games can change on a daily basis, this dissertation describes how to train agents that support developers in adapting agents to their design change workflow.

Reward Function. Engineering a good reward function is a challenging task, even more so if the users of these tools are not experts in machine learning. Our demonstration-efficient inverse reinforcement learning algorithm and policy fusion methods (Chapters 5 and 6) provide a form of fine control over the final behavior of agents without defining, changing, or engineering a reward function. DE-AIRL requires only a few demonstrations and little training time.

Behavior Quality and Novel Testbeds. Our experiments illustrate the potential of the techniques proposed in this dissertation to increase the quality of agent behavior. Our agents are not super-human, but rather learn behaviors suitable for game production, whether we want to use them as NPCs or for automatic playtesting. We tested these abilities in two new and open source game environment testbeds – DeepCrawl, outlined in Chapter 3, and the navigation environment described in Chapter 7 – which we now encourage the research community to use and to continue and expand this research.

Interpretability. Our studies with industry experts showed that deep analysis and understanding of reinforcement learning behaviors are fundamental in order to allow designers to make actionable design decisions, for example whether or not to include a trained behavior in the final release of the game. There is still a long way to go towards true interpretability in reinforcement learning, but we hope our curiosity-conditioned proximal trajectories algorithm described in Chapter 7 can provide a good example of how to support interpretation.

Usability. The studies conducted in this dissertation showed that agents can be trained to a high standard using only small neural networks and a few days of training. Chapter 8 demonstrated how, in some cases, the training can be completed in a matter of minutes. Although there is still some skepticism among professional game developers about the usability of machine learning, the approach described in Chapter 8 suggests that imitation learning is a powerful technique that can close the gap between academic findings and game industry.

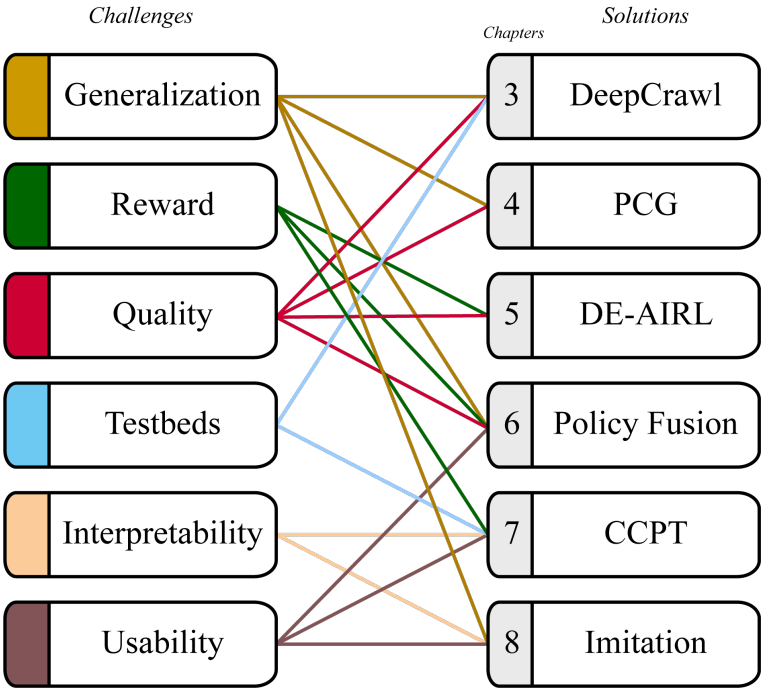


Figure 9.1: List of challenges that interfere with the wide application of DRL in video game development, alongside with solutions proposed in this dissertation. To the left of each solution, we highlight the chapter of this manuscript where the interested reader can read about that topic.

Appendix A

Publications

Peer Reviewed Conference Papers

1. **Alessandro Sestini**, Alexander Kuhnle, and Andrew D. Bagdanov. "Policy Fusion for Adaptive and Customizable Reinforcement Learning Agents". *In proceedings of Conference on Games (CoG)*, 2021. **Candidate's contributions**: contributed to literature review, algorithm design, theoretical analysis; designed and carried out the experiments.

2. **Alessandro Sestini**, Alexander Kuhnle, and Andrew D. Bagdanov. "Demonstration-efficient Inverse Reinforcement Learning in Procedurally Generated Environments". *In proceedings of Conference on Games (CoG)*, 2021. **Candidate's contributions**: contributed to literature review, algorithm design, theoretical analysis; designed and carried out the experiments.

Peer Reviewed Journal Papers

1. **Alessandro Sestini**, Linus Gisslén, Joakim Bergdahl, Konrad Tollmar and Andrew D. Bagdanov. "Automatic Gameplay Testing and Validation with Curiosity-Conditioned Proximal Trajectories". *In Transaction on Games (ToG)*, 2022. **Candidate's contributions**: contributed to literature review, algorithm design, environment design, visualization tool design, theoretical analysis; designed and carried out both quantitative and qualitative experiments.

Peer Reviewed Workshop Papers

1. **Alessandro Sestini**, Joakim Bergdahl, Konrad Tollmar, Andrew D. Bagdanov and Linus Gisslén. "Towards Informed Design and Validation Assistance in Computer Games Using Imitation Learning". *Accepted at NeurIPS-22 Workshop on Human-in-the-Loop Learning*, 2022. **Candidate's contributions**: contributed to literature review,

algorithm design, environment design, theoretical analysis; designed and carried out both quantitative and qualitative experiments.

2. **Alessandro Sestini**, Alexander Kuhnle, and Andrew D. Bagdanov. "Deep Policy Networks for NPC Behaviors that Adapt to Changing Design Parameters in Roguelike Games". *AAAI-21 Workshop on Reinforcement Learning in Games*, 2021. **Candidate's contributions**: contributed to literature review, algorithm design, theoretical analysis; designed and carried out the experiments.

3. **Alessandro Sestini**, Alexander Kuhnle, and Andrew D. Bagdanov. "DeepCrawl: Deep Reinforcement Learning for Turn-based Strategy Games". *AIIDE-19 Workshop on Experimental AI in Games*, 2019. **Candidate's contributions**: contributed to literature review, algorithm design, environment design, theoretical analysis; designed and carried out the experiments.

Papers Under Review

1. Tommaso Aldinucci, Enrico Civitelli, Leonardo di Gangi, **Alessandro Sestini**. "Contextual Decision Trees". *Submitted to Machine Learning (MACH)*, 2022. **Candidate's contributions**: contributed to methodology design and literature review.

Bibliography

Agarwal, S., Herrmann, C., Wallner, G., and Beck, F. (2020). Visualizing AI playtesting data of 2D side-scrolling games. In *2020 IEEE Conference on Games (CoG)*, pages 572–575.

Alonso, E., Peter, M., Goumard, D., and Romoff, J. (2021). Deep reinforcement learning for navigation in AAA video games. In Zhou, Z.-H., editor, *Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI-21)*, pages 2133–2139.

Amodei, D., Olah, C., Steinhardt, J., Christiano, P., Schulman, J., and Mané, D. (2016). Concrete problems in AI safety. *arXiv preprint arXiv:1606.06565*.

Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong, R., Welinder, P., McGrew, B., Tobin, J., Abbeel, P., and Zaremba, W. (2017). Hindsight experience replay. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS)*, pages 5055–5065.

Arulkumaran, K., Deisenroth, M. P., Brundage, M., and Bharath, A. A. (2017). Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38.

Asperti, A., Cortesi, D., and Sovrano, F. (2018). Crawling in Rogue's dungeons with (partitioned) A3C. *arXiv e-prints*, page arXiv:1804.08685.

Aubret, A., Matignon, L., and Hassas, S. (2019). A survey on intrinsic motivation in reinforcement learning. *arXiv preprint arXiv:1908.06976*.

Aytemiz, B., Jacob, M., and Devlin, S. (2021). Acting with style: Towards designer-centred reinforcement learning for the video games industry. In *CHI 2021 Workshop on Reinforcement Learning for Humans, Computer, and Interaction (RL4HCI)*. Association for Computing Machinery (ACM).

Bain, M. and Sammut, C. (1995). A framework for behavioural cloning. In *Machine Intelligence 15*, pages 103–129.

Baker, B., Kanitscheider, I., Markov, T., Wu, Y., Powell, G., McGrew, B., and Mordatch, I. (2019). Emergent tool use from multi-agent autocurricula. *arXiv preprint arXiv:1909.07528*.

Beattie, C., Leibo, J. Z., Teplyashin, D., Ward, T., Wainwright, M., Küttler, H., Lefrancq, A., Green, S., Valdés, V., Sadik, A., et al. (2016). Deepmind lab. *arXiv preprint arXiv:1612.03801*.

Bellemare, M., Srinivasan, S., Ostrovski, G., Schaul, T., Saxton, D., and Munos, R. (2016). Unifying count-based exploration and intrinsic motivation. *Advances in neural information processing systems*, 29.

Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279.

Bellman, R. (1954). The theory of dynamic programming. *Bulletin of the American Mathematical Society*, 60(6):503–515.

Bengio, Y., Louradour, J., Collobert, R., and Weston, J. (2009). Curriculum learning. In *Proceedings of ICML*.

Bergdahl, J., Gordillo, C., Tollmar, K., and Gisslén, L. (2020). Augmenting automated game testing with deep reinforcement learning. In *IEEE Conference on Games (CoG)*.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym.

Brown, D., Goo, W., Nagarajan, P., and Niekum, S. (2019). Extrapolating beyond suboptimal demonstrations via inverse reinforcement learning from observations. In *International Conference on Machine Learning*, pages 783–792. PMLR.

Brys, T., Harutyunyan, A., Vrancx, P., Taylor, M. E., Kudenko, D., and Nowé, A. (2014a). Multi-objectivization of reinforcement learning problems by reward shaping. In *2014 international joint conference on neural networks (IJCNN)*, pages 2315–2322. IEEE.

Brys, T., Nowé, A., Kudenko, D., and Taylor, M. (2014b). Combining multiple correlated reward and shaping signals by measuring confidence. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 28.

Burda, Y., Edwards, H., Storkey, A., and Klimov, O. (2018). Exploration by random network distillation. In *International Conference on Learning Representations (ICLR)*.

Cai, X.-Q., Ding, Y.-X., Jiang, Y., and Zhou, Z.-H. (2019). Expert-level Atari imitation learning from demonstrations only. *arXiv preprint arXiv:1909.03773*.

Carry Castle Studio (2022). Source of madness. https://sourceofmadness.com.

Chang, K., Aytemiz, B., and Smith, A. M. (2019). Reveal-more: Amplifying human effort in quality assurance testing using automated exploration. In *2019 IEEE Conference on Games (CoG)*, pages 1–8. IEEE.

Chen, L., Lu, K., Rajeswaran, A., Lee, K., Grover, A., Laskin, M., Abbeel, P., Srinivas, A., and Mordatch, I. (2021). Decision transformer: Reinforcement learning via sequence modeling. *Advances in neural information processing systems*, 34:15084–15097.

Chevalier-Boisvert, M. (2018). gym-miniworld environment for openai gym. https://github.com/maximecb/gym-miniworld.

Chevalier-Boisvert, M., Willems, L., and Pal, S. (2019). Minimalistic gridworld environment for openai gym. https://github.com/maximecb/gym-minigrid.

Christiano, P. F., Leike, J., Brown, T., Martic, M., Legg, S., and Amodei, D. (2017). Deep reinforcement learning from human preferences. In *Advances in Neural Information Processing Systems*.

Cobbe, K., Hesse, C., Hilton, J., and Schulman, J. (2019). Leveraging procedural generation to benchmark reinforcement learning. *arXiv preprint arXiv:1912.01588*.

de Woillemont, P. L. P., Labory, R., and Corruble, V. (2021). Configurable agent with reward as input: a play-style continuum generation. In *2021 IEEE Conference on Games (CoG)*, pages 1–8. IEEE.

Delalleau, O., Peter, M., Alonso, E., and Logut, A. (2019). Discrete and continuous action representation for practical RL in video games. *Proceedings of AAAI-20 Workshop on Reinforcement Learning in Games*.

Devlin, S., Georgescu, R., Momennejad, I., Rzepecki, J., Zuniga, E., Costello, G., Leroy, G., Shaw, A., and Hofmann, K. (2021). Navigation turing test (NTT): Learning to evaluate human-like navigation. In *2021 International Conference on Machine Learning (ICML)*.

Druce, J., Harradon, M., and Tittle, J. (2021). Explainable artificial intelligence (XAI) for increasing user trust in deep reinforcement learning driven autonomous systems. *arXiv preprint arXiv:2106.03775*.

Duan, Y., Andrychowicz, M., Stadie, B., Jonathan Ho, O., Schneider, J., Sutskever, I., Abbeel, P., and Zaremba, W. (2017). One-shot imitation learning. *Advances in neural information processing systems*, 30.

Ecoffet, A., Huizinga, J., Lehman, J., Stanley, K. O., and Clune, J. (2021). First return, then explore. *Nature*, 590(7847):580–586.

Faußer, S. and Schwenker, F. (2011). Ensemble methods for reinforcement learning with function approximation. In *International Workshop on Multiple Classifier Systems*, pages 56–65. Springer.

Finances Online (2022). Number of gamers worldwide 2022/2023: Demographics, statistics, and predictions. https://financesonline.com/number-of-gamers-worldwide/.

Finn, C., Abbeel, P., and Levine, S. (2017). Model-agnostic meta-learning for fast adaptation of deep networks. In *International conference on machine learning*, pages 1126–1135. PMLR.

Finn, C., Christiano, P., Abbeel, P., and Levine, S. (2016a). A connection between generative adversarial networks, inverse reinforcement learning, and energy-based models. *arXiv preprint arXiv:1611.03852*.

Finn, C., Levine, S., and Abbeel, P. (2016b). Guided cost learning: Deep inverse optimal control via policy optimization. In *International Conference on Machine Learning*.

Fu, J., Luo, K., and Levine, S. (2018). Learning robust rewards with adverserial inverse reinforcement learning. In *International Conference on Learning Representations*.

Fujimoto, S. and Gu, S. S. (2021). A minimalist approach to offline reinforcement learning. *Advances in neural information processing systems*, 34:20132–20145.

Gisslén, L., Eakins, A., Gordillo, C., Bergdahl, J., and Tollmar, K. (2021). Adversarial reinforcement learning for procedural content generation. In *2021 IEEE Conference on Games (CoG)*, pages 1–8.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial nets. In *Advances in Neural Information Processing Systems*.

Gordillo, C., Bergdahl, J., Tollmar, K., and Gisslén, L. (2021). Improving playtesting coverage via curiosity driven reinforcement learning agents. In *2021 IEEE Conference on Games (CoG)*, pages 1–8.

Guss, W. H., Houghton, B., Topin, N., Wang, P., Codel, C., Veloso, M., and Salakhutdinov, R. (2019). MineRL: a large-scale dataset of minecraft demonstrations. In *International Joint Conference on Artificial Intelligence*.

Hakhamaneshi, K., Zhao, R., Zhan, A., Abbeel, P., and Laskin, M. (2021). Hierarchical few-shot imitation with skill transition models. *arXiv preprint arXiv:2107.08981*.

Hans, A. and Udluft, S. (2010). Ensembles of neural networks for robust reinforcement learning. In *2010 Ninth International Conference on Machine Learning and Applications*, pages 401–406. IEEE.

Harmer, J., Gisslén, L., del Val, J., Holst, H., Bergdahl, J., Olsson, T., Sjöö, K., and Nordin, M. (2018). Imitation learning with concurrent actions in 3d games. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE.

Ho, J. and Ermon, S. (2016). Generative adversarial imitation learning. In *Proceedings of the 30st International Conference on Neural Information Processing Systems (NIPS)*, pages 4565–4573.

Holmgård, C., Green, M. C., Liapis, A., and Togelius, J. (2018). Automated playtesting with procedural personas through MCTS with evolved heuristics. *IEEE Transactions on Games*, 11(4):352–362.

Huang, S. and Ontañón, S. (2020). Action guidance: Getting the best of sparse rewards and shaped rewards for real-time strategy games. *arXiv preprint arXiv:2010.03956*.

Huang, S., Papernot, N., Goodfellow, I., Duan, Y., and Abbeel, P. (2017). Adversarial attacks on neural network policies. *arXiv preprint arXiv:1702.02284*.

Ibarz, B., Leike, J., Pohlen, T., Irving, G., Legg, S., and Amodei, D. (2018). Reward learning from human preferences and demonstrations in Atari. In *Advances in Neural Information Processing Systems*.

Icaza, M. (2019). Tensorflowsharp: Tensorflow API for .NET languages. Accessed: January 18, 2019.

Jacob, M., Devlin, S., and Hofmann, K. (2020). It's unwieldy and it takes a lot of time. Challenges and opportunities for creating agents in commercial games. In *16th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*.

Jaunet, T., Vuillemot, R., and Wolf, C. (2020). DRLViz: Understanding decisions and memory in deep reinforcement learning. In *Computer Graphics Forum*, volume 39, pages 49–61.

Juliani, A., Berges, V.-P., Vckay, E., Gao, Y., Henry, H., Mattar, M., and Lange, D. (2018). Unity: A General Platform for Intelligent Agents. *arXiv e-prints*, page arXiv:1809.02627.

Juliani, A., Khalifa, A., Berges, V.-P., Harper, J., Teng, E., Henry, H., Crespi, A., Togelius, J., and Lange, D. (2019). Obstacle tower: A generalization challenge in vision, control, and planning. In *International Joint Conference on Artificial Intelligence*.

Justesen, N., Torrado, R., Bontrager, P., Khalifa, A., Togelius, J., and Risi, S. (2018). Illuminating generalization in deep reinforcement learning through procedural level generation. *Proceedings of NIPS Workshop on Deep Reinforcement Learning*.

Kempka, M., Wydmuch, M., Runc, G., Toczek, J., and Jaśkowski, W. (2016). ViZDoom: a Doom-based AI research platform for visual reinforcement learning. In *IEEE Conference on Computational Intelligence and Games*. IEEE.

Knox, W. B. and Stone, P. (2009). Interactively shaping agents via human reinforcement: the TAMER framework. In *Proceedings of the 5th international conference on Knowledge capture*, pages 9–16.

Konda, V. R. and Tsitsiklis, J. N. (2003). On actor-critic algorithms. *SIAM J. Control Optim.*, 42(4).

Kuhnle, A., Schaarschmidt, M., and Fricke, K. (2017). Tensorforce: a tensorflow library for applied reinforcement learning. Web page. Accessed: January 6, 2019.

Kumar, A., Zhou, A., Tucker, G., and Levine, S. (2020). Conservative Q-learning for offline reinforcement learning. *Advances in Neural Information Processing Systems*, 33:1179–1191.

Küttler, H., Nardelli, N., Miller, A. H., Raileanu, R., Selvatici, M., Grefenstette, E., and Rocktäschel, T. (2020). The nethack learning environment. *arXiv preprint arXiv:2006.13760*.

Le, H. M., Yue, Y., Carr, P., and Lucey, P. (2017). Coordinated multi-agent imitation learning. In *International Conference on Machine Learning*, pages 1995–2003. PMLR.

Lučić, M., Tschannen, M., Ritter, M., Zhai, X., Bachem, O., and Gelly, S. (2019). High-fidelity image generation with fewer labels. In *International Conference on Machine Learning*.

Mao, X., Li, Q., Xie, H., Lau, R. Y., Wang, Z., and Paul Smolley, S. (2017). Least squares generative adversarial networks. In *Proceedings of the IEEE international conference on computer vision (ICCV)*, pages 2794–2802.

Mescheder, L., Geiger, A., and Nowozin, S. (2018). Which training methods for GANs do actually converge? In *2018 International conference on machine learning (ICML)*, pages 3481–3490.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

Monteiro, J., Gavenski, N., Granada, R., Meneguzzi, F., and Barros, R. (2020). Augmented behavioral cloning from observation. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE.

Mugrai, L., Silva, F., Holmgård, C., and Togelius, J. (2019). Automated playtesting of matching tile games. In *2019 IEEE Conference on Games (CoG)*, pages 1–7. IEEE.

Ng, A. Y. and Russell, S. (2000). Algorithms for inverse reinforcement learning. In *International Conference on Machine Learning*.

Oh, I., Rho, S., Moon, S., Son, S., Lee, H., and Chung, J. (2019). Creating Pro-Level AI for Real-Time Fighting Game with Deep Reinforcement Learning. *arXiv e-prints*, page arXiv:1904.03821.

Open AI (2016). Faulty reward functions in the wild. https://openai.com/blog/faulty-reward-functions/.

OpenAI, Berner, C., Brockman, G., Chan, B., Cheung, V., Debiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., Józefowicz, R., Gray, S., Olsson, C., Pachocki, J., Petrov, M., de Oliveira Pinto, H. P., Raiman, J., Salimans, T., Schlatter, J., Schneider, J., Sidor, S., Sutskever, I., Tang, J., Wolski, F., and Zhang, S. (2019). Dota 2 with large scale deep reinforcement learning. *arXiv preprint 1912.06680*.

Pathak, D., Agrawal, P., Efros, A. A., and Darrell, T. (2017). Curiosity-driven exploration by self-supervised prediction. In *2017 International conference on machine learning (ICML)*, pages 2778–2787.

Peng, X. B., Berseth, G., and Van de Panne, M. (2016). Terrain-adaptive locomotion skills using deep reinforcement learning. *ACM Transactions on Graphics (TOG)*, 35(4):1–12.

Peng, X. B., Chang, M., Zhang, G., Abbeel, P., and Levine, S. (2019). Mcp: Learning composable hierarchical control with multiplicative compositional policies. *arXiv preprint arXiv:1905.09808*.

Peng, X. B., Ma, Z., Abbeel, P., Levine, S., and Kanazawa, A. (2021). AMP: adversarial motion priors for stylized physics-based character control. *ACM Trans. Graph.*, 40(4).

Pleines, M., Zimmer, F., and Berges, V. (2019). Action spaces in deep reinforcement learning to mimic human input devices. In *2019 IEEE Conference on Games (CoG)*, pages 1–8.

Politowski, C., Guéhéneuc, Y.-G., and Petrillo, F. (2022). Towards automated video game testing: Still a long way to go. *arXiv preprint arXiv:2202.12777*.

Press, W. H. and Teukolsky, S. A. (1990). Savitzky-golay smoothing filters. *Computers in Physics*, 4(6):669–672.

Puiutta, E. and Veith, E. (2020). Explainable reinforcement learning: A survey. In *International cross-domain conference for machine learning and knowledge extraction*, pages 77–95.

Radford, A., Metz, L., and Chintala, S. (2016). Unsupervised representation learning with deep convolutional generative adversarial networks. In *International Conference on Learning Representations*.

Randløv, J. and Alstrøm, P. (1998). Learning to drive a bicycle using reinforcement learning and shaping. In *ICML*, volume 98, pages 463–471.

Reddy, S., Dragan, A. D., and Levine, S. (2019). Sqil: Imitation learning via reinforcement learning with sparse rewards. In *International Conference on Learning Representations*.

Risi, S. and Togelius, J. (2019). Procedural content generation: from automatically generating game levels to increasing generality in machine learning. *arXiv preprint arXiv:1911.13071*.

Roa-Vicens, J., Wang, Y., Mison, V., Gal, Y., and Silva, R. (2019). Adversarial recovery of agent rewards from latent spaces of the limit order book. In *Advances in Neural Information Processing Systems*.

Ross, S., Gordon, G., and Bagnell, D. (2011). A reduction of imitation learning and structured prediction to no-regret online learning. In *2011 International Conference on Artificial Intelligence and Statistics (ICAIS)*.

Roy, J., Girgis, R., Romoff, J., Bacon, P.-L., and Pal, C. (2021). Direct behavior specification via constrained reinforcement learning. *arXiv preprint arXiv:2112.12228*.

Rummery, G. A. and Niranjan, M. (1994). *On-line Q-learning using connectionist systems*, volume 37. Citeseer.

Schaarschmidt, M., Kuhnle, A., Ellis, B., Fricke, K., Gessert, F., and Yoneki, E. (2018). LIFT: Reinforcement learning in computer systems by learning from demonstrations. *CoRR*, abs/1808.07903.

Schulman, J., Levine, S., Abbeel, P., Jordan, M., and Moritz, P. (2015). Trust region policy optimization. In *International Conference on Machine Learning*.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal Policy Optimization Algorithms. *arXiv e-prints*, page arXiv:1707.06347.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587).

Sinha, S., Mandlekar, A., and Garg, A. (2022). S4rl: Surprisingly simple self-supervision for offline reinforcement learning in robotics. In *Conference on Robot Learning*, pages 907–917. PMLR.

Stadie, B. C., Levine, S., and Abbeel, P. (2015). Incentivizing exploration in reinforcement learning with deep predictive models. *arXiv preprint arXiv:1507.00814*.

Stahlke, S., Nova, A., and Mirza-Babaei, P. (2020). Artificial players in the design process: Developing an automated testing tool for game level and world design. In *Proceedings of the Annual Symposium on Computer-Human Interaction in Play*, pages 267–280.

Statista (2022). Value of the global video games market from 2012 to 2021. https://www.statista.com/statistics/246888/value-of-the-global-video-game-market/.

Strehl, A. L. and Littman, M. L. (2008). An analysis of model-based interval estimation for markov decision processes. *Journal of Computer and System Sciences*, 74(8):1309–1331.

Sun, Y., Gomez, F., and Schmidhuber, J. (2011). Planning to be surprised: Optimal bayesian exploration in dynamic environments. In *International conference on artificial general intelligence*. Springer.

Sutton, S., R., Barto, and G., A. (1998). *Introduction to Reinforcement Learning*. MIT Press, 1st edition.

Syed, U. and Schapire, R. E. (2008). A game-theoretic approach to apprenticeship learning. In *Advances in Neural Information Processing Systems*.

Todorov, E., Erez, T., and Tassa, Y. (2012). Mujoco: A physics engine for model-based control. In *International Conference on Intelligent Robots and Systems*.

Torabi, F., Warnell, G., and Stone, P. (2018). Behavioral cloning from observation. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, pages 4950–4957.

Tucker, A., Gleave, A., and Russell, S. (2018). Inverse reinforcement learning for video games. In *Proceedings of NIPS Workshop on Deep Reinforcement Learning*.

Unity (2019). Unity game engine. Web page. Accessed: January 18, 2019.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In *31st Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS)*, pages 5998–6008.

Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., et al. (2019). Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 575(7782):350–354.

Vithayathil Varghese, N. and Mahmoud, Q. H. (2020). A survey of multi-task deep reinforcement learning. *Electronics*, 9(9):1363.

Wallner, G. and Kriglstein, S. (2013). Visualization-based analysis of gameplay data–a review of literature. *Entertainment Computing*, 4(3):143–155.

Watkins, C. J. and Dayan, P. (1992). Q-learning. *Machine learning*, 8(3):279–292.

Wiering, M. A. and Van Hasselt, H. (2008). Ensemble algorithms in reinforcement learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 38(4):930–936.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256.

Wurman, P. R., Barrett, S., Kawamoto, K., MacGlashan, J., Subramanian, K., Walsh, T. J., Capobianco, R., Devlic, A., Eckert, F., Fuchs, F., et al. (2022). Outracing champion Gran Turismo drivers with deep reinforcement learning. *Nature*, 602(7896):223–228.

Xiao, G., Southey, F., Holte, R. C., and Wilkinson, D. (2005). Software testing by active learning for commercial games. In *AAAI*, pages 898–903.

Xu, B., Hu, X., Tang, X., Lin, X., Li, H., Rathod, D., and Filipi, Z. (2020). Ensemble reinforcement learning-based supervisory control of hybrid electric vehicle for fuel economy improvement. *IEEE Transactions on Transportation Electrification*, 6(2):717–727.

Yaeger, L. et al. (1994). Computational genetics, physiology, metabolism, neural systems, learning, vision, and behavior or poly world: Life in a new context. In *SANTA FE INSTITUTE STUDIES IN THE SCIENCES OF COMPLEXITY-PROCEEDINGS VOLUME-*, volume 17, pages 263–263. ADDISON-WESLEY PUBLISHING CO.

Yannakakis, G. N. and Togelius, J. (2018). *Artificial intelligence and games*, volume 2. Springer.

Zambaldi, V., Raposo, D., Santoro, A., Bapst, V., Li, Y., Babuschkin, I., Tuyls, K., Reichert, D., Lillicrap, T., Lockhart, E., et al. (2018). Deep reinforcement learning with relational inductive biases. In *International Conference on Learning Representations*.

Zhang, B., Rajan, R., Pineda, L., Lambert, N., Biedenkapp, A., Chua, K., Hutter, F., and Calandra, R. (2021). On the importance of hyperparameter optimization for model-based reinforcement learning. In *International Conference on Artificial Intelligence and Statistics*, pages 4015–4023. PMLR.

Zhang, Q. and Couloigner, I. (2005). A new and efficient K-Medoid algorithm for spatial clustering. In *International conference on computational science and its applications*, pages 181–189.

Zhao, Y., Borovikov, I., Silva, F. D. M., Beirami, A., Rupert, J., Somers, C., Harder, J., Kolen, J., Pinto, J., Pourabolghasem, R., et al. (2020). Winning isn't everything: Enhancing game development with intelligent agents. *IEEE Transactions on Games*.

Zheng, Y., Xie, X., Su, T., Ma, L., Hao, J., Meng, Z., Liu, Y., Shen, R., Chen, Y., and Fan, C. (2019). Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering*. IEEE.

Ziebart, B. D., Maas, A., Bagnell, J. A., and Dey, A. K. (2008). Maximum entropy inverse reinforcement learning. In *National Conference on Artificial intelligence*.