AMERICAN UNIVERSITY OF BEIRUT

PROCEDURAL CONTENT GENERATOR FOR PLATFORMER LEVELS

by
SARAH BASSEM KATERJI

A thesis
submitted in partial fulfillment of the requirements
for the degree of Masters of Science
to the Department of Computer Science
of Faculty of Arts and Sciences
at the American University of Beirut

Beirut, Lebanon
May 2022

AMERICAN UNIVERSITY OF BEIRUT

PROCEDURAL CONTENT GENERATOR FOR PLATFORMER LEVELS

by

SARAH BASSEM KATERJI

Approved by: [illegible]

Dr. Shady Elbassuoni, Associate Professor	Advisor
Computer Science	

[illegible]

Dr. Mohamad El Baker Nassar, Assistant Professor	Member of Committee
Computer Science	

[illegible]

Wassim El Hajj, Associate Professor	Member of Committee
Computer Science	

Date of thesis defense: April 28, 2022

AMERICAN UNIVERSITY OF BEIRUT

THESIS RELEASE FORM

Student Name: ______________________________
Katerji Sarah Bassem

I authorize the American University of Beirut, to: (a) reproduce hard or electronic copies of my thesis; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes

✓ **As of the date of submission of my thesis**

___ **After 1 year from the date of submission of my thesis .**

___ **After 2 years from the date of submission of my thesis .**

___ **After 3 years from the date of submission of my thesis .**

[illegible] 12/05/2022

Signature Date

Acknowledgements

Throughout my journey in writing this thesis, I have received great support and assistance from all the faculty and staff at AUB.

First and foremost I am extremely grateful to my supervisor, Dr. Shady Elbassuoni, and committee members, Dr. Mohamad El Baker Nassar and Dr. Wassim El Hajj for their invaluable expertise, continuous advice, and patience during my Master study.

In addition, I would like to express my gratitude to my parents, and my sister for their tremendous understanding, encouragement, and unconditional support throughout my academic journey.

Last, I would like to thank my colleagues at AUB who provided me with great support and stimulating discussions.

Abstract of the Thesis of

Sarah Bassem Katerji for Masters of Science
Major: Computer Science

Title: Procedural Content Generator For Platformer Levels

Procedural Content Generation (PCG) are algorithms that can generate game content or levels with little to no human intervention. As discussed by PCGML, datasets, and benchmarks in the field of game generation are very limited and lack video gameplay data. Furthermore, there is no unified clear framework for the evaluation of GAN-based PCG algorithms. Therefore, in this thesis, we provide a new clean video gameplay dataset composed of two games Super Mario Bros and Super Mario Bros Lost Levels. We show that learning from language in Platformer PCG outperforms learning from video frames. Moreover, we discuss three approaches to extract meaningful data from the two games to perform learning from language. The approach generates a variety of levels learned from different sources (one level, multiple levels, multiple games). Thus, we show that learning from multiple games is possible with GANs learning from langauge. Furthermore, we categorize several evaluation approaches used in the literature into style difference, playability, and fun and provide an evaluation framework for each to compare different GAN architectures (Simple GAN, DCGAN and WGAN) over different datasets and we show that in most cases WGAN learning outperforms GAN and DCGAN.

Table of Contents

Illustrations

Tables

Abbreviations

ConvNets	Convolutional Neural Networks
CMA-ES	Covariance Matrix Adaptation Evolution Strategy
DCGAN	Deep Convolutional Generative Adversarial Networks
EMD	Earth Mover Distance
FID	Frechet Inception Distance
GAN	Generative Adversarial Networks
LSTM	Long Short-Term Memory
LVE	Latent Variable Evolution
MCTS	Monte Carlos Tree Search
MTM	Multi Template Matching
NPC	Non Playable Character
PCG	Procedural Content Generation
PCGML	Procedural Content Generation via Machine Learning
SMB	Super Mario Bros
SSIM	Structural Similarity Index Measure
W1L1	World 1 Level 1
WGAN	Wasserstein Generative Adversarial Network
WGAN-GP	Wasserstein Generative Adversarial Network + Gradient Penalty

Chapter 1
Introduction

The video game industry is an important sector in the technology field. Every year, thousands of popular games are produced and released. In 2019 alone, around 8290 games were published and played by millions of users around the world [1]. Despite the rewards the industry brings, game development is a very demanding business as it requires large funds to afford needed technology (memory and processing), quality, and skills. [2]. The players' continuous thrive for more game content and faster delivery keeps the game industry pushing forth to meet their customers' expectations. Therefore, gaming companies have a growing need to automate certain game development tasks to deliver a variety of high-end video games in a timely, cost-friendly and efficient manner. [2].

Procedural Content Generation (PCG) is defined as the automated creation of video game content by using algorithms to minimize human intervention in the development process. Such algorithms are used by gaming companies to respond to the player's needs and solve the issues previously mentioned. The very first approach of simple and straightforward PCG was seen in the game "Elite" (1984). The creators of the game used random numbers to make the algorithm generate worlds and levels without the need for explicit code for each level. Using this simple form of PCG allowed them to fit a full game in 22 KB of memory thus saving memory which was a scarce resource at the time [2], [3]. It is interesting to note that PCG in its simplest form (pseudo random generators) is currently extensively used in endless runner games such as Temple run [4]. Indeed, the sole purpose of the game is to go for as long as the player is alive. Thus, the need for PCG to continuously generate a series of platforms and obstacles at run time [5].

However, as shown in the literature, PCG is a promising tool for game developers and designers and is not bound to pseudo-random generators only. Indeed, some advanced applications of PCG include but are not limited to the automation of game components creation, game style transfer, game expanding, and level design. Researchers have been exploring techniques to generalize and parameterize PCG as well as expand its knowledge of how to build a playable and enjoyable game. The ultimate goal of "general game generation" is hard to solve. Therefore, the problem has been approached in the literature as a smaller, more focused goal i.e. the generation of game level structure or level section structure within a specific game genre. Given that, several researchers tackled PCG in different ways. These techniques range from evolutionary algorithms such as "Patterns as Objectives for Level Generation" [2] and probabilistic techniques such as Monte-Carlo tree search [6] to machine learning models such as LSTM [7] and GAN [8]. It is interesting to note that the results presented by most papers are promising but there remain multiple areas of improvement that can be achieved through experimentation. Most of the research is focused on learning game content from one level or a small target group of levels (generalization of PCG). Moreover, a considerable amount of research papers use the source or binary game code that is specific for each game to extract the required data (dataset and language unification). Furthermore, PCG should generate balanced and playable levels (quality of outcome) but there doesn't seem to be a clear unified PCG evaluation technique.

Thus, the main objective of this thesis is to explore, expand and experiment further in procedural content generation in the paltformer game genre. We aim to present the following contributions:

1. Describing the state-of-the-art techniques for learning platformer game design and analyzing their advantages and disadvantages.
2. Exploring and improving platformer dataset.
3. Exploring frame generation in GAN in image format and discussing it's limits.
4. Improving and expanding the work done by Volz et al. [8].
5. Stepping into generalization of PCG by attempting multiple level learning.
6. Proposing a technique for a level-based PCG evaluation and applying it to the improved PCG algorithm.

The remaining of the thesis is organized as follows. Chapter 2 provides a general introduction to key terms and concepts related to this research. Chapter 3 presents a literature review of the work done in the field of platformer level generation as well as a survey of those techniques. Chapter 4 explains the methodology used to build several GAN PCG generators and defines a framework to evaluate these generators. Chapter 5 discusses the process of the experiments and analyses the results with evaluation. Finally, Chapter 6 concludes the paper and summarizes the work.

Chapter 2
Definitions

The following section presents a general introduction of key terms and concepts related to this thesis. We define video games, PCG and GANs and OpenCV.

2.1 Video Games

Understanding what a video game is and what makes a level fun and playable constitute the basis for building a good PCG algorithm. In this section, we will also define the platformer game genre and provide a quick overview of the Super Mario Bros game.

2.1.1 *Definition of Video Games*

In 1949, Johan Huizinga defined the term ”game” as an engaging free activity that happens within its boundaries of time, space and rules [9]. In 1978, Bernard Suits said that playing a game is a voluntary activity in which the player aims to overcome unnecessary obstacles [10]. Today, game designers such as Scott Rogers define games as an activity that requires at least one player, has rules and a victory condition [11]. From these definitions, we can summarize the concept of a game as a set of problems or obstacles that at least one player chooses to solve by their own will. The player is bound by space, time and fixed rules during a game session and can either win or lose. Finally, a video game is a game that is played virtually on a computer machine.

2.1.2 *Genres*

Game genres are labels used to classify games depending on their gameplay style [11]. Some examples of common genres are action, adventure, shooter, strategy and puzzle. In this thesis, we are mostly interested in the 2D platformer genre. They are characterized by their gameplay that relies on jumping and walking on platforms and avoiding obstacles to move the player from a start position to a finish line. A few examples of platformers include popular game titles such Mario, Sonic the hedgehog and Jazz Jackrabbit shown in Figure 2.1. It is interesting to note that such games contain common game elements such as gaps, ground, tiles, enemies, flying platforms and similar mechanics. This observation can help in building a common language for the platformer genre when a high level of abstraction is needed.

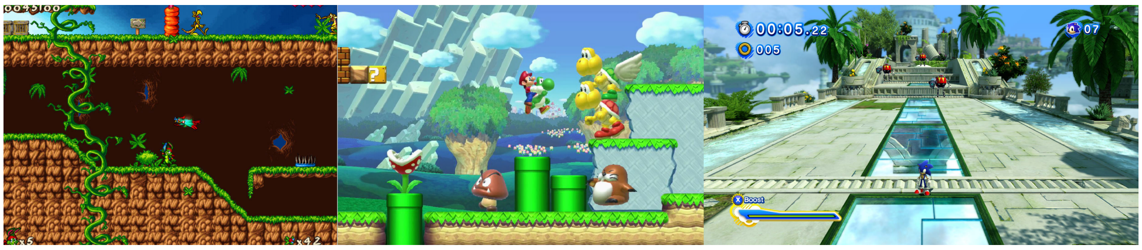
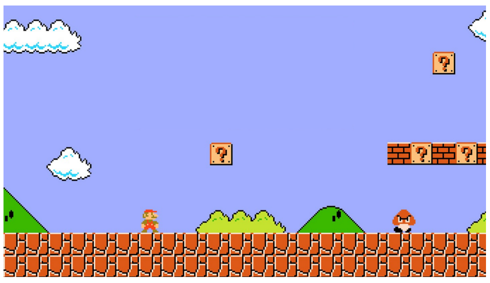


Figure 2.1: Examples of platformer games, from left to right: Jazz Jackrabbit 2 (2D), Mario (2.5D) and Sonic Generations (Mix 3D & 2.5D).

2.1.3 *Super Mario Bros*

Super Mario Bros (SMB) is a 2D platformer video game, published by Nintendo and available on NES (Nintendo Entertainment System). The player needs to control Mario to traverse side-scrolling stages while avoiding hazards and enemies. Interesting bonuses and power-ups are hidden in the blocks [12]. It is worth mentioning that SMB is widely used in research papers due to the block-tile nature of the game's levels as well as its simple design. The game is composed of eight worlds. Every world is made up of three levels and a boss level. Mario's levels have different themes but share similar mechanics with the exception of the boss and underwater levels that are quite different. Thus, most researchers exclude these levels from the level corpus. Figure 2.2a shows a section of SMB (NES) from World 1, Level 1 (W1L1) whereas Figure 2.2b presents a sprite sheet that shows the building blocks/tiles of the game [13].


(a) SMB section

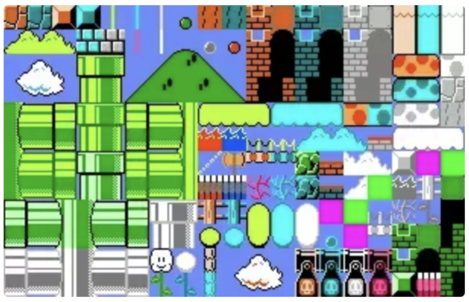

(b) SMB Spritesheet

Figure 2.2: (a) Super Mario Bros level section from world 1 level 1 and (b) sprite sheet containing some of the building blocks used in the game.

2.1.4 *Level Attributes*

A level is considered to be a good level if it is:

1. **Playable:** we define a playable level as a level in which the player can transition from the "start" state to the "end state". An example of non-playable platformer level is one in which the player's jump ability cannot overcome a wall that is too high and thus, keeps the player from reaching the finish line. Unplayable levels can cause player frustration and as consequence, a good PCG algorithm should maximize the playability of a level.

2. **Fun:** we define fun levels as levels that are enjoyable to play. However, stating whether a level is enjoyable or not is not an easy task as "fun" is considered to be a broad subjective topic. For instance, Designer Marc LeBlanc defines fun as a set of eight concepts (or categories) that are related to human psychology: fellowship, fantasy, discovery, narrative, expression, challenge and submission [11]. Other designers add up concepts such as immersion, beauty, competition, power and much more. Some researchers try to approximate fun by looking at the psychological concept of challenge specifically with "rhythm groups" inspired by Smith et Al. and "flow" inspired by Csikszentmihalyi. Both concepts can be defined as the momentum of a game (low challenge sections followed by high challenge sections). In short, a good PCG algorithm should maximize the fun of a level.

2.2 Procedural Content Generation

In this section, we define Procedural Content Generation (PCG), present some of its applications and define PCG evaluation.

2.2.1 *Definition*

Procedural Content Generation (PCG) is the automated creation of game content such as characters, weapons, maps, stories, levels, or full video games. PCG requires little to no human intervention and uses algorithms to make the game development process easier and quicker. Some PCG algorithms are simple and already implemented in the games with the help of pseudo-random numbers (example: "Elite") and object pooling (example: endless games) [3]. However, PCG can also provide more complex behavior with the help of machine learning and computer vision in which the machine "learns" the design principles of video games. Below is a list of some of the tasks/benefits PCG could offer:

1. Increasing replay value of an already existing game by generating new novel levels that follow the design rules of the targeted game.

2. Saving time and cost of production.

3. Minimizing the storage required by a game.

4. Increasing the variety of in-game items by variation of base sprite or 3d model (example: variation in weapons, NPCs, etc.).

5. Run-time generation of platforms, obstacles and environments in endless runner games. In this implementation, PCG becomes the very core mechanic of the game.
6. Generating blended games where two or more game styles are merged together to form a new unique game.
7. Generating progressive stories that adapt to the player's choices.
8. Transferring game style from one game to another different game.
9. Generating a complete game that belongs to a given style.
10. Extracting level design rules from successful games to aid game designers with good suggestions.
11. Creating adaptive games that can generate levels with the desired difficulty to adhere to the skills of any player for a better game experience.

2.2.2 *Evaluation*

In a set of research papers, a PCG is evaluated by its' ability to generate a variety of levels by using the concept of expressive range which is composed of two metrics: linearity and leniency [14]. However, the expressive range alone cannot give a clear idea of how well a PCG algorithm performs. Therefore, other researchers evaluate the performance of their PCG algorithm based on the quality of output levels by assigning scores to fun and playability. A PCG can also be evaluated in terms of speed and complexity.

2.3 Generative Adversarial Network

In this section, we introduce Generative Adversarial Networks and analyze some of it's variations.

2.3.1 *Definition*

Generative Adversarial Networks (GANs) is a machine learning technique that was suggested by Ian Goodfellow et al. in 2014 in an attempt to teach machines to generate new unseen data (ex: human faces never seen before). A GAN consist of two simultaneously trained models: the Generator and the Discriminator. The Generator is trained to generate data that resembles real data (ex: fake realistic faces) whereas the Discriminator is trained to discern the real data from the fake one (ex: this is a fake face, this is a real face). By training the two models at the same time, the generator and discriminator will work to improve each other's performance in a competitive manner. Figure 2.3 shows a GAN diagram [15].

Generator Network:

- Input: a random noise vector (z) as input which is a starting point to synthesize fake data.
- Output: x* denoting fake data (ex: fake images)
- Goal: Given Z, Generate fake data x*
- Iterative training: Update weight and biases to maximize the Discriminator's misclassification probability

Discriminator Network:

- Input: X which is either real data (x) provided by the dataset (ex: MNIST) or fake data (x*) output provided by the generator network
- Output: The probability of X being real
- Goal: Given X, find whether X is real data or fake data (classification problem)
- Iterative training: Update weight and biases to maximize the Discriminator's correct classification

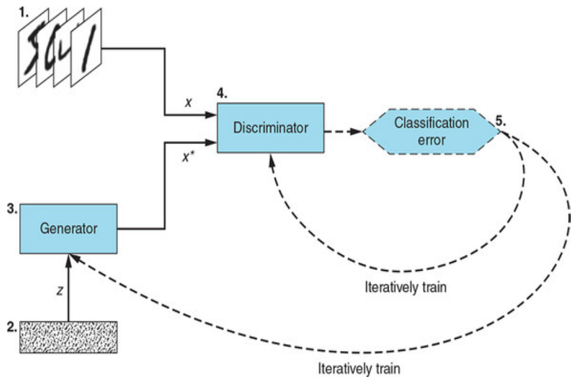
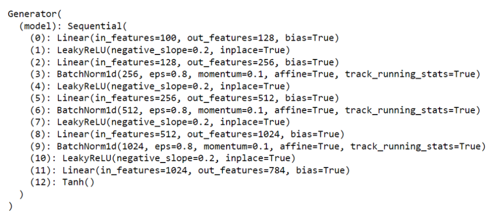


Figure 2.3: The two GAN networks, their inputs and outputs and their interactions [16]

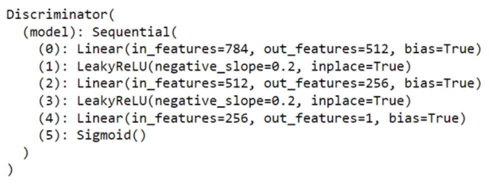
A GAN network is a zero-sum game (minimax), i.e GAN converge when the generator generates data that is indistinguishable from real data and the discriminator can at most randomly guess whether the data provided is real or fake. Figure 2.4 presents a variation of simple GAN architecture inspired by Ian Goodfellow et al to generate MNIST data. The generated numbers are shown in Figure 2.5 [16].

In some cases, the Gan's generator might learn one valid output that always fools the generator. Thus, the generator will only generate that one output or a small set of outputs without variation and this is due to the generator over-optimizing for a particular discriminator at each iteration. This phenomenon is referred to as mode collapse [16].



```
Generator(
  (model): Sequential(
    (0): Linear(in_features=100, out_features=128, bias=True)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Linear(in_features=128, out_features=256, bias=True)
    (3): BatchNorm1d(256, eps=0.8, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Linear(in_features=256, out_features=512, bias=True)
    (6): BatchNorm1d(512, eps=0.8, momentum=0.1, affine=True, track_running_stats=True)
    (7): LeakyReLU(negative_slope=0.2, inplace=True)
    (8): Linear(in_features=512, out_features=1024, bias=True)
    (9): BatchNorm1d(1024, eps=0.8, momentum=0.1, affine=True, track_running_stats=True)
    (10): LeakyReLU(negative_slope=0.2, inplace=True)
    (11): Linear(in_features=1024, out_features=784, bias=True)
    (12): Tanh()
  )
)
```

(a) Simple GAN Generator



```
Discriminator(
  (model): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Linear(in_features=512, out_features=256, bias=True)
    (3): LeakyReLU(negative_slope=0.2, inplace=True)
    (4): Linear(in_features=256, out_features=1, bias=True)
    (5): Sigmoid()
  )
)
```

(b) Simple GAN Discriminator

Figure 2.4: A simple GAN architecture applied for MNIST digits generation

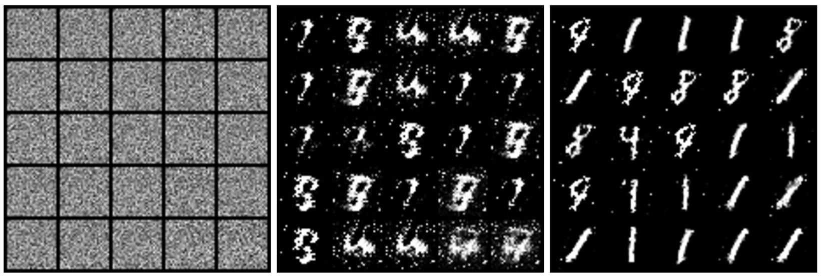
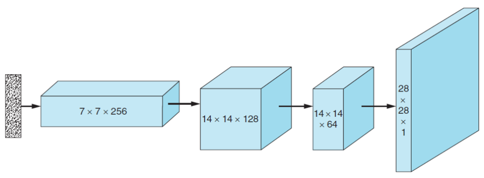


Figure 2.5: Mnist numbers generated by GAN architecture from Figure 2.4. Results from first, intermediate and last iteration in training respectively from left to right.

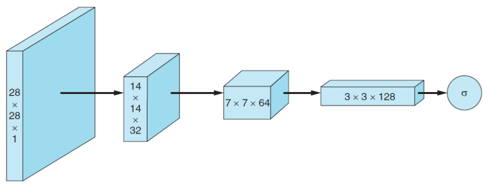
2.3.2 *Deep Convolution GAN (DCGAN)*

DCGAN (Deep Convolutional Generative Adversarial Networks) were presented by Alec et al. in 2016 as a more advanced architecture of GAN that uses Conv nets as the underlying technology. Throughout their research, they show that DCGAN results in more stable training for a certain range of datasets and it allows the training of higher resolution and deeper generative models. Conv nets have been acclaimed for their ability to find areas of correlation within an image hence being a good option when generating this kind of data [17] Figure 2.6 presents an overview of the DCGAN's generator and discriminator models. The generator's main goal remains the same: take a vector (Z) and generate an image (ex: 28 x 28 x 1 MNIST digit image). As shown in Figure 2.6a, the generator will start with a small width and height and a large depth (7 x 7 x 256). The hidden layers take care of minimizing the depth to reach the correct number of channels (1 for grayscale, 3 for color, etc.) whereas the width and height increase to reach the original size of the image (28 x 28). Each layer in the generator is coupled with batchnorm (to help stabilize the training process) and relu except for the last output which uses tanh. On the other hand, the discriminator takes as input either a real entry x or a fake entry x* (ex: Mnist image digit such as 28 x 28 x 1) and seeks to lower the height and width while increasing the depth. As displayed in Figure 2.6b, the final output is a scalar that gives that classifies the entry as either real or fake. As with the generator, each layer should be followed by batchnorm and leak-relu layers [16].

Figure 2.7 presents a variation of DCGAN architecture to generate MNIST data. The generated numbers are shown in Figure 2.8 and can be compared to those



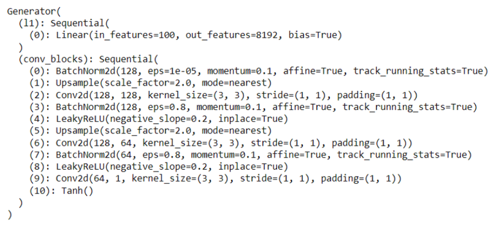
(a) DCGAN Generator



(b) DCGAN Discriminator

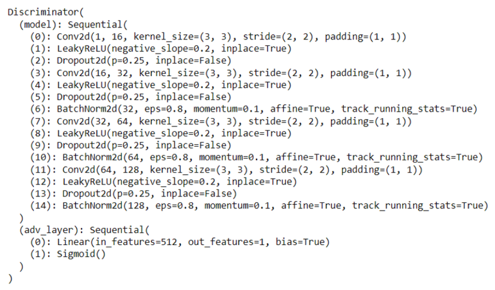
Figure 2.6: DCGAN Generator and Discriminator Networks [16]

generated by GAN in Figure 2.5.



```
Generator(
  (l1): Sequential(
    (0): Linear(in_features=100, out_features=8192, bias=True)
  )
  (conv_blocks): Sequential(
    (0): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (1): Upsample(scale_factor=2.0, mode=nearest)
    (2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): BatchNorm2d(128, eps=0.8, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Upsample(scale_factor=2.0, mode=nearest)
    (6): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): BatchNorm2d(64, eps=0.8, momentum=0.1, affine=True, track_running_stats=True)
    (8): LeakyReLU(negative_slope=0.2, inplace=True)
    (9): Conv2d(64, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (10): Tanh()
  )
)
```

(a) DCGAN Generator



```
Discriminator(
  (model): Sequential(
    (0): Conv2d(1, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Dropout2d(p=0.25, inplace=False)
    (3): Conv2d(16, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Dropout2d(p=0.25, inplace=False)
    (6): BatchNorm2d(32, eps=0.8, momentum=0.1, affine=True, track_running_stats=True)
    (7): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (8): LeakyReLU(negative_slope=0.2, inplace=True)
    (9): Dropout2d(p=0.25, inplace=False)
    (10): BatchNorm2d(64, eps=0.8, momentum=0.1, affine=True, track_running_stats=True)
    (11): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (12): LeakyReLU(negative_slope=0.2, inplace=True)
    (13): Dropout2d(p=0.25, inplace=False)
    (14): BatchNorm2d(128, eps=0.8, momentum=0.1, affine=True, track_running_stats=True)
  )
  (adv_layer): Sequential(
    (0): Linear(in_features=512, out_features=1, bias=True)
    (1): Sigmoid()
  )
)
```

(b) DCGAN Discriminator

Figure 2.7: DCGAN Generator and Discriminator Layers

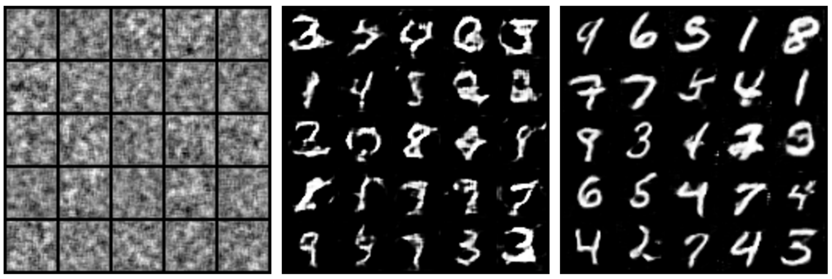
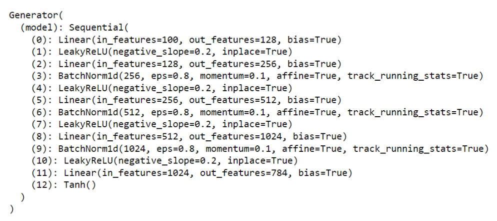


Figure 2.8: Mnist numbers generated by DCGAN architecutre from Figure 2.7. Results from first, intermediate and last iteration in training respectively from left to right.

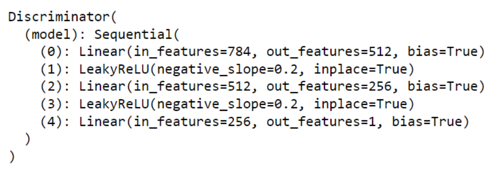
2.3.3 *Wasserstein GAN*

Wasserstein GAN (WGAN) was proposed by Martin et al. in 2017. WGAN gained its fame because it brought significant improvement to the loss function and tends to generate better results in a large number of experiments. In addition, this architecture provides a clear stopping criteria whereas it uses the earth mover's distance (EMD) as loss function instead of BCE. EMD also enables WGAN to bypass the mode collapse and vanishing gradient (generator cannot learn when discriminator is too strong) problems faced by simple GANs and DCGANs. This allows for better and higher-quality image generation. In WGAN the keyword "critic" refers to what was previously known as generator [18] . We test the WGAN architecture that is displayed in Figure 2.9. A sample of the generated results is presented in Figure 2.10. It is interesting to note that WGAN-GP shares the same implementation as WGAN but it adds a penalty on the norm of weights from the critic network. The penalty is added to ensure that the critic network satisfied the Lipschitz condition. Figure 2.11 displays a sample of generated digits which visually seems to be better and clearer than WGAN [16].



```
Generator(
  (model): Sequential(
    (0): Linear(in_features=100, out_features=128, bias=True)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Linear(in_features=128, out_features=256, bias=True)
    (3): BatchNorm1d(256, eps=0.8, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Linear(in_features=256, out_features=512, bias=True)
    (6): BatchNorm1d(512, eps=0.8, momentum=0.1, affine=True, track_running_stats=True)
    (7): LeakyReLU(negative_slope=0.2, inplace=True)
    (8): Linear(in_features=512, out_features=1024, bias=True)
    (9): BatchNorm1d(1024, eps=0.8, momentum=0.1, affine=True, track_running_stats=True)
    (10): LeakyReLU(negative_slope=0.2, inplace=True)
    (11): Linear(in_features=1024, out_features=784, bias=True)
    (12): Tanh()
  )
)
```

(a) WCGAN Generator



```
Discriminator(
  (model): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Linear(in_features=512, out_features=256, bias=True)
    (3): LeakyReLU(negative_slope=0.2, inplace=True)
    (4): Linear(in_features=256, out_features=1, bias=True)
  )
)
```

(b) WCGAN Discriminator

Figure 2.9: WCGAN Generator and Discriminator Networks

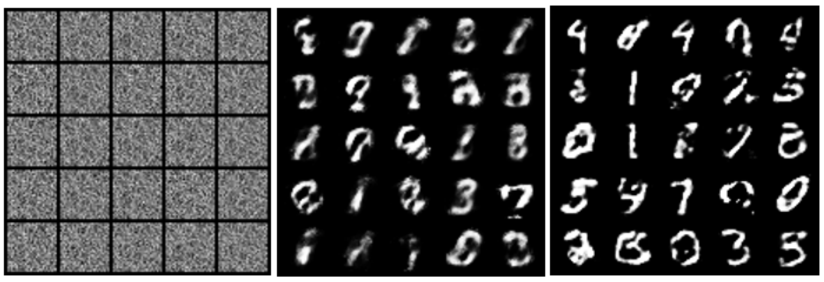


Figure 2.10: Mnist numbers generated by WGAN architecutre from Figure 2.9. Results from first, intermediate and last iteration in training respectively from left to right.

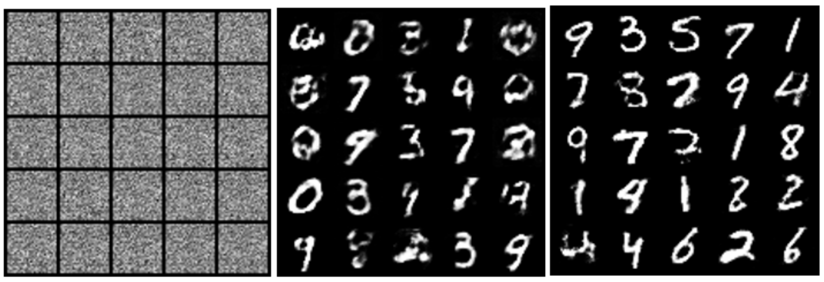


Figure 2.11: Mnist numbers generated by WGAN-GP. Results from first, intermediate and last iteration in training respectively from left to right.

2.3.4 *Fréchet Inception Distance*

Fréchet Inception Distance (FID) measures the distance between the Inception-v3 activation distributions for generated and real examples in a GAN network. It is used as means to assess the realism and variation of the generated images as well as their diversity. FID was inspired by its predecessor, the Inception Score which evaluates the distribution of generated images, whereas FID compares the distribution of the generated images to the distribution of real images that were used to train the generator model. Furthermore, the FID does not compare images pixel by pixel in a similar fashion to L2 norm but rather compares the mean and standard deviation of one of the ConvNet layers in the Inceptionv3 model. Hence, the metric is represented by the squared Wasserstein metric between two multidimensional Gaussian distributions [19]. The FID score formula can be written as follows:

$$FID_{(x,g)} = ||\mu_x - \mu_g||_2^2 + \mathrm{Tr}(\Sigma_x + \Sigma_g - 2(\Sigma_x\Sigma_g)^{1/2})$$

where (μ_x, μ_g) and (Σ_x, Σ_g) are the respective means and covariances of the samples from the true data distribution and the generator's distribution. Moreover, a lower FID score indicates that the generated output is very similar to the original input. Figure 2.12 shows an example of several images with different noise levels. The more noisy an image is, the lowest the FID score [20].

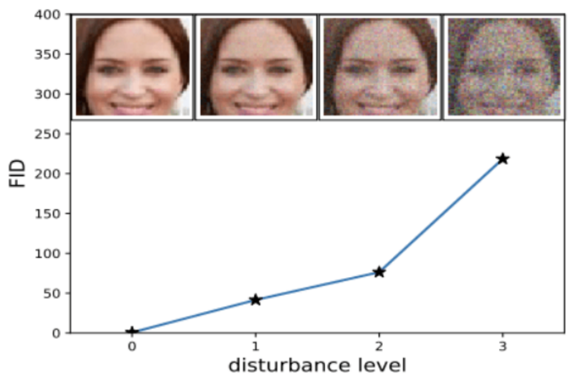


Figure 2.12: The FID score for images with different disturbance levels by Brownlee

2.4 OpenCV

OpenCV is a free cross-platform library for real-time computer vision functionalities originally developed by Intel. This library comes in very handy when performing machine learning tasks with visual inputs such as images and videos as it helps the machine gain an understanding of these types of data [21].

2.4.1 *Multi Template Matching*

Template Matching is the process of finding the location of a template image in a larger image. OpenCV comes with the function cv.matchTemplate() that searches for the template image by sliding it over the input image in a similar fashion to 2d convolution and then compares the template with each taken patch of the original image as shown in Figure 2.13. The function returns the location of the best match. On the other hand, Multi Template Matching is the process of finding the same template image multiple times in the original image as depicted in Figure 2.14 [21].

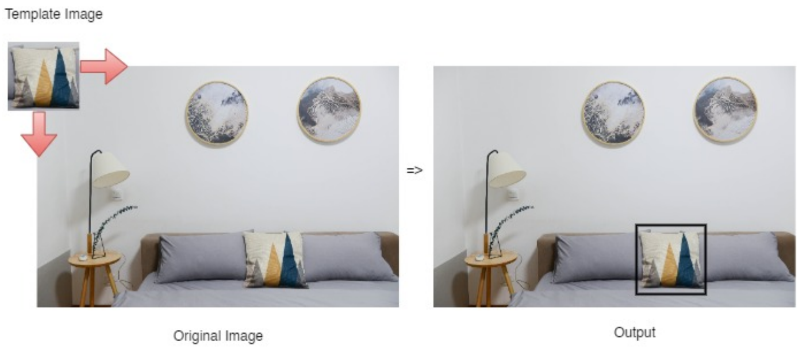


Figure 2.13: Demonstrating an Example of OpenCV Template Matching

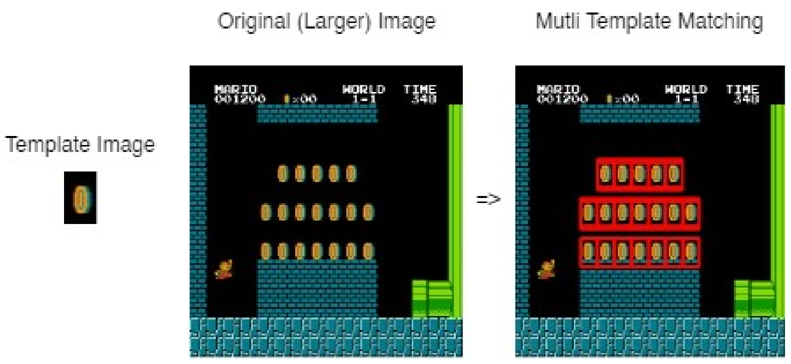


Figure 2.14: Demonstrating an Example of Multi Template Matching by OpenCV

Chapter 3
Literature Review

In this chapter, we look at the literature and discuss the research papers related to PCG algorithms that attempt to understand design knowledge to create novel game levels. It is interesting to note that most research carried out in this field uses the Super Mario Bros game (boss and underwater levels excluded) as the main source of data for their algorithms. The reason for the popularity of this particular game among researchers is the linearity, one-direction & sprite-tiling nature of the game. It also constitutes the first step toward a more general platformer PCG algorithm. Moreover, we also divide the literature review into three main sections of interest (1) Dataset, (2) PCG Methodologies and (3) PCG Evaluation.

3.1 Dataset

There is a lack of publicly available high-quality clean game datasets for PCG given that it is a developing field of research as noted by Summerville et al [22]. They discuss three public datasets:

- VGLC: contains level data and lacks gampeplay information (published paper, most popular)
- Opengameart: contains object models (online website, rarely used)
- Squidi.net: contains gameplay mechanic (online website, rarely used)

Before VGLC and PCGML, to our knowledge, most authors would recreate a data-extraction algorithm from either level maps available online with computer graphic tools or directly from game code.

The VGLC (video game level corpus) is a game dataset made up of 12 games with a total of 428 levels, created by Summerville et al. The corpus is ready for use for machine consumption in several ML and AI applications for video games such as generation, design knowledge and style transfer. For the Mario Game, the authors convert a long pictorial map of each level into a parsable text composed of a set of characters with each tile mapping to one predefined character. The source of the maps is not clear but might be derived from either the game's binary or manually reconstructed by online fans of the franchise. Figure 3.1 shows the mapping of Mario's sprites/tiles into specific characters whereas Figure 3.2 shows the picture and text format of a section of SMB world 1 level 1 [23]. It is interesting to note that this method also gives the advantage of lowering the data size as it maps every collection of pixels into one single character thus making the learning process easier and faster. However, we should also point out that all enemy types in Mario are grouped under the generic "E" character thus losing some information about the specific type, difficulty and behavior of the enemy. The same applies to some other types of items defined by VGLV in the mapping.

Unlike other previously seen research papers which make use of the VGLC dataset or extract their own data directly from SMB's game code or online art, Matthew et al. make use of video footage of the Super Mario game, also referred to as gameplay video as a main source of data. The authors argue that using gameplay videos has multiple benefits [24]:

- Interchangeable formats: using the same algorithm for any kind of video format as one can easily convert from one video format to the other.
- Includes player experience: gameplay videos include the reaction of the player, the path they take and their experiences. No two gameplays can be considered the same.
- Publicly available & Large Corpora: with the increasing popularity of the "let's play" and "long plays" over YouTube and social media, more people are indulging in the gaming community and sharing their gameplays online.

In an attempt to game engine learning, Matthew et al. also use gameplay videos as the main source of data. The authors develop a tool that relies on OpenCV to parse each frame into a list of sprites with their numbers and spatial information. A greedy matching algorithm is run on the frame-sprites information to get the amount of shift between the adjacent frame and its closest neighbor. Finally, they extract facts such as animation, spatial, relationshipX, velocityX and so on for each sprite. These facts are then used by the Author to learn rules (engine) that can search frames and build playable levels [25]. Hence, one can argue that video gameplay as a data source can provide additional information and benefits that a still map or VGLC dataset cannot.

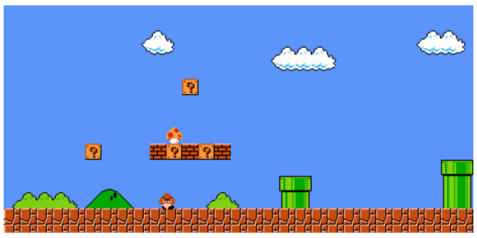
As discussed by Summerville et al., creating new datasets is necessary for refining and expanding the field [22]. To our knowledge, usage of video gameplay hasn't been extensively explored and there doesn't exist any clean unified video dataset. As such, we aim to provide such dataset and discuss its details in chapter 4.

3.2 PCG Methodologies

In this thesis, we divide the research carried out in the field of PCG into three broad categories: Evolutionary Search-Based PCG, Probabilistic PCG and PCGML.

```
{"tiles" : {"X" : ["solid","ground"],
            "S" : ["solid","breakable"],
            "-" : ["passable","empty"],
            "?" : ["solid","question block", "full question block"],
            "Q" : ["solid","question block", "empty question block"],
            "E" : ["enemy","damaging","hazard","moving"],
            "<" : ["solid","top-left pipe","pipe"],
            ">" : ["solid","top-right pipe","pipe"],
            "[" : ["solid","left pipe","pipe"],
            "]" : ["solid","right pipe","pipe"],
            "o" : ["coin","collectable","passable"],
            "B" : ["solid","bullet bill","hazard","enemy"],
            "b" : ["solid","bullet bill"]  } }
```

Figure 3.1: VGLC mapping of tiles into characters for the SMB game


(a) Image format

```
------------------------------
------------------------------
------------------------------
------------------------------
------------------------------
------------Q-----------------
------------------------------
------------------------------
------------------------------
------Q---S?SQS---------------
----------------------------<>
------------------<>--------[]
-----------E------[]--------[]
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

(b) Parsable text format

Figure 3.2: A section of SMB level in picture format and its text parsable format provided by VGLC

3.2.1 *Evolutionary Search-Based PCG*

In 2012, Steve Dahlskog et al. discuss the relation between PCG and design patterns in games. They explain that games are a collection of well-tested and playable designed structures that, when combined, give meaning and context to the game. Thus, a PCG algorithm must generate structures and levels that make sense to the player and not just create a random combination of game pieces. In addition, a level should lead the player to neither frustration (too difficult) nor boredom (too easy). Given that, Dahlskog et al. argue that already existing games such as Super Mario Bros (SMB), which have been well planned and extensively tested, could be used to build a PCG algorithm that can generate meaningful levels. Dahlskog et al. start their research by manually studying and extracting the patterns (entities) from the levels of Super Mario Bros, excluding boss levels and underwater settings due to the difference in gameplay. Using gameplay and meaning, they find twenty-three patterns/structures in SMB that are re-used throughout the game. Figure 3.3 displays some of these structures. The authors discuss that a PCG algorithm can feed on these already-meaningful patterns, slightly change them (increase/decrease gap length, add an enemy, etc.) then randomly combine them while taking into consideration the difficulty parameter of each pattern. Thus, a PCG can still form a meaningful new level. In this sense, one can argue that their approach to the problem might generate a limited output of new levels as a variation only depend on parameters. [3]

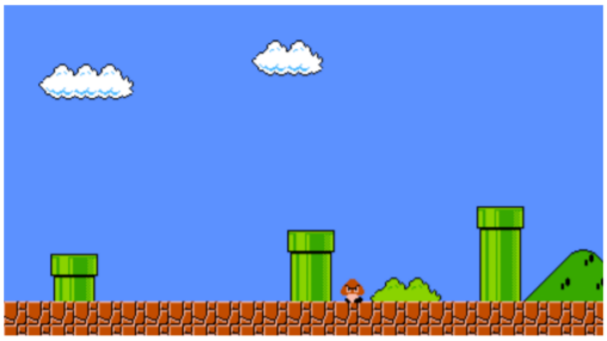


Figure 3.3: Empty Valley followed by Enemy Valley patterns by Steve Dahlskog et al.

In 2013, Steve Dahlskog et al. carried on their work by building a PCG that uses previously discovered structures (later referred to as meso-patterns) as objective in an evolutionary algorithm. Their goal was to produce levels by recombining vertical slices (later referred to as micro-patterns) from the SMB game all the while maximizing the number of known structures. Figure 3.4 depicts some of the vertical slices that were extracted by the authors from the SMB game with each slice consisting of 1 tile in width and 14 tiles in height. Moreover, the authors invited test-players to evaluate a level generated by three different fitness functions (reverse, actual and pattern-based). The authors claim that the levels generated by the fitness function that rewarded partial and full patterns were more fun to play and appeared more similar to the actual levels and can generate a larger set of new levels while retaining the design of the original game. The authors report that their algorithm cannot support the concept of beats (rhythm) very well and that fine-tuning fitness function demands attention when adding new patterns.The authors also report that the generator fails to create a natural flow in the level. [2]

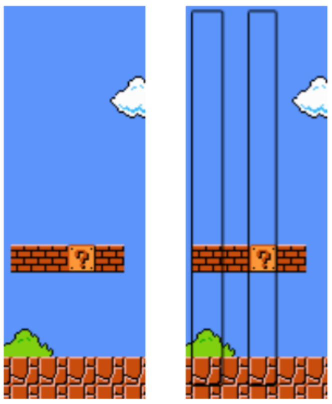


Figure 3.4: 2-path pattern previously defined by Steve Dahlskog et al in 2012 and the 2 vertical slices extracted from this section used by Dahlskog et al in 2013.

Steve Dahlskog et al. published yet another paper in 2014 in which they propose the use of “macro-patterns” as objective for the evolutionary algorithm. They define macro-patterns as the sequence of meso-patterns that will provide a full-level overview in an attempt to output a more meaningful level. This time, the fitness function does not only reward the number of meso patterns but it also looks for their order and sequencing (macro pattern). They compare three algorithms FFmeso (reward meso-patterns equally), FFmesoB(weighted reward of meso-patterns) and FFmacro (FFmesoB with extra reward if the order of patterns is similar to the original game). To compare the three generators, the authors use Smith's and whitehead's metrics Linearity and Leniency. Thus, the authors claim that FFmacro and FFmesoB perform better than FFmeso in terms of variety and level difficulty but FFmacro slightly performs better than FFmesoB. However, they report that FFmacro is tenfold slower in generation and was deemed unusable in real-time scenarios. [26]

Despite generating playable levels in all three attempts, one can argue that the approach presented is very game-specific since the patterns are carefully picked and manually studied. Moreover, the methodology in question introduces other variables of uncertainty such as "level of abstraction" that defines how large a meso and macro pattern are and the fitness function which requires to be varied and fine-tuned. Table 3.1 presents a summary of the papers, their advantages and disadvantages.

Paper	Methodology	Advantages	Disadvantages
Patterns and Procedural Content Generation (2012)	Find all repeated patterns (section of game) in SMB manually.	Building block for next set of papers.	Requires understanding & analysis of the game and generator has limited output space.
Patterns as Objectives for Level Generation (2013)	Use vertical slices of the map as alphabet and a fitness function that rewards patterns.	Generates set of new levels that retain design of original game.	Doesn't support rhythm groups, fitness function needs fine-tuning when adding new patterns, is game-specific.
Multi-level Level Generator (2014)	Macro-patterns (sequence of meso-patterns) as objective.	Performs better in variety & level difficulty, also fixes flow problem.	Slower algorithm and game-specific.

Table 3.1: Evolutionary Search-Based PCG techniques Summary

3.2.2 ***Probabilistic PCG***

In 2014, Julian et al. propose the use of n-grams to generate levels that are represented by slices. In their experiments, the authors vary the values of n (1,2,3) and the number and types of levels in the corpus (one level, similar levels, different levels). They claim that the method is fast and reliable and can generate a variety of levels that still retain the style of the original SMB levels, especially when using tri-grams. The authors base this claim on the Pearson coloration for the metrics of the expressive range between the generated levels and the original levels. However, when using multiple levels in the corpus could lead to a surprising shift in the style. The authors report that the method is only limited to linear 2d games. Moreover, a game-specific pruning process is required to decrease boring sections and it relies on the code of the game as with the previous papers.[13]

Snodgrass et al. suggest using Markov Chains as a way of learning level structures from human-authored maps so that they can generate new level maps. Markov chains model probabilistic transitions between different states. The authors claim that the method can generate novel levels. However, a considerable number of generated levels are unplayable. Since most of the tiles are empty, the transition would happen from or to an empty tile which would cause a problem. It also misses the height of the tiles in platformer games (most tiles above are empty such as sky, most tiles below are interactive such as block, ground, etc.). They attempt to fix these problems by using look-ahead, backtracking and splitting the level into multiple horizontal slices and applying a Markov chain on each. [27]

Summerville et al. build upon the work of Snodgrass et al. and Julian et al. and propose the use of Monte Carlo Tree Search (MCTS) to guide the Markov chain so it generates more playable levels. The role of MCTS is to use random sampling during search to find areas that are playable by balancing the need to exploit (playable) and explore (near unplaybale i.e hard to play). The authors use a 2-slices history to generate the chain with a transition of up to 8 slices. [6]

Matthew et al. use a novel approach in which they extract data from several SMB video gameplay available online. The authors perform five major tasks

1. Defining sections by measuring differences between frames.
2. Categorising sections into high and low interaction areas depending on the number of frames.
3. Clustering data into similar themes using K Means++ and euclidean distance on vectors of sprites count.
4. Learning generative probabilistic models of themes as inspired by the work of Kaleogerakis et al. in the creation of 3D models.
5. Generating level sections through a recursive walk of the model.

It is interesting to note that the generator only creates sections of levels rather than full levels all the while discarding low interaction section which should be included to provide rhythm groups. Although the work of Matthew et al. provides room to edit playability and similarity, they do not provide a means of controlling level difficulty. [24]

Table 3.2 presents a summary of the papers, their advantages and disadvantages.

Paper	Methodology	Advantages	Disadvantages
Linear levels through n-grams (2014)	Use N-grams to generate levels with slices as vocabulary.	Fast, reliable & generates variety of levels that retain style of original game using tri-grams.	Learning from multiple levels cause undesired shift in style, limited to 2D linear games, requires game-specific pruning process.
Monte Carlo Tree Search to Guide Platformer Level Generation (2015)	Balances need to explore (hard levels) & exploit (playable levels).	Generates set of new levels that retain design of original game & increases space of playable levels.	Sacrifices specificity for better speed, is game-specific.
Toward Game Level Generation from Gameplay Videos (2016)	Learning generative probabilistic models of themes.	Provides controllable parameters (playability and similarity).	Ignores low interaction sections, no control over level difficulty, generates sections not full levels.

Table 3.2: Probabilistic PCG techniques Summary

3.2.3 ***PCGML***

PCGML (Procedural Content Generation via Machine Learning) is defined by Summerville et al. as "the generation of game content by models that have been trained on existing game content". [22]

In 2016, Summerville et al. attempt to solve the problem of level generation with sequence learning that has been commonly used for translation, speech understanding and video captioning. Hence, the authors propose LSTM Recurrent Neural Networks as a method to train a level PCG. The authors opt for the use of each tile as a character in a sequence with each tile being assigned a type such as solid, enemy, etc. It is interesting to note that due to applying a level of abstraction to some types, one can argue that crucial information is lost. The authors train eight networks with the combinations of:

1. Snaking: Defines how the tiles are read in sequence as either (1) bottom to top or (2) snaking as in alternating between bottom to top and top to bottom. For the latter, the authors claim better locality as important information like the "pipe" structure is separated by fewer blocks as seen in Figure 3.5.

2. Path Information: defines whether to embed the player's path to solve the level or not. The path information is provided by a tile A* algorithm and used as means to increase the number of playable levels by the LSTM network.

3. Column Depth: defines whether to embed a sense of progression in the level or not. This is done by including a special incremental character every 5 columns to denote progress.

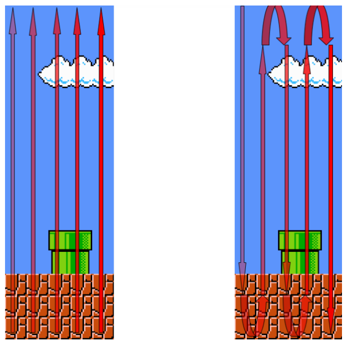
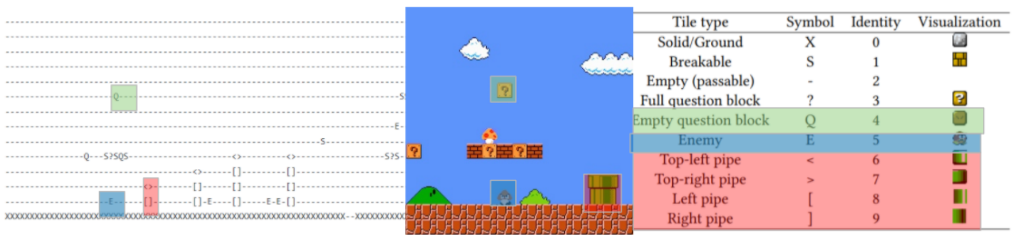


Figure 3.5: Pipe separated by fewer tiles when snaking by Summerville et al

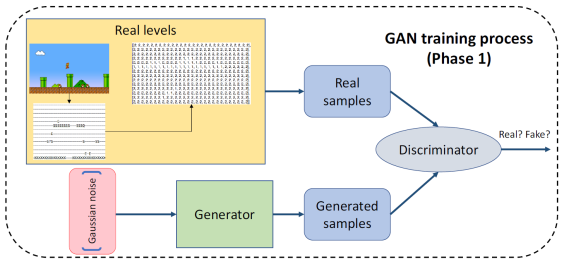
Summerville et al. claim that the networks with Path information perform two times better than those without. Their experimentation results also confirm that the network including all three components (snaking, path information and column depth) performs the best out of all 8 networks. It is worth mentioning that the algorithm used works best for left-to-right games such as SMB where there is a linear mapping between space and time. [7]

In 2018, Vanessa Volz et al. trained a DCGAN to generate super Mario level sections using a level from the VGLC dataset. As shown in Figure 3.6, the authors assign an identity number (table on right) for each of the symbols provided in the VGLC dataset (representation on left). Each integer is then converted into a one-hot encoding vector thus making the data ready to be fed to the DGAN network as presented in Figure 3.7a. The authors then apply CMA-ES (evolving vectors of real numbers) and LVE to the latent vector to search the generator's space for the fittest playable levels. In each evolutionary iteration, the latent vector is updated to abide by some parameters such as the number of tiles, enemies and playability as depicted in Figure 3.7b [8].

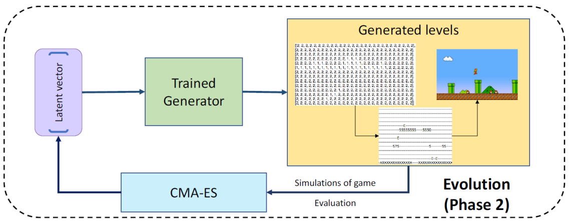


Tile type	Symbol	Identity	Visualization
Solid/Ground	X	0	▨
Breakable	S	1	▨
Empty (passable)	-	2	
Full question block	?	3	▨
Empty question block	Q	4	▨
Enemy	E	5	▨
Top-left pipe	<	6	▨
Top-right pipe	>	7	▨
Left pipe	[	8	▨
Right pipe	]	9	▨

Figure 3.6: Level Representation used by Volz et al in 2018



(a) Phase 1: Gan Training Process



(b) Phase 2: Parsable Text Format

Figure 3.7: Volz et al. Methodology using GAN followed by Latent Vector evolution

Table 3.3 presents a summary of the papers, their advantages and disadvantages.

Paper	Methodology	Advantages	Disadvantages
Platformer Level Generation Via LSTMs (2016)	Use of LSTM (sequence learning in translation and speech recognition).	Generates promising results, and incorporates playability in the learning process.	Only works on left to right games such as SMB (sequence).
Evolving Mario Levels in the Latent Space of Deep Convolutional Generative Adversarial Network (2018)	Use WGAN to generate levels and CMA-ES (latent space evolution) to evolve latent vector to search for playable level (survival of the fittest).	Generates promising results, uses VGLC dataset, isn't game specific and searches the space for playable levels.	Trains on specific set of levels, limited control for designer, doesn't provide comparison to style of original game.

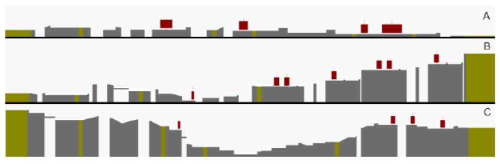
Table 3.3: PCGML techniques Summary

3.3 PCG Evaluation

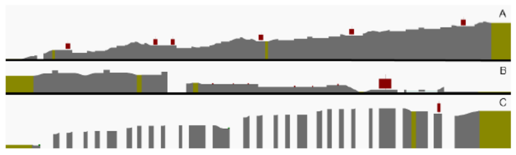
In this section, we discuss the different techniques in the literature that are used to evaluate and compare PCG algorithms. To have a good PCG, it should generate a variety of levels that are similar to the original game yet different from one another. Moreover, the generator should generate levels that are playable and fun [22]. As noted by Sumerville et al. there is no widely used standardized evaluation benchmark.

The most straightforward way to evaluate a PCG is to sample participants to play the generated levels to collect surveys. Dahlskog et al. survey players with three scores (1) boring-fun level (2) similar-not similar to original SMB game and (3) easy-hard to play. One can argue that the concept of fun is subjective and that sampling a large enough population could help us approximate the success of the generated levels [2].

In 2010, Gillian et Al. claim that a useful PCG can generate a collection of levels that differ from one another. For that purpose, they provide a rigorous method to capture the range of content referred to as expressive range of a PCG given two metrics: linearity and leniency. The first metric is found by using linear regression to fit a line to the center point of all geometry of the level then calculating the normalized mean absolute error with an expected value between 0 (highly linear) and 1 (highly non-linear) as depicted in Figure 3.8a. The other metric is calculated by adding a positive number for any gaps and enemies and a negative number for safe jumps. Thus, a higher score indicates less leniency as depicted in Figure 3.8b. [14]



(a) Levels ranging from highly linear (A) to highly non-linear (C)



(b) Levels ranging from highly Lenient (A) to highly non-lenient (C)

Figure 3.8: Linearity and Leniency as explained by Gillian et al.

Some early research papers used expressive range as a technique to compare two PCG algorithms. As example, figure 3.9 shows the expressive range of three generators provided by Dahlskog et al [26].

However, it seems less common in recent publications. We argue that expressive range alone is not sufficient to evaluate a PCG since it does not provide any direct insights about playability and level fun. Though, we also argue that it is a useful tool that should not be discarded and should be used with other metrics to evaluate PCG algorithms as it provides the following benefits:

- Understanding how a level generator behaves.
- Capturing PCG bias toward easy vs hard levels or linear vs non-linear levels.
- Comparing two generators in terms of output and similarity [26].
- Comparing the similarity between original games and PCG generated games as a means of evaluating the new PCG [13].

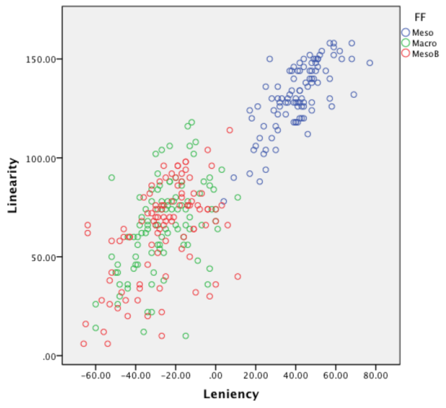


Figure 3.9: Expressive range of three generators (FFMeso, FFMesoB and FFMacro) by Dahlskog et al. in 2012.

To approximate the style score, Matthew et al. measure the distance from a generated section to the closest original level section. In other words, they calculate the edit distance of a sprite for a generated level and its closest originator. [24]

In their evaluation process, Matthew et al. consider a section of level to be playable if it meets the following conditions:

1. There is a platform on the left of the level section that the player can jump to from a previous section.
2. There is a platform on the right of the level section that the player can jump from to the next section.
3. There is a path between these two sprites.

An A* agent plays through the section to test the third condition. [24]

Summerville et al. propose the use of a hand-tweaked score metric in the selection process of the Monte Carlo Tree Search [6]. The authors define the score as $score = S + g + e + r$ [6] where.

- S is 0 if the level is solvable, a large negative number otherwise. A level is solvable if the A* agent can complete the level.
- g as function for desired number of gaps
- e as function of desired number of enemies
- r as function for desired number of rewards

In another paper, Summerville et al. attempt to use a collection of style metrics in combination with leniency and linearity to compare the generated level to the original levels. These metrics are detailed below.

- The level completion percentage of the A* agent.

- The percentage of the level that is empty.
- The percentage of empty level that is reachable by the player referred to as negative space of level.
- The percentage of the level that is filled with interesting sprites excluding solids and empty blocks.
- The total number of jumps in the level.
- The number of meaningful jumps in the level on occurrence of enemies or gaps.

The authors calculate the mean value of each of the metrics over a large set of generated levels and then compare them to the standard deviation of these values derived from the original levels. [7]

It is hard to quantify difficulty in a game given that it is quite subjective yet it can be approximated. In an attempt to measure difficulty, Canossa et al. define some interesting metrics such as Lenience differentiation (ratio of hazards to enemies) and threat Level (density and frequency of clusters of enemies). [28]

Chapter 4
Methodology

In this thesis we focus on the platformer games and our goal is to learn level design knowledge with GANs so that we can generate novel levels. We also focus on functional game content which is defined by Summerville et al. as "Artifacts, that if they were changed, could alter the in-game effects of a sequence of player actions". Hence, we are only interested in generating elements that are in direct relation to the gameplay such as platforms, enemies and so on rather than sound effects, GUI and other decorative sprites. The methodology is divided into three sections: Dataset, PCG methodology and PCG evaluation.

4.1 Dataset

As previously discussed by Summerville et al, there is a lack of clean datasets in the field of PCGML and any new datasets are welcome [22]. Given that VGLC does not provide video data, we aim to come up with a publicly available high quality dataset of Super Mario Bros gameplay videos that can be used for several PCGML and non-PCGML related tasks.

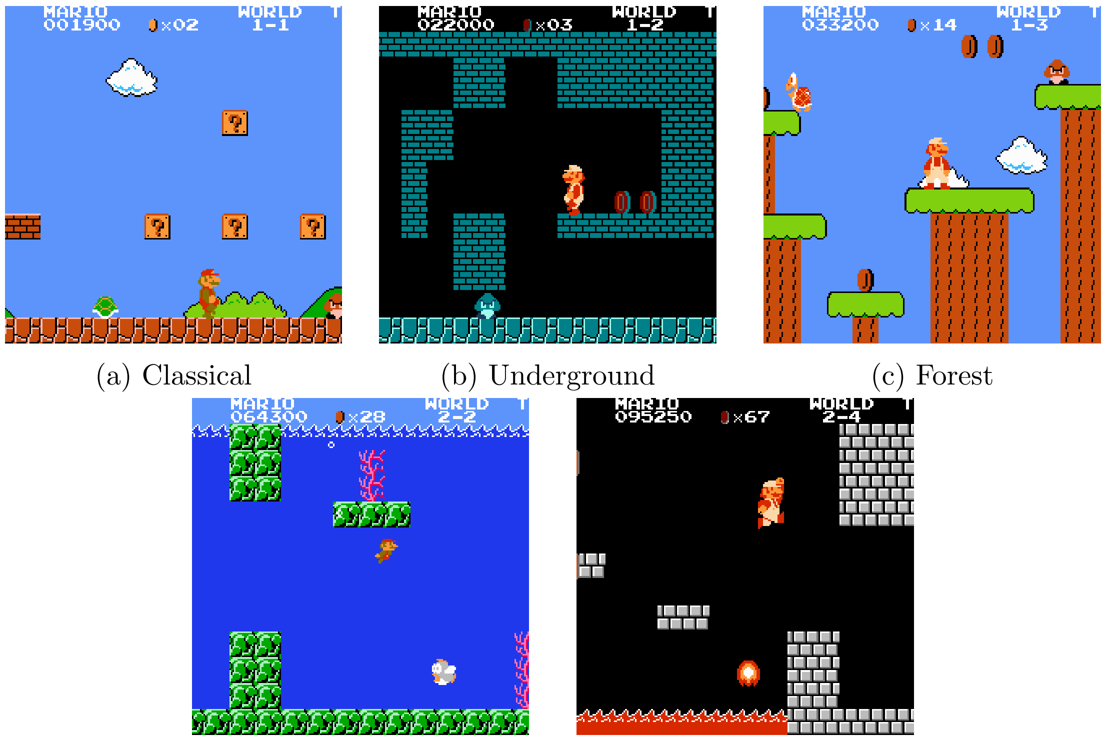
In addition to the previously mentioned benefits of using video gameplay data by Matthew et al. in chapter 3, the new dataset can also increase the number of data considerably by capturing every possible enemy state (example: when enemy moves from left to right). This could potentially make up for the "small" dataset issue faced by researchers in the field.

The dataset we present includes two games: Super Mario Bros and Super Mario Bros Lost Levels. It contains a total of 64 videos as both games include 32 high-quality videos with clean sound data in the AVI format with each featuring Mario going through one full level. All of the videos are free from death to avoid the black screen that could hinder machine learning tasks or impose additional computer vision requirements.

The dataset includes a class Video that extracts frames from this dataset and offers some other functionalities such as drawing a frame, cropping frames, transforming to grayscale, dropping a range of frames and saving the frames to a specific output folder. This Video class can be later extended with more functionalities such as grouping frames into interesting and non-interesting sections or extracting gameplay data, linearity of a level and more. It also includes a test driver code that is used to extract frames and prepare them for our PCG algorithm.

In some cases, we group the levels under one of 5 themes: classic theme, forest theme, underground theme, underwater theme and boss theme as these themes have considerable visual or functional differences. An example of each theme is displayed in Figure 4.1. For this thesis, we focus on the classic theme which we define as the following set of levels:

1. w1l1, w2l1, w4l1, w5l1, w5l2 and w8l1 and w8l2 from Super Mario Bros game.
2. w1l1, w3l2, w4l2, w5l1, w6l1 and w8l1 from Super Mario Bros Lost Level game.


MARIO 001900 ×02 WORLD 1-1
MARIO 022000 ×03 WORLD 1-2
MARIO 033200 ×14 WORLD 1-3
MARIO 064300 ×28 WORLD 2-2
MARIO 095250 ×67 WORLD 2-4
(a) Classical (b) Underground (c) Forest

(d) Underwater (e) Boss

Figure 4.1: Categorization of SMB levels into themes

4.2 PCG Methodology

We aim to train a GAN-based PCG using GAN, DCGAN and WGAN architectures. The first thing that comes to mind when generating mario frames with any GAN architecture is to replicate the MNIST problem detailed in chapter 2. However, this technique faces some issues as will be shown in chapter 5 and therefore, we transition into a different way of representing the data before feeding it to the GAN network. In this section, we explore both techniques of data representation, discuss the methodology of learning and present their advantages and disadvantages.

4.2.1 *Extracting Frames*

We create a video object for each level and generate frames. Given that sometimes a frame can be repeated due to a player spending time in the same level area, there might be repetition in the data and this might lead to overfitting in our GAN model. Therefore, we incorporate the ability to skip frames. The original frame size is (224, 256, 3) but we crop the frames to a square format of (208, 208, 3) noting that the last number denotes the number of channels. We save each collection of frames in its own folder labeled with the name of the level as shown in Figure 4.2. We also register some statistics such as the number of frames and the number of dropped frames.

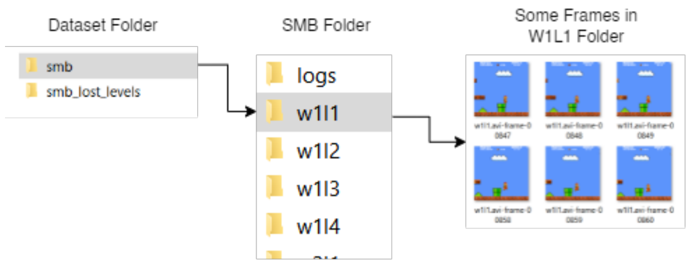


Figure 4.2: Storage Organization of the frames data

4.2.2 *Learning from Frames*

The first thing that comes to mind when training GANs is to provide the data in the form of images in a similar format to the MNIST dataset discussed in chapter 2. In this scenario, GAN tries to learn from frames and generate new unseen frames that can then be used to generate a level. We aim to test the robustness of this method by supplying it with one level of SMB and then creating a separate dataset with several levels of SMB belonging that fall under the classic theme. We resize all frames to 64 x 64 px to make the learning process faster and more stable. Then we normalize the images to suit the GAN architecture. After that, we feed the transformed images to GAN, DCGan and Wgan networks and vary the hyper-parameters (learning rate, batch size and latent vector dimension) to explore the outcome. We find the best combination of hyper-parameters by picking the model that has the smallest total loss, calculated as the average between the generator's loss and discriminator's loss. To evaluate the output of the final models, we get the FID score of each. As we will discuss in chapter 5, most of models generate low quality output images making it hard to run other openCV operations to complete the task of level generation or PCG evaluation.

4.2.3 *Learning From Language*

To get better results, we change the data representation. In what follows, we explain our approach and the different methods to extract data from frames.

Instead of passing frames with 64 x 64 px as input, we opt for Volz et al's technique in which they first transform the image into a smaller, more compressed representation made up of integers [8]. As previously discussed in chapter 3, VGLC provides a language that the authors then transform into integers as presented in Figure 3.6. In VGLC, the authors attempt to generalize game language. However, in some cases, a designer could want the language to be more specific without losing important information such as the enemy type. Therefore, we attempt to provide a technique in which the designer can adjust the level of abstraction for the game language to fit different needs. An example of this language for SMB game is given in Figure 4.3. The language is of the form 'General Name of Item':([list of sprite path that maps to the Item],[Item to which we map as image path]). For example, all three animated coin sprites from the classical theme as well as all three animated coin sprites from the underground theme map to the same "coin" item and will eventually be replaced by the coin.png image in the generation process. A general overview of the grouping scheme for the sample presented in Figure 4.4 can be seen in Figure 4.5. It is interesting to note that several blocks can map to one output item because it can bring two benefits:

- Allow the user to map different types of sprites to one item. As example, a designer could make the goomba, koopa troopa and bullet bill enemies all map to the enemy object (many to one mapping) or map each one of them it's own object (one to one mapping) thus giving the designer control over the level of abstraction of the language.
- Incorporating the different animated sprites or variations of the same sprite and mapping them to one same object. As example, a designer could choose to add multiple coin colors and consider them as one object labeled "coin". This also helps in solving the issue of detecting animated objects in video which isn't previously seen in map data with VGLC.

```
sprites = {
    'ground':(['ground_classical.png', 'ground_underground.png'], ['ground.png']),
    'block':(['block_classical.png', 'block_underground.png'], ['block.png']),
    'coin':(['coin_classical_animation_1.png','coin_classical_animation_2.png','coin_classical_animation_3.png',
             'coin_underground_animation_1.png', 'coin_underground_animation_2.png', 'coin_underground_animation_3.png'],
            ['coin.png']),
    'goomba':(['goomba_classical.png','goomba_underground.png'], ['goomba.png'])
}
```

Figure 4.3: A sample sprite dictionary that covers the "coin", "block", "ground" and "goomba" objects in both classical theme and underground theme.

Figure 4.5 summarizes our goal of transforming frames into a custom language of integers. The last column of Figure 4.5 denotes the rebuild of the frame using different variations of sprites.

Given a large map of the level, it is easy to run a 16x16 window to extract the block items in SMB. However, it is less intuitive with frames as some of them might start with half or quarter a block which is a problem that doesn't exist in the data extraction from full map. To better illustrate the issue, we take a closer look at the

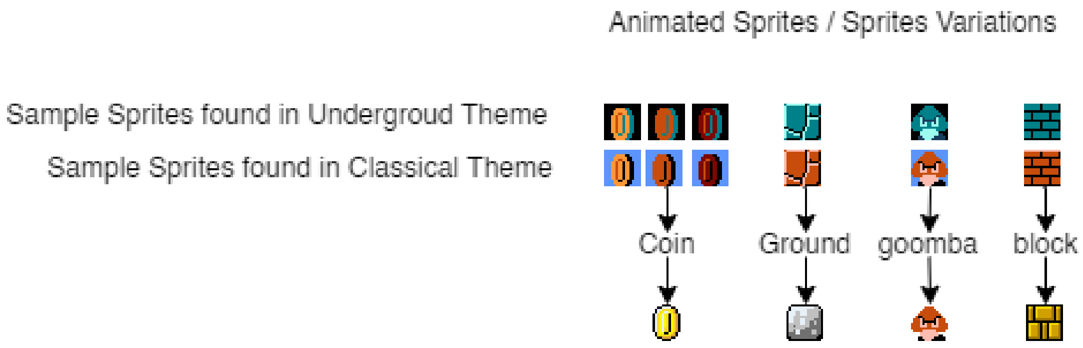
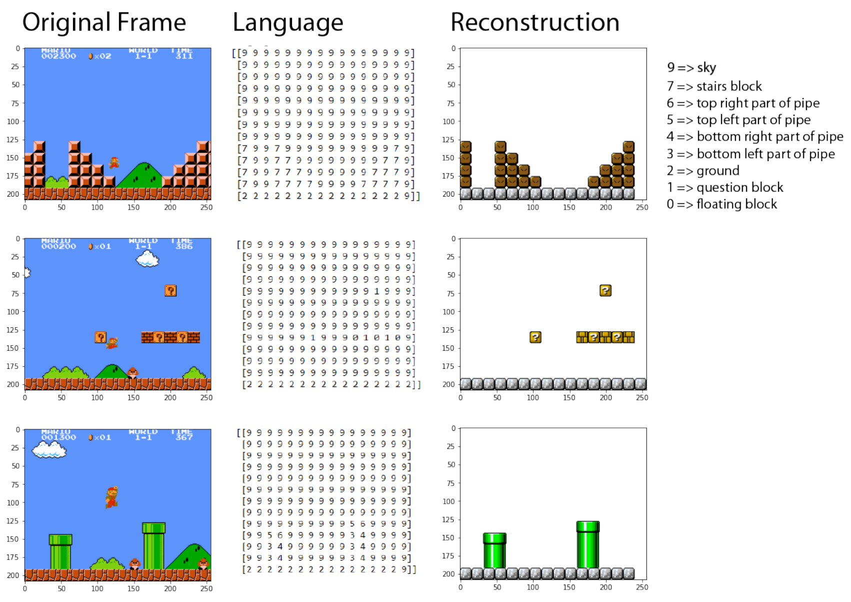

Animated Sprites / Sprites Variations
Sample Sprites found in Undergroud Theme
Sample Sprites found in Classical Theme
Coin
Ground
goomba
block

Figure 4.4: A diagram to visually showcase the level of abstraction used by the language displayed in Figure 4.3

question block displayed in Figure 4.6. The red and green boxes are 16px by 16px each. As we can see, the question mark block is shifted as part of it extends to the green box, making the mapping process used in the still map technique hard to apply on frames.

To solve the issue, we propose three methods to extract sprites from the frame as detailed below.

- PX by PX: an aggressive brute force technique that searches the frames for items by comparing each sprite's pixels (defined in the dictionary under the name of the item) of each item to the window of 16px x 16px (or a window of customized size for larger shapes) and shifts the window by 1px at every iteration. In this method, we vary the sprite to window matching algorithm by using full match, partial match, quick match, quick partial match and SSIM match detailed below.
  - Full Match: compares every pixel value of the window to the corresponding pixel value in the sprite. If there is one mismatched pixel, the sprite type will be rejected.
  - Partial Match: a less aggressive full match that allows up to 10% mismatch. This method is expected to take more time to run.
  - Quick Match: compares every pixel in the center of the window (cropping the window by n pixels from each side) to the corresponding pixel value in the sprite. If there is one mismatched pixel, the sprite type will be rejected. This matching technique is proposed to allow some room for variation in the background of a sprite when it's an enemy behind a bush versus an enemy with just the sky in the background.
  - Quick Partial Match: a less aggressive quick match that allows up to 10% mismatch. This method is expected to take more time to run than the Quick Match.
  - SSIM Match: compares a window and a sprite using the Structural Similarity Index Measure. The closer the SSIM to 1, the more similar the


Original Frame
Language
Reconstruction
9 => sky
7 => stairs block
6 => top right part of pipe
5 => top left part of pipe
4 => bottom right part of pipe
3 => bottom left part of pipe
2 => ground
1 => question block
0 => floating block

```
[[9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9]
 [9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9]
 [9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9]
 [9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9]
 [9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9]
 [9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9]
 [9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9]
 [9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9]
 [7 9 9 7 9 9 9 9 9 9 9 9 9 9 7 9]
 [7 9 9 7 7 9 9 9 9 9 9 9 9 7 7 9]
 [7 9 9 7 7 7 9 9 9 9 9 9 7 7 7 9]
 [7 9 9 7 7 7 7 9 9 9 9 7 7 7 7 9]
 [2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 9]]
```

```
[[9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9]
 [9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9]
 [9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9]
 [9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9]
 [9 9 9 9 9 9 9 9 9 9 9 9 1 9 9 9]
 [9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9]
 [9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9]
 [9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9]
 [9 9 9 9 9 9 1 9 9 9 0 1 0 1 0 9]
 [9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9]
 [9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9]
 [9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9]
 [2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2]]
```

```
[[9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9]
 [9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9]
 [9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9]
 [9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9]
 [9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9]
 [9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9]
 [9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9]
 [9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9]
 [9 9 9 9 9 9 9 9 9 9 5 6 9 9 9 9]
 [9 9 5 6 9 9 9 9 9 9 3 4 9 9 9 9]
 [9 9 3 4 9 9 9 9 9 9 3 4 9 9 9 9]
 [9 9 3 4 9 9 9 9 9 9 3 4 9 9 9 9]
 [2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 9]]
```

Figure 4.5: Transforming some frames from W1L1 of SMB into custom language with reconstruction (style transfer)

two images are. If the SSIM score is below a given threshold, the sprite type will be rejected.

- CV2M: a technique that calculates the normalized correlation coefficient to determine how similar the pixel intensities of the initial image and the template are. OpenCV provides such functionality and can be further augmented to detect multiple instances of the template within the original image. In some cases, this method will result in a considerably large set of matches caused by overlap. The non-maxima suppression (NMS) method can be used to filter out the unneeded matches. In this method, we are required to vary two important hyper-parameters: threshold score used by the CV detection algorithm and overlap threshold used by the NMS detailed below.
  - Threshold score: CV2M returns a matching score for each item detected, those with low scores are usually false positives so we aim to drop all matched elements that have a matching score lower than the threshold score.
  - Overlap Threshold: CV2M can match multiple items for the same template. However, in some cases, these items overlap. The overlap threshold helps to prune the detected items that have high intersection-over-union (IOU) overlap with previously detected items. At the end of the process, all items that have an IOU smaller than the overlap threshold will be omitted.
- Multi Template Matching (MTM): a technique to achieve object-recognition

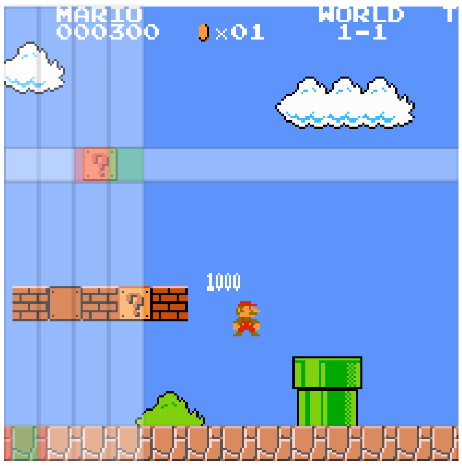


Figure 4.6: A frame example that starts with quarter a block (ground) in which tiles cannot be identified by running a window of 16x16 px

in images with repetition using one or several template images. Similar to the CV2M, it is based on openCV and we are required to vary the threshold score and overlap threshold.

We aim to test and experiment with these techniques in terms of speed and quality of output. To measure the quality of output we sample a total of 85 frames from three levels that belong each to the themes of interest: classic, underground and forest. We also describe these frames in our language manually. Then we compare our manual output to the generated output. We recall that the frames are cropped to 208 x 208 which is transformed into 13 x 13 blocks with each block expanding into the 16p x 16px sprite (208/16 = 13). We have a total of 169 integers in a 2D matrix for generated and manual output as we opt for a full matrices comparison. For each method, we define the matching error which is calculated by the formula

$$error = \frac{1}{l}\frac{1}{f}\sum_{i=1}^{i=l}\sum_{i=1}^{i=f}\frac{m_f}{t_f}$$

where

l: number of levels used in the experiment i.e. 3 in this case.

f: number of frames

m_f: number of misclassified entries in a given frame

t_f: total number of entries in a given frame

We also measure the time it takes for each algorithm to extract sprites while it's running. As displayed in Table 4.1, the CV2M and MTM technique outperforms the aggressive PX by PX extraction method in both performance and speed when tested on the previously manually annotated benchmark. However, we find that the MTM approach generates better results when tested on the classic genre which is our main target for this thesis. We then use it to save an array file per frame which we use as input for the GAN algorithm in the next step.

Technique	Accuracy (%)	Time (Seconds)
PX (Full Match)	76.36	1417.41
PX (Partial Match)	76.51	9324.09
PX (Quick Match)	76.61	749.16
PX (Quick Partial Match)	76.18	4902.24
PX (SSIM Match)	77	68934.32
MTM	77.40	11.23
CV2M	77.77	10.87

Table 4.1: The Accuracy and time measured for each matching technique on the manually labeled benchmark

At this point and similar to our first method, we explore running the three GAN algorithms (Simple GAN, DCGAN and WGAN) and we vary the input data provided as a single level, multiple levels under the same theme and two different yet similar-in-style games with the same theme. We detail each of our experiments in chapter 5.

4.3 PCG Evaluation

PCG evaluation is not an easy task as previously analyzed in chapter 3. We attempt to categorize PCG evaluation techniques and propose a solid framework. We recall that a level should be similar to the original games, fun and playable. As such, we will need to calculate our score based on these three notions. We first explore each of the notions on its own and explain the approach to calculate them.

4.3.1 ***FID***

As discussed in Chapter 3, the Frechet Inception Distance can help us with insights on how close a set of generated images are to a set of original images. The usage of FID is straightforward for learning from frames. However, it requires language data to be converted back to images in both original and generated samples in the case of learning from language. Both input and output images should share the same base of sprites.

4.3.2 ***Style***

We define the Style Diff metric as to how similar the generated levels are to the real levels of the original game. To quantify the style, we want to calculate style related metrics detailed below that define a given level.

1. Linearity: measures the profile of a level. To calculate linearity, we first find the highest points in a given vertical sequence of similar elements (sprites) and fit a line to these points. The aim is to calculate the R-squared goodness-of-fit measure to find how near points are to the fitted line. An example is displayed in Figure 4.7

2. Leniency: measures how difficult a level is in terms of structure. To calculate this score we add +0.1 for each enemy or gap found in the target level.
3. Empty Space: measures the percentage of the level that is empty by taking sprites as unit measure.
4. Interesting Space: measures the percentage of the level that is not empty, ground, normal block platform or stair.
5. Enemy Space: measures the percentage of the level that is enemy.
6. Gap Space: measures the percentage of the level that is gap. A gap is defined only at the first horizontal line of the level.
7. Negative Space: measures the percentage of the level that is passable by the player. A given position is considered passable by the player if it is within jump reach (three horizontal blocks) and is preceded by a sprite that can be stood on top such as pipe, ground or solid block.

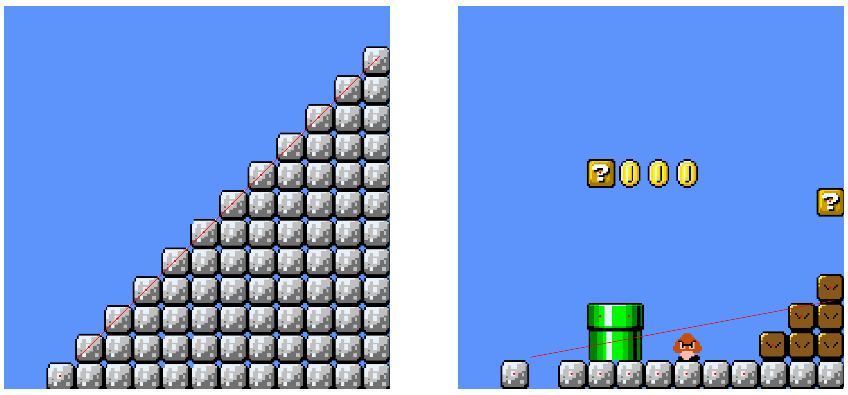

(a) Section with Linearity = 1.0 (b) Section with Linearity = 0.08

Figure 4.7: Calculating Linearity with R2 measure on red dots

The main goal is to see how different the original levels are to the generated levels in each of the metrics defined above. Hence, for each metric, we calculate the difference between the mean value of the original metric and the mean value of the generated metric with the following formula.

$$Metric_{Diff} = |OriginalMetric_{average} - GeneratedMetric_{average}|$$

To calculate the $Metric_{average}$ value, we first calculate the metric on each individual level of the set of generated levels or original levels trained on then calculate the mean by using the following formula.

$$Metric_{average} = \frac{1}{nblevels} \sum_{n=1}^{nblevels} Metric_n$$

To get one final number that indicates the similarity between generated levels and original levels, we calculate the balanced sum of all seven $Metric_{Diff}$ calculated by using the following formula.

$$StyleDiff = \sum_{m=1}^{7} Metric_{Diff_m}$$

The closer the Style Diff is to 0, the more similar the generated levels are to the original levels. In this sense, the goal of a generator is to produce a Style Diff that is neither too close to 0 (low originality of generated levels) nor too high (very far from the set of original levels).

4.3.3 *Playability*

Perhaps the most important matter in game generation is playability. If a level is not playable, then it is not useful. To calculate the playability of a level, we assign a score of 1 for a playable level or a score of 0 for an unplayable level. We run an AI Agent, inspired by Robin Baumgarten and implemented by Volz et al. with some modifications, on a set of generated levels to calculate the average score of playability.

4.3.4 *Fun*

As explained in chapter 2, fun is a subjective matter that is hard to quantify. It is best approximated through sampling real players to test the generated games and fill a survey with the level of excitement or happiness they felt while playing the level. This experiment is out of the scope of the thesis.

Chapter 5

Experiments & Evaluation

In this chapter, we present the process of experiments and we discuss and evaluate the different results for the two learning methods: learning from frames and learning from language.

5.0.1 *Learning from Frames*

In this set of experiments, we run three versions of GANs: simple Gan, DCGAN and WGAN. For each GAN version, we vary some hyper parameters such as bach size [32, 64], latent dimension [32, 50, 100, 200] and learning rate [0.0001, 0.0002]. Moreover, we fix the number of training epochs to 500, image size to 64, number of channels to 3 in all three GAN architectures. For the WGAN architecture, we fix the number of critics to 5 and the clip value to 0.01. In some cases, we prematurely stop the training process if the Generator or Discriminator diverges with a loss of greater than 50 or equal to 0 as we noticed that training further does not improve the generator. Figure 5.1 depicts the learning process of one of the conducted experiments at several stages starting from random noise at the initial stage, an image where it learns the sky, clouds and ground at an intermediate step, all the way to a more meaningful image.

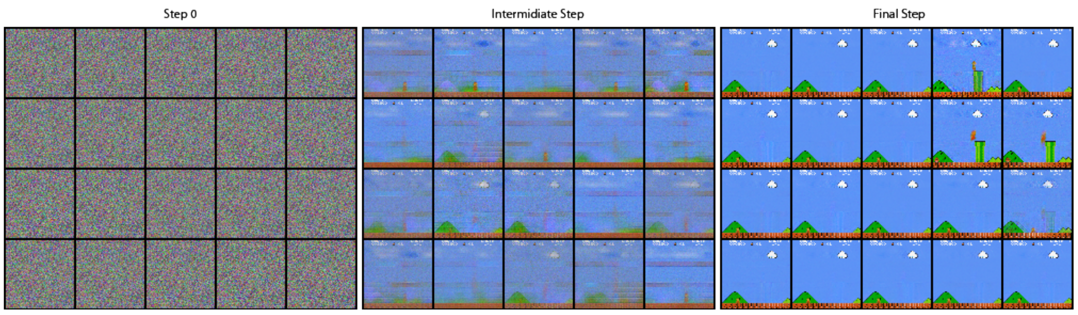


Figure 5.1: Learning Process of Simple GAN trained on (D_1)

We end up with 16 experiments for each architecture. Then, we pick the experiment that has the least mean total loss taking into consideration the losses of both generator and discriminator to bring the number of winning experiments to 3.

We repeat the same set of experiments on three different dataset variations which include:

1. World 1 Level 1 of Super Mario Bros without skipping any frames (D_1)
2. World 1 Level 1 of Super Mario Bros with skipping every 10 frames ((D_2)
3. Classical themed levels of Super Mario Bros with skipping every 10 frames (D_3)

Hence, we have a total number of 9 generators to evaluate by calculating the FID score on a large sample of original images paired with generated images from each generator. Table 5.1 shows the FID scores for all of the 9 experiments.

Dataset	Simple GAN	DCGAN	WGAN
D_1	167.19	194.33	181.74
D_2	224.84	236.73	233.97
D_3	231.08	215.21	201.83

Table 5.1: The FID scores of the three GAN architectures (Simple GAN, DCGAN, WGAN) over the three datasets (D_1, D_2, D_3)

If we rely on the FID score, we find that Simple GAN model on D_1 performs the best. This is confirmed by looking at the sample of images generated by this model in comparison to the ones generated by DCGAN and WGAN on the same set as shown in Figure 5.2.

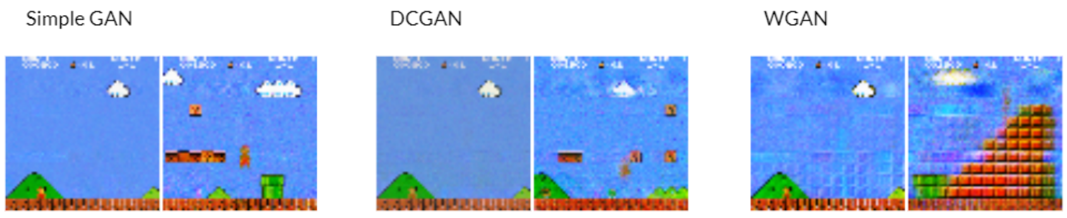


Figure 5.2: Output Samples of Simple GAN, DCGAN and WGAN trained on D_1

We also notice that WGAN-D_1's FID score is lower than WGAN-D3's FID score and this is again reflected in the sample results as displayed in Figure 5.3

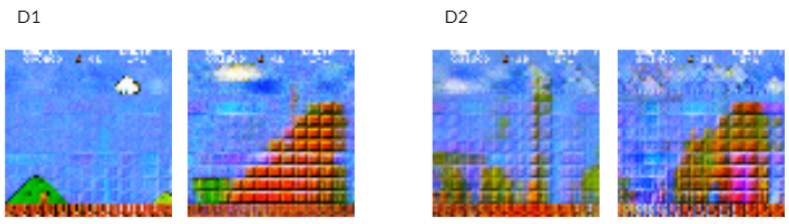


Figure 5.3: Output Samples of WGAN trained on D_1 and D_3

In other cases, FID score seems to be less reliable, especially when it comes to comparing the three GAN architectures across D2 where skipping frames is applied as seen in Figure 5.4.

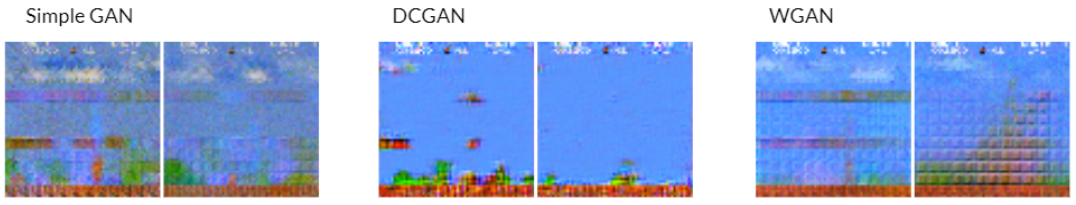


Figure 5.4: Output Samples of Simple GAN, DCGAN and WGAN trained on D_2

This variation in FID score could be due to the nature of generated images mixing up between several frames. As a result, we find that the quality of images generated by the simple GAN iN D_1 have the lowest FID score and represent the best visual outputs even though they are clearly plagued by mode collapse. Moreover, the poor quality of generated images in DCGAN and WGAN could be related to training on a very small dataset (around 3000 images in world 1 level 1) while also requiring an image size of 64x64 whereas MNIST dataset contains 20 times more the number of images with an image size of 28x28 in addition to the fact that mario frames are well varied whereas in MNIST dataset there is less variation in images.

5.0.2 *Learning from Language*

In this set of experiments, we also run three versions of GANs: simple Gan, DCGAN and WGAN. For each GAN version, we vary some hyper parameters such as bach size [32, 64], latent dimension [16, 32, 50, 100] and learning rate [0.0001, 0.0002]. Moreover, we fix the number of training epochs to 250 and the image size to 16 x 16 px. For the WGAN architecture, we fix the number of critics to 5 and the clip value to 0.01. Given that the number of channels is related to the number of elements/sprites that are incorporated in the learning process, we fix the number of channels to 18 for training on D_1 and to 23 for the other datasets since they include more sprites. In a similar way to the previous set of experiments, we prematurely stop the training process when needed. Figure 5.5 depicts the learning process of one of the conducted experiments at several stages starting from random sprites spread all over the place to a more meaningful section with better sprite positioning.

We also use the same picking criteria and end up with a single Simple GAN, DCGAN and WGAN model for each of the following dataset variations:

1. World 1 Level 1 of Super Mario Bros without skipping any frames (D_1)
2. World 1 Level 1 of Super Mario Bros with skipping every 10 frames (D_2)
3. Classical themed levels of Super Mario Bros with skipping every 10 frames (D_3)

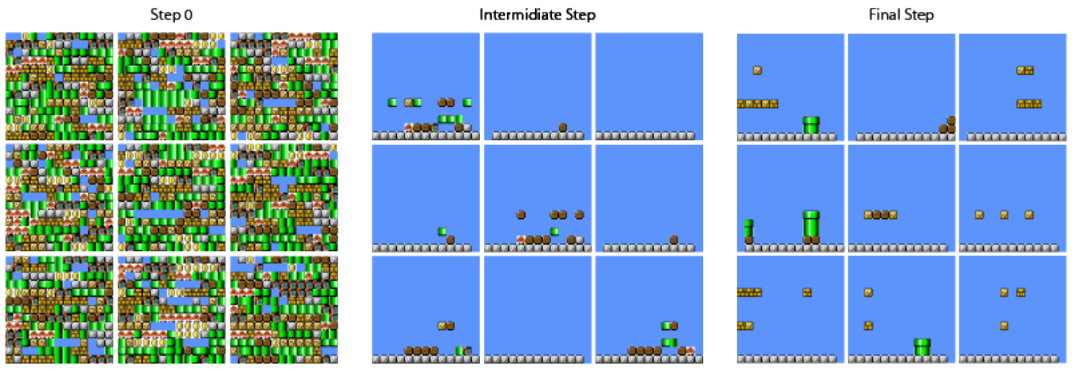


Figure 5.5: Learning Process of Simple WGAN trained on (D_1)

4. Classical themed levels of Super Mario Bros and Super Mario Bros Lost Levels with skipping every 10 frames (D_4)

In the sampled images, we see the ability of the generators to produce good quality output with sections that are similar to the original levels such as the ones presented in Figure 5.6. We also see some new structures that were previously unseen in the training levels, yet look interesting to play as depicted in Figure 5.7. Some generators can retain full pipes, a problem that were faced by many in the literature. However, some generators generate poor stylistic sections of SMB as displayed in Figure 5.8 with incomplete pipes or less meaningful stair structures.

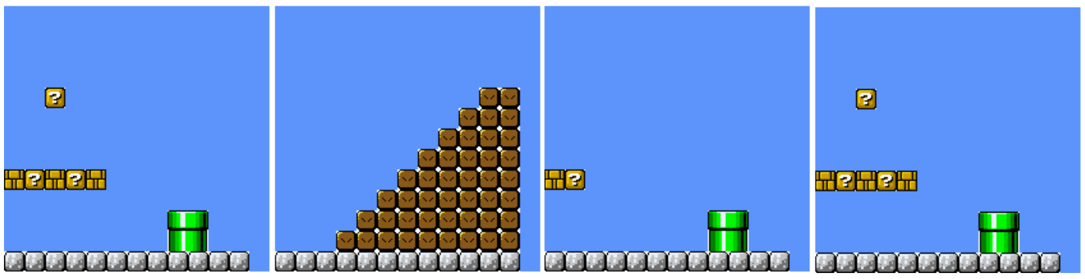


Figure 5.6: Good Quality Examples (Similar Style)

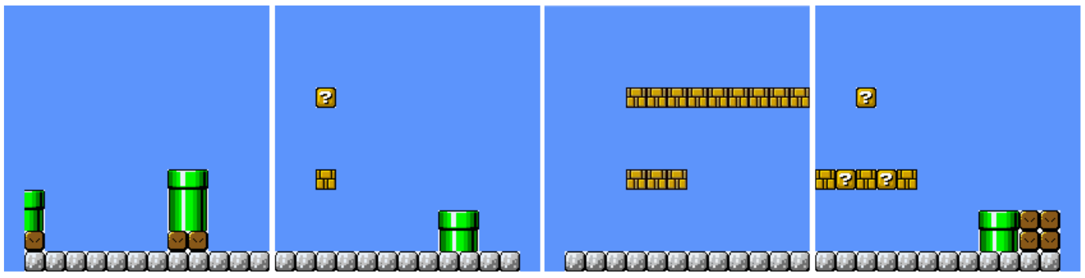


Figure 5.7: Good Novel Examples

To construct a full level, we simply append the generated sections together. Some examples of generated levels are depicted in Figure 5.9

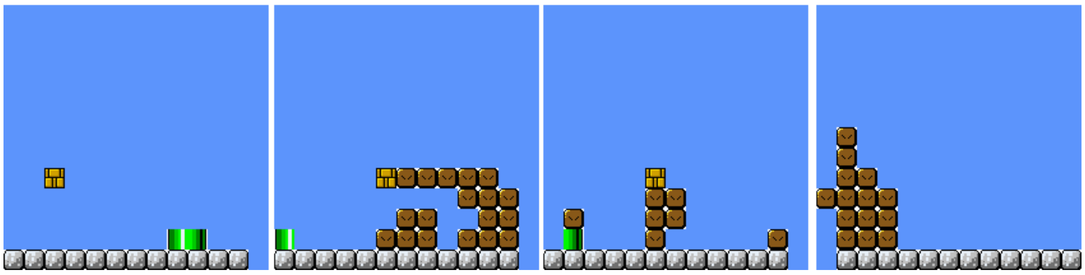


Figure 5.8: Poor Quality Examples

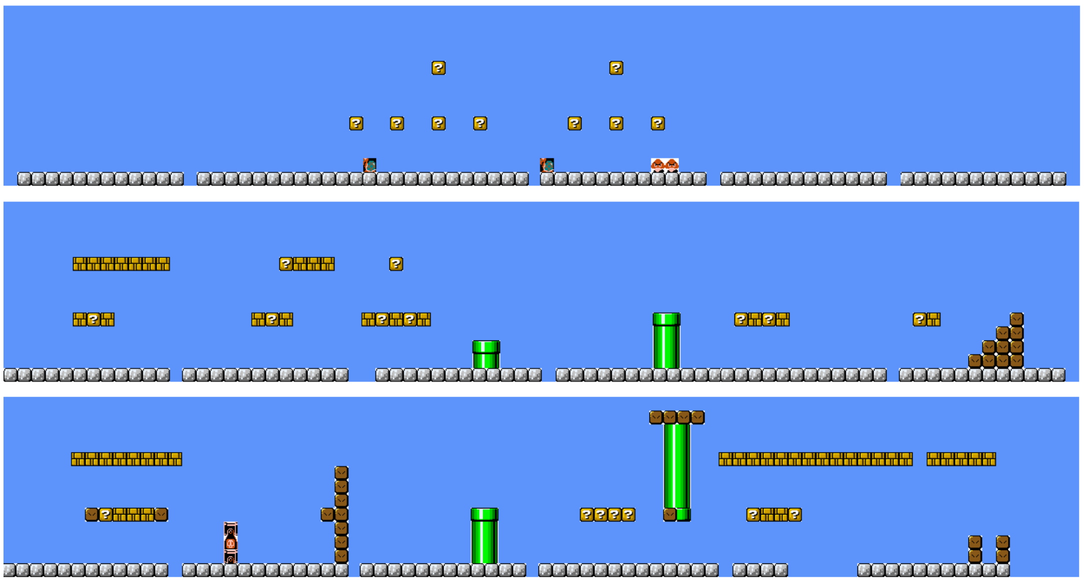


Figure 5.9: Some Examples of Generated Levels

To better evaluate the 12 generators, we run the following evaluation steps on each: (1) Calculate the FID score (2) Calculate the Style Diff Metric (3) Calculate the Playability score. Figures 5.10 to 5.13 depict five sample outputs per GAN architecture across datasets D_1 to D_4 along with their FID, Style Diff ($Sty.$) and Playability (P) scores.

As shown in Figure 5.10, the GAN in D_1 has a lower Sty Diff and a higher playability scores than the WGAN architecture but a higher FID score. This can be explained by GAN falling into the mode collapse issue where it generates low novelty levels causing the Style Diff to be low. However, WGAN solves the issue of mode collapse by generating slightly more novel sections and hence increasing the Style Diff and slightly lowering the playability score. Since WGAN's generator is able to cover more ground being more representative of the original data, it makes sense for the FID score to be lower. DCGAN's scoring metrics indicate the poor quality of generated output.

As shown in Figure 5.11, training on one level with frame skipping generates poor quality results in GAN and DCGAN in which there is high mode collapse despite having a high playability. However, WGAN is able to capture more insights on the

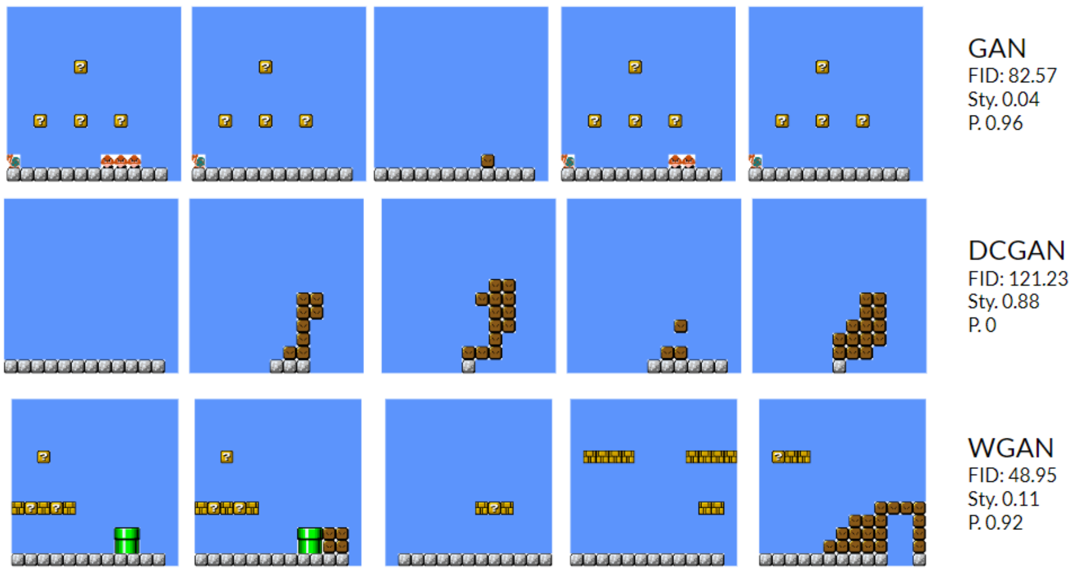


Figure 5.10: Output Samples of Simple GAN, DCGAN and WGAN trained on (D_1)

original frames despite the very low number of training data as it generates the best quality results once again with a playbility of 1, FID score of 46.66 and a Style Diff of 0.14.

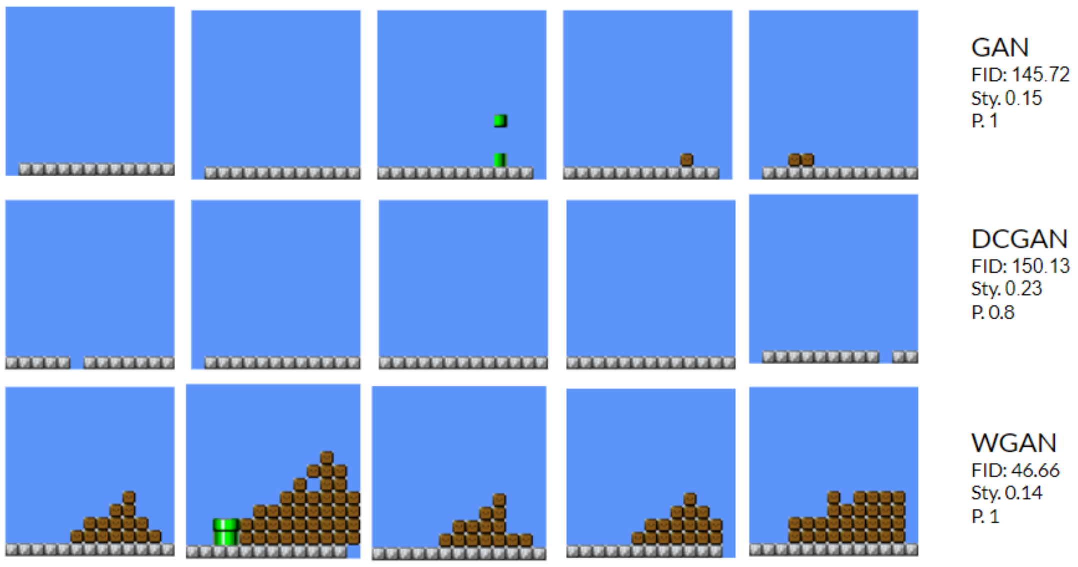


Figure 5.11: Output Samples of Simple GAN, DCGAN and WGAN trained on (D_2)

As depicted in Figure 5.12, training on several levels under the classic theme is possible and seems more robust with WGAN architecture. Moreover, it generated more varied samples in terms of manual inspection and relying on style Diff as well as its ability to score better in terms of FID. It is interesting to note that the playability is lower than the one provided by DCGAN and GAN due to their low variation nature.

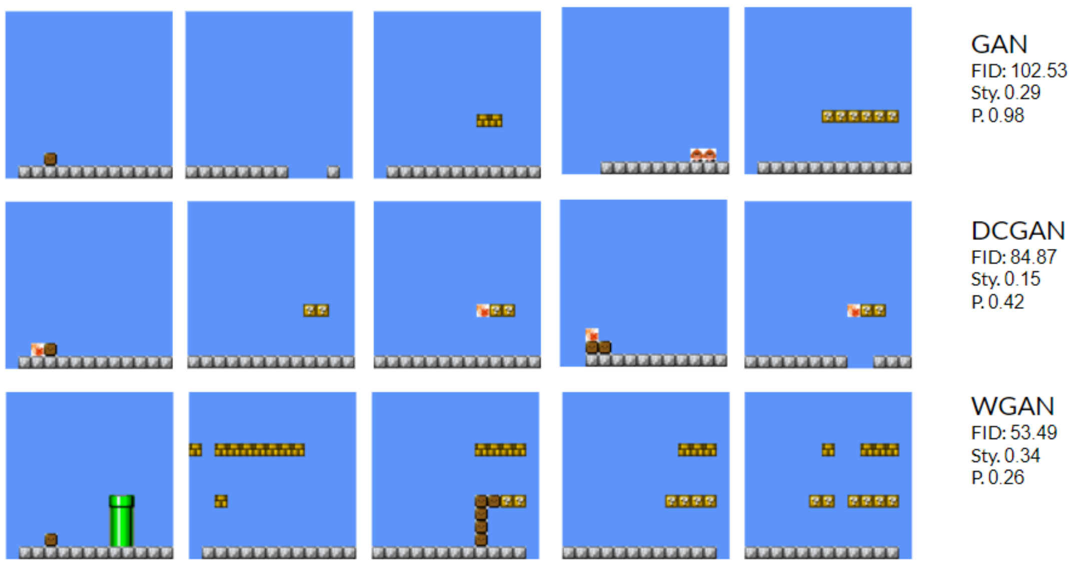


Figure 5.12: Output Samples of Simple GAN, DCGAN and WGAN trained on (D_3)

As shown in Figure 5.13, it is also possible to get meaningful output when training on two different games with a similar theme. We see a varied collection of samples that are provided by GAN and WGAN whereas DCGAN generates less quality output. In the case of WGAN, we note the large wall that was picked during training due to the high frequency of repetition of that wall in game. However, in the original game, this wall comes with either a spring or an additional block that makes it possible to overcome. However, in the generation process, we did not incorporate the spring which explains the low playability score.

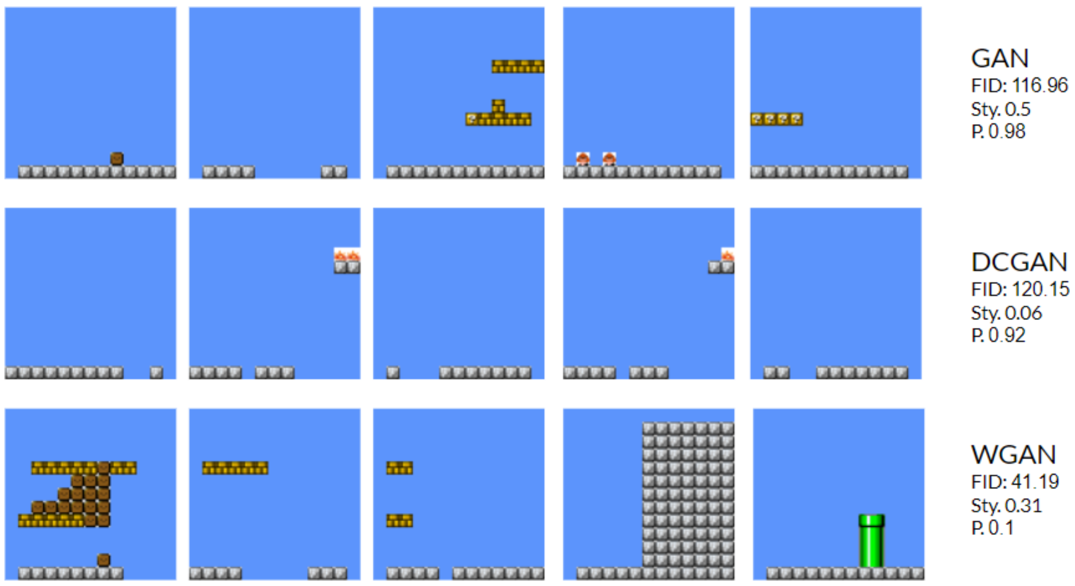


Figure 5.13: Output Samples of Simple GAN, DCGAN and WGAN trained on (D_4)

In conclusion, we find the GAN approach to have a low style diff score in general as it seems to have mode collapse issues most of the time. The high playability score is due to the availability of already existing sections from the original game that are known to be solvable. On the other hand, WGAN architecture has a higher style diff making the generated sections novel yet still related to SMB design rules. As a result, we note a decrease in the playability score as some of the generated sequences are not playable. This score can be slightly improved if more elements such as the spring are learned in the case of D_3 and D_4. Finally, the DCGAN architecture seems to be performing poorly and this indicates the need to experiment further with the architecture at hand to generate better results.

Chapter 6
Conclusion & Future Work

In this thesis, we described the state-of-the-art techniques for learning platformer game design and analyzed their advantages and disadvantages. Then, we explore the available datasets such as VGLC and include a video gameplay datasets for SMB and SMB Lost Levels as high quality footage to not only increase the number of data input by providing different positions of enemies and other objects but also to retain gameplay information in case needed for future research. Moreover, we provided a tool along the dataset to perform frame extraction, grayscaling and other useful image transformation and preprocessing functionalities.

Furthermore, we have been able to study and research two methods of learning defined as (1) learning from frames and (2) learning from language. We noted that learning from frames is not reliable and generates poor quality results due to the low similarity between different frames of the game in addition to training the networks on a small dataset despite the frame augmentation when compared to the actual MNIST dataset. The only exception is in Simple GAN network trained on World 1 Level 1 of SMB which surprisingly generates good results due to the mode collapse issue that is commonly known in Simple GAN architectures. On the other hand, learning from language requires the extraction of data (sprites) from frame images as a preprocessing step before being fed as input to the network. Given that it's harder to run a window on the frame to pick the sprites due to the shifting issue, we proposed three methods of extraction of which CV2M and MTM provided good results when tested against a manually labeled benchmark. After extracting the sprites, we run a series of experiments with GAN, DCGAN, WGAN on several datasets and provide a solid evaluation framework by separating the process into four categories (1) FID, (2) Playability, (3) Style Diff and (4) Fun. We find that WGAN generates good results on all four datasets including D_4 which is composed of two games. The results of WGAN have the smallest FID score, a Style Diff score that falls within an acceptable range that is neither too low (not novel) like the ones generated by GAN that has the mode collapse issue nor too high (output that does not learn well from the input game). However, the playability score given by WGAN should be improved to generated a larger number of playable levels.

6.0.1 ***Future Work***

GAN Procedural Content Generation has proved to be a good technique to generate platformer levels. However, we would like to improve the current WGAN architecture to (1) increase the space of generated playable level and (2) incorporate the missing spring sprite. We would also like to analyze other GAN architectures such as WGAN-GP which is a more stable WGAN architecture. Moreover, the technique presented seems to generalize well over two similar games and hence it would be interesting to test it on a larger collection of games not limited to the MARIO genre. Moreover, it would be very beneficial to implementing designer features to generate levels with requested parameters such as a predefined number of enemies or a desired difficulty as well as porting the generated language to game code that could run with Popular Game Engines such as Unity to make the creation of game less tedious. Finally, GAN PCG opens the door for other areas of research such as the ability to generate adaptive games based on player skill. In short, if the game is played by an experienced player, the generator will generate a more difficult level. On the other hand, if it is played by a beginner, it will generate a simpler level.

Bibliography

[1] C. Gough, *Number of games released on steam 2019*. www.statista.com/statistics/552623/number-games-released-steam/, Accessed: 2022-07-5 2019.

[2] S. Dahlskog and J. Togelius, “Patterns as Objectives for Level Generation,” *In Second Workshop on Design Patterns in Games (DPG 2013) Co-located with FDG 2013*, 2013.

[3] ——, “Patterns and procedural content generation,” *Association for Computing Machinery (ACM)*, 2012.

[4] M. Krishnan, *How did they do it: Temple run’s endless dash*. www.openxcell.com/blog/how-did-they-do-it-temple-runs-endless-dash/, Accessed: 2022-07-5 2015.

[5] L. A. Ripamonti, M. Mannalà, and D. Maggiorini, “Procedural content generation for platformers: designing and testing FUN PLEdGE.,” *Multimedia Tools and Applications*, 2017.

[6] A. Summerville, S. Philip, and M. Mateas, “MCMCTS PCG 4 SMB: Monte carlo tree search to guide platformer level generation,” *In proceedings of 11th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2015.

[7] A. Summerville and M. Mateas, “Super Mario as a String: Platformer Level Generation Via LSTMs,” *Association for the Advancement of Artificial Intelligence (AAAI)*, 2016.

[8] V. Volz, S. M. Lucas, J. Schrum, J. L. A. Smith, and S. Risi, “Evolving Mario levels in the latent space of a deep convolutional generative adversarial network,” *GECCO 2018 - Proceedings of the 2018 Genetic and Evolutionary Computation Conference*, 2018.

[9] J. Huizinga, *Homo Ludens: A study of the play-element in culture*. Taylor and Francis, 1949, ISBN: 0-7100-0578-4.

[10] B. Herbert, *The Grasshopper: Games, Life, and Utopia*. University of Toronto Press, 1978, ISBN: 0-8020-230 1-0.

[11] S. Rogers, *Level Up!: The Guide to Great Video Game Design*. Wiley, 2010, ISBN: 78-0-470-68867-0.

[12] “Nintendo, Super Mario Bros [Digital Game],” 1985.

[13] S.Dahlskog, J. Togelius, and M. Nelson, "Linear levels through n-grams," *Proceedings of the 18th International Academic MindTrek Conference*, 2014.

[14] G. Smith and J. Whitehead, "Analyzing the Expressive Range of a Level Generator," *PCGames 2010, CA, USA, ACM*, 2010.

[15] I. Goodfellow, J. Pouget-Abadie, M. Mirza, *et al.*, "Generative adversarial nets," *Advances in neural information processing systems*, vol. 27, 2014.

[16] J. Langr and V. Bok, *GANs in Action: Deep Learning with Generative Adversarial Networks*. Manning Publications Co, 2019, ISBN: 1-61729-556-6.

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

[18] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein gan. arxiv 2017," *arXiv preprint arXiv:1701.07875*, vol. 30, p. 4, 2017.

[19] M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler, and S. Hochreiter, "Gans trained by a two time-scale update rule converge to a local nash equilibrium," *Advances in neural information processing systems*, vol. 30, 2017.

[20] *Gan in pytorch with fid*, https://www.kaggle.com/code/ibtesama/gan-in-pytorch-with-fid/notebook, Accessed: 2022-07-5.

[21] "Template matching," *Open Source Computer Vision Library*, 2022.

[22] L. A. Ripamonti, M. Mannalà, and D. Maggiorini, "Procedural Content Generation via Machine Learning (PCGML).," *Multimedia Tools and Applications*, 2018.

[23] A. Summerville, S. Snodgrass, M. Mateas, and S. Ontanon, "The VGLC: The Video Game Level Corpus," *aaai*, 2016.

[24] M. Guzdial and M. Riedl, "Toward Game Level Generation from Gameplay Videos," *Procedural Content Generation Workshop 2015*, 2016.

[25] M. Guzdial, B. Li, and M. O. Riedl, "Game Engine Learning from Video," *IJCAI*, 2017.

[26] S.Dahlskog and J. Togelius, "A multi-level level generator," *In IEEE Conference on Computatonal Intelligence and Games, CIG*, 2014.

[27] S. Sondgrass and S. Ontanon, "Generating Maps Using Markov Chains," *aaai, 2013*, 2016.

[28] A. Canossa and G. Smith, "Towards a Procedural Evaluation Technique: Metrics for Level Design," *In The 10th International Conference on the Foundations of Digital Games*, 2015.