Cooperative Reinforcement Learning for Procedural Content Generation: A Super Mario Bros Study

Student Name: Alejandro García del Valle
Supervisor Name: Tom Friedetzky
Submitted as part of the degree of MSci Natural Sciences to the
Board of Examiners in the Department of Computer Sciences, Durham University

Abstract—Procedural content generation (PCG) is the process through which game content is generated algorithmically. While research on machine learning and deep learning methods for PCG has exploded in recent years, reinforcement learning (RL) to generate game content remained vastly under-explored until recently. PCG via RL (PCGRL) is a novel development in the field with numerous advantages. This project aims to implement a new PCGRL framework for Super Mario Bros. and explore multiple solver paradigms and their effects on generators. We explore cooperative solvers, which learn from the content created by the generator using RL, and static solvers. Furthermore, we research the effects of solver populations and analyse the performance of generators trained with this solver paradigm. Despite multiple limitations, our results show that generators trained with cooperative solvers are capable of adapting to the skill of the solver and create more challenging levels on average. However, we are unable to determine the impact that a solver population has on the generator.

Index Terms—online level generation, platform games, procedural content generation, reinforcement learning

1 Introduction

The development of a video game is an intricate and arduous process that involves numerous phases [1]. The process begins with concept development and pre-production, culminating with the release to manufacturing. Each subphase during development may incorporate vastly different disciplines; for example, during the art production plan, visual artists create concept art and set the artistic tone of the game, while during prototype development, programmers create the code that will run a basic version of the game [1]. During the last twenty years, games have become more sophisticated and expansive, and this rising complexity combined with a convoluted production process has led to increasing costs [2]. As a result, game publishers employ different strategies to manage costs. A common approach to reducing expenses is to automate as much of the development process as possible.

Procedural content generation for games is the process through which content, such as levels and characters, is generated algorithmically in an autonomous manner or with limited user input [3]. It is used commercially to increase game replayability and reduce development time and resources. It can also decrease storage requirements as content does not need to be included with a copy of the game; instead, it can be generated during gameplay. An example is *Minecraft* where PCG is used to generate a new map every time the player starts a new game. However, despite widespread use, the PCG methods utilised in the video game industry lack generality. They typically rely on hand-crafted heuristic algorithms designed for a specific game or series of games [4]. Consequently, they cannot be used to generate content for any other games.

While procedural content generation has been used commercially for decades, academic research has only emerged over the last two decades. The Super Mario Bros. (SMB) Level Generation Competition [5] in 2010 was the first academic PCG competition. Togelius et al. [6] produced the first survey of PCG research in 2011, and the field has significantly developed since then. There have been numerous attempts at creating systems that generate entire games autonomously or semi-autonomously, for example, in [7]. However, most PCG research typically focuses on one or two components of the game development process, such as terrain generation. This focus is due to the difficulty of the task: developing a game involves the design and creation of many different elements that work together to produce the desired gameplay experience. Additionally, game content such as levels and rule sets are constrained by playability requirements, complicating the generation process further. We refer to the type of content with specific functionality requirements as functional content.

Over the past few years, the focus on PCG has shifted away from search-based methods [6] to machine learning (ML) and deep learning (DL) approaches [4], [8], [9], [10]. A wide array of ML and DL techniques have been explored and tested for PCG; however, reinforcement learning has remained under-explored in the field until recently. Khalifa et al. [11] are the first to generate game levels through reinforcement learning. They propose a new framework, PCG via reinforcement learning, where an agent, referred to as the generator, is trained to create content using RL. The validity of this approach is proved empirically: they test the framework on three different games and find that

it can generate a high degree of playable levels. Gisslén et al. [12] adapt this framework and introduce adversarial reinforcement learning for PCG (ARLPCG). This framework consists of a generator and an agent that plays the created content, denominated the solver. The generator and the solver compete and learn from each in an adversarial manner. There exists preceding research that explores the concept of a generator and a solver that play against each other; however, Gisslén et al. are the first to use RL to train both agents.

1.1 Motivation

With increasing development costs, rising competition, and a growing market, procedural content generation has the potential to revolutionise video game development [2], [13]. When combined with other AI advancements, PCG methods could reduce costs by automating parts of the development process such as designing textures, creating non-player characters, and even generating entire stories [14]. Additionally, PCG could be used as part of mixed-initiative approaches where a human designer cooperates with an AI to generate game content, aiding development [15]. PCG techniques can also assist game content evaluation by determining whether created content meets functionality requirements and any other criteria defined by the developers [4]. However, despite its potential and the promising developments in current PCG research, there is a wide gap between the methods used in the industry and the methods explored in the academic community [16]. While researchers are working to bridge this gap, state-of-the-art techniques are still too unpredictable and more research is required before they are used commercially [16], [17].

PCG research could also transform how players experience games. PCG methods are commonly used during gameplay to improve replayability by algorithmically generating novel content. In recent years, researchers have been exploring techniques such as experience experience-driven PCG (EDPCG) [18] where the system develops a model of the user experience and uses it to generate content suited to the preferences and skills of the player. This research direction could revolutionise the player experience by providing each user with a personalised experience.

Additionally, advancements in PCG could have benefits beyond the gaming industry. Procedurally generated content could also be used to increase the generality of ML methods [8]. A common problem when training ML models is overfitting. The model learns a particular policy that performs well during training but fails to generalise to unseen environments. This problem can be addressed by increasing the training dataset size; however, real-world data is not readily available in every domain. Researchers could mitigate this issue using PCG methods to generate additional content for training. For example, Justesen et al. [19] show that procedurally generated game levels can improve the generalisation ability of deep reinforcement learning agents when playing certain games. As machine learning becomes increasingly prevalent, PCG techniques could be crucial in preventing overfitting and making ML models more robust.

Procedural content generation via reinforcement learning is a novel framework that could profoundly impact the field. An advantage of RL over other ML methods is that it does not require any training data. Instead, an RL agent learns by interacting with the training environment. Game content such as levels and maps is often scarce; therefore, RL is well suited for PCG methods. Additionally, PCGRL is particularly appropriate for mixed-initiative and experience-driven approaches as it can adapt to the needs of the user. Shu et al. [20] implement MarioPuzzle, an experience-driven PCGRL (EDRL) framework for SMB. They demonstrate that their framework is capable of adapting to the experience of the player.

Furthermore, the framework proposed in [11] can be generalised to any game with the appropriate reward function. Gisslén et al. [12] demonstrate that PCGRL combined with an adversarial RL solver can generate game levels that are both playable and challenging. While these are encouraging results, the technique is still in its infancy, and more research is required to prove its validity.

In this project, we explore PCGRL for SMB levels and build a framework in Python for this purpose. We choose to test our framework on this game as it has been a standard benchmark in PCG research since the inception of the field. We denominate our framework Cooperative Reinforcement Learning for PCG (CRLPCG). It is inspired by ARLPCG [12]; however, in this project, we focus on cooperative RL solvers that work with the generator to create playable levels. The nature of the relationship between generator and solver is symbiotic rather than adversarial. While this concept has been explored for SMB before, the use of an RL solver has never been implemented effectively [21].

Shu et al. [20] implement MarioPuzzle, a PCGRL framework for SMB that uses an A* agent as a solver. This implementation relies on an updated version of the Mario AI Framework, written in Java, as a platform to run the game and test generated levels. We explored different approaches to augment this framework and replace the A* solver with an RL agent. However, most available deep learning and reinforcement learning libraries are written in Python; for example, OpenAI Gym, the dominant library in RL research, is written in Python. We attempted training a solver in the Gym environment for SMB and loading the trained model in Java using the Deep Java Library (DJL) [22]. Despite its benefits, DJL only allows for model inference, and therefore, we would be unable to update our model as the solver plays generated levels. For these reasons, we conclude that a new Python framework is required to achieve our research purposes.

1.2 Objectives

The project explores the following research question: *how can we employ cooperative reinforcement learning for procedural content generation in Super Mario Bros?* We list a set of objectives to address the proposed research question:

- **Minimum**. Create an environment to train an agent to play SMB levels. The environment must be capable of rendering levels generated through PCG.
- **Minimum**. Create an environment to train an agent to generate SMB levels. The environment must include a solver agent to test level playability.

- **Intermediate**. Combine the previous environments to create the CRLPCG framework for level generation in SMB.
- **Intermediate**. Train a generator agent using a cooperative solver and compare its performance to a baseline implementation.
- **Advanced**. Explore different methods to improve and augment the CRLPCG framework.
- **Advanced**. Critically evaluate and compare the results of the different approaches employed.

2 Related Work

The field of procedural content generation for digital and non-digital games has seen increasing popularity over the past two decades. Togelius et al. [6] published the first survey of PCG in 2011, with an explicit focus on search-based methods; two years later, Hendrikx et al. [14] published a comprehensive survey of PCG for games and created a taxonomy of the entire field for the first time. These papers identified a wide range of promising research areas and have influenced subsequent work in the field. In this section, we examine and summarise relevant PCG research, current approaches to content generation, and the state-of-the-art methods pertinent to our project.

PCG methods are not limited to one specific game content type; however, researchers in the field typically focus on functional content, specifically on game levels [4], [9], [10], [23]. The scarcity of work on PCG to generate non-functional content may be explained by the vast amount of existing research in other fields that indirectly address the topic. For example, generating cosmetic content like sprite sheets falls under the scope of image generation research. Furthermore, levels and maps act as the primary form of interaction between the game and the player [23]. Therefore, the focus on level and map generation is understandable as they are an essential component in the majority of games.

There are various PCG approaches to level and map generation. However, we can divide them into two categories: autonomous generation, where the agent has complete control over the design process, and mixed-initiative approaches, where content creation is done by both a human designer and an AI designer [4], [10], [23]. Most research in the field, including the framework proposed in this project, focuses on autonomous generation as this is the primary role PCG performs in games [23].

In the past ten years, developments in the field of machine learning have impacted PCG methods [4], [8], [9], [10]. Procedural content generation via machine learning (PCGML) refers to the generation of game content using ML models [4]. These models are trained on existing content, usually human-designed. While PCGML methods have shown remarkable results, it is essential to note that ML models are not guaranteed to generate content that meets functionality requirements. For example, a Generative Adversarial Network (GAN) [24] trained on the original SMB levels can generate content that aesthetically resembles the training data; this artificial content, however, may not be playable. Furthermore, ML models usually require a large amount of training data to produce satisfactory results. Machine learning methods for PCG are limited as large training datasets are not readily available for every game.

The issues mentioned above are at the core of current research in PCGML. It is crucial to evaluate generated content to determine its quality and whether it is functional. This task poses a challenge as game content evaluation involves a high degree of subjectivity, and there is no generalisable approach that applies to every game. The only metric we can comprehensively use to assess game content is playability. Researchers typically use an AI agent, a solver, that plays generated content to determine whether it is playable. The framework proposed in [25] is a recent example of this technique. Generative Playing Networks (GPNs) [25] consist of two models: a generator function and a solver. The generator model produces random latent variables, which are then mapped to a game level. The solver plays the generated content and learns using RL. The generator then uses the feedback from the solver to learn how to create playable levels. This process ensures that generated content meets basic functionality requirements. For this reason, using an ML model as a solver has become a widespread paradigm in the field [4], [8], [9], [10].

While most ML methods require training data to learn, reinforcement learning techniques do not have this constraint. Below we will explore and analyse the current state-of-the-art in PCGRL.

2.1 Procedural Content Generation via Reinforcement Learning

Reinforcement learning is often used to teach agents how to play games. Two well-known examples are AlphaGo [26] and AlphaGo Zero [27]. AlphaGo was trained to play Go using different techniques, including RL. It became the first program to beat a world champion. AlphaGo was succeeded by AlphaGo Zero, which was trained solely using RL and managed to beat its predecessor 100-0. Despite its success in playing games, using RL to generate game content has not been the subject of academic research until recently. Shaker [28] proposes the use of intrinsically motivated reinforcement learning (IMRL) [29] for PCG. Shaker argues that IMRL can help generate diverse and interesting content, be used as part of mixed-initiative tools, and generate content tailored to the player. Chen et al. [30] implement the first framework that uses RL to generate game content: a deck-recommendation system for collectable card games that uses Q-learning [31] to learn a deck search policy. They show that their framework could build winning-effective decks using less computational resources than search-based methods.

Procedural content generation via reinforcement learning (PCGRL) for autonomous generation of game levels was first explored in [11]. Khalifa et al. frame level generation as an iterative improvement task. Instead of generating the whole level at once, the agent observes the current state and takes an action, i.e. it makes a change in the level. A reward and a new observation follow this action, thus framing the problem of content generation as a Markov Decision Process (MDP) [32], which we can attempt to solve using RL. The model in PCGRL does not search content space to find the most suitable content, as search-based methods do [6]. Instead, the model searches for a policy

to generate the content that returns the highest expected cumulative reward. Therefore, the model searches the content generator space rather than the content space itself. This concept was first proposed in [33]; however, this work used interactive evolution, making it less scalable than PCGRL. The framework is divided into three key components: a problem module, a representation module, and the change percentage.

Khalifa et al. [11] test their system on three two-dimensional games: *Binary*, *Zelda*, and *Sokoban*. For each game, multiple agents are trained using the different representation schemes proposed in [34]. Proximal Policy Optimisation (PPO) [35] is used to train these agents. The results show that PCGRL can generate a high percentage of solvable levels for two of the games tested, but it struggles with the game *Sokoban*. The researchers argue that this could be due to the simple *Sokoban* solver used. The solver is used to determine level playability, and they believe that a more powerful agent could address the issue. Despite the high success rate generating playable levels for *Binary* and *Zelda*, the researchers observed that the agent also struggles to create challenging levels. The games used to test the framework had simple goals, such as making sure that a level has a certain number of objects or that there exists a path between two points on the map. The agents are rewarded for creating levels that meet minimum playability requirements and are not encouraged to generate difficult levels.

Additionally, the reward functions used are specific to each game. Testing the framework on a new game would require designing a new reward function, limiting the potential use of the framework for games where levels are complicated and determining playability is hard. Regardless of these drawbacks, the proposed method has numerous advantages: once the agent is trained, it can generate levels faster than other PCG methods; it does not require training data as the agent learns by interacting with the environment; and it is well-suited to mixed-initiative methods, as suggested in [28].

2.2 Adversarial Reinforcement Learning for Procedural Content Generation

Gisslén et al. [12] further explore PCGRL for game levels. They propose adversarial reinforcement learning for procedural content generation. The architecture is similar to a GPN: it consists of a generator agent and a solver agent. The difference is that both agents are trained using RL, specifically PPO. The two agents have an adversarial relationship. The solver agent learns to play generated levels which teaches the generator how to create more challenging levels. Note that the generator uses the turtle representation scheme proposed in [34] and explored in [11]. With this representation scheme, the generator observes and changes a local segment in the level.

This framework addresses one of the problems with the previous work on PCGRL: generating challenging levels that are playable. The RL solver encourages the generator to create more difficult content that is also playable. Furthermore, auxiliary inputs in the reward function of the generator are also used to control the difficulty of the levels and the behaviour of the agent indirectly. The reward function has two components: an internal reward, which comes from the actions of the generator, and an external reward, which depends on the performance of the solver. The auxiliary input is used to control the external component of the reward function by randomly generating a value $\lambda \in [-1, 1]$ and using it as a scaling factor in the function.

Their data shows that higher auxiliary values correspond to easier levels, while lower auxiliary inputs result in more difficult ones. Furthermore, the results show that auxiliary inputs can also change the style of the generated level by rewarding the generator for certain behaviours in the solver agent. The paper also demonstrates that an agent trained as an adversarial solver generalises better to unseen content than agents trained on rule-based PCG or ARLPCG with a fixed auxiliary input. However, a potential limitation of this model is that, due to the nature of RL agents, the generator could be learning to exploit certain behaviours in the solver. Therefore, the output of the generator may exhibit characteristics meant to abuse the specific behaviour of the solver, leading to less diverse generated levels. Additionally, the framework is only tested with two simple games; and the representation scheme that the generator uses limits it to only being able to make a few changes at a specific location in each time step. These limitations make it harder to determine how the framework would perform with a more complex game.

Engelsvoll et al. [21] explore PCGRL with an RL solver for SMB. The generator and solver agents are trained using Deep Q-Networks (DQNs) [36]. The researchers experiment with two approaches to level generation: Seq2Tile, where the generator observes a sequence of tiles and generates Q-values for a single tile; and Seq2Seq, where the generator receives the same observation but generates Q-values for a sequence of tiles rather than an individual tile. The highest Q-value is used to choose the tile or sequence of tiles placed in the level. The results show that the proposed framework can generate playable levels using both methods, although Seq2Tile runs considerably slower due to its nature. Due to time and resource limitations, the researchers cannot determine whether the generator can adapt the level of difficulty to match the skill of the solver. Therefore, while the proposed method is of interest, we cannot draw any decisive conclusions from this work.

2.3 Experience-Driven Procedural Content Generation via Reinforcement Learning

Shu et al. [20] implement MarioPuzzle, a framework that combines PCGRL and experience-driven PCG [18] to generate SMB levels segment by segment. The architecture is similar to ARLPCG: there is a generator agent trained using RL and a solver agent, and the reward function of the generator partially depends on the performance of the solver. However, the solver does not use RL; it is an A* agent created for the 2009 Mario AI Competition [37]. Additionally, unlike in previous implementations of PCGRL, the RL agent does not generate the content directly. Instead, it generates a latent space vector that is then transformed into a level segment by MarioGAN [38], a Generative Adversarial Network (GAN) [24] trained on SMB levels. Therefore,

during training, the RL generator is not learning to generate content independently; it is learning to sample the latent space of the trained GAN. The generated segment is then repaired by an evolutionary model assisted by a trained multilayer perceptron (MLP) [39]. The solver agent finally tests the repaired segment to determine playability.

Shu et al. explore various metrics to reward and evaluate the RL agent: novelty, diversity, and playability. They find that using a normalised combination of these three metrics as a reward function results in the best performance and generates a high percentage of diverse playable levels. Additionally, the researchers demonstrate that the framework can generate content quickly enough so that a human player could play generated segments while more content is being created. This approach to PCG is often referred to as online content generation, in contrast to offline content generation, where game levels would be entirely generated before they are played. They argue that their method could be used for online generation in any game where levels can be segmented and a learned game content representation exists.

Despite the discussed benefits, it is necessary to highlight the limitations of the framework. The researchers note that the use of a predictable A* agent to measure playability limits the ability of the generator to create challenging and exciting levels as the generator learns to maximise the playability component of the reward function. Furthermore, the researchers use the diversity metric as a proxy for fun. They justify this decision using Koster's *theory of fun* [40]. Koster explores the link between learning and fun and suggests that games are fun when the perceived patterns in the game are neither too unfamiliar nor too familiar. Whether diversity in SMB levels is an appropriate measure of fun is debatable; it is hard to quantify what makes a Mario level enjoyable due to the subjective nature of the issue.

2.4 Summary and Proposition

The use of reinforcement learning as a method for PCG is a recent development. The work discussed in the previous sections comprises the current state-of-the-art methods in PCGRL for game levels. Khalifa et al. [11] demonstrate that PCGRL is capable of generating playable levels but also that it may struggle to create challenging levels if the solver agent is not powerful enough. Gisslén et al. improve PCGRL using an RL solver agent. The interactions between the solver and the generator cause them to become better at their task. Due to these interactions, the generator can create challenging levels at the end of the training process. However, the researchers test the ARLPCG framework on simple games. Therefore, more research on the use of RL solvers in PCGRL is essential to demonstrate the validity of the concept.

Furthermore, as discussed previously, the scarcity of extensive datasets for game content is a significant limiting factor of ML methods in PCG. The PCGRL framework addresses this issue as RL does not require training data. We believe that this framework addresses one of the core problems in PCGML, and therefore more research on the topic is crucial. Moreover, state-of-the-art methods in PCG are yet to be used commercially, to the best of our knowledge. Therefore, further research on PCGRL increases the possibility of integrating the method into the video game industry.

In this project, we combine the frameworks and ideas proposed in [12], [20] and introduce the CRLPCG framework for SMB. We explore using an RL cooperative solver agent with an RL generator to create SMB levels. Our framework is inspired by the MarioPuzzle framework [20]. We adapt some aspects of this framework for our research aims. Finally, we evaluate our framework using the playability, diversity, and novelty metrics proposed in [20].

To further explore PCGRL and the current state-of-the-art techniques, we experiment with populations of solvers. An RL generator learning to exploit observed behaviours in the solver is a recurring problem in PCGRL research. The framework proposed in [12] addresses this issue using an RL solver. However, Gisslén et al. argue that this phenomenon may still occur, although to a lesser extent. They suggest using a population of solvers to mitigate this issue further and improve the diversity of the generated content, as the generator may find it more difficult to discover exploits in the behaviour of multiple solver agents. Therefore, we decide to explore the use of solver populations and their effects on the generated content.

3 Methodology

In this section, we explain our methodology in detail. We create two different environments, *Mario Play* and *Mario PCG*, to achieve the objectives of the project. Our solution incorporates various tools, methods, and frameworks, explained in the following subsections.

3.1 Implementation Tools

OpenAI Gym [41] is an open-source Python library for reinforcement learning. It provides a wide range of standard environments to test and benchmark new RL methods and a simple interface that facilitates the implementation process. It also provides the necessary building blocks to create custom environments and efficiently integrate them with the library. Over the last few years, Gym has become the standard in the field and has made it possible to reproduce existing RL research. For these reasons, we use this library to create *Mario Play* and *Mario PCG*. Furthermore, using the Gym *Env* class to create our environments allows us to train RL agents with the Stable Baselines 3 (SB3) library [42]. The SB3 library contains multiple PyTorch implementations of RL algorithms that work with the Gym library to facilitate the training process.

The environments *Mario Play* and *Mario PCG* rely on a Python implementation of SMB to render the game. While there are multiple versions of SMB in Python, including the SMB Gym environment, we require an implementation that can render levels generated using PCG. The Gym environment for SMB only includes the original levels. It does not make it possible to render custom levels which is a requirement for our project. Therefore, we choose to use the Python SMB implementation in [43]. This program takes a JSON representation of an SMB level and makes it playable using Pygame, a Python library for game development [44]. We can render and test generated levels by saving them in

the appropriate format. We further adapt this framework by creating two new classes, *AI_Input* and *AI_Mario*, which allow our environments and the RL agents to communicate with this SMB implementation.

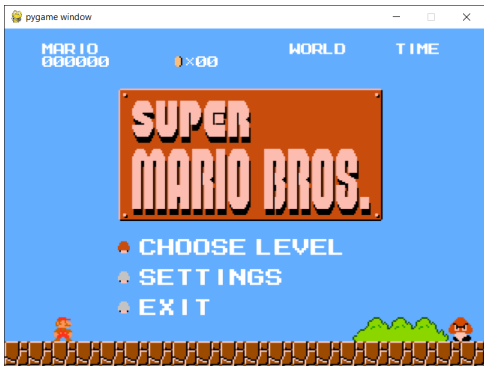

Fig. 1: The Super Mario Python framework in Pygame [43].

.

3.2 Reinforcement Learning Methods

Reinforcement learning is one of the main paradigms in machine learning, alongside supervised and unsupervised learning [45]. It focuses on learning through trial-and-error and delayed reward. There are three basic concepts at the core of RL: an environment with a particular set of rules, an agent that observes and interacts with the environment, and a reward signal that indicates progress towards a predetermined objective [45], [46]. Problems in RL are generally formalised as Markov Decision Processes (MDPs), where the agent must repeatedly choose the best action to take based on the current state of the environment. At each time step t, the agent receives an observation o_t of the current state s_t of the environment. The agent then takes action a_t, and the environment transitions to the next state s_{t+1}. The action is chosen according to a policy π, which maps s_t to a_t. Based on s_t, a_t, and s_{t+1}, the agent receives a reward r_{t+1} and a new observation o_{t+1}. This feedback from the environment is used to update our policy. The goal in RL is to find the optimal policy π^* that will maximise expected cumulative reward, referred to as expected return.

We use Proximal Policy Optimisation to train our RL agents [35]. We choose this method for multiple reasons. Firstly, the researchers in [11], [12], [20] use PPO to train their RL agents as well. While they do not justify the reasoning behind their choice, we believe that this technique is selected because it is simple to implement and understand. Nevertheless, its performance is close if not comparable to state-of-the-art RL methods. Furthermore, by choosing PPO, we can better evaluate our results against the previous work on PCGRL. Finally, PPO-Clip, a variant of PPO, is included in the SB3 library. We can use this implementation to facilitate the training process.

PPO belongs to a branch of RL referred to as policy optimisation methods. In order to understand policy optimisation and PPO, we first need to define value and advantage functions [45], [46]. A value function estimates the expected return if we start in a state s or a state-action pair (s, a) and act according to a given policy π. The advantage function indicates how much better it is to take action a in state s over randomly sampling an action according to a given policy π. The advantage function is defined as the difference between the action-value value function $Q^\pi(s, a)$ and the value function $V^\pi(s)$.

Policy optimisation methods optimise the parameters θ of the model policy π to maximise expected return. The optimisation process is typically done through gradient ascent. However, gradient ascent might cause a change in the parameters that is too large, leading to worse performance. This idea is what motivates PPO-Clip. We calculate the direction of gradient ascent using the advantage function $A^{\pi_\theta}(s, a)$. However, to avoid taking a too large step, we clip the size of the update using a hyperparameter ϵ. This technique ensures that policy updates are reasonable.

3.3 Mario Play Environment

We implement *Mario Play* using the Gym *Env* class to train our solver model. The environment uses the Super Mario Python [43] framework to render levels for the reasons discussed previously. We use Pygame to capture the RGB image from the game window as an integer array. We then employ the *transforms* module from the Torchvision framework [47] to convert the image array to grayscale and downsample it to a width and height of 84×84. We do this to simplify and remove unnecessary detail from the environment observations, which reduces the number of trainable parameters in our *Mario Play* model. We define the observation space of the environment as a $(1, 84, 84)$-dimensional box using the *Box* class from Gym [41].

In the Super Mario Python framework, the player has ten different actions they can take. Therefore, we define the action space of the environment using the *Discrete* class from Gym. Our model produces a number from 0 to 9. Each number represents a different combination of the form (a_1, a_2, a_3), where $a_1 \in \{\text{stay, move right, move left}\}$, $a_2 \in \{\text{no jump, jump}\}$, and $a_3 \in \{\text{no boost, boost}\}$. We exclude the combinations where the agent does not move and boosts as this is equivalent to not moving and the combination where the agent jumps up and boosts as this is equivalent to jumping up and not boosting. The output from our model is received and processed by the Super Mario Python framework through the *AI_Input* class.

It is important to remember that the goal of this environment is to train an agent that can act as a solver in our other environment, *Mario PCG*. We need the solver to determine the playability of levels generated through PCG. We consider a level segment playable if the agent can reach the rightmost part of the segment, as in [20]. Therefore, the reward function in *Mario Play* should reward the agent for moving right and completing the level. We adapt the reward function from the Gym SMB environment [48] for our purposes and define the following function:

$$reward = \begin{cases} x_t - x_{t-1} & \text{if Mario has not won or lost,} \\ 100 & \text{if Mario has won,} \\ -100 & \text{if Mario has lost,} \end{cases}$$

,

where x_t is the x-coordinate of Mario at time step t.

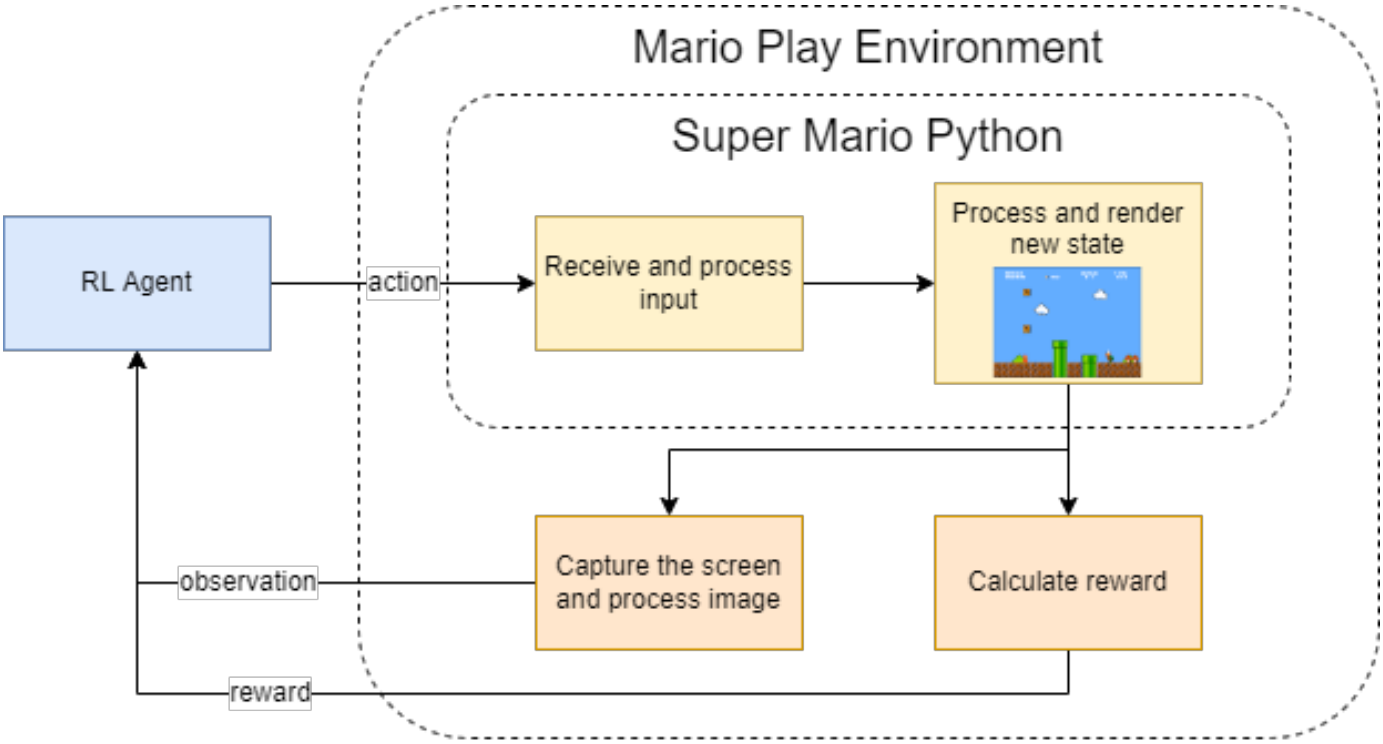

Fig. 2: System architecture for the Mario Play environment.

3.4 MarioPuzzle

We considered expanding MarioPuzzle [20] for our research purposes, but we decided to implement a new system due to the limitations of DJL, as discussed previously. Despite this drawback, the novel framework developed by Shu et al. combines multiple state-of-the-art methods and delivers impressive results. Inspired by this work, we adapt their architecture and utilise multiple components employed in MarioPuzzle, namely MarioGAN, the CNet-assisted evolutionary repairer, and the reward functions. We explain the reward functions from MarioPuzzle below and MarioGAN and the repairer in the following two sections.

As discussed previously, Shu et al. use level diversity as a metric for fun. It is measured by calculating tile-based Kullback–Leibler divergence [49] between the current segment and recently generated tiles, shown in Equation 1 in [20]. They use this function to calculate the average diversity and standard deviation in human-designed levels. The RL agent is rewarded for generating segments whose diversity is within the pre-determined ranges. The reward function for this metric is shown in Equation 2 in [20]. In this paper, we refer to this metric as diversity.

The second metric, novelty, measures historical deviation. It uses tile-based KL-divergence to calculate the average diversity between the current segment and the most similar segments generated recently, as shown in Equation 3 in [20]. While this metric and the fun metric might seem similar, they serve different purposes: the fun metric rewards the agent for keeping diversity in a particular range and historical deviation rewards the agent for generating diverse levels.

The final metric is playability. Shu et al. calculate playability using the A* agent from the Mario AI framework [50], [51]. The generated segment s is concatenated with the three previous segments. They consider a segment playable if the solver reaches the right-most end of s. The reward is the number of playable segments that the generator creates in the current episode, encouraging the generation of playable levels. We replace the A* agent with an RL solver in this project.

Shu et al. demonstrate that using a normalised combination of the three metrics as a reward function causes the RL generator to create the highest percentage of playable levels. Therefore, we use the same function to reward our generator agent.

3.5 MarioGAN

In a Generative Adversarial Network [24], we have a generator that creates content and a discriminator to classify data as real or fake. These two components compete; the generator attempts to produce data capable of fooling the discriminator, while the discriminator attempts to distinguish between increasingly realistic data. When training is complete, the generator can produce synthetic data following an identical distribution to members of the original dataset.

Volz et al. [38] train a DCGAN [52] using the Wasserstein GAN [53] algorithm to generate SMB levels. The generator is trained using existing SMB levels. It learns to map a Gaussian noise vector $v \in [-1, 1]^{32}$ to an integer array representation of a level. Subsequently, the discriminator is trained to distinguish between the original SMB levels and the generated content. This adversarial process is repeated multiple times. When training is complete, the DCGAN

model is capable of generating realistic levels, albeit with a limited amount of errors.

3.6 CNet-Assisted Evolutionary Level Repairer

Shu et al. [39] create a system to detect and repair defects in SMB levels generated through PCG. They train an MLP model, denominated CNet, to detect whether an error has occurred and where it has occurred. The model examines every tile in the generated level individually, accepting as input a y-coordinate and the surrounding tiles encoded as one-hot vectors. The output of the model is a probability distribution over the different tile types. The researchers find the CNet performs well detecting illegal tiles and recommending alternative legal tiles on levels from the Mario AI Framework [50]; however, its performance worsens when applied to levels generated by MarioGAN. They argue that when MarioGAN generates an erroneous tile, it is likely to generate surrounding tiles that are also illegal. This clustering of illegal tiles impairs the ability of the model to detect an error and recommend an alternative tile. The researchers combine the CNet with a genetic algorithm that searches for the optimal replacement scheme to address this issue.

3.7 Mario PCG Environment

We structure the *Mario PCG* environment similarly to MarioPuzzle. However, we use the *Mario Play* environment to render generated content and an RL agent as a solver. The architecture of the environment is shown in Figure 3. We define the observation and action spaces of the environment as 32-dimensional vectors, as this is the shape of the latent space of MarioGAN. The generator observes the last segment placed as a latent space vector. Subsequently, it chooses which segment to place next and receives a reward from the environment accordingly, thus framing the task as an MDP. The training procedure is presented in Algorithm 1.

Unlike in MarioPuzzle, we calculate playability using an RL solver. We require a trained solver before training a generator agent in the *Mario PCG* environment. We initially considered using an untrained agent as a solver; it could play generated segments and learn at the same time as the generator. However, we decided against this approach for various reasons. First, we believe that an untrained solver could negatively affect the learning process of the generator. As discussed previously, one of the components in the reward function of the *Mario PCG* environment is playability. An untrained solver does not know how to complete SMB levels. Therefore, initially, it would be incapable of determining whether a generated segment is playable. The ineffectiveness of the solver would hinder the training process of the generator: it may be generating playable segments, but it would not receive the appropriate reward.

Furthermore, such a training procedure would be exceedingly time-consuming. From initial experimentation, we found that training a functional agent on the *Mario Play* environment using available hardware takes approximately twelve hours. Using an untrained agent as a solver would be a more protracted process as it would involve simultaneously training a solver and a generator. Therefore, given the time frame of this project, this approach is not suitable. For these reasons, we decide to train an agent in the *Mario Play* environment before using it as a solver in the *Mario PCG* environment.

Algorithm 1 Training procedure for the generator

Require: $\pi_G : [-1, 1]^{32} \rightarrow [-1, 1]^{32}$, RL generator
Require: $G : [-1, 1]^{32} \rightarrow segment$, MarioGAN
Require: $R : segment \rightarrow segment$, repairer
Require: π_S, RL solver
Require: E, SMB environment
Require: $P : (\pi_S, E, segment) \rightarrow \{False, True\}$, playability reward
Require: $N : (segment, segmentList) \rightarrow n \in [0, 1]$, novelty reward
Require: $D : (segment, segmentList) \rightarrow d \in [0, 1]$, diversity reward
Require: T, number of training steps for the generator
Require: T_S, number of training steps for the solver
Require: N, number of segments per episode

```
t ← 0
while t < T do
    done ← False
    n ← 0
    segments ← Empty List
    observation ← [0]^32
    while n < N and not done do
        t ← t + 1
        action ← π(observation)
        segment ← G(action)
        segment ← R(segment)
        playable ← P(π_S, E, segment)
        reward ← 0
        if not playable then
            if π_S is cooperative then
                Train π_S on segment
                playable ← P(π_S, E, segment)
                done ← not playable
            else
                done ← True
            end if
        end if
        if playable then
            reward ← 1 + N(segment, segments) + D(segment, segments)
            n ← n + 1
            Append segment to segments
        end if
        observation ← action
        Update π_G with (reward, observation)
    end while
end while
```

3.8 Experimental Procedures

We investigate two solver types, *static* and *cooperative*, to achieve our research purposes. We define a static solver as an agent that plays generated segments but does not change throughout the training process of the generator. When the agent encounters a level segment that it cannot complete, we

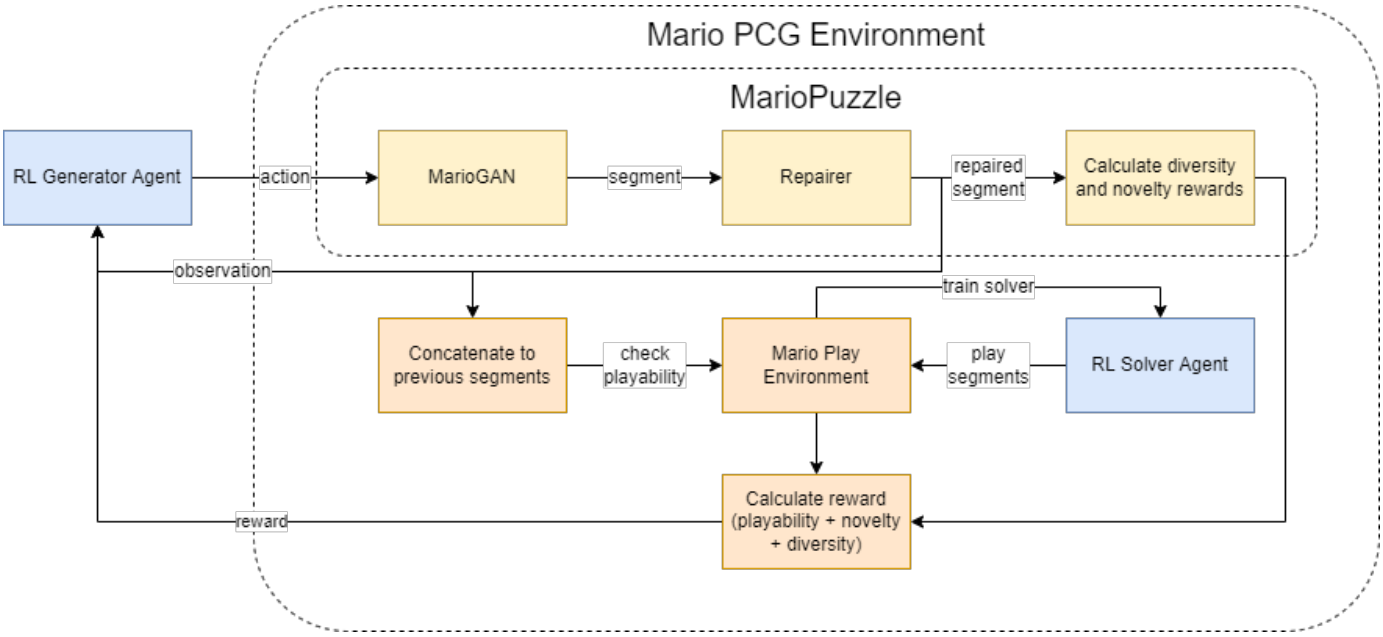

Fig. 3: System architecture for the Mario PCG environment.

consider the segment unplayable and conclude the current episode. In contrast, a cooperative solver plays generated content and learns throughout the training process. When it fails to complete a segment, we train the agent on the generated content for a short period. After the training is complete, the solver plays the segment again and determines its playability. This process is described in 1. We use the static as our baselines implementation for PCGRL. We perform multiple experiments using both solver types to explore and contrast the differences between PCGRL and CRLPCG.

In addition to experimenting with different solver types, we also explore how using a solver population during training, rather than an individual solver, affects the generator. We train different generators using an individual solver and a population of four solvers. We implement the solver population using the *SubprocVecEnv* class from SB3 [42] which allows us to create a vectorised environment consisting of four independent *Mario PCG* environments, each in its own process and with its own solver. We choose a population of four solvers as we are limited to four cores on the available hardware. We attempted to create larger solver populations, but this led to memory issues.

4 Mario Play Results

We train multiple agents in the *Mario Play* environment using the PPO implementation in SB3. Initially, we experimented with the SB3 implementations of DQNs [36] and Advantage Actor-Critic (A2C) [54] as well. However, we encountered memory problems whilst training the DQN, and we found that PPO performs significantly better than A2C. Due to the time limitations of the project, we use PPO to train our model in subsequent experiments. The model uses a Convolutional Neural Network to process the observations from *Mario Play* and produce a discrete output $a \in [0, 9]$.

We assess the performance of an agent by measuring the average win ratio and the average percentage of the level the agent traverses, which we refer to as the average level progress. We are interested in training an agent that can act as a solver in the *Mario PCG* environment, where it is supposed to determine whether a level segment is playable. Therefore, we consider these metrics, average win ratio and average progress, appropriate for the task as we require a solver capable of traversing and completing segments to determine playability correctly.

After initial experimentation, we find that an agent trained on the basic Mario Play environment does not perform as expected. Hence, we explore various strategies to improve our agent. First, we test the *SubprocVecEnv* and *VecFrameStack* classes from SB3. We use the *SubprocVecEnv* class as it allows us to train our agent in multiple environments simultaneously, which accelerates the training process. The class *VecFrameStack* stacks the last n frames and allows our agent to observe what is currently occurring in the environment and what occurred in the last $n - 1$ frames. We find a modest improvement in performance with these additions. However, the trained model learns to almost exclusively jump forward while boosting. It does not learn to avoid enemies or jump at appropriate times to avoid gaps. We experimented with different reward functions to encourage the agent to behave differently, resulting in similar or worse performance.

We believe that this lack of learning happens because when the agent chooses a suicidal action, it does not immediately receive punishment for its action. For instance, when the agent jumps into a gap, it does not receive punishment for the action until the environment detects that Mario has lost, which happens approximately 60 frames later. To address this issue, we implement a *Skip_Frame* wrapper class for *Mario Play*. It accepts an action from the model as input and takes this same step in the original environment m times. It returns an observation of the environment at

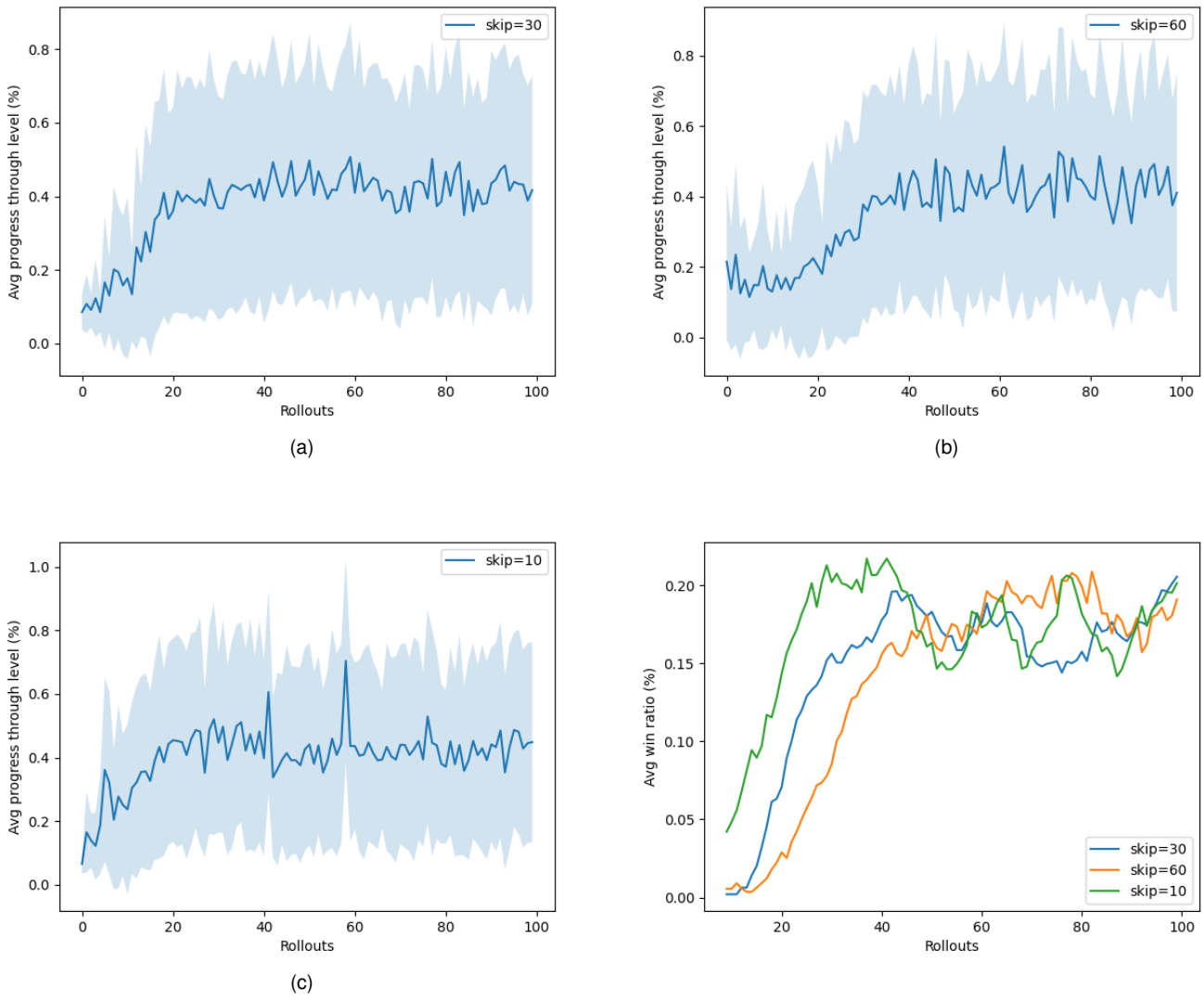

Fig. 4: Graphs 4a, 4b, and 4c show the average level progress an agent obtained during training when setting the *skip frame* variable to 60, 30, and 10, respectively. Graph 4d shows the average win ratio during training for these models.

the end of this process and hence skips m frames. We find a significant performance improvement when using this modification. Therefore, we experiment with different values for the *skip* variable. The results are shown in figure 4.

4.1 Training Results

We can see in 4c that the agent trained with $skip = 10$ seems to perform better initially; however, all agents converge to the same point. Furthermore, we observe that the agents still learn to jump forward while boosting as a general strategy. However, we notice that agents trained with a higher value for the *skip* variable seem to learn a more sophisticated strategy initially. For example, the agent with $skip = 60$ performs worse at the start of training, but it learns to stop and avoid enemies and gaps. Nonetheless, the model ultimately learns to behave similarly to the other agents. We believe that training this model for longer than we have and with different hyperparameters might improve performance; however, this is not possible due to time constraints.

4.2 Evaluation Results

We evaluate the trained agents on 12 of the original SMB levels. Additionally, we include in the evaluation a random agent, an agent that constantly jumps forward while boosting, and the *skip=10* model after receiving additional training as a cooperative solver. The results of the evaluation are shown in 1. We observe a high degree of variation in performance across levels, which explains the high value for the standard deviation of level progress. The agents perform well in certain levels, such as World 7 Level 1 and World 1 Level 2. However, for the other levels, the average level progress was generally between 10% and 40%. In addition, we observe that an agent that constantly jumps forward while boosting performs marginally worse than our trained

agents. Hence, we can deduce that learning occurred during training, albeit limitedly.

Despite similar performance across all experiments, the model with *skip=10* obtains a higher value for level progress and completes two levels. Therefore, we decide to utilise it as a solver in the *Mario PCG* environment.

TABLE 1: The table presents the evaluation results for the three agents trained on *Mario Play*. We test the agents on 12 of the original SMB levels and record the level progress as a percentage, with the corresponding standard deviation, and the number of levels the agent completes.

Agent Name	Average progress (%)	Levels completed
skip=10	39.40 ± 30.79	2
skip=30	37.15 ± 29.34	1
skip=60	31.32 ± 26.95	1
Forward-Jump-Boost	28.60 ± 18.64	0
Random	3.60 ± 2.00	0
Cooperative Solver	43.98 ± 32.46	2

5 MARIO PCG RESULTS

We train five generators with different solver configurations to fulfil the research objectives of the project. First, we train two generators with individual solvers, one static and the other cooperative. Then we repeat the process but use a population of solvers rather than an individual solver. The fifth generator is a hybrid: it has two static and two cooperative solvers. These agents are trained using the SB3 implementation of PPO with a Multilayer Perceptron (MLP) policy. We select this policy as our observation space is a 32-dimensional vector, which suits the structure of an MLP. Finally, we measure the performance of a generator using the diversity, novelty, and playability metrics described previously. We separate our training and evaluation results into three subsections to differentiate between the different solver paradigms we explored.

While we wish to replicate the hyperparameters used in MarioPuzzle to compare results, we make particular adaptations to make the experiments suitable for our time frame. For example, Shu et al. [20] set the rollout size to 128. The rollout size determines the number of steps our agent takes in the environment before updating its policy using the collected data. We find that training over sufficient rollouts with this hyperparameter is a prolonged process. Therefore, we define the rollout size to be 16. This modifications means that our agent updates its policy more regularly, which causes highly variable results during training. However, in the long-term, performance should converge regardless of rollout size.

5.1 Training Results

In this section, we discuss the training data we collect while completing the different experiments. In order to achieve our project objectives, we analyse the experimental data, explore how each model compares to its counterparts, propose a reason why the experiments transpire as they do, and justify our reasoning.

5.1.1 Cooperative and Static Solvers

We observe that generators with cooperative solvers obtain higher playability rewards than agents with static solvers during training. This is shown in figure 5. This result is consistent with our methodology: whenever a cooperative solver encounters an unplayable segment, we train it on the generated content. Therefore, we expect better performance when we test the solver again on the segment. Due to time restrictions, we train cooperative solvers for two rollouts, with 64 steps per rollout. The cooperative solvers develop new strategies to complete the segments despite the short training period. For example, on its first attempt, the solver might jump forward while constantly boosting, which causes it to fall into a gap; however, during the training process, it learns to jump at the right time to avoid the gap. After training, the cooperative solvers are more likely to complete the segment and determine segment playability correctly, leading to higher playability rewards. We believe this motivates the generator to create more challenging levels. We test this hypothesis during the model evaluation.

The diversity and novelty metrics are calculated independently from the solver. Hence, we expect the solver type to have no immediate effect on these components of the reward function. Our assumption is confirmed by the data collected during training.

5.1.2 Individual Solver and Solver Population

Figure 6a demonstrates that using a population of static solvers has no apparent advantage, in terms of playability, over an individual static solver. The similarity between both experiments may be due to the use of the same *Mario Play* model as a solver across all experiments. We do this to maintain consistency and produce comparable results. Hence, all instances of static solvers behave in the same manner, and we, therefore, expect these solvers to produce similar playability rewards during training.

In contrast, we can observe in Figure 6b that the generator with a cooperative population appears to outperform the model with an individual cooperative solver in terms of playability. In the cooperative population generator, the solvers are tested and trained on distinct level segments, and therefore they develop marginally different strategies to complete these segments. The generator tests new segment configurations using these solvers. While one solver might not be able to determine playability correctly, another solver might as it learns a different approach for SMB levels. We conjecture that using a cooperative solver population induces the generator to learn the latent space of playable segments sooner than a generator with an individual solver. We believe the data collected during training supports this inference; however, we consider it insufficient to prove it ultimately.

Despite similar results in both diversity and novelty, generators with solver populations seem to marginally outperform individual solver generators in diversity, while the opposite is true for novelty. We can observe this in Figure 8. However, we consider our data inadequate to draw any decisive conclusions regarding these metrics due to the high value of the standard deviations.

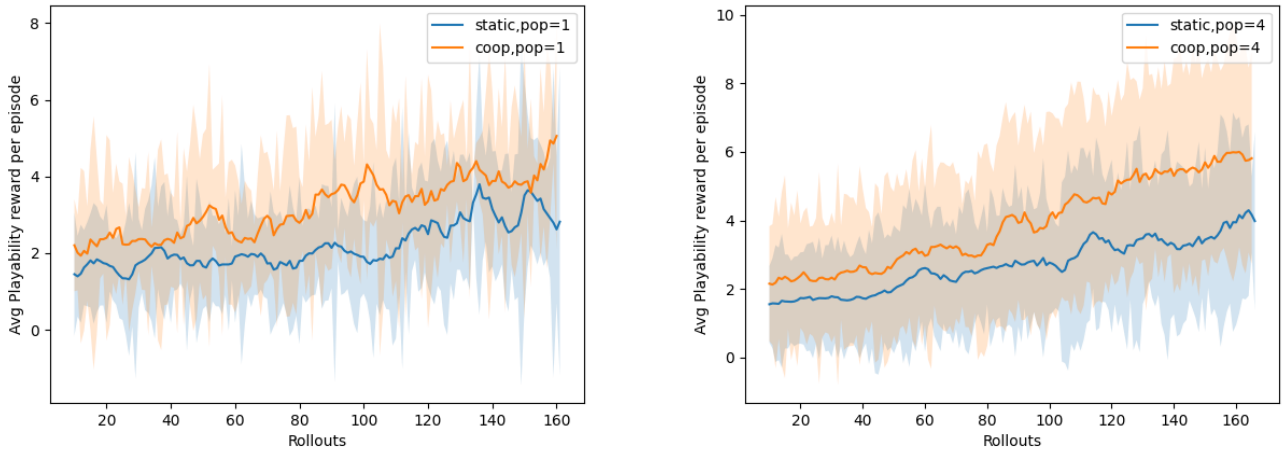

Fig. 5: Graphs 5a and 5b demonstrate how cooperative solvers increase the average playability reward per episode during training. The label *"coop,pop=4"* refers to a generator trained with a population of four cooperative solvers.

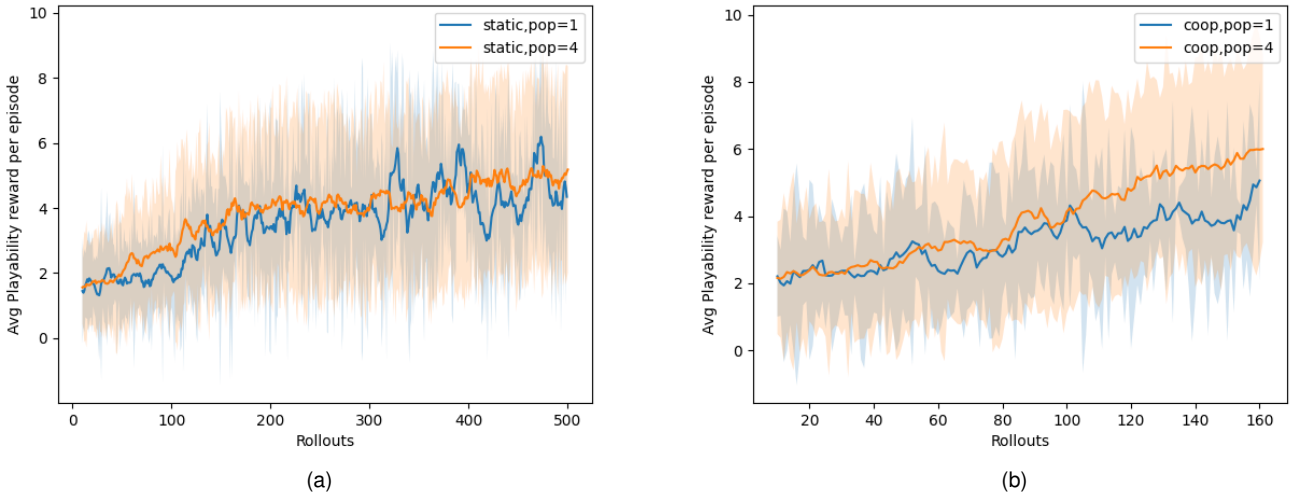

Fig. 6: Graphs 6a demonstrates that using a population of static solvers has minimal impact on playability. Graph 6b indicates that a cooperative population generator outperforms a generator with an individual cooperative solver in the long-term.

5.1.3 Hybrid Solver Population

We observe in Figure 7 that the hybrid model receives similar playability rewards as the static generators. The hybrid model is comprised of two cooperative solvers, and we may therefore expect it to perform similarly to the cooperative generators. However, we suspect that the difference in performance between these generators may result from the static solvers in the hybrid model counterbalancing the effects of the cooperative agents. The cooperative solvers produce higher playability rewards, which we believe encourages the generator to create more challenging segments. Consequently, the static solvers may struggle to complete these harder segments, yielding lower playability rewards and reducing the overall average. In order to test this conjecture, we can measure the average playability score for each solver in the hybrid model separately. We leave this for future work.

5.2 Evaluation Results

We evaluate the trained agents over 50 episodes of level generation. We include a random agent in the evaluation for comparison. During the evaluation, if a segment is determined to be unplayable or the agent generates ten playable segments, we conclude the current episode. The results are shown in Table 2 and illustrated in Figure 9. We record playability, diversity, and novelty. Additionally,

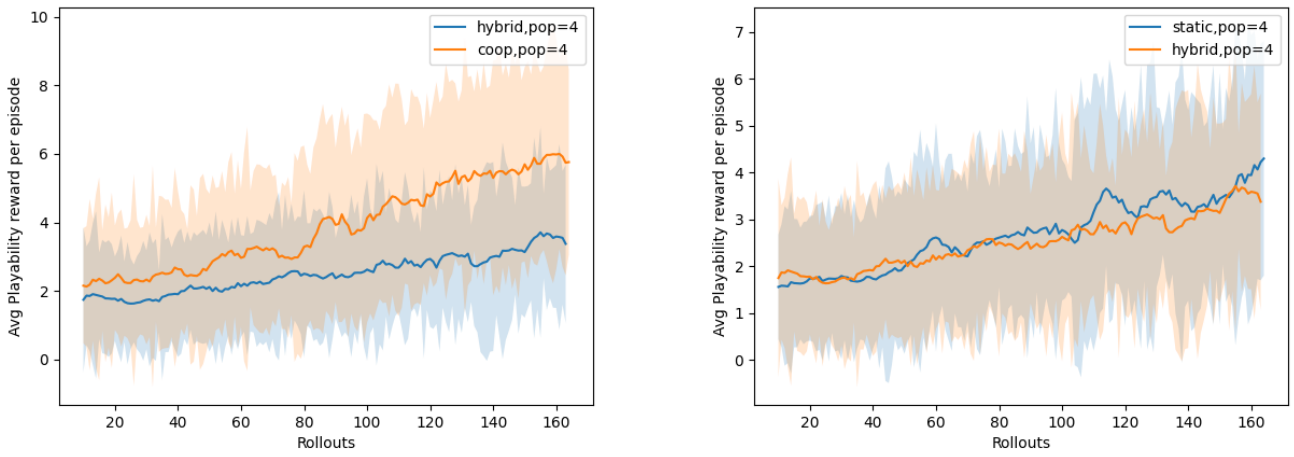

Fig. 7: Graphs 7a and 7b show that the generator with a hybrid population of solvers performs similarly to the static population generator during training, in terms of playability .

we calculate the average number of enemies and gaps per segment as proxies for level difficulty. Mario can lose a game by either running into an enemy, falling into a gap, or running out of time. We notice that on the rare occasion where the *Mario Play* agents run out of time is for one of two reasons. It is either because the generated segment was unplayable; or because our agent became trapped and was unable to escape as it learns never to move left. Therefore, we consider the average number of enemies and gaps per segment appropriate metrics for level difficulty, but not the average time to complete a level.

5.2.1 Cooperative and Static Solvers

We observe that generators trained with static solvers perform better on playability in the evaluation, the opposite of what occurs during training. During training, static solvers fail to complete level segments more regularly than cooperative solvers, thus motivating the generator to create easier levels to maximise playability. The results in Table 2 support our hypothesis that cooperative solvers encourage the generator to produce more difficult levels. This inference is further reinforced by the data on the average number of enemies and gaps per segment. Table 2 demonstrates that generators with cooperative solvers create levels with more gaps and enemies, which we consider equivalent to more challenging levels.

In terms of novelty and diversity, we can see generators with static solvers perform better than cooperative models. However, it is difficult to determine the reason why this occurs without further experimentation.

5.2.2 Individual Solver and Solver Population

During training, we observe that the generator with a cooperative solver population outperforms the other cooperative agent in playability. However, at the time of evaluation, the generators with solver populations perform similarly to their individual solver counterparts across all metrics. Therefore, we cannot conclusively determine whether a population of solvers had any significant impact on our experiments.

5.2.3 Hybrid Solver Population

While the hybrid model performed similarly to the generators with static solvers during training, we notice that its evaluation results are comparable to the results of the cooperative models across all metrics. Despite the relatively low rewards during training, the hybrid generator creates challenging levels, as demonstrated in Table 2. This discrepancy between training and evaluation results supports our previous conjecture regarding the effects of training with static and cooperative solvers on the hybrid generator. We believe that the generator learns to create challenging levels during training because of the cooperative solvers, but the inclusion of static solvers reduces the average playability reward.

6 Evaluation

In this section, we discuss our results and solution. We compare our work to MarioPuzzle [20] and ARLPCG [12]. Additionally, we examine and explain the limitations of our project.

Our work is consistent with the results obtained in [12]. We demonstrate that using an RL solver that cooperates and learns with the generator induces the agent to create content suitable for the skill of the solver. Cooperative solvers encourage the generator to create more challenging content than generators with static solvers do. Similarly, Gisslén et al. [12] observe that the use of an adaptive RL solver teaches the generator to create content that matches the skill of the solver agent. Furthermore, they observe that agents trained as solvers on procedurally generated content can generalise better to unseen environments. We notice that the model trained as a cooperative agent performs better in *Mario Play* than the other agents, including the same model

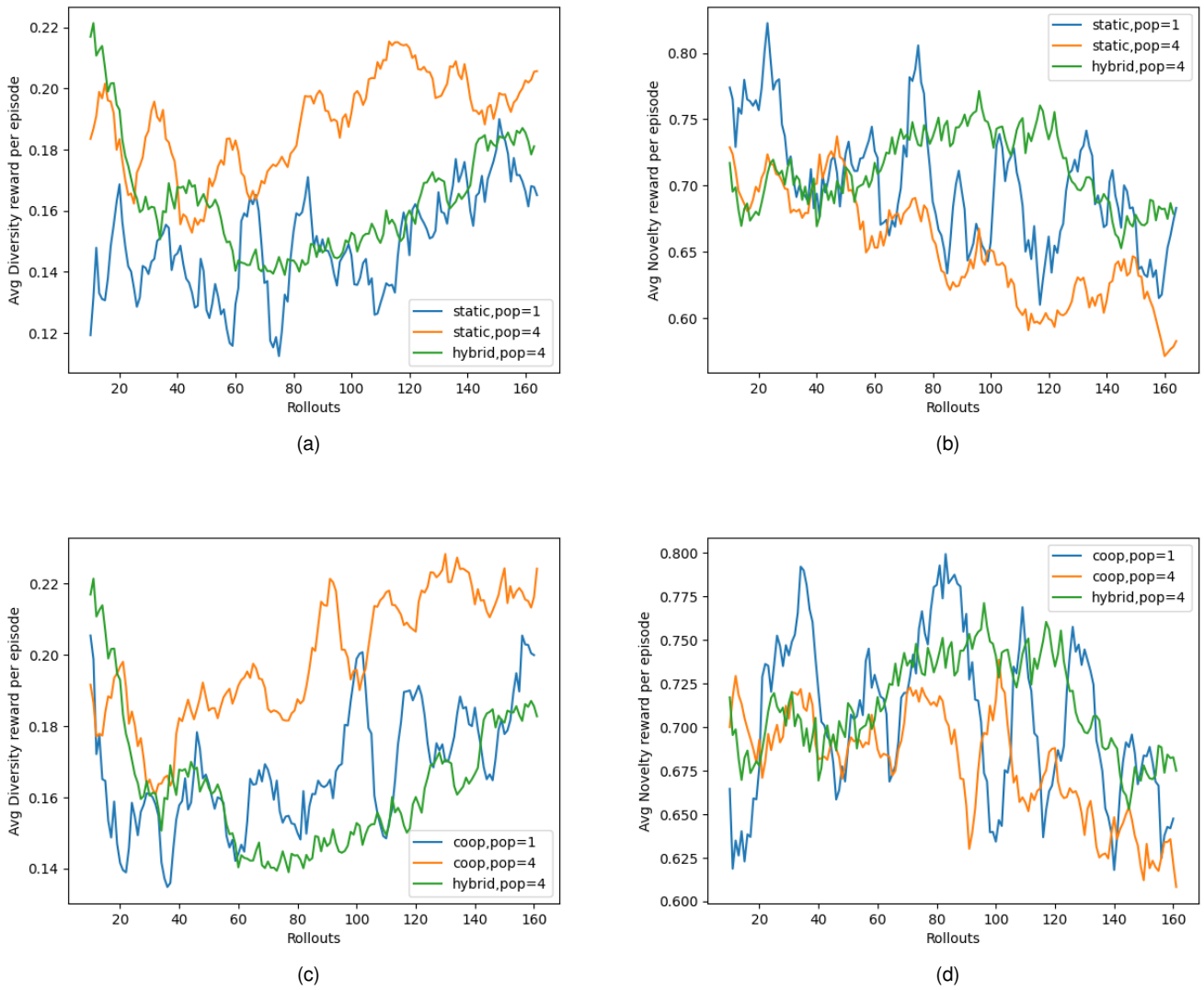

Fig. 8: The graphs show the average diversity and average novelty reward per episode.

TABLE 2: The table presents the evaluation results for the five generators trained on *Mario PCG*. During the evaluation, we record the following: playability as a percentage, i.e. how many playable segments the agent generates out of ten; the average diversity; the average novelty; the average number of enemies per segment; and the average number of empty tiles on the ground (gaps) per segment.

Solver Configuration	Playability	Diversity	Novelty	Enemies	Gaps
Static, Individual	51.02 ± 32.53	0.19 ± 0.11	0.57 ± 0.21	0.58 ± 0.33	0.31 ± 0.29
Static, Population	50.82 ± 29.44	0.18 ± 0.11	0.48 ± 0.23	0.51 ± 0.38	0.26 ± 0.30
Cooperative, Individual	31.63 ± 26.83	0.11 ± 0.11	0.42 ± 0.26	0.95 ± 0.62	0.46 ± 0.90
Cooperative, Population	34.88 ± 27.09	0.13 ± 0.10	0.46 ± 0.17	0.90 ± 0.56	0.43 ± 0.72
Hybrid, Population	$32.0.4 \pm 25.95$	0.11 ± 0.09	0.45 ± 0.24	0.86 ± 0.65	0.39 ± 0.78
Random	15.51 ± 14.01	0.06 ± 0.09	0.30 ± 0.24	0.64 ± 0.62	0.60 ± 1.00

that has not received additional training. This is shown in Table 1.

In terms of our evaluation metrics, we notice that our generators perform significantly worse than the agents trained on MarioPuzzle [20]. The exception is the diversity metric, where our generators perform better. However, we do not believe that the disparity in performance is due to the use of adaptive RL solvers. It is apparent that the difference in results is due to the limitations of the models trained on *Mario Play*, which we discuss below. We believe that with a better solver agent, we could duplicate our outcomes and obtain results that reflect the data in [20].

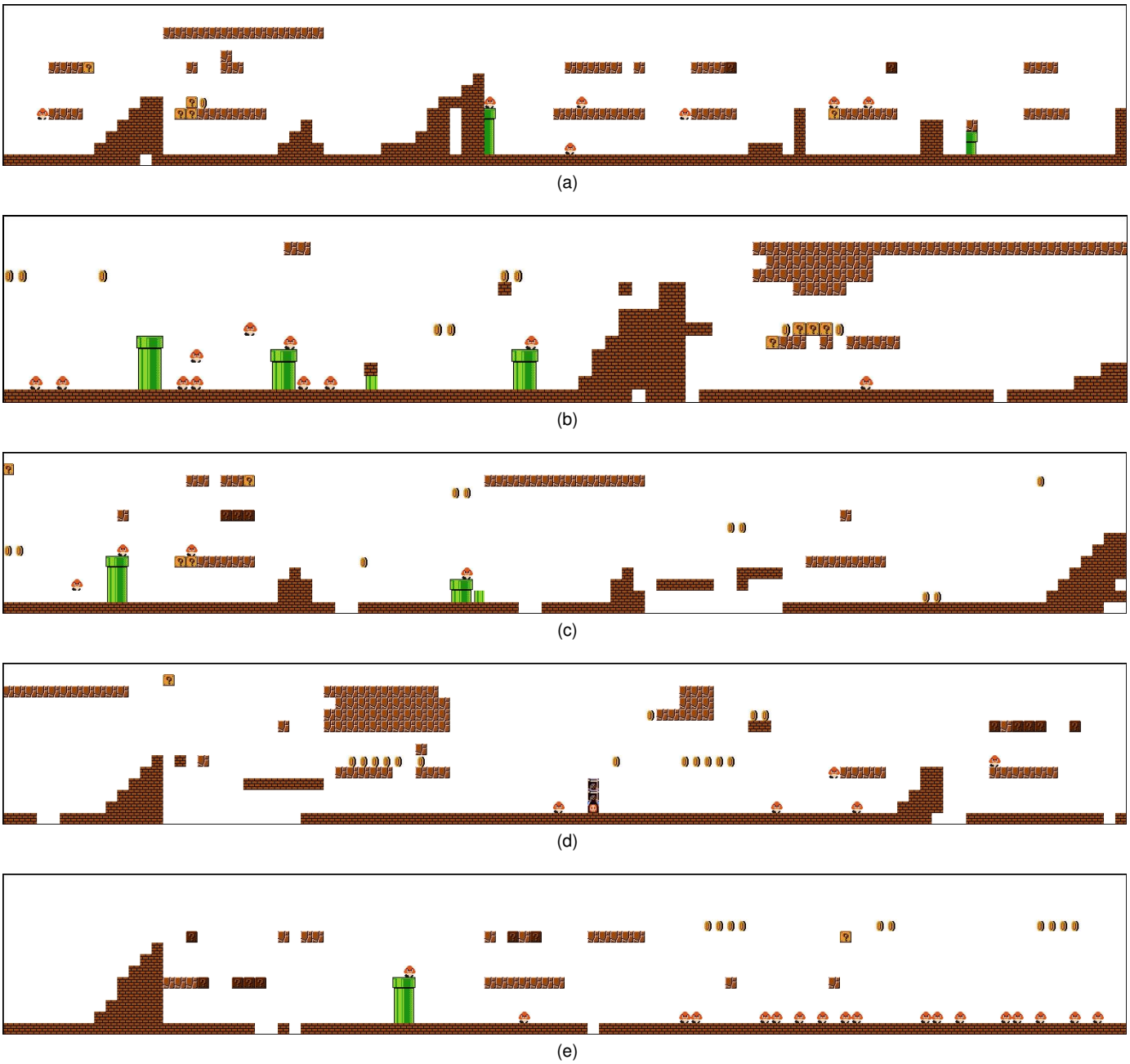

Fig. 9: Levels created by a generator trained with: 9a an individual static solver, 9b a population of static solvers, 9c an individual cooperative solver; 9d a population of cooperative solvers; and 9e a population of two cooperative and two static solvers.

6.1 Project Limitations

There were numerous limitations in our approach. As previously mentioned, the main limitation of the project is the quality of the solver agent. We observed during training that the generator would occasionally create a playable segment that the solver would be unable to complete. Hence, the segment would be labelled as unplayable despite being functional. This limitation is marginally addressed in generators with cooperative solvers as they receive additional training, making them more likely to determine playability correctly. However, despite this minor improvement, we believe that the solver hampered the learning process of the generators across all experiments. Additionally, conducting the *Mario Play* experiments delayed the PCG experiments, which are the main focus of this project.

While we cannot demonstrate that solver populations positively impact the generator, we cannot conclude that they are ineffective either. Note that we chose the best performing agent from the *Mario Play* experiments and use this model as a solver in all generators. Therefore, we would expect all the solvers to behave identically; even in hybrid and cooperative populations, the difference between the solvers is minimal. This similarity between solvers counteracts our initial reasoning regarding why solver populations might be advantageous. We postulated that a population of solvers might help prevent the generator from exploiting

observed behaviours in an individual solver. However, if all solvers act similarly, the generator can still learn to take advantage of recurring behaviours in all agents.

We occasionally encountered Pygame segmentation faults during training. These errors interrupted the training process on multiple occasions. We were unable to determine the source of this error; however, we noticed that it would happen most frequently when running memory-intensive processes, such as training a DQN in *Mario Play* or running a vectorised environment with more than four individual environments. To mitigate this issue, we decided to only train policy optimisation methods, as they are less memory intensive, and to limit the number of independent environments inside a vectorised environment to the number of available cores in our hardware. Additionally, we used checkpointing to save the training data and the model in intervals.

The final limitation we discuss is time. Time constraints impacted every aspect of the project. We devoted a significant portion of the time allotted for the project attempting to augment the MarioPuzzle framework using DJL. We explored different approaches to fulfil our objectives using this library; however, we realised it was untenable and adjusted our solution accordingly. We believe that if we had pursued the current approach from the beginning, we could have addressed some of the limitations mentioned previously and implemented additional features, which we discuss in the next section.

7 Future Work

We wish to address the limitations discussed in the previous section as future work. We believe that with further hyperparameter tuning, we are capable of training an agent that performs better in the *Mario Play* environment. Beyond our current implementation, we wish to explore different RL methods and model policies for our solver agent and investigate the best performing architecture. For instance, we believe that a Recurrent CNN [55] as a model policy, rather than a regular CNN, might improve performance. Furthermore, we wish to implement a Mario agent that does not use RL. For instance, we would like to adapt the A* agent from the 2009 Mario AI Competition [37], which would allow us to compare our framework to MarioPuzzle effectively.

In this project, we explore static and cooperative solvers. In addition, we wish to explore adversarial solvers that compete with the generator, as proposed in [12]. This type of solver would be simple to implement with our current framework; we merely need to modify how our generator is rewarded. However, we do not implement it in the project due to time constraints and leave it as future work instead. Additionally, Gisslén et al. [12] use auxiliary inputs in the reward function of the generator to indirectly control the difficulty and style of the generated content. In future research, we could use auxiliary inputs to control the style of generated SMB levels; for example, we could include a component in the reward function related to the number of enemies the solver agent defeats while testing the level segments.

To further explore and advance the use of RL solvers in PCGRL, we believe that future research should focus on hybrid solver populations. Our work demonstrates that using a population of solvers that behave identically does not impact the generator. However, based on our experiments, we hypothesise that using a hybrid population of distinct solver agents would teach the generator to create content that can be solved with numerous approaches, leading to greater diversity in generated content. Furthermore, future work might consider using a hybrid population of solvers to manage level difficulty indirectly. For instance, our solver population might include a simple deterministic agent and a trained RL agent. If the deterministic solver cannot complete a level segment, but the RL solver can, we might consider the current segment appropriately difficult and reward the generator. We would use the RL solver to determine playability, while the deterministic solver would determine difficulty. Future research could explore the effectiveness of different solver population configurations.

8 Conclusion

In this project, we implemented two different environments, *Mario Play* and *Mario PCG*, and combined them to create a new PCGRL framework for SMB. This framework is usable and conveniently adaptable for future research on the topic. Therefore, we fulfil the fundamental objectives of this project. Additionally, we explore the use of RL solvers in PCGRL for SMB and how two different types of solvers, static and cooperative, affect the generators. Despite the limitations of our solver agents, we demonstrate that using an adaptive RL solver compels the generator to create content that matches the skill of the solver. Furthermore, our results indicate that generators trained with cooperative agents create more difficult content on average than agents trained with static solvers. We conclude that RL solvers that learn and cooperate with the generator can improve the ability of the agent to create challenging and engaging content. Finally, while we are unable to determine the efficacy of solver populations, our work leads us to suspect that a generator trained with a hybrid population of solvers that are significantly distinct from each other might be able to generate more diverse and entertaining content.

REFERENCES

[1] B. Bates, *Game Design*. Thomson Course Technology, 2004.

[2] R. Koster, "The cost of games," 2018. [Online]. Available: https://venturebeat.com/2018/01/23/the-cost-of-games/

[3] N. Shaker, J. Togelius, and M. J. Nelson, *Procedural content generation in games*. Springer, 2016.

[4] A. Summerville, S. Snodgrass, M. Guzdial, C. Holmgård, A. K. Hoover, A. Isaksen, A. Nealen, and J. Togelius, "Procedural content generation via machine learning (pcgml)," *IEEE Transactions on Games*, vol. 10, no. 3, pp. 257–270, 2018.

[5] N. Shaker, J. Togelius, G. N. Yannakakis, B. Weber, T. Shimizu, T. Hashiyama, N. Sorenson, P. Pasquier, P. Mawhorter, G. Takahashi, G. Smith, and R. Baumgarten, "The 2010 mario ai championship: Level generation track," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 4, pp. 332–347, 2011.

[6] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation: A taxonomy and survey," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 172–186, 2011.

[7] M. Cook, S. Colton, and J. Gow, "The angelina videogame design system—part i," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 9, no. 2, pp. 192–203, 2017.

[8] S. Risi and J. Togelius, "Increasing generality in machine learning through procedural content generation," *Nature Machine Intelligence*, vol. 2, no. 8, pp. 428–436, 2020.

[9] J. Liu, S. Snodgrass, A. Khalifa, S. Risi, G. N. Yannakakis, and J. Togelius, "Deep learning for procedural content generation," *Neural Computing and Applications*, pp. 1–19, 2020.

[10] T. Gao, J. Zhang, and Q. Mi, "Procedural generation of game levels and maps: A review," in *2022 International Conference on Artificial Intelligence in Information and Communication (ICAIIC)*, 2022, pp. 050–055.

[11] A. Khalifa, P. Bontrager, S. Earle, and J. Togelius, "Pcgrl: Procedural content generation via reinforcement learning," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 16, no. 1, 2020, pp. 95–101.

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

[13] "Gaming market - growth, trends, covid-19 impact, and forecasts (2022-2027)." [Online]. Available: https://www.mordorintelligence.com/industry-reports/global-gaming-market

[14] M. Hendrikx, S. Meijer, J. Van Der Velden, and A. Iosup, "Procedural content generation for games: A survey," *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, vol. 9, no. 1, pp. 1–22, 2013.

[15] A. Liapis, G. Smith, and N. Shaker, "Mixed-initiative content creation," in *Procedural content generation in games*. Springer, 2016, pp. 195–214.

[16] G. Lai, W. Latham, and F. F. Leymarie, "Towards friendly mixed initiative procedural content generation: Three pillars of industry," in *International Conference on the Foundations of Digital Games*, ser. FDG '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3402942.3402946

[17] M. Guzdial, N. Liao, J. Chen, S.-Y. Chen, S. Shah, V. Shah, J. Reno, G. Smith, and M. O. Riedl, "Friend, collaborator, student, manager: How design of an ai-driven game level editor affects creators," in *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, ser. CHI '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1–13. [Online]. Available: https://doi.org/10.1145/3290605.3300854

[18] G. N. Yannakakis and J. Togelius, "Experience-driven procedural content generation," *IEEE Transactions on Affective Computing*, vol. 2, no. 3, pp. 147–161, 2011.

[19] N. Justesen, R. R. Torrado, P. Bontrager, A. Khalifa, J. Togelius, and S. Risi, "Illuminating generalization in deep reinforcement learning through procedural level generation," *arXiv preprint arXiv:1806.10729*, 2018.

[20] T. Shu, J. Liu, and G. N. Yannakakis, "Experience-driven pcg via reinforcement learning: A super mario bros study," in *2021 IEEE Conference on Games (CoG)*, 2021, pp. 1–9.

[21] R. N. Engelsvoll, A. Gammelsrød, and B.-I. S. Thoresen, "Generating levels and playing super mario bros. with deep reinforcement learning using various techniques for level generation and deep q-networks for playing," Master's thesis, University of Agder, 2020.

[22] "Deep java library." [Online]. Available: https://github.com/deepjavalibrary/djl

[23] G. N. Yannakakis and J. Togelius, *Artificial intelligence and games*. Springer, 2018, vol. 2.

[24] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," *Advances in neural information processing systems*, vol. 27, 2014.

[25] P. Bontrager and J. Togelius, "Learning to generate levels from nothing," in *2021 IEEE Conference on Games (CoG)*, 2021, pp. 1–8.

[26] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[27] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, "Mastering the game of go without human knowledge," *nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[28] N. Shaker, "Intrinsically motivated reinforcement learning: A promising framework for procedural content generation," in *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, 2016, pp. 1–8.

[29] S. Singh, R. L. Lewis, A. G. Barto, and J. Sorg, "Intrinsically motivated reinforcement learning: An evolutionary perspective," *IEEE Transactions on Autonomous Mental Development*, vol. 2, no. 2, pp. 70–82, 2010.

[30] Z. Chen, C. Amato, T.-H. D. Nguyen, S. Cooper, Y. Sun, and M. S. El-Nasr, "Q-deckrec: A fast deck recommendation system for collectible card games," in *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, 2018, pp. 1–8.

[31] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3, pp. 279–292, 1992.

[32] R. Bellman, "A markovian decision process," *Journal of mathematics and mechanics*, pp. 679–684, 1957.

[33] M. Kerssemakers, J. Tuxen, J. Togelius, and G. N. Yannakakis, "A procedural procedural level generator generator," in *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, 2012, pp. 335–341.

[34] D. Bhaumik, A. Khalifa, M. C. Green, and J. Togelius, "Tree search vs optimization approaches for map generation," 2020.

[35] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017.

[36] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[37] J. Togelius, S. Karakovskiy, and R. Baumgarten, "The 2009 mario ai competition," in *IEEE Congress on Evolutionary Computation*, 2010, pp. 1–8.

[38] V. Volz, J. Schrum, J. Liu, S. M. Lucas, A. Smith, and S. Risi, "Evolving mario levels in the latent space of a deep convolutional generative adversarial network," in *Proceedings of the genetic and evolutionary computation conference*, 2018, pp. 221–228.

[39] T. Shu, Z. Wang, J. Liu, and X. Yao, "A novel cnet-assisted evolutionary level repairer and its applications to super mario bros," in *2020 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2020, pp. 1–10.

[40] R. Koster, *Theory of fun for game design*. " O'Reilly Media, Inc.", 2013.

[41] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[42] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-baselines3: Reliable reinforcement learning implementations," *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021. [Online]. Available: http://jmlr.org/papers/v22/20-1364.html

[43] M. Bauer, "super-mario-python: super mario in python and pygame," 2017. [Online]. Available: https://github.com/mx0c/super-mario-python

[44] P. Community, "Pygame," 2000. [Online]. Available: https://www.pygame.org

[45] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT Press Ltd, 2018.

[46] J. Achiam, "Spinning Up in Deep Reinforcement Learning," 2018.

[47] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[48] C. Kauten, "Super Mario Bros for OpenAI Gym," GitHub, 2018. [Online]. Available: https://github.com/Kautenja/gym-super-mario-bros

[49] S. M. Lucas and V. Volz, "Tile pattern kl-divergence for analysing and evolving game levels," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2019, pp. 170–178.

[50] S. Karakovskiy and J. Togelius, "The mario ai benchmark and competitions," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 55–67, 2012.

[51] A. Khalifa, "Mario-ai-framework," 2021. [Online]. Available: https://github.com/amidos2006/Mario-AI-Framework

[52] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," *arXiv preprint arXiv:1511.06434*, 2015.

[53] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein generative adversarial networks," in *International conference on machine learning*. PMLR, 2017, pp. 214–223.

[54] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *Proceedings of The 33rd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. F. Balcan and K. Q. Weinberger, Eds., vol. 48. New York, New York, USA: PMLR, 20–22 Jun 2016, pp. 1928–1937. [Online]. Available: https://proceedings.mlr.press/v48/mniha16.html

[55] S. Behnke, *Hierarchical neural networks for image interpretation*. Springer, 2003, vol. 2766.