Doctoral Dissertation

Using Reinforcement Learning to Generate Game Content with Quality and Diversity

Sanggyu Nam

Supervisor Prof. Kokolo Ikeda

Graduate School of Advanced Science and Technology
Japan Advanced Institute of Science and Technology
(School of Information Science)

September, 2022

Abstract

Procedural content generation (PCG) has arisen as one major research field in games. PCG aims to provide content that meets specific criteria of quality, such as playability, difficulty, and entertainment. In addition, the generated game content should be different from those currently offered. Therefore, achieving quality and diversity becomes one of PCG's most important objectives.

Machine learning is a representative algorithm for PCG and is called procedural content generation via machine learning (PCGML). PCGML usually requires data for learning. However, collecting sufficient and good data can be costly. Another typical method is search-based PCG, where content is generated by evolutionary and other metaheuristic search algorithms. The search-based PCG does not require training data. Instead, it tries to optimize the given evaluation functions. The optimization process is time-consuming, making search-based PCG challenging to generate content immediately on demand.

To solve these problems, we apply reinforcement learning (RL), which does not require training data but learns from interactions with the environment. Another advantage of RL is that once the training is done, it can generate content quickly when required. In this dissertation, we target two kinds of games with distinct characteristics, turn-based role playing games (turn-based RPG) and Super Mario Bros. (Super Mario).

For turn-based RPG, we train RL agents to generate stages, where a stage is a series of events such as battles and recovery. For Super Mario, we train RL agents to generate levels, where a level consists of tiles such as walls and enemies. The generation is a challenging task since components in the stages and levels, such as the events and tiles, are highly correlated.

In order to address this challenge using RL, it is necessary to formulate the problem by a Markov decision process (MDP). Thus, we formulate the stage/level generation into MDPs, where the ideas can be generalized to other PCG problems. For the rewards in MDP, hand-crafted evaluation functions which reflect players' enjoyment are used to evaluate generated content.

For the stage generation problem, two classical RL algorithms are adopted. One is a deep Q-network (DQN) for discrete action spaces, and the other is a deep deterministic policy gradient (DDPG) for continuous action spaces. Experiments show that both DQN and DDPG can generate good stages, where those by DDPG generally receive higher scores from the evaluation function.

Next, we try to apply similar approaches to the level generation for Super Mario. However, Super Mario's level generation has several distinct challenges that do not exist in the stage generation. One is related to how to represent the levels. Unlike events in turn-based RPG that can be straightforwardly represented by a small number of parameters, tiles in Super Mario levels need to be represented by a large number of parameters. For this challenge, we employ conditional generative adversarial networks (CGAN), which have succeeded in generating images. Another challenge is related to how to evaluate the levels' difficulty. For this challenge, we employ a human-like AI agent and have it play the generated levels. We use twin delayed DDPG (TD3) for the level generation problem. As a result, the generated levels receive high evaluation values indicating good quality.

Finally, virtual simulations that give rewards to intermediate actions are employed to obtain content with even higher quality. In addition, we introduce noise to avoid generating similar stages/levels while trying to keep the quality high. The experimental results highlight that the proposed methods successfully generate good and diverse stages/levels for turn-based RPG and Super Mario.

Keywords: Reinforcement learning, Procedural content generation, Turn-based RPG, Super Mario, Quality-Diversity.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Procedural generation has received considerable research attention in many application domains for cost-saving and quality assurance benefits. One crucial area of procedural generation is to create computer game content, and the technique is well known as procedural content generation (PCG). PCG signifies to procedually generate game content with any kinds of algorithms.

One example of PCG usage is for generating various game elements such as levels [1], dungeons [2], maps [3], characters [4], and items [5]. The primary goal of PCG is to enhance game replayability and reduce financial and memory burdens. In addition to these purposes, PCG can be used for training AI players in diversely generated levels to increase their generality [6] so that using the AI players to do auto playing test can produce more reliable results. Previous works have applied PCG to many different games, each with distinct characteristics and challenges. The majority of PCG studies is on a subset of game genres such as platformer games [1][7], racing games [8][9], and puzzle games [10].

To date, PCG has aimed to generate content that qualifies the standards of game developers. In addition to quality requirements, PCG needs to ensure content novelty since players expect new content in every launch. In this dissertation, the generated content evaluated according to defined criteria is called quality, and content that is different and novel from others is called diversity.

This dissertation applies PCG to two kinds of games with distinct characteristics, namely turn-based role playing games (turn-based RPG) and Super Mario series. Turn-based RPG is a renowned game genre with minor recognition in the PCG research field. [1]

[1]The definition of turn-based RPG may vary from broad to narrow. In this dissertation, turn-based RPG is considered as games like Wizardry, Darkest Dungeon, or Pokémon.

Most RPGs share one typical flow. Players control characters according to the storyline and defeat the final boss. During the storyline, players can reinforce their characters from a series of events that occur after another, such as rewards from battles, obtain items from treasure boxes, and trade with merchants. Players experience this sequence of events several times during gameplay, and this dissertation defines it as *a stage* of turn-based RPGs. In well-designed and challenging RPGs like Darkest Dungeon [11] or the Mysterious Dungeon series [12], if players attempt to defeat all consecutive monsters using all of their resources, then they may fail to defeat the boss. Alternatively, if players ignore all enemies, then the characters may not have adequate strength to defeat the boss. Because of these unique features, under many circumstances, players need to devise strategies, such as winning a tough battle using all resources or saving items for more important battles later. These lead to the entertainment of turn-based RPGs.

Stages of turn-based RPGs should be well designed so that players need to consider their strategies carefully, which makes them feel challenging and entertaining. It is crucial to make a balance between events in the stages, such as the locations, frequencies, and statuses of enemies, the locations and effectiveness of recovery points, and the effects of items. Game designers have employed constructive PCG methods (usually hand-designed rules) to generate stages in many commercial turn-based RPGs, which gives designers a high level of control on the generated content. However, it requires experts' dedication to thoroughly create rules or decide parameters, which still cannot guarantee adequate game balance. For example, Disgaea [13] is one of the famous turn-based RPGs that have parameters in a wide range. Despite that, many enemies in later phases of the game can be defeated by only one attack as player characters get stronger rapidly compared to enemies. Providing diverse stages to players is another aspect to consider for entertainment. Players need to develop different strategies under different circumstances, which usually makes play enjoyable, especially in turn-based RPGs. Constructive PCG methods also have potential issues about lacking diversity in that players may somehow find rules in the stages.

We further break down a level into several *patterns*. Thus, the level generation is to place patterns one by another. The layouts and types of tiles directly affects the players' play styles, so players may enjoy the level or get bored of the level. It is essential to place a proper pattern from the previous pattern. Even if one pattern is enjoyable on its own, the difficulty can changes drastically when it is concatenated with improper patterns. Well-designed levels should look natural. Also, since players have a large variety of skills, levels with the corresponding difficulty are desired.

Researchers have tried several approaches for PCG. One example is pro-

cedural content generation via machine learning (PCGML) [14]. In most PCGML studies, models learn how to generate content using existing content. When it is easy to obtain substantial content created by human designers, it might be possible to generate game content based on the distribution of prepared data using generative models such as variational autoencoders (VAE) [15], PixelCNN [16], or generative adversarial networks (GAN) [17]. However, concerning actual game development, it is not easy to collect a sufficient amount of content for training. Thus, it may result in generating similar content to existing one and lacking diversity.

There is another approach called search-based PCG [18]. The optimization systems in search-based PCG mainly consist of two parts, the generator and the evaluator. Usually, game content is represented by parameters and generated by repeating the following processes, 1) the evaluator grades content by one or a vector of real numbers instead of a simple acceptance or rejection, and 2) the generator aims to find better parameters. Search-based PCG does not require training data; instead, it requires evaluation functions, usually designed by humans. Designers can tailor the evaluation functions to their preference in games, or to some specific players' skills or taste. Typically, search-based PCG generates content by optimizing the evaluation values, and in many cases, the optimization is based on genetic algorithms (GA). Some efforts may be required to enable GA to generate diverse content [19].

Studies about PCG has been conducted mainly by adopting PCGML and search-based PCG. Both PCG methods, PCGML and search-based PCG, have advantages and disadvantages and are used based on the different application and restrictions. To date, there are no effective methods to handle specific conditions, such as lack of training data and the need for a prompt generation. Therefore, this dissertation addresses the following research questions by proposing a novel PCG method to obtain diverse and high-quality content.

RQ1 What kind of method can handle the lack of training data and the need for an online generation?

RQ2 Can the method obtain both diversity and good quality?

RQ3 Which games can the method be applied to?

We address the questions by proposing reinforcement learning (RL) to generate high-quality and diverse game content for two game genres with different characteristics. The work by us [20][21] and Khalifa et al. [22] was undertaken independently, and to our best knowledge, ours is the first to train

RL agents to generate game levels/stages from scratch. The main reason why we introduce RL is that there is no need to collect data and it can do online generation with the potential of getting diverse content. The general idea on transforming levels/stages generation into an RL problem in both works are the same. In more detail, both formulate the problem into Markov decision processes, where states are generated levels/stages, actions are to modify the levels/stages, and rewards are the degree of how good the generated levels/stages are. Khalifa et al. [22] put more emphases on the generality of the method and on generating playable games, while this dissertation emphasizes more on the balance of events in stages, the non-monotony of the layout with proper balance (i.e., quality) in levels, and the diversity of stages/levels. In our design, each stage/level comprises n events/patterns, and the generation of a stage/level means deciding the n events/patterns in sequence. An evaluation function is designed to evaluate the completed stages/levels and give rewards. In addition to generating good stages/levels, we also introduce a noise selection that selects good but different actions from the learned policy on purpose in order to generate diverse stages/levels.

This dissertation is written based on our journal article [23], with additional generation of levels for Super Mario (Section 3.2 and Chapter 6). The structure of the rest of this dissertation is as follows. Chapter 2 describes the background of our work. Chapter 3 introduces target games, turn-based RPG and Super Mario. Chapter 4 presents general approaches to PCG problems using RL. Chapters 5 and 6 explain approaches to generate stages and levels including experimental results, respectively. Finally, Chapter 7 includes the conclusions and discussions on future works.

Chapter 2

Literature review

In this section, we first give an overview on PCG. Next, Section 2.2 and 2.3 introduce research on search-based PCG and PCGML, respectively, which are representative PCG approaches. As a new approach, we adopt reinforcement learning, where the general concepts of reinforcement learning are reviewed in Section 2.4.

2.1 PCG overview

PCG [19] creates game content algorithmically. The generation process can happen upon players' demands (online) or during game development (offline), where the time constraint usually makes the former more challenging. For both online and offline generation, the content should have high quality. In addition, diversity [24][25] is an important factor to keep players' long-term enjoyment in terms of freshness. Related to players' enjoyment, some studies considered perspectives such as difficulty [26] [27] [28] [29] and entertainment [30].

In addition to the above, PCG can be categorized based on various perspectives [31]. When considering methodologies based on artificial intelligence, two major groups are search-based PCG and PCG via machine learning (PCGML), which will be introduced more in the following sections.

2.2 Search-based PCG

Search-based PCG [18][8][9][31] is one special case of generate-and-test algorithms that aims to generate good content efficiently, even for complex games. Usually, generate-and-test algorithms involve the repeated processes of generating content through the stochastic procedure and then filtering

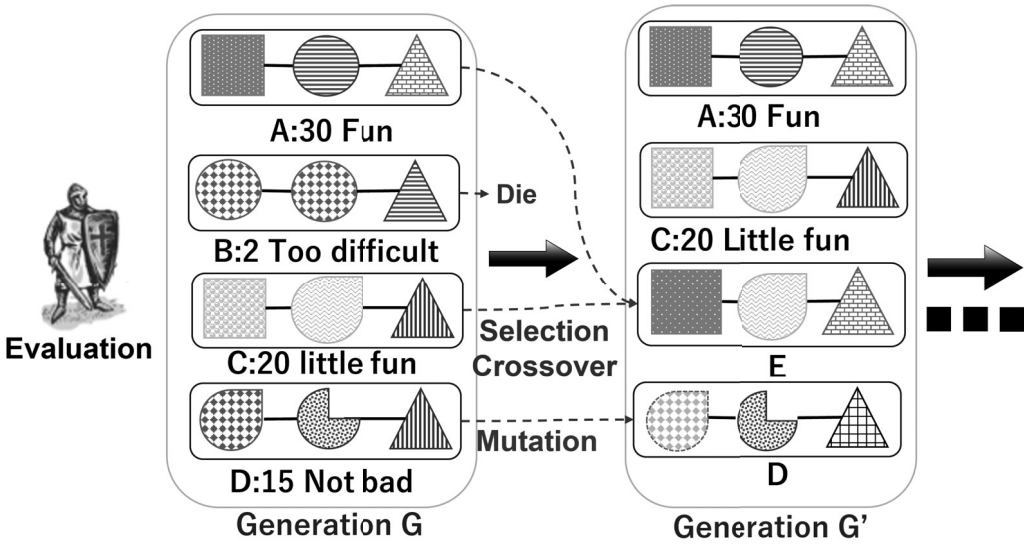


Figure 2.1: Generation of stages using GA. Each stage is represented as an individual containing three discrete events. An evaluator assesses individuals (A, B, C, D) in generation G, and good individuals (A, C) are selected. Through crossover and mutation operations, better genes are passed to the next generation G'.

by evaluation functions until good content is obtained. However, when the target game is complex and some degree of quality is required, generating good content is quite tough as the probability of finding good content with random generation is relatively low whereas search-based PCG can generate good content in complex games [32][8][9]. Search-based PCG mainly applies genetic algorithms (GAs), which keep evolving existing content to obtain content with better evaluation values. Fig. 2.1 shows an example of the generation of stages in a turn-based RPG using GA. By repeating the operations of selection, crossover, and mutation, the GA aims to generate better stages from previously generated ones. Search-based PCG can also achieve adaptive generation according to the design of evaluation functions. For example, Togelius et al. [8] used entertainment features from player logs and optimize fitness to specific players.

While search-based PCG is efficient compared to naive generate-and-test algorithms, still, it has two potential issues, the diversity of content and online generation. Since GAs tend to generate similar individuals, search-based PCG may have issues about generating diverse content. Loiacono et al. [9] tried to generate diverse content (tracks in racing games) based on Togelius et al.'s work [8]. Gravina et al. [24] proposed a new concept,

PCG through quality-diversity, as a subset of search-based PCG that aims to maintain both the quality and diversity of generated content. For example, Alvarez et al. [33] used the MAP-Elites algorithm to generate designer-interactive dungeons. No matter whether diversity is required, search-based PCG needs a large number of evaluations, in other words, trials and errors. Thus, online generation may be difficult, especially when the evaluations are slow.

2.3 PCG via machine learning

PCGML [14] is a framework for generating content using machine learning. The generation models learn how to generate content using existing content as training data. Different from search-based PCG, PCGML generates content directly from the trained models.

PCGML has been realized by many different approaches and applied to generate various kinds of game content, mainly game levels. Summerville and Mateas [1] generated Mario levels by a kind of neural network called long short-term memory (LSTM), known to be good at predicting the next item in a sequence. They represented the 2-dimension levels as strings, where each character stood for a tile on the map. Given a seed sequence (i.e., several pre-assigned tiles), the LSTM generated a level tile-by-tile until reaching an end-of-level terminal symbol. Lee et al. [34] trained convolutional neural networks to predict resource locations in StarCraft II maps, which was considered a potentially useful tool for map designers. Summerville et al. [35] learned the room-to-room structures of Zelda dungeons by Bayesian networks and generated new levels that have similar statistical properties to given levels. Generative adversarial networks (GAN) and variational autoencoder (VAE) have become popular approaches in recent years, which successfully generated similar but new images from input datasets. The former has been adopted to PCG for generating Mario levels [7] [36], Zelda levels [37], Doom levels [38], and an educational game for middle school students [39]. The latter has been adopted to PCG for generating Mario levels [40] [41], lode runner levels [42] and levels of six platformer games [43].

PCGML may have issues such as not having enough data for learning, determining what features should the generated content use and what kind of data should be collected. For example, if the turn-based RPG stages are the target, it is difficult to say whether the collected stage data are associated with appropriate difficulty, or whether the stages require specific strategy behaviors. It means that gathering data involves grasping the characteristics of the content, so it is difficult and expensive, and noise is likely to be in-

cluded. In addition to this problem, it is hard for the content of other games to be reused for the target games because the content of different games has a different structure. These are some of the issues that should be considered when applying PCGML.

2.4 Reinforcement learning

We adopt reinforcement learning (RL) as a new approach aiming to solve the above difficulties. For RL problems, the environments are usually modeled by Markov decision processes (MDP), a mathematical formulation widely employed when studying optimization problems. MDP is usually represented by (S, A, P, R, γ) where S is state space, A is action space, $P_a(s, s') = \Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$ is the probability of transition from s to s' by performing action a at time t, $R_a(s, s')$ is the immediate reward received from the transition, and $\gamma \in [0, 1]$ is the discount factor which shows how much the future cumulative rewards is considered compared to the immediate rewards. Every next state s' depends only on the current state s and the action a. The state transition is independent of past states, which satisfies the Markov property.

Many RL algorithms employ MDP as mathematical formulation. Although this makes RL can handle the decision-making process mathematically, one difficulty can arise. In many RL environments, the immediate reward does not clearly reflect the goal of the environment. It is sparsely happened or is delayed until the end of the episode. Therefore, it is challenging to grasp how each action among a sequence of actions influences the final outcome. This challenge is called a credit assignment problem (CAP) [44].

To address the temporal credit assignment problem, various methods were proposed, which can be roughly divided into two groups [45]: assigning correct credit based on gradients and based on extra values or targets. As an example of the former, Ferret et al. [46] introduced a new transfer learning approach that used a self-attentive architecture to assign credit in a backward view. For the latter, the main idea is to use surrogate rewards for the actions. For example, Arjona-Medina et al. [47] and Liu et al. [48] decomposed the returns of episodes back to the actions. Harutyunyan et al. [49] assigned credits according to the likelihood that the actions led to the observed outcome. Yu et al. [50] applied a Monte-Carlo method to estimate the rewards of intermediate actions in the work of SeqGAN, which aimed to generate sequences of discrete tokens (e.g., sentences).

Different from supervised learning and unsupervised learning, RL does not require training data but does self-learning by interacting with the environment. When all state-action pairs can be listed, whose Q-values can

be stored in a table, it is guaranteed that the optimal policy can be learned given sufficient time. However, this condition is difficult to satisfy in most of the real-world problems. Thus, function approximators such as convolutional neural networks (CNNs) are used to approximate the Q-values. Mnih et al. [51] succeeded in creating AI that is stronger than humans in about half of the 49 tested Atari2600 games by the proposed deep Q-network. Badia et al. [45] succeeded in creating AI that is stronger than humans in all of the 57 tested Atari2600 games in Arcade Learning Environment [52] by solving two issues, CAP and exploration.

Some important features of RL for solving problems are summarized as follows.

- The problems should be able to be modeled by MDP.
- Reward functions should be carefully designed so that the desired policies can be obtained when the rewards are maximized.
- States can be represented in various ways, and the way of representation is important since it influences the computational complexity of the problem.
- Depending on the RL models, outputs can be stochastic or deterministic.
- Learning usually requires time. However, once the learning is done, selecting actions is often fast.

Guzdial et al. [53] and Sheffield and Shah [2] formulized level generation as MDP, though their methods were based on supervised learning. As an independent work of our work, Khalifa et al. [22] used RL for generating game levels. They designed three kinds of representations for two-dimensional game levels and applied them to three games. Some more comparisons with our design will be made in Section 4.4.

Chapter 3

Target games and content to generate

This dissertation proposes an effective method under conditions where training data is limited and online generation is required. In order to validate our proposed method, a relatively simple turn-based RPG is employed as the first target game. If the target game is too complex, it is challenging to determine whether failures are from the method or complicated game settings. After validating with the turn-based RPG, the method is applied to the Super Mario, a more complicated game, to guarantee its effectiveness. This chapter describes the detailed configuration and information of the two target games. First, information on events, discussion of elements in desirable stages, and configuration of implemented turn-based RPG platform are outlined in Section 3.1. Second, Section 3.2 provides information on the components (e.g., Mario and tiles) of levels in Super Mario. It also outlines the discussion of elements in desirable levels and the configuration of employed two Super Mario platforms. Last, the differences between the two games and their challenges for generations are compared in Section 3.3.

3.1 Stages of turn-based RPGs

Most turn-based RPGs have unique systems but have several common elements. For example, players need to grow their characters and finally defeat the boss. A stage usually consists of several events, which can be roughly divided into battle events and non-battle events. The arrangement of events greatly influences the enjoyment of playing.

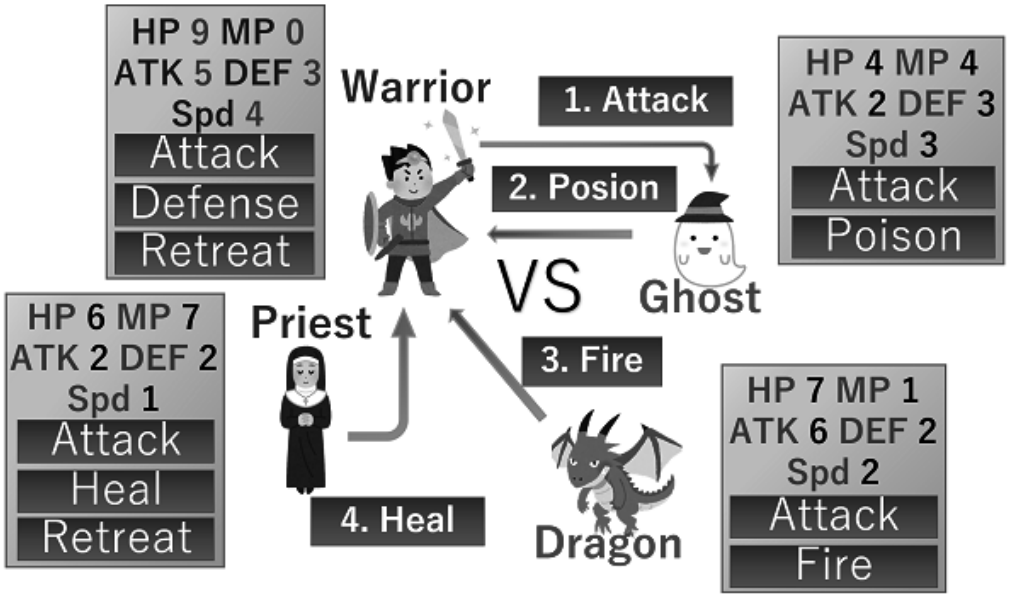


Figure 3.1: An example turn-based RPG battle event and the flow of one turn. The warrior with the highest speed first attacks the ghost. The next fastest ghost performs a poison action on the warrior. Next, the dragon uses a fire skill on the warrior. Finally, the priest heals the damaged warrior.

3.1.1 Events details

In a battle event, a player's team and an enemy's team fight with each other until one team defeats the other team or flees from the battle. In most turn-based RPGs, the status of the characters is preserved after battles. For example, damage received during a battle is not recovered after the end of the battle. Thus, selecting a short-term strategy like just winning the immediate battle is usually not appropriate, and a long-term strategy considering future battles is required. That is the most crucial factor in turn-based RPG.

Battles in turn-based RPG proceed based on turns. In each turn, each character performs one action in order, which is typically decided by a parameter called speed. Fig. 3.1 shows an example of a sequence in one turn in a two-versus-two battle, including characters' status information. In the figure, the square boxes next to the characters show the characters' names, health points (HP)/maximum HP, magic points (MP)/maximum MP, attack (ATK), defense (DEF), speeds (SPD), and possible actions, from the top and the left.

Non-battle events occur outside battles, such as inns where players can recover their characters, blacksmiths who reinforce players' weapons, item

shops providing items for the next expedition, and treasure chests where players can get weapons during the expedition. Settings of the non-battle events can also significantly affect the difficulties or strategies of the battles. Therefore, it is not advisable to design battle and non-battle events separately.

Players in turn-based RPGs advance the stories by repeating non-battle events and battle events while determining their choices in each event. In this sense, players may prefer stages requiring them to make various decisions (e.g., focusing on weak enemies first or trying to defeat all encountered enemies). Conversely, if all decisions from players produce similar results, they may get bored with the game.

3.1.2 Desirable stages

Turn-based RPGs usually contain several separated scenes, such as the base towns, the dungeons in which enemies appear, and the fields connecting scenes to others. Fig. 3.2 shows an example of a simplified turn-based RPG. At the base town, characters can buy items and heal their wounds. However, in order to recover damages received in dungeons or fields, characters need to consume medicine, use magic, or find a recovery point. The game flow can be roughly grouped into four courses in general. The four courses, as shown in Fig. 3.2, are (a) leveling up and collecting items, (b) exploring the world, (c) going through dungeons and defeating mid-bosses, and (d) moving to other base towns. Players play the game by repeating the four courses as they like, with the goal of defeating the final boss.

Different designs are required for different courses. In this dissertation, the designs of the stories, the videos, and the audio of turn-based RPGs are skipped and target the relations between events for course (c). Usually, when players want to defeat a mid-boss in a dungeon, their characters need to explore the whole dungeon from the beginning to the boss without withdrawing or being defeated. Otherwise, they need to start from the very beginning of the dungeon next time. Defeating enemies in the dungeon helps the player characters to level up so that they may become strong enough to defeat the mid-boss. However, in order to start the mid-boss battle with a good condition, they also need to manage the medicine or magic available to recover damage properly in intermediate battles. The following are factors that we consider to make players feel satisfied with the course (c) stages.

- Which strategies are valid is a crucial issue in RPGs. We consider that stages are not enjoyable when no strategy is effective or all of them are. Also, for stages with different designs and settings, it is boring

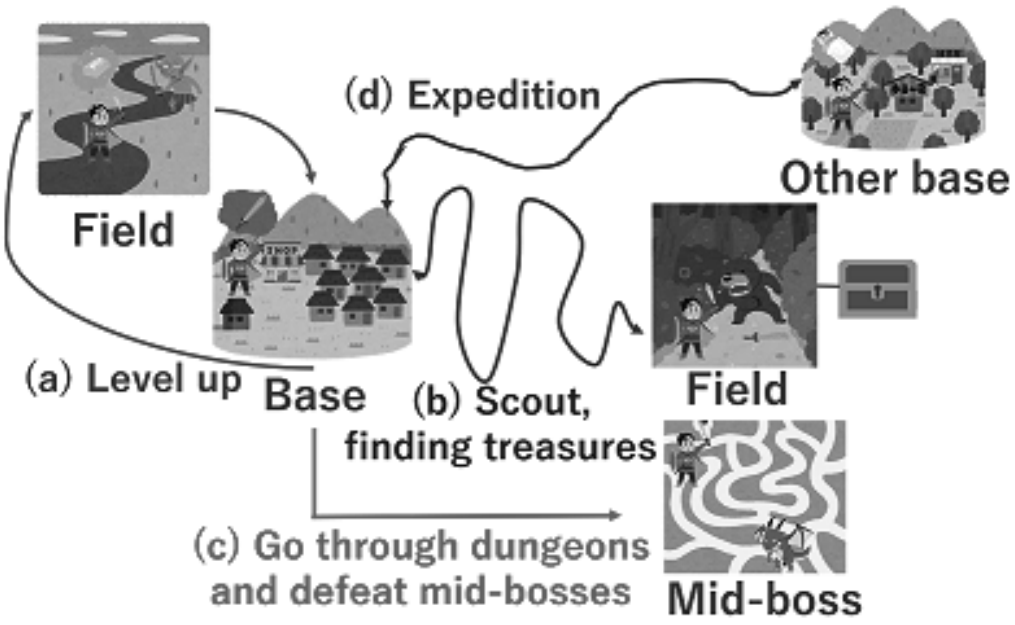


Figure 3.2: Four courses in an example turn-based RPG. (a) Base → Level up, get items and gold in the field → Return to the base, (b) Base → Explore the field or dungeon → Scout or pick up treasure, (c) Base → Explore the dungeon preserving items → Defeat the mid-boss, (d) Base → Move to field → Move to a new base town.

Table 3.1: Comparison of typical turn-based RPGs and our platform

	Typical turn-based RPG	Research platform
Stage structure	diverse stage structures	one-way battle and recovery events
Game objective	defeating the boss after a long journey	defeating the boss after proceeding events
Number of controllable members	≥1	1
Character parameters	many parameters	HP, ATK, SPD
Order of actions in battles	by SPD with unique rules	player character's SPD > enemies'
Actions in battles	various skills	attack, retreat
Reward of winning a battle	leveling up, gold, items, etc	increasing ATK (10%)
Result of retreating a battle	game over, nothing happen, etc	losing HP (15%)

if players can stick to a single strategy, or some specific strategies are obviously too good or bad. Stages should be designed so that players can enjoy thinking about their choices.

- Enemies should have proper strength according to the timing of their appearance. Hard to defeat strong enemies in early stages or too-weak enemies in late stages are unreasonable.

Game balancing has been known to be crucial but difficult, and turn-based RPGs are such an example [54]. Even in simpler designs (e.g., our platform described in the next section), satisfying the factors mentioned above is not easy, not even to say more complex designs, e.g., skill-based systems that players can try all available skills in the skill-spaces without designer-imposed limitations [55].

3.1.3 Game platform

As mentioned in the previous section, we only focus on the relations between events in this dissertation, particularly battle events and recovery in non-battle events. As there was no proper environment for our research, we designed an extensible platform that contains the most fundamental elements in turn-based RPGs. Table 3.1 summarizes the comparison between typical

turn-based RPGs and our platform. Depending on the settings, our platform can be adapted to various types of turn-based RPGs. Section 5.1 formulates the stage generation problem in our platform into a general MDP problem. We implemented a simplified version for the experiments.

3.2 Levels of Super Mario

Super Mario Bros. (Super Mario) [56] is a game from Nintendo that with sales of more than 40 million copies [57]. It has several core systems that are the foundation for the later Super Mario games. Fig. 3.3 illustrates a gameplay scene in Super Mario. During gameplay, a player controls Mario to pass various obstacles and reach the goal on the right-most side within the time limit. The player loses the game immediately once Mario without power-up state gets damaged by enemies. If Mario is in a power-up state from obtaining a mushroom, he can withstand one damage from enemies. In the power-up state, Mario also can destroy some types of tiles by jumping and bumping his head against them. However, if Mario falls into the hole or gameplay exceeds the time limit, the player loses even when Mario is in the power-up state. Regardless of the power-up state, Mario can defeat enemies by stomping on them, as illustrated in Fig. 3.4.

A Super Mario level contains a two-dimensional structure and various types of tiles. The level can be divided by a set of fixed-size tile patterns.

3.2.1 Level components

In a Super Mario level, there are various types of tiles, which are either obstacles or useful items for players. Obstacles include enemies, holes, and walls. Useful items are coins and mushrooms. Except for the pipe tiles, each tile type works as designed in the original version of Super Mario Bros. This dissertation also excludes some types of tiles, enemies, and complicated gimmicks for research purposes. Table 3.2 summarizes the tiles and their functions used in this dissertation.

Mario cannot pass through wall type of tiles. Only a destroyed brick tile is passable. A brick tile can be destroyed when the power-up Mario jumps and hits his head on it. Any objects that fall into a hole are destroyed, including Mario. The layout of wall and hole tiles affects the gameplay. The level is not playable if it contains too high stacks of wall tiles or too wide hole that Mario cannot jump through them. Enemy tiles move to the left or right, and Mario gets damaged when he physically contacts them. Overall, enemies and holes are the main elements that affect the difficulty of levels.

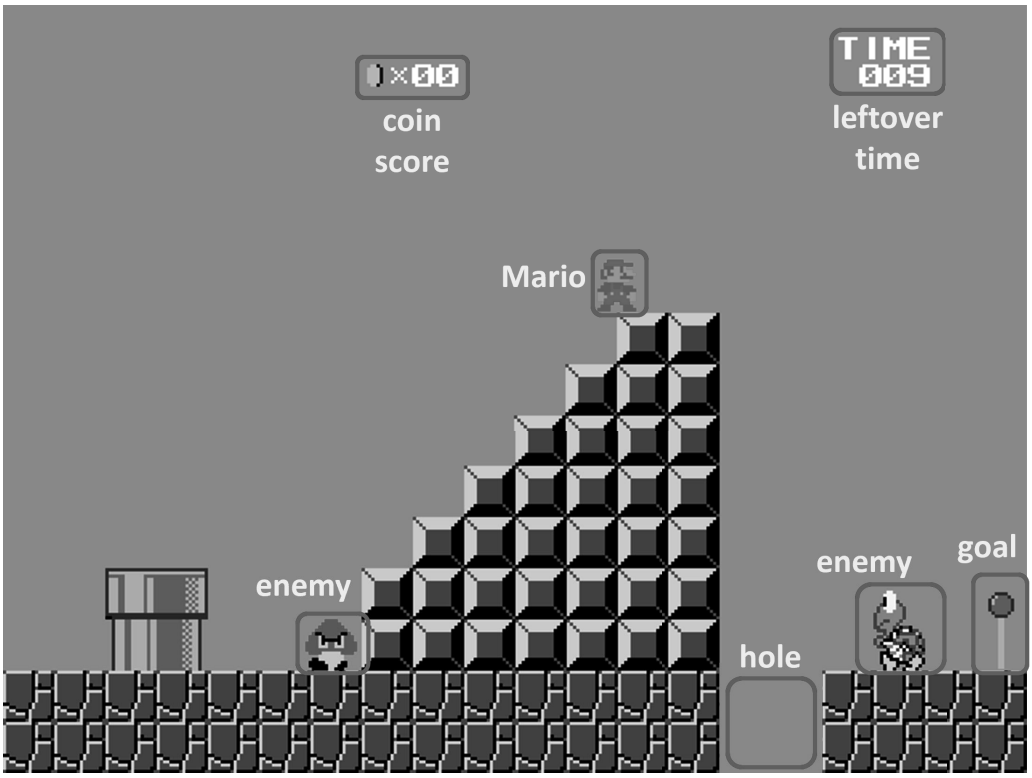


Figure 3.3: A example scene of the Super Mario. The objective is to reach the goal placed in the right-most side in time. The time displayed in the upper right shows the time remaining in the level. The coin score refers to the coins that Mario has acquired during level play.

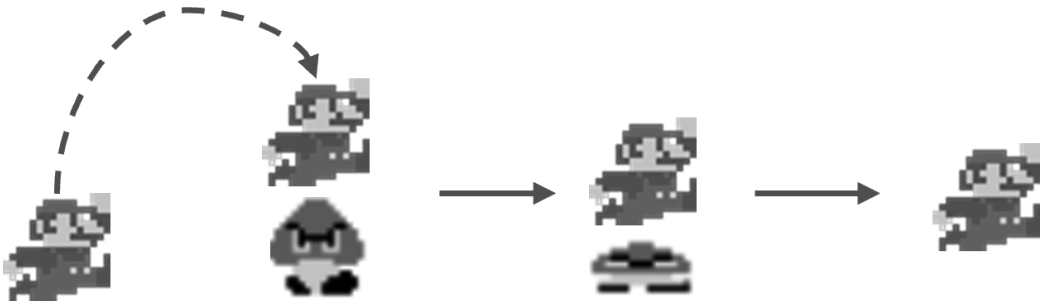
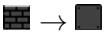


Figure 3.4: A series of processes in which Mario defeats an enemy by stomping on them.

Table 3.2: Description of Mario tiles used in this dissertation.

Tile	Type	Description	Destructible
 (brick)	wall	it does not function	o (power-up)
 (coin brick)	wall	if Mario bumps head on it Mario obtains a coin and it becomes empty block	x
 (coin)	item	if Mario touches it Mario obtains a coin	o (touch)
 (coin box)	wall	if Mario bumps head on it Mario obtains a coin and it becomes empty block	x
 (item box)	wall	if Mario bumps head on it it generates a mushroom and it becomes empty block	x
 (empty block)	wall	it does not function	x
 (block)	wall	it does not function	x
 (block)	ground	it does not function only place on the ground	x
 (pipe)	wall	it does not function	x
 (hole)	hole	if Mario falls into a hole Game is over	x
 (goomba)	enemy	if Mario touches it Mario get damage. Goomba move left or right side	o (stomp) (fall into a hole)
 (koopa)	enemy	if Mario touches it Mario get damage. Koopa move left or right side it becomes a shell when it is stomped	x
 (shell)	enemy	if Mario touches or stomps it shell is fired to the opposite way fired shell remove destructible tiles return to koopa after some time	o (fall into a hole)
 (goal)	etc	if Mario touches this goal level is clear	x

3.2.2 Desirable levels

The history of the Super Mario series has been ongoing since 1985 until nowadays. As with other popular games, diverse groups of players enjoy the Mario series. As much as the appropriate difficulty is a crucial factor in turn-based RPGs, it is also essential in Super Mario. Different level difficulties are required for entertaining various groups of players. An example can be found in "Super Mario Maker 2" [58], one of the games in the Super Mario series. The game employs a level creation system that players can enjoy levels generated by other users. Players can choose level difficulty based on their preferences, including Easy, Normal, Expert, and Super Expert. The difficulties of levels are determined based on the users' play logs.

Another crucial factor in designing levels is non-monotony. A pattern requires a particular sequence of actions to pass through. Levels with redundant patterns require repetitive series of actions. It is undesirable if players need to input the same series of actions continuously because they will feel monotonous. Therefore, in this research, we assume that locally similar patterns are monotonous and deteriorate players' enjoyments. This dissertation summarizes the characteristics of desired levels in Super Mario as follows.

- The level should be appropriately difficult to suit each group of players. Because enemy and hole tiles mainly affect the difficulty of levels, they should be well-placed. The difficulty of the level can be measured using play logs.
- Sequences of actions (i.e., how a level is played) affect the entertainment of Super Mario. If similar patterns are locally placed, players have to make repetitive sequences of actions in short times. Therefore, levels should be designed so that players can try various sequences.

3.2.3 Game platform

This dissertation uses cloned versions of Super Mario with reduced features (e.g., dash). The removed features are excluded because they either exist in a few specific levels or are too complicated. Two clone platforms are employed, *python Mario* [59] and *Mario AI Framework* [60], and implemented in Python and Java, respectively.

Python Mario is a modified clone based on the Super-Mario-Python [59]. An additional level creation system, machine learning-related components, write/read levels, and experiment supporter functions are included. Mario AI Framework supports many functions related to artificial intelligence research, including AI agents and level generators. In this dissertation, only functions

Table 3.3: Comparison of commercial Super Mario and two Clone Super Mario configurations.

	Super Mario Bros.	Python Mario	Mario AI Framework
Level structure	A level consists of $15 \times n$ structure The bottom 2 rows are only for ground or hole.		
Level objective	Reach the goal placed in the right-most side		
Power-up state	Mushroom and Fire flower (can shoot fire ball) power-up	Only power-up from mushroom is used	
coin score	100 coins gives Mario an extra life	Obtain coin score but does not give an extra life	
Actions	Jump, dash, move, fire	Dash and fire are disabled	
Usage for research	-	Level generation, machine learning related, , write/read levels, and etc	Gameplay from AI agents

related to AI agents are used. Table 3.3 summarizes the comparison between the commercial Super Mario and the two Super Mario clones.

3.3 Summary of unique features of the target games

In this dissertation, two distinct game genres inheriting different characteristics, including turn-based RPGs and Super Mario, are employed to generate different types of content and stages/levels. This section clarifies the characteristics of the two games and their differences. Table 3.4 summarizes the comparison of core distinctions between two games. The most significant difference is the time at which players decide on actions. In the turn-based RPG, the action is decided every turn. While in Super Mario, the action is input in each frame. Typically, the required time cost of the one level/stage's gameplay is t turn and f frame, usually t significantly lower than f. One frame of Super Mario normally lasts around 0.017 seconds (60 frames per second) [61]. One level requires roughly 30-60 seconds to clear; thus, f is around 1,800-3,600 frames. In most turn-based RPGs, one battle rarely takes more than ten turns.

Another key difference is that the stage is one-dimensional composition, while the level is two-dimensional. In the simplified turn-based RPG, one-way events need to be generated, which is relatively simple. Whereas placing tiles on a two-dimensional level is more complex.

Table 3.4: Comparison of turn-based RPGs and Super Mario

	Turn-based RPGs	Super Mario
A stage/level composition	One-dimensional series of events	Two-dimensional layouts of tiles
Number of controllable characters	>=1	1
Action decision	Turn-based (discrete time)	Frame-based (continuous time)
Possible action	Command action (attack, magic, item, etc)	Move (right left), jump, etc

Chapter 4

PCG via RL considering quality and diversity

Before discussing the stages/levels generation methods, Section 4.1 firstly describes general concepts for applying RL to PCG. Second, Section 4.2 introduces a method called *virtual simulation* to help improve the quality of generated stages/levels. Virtual simulated rewards are given to intermediate actions during training. Third, Section 4.3 presents two methods for increasing the diversity of generated stages/levels. One is the *randomized initialization*, which generates stages/levels from randomly initialized events/patterns. The other is the *diversity-aware greedy policy*, which generates diverse stages/levels while maintaining quality. Last, a brief comparison of Khalifa et al.'s work [22], and ours is made in Section 4.4.

4.1 Application of RL to PCG

As a novel AI-based approach for PCG, we proposed to apply reinforcement learning (RL) [20][21][23]. Table 4.1 compares PCGML, search-based PCG, and RL from different aspects. For example, PCGML requires training data, while the other two require evaluation functions to tell how promising the generated content looks. Both PCGML and RL need a relatively long learning time. However, once the learning is finished, the generation costs are low. In contrast, search-based PCG does not require a learning process but consumes time when generating content, which is more challenging for online generation. To sum up, RL's advantages include low generation costs and no need to prepare training data.

To apply RL, we need to formulate PCG problems as MDP. The following explains how MDP elements (i.e., state, action, transition probability

Table 4.1: Comparison between three kinds of PCG methods

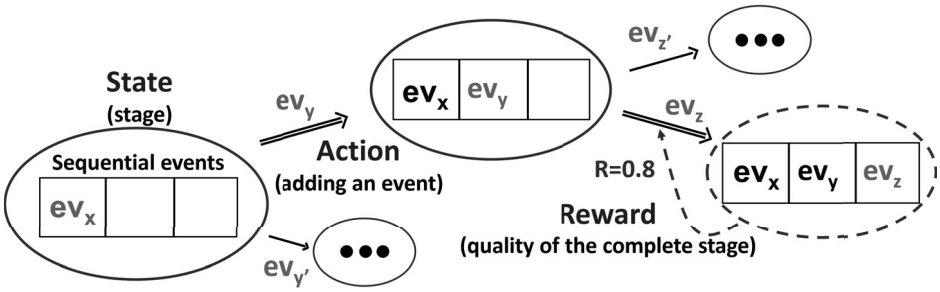
	Training data	Evaluation function	Learning cost	Generation cost
PCGML	necessary	–	high	low
Search-based PCG (GA)	–	necessary	–	high
Reinforcement Learning	–	necessary	high	low

function, and reward function) may look like using an example shown in Fig. 4.1, which tries to generate stages/levels consisting of three sequential events/patterns. At the initial state, the first event/pattern has been determined while the rest two remain blank. An action is to add an event/pattern to the next blank one. For simplicity of discussions, let the state transition be deterministic. Namely, when adding an event/pattern, the next blank event/pattern always becomes that one. Thus, states in this PCG problem are stages/levels with different event/pattern combinations, including blank events/patterns. After all three events/patterns are decided, we can use an evaluation function to get the quality of the complete stage/level, which can be used as the reward function in the MDP.

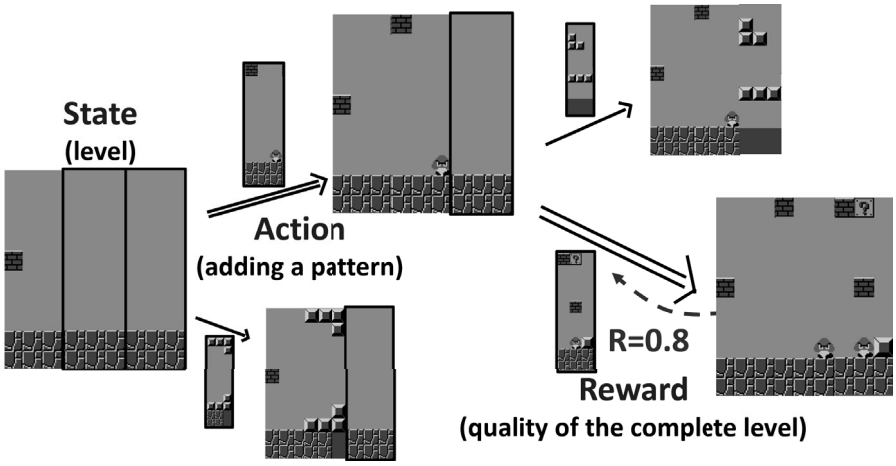
We believe that the ideas of formulating PCG into MDP and applying RL are general and can handle many game genres. As long as the MDP formulation of a PCG problem is finished, RL algorithms can be applied to learn good policy (i.e., what actions to take for given states) based on the provided reward function. After the learning, the policy serves as the content generator at relatively low generation costs.

4.2 Method for improving quality

With the application of MDP, RL can be applied to learn to generate stages/levels. However, simple RL algorithms are insufficient to obtain high-quality content. The evaluation function proposed in this dissertation has a small signal with intermediate rewards, i.e., 0 in the turn-based RPG and 0 to 0.01 in Super Mario. Its details will be discussed later in Sections 5.1.4 and 6.3.4. One concern is that insufficient reward signals or credits are delivered to the intermediate actions because of the zero or small signal. This is an instance of the temporal credit assignment problem [44], a long-studied subfield of RL. Conceptually speaking, the problem aims to figure out how each action



(a)



(b)

Figure 4.1: Illustrations of the MDP for generating stages/levels consisting of three sequential events/patterns as an example.

among a sequence of actions influences the final outcome. When the reward signal is sparse or delayed, RL may get stuck in local optima, especially for deep neural networks.

In this dissertation, virtual simulation (VS) is proposed to provide intermediate actions with rewards by a Monte-Carlo method, where the ideas are similar to SeqGAN [50]. For each intermediate action, the algorithm applies the current policy with an exploration policy, such as ϵ-greedy, to generate several completed stages/levels. The average evaluation of these stages/levels then serves as the immediate reward of the action. The experimental details and settings are in Sections 5.2.1 and 6.6.1. By doing this, we do not need extra efforts to design immediate rewards for intermediate actions. The method is expected to be more time-consuming for learning, but the offline cost is not a significant problem in PCG. One of the major benefits of RL is that even if learning takes a long time, it is possible to provide content immediately once the learning is over, i.e., online generation.

4.3 Methods for increasing diversity

One primary goal of PCG is to automatically (or semi-automatically) provide players with diverse content even for the same game. For example, after clearing a stage/level, players may want to try different settings other than the same one, or they may get bored soon. It is better that the method can generate several diverse and good stages/levels instead of only the best one. However, this goal is not considered in the proposed RL as long as the diversity of stages is not included in the reward function. Our designed evaluation function indicates how good one given stage/level is. To consider diversity, several stages/levels should be compared at once, which is expected to be complicated. Instead of revising the evaluation function, we propose two ways to address the diversity issue.

4.3.1 Randomized initialization

We propose to give randomness to several beginning events/patterns for each stage/level instead of an empty stage/level as the first method to address diversity[1] In other words, the matrices of an empty stage/level are assigned with some random values as first elements and then used as the initial states of stage/level generation. We consider this method to be able to balance the quality and the diversity of stages/levels to some extent in the following

[1]The method was called *random initial stage* and *randomized event initialization* in our previous work [21] [23].

senses. For an extreme end, assume that no events/patterns are randomly initialized and a model has been well trained by RL. When the model is used to generate stages/levels (for players to play), it is common to select the best actions (i.e., event parameters or patterns). As a result, only one good stage/level can be generated. For the other extreme end, when all event parameters/patterns are randomly assigned, the stages/levels will have a high diversity but are highly likely to have low quality.

A reasonable idea in between is to randomly initialize several events/patterns and make the RL model finish the rest. During training, RL algorithms are also provided with such randomly initialized stages/levels as the initial states/levels. We expect that RL algorithms can learn how to generate good stages/levels from random initialization. In this way, with diverse initial stages/levels, we can generate diverse and good stages/levels. However, in some cases, even if the initial stages/levels are different, the later parts may end up being similar, especially when long stages/levels are generated. Lack of diversity can be alleviated by randomly initializing more events/patterns, while this may result in stages/levels with low evaluations. For this method, the number of events/patterns to initialize determines the trade-off between the quality and diversity of stages/levels, and detailed results will be shown in Sections 5.2.3 and 6.6.2.

4.3.2 Diversity-aware greedy policy

Getting diversity from the initial stages/levels is not enough as the diversity only depends on the initial states. As the second method to address diversity, we propose to sample *not-bad-but-distant* actions based on Q-values, which we call diversity-aware greedy policy (DAGP)[2]. Compared to the greedy policy that always takes the best actions, the proposed method may take worse actions where stochastic noise is introduced for increasing stage/level diversity. In this way, even when the generation starts with the same initial stage/level, we can still get quite different stages.

The most critical part of DAGP lies in how to select not-bad-but-distant actions. If improper noise is introduced, stages/levels with low evaluations are likely to be generated. The method has two prerequisites: (a) good actions distant from the best-evaluated ones exist, and (b) the Q-values of those good actions can be estimated with some accuracy. The experiments in Section 5.2.2 will confirm that the two prerequisites are indeed satisfied. From the results, we conclude that it is possible to generate diverse and high-quality stages/levels based on Q-values and present a noise introduction

[2]The method was called *stochastic noise policy* in our previous work [21].

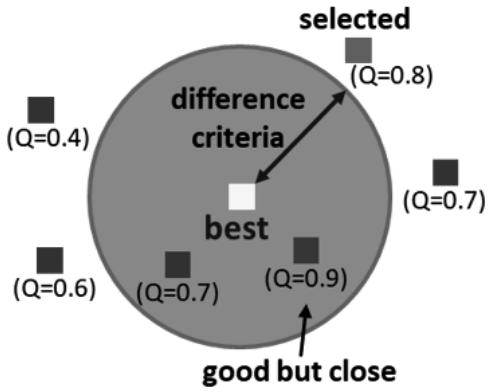


Figure 4.2: Example of choosing an action that is not-bad-but-distant by DAGP.

algorithm to select such not-bad-but-distant actions as follows.

i) Decide the stage structure /level length and train the model as usual.

ii) Determine where and how noise is introduced (noise event/pattern). The place can be fixed in advance or randomly selected during generation. Also, determine the number of candidate events/patterns n and the criteria for the difference between a candidate and the best (i.e., Euclidean distance d for event parameters and KL-divergence $kl_{pattern}$ for patterns).

iii) Except for the noise events/patterns, greedily decide actions (event parameters/patterns) according to the learned policy. In the case of noise events/patterns, an action is decided as shown in Fig. 4.2. First, generate n random actions as candidates. Candidates in the gray zone are rejected because it does not meet the criteria, as it means that they are close to the best action (i.e., Euclidean distance is less than d or KL-divergence is less than $kl_{pattern}$). If there is no candidate, then the action is greedily decided. Among the remaining candidates, select the one with the highest Q-value.

4.4 Comparison with Khalifa et al.'s work

Khalifa et al. [22] also applied RL to PCG to generate game levels and formulated the problem into MDP. In the comparison, theirs are referred to

as levels following their paper and ours as stages and levels. Their levels were represented by 2D integer matrices, indicating the layouts of the maps where different integers stand for different objects. The layouts were initialized randomly, and the actions were to *change* the objects of tiles in the maps. We also represent stages/levels by matrices while the meaning is different. In both designs, the actions modify stages/levels, where their approach changes existing values and ours assigns new values. They proposed three ways to locate the value to change, and one of which following a predefined order is similar to ours.

Their reward functions were manually designed to reflect whether the levels got closer to the goals of the game. For example, since PacMan has only one player, adding a player object when there is none receives positive rewards. Their goal was to generate playable levels that obey the game rules, while ours focuses on the difficulty with players' engagements. They required an additional function to determine whether the goals were reached so that the generation process could be terminated.

To achieve the diversity of levels, they set a parameter called change percentage, which limited the numbers of tiles that could be changed from the initial levels. Smaller change percentages were suggested since too-high values were expected to cause the generator to override most of the initial levels aiming at few optimal solutions. We also set random initial values, though our approaches differ in terms of theirs. In addition, we introduce noise to choose not-bad-but-distant actions to make the generated stages/levels more diverse.

Chapter 5

Generation of turn-based RPG stages

In this chapter, we present the MDP formulation of our platform in Section 5.1, which is required when applying RL for stage generation. The MDP formulation includes state and action representations, state transition function, and reward function. For the reward function, we design an evaluation function to rate completed stages. Next, experimental results and discussions are provided in Section 5.2. Finally, Section 5.3 concludes the overall chapter.

5.1 Markov Decision Process

This section explains how to formulate stages generation in our turn-based RPG into MDP.

5.1.1 State representation: incomplete stage

Each stage in our platform is represented by a real number matrix, indicating a series of one-way battle events and recovery events, with a boss event at the end. An empty stage is represented by a zero matrix filled with zeros. In the matrices, each column contains three values, i.e., enemy's HP, enemy's ATK, and player character's recovery. In more detail, the former two form a battle event, and the last one forms a recovery event. Fig. 5.1 is a simple example consisting of five events: enemy-enemy-recovery-enemy-boss.

We represent enemies' HP and ATK values by integers in some reasonable ranges. In Fig. 5.1, the second enemy's HP is 70, while the range is 20 to 120. Similarly, the ATK is 10, while the range is 5 to 30. For bosses, both

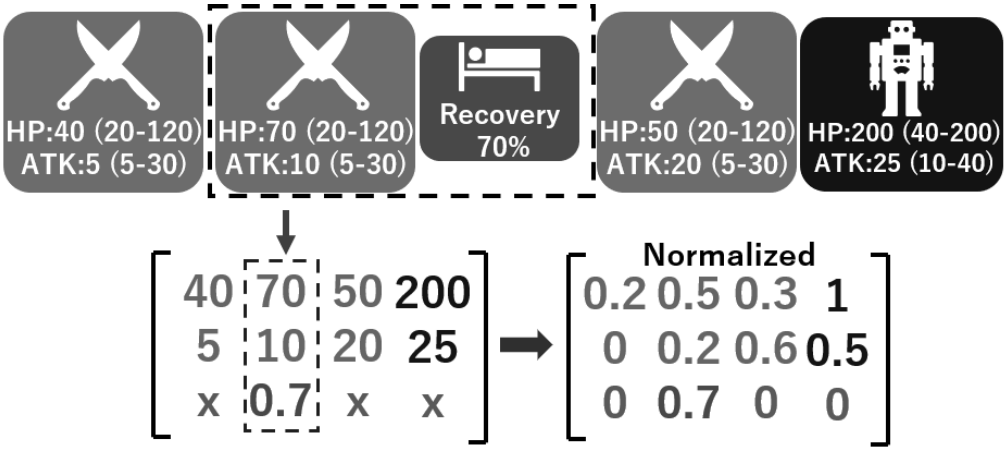


Figure 5.1: Example of converting the battle-battle-recovery-battle-boss stage to a 3×4 matrix.

lower bounds and upper bounds for HP and ATK are usually higher than those of enemies. To make our representation generalize to any ranges, we normalize the values into $[0, 1]$ according to the given ranges. For example, HP of 70 in the range of 20 to 120 is normalized to (70-20)/(120-20)=0.5.

As for the third element in a column, the recovery event, the value ranging from 0 to 1 represents the player character's recovery rate relative to max HP. For example, assuming that the player character's HP/max HP is 20/100, a recovery rate of 0.7 makes the player character's HP become 20+0.7×100=90. The max HP bounds the recovery, i.e., even with a recovery rate of 1.0, the player character's HP becomes 100 instead of 120 after the recovery event.

It can be seen that even in the same range of $[0, 1]$, the values mean quite different things between battle events and recovery events. Also, battle events require two values, while recovery events only require one. Considering the topological structure of stage representation, we attach one recovery event after every battle event. However, it is unusual that each battle event follows a recovery event in turn-based RPG stages. For battles without recovery events (the 1st and 3rd columns of the matrix in Fig. 5.1), the recovery values are set to 0. In this way, a series of battle-recovery events, including battles without immediate recovery events and the final boss, can be represented by concatenating columns with three real number values ranged in $[0, 1]$. A stage containing b battle events is represented by a $3 \times (b + 1)$ matrix, the *+1* for boss.

Figure 5.2: DQN for stage generation in this dissertation.

5.1.2 Action representation and RL

Actions in our platform are defined to fill in the stage matrices sequentially. In other words, an action decides one or some real number values of the given incomplete stage matrix. Many classical RL algorithms, including deep Q-network (DQN) [51], assume discrete actions, i.e., action spaces being finite sets. Meanwhile, algorithms coping with continuous actions, such as deep deterministic policy gradient (DDPG) [62], are also proposed. We define both discrete and continuous action representations so that our platform is applicable to both kinds of RL algorithms.

For discrete actions, we divide the range of $[0, 1]$ into k sections, and thus, there are $k + 1$ actions. Each action stands for the lower bound of the corresponding section, except that the $(k + 1)$st action stands for the overall upper bound, i.e., 1. Assuming $k = 100$, the 101 actions are 0, 0.01, 0.02, 0.03, ..., and 1. DQN takes an incomplete stage matrix as the input and outputs $k + 1$ action values (Q-values), as shown in Fig. 5.2. The $k + 1$ Q-values represent how the $k + 1$ possibilities of the next parameter look promising for the given stage. The interpretation of *the next* may vary in different implementation. In this dissertation, we fix the order to 1st enemy's HP $\rightarrow$ 1st enemy's ATK $\rightarrow$ 1st recovery rate $\rightarrow$ ... $\rightarrow$ the boss's ATK. In this manner, one stage parameter is determined at a time by selecting the one with the highest Q-value or by methods that introduce exploration such as ϵ-greedy.

Considering that the three values in the same column may influence each other, it is more natural to output them at a time. However, the case of discrete actions falls into the curse of dimensionality. In the example, the output increases to $101^3 = 1030301$, an extremely large value that may cause problems during learning. Continuous action algorithms directly output numerical values in the given ranges. Thus, it is easy to output three values at once, representing one column. This is done by the *actor* of DDPG, while the

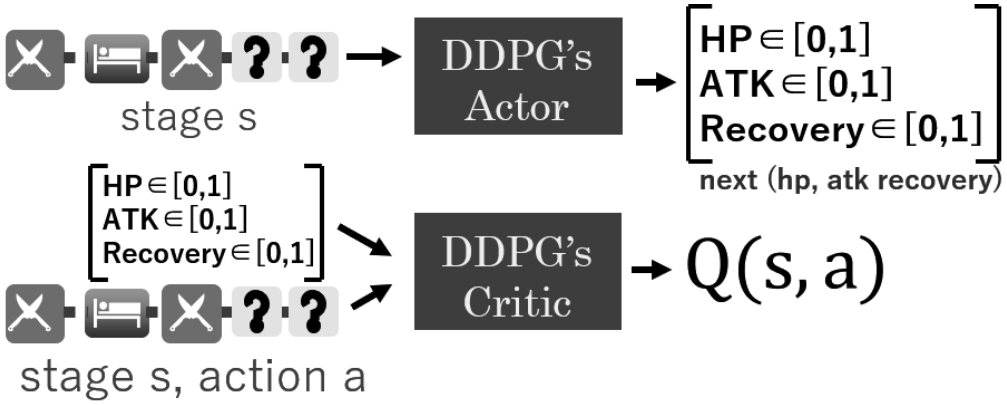


Figure 5.3: DDPG for stage generation in this dissertation.

Q-value is evaluated by another model called the *critic*, as shown in Fig. 5.3. More specifically, the actor takes an incomplete stage matrix as the input and outputs three real number values. For example, (0.5, 0, 0.8) means a 50% of enemy's HP in the given range, a minimum value of enemy's ATK, and a recovery rate of 0.8. The critic then takes the same stage matrix and the actor's action as the input and outputs the Q-value.

5.1.3 State transition

Given an incomplete stage matrix as a state and a real number value (or a column) as an action, the matrix is always filled in by the given value(s) exactly. For example, given a zero matrix and a value of 0.5, the first element in the first column of the matrix becomes 0.5, meaning that the enemy in the first battle event is assigned an HP of 50% in the given range. Stage generation terminates when the matrices are fully filled.

5.1.4 Reward function

In this dissertation, our purpose is to create stage generators that can generate enjoyable stages. We define a stage evaluation function and use it as the reward function of the MDP. The evaluation function focuses on player engagement from seven perspectives, where the major one is the appropriateness of difficulty.

Although there are various possibilities to define the quality of stages, the difficulty is considered one of the most important factors that influences player engagement [63] [64] [65]. Also, the difficulty assessment is different from game to game. As a simple example, we define a strategy to be a

sequence of action selections from the beginning to the end of a stage and then use the ratio of winning strategies among all possible strategies as the difficulty of our turn-based RPG stages. Generally speaking, good stages should have moderate ratios of winning strategies to avoid situations like players cannot win whatever they do or can win whatever they do.

In more detail, players have two available actions, attack and retreat, for each battle event in our platform. When there are $b + 1$ battle events including a boss event, the possible number of strategies is 2^b as it is not allowed to retreat in the boss battle. In this condition, it is evident and deterministic whether a boss is beatable or not. We calculate the ratio of defeating the boss among all strategies as the winning rate to represent the difficulty of a stage.

Deciding the proper target winning rate on our game setting could be an issue. However, the preferred Mario series until the present day varies depending on the games, and it is difficult to say which values are the most reasonable, which we leave out of the scope of this dissertation. In this study, we assume 30% to be the most favorable winning rate and leave it as a tunable parameter [1]. In our evaluation function, as shown in Fig. 5.4, the scores are 0 when the winning rates are higher than 60% and linearly increase from 0 to 1 in the range of 0%-60% as the winning rates go closer to 30%.

There are many other entertaining factors in addition to difficulty, and those factors are different for each player [66]. Accurately predicting the satisfaction of human players is not the subject of this dissertation, so we only consider some common factors, such as dramatic surviving and tough wins. The following paragraph describes the details of the evaluation function. Although this specific evaluation function is applied in our method, it can be replaced by any other calculable evaluation functions. Similar to search-based PCG, adaptive generation can be achieved by adjusting the evaluation function.

The employed evaluation function is weighted from seven sub-functions, $f(x) = \Sigma_{i=1}^{7} w_i f_i(x)$, with considering the following features of stage x: The number of events (n_{events}), the number of winning strategies (n_{win}), the number of the strategies that retreat from the i^{th} enemy ($n_{retreat_i}$), the number of strategies that have dramatic surviving moments ($n_{dramatic}$), the number of winning strategies that have monotonous actions (n_{mono}), character recovery rate at the i^{th} recovery event, ($c_{recover_i}$), the i^{th} enemy's normalized ATK and

[1] Some readers may consider that a winning rate of 30% results in too difficult stages. However, our definition of the winning rate implies that the player selects each action with equal probabilities (i.e., a random player). Rational players usually do not play randomly, which we expect to have higher chances to clear the stages.

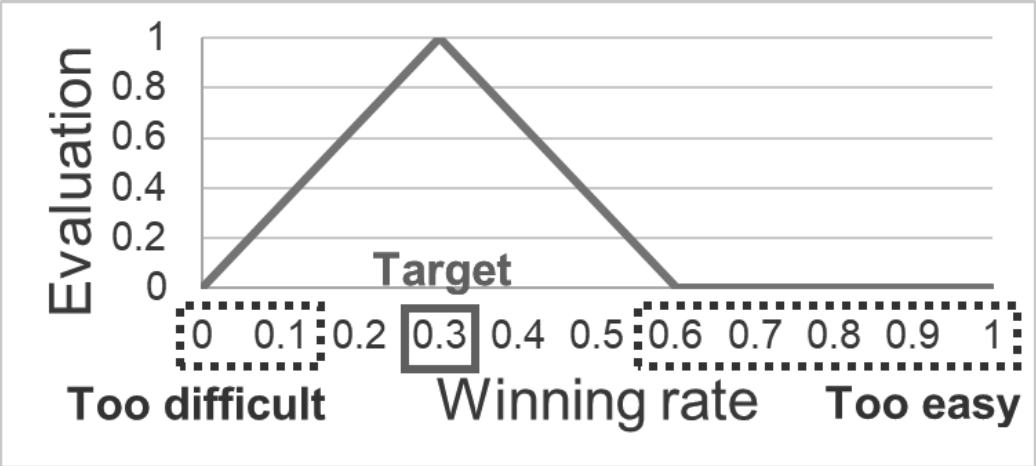


Figure 5.4: An evaluation based on a winning rate. (x-axis: winning rate, y-axis: evaluation based on the winning rate)

HP within [0, 1] (e_{ATK_i} and e_{HP_i}).

- f_1: the winning rate soundness. The winning rate should not be too low nor too high, as shown in Fig. 5.4, $w_1 = 0.4$.
- f_2: bonus for dramatic surviving. Players often feel pleasant when surviving dramatically from a crisis (when reaching the i^{th} recovery event with an HP $\leq 0.3\times$(max HP) and $c_{recover_i} \geq 0.5$). $n_{dramatic}/n_{win}$, $w_2 = 0.1$.
- f_3: bonus for moderate parameter ranges. Enemies or recovery events within moderate ranges (enemy parameters within [0.05, 0.95] and recovery rates within [0.2, 0.8]) seem more natural. $(\Sigma_i g(c_{recover_i}, 0.2) + \Sigma_i \min(g(e_{ATK_i}, 0.05), g(e_{HP_i}, 0.05)))/n_{event}$, where $g(x, y) = \min(x/y, (1-x)/y, 1)$, $w_3 = 0.2$.
- f_4: bonus for tough wins. If the character after defeating the boss has a low HP ($\leq 0.4\times$(max HP)), it means that the stage is challenging. Average of $\min((1 - \text{final HP}(\%))/0.6, 1)$ in winning strategies, $w_4 = 0.1$.
- f_5: penalty on early escapes. If the strategy of escaping in the early battles are effective, the game flow may be considered unnatural. $1 - (3n_{retreat_1} + 2n_{retreat_2} + n_{retreat_3})/6n_{win}$, $w_5 = 0.05$.

- f_6: penalty on weak enemies in later phases. It is irrational that later enemies are too weak. $1 - (0.7 - \min(0.7, (\Sigma_{i \in \{\text{last two enemies}\}}(e_{ATK_i} + e_{HP_i}))/4))/0.7$, $w_6 = 0.1$.
- f_7: penalty on monotonous strategies. Winning a stage with monotonous strategies, i.e., only attacks or only retreats, can make players boring. $1 - 0.5 \times n_{mono}$, $w_7 = 0.05$.

5.2 Experiments and discussions

In this section, the results of generating stages with high quality are included in Section 5.2.1. The learned Q-values are analyzed in Section 5.2.2. Finally, the results of generating diverse stages are presented in Section 5.2.3.

5.2.1 High-quality stage generation

We employed two RL algorithms with different action spaces to generate turn-based RPG stages by the proposed PCG approach. More specifically, the two RL algorithms were DQN [51] for discrete actions and DDPG [62] for continuous actions. We also included two non-RL methods for comparison, though different kinds of methods have different assumptions and advantages, as discussed in Section 4.1, and might not be directly comparable. The random generation method decided event parameters by a uniform distribution within $[0, 1]$. The supervised learning (SL) method took an incomplete stage as the input and decided the parameters in the next column (i.e., enemy's HP, enemy's ATK, player's recovery rate), similar to the actor of DDPG.

As described in Section 5.1.1, we represented turn-based RPG stages by matrices of three rows and $b+1$ columns, which contain b battle events and a boss battle. For the third row, if there was no recovery event after the battle, the value was fixed to 0. The number of battles b and the stage structure (i.e., the arrangement of battle and recovery events) were determined in advance and then fixed.

Experimental setup

We conducted the experiments under the following settings.

- The codes were implemented in Python 3.6, and the used libraries and the machines for experiments are listed in Table 5.1. The network settings of DQN, DDPG, and VS-DDPG are listed in Table 5.2, where

Conv x means a convolutional layer with x filters of size 1×3 and FC x means a fully-connected layer with x nodes.

- The composition of a stage was the one from Fig. 5.5a, and a longer stage (Fig. 5.5b) was used to compare the effect of virtual simulation. Each stage was represented by 3×7 and 3×9 matrices, respectively. Among these, the event parameters in the first one or two columns were determined randomly, which were initial states.

- As described in Section 5.1.2, the significant difference between DQN and DDPG is the action space. DQN was for discrete actions, and DDPG for continuous actions. In our setting, DQN had 101 actions[2], each representing a possible setting (0, 0.01, ..., 1) for the next event parameter (enemy HP, enemy ATK, or player's recovery rate). DDPG's actor output three real values in $[0, 1]$ at once for the event parameters in the same column. The designs led to a minor difference in how many times of input/output were required to complete a stage. For the example of Fig. 5.5a, assume that one column of the stage matrix is initialized. DQN requires 3×6 because it decides event parameters sequentially one at a time (even when there is no recovery event). In contrast, DDPG only requires 6 because it decides one column at a time.

- Virtual simulation presented in Section 4.2 was applied to DDPG (abbr. VS-DDPG). The number of virtual simulating episodes was 5, and virtual simulation was applied from the beginning of the training.

- SL had the same network structure as the actor of DDPG, except that batch normalization layers were added to prevent overfitting. In practice, preparing training data with high quality is a critical issue for SL methods. In this experiment, we simply employed the random generation method and only collected stages whose evaluations were over 0.7. It cost 10 hours to obtain 20000 nine-event stages (Fig. 5.5a). For more complex games, we expected the preparation of training data to be more costly.

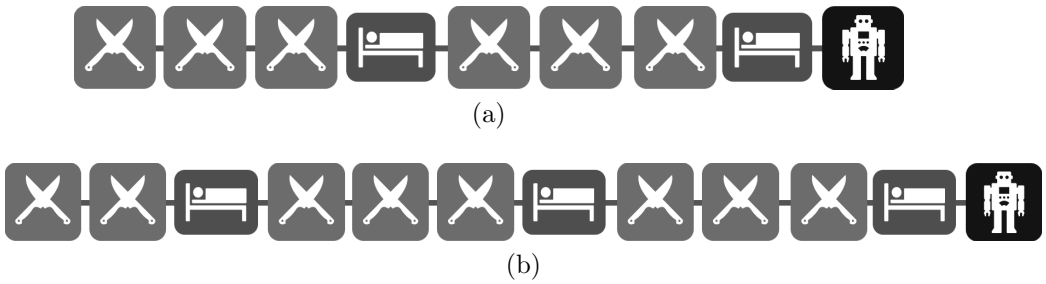
[2]When doing discretization, proper settings of granularity (i.e., number of actions) depend greatly on the ranges of event parameters. In some preliminary experiments, we obtained poor results if the number of actions was too low (e.g., 11). Considering that discretization is a general and challenging issue, detailed investigations are made out of this dissertation, which we focus more on that the proposed approach is applicable to both discrete and continuous actions.

Table 5.1: Used tools and version

Tool	Spec (version)		
GPU	NVDIA GeForce GTX 1070		
CPU	Intel Core i7-7700 3.60GHz		
RAM	16GB	tensorflow-gpu	1.10.0
CUDA and cuDNN	8.0 and 6.0.21	Keras	2.2.2

Table 5.2: DQN and DDPG setup

Parameter	Value	
	DQN	DDPG & VS-DDPG
Layers	Conv 36→64→ FC 128→256→256→256	Conv 256→256→256→ FC 128→128→128
Memory size	300000	
Learning rate	0.25×10^{-4}	Actor: 1×10^{-5} Critic: 1×10^{-4}
Target network	hard update (3000)	soft update (0.001)
Batch size	128	64
Discount factor	0.9	
Exploration	ϵ-greedy ($1 \rightarrow 0.1$)	OU noise with ϵ-greedy ($1 \rightarrow 0.1$)



(a)

(b)

Figure 5.5: Composition of the stage having (a) nine and (b) twelve events, where each square represents battle (red), recovery (blue), and boss (black) events.

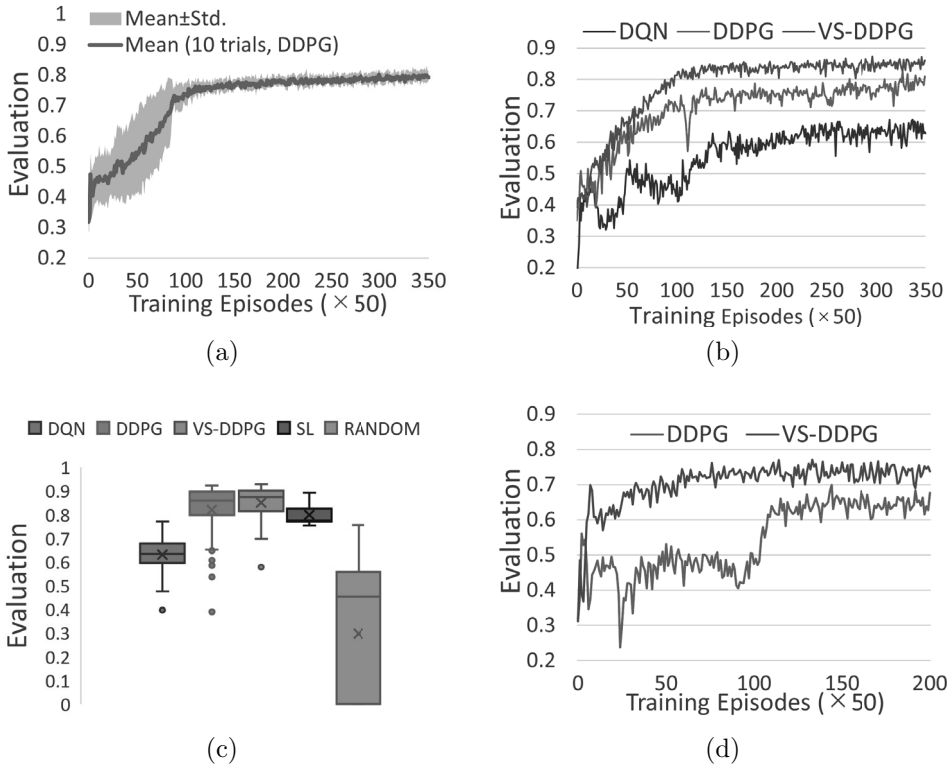


Figure 5.6: The average evaluations of 50 stages: (a) DDPG's 10-trial training curve of the 9-event stage, (b) DQN, DDPG, VS-DDPG's training curves of the 9-event stage, converging around 0.65, 0.83, 0.87, (c) box plot of the 9-event stage after training, and (d) DDPG and VS-DDPG's training curves of the 12-event stage, converging around 0.65 and 0.75. For training curves, the action selection involved exploration, such as ϵ-greedy.

Result

Since randomness was involved in the training, we investigated how training curves were influenced and ran DDPG under the same setting for 10 trials. DDPG was trained for 50×350=17500 episodes to generate 9-event stages. Fig. 5.6a shows the mean evaluations and the standard deviations of the 10 trials. Each data point was the average evaluation from 50 episodes, where one episode means generating one stage from an initial one. Note that the evaluations were collected from stages generated during training, which means that the exploration was also involved. The standard deviations before 50×100 episodes were relatively high, mainly because of higher exploration (ϵ-greedy and OU-noise) in the early phases of training. In addition, we tried

several network structures and hyperparameters, obtaining similar results. In the following experiments, we only show the results of one trial under the settings described in Section 5.2.1.

Fig. 5.6b shows the training curves of DQN, DDPG, and VS-DDPG on generating 9-event stages. The results showed that both DDPG and VS-DDPG were better than DQN (under our experiment settings). Especially, VS-DDPG converged at the highest evaluation value of 0.87. Fig. 5.6c shows the box plot of evaluation values of 50 stages after training, including two baselines, random generation and SL. Stages made by the random generation had a wide range of evaluation values, and most stages were lowly evaluted. As for the remaining four, (VS-)DDPG generated stages with the highest average evaluations, though some bad stages were also obtained. SL could generate highly evaluated stages most consistently, but the average evaluations were slightly worse than (VS-)DDPG. Assume that we only want to collect stages whose evaluations were over 0.9. Taking DDPG and the random generation as examples, the former could obtain one such stage every 3-4 trials (about 0.2 seconds), while the latter required about 2000 trials (80 seconds). DDPG's speed was about 400 times faster than random generation.

We further conducted an experiment to compare DDPG and VS-DDPG for generating longer stages. The results of the 12-event stages (Fig. 5.5b) are shown in Fig. 5.6d. As expected, VS-DDPG performed obviously better than DDPG as the problem of delayed rewards was considered more serious for longer stages. About the training time of 9- and 12-event stages for 10000 episodes, VS-DDPG took 300 and 2500 minutes, while DDPG took 60 and 160 minutes. Even under the same training time, DDPG could not reach the same evaluation values.

Fig. 5.7 shows two stages generated by random (top) and VS-DDPG (bottom). The upper stage has strong enemies in the early phases with low recovery and somewhat weak enemies in the later phases; hence, it has a lower evaluation value. In contrast, the bottom one is generated from the first two columns of the upper one, but it has a better evaluation value by filling adequate values with implicitly considering the requested conditions by the reward function.

As described above, if an evaluation function is given, the RL model can learn a policy that can generate highly evaluated stages. We have shown that our proposed approach for generating high-quality stages is promising.

5.2.2 The relation between Q-value and evaluation

In the previous section, generating good stages is the main goal. In this section, we further shift the goal to generate diverse stages while maintaining

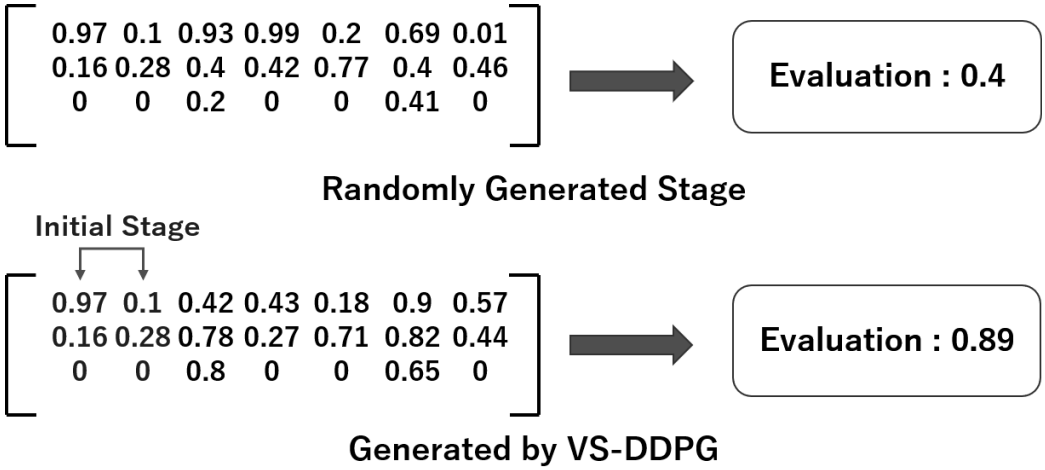


Figure 5.7: Stage examples (top: random generation, bottom: VS-DDPG), where each column represents event parameters of battle and recovery, i.e., HP, ATK, and recovery rate from the top.

quality. The policy of RL generally takes the best-evaluated actions, but when considering diversity, it is better to choose other actions. For actions other than the best-evaluated ones, two questions about evaluations arise: (1) whether a distant action from the best-evaluated one gets an extremely worse evaluation value and (2) whether the Q-values of distant actions are inaccurate due to lack of learning. Therefore, we first compare the distributions of Q-values and evaluation values to confirm the distribution of good actions and how trustworthy Q-values are.

Experimental setup

From Section 5.2.1, VS-DDPG was clearly better than others and was thus used in the rest of the experiments. The stage composition was the same as Fig. 5.5a, and the first one battle event was randomly initialized. The actor of VS-DDPG then completed the stage. The Q-value distribution for each battle event was collected from the critic.

Unlike the critic of VS-DDPG, which can also evaluate incompleted stages, our evaluation function requires completed stages. Thus, we applied the actor of VS-DDPG to finish the rest of the events by selecting the most promising actions and then used the evaluations of such completed stages for incomplete stages. The evaluations were the highest ones the actor could reach.

Distribution of Q-values and Evaluation Values

For VS-DDPG, we obtained the Q-value distribution by sampling actions and inputting them into the critic. An action was represented by a three-dimensional vector of $[0, 1]$ (HP, ATK, recovery rate). To make the results easier to understand, we used the recovery rate from the actor and varied HP and ATK.

Fig. 5.8a (left) shows the distribution of the second battle event's Q-values, with the first battle event randomly initialized. The Q-values are the highest around (0.1, 0.55). When both go too high, which means the enemy is too strong in early stages, the Q-values become lower. The scale of the Q-values is 7.18 to 7.32, which shows the difference gap is small, and the value is overestimated than the evaluation value. Lillicrap et al. [62] have pointed out such overestimations of Q-value when the target problem is complicated.

Fig. 5.8a (right) shows the distribution of the evaluation values of the second battle event. It can be seen that multiple good actions exist. Also, the general trends between Figs. 5.8a left and right are similar. The distributions of the Q-values and evaluation values for the 3rd-7th battle events are drawn in Fig. 5.8b-5.8f in similar ways. The results also demonstrate that the critic of VS-DDPG learns the general trends.

Fig. 5.9 shows the relations between the Q-values normalized to $[0, 1]$ and the evaluation values from Fig. 5.8. The Pearson correlation coefficients were 0.742, 0.787, 0.806, 0.849, 0.383, and 0.884, respectively. Except for Fig. 5.9e, all had highly positive correlations. In other words, the Q-values by the critic were generally trustworthy.

5.2.3 Diverse stage generation

In this section, we first define two indicators to evaluate the diversity of the generated stages and then demonstrate the effectiveness of our approaches proposed in Section 4.3.

Diversity assessment

The first indicator calculates the average squared difference (ASD) between the event parameters of two stages. Higher values of parameter ASD mean that two stages have distant HP, ATK, and recovery rates.

The other is the number of winning strategies that are different in two stages. As introduced in Section 5.1.4, winning strategies are those that can beat the boss. In the experiments, the stages contained six enemy events and a boss event, i.e., Fig. 5.5a. For each enemy event, two actions, attack and

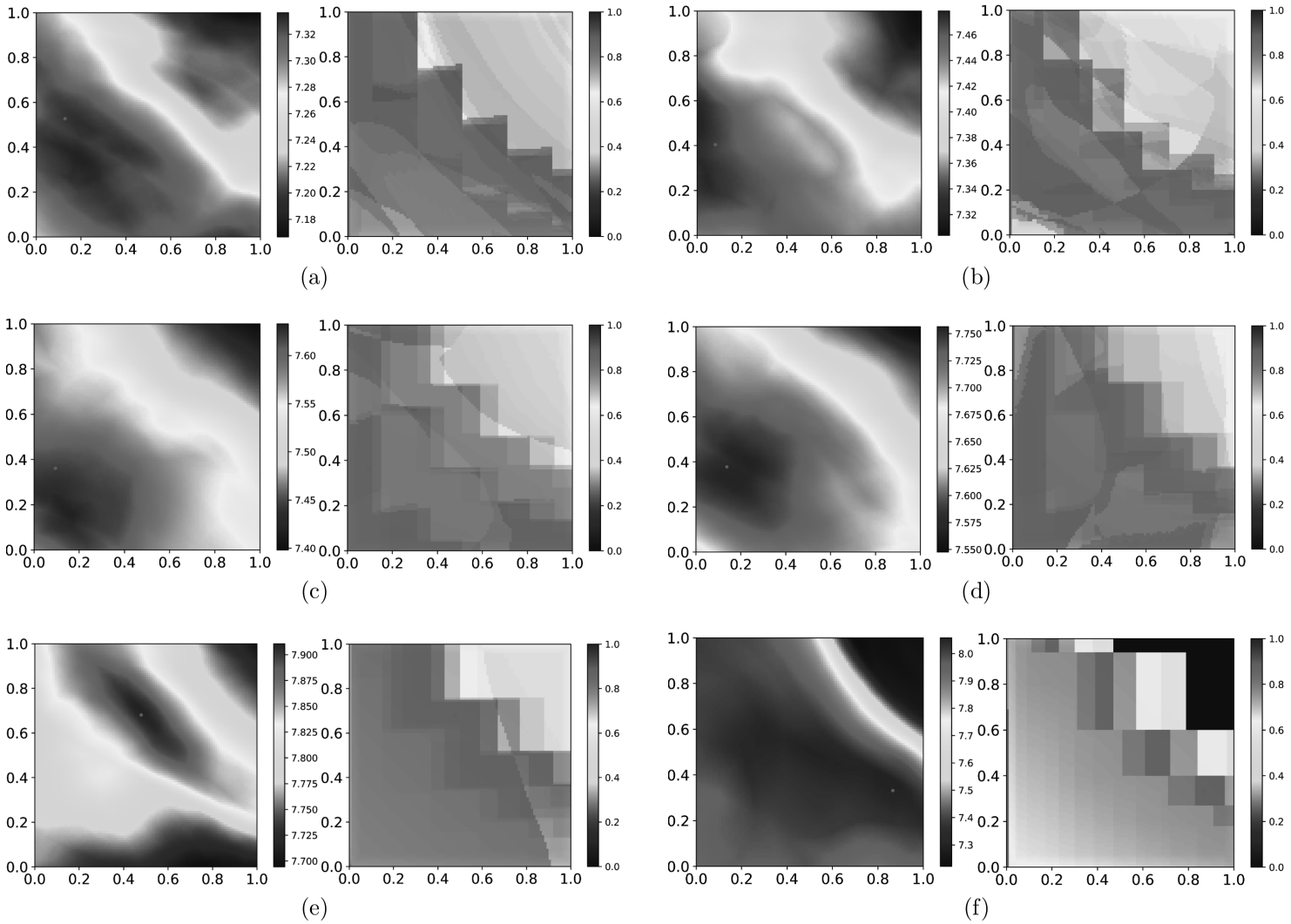


Figure 5.8: Distribution of Q-values (a-f left) and evaluation values (a-f right) for the 2nd-7th battle events (x-axis: HP, y-axis: ATK). Colors closer to red have higher values.

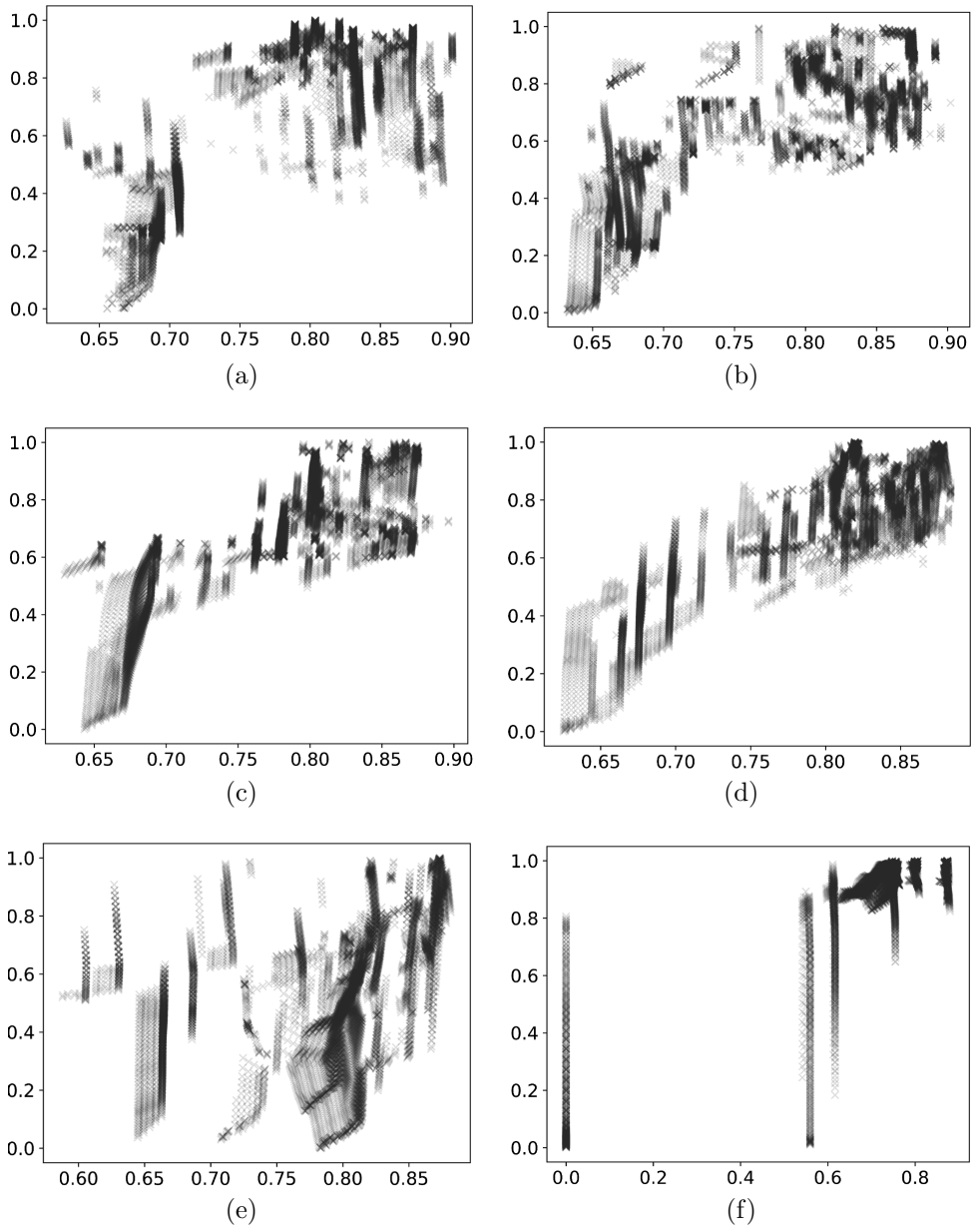


Figure 5.9: Relations between Q-values and evaluation values from Fig. 5.8 for the 2nd-7th battle events, where the Pearson correlation coefficients are 0.742, 0.787, 0.806, 0.849, 0.383, and 0.884. (x-axis: evaluation values, y-axis: normalized Q-values)

Table 5.3: Average ASDs and average rewards from 200 stages (Fig. 5.5a) generated when the first c columns were randomly initialized.

c	1	2	3
ASD	0.727 (±0.024)	1.040 (±0.030)	1.290 (±0.020)
Reward	0.797 (±0.006)	0.766 (±0.006)	0.711 (±0.015)
c	4	5	6
ASD	1.443 (±0.028)	1.552 (±0.027)	1.742 (±0.027)
Reward	0.667 (±0.013)	0.598 (±0.006)	0.533 (±0.018)

retreat, were available. Thus, the total number of possible strategies was 2^6. A higher difference means that efficient strategies in one stage do not work in the other, and thus players need to try different actions for different stages.

Result

Our approaches for diverse stage generation require a well-trained model. In the experiments, we employed VS-DDPG for generating stages and evaluating actions. The stage composition was the one in Fig. 5.5a. During training, we randomly selected an integer $1 \leq c \leq 5$ and initialized the first c columns to random values, where it was $1 \leq c \leq 2$ in the previous experiments. This modification aimed to explore more stages and learn Q-values better.

We first experimented on randomized initialization (RI) and generated 200 stages for each integer $c \in [1, 6]$ where the first c columns were randomly initialized. Table 5.3 shows the average ASDs and rewards for each c. Note that the stages contained seven columns, so randomly initializing six meant that almost all event parameters were randomly decided. As expected, with more randomly-initialized columns, the average ASDs increased while the average rewards decreased.

As discussed in Section 4.3 and shown in Table 5.3, the diversity was limited by only employing RI, especially when high quality was also desired. In the next experiment, our proposed DAGP (diversity-aware greedy policy) was applied, where the first column was randomly initialized, and the rest were set to noise events. We prepared 50 initial stages to see whether different initial stages influence DAGP's results. We tried several values for d (distance threshold) and n (candidate stage number), and each setting generated 50 stages from each initial stage. The goal was to investigate how different stages could be generated from the same initial one. High values of d prevented actions from being too close to the actor's action, and thus, we expected to generate diverse stages. Also, we expected high values of n (many candidate actions) to increase the chances of obtaining good stages.

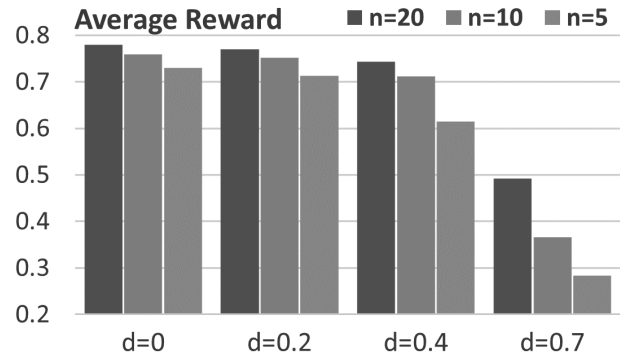
Fig. 5.10 shows the average reward, the parameter ASD, and the number of different winning strategies of different d and n settings. Note that when calculating the parameter ASD and the number of different winning strategies, a stage was compared only to another that was generated from the same initial stage, instead of calculating among all 50×50 stages. The results in Fig. 5.10 were the averages over the 50 initial stages. As expected, higher n indeed led to stages with higher average rewards, as shown in Fig. 5.10a, and higher d led to higher parameter ASD and the number of different winning strategies, as shown in Figs. 5.10b and 5.10c. The only exception for diversity was the number of different winning strategies when d was 0.7. The reason was related to the low quality of stages, either impossible to clear or too easy to clear.

Except the case of $d = 0.7$ in Fig. 5.10c, we observed that the quality (average reward) had an opposite trend to the diversity (parameter ASD and number of different winning strategies). We suspected that the phenomenon was related to the Q-values learned by the critic, as shown in Fig. 5.8. Even though multiple actions had high evaluation values (Fig. 5.8a-5.8f (right)), the critic tended to have Q-values centered at one of the actions (Fig. 5.8a-5.8f (left)). Since DAGP selected actions with the highest Q-values from far-enough actions, when d decreased or when n increased, it was more likely to get an action closer to the actor's action. Thus, increasing n was likely to decrease the diversity, while decreasing d was likely to increase the average rewards.

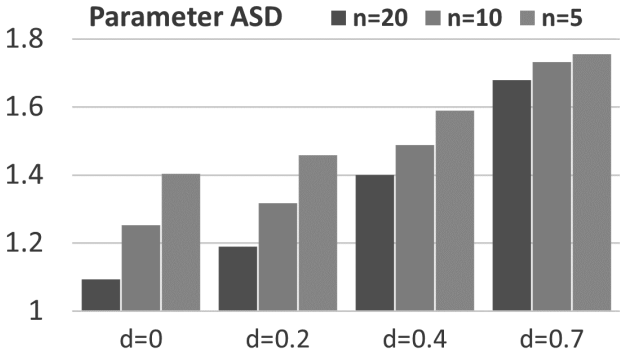
Fig. 5.11 shows example stages by VS-DDPG (top) and DAGP (bottom). The former was evaluated as 0.889, and the later as 0.878. The stage parameter ASD between the two stages was 1.449, and the number of different winning strategies was 11. Although the first event was the same, it could be seen that different good stages were generated by introducing noise.

Although DAGP was shown to be able to generate diverse stages, one minor problem remained. Sometimes, DAPG generated stages with low evaluations (e.g., 0.3), which should be discarded to keep high quality. We further conducted an experiment to investigate the influence on quality and diversity by filtering out bad stages. Since such removal was expected to increase the cost of online generation, we also investigated the generation cost. In addition to DAGP with VS-DDPG (abbr. RL-DAGP), we included two other algorithms for comparison, DAGP with SL (SL-DAGP) and random generation.

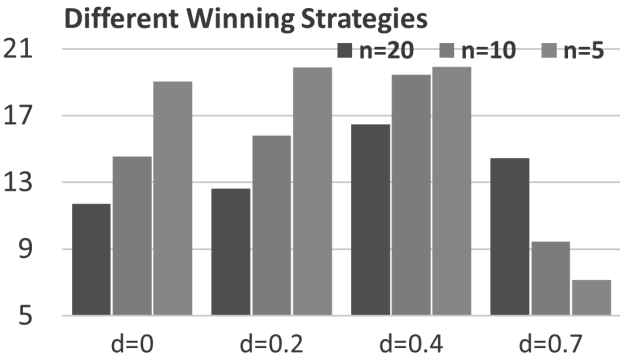
To apply DAGP in SL, we added a network similar to the critic of DDPG that evaluated given stage-action pairs. The evaluator network was trained by 10000 randomly generated stages regardless of their evaluations. Different from the generator network that only required good stages, the evaluator net-



(a)



(b)



(c)

Figure 5.10: Evaluation of DAGP results averaged from 50 initial stages where each had 50 stages generated. (a) average reward, (b) parameter ASD, (c) number of different winning strategies. Each blue, orange, and gray bar is for $n = 20$, 10, and 5, respectively, and each group of bars represents the result when $d = 0$, 0.2, 0.4, 0.7 from the left.

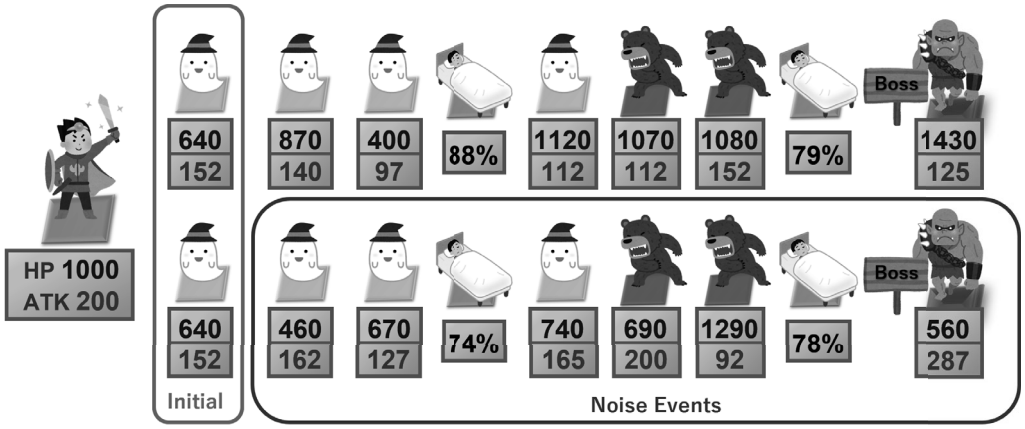


Figure 5.11: Stages generated by the actor policy (top) and DAGP with the first event fixed and the rest set to noise events (bottom).

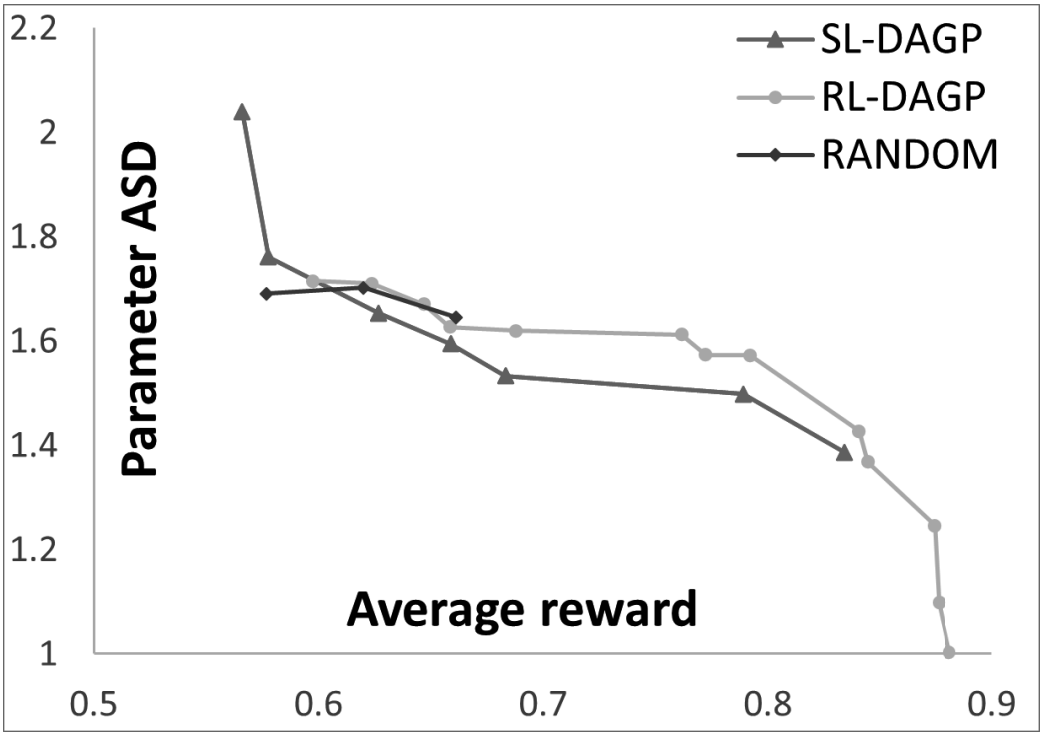


Figure 5.12: The Pareto frontier to average rewards and parameter ASD for RL-DAGP, SL-DAGP, and random generation.

work's training data should also contain bad stages so that it could predict the evaluations of state-action pairs better. For the three algorithms, we prepared five initial stages and let each setting collect 50 stages from each initial stage. DAGP parameters were $n \in \{5, 10, 20\}$ and $d \in \{0, 0.2, 0.4, 0.7\}$, and stages with evaluation values lower than v were discarded, $v \in \{0, 0.5, 0.7, 0.8, 0.85\}$. Note that a v of 0 meant that no stages are discarded. We omitted the results of settings that could not obtain 50 stages within 500 trials since those settings would require more than 3 seconds to get one stage, which we considered unsuitable for online generation.

Fig. 5.12 shows the Pareto frontier to the average rewards and parameter ASD. RL-DAGP dominated SL-DAGP in most cases. Random generation could generate diverse stages but the quality was much lower. As discussed in Section 5.2.1 (the 2nd paragraph), when the random generation was employed to collect good stages with evaluations over 0.9, it required about 80 seconds to get one stage, which was time-consuming. From the experiments, RL-DAGP was the most promising to generate high-quality and diverse stages online.

5.3 Chapter conclusion

In this chapter, we proposed the use of reinforcement learning with the theme of stage generation in turn-based RPG. The stage generation problem was formulated into a Markov decision process, where states were incompleted or completed stages represented by real-number matrices and actions were to fill in the matrices. We defined an evaluation function to consider the difficulty and several entertaining factors of stages. By using the evaluation function to give rewards in reinforcement learning, our proposed method successfully generated highly evaluated stages.

We applied DQN and DDPG for discrete and continuous actions, respectively. For discrete actions, the range was divided into a fixed number of sections, and one value in the stage matrix was decided at a time. In contrast, it was easier for continuous actions to decide several values (of related events) at once, which was expected to handle related events more effectively. DDPG indeed generated better stages than DQN under our experiment settings. When our proposed method is used in other PCG problems, action types may be a factor that affects the performance.

Under our reward design, the delayed rewards made the model hard to efficiently learn to evaluate actions, especially when the event sequence was long. To overcome the difficulty, we introduced virtual simulations to provide intermediate actions with rewards. Experiments showed that virtual

simulations helped to improve the quality of stages.

Since the primary purpose of this research is to generate good and diverse stages, we proposed randomized initialization that assigns random parameters to several beginning events and a diversity-aware greedy policy that chooses actions distant but not bad compared to the best actions from the model's view. The experiments demonstrated the effectiveness of the proposed approaches in obtaining good and diverse stages for online generation.

Chapter 6

Generation of Super Mario levels

In this chapter, a level generation in Super Mario is described. First, the two challenges not appearing in the turn-based RPG stage generation problem are discussed in Section 6.1. A solution for each challenge, the function approximator and the human-like AI agent, is explained in Sections 6.3.4 and 6.4, respectively. The overall process of the level generation and each component including MDP formulation for the Super Mario level generation are presented in Section 6.2-6.5. Next, experimental results and discussions are provided in Section 6.6. Finally, Section 5.3 concludes the overall chapter.

6.1 New challenges

Super Mario's level has a $15 \times n$ two-dimensional structure. If a level is gradually filled by a part of the matrix (e.g., 15×4 pattern) starting from the left end, it can be considered the same manner as determining the parameter values of turn-based RPG's stages RPG as shown in Fig. 4.1. The general concept of formulating a level generation problem into MDP is similar to the stage generation problem. However, there are some differences in formulating each problem into MDP. The differences in MDPs and the characteristics of the games raise two different challenges.

In the turn-based RPG's stage generation problem in this dissertation, each vector in a stage matrix can be directly converted to parameter values (e.g., HP, ATK) using the ratio of parameter ranges. It is a simple and intuitive way to represent a stage.

In Super Mario, it seems to be reasonable to represent levels as states of MDP. Then a 15×4 pattern of the level becomes the actions of RL.

DDPG training was proven efficient in environments with less than 10 dimensions of action space (e.g., Hopper-v1 with 3 dimensions or Walker2d-v1 with 6 dimensions). However, the training becomes difficult for environments with more than 10 dimensions (e.g., Humanoid-v1 with 17 dimensions) [67]. Therefore, we decided to use low-dimensional vectors as the output of RL, and map them to the corresponding level pattern. In order to do so, GAN, a function approximator that outputs the pattern from the vector, is employed. The detail descriptions of the function approximator and the MDP formulation methodology are explained in Sections 6.2 and 6.4, separately.

The second challenge is the evaluation of the difficulty of generated content. In turn-based RPG, the difficulty appropriateness is assessed using the winning rate as explained in Section 5.1.4. A strategy is a sequence of action selections, and the stage is tested with all possible strategies. Then, the winning ratio indicates how difficult the stage is. In Super Mario, the difficulty is also an essential factor. The action choices in Super Mario are broader and deeper than those in turn-based RPGs. Thus, finding every possible action sequence in Super Mario is nearly impossible and pointless. We adopt human-like AI agents and assume they represent human players. By letting the AI agents play the levels, the degree of difficulty from human ability is measured based on their play logs. Detailed configuration of the human-like AI agents and the evaluation method are described in Section 6.3.4

6.2 Overall process of the level generation

The overall process of the level generation is shown in Fig. 6.1. (1) A level in the generation process is transformed into a state (15 $\times$ n matrix with 3 channels, instead of image input). (2) Then RL outputs action, which is a vector instead of a pattern itself. (3) It is converted to a pattern through the function approximator. (4) The mapped pattern is repaired through Matching. (5) The repaired pattern is added to the incomplete level as the subsequent step. (6_1) Monotony is evaluated if the level is incomplete. (6_2) Otherwise, the level's difficulty is evaluated using human-like AI agents. From Section 6.3-6.5, details of each component in a level generation are described.

6.3 Markov Decision Process

In this section, we explain how to formulate State(incomplete level), Action(vector), state transition, and reward function of MDP.

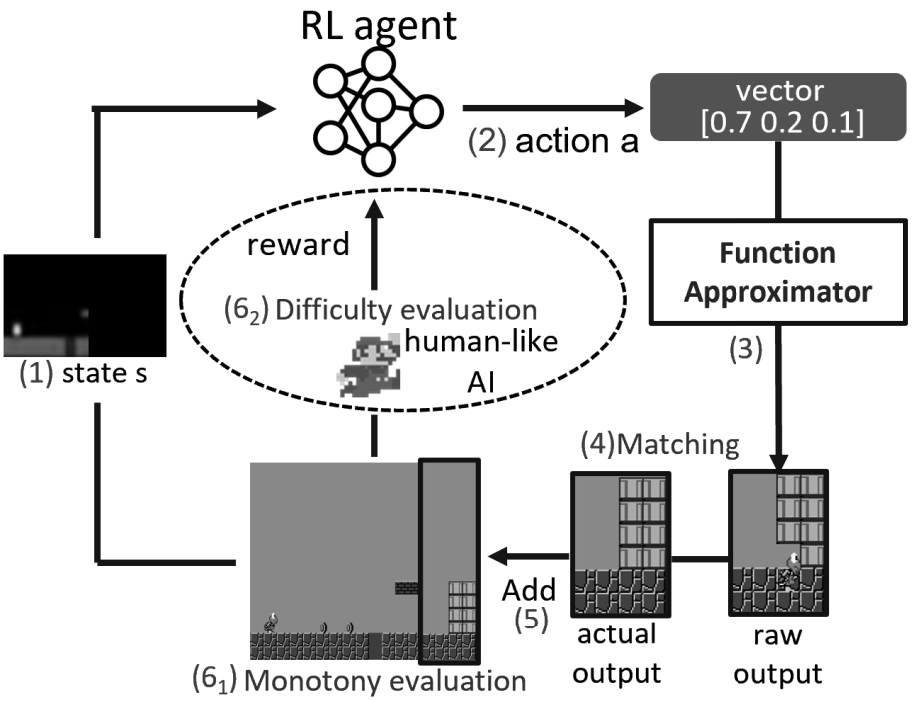


Figure 6.1: The overall process of the level generation.

6.3.1 State representation: incomplete level

In turn-based RPG, HP, ATK, and recovery are the components of a stage. Our method represents the components in a matrix with 3 rows and n columns, where n is the number of battle events. On the other hand, Super Mario, walls, enemies, holes, and etc. are the components of a level. The size of the level is $15 \times n$, where 15 is the height and n is the length of the level.

There are various ideas for representing the incomplete level and delivering it to RL as a state (e.g., a raw image of the level or a 15×n matrix consisting of 10 different tiles, or etc.). We semantically reduce the matrix of 10 different tiles into three binary matrix channels: walls, enemies, and holes. We considered this manner most suitable for RL's CNN to grasp the state and output actions properly. The goal information is excluded because it is always located on the right-most.

A level example shown in Fig. 6.2(a) can be represented by Fig. 6.2(b). Each color, red, green, and blue, represent a wall (c), an enemy (d), and a hole (e), respectively.

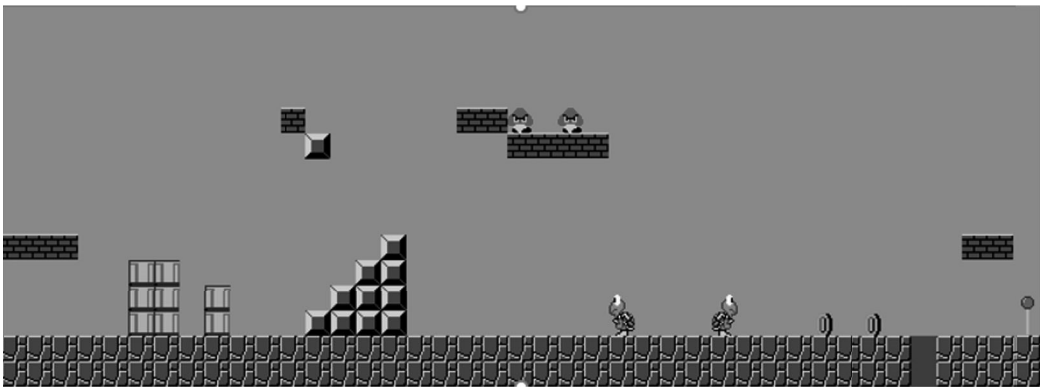
The levels, including complete and incomplete ones, are in a matrix of the same size, indicating a series of patterns. An empty level is represented by a matrix filled with minus one.[1]

6.3.2 Action representation and RL

A 15×4 pattern is added to the incomplete level. However, the actual RL model outputs three real values in $[0, 1]$. The RL action (vector) is converted into the pattern through the function approximator. Then converted outputs are filled in the sequential level matrices.

Since the RL methods needs to output a vector of real values, a variation of DDPG [62], so-called the Twin Delayed DDPG (TD3) [68], is adopted for level generation. The actor of TD3 outputs numerical values of vector in the range of $[0, 1]$, and the Q-value is evaluated by another model called critic as shown in Fig. 6.3. The actor takes an incomplete level matrix as the input and outputs three real number values. The critic takes the same level, and the actor's action as the input and outputs two Q-values and minimum value are used for training.

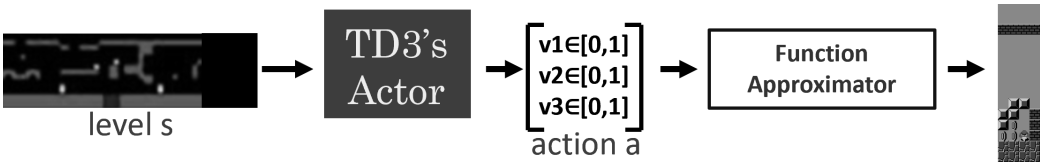
[1] In turn-based RPG's stage generation, the zero vector represents the minimum value of event parameters. However, the zero vector in Super Mario's level generation is an input to the function approximator, which can map the zero vector to a pattern. Therefore, a matrix is filled with minus ones instead of zeros to distinguish them in particular.



(a)

(b) (c) (d) (e)

Figure 6.2: An example of level representations. (a) is the example level in Super Mario Game. (b) is an integrated image of (c), (d), and (e). (c) is a wall channel image, (d) is a enemy channel image, and (e) is a hole channel image.



(a)

(b)

Figure 6.3: TD3 for level generation in this dissertation.

6.3.3 State transition

Differ from the state transition in Section 5.1.3, given an incomplete level matrix as a level and a real number vector as action, the matrix is not filled in by the action vector. Instead, a vector is transformed into a pattern. Then the pattern is represented as a pattern matrix which is explained in Section 6.3.1. The pattern matrix is filled in the incomplete level matrix. Level generation terminates when the level is entirely filled by patterns.

6.3.4 Reward function

A level evaluation function is defined and used as the reward function of the MDP. The evaluation function highlights two aspects, difficulty and monotony. The difficulty appropriateness is essential for player enjoyment. The monotony is another important indicator. If the pattern is repetitive, the player may feel bored.

We employ a computer test player to evaluate the difficulty of the generated Mario levels. The AI agent is not very strong and is designed to make human-like mistakes. This is for estimating the difficulty perceived by human players. Moreover, a new concept *virtual damage* is introduced for evaluating how dangerous the level is and how fatal the mistakes are. The detailed implementation of this agent is described in Section 6.5.

In summary, levels are evaluated in three manners. The first is to ensure that the level is playable. Level difficulty or monotony is pointless if the level is not playable. Playability is evaluated by simulation with a strong A* AI agent implemented by R. Baumgarten described in [69]. If the base A* AI agent cannot complete a level, the level is considered not playable.

The second evaluates difficulty. This dissertation focuses primarily on virtual damage caused by human-like AI agents to express the level's difficulty which is described in Section 6.5. Virtual damage occurred by enemies ($enemy_{dmg}$) and holes ($hole_{dmg}$) is treated with different weights. Then the total damage $total_{dmg}$ is calculated as the linear sum of them (Eq. (6.1)). The linear sum of their counts is the total damage.

$$total_{dmg} = enemy_{dmg} \cdot 1 + hole_{dmg} \cdot 1.1 \tag{6.1}$$

The behavior of our AI agent is stochastic. The value of $total_{dmg}$ varies each time. Therefore, we perform 10 trials and take the average value for accurate difficulty estimation. A large $total_{dmg}$ means that the level is difficult, while a small $total_{dmg}$ means that the level is easy. Finally, we compute the appropriateness of the difficulty, r_{diff}, using the function shown in Fig. 6.4.

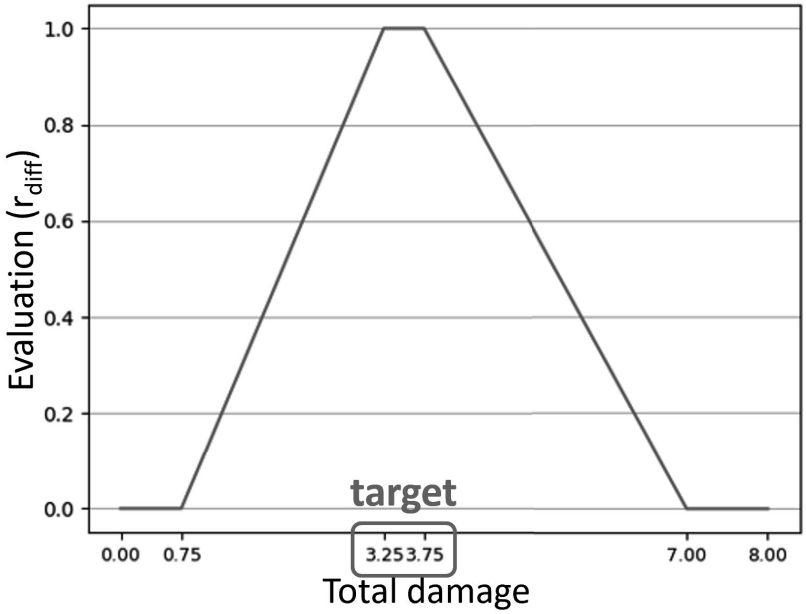


Figure 6.4: An evaluation based on the total score. (x-axis: total score, y-axis: evaluation based on the total score)

The third evaluation is the monotony of the level. It corresponds to the trajectory of the player's gameplay. As mentioned in Section 3.2.1, the decision of action sequences in a pattern is mostly affected by the layout of wall and hole tiles. Similar patterns lead to repetitive action decisions and monotony. Therefore, it is undesirable if the appearance of the newly generated pattern is the same as that of the previous patterns. To evaluate monotony, wall and hole tiles of nth pattern (p_n) is compared with previous four patterns (p_{n-1}, p_{n-2}, p_{n-3}, p_{n-4}). A higher penalty is given for the more recently generated pattern. Let $id(p, q)$ be a function that $id(p, q) = 1$ if and only if two patterns p and q have the same wall tiles. Then, the monotony evaluation value, r_{mono}, is computed by Eq. (6.2).

$$\begin{aligned} r_{mono} = {} & 8/15 \cdot id(p_n, p_{n-1}) + 4/15 \cdot id(p_n, p_{n-2}) \\ & +2/15 \cdot id(p_n, p_{n-3}) + 1/15 \cdot id(p_n, p_{n-4}) \end{aligned} \tag{6.2}$$

Monotony is assessed after a pattern is generated. While the difficulty evaluation is evaluated once the level is fully generated. The weight monotony evaluation value (w_{mono}) is specified 0.01.

6.4 Function approximator: from a vector to a pattern

The function approximator is employed to map a vector to the corresponding pattern. The vector is the action output of the RL agent, and the corresponding pattern is to be added to the current incomplete level, as mentioned in Section Section 6.1. Candidates for the function approximator are a self-organizing map (SOM) [70], GAN [17], CGAN [71], etc. We employed CGAN because it can include previous patterns as input and improve the output's naturalness.

For a well-made level, subsequent patterns have similar features in appearance to the previous ones. It is challenging to define naturalness. However, we believe these characteristics contribute to naturalness from the player's viewpoint in Super Mario. Levels become unnatural when different features of patterns are concatenated abruptly. Therefore, level generation also requires consideration of connectivity between patterns.

6.4.1 Verification: importance of natural connectivity

A preliminary experiment was conducted to confirm this theory. In this experiment, the human-designed level is compared with levels whose parts are randomly replaced by other patterns. Through this experiment, we can see how low connectivity affects the naturalness of appearance. The experiment indicated that low connectivity affects the naturalness of appearance. The preliminary experiment was conducted under the following setting and order.

Preparation of patterns

- 10 Levels in original Super Mario Bros were reproduced using Python Mario.
- Additional 12 custom levels were collected from five volunteers.
- Each level is divided by patterns having a fixed size of 15×4.
- 6,232 patterns (Pat_{legal}) are extracted using a sliding window algorithm with a sliding size of 4.
- For a level L whose length is n, we can obtain $n - 3$ patterns PL_1 (x=1..4), PL_2 (x=2..5), .., PL_{n-3}.

Level generation procedure

i) Choose any level (L) among 22 levels.

ii) Select any pattern (PL_i) from the Super Mario level L and add it to an empty level.

iii) Choose the next connected pattern PL_{i+4} with probability p or choose other random pattern i and level L. Concatenate it in the level in the process of generation until the level is completed.

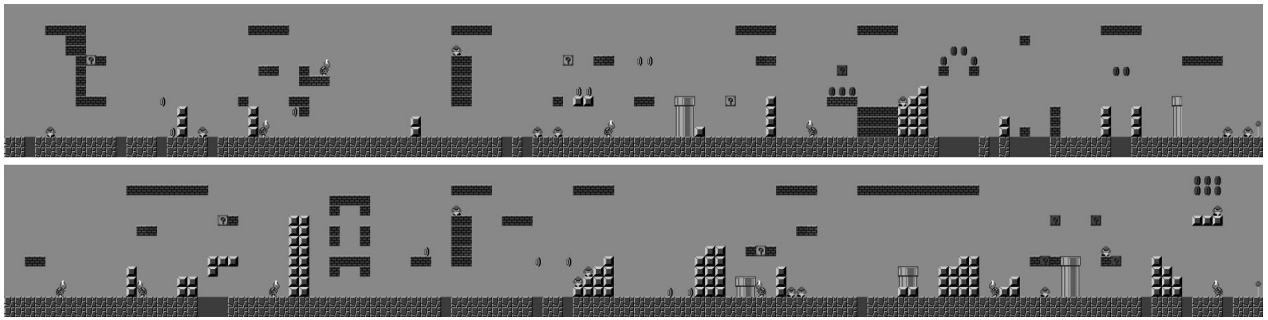
Fig. 6.5 displays the generated levels when p are 0, 0.3, 0.9 in order. The larger the value of p, the closer to the level at which the connection from Pat_{legal} is sustained. Several abrupt patterns emerge in levels generated by using smaller values of p and the levels seem unnaturally connected in some places.

Preliminary results indicated that the function approximator should handle mappings between vectors to patterns and consider natural connectivity. In order to reflect the natural connectivity in the function approximator, we employed CGAN, which takes n of previous patterns as a condition. From the assumption that the three patterns and the next pattern at the human-designed level are naturally connected, three previous patterns are given to CGAN as the condition to output a naturally connected pattern.

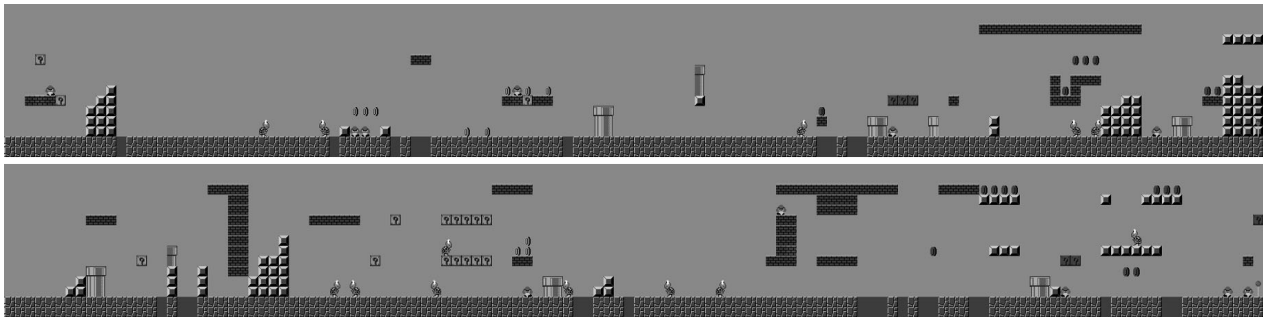
Fig. 6.6 outlines the procedure of a simplified version of CGAN employed in this dissertation[2]. There are two neural network models, a generator and a discriminator, which is trained by competing each other. After training process is done, the generator is used for mapping a vector (noise) into a pattern aiming for a natural connection. The detailed training processes are as follow.

- The set of four patterns ($P_i, P_{i+4}, P_{i+8}, P_{i+12}$) (15×16) are given as training data.
- The last pattern (P_{i+12}) is cut, and the first three patterns (P_i, P_{i+4}, P_{i+8}) are given to the generator with a vector (noise).
- The generator outputs a pattern (P_{out}) and is concatenated to the set of P_i, P_{i+4}, P_{i+8}.
- The discriminator receives the set of $P_i, P_{i+4}, P_{i+8}, P_{i+12}$ and the set of P_i, P_{i+4}, P_{i+8} followed by P_{out}.

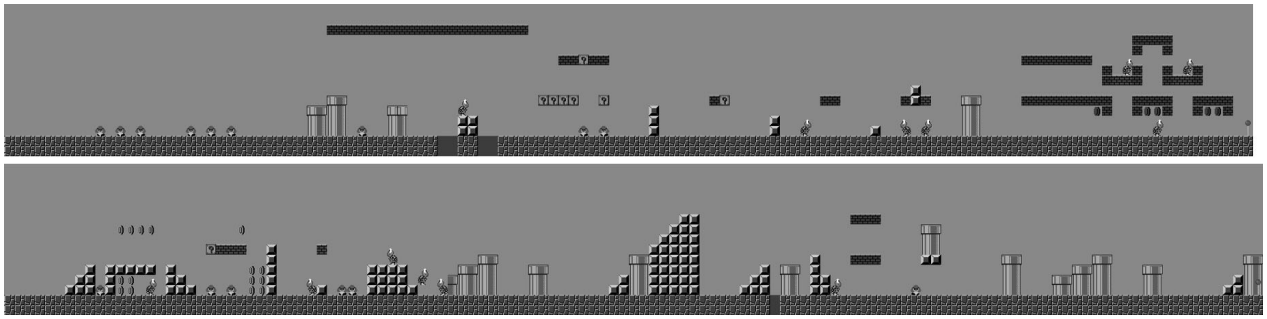
[2]Data duplication is applied in an actual version of CGAN to improve performance. The details are explained in the Appendix A.



(a) p=0



(b) p=0.3



(c) p=0.9

Figure 6.5: Levels generated to investigate the impact of influence of the connectivity between patterns. Whenever the pattern is generated, original connection is maintained with probability p or new random pattern is added. Each level is generated using $p = 0$, 0.3, 0.9 from top to bottom. Note that p=0 is equivalent to the random generation.

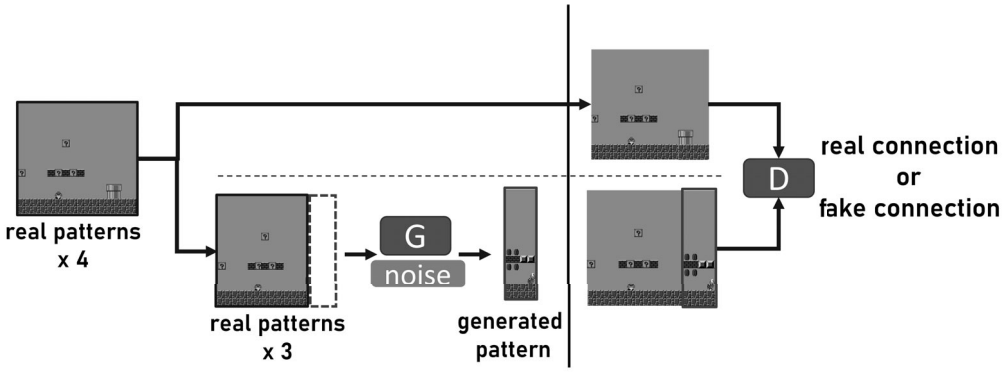


Figure 6.6: A structure of a simplified version of CGAN for Super Mario pattern generation.

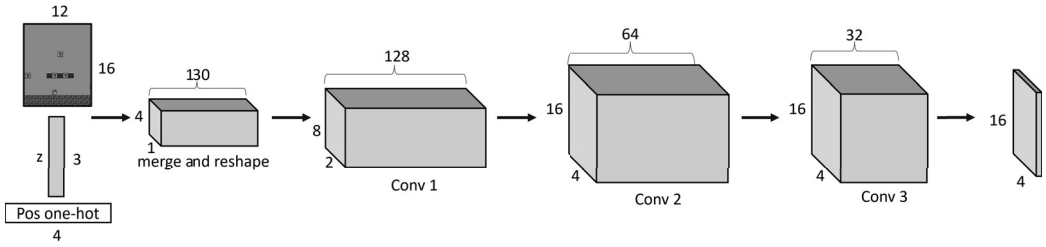
Table 6.1: CGAN setup

Parameter	Value
Level data	10 original + 12 custom levels → 6232 patterns by sliding window (size 1)
Learning rate	0.00005
Noise (z) dimension	3
Batch size	16
Activation function	Leaky ReLU
Dropout	0.4

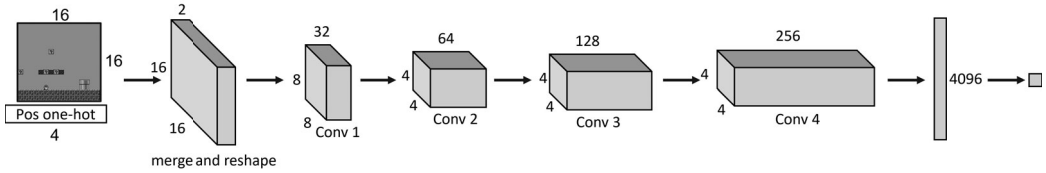
- The discriminator is trained to distinguish them apart. The generator is trained to deceive the discriminator by outputting a pattern that has a natural connection.

6.4.2 Matching and verification

Machine learning techniques do not produce perfect results. In this sense, mapping results from CGAN can contain some noise. T. shu et al.[72][73] proposed CNet-assisted Evolutionary Repairer (CNet) that detects the defects in the tile layout and fixes them. In this level generation problem, only Pat_{legal} are used. For this reason, this dissertation introduces a different method called *Matching*. Matching is the algorithm to select the closest pattern by comparing the noised pattern with all Pat_{legal}. There are many criteria for determining the distance between patterns. The best criteria investigated during preliminary experiments, Euclidean distance, is employed We employed Euclidean distance because it is the best criteria investigated during preliminary experiments. The detailed process of Matching is as follows.



(a) Generator



(b) Discriminator

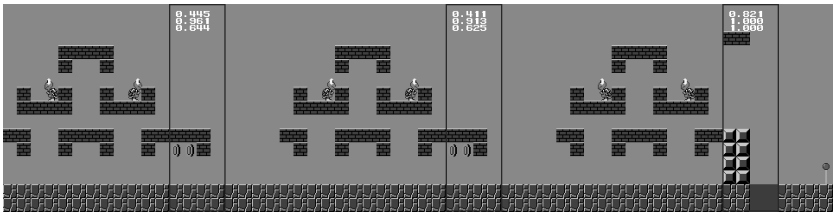
Figure 6.7: Network layers configurations of CGAN.

- Each tile has a unique integer value (e.g., Brick=0, Goomba=5). Thus, the pattern can be represented as a 15×4 matrix in which each value is an integer of tile value.
- Each element is normalized to $[0, 1]$, then each pattern can be represented as a 60-dimensional vector $\in [0, 1]^{60}$.
- Each matrix $m \in \text{Pat}_{legal}$ and the matrix from the CGAN output are compared using Euclidean distance.
- The output pattern is replaced by the pattern with the smallest value.

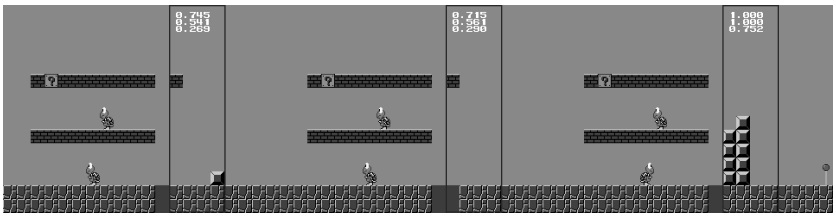
Verification of CGAN

We perform three evaluation experiments to investigate whether the proposed CGAN and matching have the expected capabilities. The first is to verify that close vectors are mapped to close patterns (and vice versa). The second is to verify the continuity of the mapping from vectors to patterns. The third is to verify the naturalness of the generated levels.

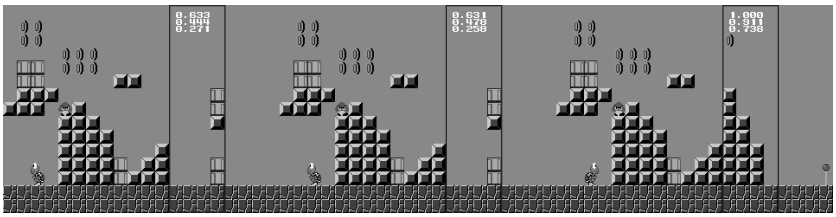
Vector spaces of the function approximator should map close values to similar patterns and different results to distant values. Fig. 6.8 shows the examples of three patterns (red square) followed by previous patterns after applying Matching. The same previous patterns were given as input to the CGAN, and each input vector is shown on the upper side in the red square.



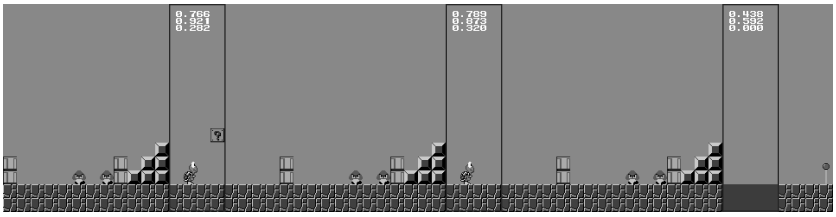
(a)



(b)



(c)



(d)

Figure 6.8: Comparison of patterns from different vectors.

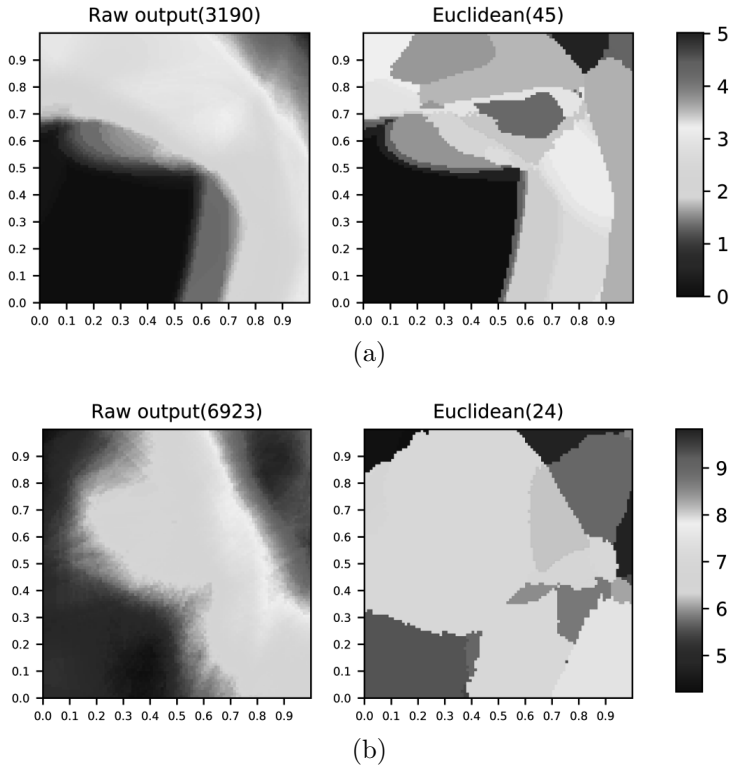


Figure 6.9: Distribution of patterns from CGAN (a-b left) and patterns from Matching (a-b right). The closer the color is to red, the greater the value of Euclidean distance from the empty pattern.(x-axis: 2nd vector value, y-axis: 3rd vector value).

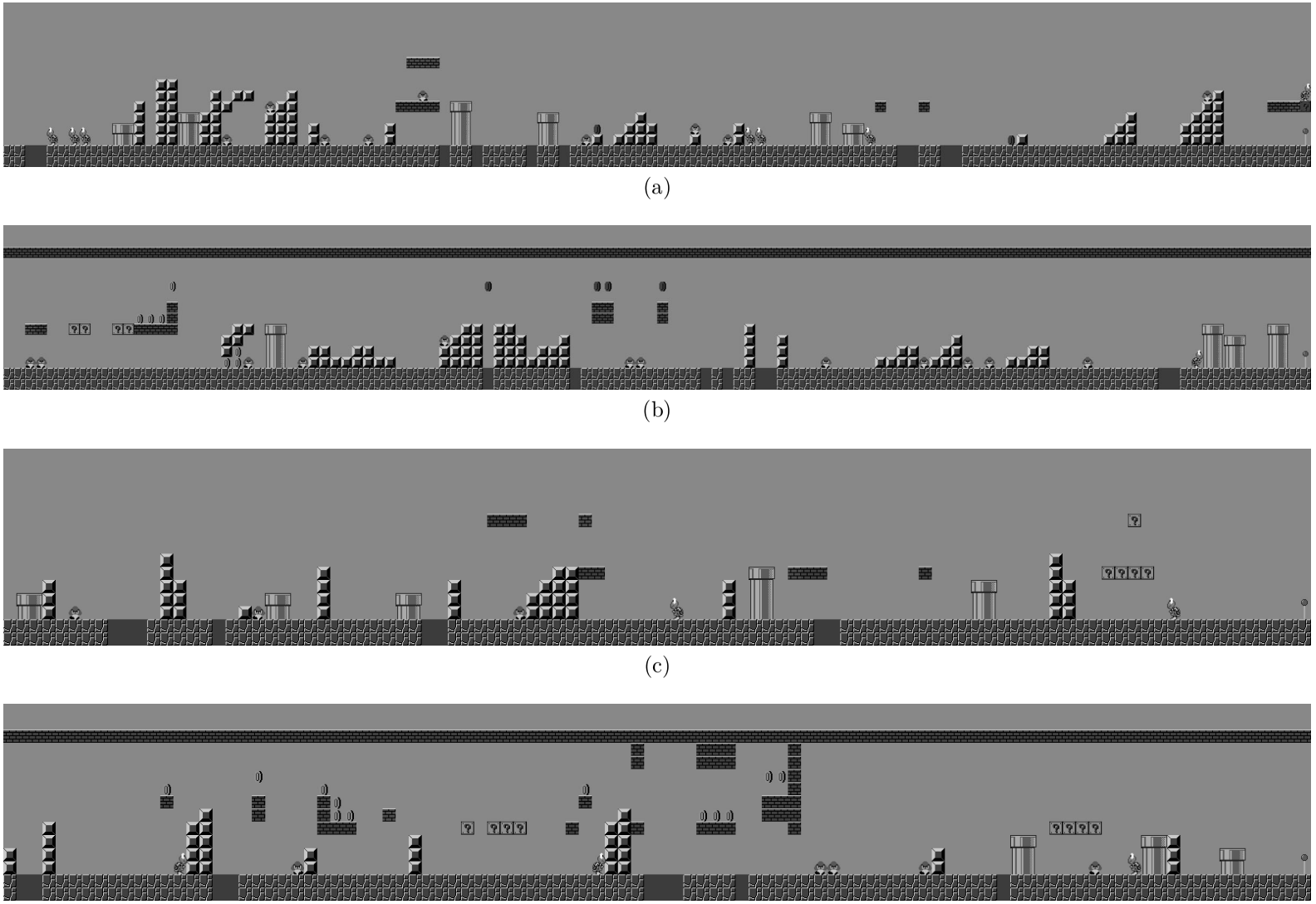
The first pattern (p_1) was from a random vector (v_1), the second pattern (p_2) was mapped using a vector (v_2) close to v_1, and the third pattern (p_3) was a result from a vector (v_3) distant from v_1. The results exhibited that patterns from close vectors, v_1 and v_2, produced similar results and patterns from distant vectors, v_1 and v_3, were different.

The next step is to verify the continuity of the mapping from vectors to patterns. However, it is difficult to visualize both 3-dimensional vectors and 60-dimensional patterns. To comprehend the continuity of the mapping between them, the first value of the vector was fixed as zero, and the second and third values were varied for the entire 3-dimensional vectors. 101×101 vectors (by 0, 0.01, 0.02, ..., and 1) were investigated. Then, the Euclidean distance between the pattern corresponding to the vector and the empty pattern was plotted. Fig. 6.9 shows the plot results with and without Matching, including the number of generated patterns from 10,201 vectors. Note that

the different previous three patterns are used compared to the ones used in Fig. 6.8. More red color signifies the greater distance between the pattern and the empty pattern.

Results demonstrate that vector spaces have similar patterns at close values and different patterns at distant values. The left figures had 3,190 and 6,923 distinct patterns, indicating many minor defects in raw output. Figures (a) show similar distributions with the 3,190 patterns reducing to 45, and (b) also exhibit a similar trend.

Finally, Fig. 6.10 shows 100-length of levels generated from CGAN using random vectors. A series of three consecutive patterns is chosen from Pat_{legal} and given as initial patterns. A raw pattern is generated by CGAN with three previous patterns and random noise. Then, matching is applied to repair a raw pattern into a pattern $p \in Pat_{legal}$. Then it is added to the incomplete level. When underground patterns are initially given at the beginning of the levels, as shown in (b) and (d), generated levels also explicitly show the underground patterns. The obtained levels have more natural connectivity than randomly connected patterns shown in Fig. 6.5 (a). However, since the evaluation function is not employed in this generation experiment, the levels seem to be boring in many parts.



(a)

(b)

(c)

(d)

Figure 6.10: Examples of levels generated from CGAN.

6.5 Human-like AI agents for evaluation

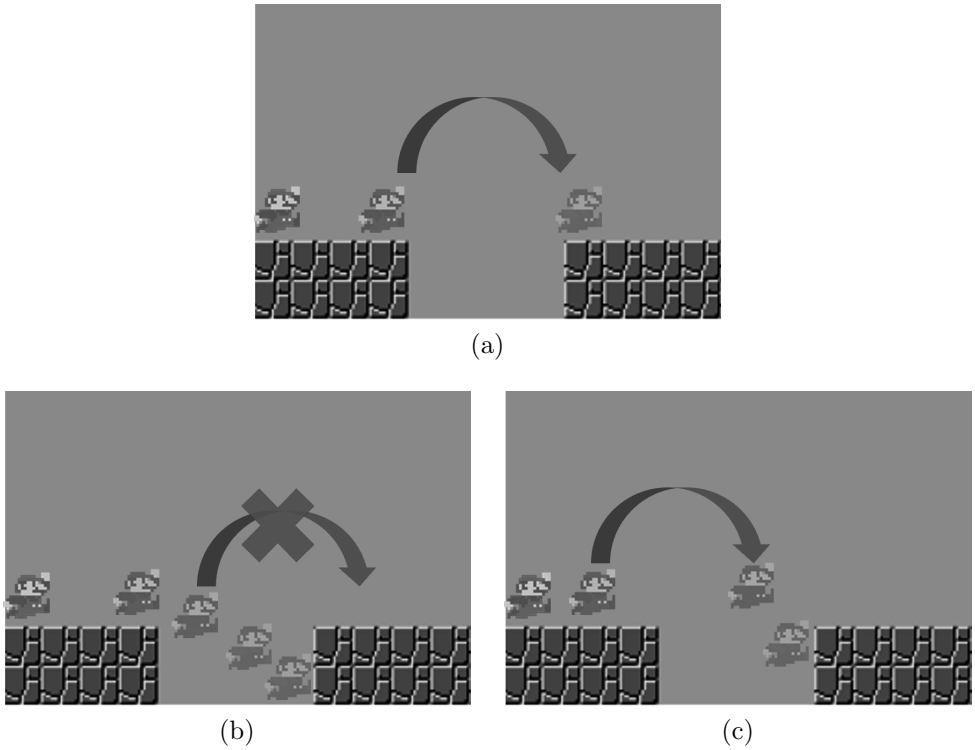
The generated levels is judged to have appropriate difficulty and is used as a reward for RL. A human-like AI agent is required for test play and assessing difficulty, as described in Section 6.3.4.

This research focuses on two aspects of human nature. The first is failures caused by inaccuracies of human nature. By nature, human players usually try to take specific actions to avoid dangers in platformer games, including Super Mario. However, processing decisions about when to execute actions in a given situation may delay or hasten the action input because of the inaccuracies of human nature. Fig. 6.11 shows examples of failures caused by the inaccuracies of human nature. Mario needs to jump over the hole to avoid falling (a). Due to the delay (b) or hasten (c) in the jump input, Mario falls into the hole. These kinds of inaccuracies are prevalent in Super Mario.

While the first aspect is about the timing, the second is about the time span of input. During gameplay, players sometimes press longer on the action than they should because humans are not as accurate as machines.

From these human-like aspects, we can estimate the level's difficulty for human players. Since such human failures occur differently by players, the behavior of our AI agents is stochastic. AI agents were implemented considering the above elements, and their details are as follows.

- An A* AI agent implemented by R. Baumgarten [69] is employed as a base AI agent.
- Because the base AI agent emphasizes the performance in competition, its goal is to complete as many levels as possible. To avoid failures, Mario's jumping frequency was minimized, and he did not try to jump over the walls if it was not necessary. Minimizing jumping is not desirable for our purpose. Therefore, we added an extra reward to motivate jumping. In particular, depending on Mario's position (Y-axis), an extra reward of 0 to 0.1 is given to the A* agent. This modification sacrifices the performance to some extent. However, our goal is to employ a human-like AI agent whose behavior is similar to human players, not a competent one.
- Some impaired movements (e.g., keeping stuck at a dead end) have been modified by searching for deeper actions.
- As stated above, the timing of human players' action decisions can be delayed or hastened. If such human-like mistakes frequently lead to a game over, then the level is considered difficult even if a strong A* can



(a)

(b) (c)

Figure 6.11: Examples of ideal movements by humans (a) and failures due to the inaccuracies of human nature (b-c).

clear it. Therefore, we perform a virtual A* simulation of the damage (contacting an enemy, falling into a hole) caused by delayed or hastened timing of actions. Suppose an action a is taken at frame t, (1) a is performed twice at frames $t+1$ and $t+2$, and A* search is done to determine whether the delay is fatal (Fig. 6.11 (b)). Or, (2) a is performed at frame $t-2$, and A* search is done to determine whether the haste is fatal (Fig. 6.11 (c)). If Mario takes damage by these simulations, we call it *virtual damage*. This virtual damage simulation occurs randomly (30%) and limitedly (once per pattern).

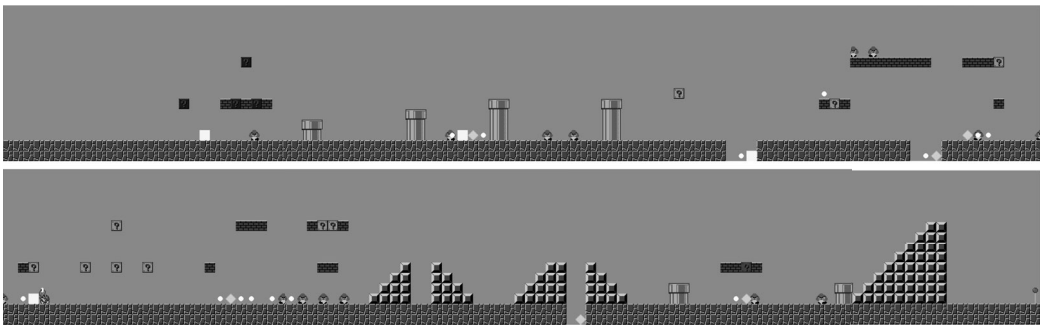
- Regarding the long press, AI agents often execute the same actions twice that occurred in the previous frame instead of executing an outcome from an A* search. Long presses also occur randomly (30%) and limitedly (twice per pattern).

- Note that the inaccuracies of human nature occur virtually, not actually. On the other hand, the long press actually occurs.

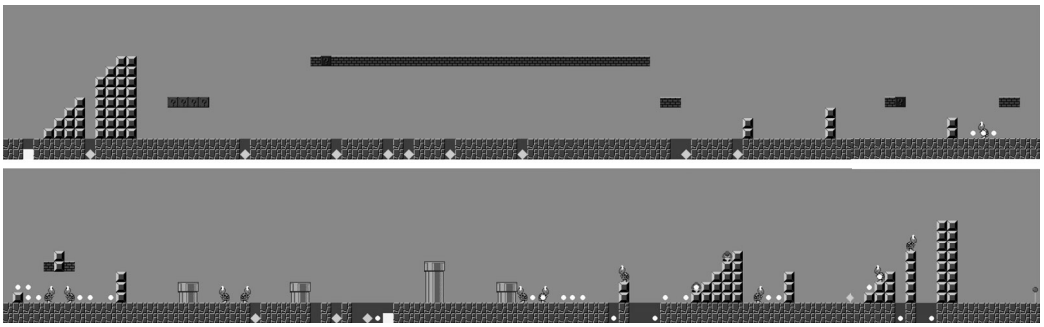
Table 6.2: Mean and standard distribution of the number of indicators of AI agents' human-like behaviors at each level.

Level	1-1	1-2	2-1	4-1	5-1
Jump count	33 (±7.2)	27.1 (±2.4)	23.4 (±10.8)	26.8 (±8.5)	19 (±2.6)
Damage (enemy)	3.5 (±1.5)	4.3 (±1.0)	4.2 (±2.5)	0.2 (±0.4)	5.1 (±1.3)
Damage (hole)	1.5 (±1.0)	1.6 (±0.5)	1.6 (±0.9)	2.1 (±0.7)	2 (±1)
Long press	36.5 (±6.9)	30.3 (±1.7)	27.1 (±10.9)	32 (±9.6)	25.8 (±2.2)
Level	6-1	6-2	8-1	8-2	8-3
Jump count	27.8 (±10.3)	20.3 (±6.1)	29.4 (±11.4)	27.9 (±4.0)	18.7 (±2.9)
Damage (enemy)	0.5 (±0.5)	0.6 (±0.7)	3.3 (±1.6)	0.7 (±0.9)	0.6 (±0.7)
Damage (hole)	2.1 (±1.3)	1 (±0.8)	4.5 (±2.5)	7.4 (±2.0)	3.3 (±1.3)
Long press	29.6 (±10.0)	23.4 (±6.6)	35.9 (±14)	30.2 (±3.4)	14.8 (±1.6)

The Table 6.2 shows the average count of AI agents' human-like behaviors in 10 trials at 10 different Super Mario original levels. The number of occurrences of each indicator for each trial is adequately different. Some complicated gimmicks applied in some levels (e.g., Piranha Plant in level 4-1, Spiny in level 6-1, water area in level 6-2, Hammer Bro in level 8-3) are excluded. Therefore, those levels are relatively easy. The results indicated that each trial behaves differently and can be considered a different player. Fig. 6.12 shows some reproduced Super Mario original levels and the locations of virtual damages from 50 trials of AI agents. Note that some virtual damages are overlapped and they are differently shown. The Locations of virtual damages are positively correlated with the risky points in the levels. Thus, virtual damages are generally reliable as difficulty indicators.



(a) Level 1-1



(b) Level 8-2

Figure 6.12: The original levels and the locations of virtual damages (yello square $dmg >= 20$, green diamond $10 <= dmg < 20$, white circle $dmg < 10$) from 50 trials of AI agents.

6.6 Experiments and discussions

In this section, the results of generating high-quality levels are presented in Section 6.6.1. Next, the result of generating diverse levels are described in Section 6.6.2.

6.6.1 High-quality level generation

In order to generate Super Mario levels by the proposed PCG approach, the RL algorithm, TD3 [68], is employed. The virtual simulation (VS) was applied to handle the credit assignment problem (CAP). TD3 and TD3 with VS were compared to evaluate the quality improvement of the proposed method.

As we metioned in Section 6.3.4, there are two types of reward: monotony reward (r_{mono}) and difficulty reward (r_{diff}). The two types of rewards are weighted in RL, and the total reward for the episode (r_{total}) is given by equation Eq. (6.3). Each reward assesses a different aspect. Thus, we investigate all the rewards while analyzing the learning trends.

$$r_{total} = r_{diff} + w_{mono} \sum r_{mono} \tag{6.3}$$

Experimental setup

The experiments were organized under the following settings.

- The codes were implemented in Python 3.6 and Java 11.0, and libraries and machines listed in Table 5.1 were used for experiments. The network settings of TD3 is noted in Table 6.3, where Conv x means a convolutional layers with x filters of size 3×3,Pool means a max pooling with pool size 2×2, and FC x means a fully-connected layers with x nodes. The layer composition was set up according to the general manner [74] of the CNN.

- The length of a level was set to 41 and each level was represented by 15×40 matrix. 10 patterns needs to be generated. The last one tile is for a goal.

- Among these, the first two patterns (Fig. 6.13 (2)) are generated by inputting random noises to CGAN and applying matching, which were initial states.

- In order to generate the first pattern, three imaginary previous patterns (Fig. 6.13 (1)) are required to be given to CGAN. A series of three

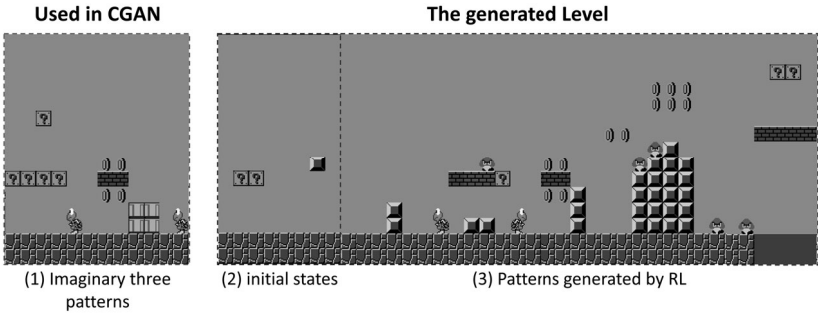


Figure 6.13: Imaginary patterns and generated patterns.

consecutive patterns are selected from Pat_{legal}. These patterns are only used for generating the first three patterns and not shown in the results.

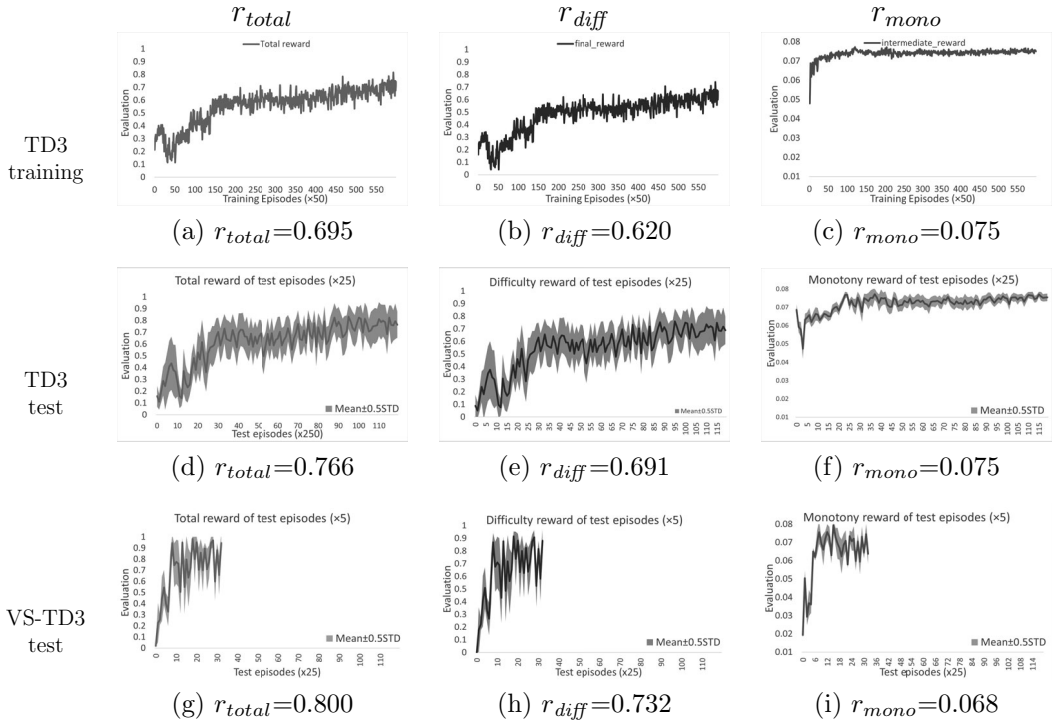
- Since two patterns were initialized in advance, 8 patterns (Fig. 6.13 (3)) needs to be generated by the actor of RL. The last one tile is for a goal.
- Virtual simulation was applied from the beginning of the training. The number of virtual simulating episodes was set to 5. This means that 5 stages are generated for each state transition. Since there were 7 state transitions per episodes, except for the last transition, it took approximately 36 times longer per episode. Each evaluation involved 10 trials of AI agents. In total, 360 trials of simulation from AI agents were conducted.

Result

TD3 was trained for 30,000 episodes, and VS-TD3 was trained for 1,000 episodes. One episode means generating one level from an initial one. For a fair assessment, VS-TD3 was trained for the same amount of time as TD3.

Once training is done, RL can generate a level instantly. The level's playability was evaluated using the base A* agent within approximately 1-2 seconds. It took around 20 seconds to obtain a difficulty reward of 10 simulations from AI agents. Then an episode of TD3 including the assessment process takes around 20 seconds. Meanwhile, VS-TD3 uses 700 seconds because it requires 36 times longer time per episode. Overall, 30,000 episodes for TD3 took around 160 hours, and VS-TD3 took 190 hours.

The learning curves are shown in Fig. 6.14, in a 3×3 matrix style. Upper three graphs are rewards in training phase of TD3. Middle three are rewards



(a) r_{total}=0.695 (b) r_{diff}=0.620 (c) r_{mono}=0.075

(d) r_{total}=0.766 (e) r_{diff}=0.691 (f) r_{mono}=0.075

(g) r_{total}=0.800 (h) r_{diff}=0.732 (i) r_{mono}=0.068

Figure 6.14: The average evaluations of levels: (top) TD3's training curves of the 10-pattern level. Each plot represented the average evaluation from 50 episodes. (middle) TD3's 25-trial of the 10-pattern level in test phase. (bottom) VS-TD3's 5-trial of the 10-pattern level in test phase. Each value indicates the average reward of last 20%.

Table 6.3: TD3 setup

Parameter	Value
Layers	Conv 32→32→Pool→ 64→64→Pool→ FC 256→128→128
Memory size	40000
Learning rate	Actor:5×10^{-6} Critic: 5×10^{-5}
Target network	soft update (5×10^{-5})
Batch size	32
Discount factor	0.99
Exploration	OU noise with ϵ-greedy (ϵ 0.5 → 0.1 at 30% of episodes)

in test phase of TD3, and bottom three are those of VS-TD3. Red (left) three show r_{diff}, Green (center) show r_{mono}, and Blue (right) show r_{total}.

Fig. 6.14 a, b, and c show the training curves of TD3 on generating levels. The average rewards of last 20% of total episodes, difficulty, and monotony were 0.695, 0.620, and 0.075. The trend of the learning curve indicates that the RL model can learn how to generate levels.

In every 250 episodes, 25 levels were generated as a test phase of TD3. The mean evaluations and the standard deviations of the levels are shown in Fig. 6.14 b, e, and f. The average rewards of last 20% of total episodes, difficulty, and monotony were 0.766, 0.691, and 0.075. This result is better than that of the training phase because there is no exploration (e.g., ϵ-greedy) applied.

For every 25 episodes, five levels were generated as a test phase of VS-TD3. The mean evaluations and the standard deviations of the levels are shown in Fig. 6.14 g, h, and i. The average rewards of last 20% of total episodes, difficulty, and monotony were 0.800, 0.732, and 0.068. Relatively small numbers of episodes were trained using VS-TD3 because a single episode requires a long time. However, VS-TD3 obtained a better reward than TD3 in the very early training.

Some levels (2 out of 50 in TD3 and 3 out of 50 in VS-TD3) were unplayable. Thus, it may be necessary to recreate those unplayable levels when applied in practical.

We estimated how much time it takes to get a high-quality level by ran-

dom generation and TD3. As mentioned above, VS-TD3 can produce a level with an expected average of 0.8 evaluation value in 1-2 seconds, including playability assessments. The generate-and-test process must be involved to obtain a level with an 0.8 evaluation value by random generation. The evaluation process takes 20 seconds. Thus, even if 1 out of 3 with 0.8 is obtained, it will take 1 minute.

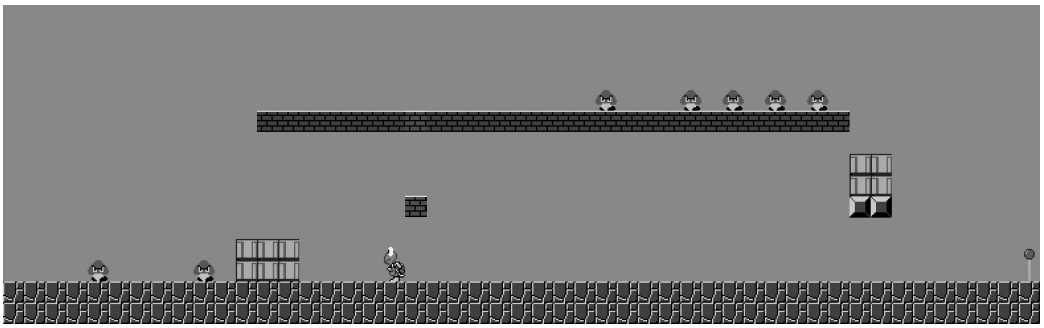
Generated levels

Fig. 6.15 shows typical examples of levels generated during training with TD3. As shown in Fig. 6.14 (d-f), in the early phase of training, the obtained level (a) had low difficulty (0.18) and monotony evaluation value (0.047). As training progresses, the evaluation of monotony was learnt first. Level (b) was relatively difficult (0.592) with high monotony evaluation value (0.075). And then, TD3 generated a level (c) that was both appropriately difficult (0.936) and non-monotonous (0.08).

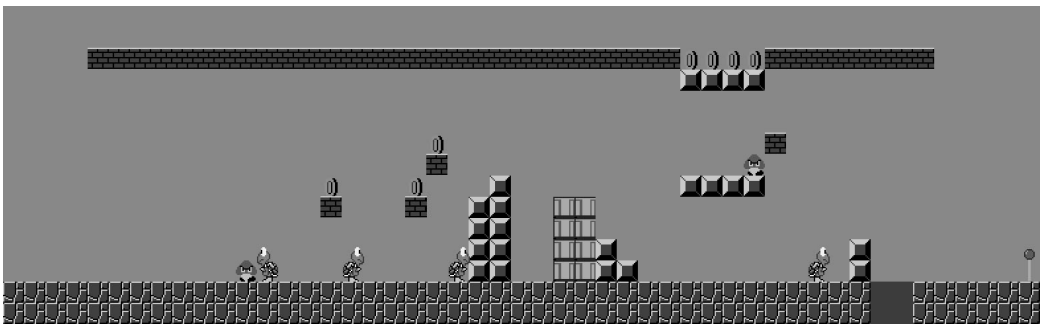
Fig. 6.16 illustrates three levels generated by TD3 (a, b) and random (c, d). Each level has the evaluation values of (0.992+0.08), (0.856+0.07), (0.236+0.064), and (0.1+0.057), in order. The upper levels have several patterns with holes and enemy tiles. Therefore, players are more likely to make mistakes from the inaccuracies of human nature. In contrast, the level (c) has fewer holes with fewer enemies and the level (d) has too many holes. Thus they are evaluated low.

Levels generated from randomly selecting patterns can result in high evaluation values. However, it does not guarantee naturalness since it is ensured by CGAN, not by the evaluation function. Fig. 6.17 shows two levels with an evaluation value over 0.9 generated by TD3 (a) and by randomly selecting patterns from Pat_{legal} (b). Each level has the evaluation values of (0.936+0.08) and (0.953+0.075), respectively. In level (a), patterns are well-connected (e.g., patterns (5-6) have a brick connection, patterns (8-9) have a block connection). In level (b), each pattern has no apparent connection and is independent.

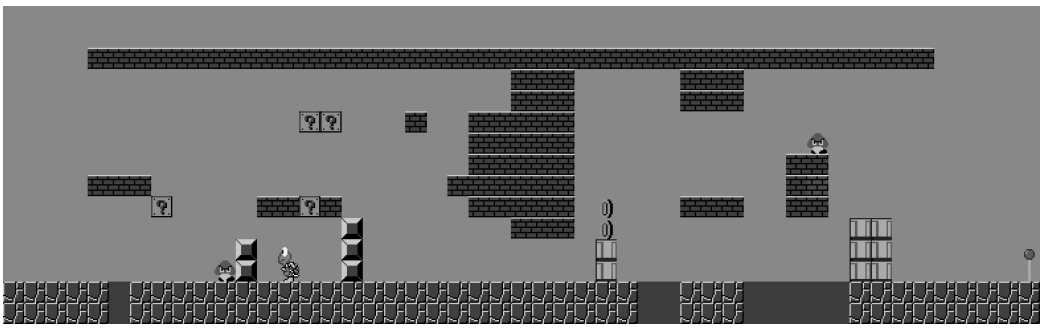
The capability of the proposed approach to generate high-quality stages in Turn-based RPG is highlighted in Section 5.2.1. The above results also indicate promising outcomes that the RL model can learn a policy to generate highly evaluated levels in Super Mario. Thus, it is reasonable to conclude that the proposed approach is promising to generate high-quality content.



(a) r_{diff}=0.18, $total_{dmg}$=1.2, r_{mono}=0.047

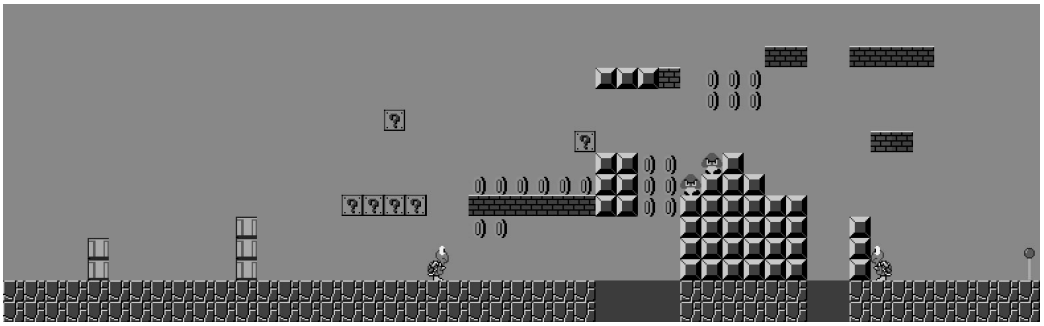


(b) r_{diff}=0.592, $total_{dmg}$=2.23, r_{mono}=0.075

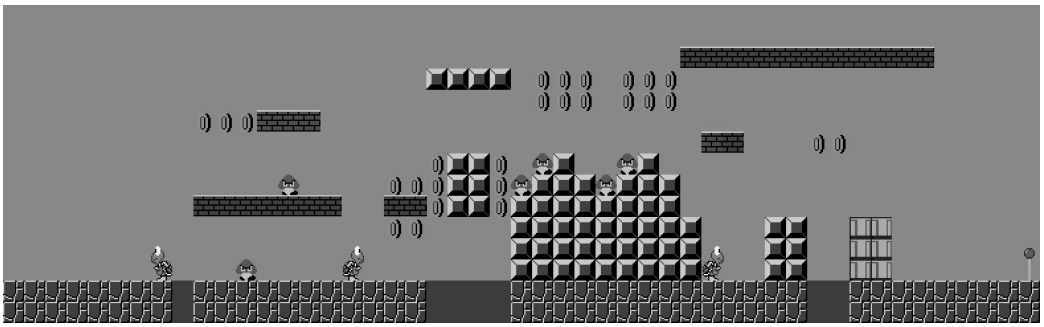


(c) r_{diff}=0.936, $total_{dmg}$=3.09, r_{mono}=0.08

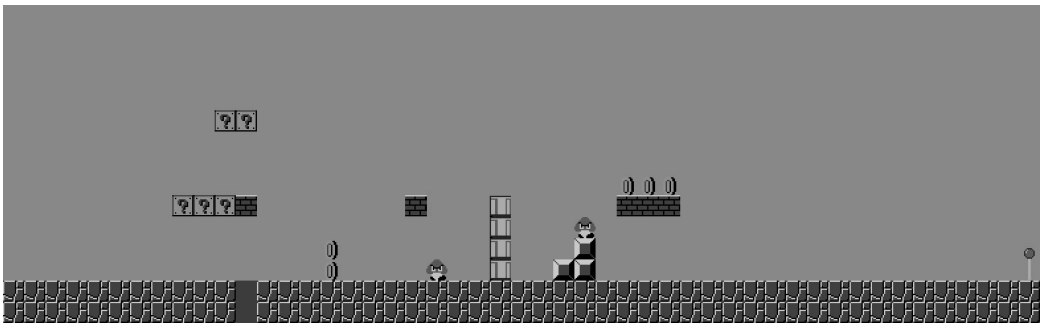
Figure 6.15: Examples of levels generated over the training.



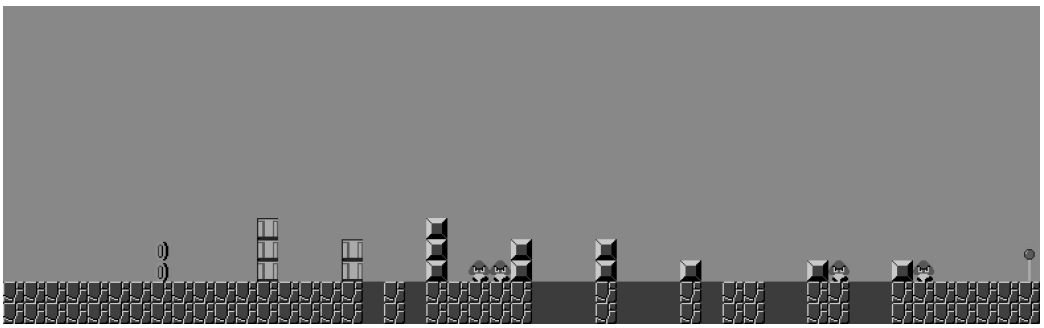
(a) r_{diff}=0.992, $total_{dmg}$=3.23, r_{mono}=0.08



(b) r_{diff}=0.856, $total_{dmg}$=2.89, r_{mono}=0.07

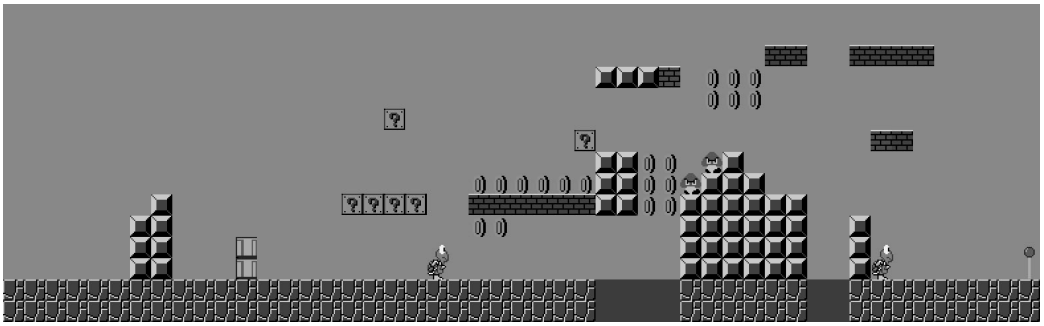


(c) r_{diff}=0.236, $total_{dmg}$=1.34, r_{mono}=0.064

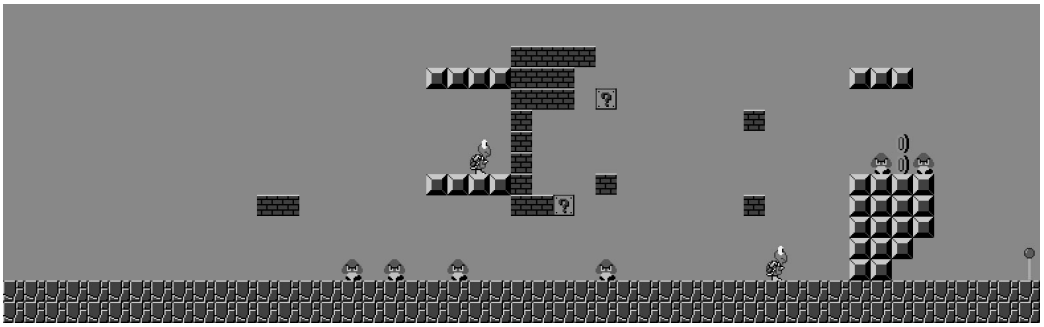


(d) r_{diff}=0.1, $total_{dmg}$=7.325, r_{mono}=0.057

Figure 6.16: Examples of levels generated by TD3 (top two) and random (bottom two).



(a)



(b)

Figure 6.17: Examples of levels with an evaluation value over 0.9 generated by TD3 (top) and random (bottom).

6.6.2 Diverse level generation

Section 6.6.1 shows that high-quality levels can be obtained with the proposed method. Next, we investigated whether the diversity of levels can be obtained while maintaining sufficient quality.

Two indicators to evaluate the diversity of the generated levels are discussed. The proposed methods for improving diversity, RI and DAGP, are applied and investigated. [3]

Diversity assessment

The first indicator is the number of different tiles between the two levels. It is an intuitive and easy-to-understand indicator to show the differences between levels.

The other is the KL-divergence between two levels (KL_{level}). Each wall and enemy tile are converted to the value, 2, 3, respectively. Then, KL-divergence of (40×15)-dimensional vectors of two patterns are calculated. The value of KL_{level} between two levels indicates a comparison of the distribution of tiles arrangement of the two levels. If the value of KL_{level} between two levels is low, the levels have the similar placements of walls and enemies even if the levels contain different tiles.

Fig. 6.18 shows four patterns with the same arrangement of three blocks. Compared to the first pattern, the second to fourth patterns have 6 different tiles. However, in terms of diversity, players may feel that the second pattern is the most similar and the fourth pattern is the least similar. KL_{level} can reflect this diversity.

Result

A diverse generation generally requires a well-trained model. Therefore, VS-TD3 was employed in this experiment. The VS-TD3 model is the same as that used in 6.6.1.

First, an experiment was conducted on randomized initialization (RI). 500 levels were generated for each integer $c \in [1, 8]$ where the first c patterns were randomly generated. Table 6.4 shows the average KL_{level}, numbers of different tiles and rewards for each c. Note that the levels contained

[3] In our stage generation problem, preliminary experiment is conducted to apply the Diversity-aware greedy policy (DAGP). We sampled the Q value and the evaluation values and investigated their relationship in Section 5.2.2. The Super Mario's evaluation function from stochastic agents has stochastic value due to the stochastic behavior of the AI agents. This raises a challenge in sampling the distribution of evaluation values. Therefore, the investigation of the relation between Q-value and evaluation is omitted.

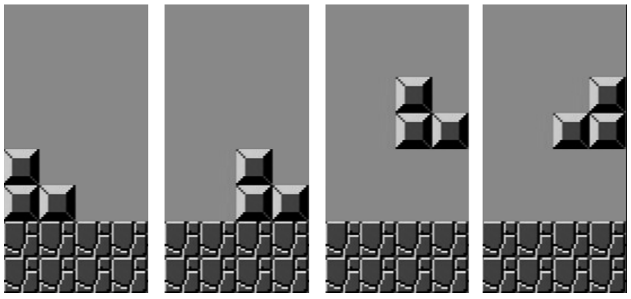


Figure 6.18: Four patterns of comparison by KL-divergence.

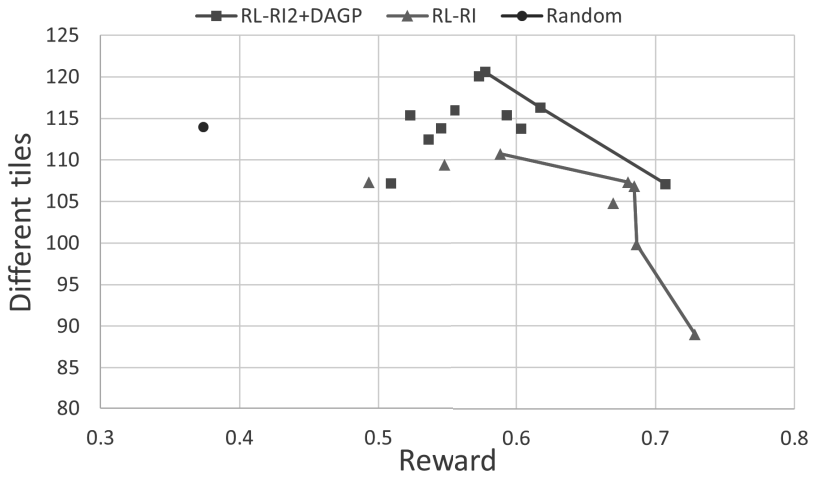
Table 6.4: Average kl_{level}, the average number of different tiles between the two levels, and average rewards from 500 levels generated when the first c patterns were randomly initialized.

c	1	2	3	4
KL_{level}	1.569 (±0.812)	1.817 (±0.734)	1.939 (±0.686)	1.911 (±0.644)
Different tiles	89 (±48)	100 (±42)	107 (±42)	105 (±39)
Reward	0.728 (±0.365)	0.687 (±0.392)	0.685 (±0.377)	0.670 (±0.394)
c	5	6	7	8
KL_{level}	1.979 (±0.644)	2.025 (±0.626)	2.000 (±0.626)	1.939 (±0.570)
Different tiles	110 (±41)	111 (±39)	110 (±40)	107 (±37)
Reward	0.680 (±0.377)	0.589 (±0.384)	0.547 (±0.346)	0.493 (±0.343)

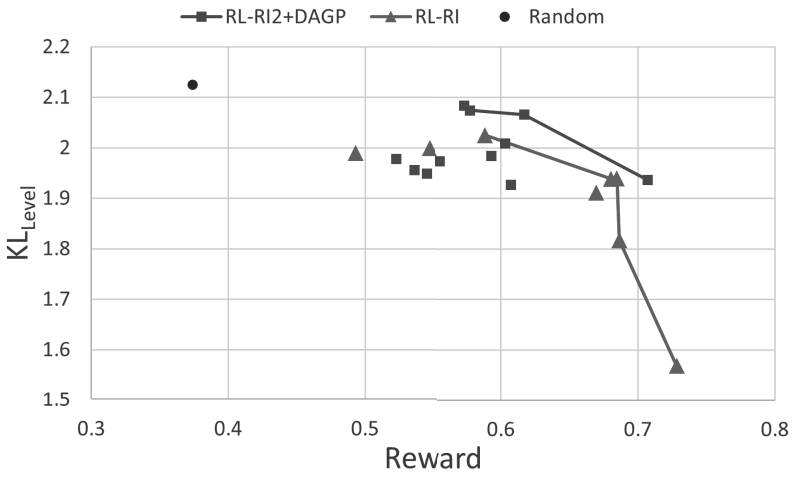
10 patterns in total. As expected, the result shows that the diversity has an increasing trend while the average rewards decrease in patterns that are more randomly initialized.

DAGP generated n candidates. Among them, those that are at least d away from Actor's action are gathered. Then, the method selects the one with the highest Q-value. DAGP was also applied in the Super Mario levels generation. The first two patterns were randomly initialized, and DAGP were used in deciding 4th, 5th, and 6th patterns. The influence of DAGP is investigated by generating levels from the same initial levels used in RI. KL-divergence between patterns ($KL_{pattern}$) is used as a distance threshold d. Several configurations of d and n were tested, and 500 levels were generated from each setting.

Fig. 6.19 compares the rewards from applying only RI (RL-RI), RI with DAGP (RL-RI+DAGP), and random generation. d=0.05, 0.1, 0.2, and n=5, 10, 20, 40 are used as settings of DAGP. The results and their Pareto frontier of the average rewards, KL_{level} and different tiles are displayed. Based on the result, random generation is inefficient in generating quality levels. Though RI could generate quality content, better quality is achieved with sacrificed



(a)



(b)

Figure 6.19: Comparison between applying only RI, RI with DAGP and random generation when d=0.05, 0.1, 0.2, and n=5, 10, 20, 40.

diversity. RL-RI2+DAGP showed better results RL-RI in terms of both quality and diversity. It can generate diverse levels without substantial loss of quality.

Generated levels

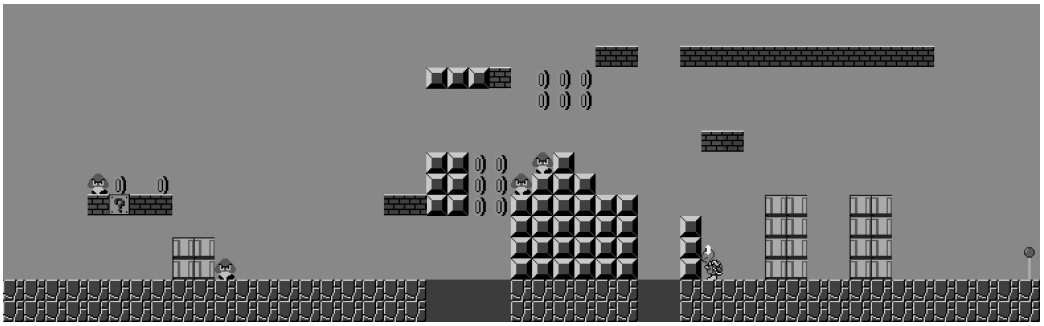
Fig. 6.20 shows example levels generated by VS-TD3 (top) and DAGP (middle and bottom). The former was evaluated as 0.991 (0.916+0.075), and the latters as 0.899(0.824+0.075) and 0.913(0.840+0.073). The KL_{level} from the VS-TD3 level was 1.748 and 1.586, and the number of different tiles was 126 and 103, respectively. The generation initiated with the same three patterns. However, by applying DAGP, diverse and good levels were generated.

6.7 Chapter conclusion

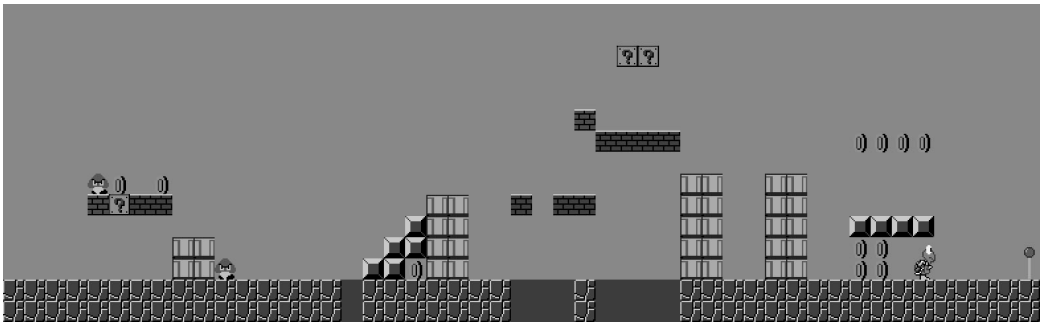
This chapter outlines the use of reinforcement learning with the theme of level generation in Super Mario. The level generation problem was formulated into a Markov decision process, where states were incompleted or completed levels represented by three binary matrix channels of walls, enemies, and holes. 3-dimensional vectors of actions from the RL model are converted to patterns and filled in the matrices. We defined an evaluation function to consider the difficulty based on human-like mistakes and monotony of levels. With the defined reward function, our proposed method successfully generated high-quality levels.

There were two new challenges in our generating levels for Super Mario. First, the training becomes difficult for environments with a high dimension of action space. The function approximator is employed to map a low-dimension vector to a pattern. Second, it is nearly impossible and meaningless to test every possible action sequence for difficulty assessment. Therefore, human-like AI agents are employed to assess the difficulty of levels for human players.

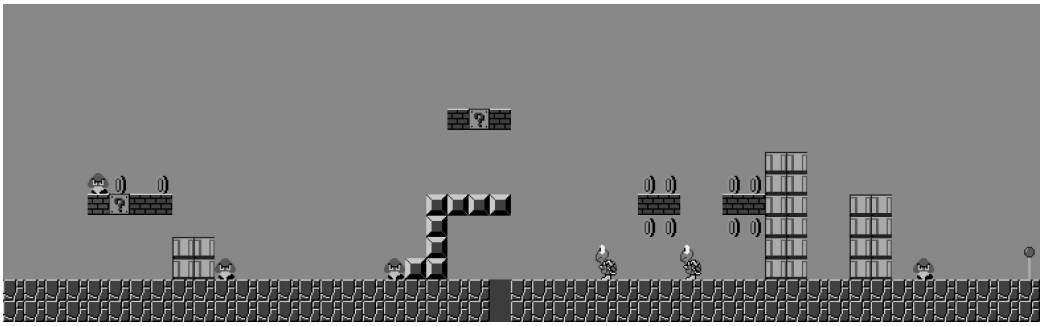
In order to simultaneously achieve high-quality and diverse levels, three proposed methods, virtual simulation, randomized initialization, and diversity-aware greedy policy, are applied. The experiments demonstrated the effectiveness of the proposed approaches in obtaining good and diverse levels.



(a)



(b)



(c)

Figure 6.20: Levels generated by the actor policy (top) and DAGP (middle and bottom) with the first two patterns fixed and the fourth, fifth, sixth patterns were set to noise patterns (bottom).

Chapter 7

Conclusions and future research directions

In this chapter, Section 7.1 concludes the research conducted in this dissertation and answers the research questions raised in Chapter 1. Then, Section 7.2 describes the possible research direction.

7.1 Conclusions

This dissertation is commenced in response to the three research questions raised. The first question is, what kind of method can handle the lack of training data and the need for an online generation? This dissertation addresses this question by proposing a method that functions under lacking training data and can generate content online. The use of RL to generate stages in turn-based RPG and levels in Super Mario is proposed. The rationale for employing RL is that it does not need training data and can generate potentially diverse content online.

The stage and level generation problems were formulated into a Markov decision process. States were incomplete or complete stages/levels represented by real-number matrices, and actions were to generate real-number vectors. In turn-based RPG, vectors were embedded directly in the matrices. Meanwhile, Super Mario has a challenge in converting vectors directly into patterns. Therefore, CGAN was introduced as a function approximator that maps vectors to patterns. Vectors were converted to the patterns through the function approximator, and the patterns were filled in level matrices.

The preliminary experiment highlighted the importance of connectivity between patterns. As a result, previous patterns are used as inputs for CGAN to ensure natural connectivity between patterns. In order to resolve the noise

of the CGAN output, Matching was adopted to select the closest pattern from the generated pattern. Experimental results showed that raw and matched CGAN outputs have similar distributions. The function approximator can perform a mapping that considers the natural connectivity before and after patterns.

Evaluation functions were defined regarding the difficulty and entertaining factors of stages/levels. In turn-based RPG, every possible strategy was tested. Then, the winning rate was used as a difficulty indicator and combined with several entertaining factors. In Super Mario, however, human-like AI agents are employed due to the game characteristics where actual human play logs cannot be simply and economically acquired. AI agents are coupled with the inaccuracies of human nature and the long press to imitate human gameplay nature. The difficulty indicator is from its play logs. The evaluation function was derived from the difficulty indicator and the monotony evaluation. Using the evaluation function to reward the RL, methods proposed in this dissertation successfully rendered highly evaluated stages/levels.

DQN and DDPG for discrete and continuous actions were applied in turn-based RPGs to investigate how action space affects the PCG problem. The range was divided into a fixed number of sections for discrete actions, and one value in the stage matrix was decided at a time. DDPG generated better stages than DQN under the experiment settings used in this dissertation. In Super Mario, action decision is continuous. Therefore, several values (of related events) were decided simultaneously, which was expected to handle related events more effectively. A continuous action model named TD3 was employed to generate Super Mario levels. The output of RL is then transferred as the input of the function approximator.

The second research question is, can the method obtain both diversity and good quality? Our reward design generates delayed or small signal rewards. This results in challenges for the model to efficiently learn to evaluate actions, especially for long event/pattern sequences. To overcome the difficulty, we introduced virtual simulations to provide intermediate actions with rewards. Experiments showed that virtual simulations helped improve the quality of stages/levels.

The primary purpose of this dissertation is to generate good and diverse stages/levels. We thus proposed randomized initialization that assigns several random events/patterns to the beginning stages/levels. We then employ a diversity-aware greedy policy that determines actions distant but not bad compared to the best actions from the model's view. The experiments demonstrated the effectiveness of the proposed approaches in obtaining good and diverse stages/levels for an online generation.

The final research question is, which games can the method be applied? The proposed method is firstly applied to a relatively straightforward genre, the turn-based RPGs. Then is it applied to the Super Mario, a more complex genre. Each game is different in nature and has distinct challenges regarding content generation problems. The method successfully generates high-quality and diverse stages/levels for both games. We expect the proposed method can handle any generation problem, even a relatively complicated one, given that modifications are made according to the game's nature. The method is applicable for generation problems as long as it can be formulated and a good evaluation function and distance measurements are given.

7.2 Future research directions

Future works include stabilizing the methods for increasing diversity in Super Mario. Evaluation values in Super Mario were stochastic. Thus, the Q and evaluation values may not have high positive correlations. That made it challenging to apply a diversity-aware greedy policy directly. RL models need to learn suitable Q-values under the stochastic reward environments. It is possible to stabilize the average reward by employing many AIs. However, the experiment is costly and thus unrealistic.

Another possible research direction is related to designing evaluation functions. Our work highlighted the difficulty and monotony in Super Mario. However, other elements, such as the placement of the coins and mushrooms, are essential to the entertainment of Mario's levels and, thus, should be taken into account. The other direction is to reflect human players' sentiments in gameplays and verify the effectiveness. We expect our method to try to maximize any given evaluation function as the current one is already considerably complicated. For example, we can try to conduct subject experiments for collecting human players' evaluations and using supervised learning to approximate those evaluations. By using the supervised learning model as the evaluation function, the method should be able to generate stages that fit the players' preferences.

Acknowledgement

It was a long journey from the master's to the doctoral course. This journey cannot be completed without massive support from many people. I would like to dedicate this section to offering my gratitude to them.

First and foremost, I would like to express my gratitude to my supervisor, Prof. Dr. Kokolo Ikeda, for his guidance in this Ph.D. journey. I faced many doubts and challenges during my research, which are now resolved with his intellectual support. I cannot find any words to express my gratitude to him.

I would also like to thank Asst. Prof. Dr. Chu Hsuan Hsueh, my assistant professor. She helped with the journal work, gave many constructive comments during the doctoral process, and contributed significantly to the research collaboration.

Besides, I would like to thank my committee members, Prof. Dr. Hiroyuki Iida, Prof. Dr. Shinobu Hasegawa, Assoc. Prof. Dr. Kiyoaki Shirai, and Prof. Dr. Ruck Thawonmas for contributing their time and efforts.

Finally, I am grateful for the support from my family, my beloved parents and sisters. I also would like to express my special thanks and love to Ms. Pavinee Rerkjirattikal for her great support in JAIST life.

Acheivement list

Domestic conference

1. SangGyu Nam, Kokolo Ikeda, Automatic Generation of States in Turn-Based RPG Using Reinforcement Learning, 23rd game programming workshop (GPW), pp 160-167, 2018-11, Haneda

2. Sila Temsiririrkkul, Takahashi Kazuyuki, SangGyu Nam, Kokolo Ikeda, An Investigation of Human-likeness in Computer Game Players, Information Processing Society of Japan, 40th GI, vol 7, pp 1-6, 2018-6, Kochi University of Technology

International conference

1. SangGyu Nam, Kokolo Ikeda, Generation of Diverse Stages in Turn-Based RPG using Reinforcement Learning, In Conference On Games (COG), pp 1-8, 2019-8, London

2. Chu-Hsuan Hsueh, Kokolo Ikeda, SangGyu Nam, and I-Chen Wu, Analyses of Tabular AlphaZero on NoGo, The 2020 Conference on Technologies and Applications of Artificial Intelligence (TAAI), pp 254-259, 2020-12, Taiwan

International journal

1. SangGyu Nam, Chu-Hsuan Hsueh, and Kokolo Ikeda. "Generation of Game Stages with Quality and Diversity by Reinforcement Learning in Turn-based RPG." IEEE Transactions on Games (TOG), pp 1-14, 2021

Bibliography

[1] A. Summerville and M. Mateas, "Super mario as a string: Platformer level generation via lstms." in Proc. 1st Int. Joint Conf. DiGRA/FDG, 2016.

[2] E. C. Sheffield and M. D. Shah, "Dungeon digger: Apprenticeship learning for procedural dungeon building agents," ser. CHI PLAY '18 Extended Abstracts, 2018, p. 603–610.

[3] J. Togelius, M. Preuss, and G. N. Yannakakis, "Towards multiobjective procedural map generation," in *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, ser. PCGames '10, 2010.

[4] G. Pickett, F. Khosmood, and A. Fowler, "Automated generation of conversational non player characters," in *INT/SBG@AIIDE*, 2015.

[5] D. Gravina and D. Loiacono, "Procedural weapons generation for unreal tournament iii," in *2015 IEEE Games Entertainment Media Conference (GEM)*, 2015, pp. 1–8.

[6] N. Justesen, R. Rodriguez Torrado, P. Bontrager, A. Khalifa, J. Togelius, and S. Risi, "Illuminating generalization in deep reinforcement learning through procedural level generation," *NeurIPS Workshop on Deep Reinforcement Learning*, 2018.

[7] 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, p. 221–228.

[8] J. Togelius, R. De Nardi, and S. M. Lucas, "Towards automatic personalised content creation for racing games," in *2007 IEEE Symposium on Computational Intelligence and Games*, 2007, pp. 252–259.

[9] D. Loiacono, L. Cardamone, and P. L. Lanzi, "Automatic track generation for high-end racing games using evolutionary computation," *IEEE Transactions on Computational Intelligence and AI in Games*, pp. 245–259, 2011.

[10] A. Liapis, C. Holmgård, G. N. Yannakakis, and J. Togelius, "Procedural personas as critics for dungeon generation," in *Applications of Evolutionary Computation*, 2015, pp. 331–343.

[11] Red Hook Studios, *Vancouver, Canada*, Darkest Dungeon, 2016.

[12] Chunsoft, *Tokyo, Japan*, Mystery Dungeon series, 1993.

[13] Nippon Ichi Software, *Gifu, Japan*, Disgaea, 2003.

[14] 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.

[15] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," in *2nd International Conference on Learning Representations, ICLR*, 2014.

[16] van den Oord et al., "Conditional image generation with pixelcnn decoders," in *Advances in Neural Information Processing Systems 29*, 2016, pp. 4790–4798.

[17] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in Neural Information Processing Systems 27*. Curran Associates, Inc., 2014, pp. 2672–2680.

[18] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation," in *Applications of Evolutionary Computation*, 2010, pp. 141–150.

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

[20] S. Nam and K. Ikeda, "Automatic generation of states in turn-based rpg using reinforcement learning," in *2018 Game Programming Workshop (GPW)*, vol. 2018, nov 2018, pp. 160–167.

[21] S. Nam and K. Ikeda, "Generation of diverse stages in turn-based role-playing game using reinforcement learning," in *2019 IEEE Conference on Games (CoG)*, 2019, pp. 1–8.

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

[23] S. Nam, C.-H. Hsueh, and K. Ikeda, "Generation of game stages with quality and diversity by reinforcement learning in turn-based rpg," *IEEE Transactions on Games*, 2021.

[24] D. Gravina, A. Khalifa, A. Liapis, J. Togelius, and G. N. Yannakakis, "Procedural content generation through quality diversity," in *2019 IEEE Conference on Games (CoG)*, Aug 2019, pp. 1–8.

[25] A. Liapis, G. N. Yannakakis, and J. Togelius, "Enhancements to constrained novelty search: Two-population novelty search for generating game content," in *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, 2013, p. 343–350.

[26] D. Hooshyar, M. Yousefi, M. Wang, and H. Lim, "A data-driven procedural-content-generation approach for educational games," *Journal of Computer Assisted Learning*, vol. 34, no. 6, pp. 731–739, 2018.

[27] A. Zook, S. Lee-Urban, M. R. Drinkwater, and M. O. Riedl, "Skill-based mission generation: A data-driven temporal player modeling approach," in *Proceedings of the The Third Workshop on PCG'12*, 2012, p. 1–8.

[28] Y. Liang, W. Li, and K. Ikeda, "Procedural content generation of rhythm games using deep learning methods," in *Entertainment Computing and Serious Games*, 2019, pp. 134–145.

[29] M. Kaidan, C. Y. Chu, T. Harada, and R. Thawonmas, "Procedural generation of angry birds levels that adapt to the player's skills using genetic algorithm," in *IEEE 4th Global Conference on Consumer Electronics*, 2015, pp. 535–536.

[30] T. Oikawa, C.-H. Hsueh, and K. Ikeda, "Improving human players' t-spin skills in tetris with procedural problem generation," *16th Advances in Computer Games (ACG 2019)*, 2019.

[31] 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*, 2011.

[32] M. Stephenson and J. Renz, "Procedural generation of complex stable structures for angry birds levels," in *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, 2016, pp. 1–8.

[33] A. Alvarez, S. Dahlskog, J. Font, and J. Togelius, "Empowering quality diversity in dungeon design with interactive constrained map-elites," in *2019 IEEE Conference on Games (CoG)*, 2019, pp. 1–8.

[34] S. Lee, A. Isaksen, C. Holmgård, and J. Togelius, "Predicting resource locations in game maps using deep convolutional neural networks," in *Proc. Artif. Intell. Interactive Digit. Entertainment Conf*, 2016.

[35] A. Summerville, M. Behrooz, M. Mateas, and A. Jhala, "The learning of zelda: Data-driven learning of level topology," *in Proc. 10th Int. Conf. Found. Digit. Games*, 2015.

[36] M. Awiszus, F. Schubert, and B. Rosenhahn, "Toad-gan: Coherent style level generation from a single example," *Artificial Intelligence and Interactive Digital Entertainment. AAAI*, pp. 10–16, 2020.

[37] R. Rodriguez Torrado, A. Khalifa, M. Cerny Green, N. Justesen, S. Risi, and J. Togelius, "Bootstrapping conditional gans for video game level generation," in *IEEE Conference on Games (CoG)*, 2020, pp. 41–48.

[38] E. Giacomello, P. L. Lanzi, and D. Loiacono, "Doom level generation using generative adversarial networks," in *2018 IEEE Games, Entertainment, Media Conference (GEM)*, 2018, pp. 316–323.

[39] K. Park et al., "Generating educational game levels with multistep deep convolutional generative adversarial networks," in *2019 IEEE Conference on Games (CoG)*, 2019, pp. 1–8.

[40] R. Jain, A. Isaksen, C. Holmgard, and J. Togelius, "Autoencoders for level generation, repair, and recognition." In ICCC Workshop on Computational Creativity and Games, 2016.

[41] A. Sarkar, Z. Yang, and S. Cooper, "Controllable level blending between games using variational autoencoders," in *Proceedings of the EXAG Workshop at AIIDE*, 2019.

[42] S. Thakkar, C. Cao, L. Wang, T. J. Choi, and J. Togelius, "Autoencoder and evolutionary algorithm for level generation in lode runner," in *2019 IEEE Conference on Games (CoG)*, 2019, pp. 1–4.

[43] A. Sarkar, A. Summerville, S. Snodgrass, G. Bentley, and J. Osborn, "Exploring level blending across platformers via paths and affordances," *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 16, no. 1, pp. 280–286, 2020.

[44] M. Minsky, "Steps toward artificial intelligence," *Proceedings of the IRE*, vol. 49, no. 1, pp. 8–30, 1961.

[45] A. P. Badia, B. Piot, S. Kapturowski, P. Sprechmann, A. Vitvitskyi, Z. D. Guo, and C. Blundell, "Agent57: Outperforming the Atari human benchmark," in *Proceedings of the 37th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, H. D. III and A. Singh, Eds., vol. 119. PMLR, 13–18 Jul 2020, pp. 507–517.

[46] J. Ferret, R. Marinier, M. Geist, and O. Pietquin, "Self-attentional credit assignment for transfer in reinforcement learning," in *IJCAI*, 2020, pp. 2655–2661.

[47] J. A. Arjona-Medina, M. Gillhofer, M. Widrich, T. Unterthiner, J. Brandstetter, and S. Hochreiter, "Rudder: Return decomposition for delayed rewards," in *Advances in Neural Information Processing Systems*, 2019.

[48] Y. Liu, Y. Luo, Y. Zhong, X. Chen, Q. Liu, and J. Peng, "Sequence modeling of temporal credit assignment for episodic reinforcement learning," *CoRR*, vol. abs/1905.13420, 2019.

[49] A. Harutyunyan, W. Dabney, T. Mesnard, M. Gheshlaghi Azar, B. Piot, N. Heess, H. P. van Hasselt, G. Wayne, S. Singh, D. Precup, and R. Munos, "Hindsight credit assignment," in *Advances in Neural Information Processing Systems*, 2019, pp. 12 488–12 497.

[50] L. Yu, W. Zhang, J. Wang, and Y. Yu, "Seqgan: Sequence generative adversarial nets with policy gradient," in *Proceedings of the Thirty-First AAAI'17 Conference on Artificial Intelligence*, p. 2852–2858.

[51] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran,

D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, feb 2015.

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

[53] M. Guzdial, N. Liao, and M. Riedl, "Co-creative level design via machine learning," in *Proceedings of the EXAG Workshop at AIIDE*, vol. 2282, 2018.

[54] W. J. Kavanagh, A. Miller, G. Norman, and O. Andrei, "Balancing turn-based games with chained strategy generation," *IEEE Transactions on Games*, pp. 1–1, 2019.

[55] A. Pantaleev, "In search of patterns: Disrupting rpg classes through procedural content generation," ser. PCG'12. Association for Computing Machinery, 2012, p. 1–5.

[56] Nintendo, *Kyoto, Japan*, Super Mario Bros., 1985.

[57] gamecubicle, *Mario Sales Data*, www.gamecubicle.com/features-mario-units_sold_sales.htm.

[58] Nintendo, *Kyoto, Japan*, Super Mario Maker, 2019.

[59] mx0c, *super-mario-python*, https://github.com/mx0c/super-mario-python, (2021).

[60] 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.

[61] KYLE ORLAND, *How a speedrunner broke Super Mario Bros.' biggest barrier*, https://arstechnica.com/gaming/2021/04/new-super-mario-bros-record-breaks-speedrunnings-four-minute-mile/.

[62] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, 2016, Conference Track Proceedings*.

[63] O. Missura and T. Gärtner, "Player modeling for intelligent difficulty adjustment," in *Discovery Science*, 2009, pp. 197–211.

[64] M. Lankes and A. Stoeckl, "Gazing at pac-man: Lessons learned from a eye-tracking study focusing on game difficulty," in *ACM Symposium on Eye Tracking Research and Applications*, 2020.

[65] B. Bostan and S. Ogut, "Game challenges and difficulty levels: lessons learned from rpgs," in *International Simulation and Gaming Association Conference*, 2009.

[66] N. Sato, K. Ikeda, and T. Wada, "Estimation of player's preference for cooperative rpgs using multi-strategy monte-carlo method," in *2015 IEEE Conference on Computational Intelligence and Games*, pp. 51–59.

[67] A. Tavakoli, F. Pardo, and P. Kormushev, "Action branching architectures for deep reinforcement learning," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, Apr. 2018.

[68] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," in *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, J. Dy and A. Krause, Eds., vol. 80. PMLR, 10–15 Jul 2018, pp. 1587–1596.

[69] J. Togelius, S. Karakovskiy, and R. Baumgarten, "The 2009 mario ai competition," 08 2010, pp. 1–8.

[70] T. Kohonen, "The self-organizing map," *Proceedings of the IEEE*, vol. 78, no. 9, pp. 1464–1480, 1990.

[71] M. Mirza and S. Osindero, "Conditional generative adversarial nets," *CoRR*, vol. abs/1411.1784, 2014. [Online]. Available: http://arxiv.org/abs/1411.1784

[72] 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)*, 2020, pp. 1–10.

[73] 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.

[74] R. Yamashita, M. Nishio, R. K. G. Do, and K. Togashi, "Convolutional neural networks: an overview and application in radiology," *Insights into Imaging*, vol. 9, pp. 611 – 629, 2018.

Appendix A

The applied structure of CGAN

In many machine learning domains, data processing is often employed as it can significantly impact the network's training. In order to improve that the generator can learn features of Pat_{legal} well, one set of four connected patterns from Pat_{legal} is duplicated in four ways as illustrated in Fig. A.1. There are four ways of cutting a pattern from four connected patterns. The first to the fourth pattern is cut from the data. Then, it is given to the generator with a vector and a one-hot position vector. The discriminator also receives a series of connected patterns with position information. Network model architectures are shown in Fig. 6.7, and the network settings of a generator and a discriminator are listed in Table 6.1. CNN is adopted as a structure of CGAN. Although, LSTM and RNN can also be considered.

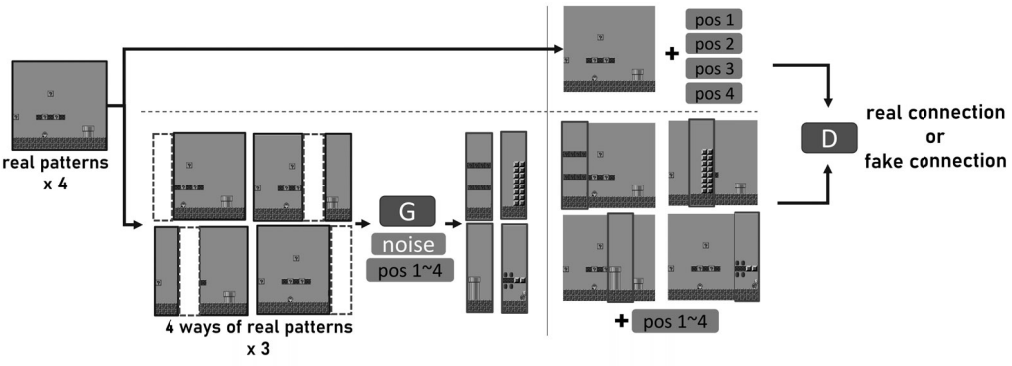


Figure A.1: A structure of CGAN for Super Mario pattern generation used in this dissertation. Training data is duplicated in four ways.