GAN-Based Game Level Generation
Engineering Design Document

Team Members: Sourish Banerjee[1]
David Chen[1]
Ya-Chuan Hsu[1]
Ujjwal Puri[1]

[1] Equal contribution

Table of Contents

1. Introduction

1.1. Overview

Game level generation has been a manual process that requires game designers, artists and engineers. There has been an uprise in the number of research conducted to have algorithms to generate gaming content. However, creating content that is not only playable but also contains specific attributes, such as required to perform long jump actions or dodge consecutive bullets, is an ongoing pursuit. We are particularly interested in exploring various Generative Adversarial Networks (GANs) to create game levels. Moreover, in the scope of this project, we would like to explore various methods to encode desired attributes in the generated game levels and quantify the quality of the created attribute-aware game levels through our designed evaluation metrics.

1.2. Motivation and Problem Description

The current trend of popular commercial games is steering in the direction of having larger, prettier, more atmospheric, and hundreds of thousands of details included. It is easy to foresee the challenge in maintaining a consistent output of high-quality gaming content as the cycle of game production is also shortening. The amount of work towards generating various game content is the key to engaging gamers and keeping them interested in continuously exploring.

Our work is inspired by the approach conducted by Volz et al. [1], improving game level generation by combining the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) with the training of a Generative Adversarial Network (GAN). However, the game levels generated through their implementation has presented issues that require further exploration. Since their generated game was Super Mario Bros, a 2D side-scrolling game, there were artifacts at the stitching of the generated frames. For example, there were arrangements of the tiles that were not as natural and easily spotted by a human eye. Another issue worth pointing out is that some tiles would formulate areas that were inaccessible to players. Finally, we tested their GAN implementation with Lode Runner -- a game that has a more complex map design with structures spanning both length and height of the map, and the generated levels looked like a mess of assets with no structural sense.

1.3. Prior Midterm Achieved Progress

We aim to focus on the following items, allowing us to build up to generating a game level with desired attributes encoded:

- Enhance the smoothness of the frame stitching for a 2D side-scrolling game via conditional deep convolutional GAN

- Design a Multi-Stage GAN for enabling complex game level generation
- Read up on Latent Space Variable evolution methods for brainstorming design ideas for post-midterm implementation
- Implement our own generated game level evaluation metrics

		September				October	
Games	Select game corpus						
	Visualize generated levels						
Conditional DCGAN	Literature review						
	Implementation						
Multi-Stage GAN	Literature review						
	Implementation						
Latent space exploration	Literature review						
Evaluation metrics	Literature review						
	Implementation						
Documentation	Presentation						
	EDD						
	Technical report						

1.4. Post Midterm Achieved Progress

With our prior work done on the Conditional DCGAN and the evaluation metrics, we were able to start implementing latent space exploration in order to find the noise for particular level characteristics we were interested in. We also continuously work on the multi-stage GAN to capture the details of more complex-structured games. The timeline for the post-midterm is shown as follow:

		October		November			
Conditional DCGAN	Literature review						
	Implementation						
Multi-Stage GAN	Implementation						
Latent space exploration	Literature review						
	Implementation						
Evaluation metrics	Literature review						
	Implementation						
Documentation	Presentation						
	EDD						
	Technical report						

2. Overview of the Games

In the project, we would like to analyze the quality of our game level generators for 2D side-scrolling games and a game with a complex 2D map. Thus, we respectively chose Super Mario Bros and the Lode Runner.

2.1. Super Mario Bros

Super Mario Bros is one of the best selling games in the world with over 40 million physical copies. It was created two years after the original version of Super Mario Bros, the Mario Bros, was released in 1983. Both games were designed by Shigeru Miyamoto and Takashi Tezuka. Unlike the original version, Super Mario Bros has more colorful game levels and is scrollable with larger characters (the name of Super Mario Bros was coined once the Super Mushroom power-up was implemented)[2].

Fig. 1: The first version of Super Mario Bros. released on October 18th, 1985.

2.1.1. Gameplay

The game starts off with a traject incident of the Mushroom Kingdom being invaded by the Koopa, a tribe of turtles that possess black magic. The Koopas ruined the peaceful town by casting the peace-loving Mushroom People to stones, bricks and horsehair plants. The only hope for the Mushroom People to return back to normal is for the daughter of the Mushroom King, Princess Toadstool, to lift the casted dark magic. Unfortunately, the Bower, King of the Koopa, was smart enough to kidnap Princess Toadstool and keep her in his castle. The destiny of the Mushroom Kingdom and its people is in the hands of the player as they control the beloved character Mario on a quest to save Princess Toadstool and restore the fallen kingdom.

[2] Fun fact: As Super Mario Bros. were designed nearly at the same time with The Legend of Zelda, they share some game elements such as the fire bars seen in the Mario castle level was actually an object initially developed for Zelda.

On the road to the castle, there are coins scattered around or hidden in special bricks marked with a question mark that awaits for Mario to collect. When more than 100 coins are collected, Mario will gain an extra life. There are also invisible blocks hidden in the map that, when hit, will pop out loads of coins or even rare items. One of the classic rare items is the Super Mushroom that doubles the size of Mario once it is eaten. Mario will most certainly encounter enemies and is able to defeat them by jumping on top of them. Another dangerous item seen on the road is the cannons that will occasionally shootout deadly bullets. Once Mario gets caught by the enemies or fails to dodge a bullet, he will lose one life. But, if Mario is in Super Mushroom mode, then he will only shrink back to his normal size without losing a life. The game ends when the player runs out of lives or successfully reaches the castle and saves the Princess.

The action space of Mario consists of moving right and left, ducking and jumping. A player can input multiple actions at the same time (eg. jumping to the right).

2.1.2. Mario AI Competition

The competition started in 2009 and ceased in 2012. The original content of the competition was to create the best controller for Mario in the game of Super Mario Bros. The measurement of the controller was based on the number of levels the bot was able to complete with a limit of 24 seconds for the controller to output an action for Mario. As it turns out, the winner of the competition utilized A* search for designing a simple and fast algorithm that is proven to be a great direction for the competition setup [2] Later on, the organizer included two other tracks for people to compete, one of which was the Level Generation Track. The generated games were evaluated by game testers playing them live at the competition event. The competition then evolved into a more general competition called the Platformer AI Competition in 2013. Even though it is “a bit” late for jumping into the Mario AI competition, there was a release of the tenth-anniversary edition of the Mario AI Framework, used for the competition, available on GitHub[3].

We build on the Mario AI Framework to evaluate our generated levels and to also play through them.

2.2. Lode Runner

This puzzle-like video game was created by Doug Smith and published by Broderbund in 1983. It was one of the first video games that included a level editor, which allows

[3] https://github.com/amidos2006/Mario-AI-Framework

players to create their own levels and play with. Interestingly, this feature was created due to Smith's neighborhood kids, whom he tested the games with, request. At the time of the release, several levels designed by the neighborhood kids were actually included.

2.2.1. Gameplay

In the game, the player controls a character who steals all the gold at each level while being chased by several guards. The level consists of several vertical and horizontal ladders for the player to escape from one platform to another. Additionally, the character has the ability to dig holes on the platform to create traps. Once a guard falls into a trap, the player is able to walk over the guard safely. And, of course, the trap would only last for a few seconds, so that the player should always be their guard. A ladder leading to the next level will appeal to the player once they finish collecting all the gold on the map.

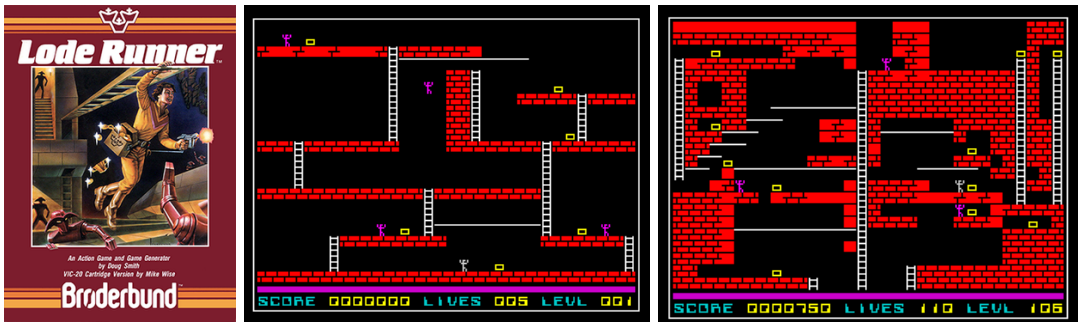


Fig. 2: On the left is the cover art for the Lode Runner [10] and the two images on the right are two different game levels.

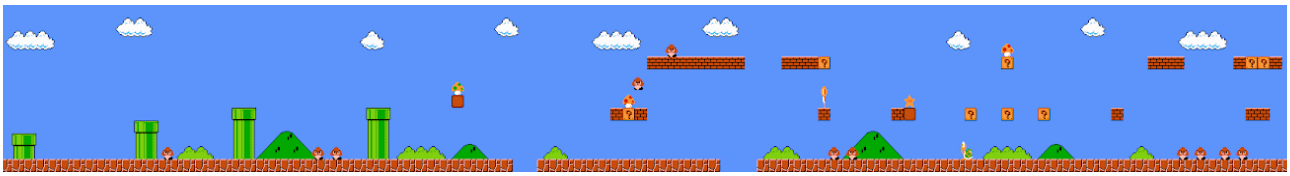
The design of the game introduces a strategy as the player can only dig holes to the sides but no beneath. The twist applies when some gold blocks are stored between bricks, and the player must plan to dig a gap at least the same number of blocks deep the gold is to successfully reach the gold without being trapped. Moreover, the guards do not always take the shortest path to the player but can move in ways the player cannot intuitively expect. Mastering the game requires some patience to lead the guard as far away from the character as possible and leave enough space for the character to collect all the gold and reach the exit ladder.

2.3. The Video Game Level Corpus (VGLC)

To actually be able to create a generator for game content, we first need to have some data on the game levels to train with. Our search for the different game corpus lead us to find various interesting frameworks and competitions with some game level corpus included, such as the General Video Game AI Gym [3], the Mario AI Framework and the

Video Game Level Corpus (VGLC) [4]. We eventually decided to mainly work with VGLC as it has the most comprehensive game level corpus for our two selected games.

The file type of the game level corpus that is included in the VGLC is in JSON format and text format. Since most games in VGLC are tile-based, the corpuses use particular symbols to represent different tile types. In order to input the data for training our GANs, we read in the text files as a two-dimensional string array and output in the same format. A python script is then written to decode the symbols and map them back to tile images to see the full generated level.



```
--------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------
-------------------------------------------------E------------------------------------------------------------
-----------------------------------------------SSSSSSSS---SSSQ-------------?----------SSS----SQQS-------------
--------------------------------------------------------------------------------------------------------------
-----------------------------------------------E--------------------------------------------------------------
-------------------------------------S------------------------------------------------------------------------
--------------------<>---------<>-----------------S?S-------------S-----SS----Q--Q--Q-----S----------SS--------
------------<>------[]---------[]-----------------------------------------------------------------------------
--<>--------[]------[]---------[]-----------------------------------------------------------------------------
--[]--------[]-E----[]-----E-E-[]-------------------------------E-E--------E----------------EE-E-E-------------
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX--XXXXXXXXXXXXXXX---XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Fig. 3: A section of Super Mario Bros. game level (top) and its annotated text symbol file (bottom)

On top of creating a mapping script in python, we also spent some effort improving the tile image files in the VGLC (as shown in Table 1). The original game tiles used for visualizing the generated Super Mario Bros level were outdated and a few were even missing. As for the Lode Runner, there were no tile images included in VGLC; thus, we created our own tile images. The generated game levels that will be later seen in the experiment section are a result of this process.

Table 1: Mapped symbols to updated tile images for Super Mario Bros.

Tile type	Symbol	Identity	Visualization
Solid/Ground	X	0	
Breakable tiles	S	1	
Background	-	2	
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	
Coin	o	10	
Cannon	B	11	
Cannon stand	b	12	

Table 2: Created tile images for Lode Runner

Tile type	Symbol	Identity	Visualization
Ladder	#	0	
Background	.	1	
Brick	b	2	
Bedrock	B	3	
Gold block	G	4	
Horizontal ladder	-	5	
Player	M	6	
Guards	E	7	

3. Prior related research

3.1. Procedural Content Generation (PCG)

Procedural Content Generation can be thought of as an algorithmic way of generating randomized content. It first saw its use in games in around the 1980s. The game Elite entirely relied on PCG for all its levels, encompassing eight galaxies, with above 200 planets to explore in each. Back then, the drive capacities were a big bottleneck for shipping all the content; hence the developers decided to leverage the memory saving benefits of using a PCG, i.e., only shipping the generation code with a seed value.

Video game graphics has also seen a rise in the use of PCG techniques to generate different assets like models, textures, and even sounds. Many recent open-world games with ambitious scales have used PCG to design assets and structure those assets together. One impressive example would be Star Citizen. It is a game that is still in development with a scale ranging from galaxies to cities. For generating such large amounts of assets and content for the game, the developers rely heavily on PCG to

generate from core assets like plantations, buildings, textures to actual layouts of rooms, buildings, cities (Fig. 5), etc.

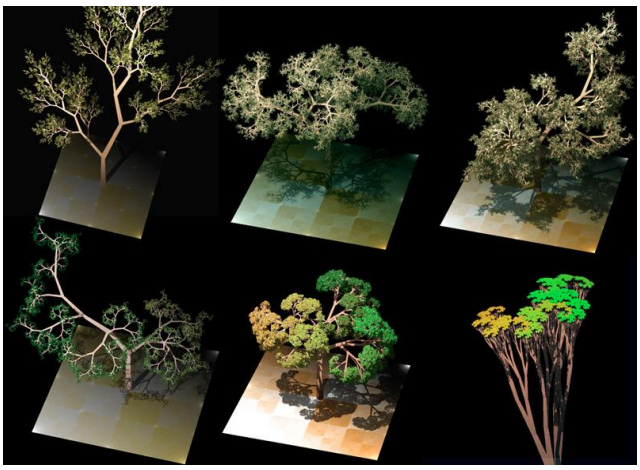


Fig. 4: Showcasing the variation in generation of trees using PCG.

Fig. 5: Star Citizen: procedurally generated city.

It is apparent that PCG saves several man-hours in generating content for games, but one of the caveats is the difficulty of controlling this random process. This massive space of potential content is bound to contain assets that are broken and unplayable. Even though the generation is restricted to always being playable, the problem of controlling desired features in the generated content is still ever-present. In our project, this is one of the aspects we aim to cover in the second half of the semester.

3.2. Generative Adversarial Networks

Generative Adversarial Networks is a generative model proposed by Goodfellow et al. [5]. A typical GAN architecture contains two sections, shown in Fig. 6, one of which is a discriminator D network that aims to distinguish between generated images and real

ones; the other is a generator *G* network that seeks to create images that fool the discriminator. The generator takes a randomized input distribution $z \sim p_z$, usually sampled from a predefined latent space, as input and defines a probability distribution p_g. The object of the GAN is to learn p_g that approximates the real data distribution p_r. This is accomplished by optimizing the joint loss function for D and G.

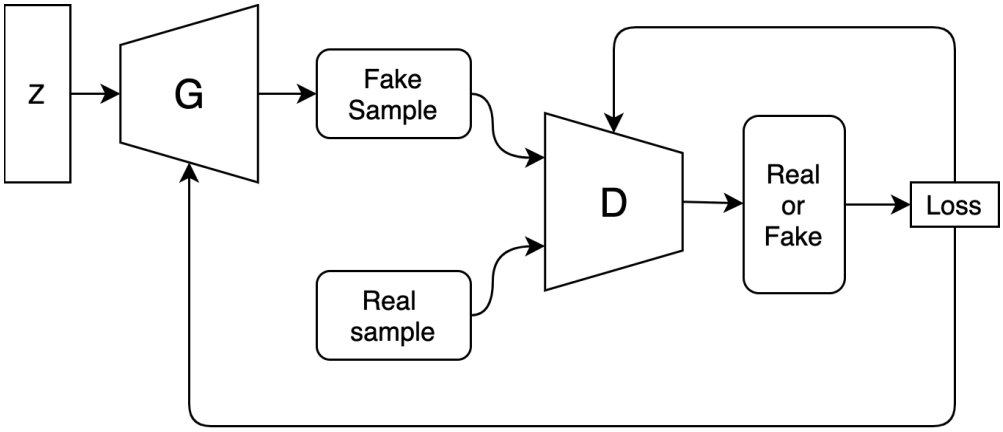


Fig. 6: An architecture of a typical Generative Adversarial Networks (GANs)

The variety of GAN architectures is large. Compared to the original GAN, which used a fully-connected neural network for both its discriminator and generator, researchers have gotten creative while experimenting with this not-long-ago proposed idea. A team of researchers had eyes on changing the unsupervised feature of the original GAN and decided to introduce a Semi-supervised GAN [12]. By taking class labels and feeding them into the D and G networks, Mirza and Osindero [13] enhanced GAN's discriminative abilities. Radford et al. [14] incorporated deep convolutional neural networks with GANs and created Deep Convolutional GAN, which is an architecture that we will be building on top of. There is also a recent proposal of including cues of all feature locations and ensuring the large receptive field without sacrificing computational efficiency in GAN, called the Self-Attention GAN [15], demonstrates some state-of-the-art results.

The generative capacity of GANs over the years has been well established. One of the best examples that depict GAN's generative capacity can be seen at https://thispersondoesnotexist.com/ (results are shown in Fig. 7). The amazing results of GAN motivates us for using GANs and its variations for this project.

Fig. 7: Generated images of fake human faces.

3.3. Level Generation Using GANs

The earliest relevant work to ours where the GAN architecture was explored to generate levels for Super Mario Bros. is done by Volz et al. in "Evolving Mario levels in the latent space of a deep convolutional generative adversarial network." [1]. They explored level generation of Mario levels using a vanilla DCGAN, where frames of size 14×28 were generated in isolation and then stitched together to create a full-sized level.

The authors also talked about not having a large enough training set for the Mario levels -- they only had one Mario level worth of data. As one could imagine, their results had issues. The stitching of these generated frames posed trouble as the frames had no context flowing through them, a theme or the desired features could not be carried across the whole level. The work also used an older text encoding scheme for the levels, which represented a *pipe* with four different symbols instead of just one symbol denoting a *pipe*. This type of encoding led to the generator not being able to learn to form full pipes, and the frames usually ended up with broken pipes littered throughout (see Fig. 8).

Our current two approaches try to counter this isolated generation of frames. The conditional DCGAN passes the previous frame as a condition for the generation of the next frame, while the multi-stage GAN first generates a structure for a level and then at the next stage fills in the details for that structure frame-by-frame.

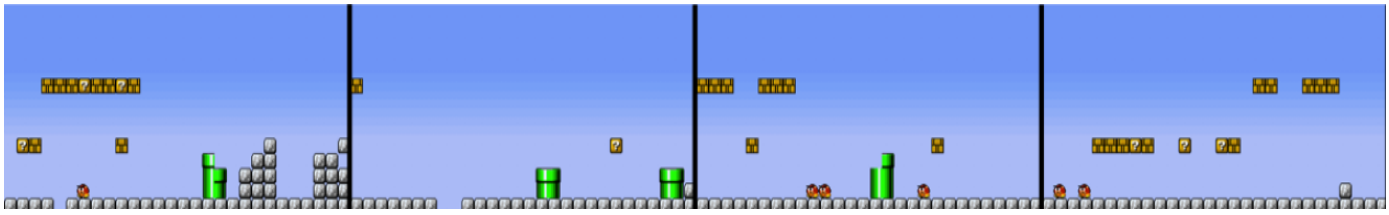


Fig. 8: Frames generated using GAN in [1], showcasing the broken pipe structures.

3.4. Controlling Generation

The work conducted by Volz et al. [1] also explored the latent space of the GAN in order to find regions corresponding to desired features using Covariance Matrix Adaptation Evolution Strategy (CMA-ES). This method requires hand crafted fitness functions that score the generated levels on their “goodness” in accordance with the desired features. Schrum et al. [6] in their work, overcome the rigidity posed by the objective scores of the fitness functions, using an interactive tool that performs the Latent Variable Evolution (LVE) while keeping humans in the loop. There is also work done by Fontaine et al. [7], which introduces a new method -- Latent Space Illumination (LSI) that uses MAP-Elites (a Quality Diversity algorithm) to generate a diverse variety of levels based on predetermined desired features.

3.5. Evaluation of Generated Levels

In terms of comparing GAN generated levels with human-generated levels, many techniques have been used to quantify the quality of generated levels. Simon and Volz [8] have shown in 2019 that it is possible to compare the tile distributions of GAN generated levels with ground-truth levels with Kullback-Leibler divergence (KL divergence). KL divergence is a measure of similarity between two probability distributions; however, in order to use this metric with level representations, they computed the occurrences of all unique 2x2 and 4x4 tiles.

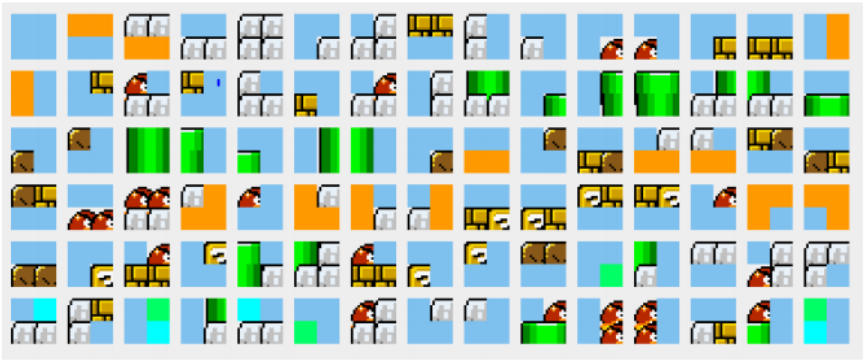


Fig. 9: Unique 2x2 tiles obtained by using a sliding window across a level [8].

Earlier research was done by Giacomello et al. [9] on the video game DOOM using a similar window-based approach that incorporates the structural similarity metric (SSIM). SSIM takes two image windows size NxN and computes a formula based on averages, variance and covariances. Agent-based evaluation metrics have also been explored in the past. However, these metrics are usually limited to whether the agent can successfully play the level or not, and is not implemented to evaluate more complex features.

4. Methodology

4.1. GAN Architectures

4.1.1. Baseline

In order to evaluate the performance of our models, we use the DCGAN architecture outlined in [1] as our baseline model. This is adapted from and trained using the WGAN algorithm [11]. The architecture details of the baseline, as well as all our implemented models (Section 4.1.2 and Section 4.1.3), are outlined in Table 3.

In addition to this, we also evaluate our frameworks on games with different characteristic features. As a sanity check, we use a helicopter game that we wrote ourselves. It is a simple environment with only three types of blocks and follows a fairly regular structure that should be trivial to generate. Super Mario Bros is a side-scrolling game in which every frame has a large percentage of empty-passable tiles. The non-empty tiles usually fall into simple patterns that should be relatively easy to generate. The challenge, however, is to model the spatial relationships between these structures across the length of a generated level in a manner that encapsulates natural human-level design characteristics. Lode Runner [10], on the other hand, has short single frame levels. However, these frames encompass highly structured non-empty tile combinations, and it is a challenging task for a traditional generator network to hallucinate similar levels while at the same time satisfying spatial constraints that guarantee playability.

4.1.2. Conditional DCGAN

Our primary motivation for conditioning the DCGAN on the previous frame was to keep passing the general context of structure across the generated frames, which then get stitched into one full level. This passed context of the previous frame also enables the DCGAN to learn the short term structural dependencies in a level.

4.1.2.1. Generator Architecture

The input to generator is comprised of selected feature channels of the previous frame (e.g. ground tiles, pipes and brick tiles) and one channel of $\mathrm{N}(0,\ 1)$ gaussian noise. It is first passed through 4 Convolutional→ReLU layers to get a compressed form of the previous frame's features. The compressed form then gets flattened and passed through 5 Transposed Convolutional→Batchnorm→ReLU layers, the activation of the output layer being Tanh.

4.1.2.2. Discriminator Architecture

The input for the discriminator is the conditioned frame stitched with the output from the generator. Our discriminator has the same architecture as that of a vanilla DCGAN discriminator. It comprises of 4 Convolutional → LeakyReLU→Dropout layers.

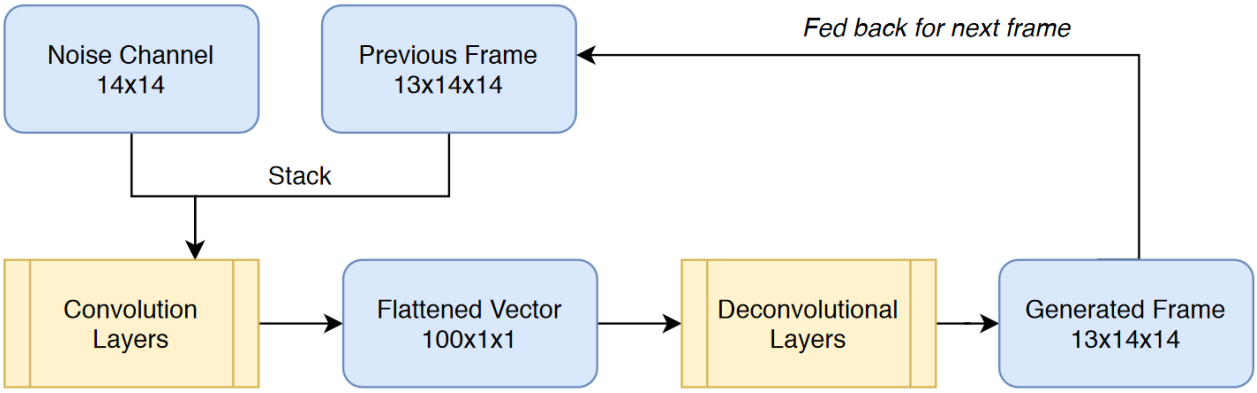


Fig. 10: Shows the overview of the generator architecture of the Conditional DCGAN

4.1.3. Multi-Stage GAN

The motivating factors for implementing a multi-stage GAN architecture are twofold: 1) we want to break up the complex task of hallucinating game levels into simpler tasks that individual GANs can tackle, and 2) we want to condition the GAN input on structural properties of levels during training so that we can preferentially generate levels with desirable structural properties from our trained GAN. We train two GAN architectures successively to achieve this. The training framework is illustrated in Fig. 11 and described in Sections 4.1.3.1 and 4.1.3.2. Architecture details are outlined in Table 3.

4.1.3.1. Stage-1 (Structure) GAN Architecture

The Stage-1 GAN trains to learn a model that can generate level structure images consisting of 1/0 bits indicating the presence/absence of non-empty tiles conditioned over desirable structural properties passed to it as input. The generator is trained on a 32-bit noise vector drawn from a $\mathrm{N}(0, 1)$normal distribution. For the condition, we try out different experiments. We achieve best results using a significant portion of the previously generated frames (output of the Color GAN from previous iterations) as condition. We also try using as condition a 5-bit vector of zeros and ones with each number representing the expected presence or absence of a structural property in the generated image. The generator

outputs a 32x32x2 image with a channel each for empty and non-empty tiles respectively.

The discriminator is trained on a 32x32xN tensor consisting of 2 channels for the input image and the other channels encoding the condition. The input is either drawn from a train set derived from human designed levels or an output from the generator. The train set is made up of 32x32 images where each pixel is either 1 or 0 indicating the presence or absence of a non-empty tile respectively. In the case where the condition is previously generated frames, they are directly appended to the input. For the images drawn from the train set, the condition frames are also present (processed from previous frames during creation of the train set from the level corpus). For the model using the structure-vector as condition, we use 5 conditional channels each containing only ones or zeros representing the presence or absence of the corresponding structural properties in the condition vector. For inputs drawn from the train set, this is derived from the original human-designed levels. For outputs from the generator being passed as input, this is derived from the randomly generated condition vector passed as input to the generator by duplicating each bit of the condition vector across entire 32x32 channels. The discriminator outputs a single bit approximating the Wasserstein distance between the modeled and true distributions.

4.1.3.2. Stage-2 (Color) GAN Architecture

The Stage-2 GAN is trained to color in a structure image for a level. The train set here comprises 32x32xN human-designed game levels (where N is the number of tiles in the target game) and their corresponding 2-channel structure images as used in training the Stage-1 GAN. Each channel of an image in the train set contains only ones or zeros indicating the presence or absence of a particular game tile as shown in Section 2.3. The generator input is structure images drawn from the train set. The generator output is a 32x32xN game level image. Unlike in the Stage-1 GAN, the loss for the generator here is a sum of both the adversarial loss as well as the L2 loss computed between the generated images and the images in the train-set corresponding to the structure channels passed to the generator for the coloring process.

The discriminator input is 32x32xN game level images along with their corresponding 2-channel structures. The input is either drawn from the training set or an output from the generator. In case of the latter, the

structure channels are the condition channels that were passed to the generator for coloring. Like in the Stage-1 GAN, The discriminator outputs a single bit approximating the Wasserstein distance between the modeled and true distributions.

The Stage-1 and Stage-2 GANs are trained separately. Finally, to generate a level, a seed image from the train set or a 5-bit condition vector is chosen based on the structural properties we desire to observe in the final generated level. This is passed to the Stage-1 generator along with some $\mathrm{N}(0,\ 1)$ noise to generate a 2-channel level structure. The Stage-2 GAN then takes in the generated structure as condition and generates the final colored generated game level as output. This process is repeated for as many iterations as needed. When using level frames as condition, for subsequent iterations, some combination of the output from previous iterations is used as condition for the Stage-1 GAN.

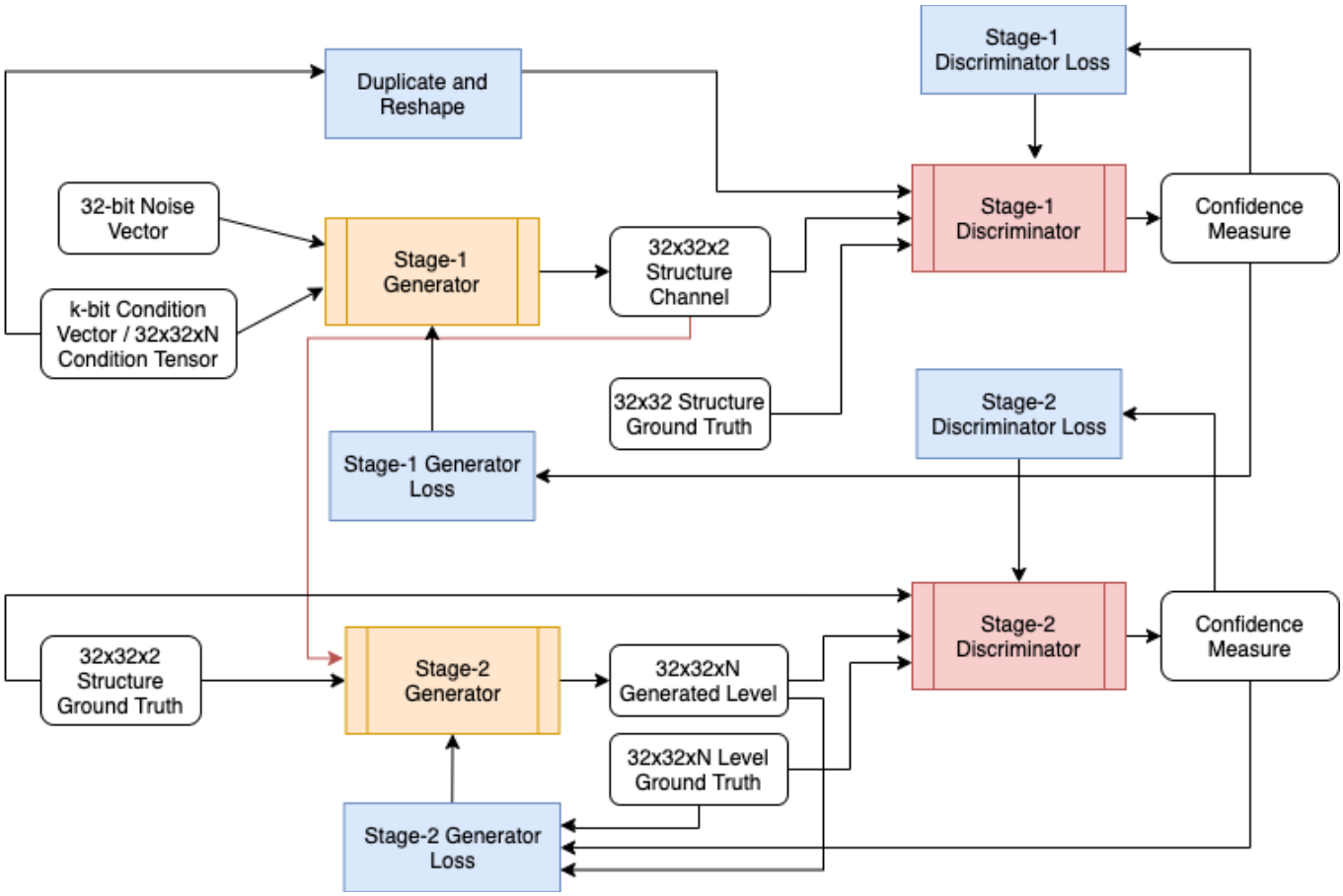


Fig. 11: Training workflow for Multi-Stage GAN architecture. Red arrow shows the output from the Stage-1 Generator is passed as a condition to the Stage-2 Generator for game-level generation after training is completed for both stages.

Table 3: GAN architecture details

Baseline GAN	**Generator**		**Discriminator**	
No. of Feature Maps	256U, 128U, 64U, 1U		64C, 128C, 256C, 1C	
Conv. Kernel Size	4, 4, 4, 4		4, 4, 4, 4	
Conditional DCGAN	**Generator**		**Discriminator**	
No. of Feature Maps	5C, 20C, 16C, 8C, 100U, 256U, 128U, 64U		13C, 32C, 64C, 128C, 1F	
Conv. Kernel Size	5, 3, 3, 2, 4, 4, 4, 4		4, 4, 4, 4	
Multi-Stage GAN (Original)	**Stage-1 (G)**	**Stage-2 (G)**	**Stage-1 (D)**	**Stage-2(D)**
No. of Feature Maps	256U*, 128U, 64U, 2U	64C, 128C, 256C, 128U, 64U, 'N'U	64C, 128C, 256C, 1C	32C, 64C, 128C, 256C, 1C
Conv. Kernel Size	4, 4, 4, 4	4, 4, 3, 3, 4, 4	4, 4, 4, 4	4, 4, 4, 4, 4
Multi-Stage GAN (Combined)	**Stage-1 (G)**	**Stage-2 (G)**	**Stage-1 (D)**	**Stage-2(D)**
No. of Feature Maps	**S1: 64C, 128C, 256C S2: 256U S3: 256U, 128U, 64U, 2U	64C, 128C, 256C, 128U, 64U, 'N'U	64C, 128C, 256C, 1C	32C, 64C, 128C, 256C, 1C
Conv. Kernel Size	S1: 4, 4, 3 S2: 4 S3: 3, 4, 4, 4	4, 4, 3, 3, 4, 4	4, 4, 4, 4	4, 4, 4, 4, 4

F is a dense layer, C is a convolution layer, U is a fractionally-strided convolution, and 'N' is the number of different tiles in the target game. All convolution layers are used with Batchnorm. Stride is 2 for all convolution layers except for the final discriminator layers where stride is 1. For the generator networks, all convolution layers are used with ReLU activations. For the discriminator networks, all convolution layers are used with Leaky ReLU activations except for the final layer, which has no activation.

* Input is 32-bit noise + 5-bit condition vector reshaped into 1x1x37 tensor.

** Input to S1 is 32x32xN condition channels. Input to S2 is 32-bit noise reshaped into 1x1x32 tensor. Input to S3 is the output of S1 and S2 concatenated along the channel axis.

4.2. Latent Space Exploration

We search the space of levels encoded by the GAN with the *covariance matrix adaptation evolution strategy* (CMA-ES), as it has been demonstrated to be efficient for optimizing non-linear non-convex problems in the continuous domain without a priori domain knowledge.

4.2.1. Level Characteristics

We experiment with several fitness functions on the generated levels; most of them are based on static properties of the generated levels, such as the tile positions. For the purpose of clear demonstration, we consider two types of tile distributions that form two distinctive level characteristics. These static structures are observed in human-crafted levels shown in the below figures.

The two different characteristics are defined to be: one characteristic is where there exist several floating tiles in the frames with lesser ground tiles, and we define this type of tile positioning as “sky levels”. The other is named the “underground levels”, which is accurate to its name, where a layer of bricks on the top of the frames is observed, forming the ceiling of the underground look. Also, tiles are expected to be attached to the ceiling tiles, and more pipes are to be generated in the frames.

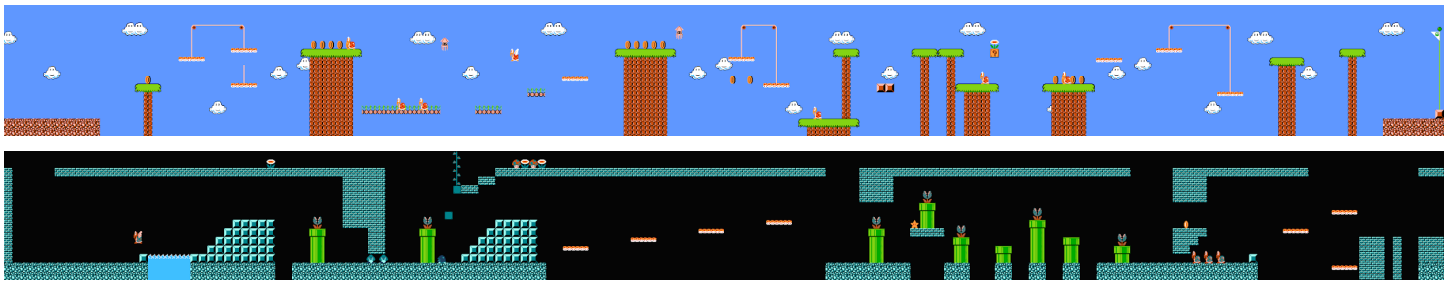


Fig 12. The top image is a human-crafted level that is the base of the “sky level” characteristic. The bottom image is the base for characterizing the “underground level”.

5. Experiment Designs

5.1. Conditional DCGAN

5.1.1. Conditional Feature Selection

This experiment explores how having the features from a previous frame as input to the generator influences the quality of the level. We employed two strategies for picking the input feature channels. First was the trivial case of inputting all the feature channels of the previous frame, ranging from ground tiles, brick tiles, pipe tiles, to enemy tiles, sky tiles, coin tiles, etc. For the second case, we handpicked the tiles that, according to us, depicted the structural aspect of the level, i.e., ground tiles, brick tiles and pipe tiles.

5.1.2. Slicing Ratio

We explored the different ratios$width_{conditional\ Frame}: width_{output\ Frame}$, for our conditional DCGAN architecture on Mario. The idea being, increasing the ratio towards conditional frames will, in essence, increase the information the $generator$ has for outputting a relatively smaller frame, hence improving the learning of short-term spatial dependencies over frames.

This experiment tested out two viable extremes of the ratios. The first where the $width_{conditional\ Frame}: width_{output\ Frame} = 1:1$, which for our implementation meant a width ratio of $14:14$. The other part of the experimented reduced the width of the output frame to 4, which made the ratio $24:4$.

5.2. Multi-Stage GAN

For the Multi-Stage GAN, one of our primary motivations was to preferentially generate levels with desirable structural properties. We tried different approaches to conditioning the input to the Stage-1 Generator in order to achieve this.

5.2.1. Using K-Bit Structure Vector as Condition

In this experiment, our condition is a k-bit structure vector. Each bit in the vector represents the presence or absence of a structural property in the game level. For our experiments, we use a 5-bit vector. The structural properties encoded in each bit are summarized in Table 4.

Table 4: Summary of structural properties encoded in condition vector

Bit Index*	Structural Property Encoded
1	Ground Fully Tiled?
2	Pipes Present?
3	Enemies Present?
4	Wall Structures (Pyramids, Blocks, etc.) Present?
5	Floating Tiles Present?

*Bit is 1 if statement is true, else 0

5.2.2. Using Previously Generated Frames as Condition

In this experiment, we combine our Conditional DCGAN architecture with our MSGAN architecture. The condition to the Stage-1 GAN is, therefore, some combination of the output from the Stage-2 GAN in previous iterations. We try out different $width_{condition} : width_{output}$ ratios; $1:1$ ratio wherein each iteration, we generate half of a level frame which is then passed in as condition to the Stage-1 GAN in the next iteration; $3:1$ ratio wherein each iteration, we generate ¼ th of a frame and where the condition to the Stage-1 GAN in each iteration is the output of the previous three iterations. In both cases, a seed image drawn from the training data is used as the condition for the first iteration(s).

5.3. Fitness Functions for Latent Space Exploration

In order to scale up the difficulty of levels as the player progresses through the levels, we will need to explore the latent space of the GAN. CMA-ES is an algorithm that has been used in the past in generating and evolving Mario levels to optimize for certain tile distributions. As described in the previous section, we optimize for static features in the level like sparse ground tiles, high enemy counts, or a greater number of tiles above a certain height.

5.3.1. Sky Level

We define the sky tiles as tiles that are floating three tiles spaces above the bottom of the frame. The number of these sky tiles is defined as the parameter α and, additionally, the number of sky tiles, which are positioned in the top half of the frame, is defined as β. The semi-overlapped definition for the sky tiles is to incentify the generator to search for noise that has more sky tiles that are not just floating but floating at least above half of the frame size. Finally, we include the count of ground tiles (positioned on the bottom of the frame) as γ to directly penalize them being generated. The full fitness equation is formalized as: $\alpha + \beta - \gamma$.

5.3.2. Underground Level

We define the number of ground tiles that belongs to the ceiling of the underground look as α, the number of ground tiles, positioned at the bottom of the frame, as β, and the number of enemies as γ. The fitness function is formulated as: $2\alpha - \frac{\beta}{2} + \gamma$. This maximizes the number of ground tiles that position on the top of the frame and penalizes them generating on the bottom of the frame. We also encourage the generation of enemies to increase the difficulty of the level.

5.3.3. Kullback-Leibler Divergence

Other than designing fitness functions manually, we also calculate the Kullback–Leibler (KL) divergence of the generated levels with respect to ground-truth levels as our fitness value. KL divergence is a measure of how similar two probability distributions are. Since our input is a text-based representation of a level and not a probability distribution, we will use a similar technique implemented in the 2019 paper by Lucas, Volz [8]. We will use a sliding window of size N x N and capture windows across our levels and ground-truth levels. Then we will use these distinct N x N squares and their occurrences as the probability distribution. The original paper uses a sliding window of size 4 x 4. However, it is possible that we can use a larger window since we have more computational power.

6. Results and Analysis

6.1. Baseline

We compare the generated results from our models with those of the baseline architecture in [1]. These results are presented in Fig. 13 and Fig. 14. We can see that our Conditional DCGAN model clearly generates levels with significantly lesser aberrations and more spatially coherent tile distributions. Our MSGAN results also show a marked improvement over our Conditional DCGAN model. We observe richer levels in terms of variation in structural patterns while maintaining minimal structural anomalies and smooth structural flow across frames which are trump points of our Conditional DCGAN architecture over the baseline.

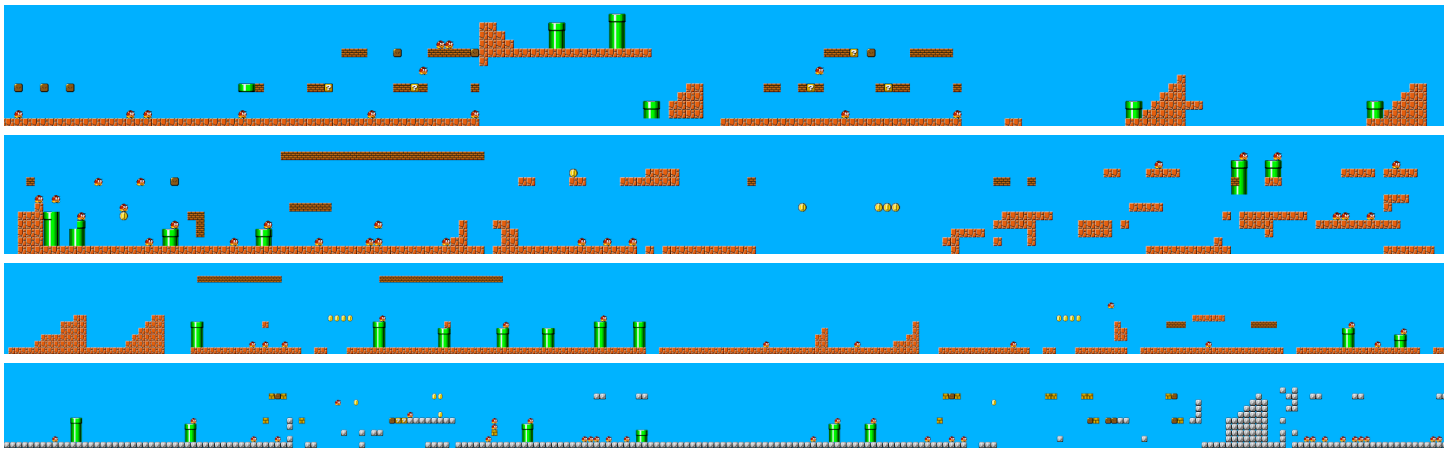


Fig. 13: From the top: 1) Examples of Super Mario Bros. levels generated by the baseline trained on the original single level dataset; 2) Examples of Super Mario Bros. levels generated by the baseline trained on the full level dataset; 3) Examples of Super Mario Bros. levels generated by our Conditional DCGAN

implementation; 4) Examples of Super Mario Bros. levels generated by our MSGAN (combined) architecture using $3{:}\,1$ ($width_{condition}$: $width_{output}$) ratio.

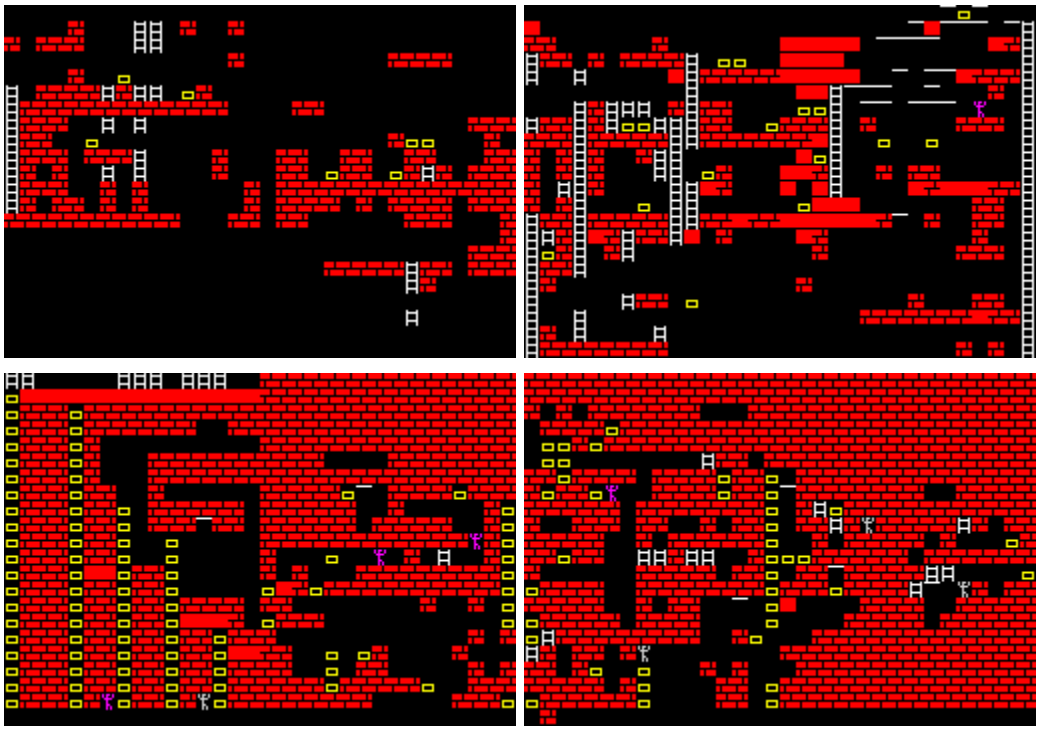


Fig. 14: Top row: Examples of Lode Runner levels generated by the baseline. Bottom row: Examples of Lode Runner levels generated by our Conditional DCGAN implementation

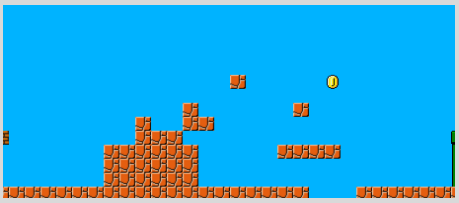
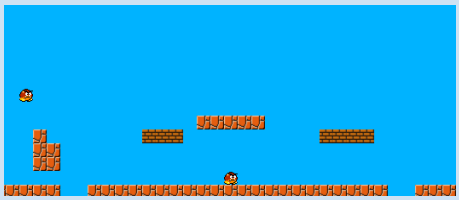
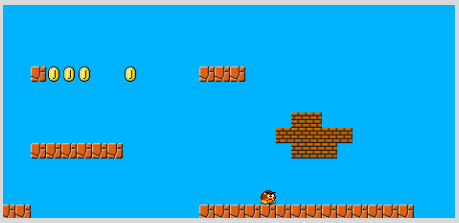
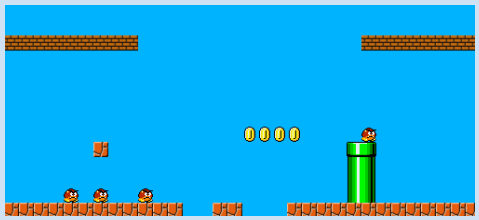
6.2. Conditional DCGAN

6.2.1. Conditional Feature Selection

As seen in the comparison shown in the image table below, handpicking the features that depict the structure of the frame produce more clean looking and continuous context flow across the level, whereas the output generated with all the feature channels passed in the input generate ragged-looking structures. Images in the bottom right image depict the flow of context in the top bricks, whereas, on the bottom left, the top is littered with ground tiles with a hunk of bricks.

One reason for this disparity between the structures might be that not removing features that are clearly inconsequential to the structure of the frame forces the generator to learn to ignore the corresponding feature channels. In contract, a pre-selection of features that define the structure of the generated levels removes this learning overhead from the generator.

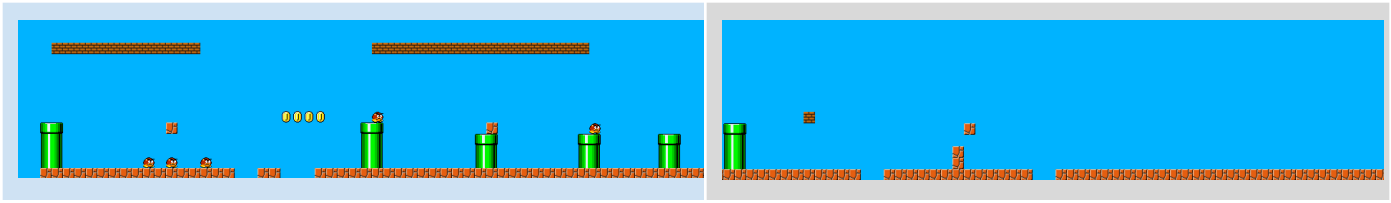
This discovery motivates us further to explore hand tagging the frames with relevant tags such as the level type -- sky, water and underground or with difficulty tags -- the count of enemies, jump height, etc., to control desired characteristics of the level.

Conditional Feature Selection	
All Feature Channels	Structural Feature Channels
	
	

6.2.2. Slicing Ratio

The experiments for different slicing ratios contradict our initial instinct--providing the additional information from the previous frame would increase the *goodness* of the generated level. What appears to have happened is that the reduced output frame width restricts the *expressive* power of the generator, and it basically ends up just generating ground tiles, as seen in the table below (right image). Keeping the ratio near 14:14 allows the generator to produce realistic levels while also providing enough contextual information from the previous frame.

Slicing Ratio $(width_{conditional\ Frame} : width_{output\ Frame})$	
14: 14	24: 4



6.3. Multi-Stage GAN

6.3.1. Using a K-Bit Structure Vector as Condition

We obtained mixed results using the condition vector as encoding. Our model could correctly decode specific properties (wall structures, absence of sky tiles) but ultimately fails to decode other properties (pipes, gaps in the ground, etc.). This model also leads to comparatively noisy generations. We concluded that using a condition vector is probably not the best approach to generating levels with desirable structural properties. It is not trivial for the network architecture to learn the mapping between the encoded bits and more subtle structural properties like gaps in the ground. Examples of noisy, unexpected results from the Stage-1 GAN using this architecture are presented in Fig. 15.

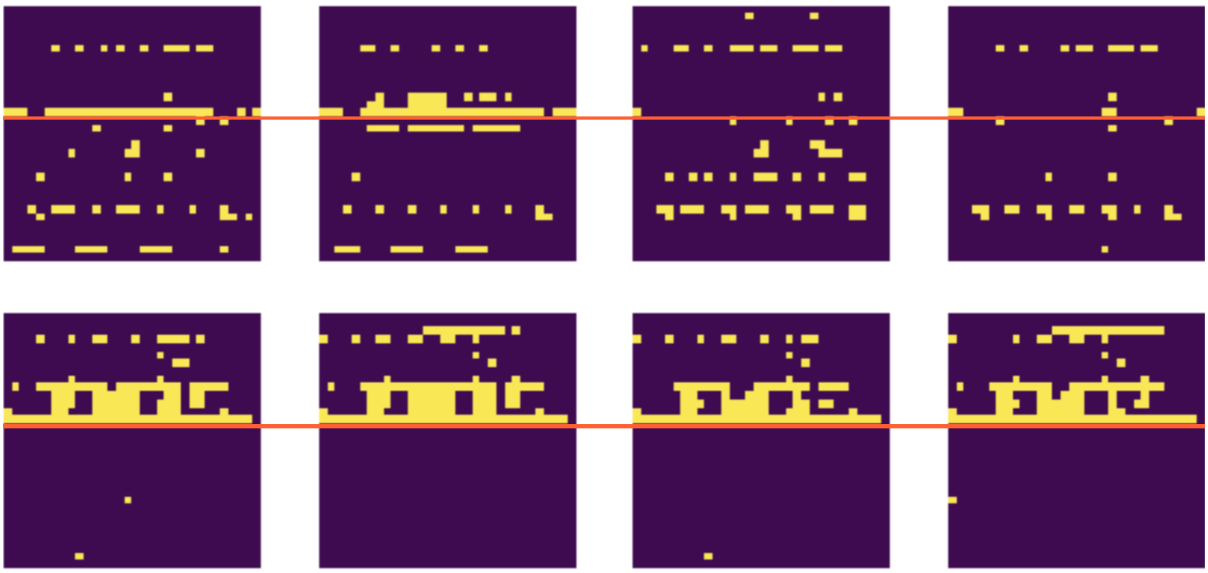


Fig. 15: Examples of frames generated by the Stage-1 GAN using condition vector as encoding. In the top row, condition is set to generate frames with pipes (failure). In the bottom row, condition is set to generate frames with wall structures (success). The red lines indicate the point from which padding starts. We can clearly see that the generated frames in the top row have significant noise (tiles being generated in the padding area).

6.3.2. Using Previously Generated Frames as Condition

Surprisingly, using a $1:1$ ($width_{condition}:width_{output}$) ratio did not work well for us. We found that the generator would often fall into a loop and generate the same frame over and over again. This is probably because our training data has several frames that are near symmetric along the Y-axis. The generator learns an identity mapping for conditions generated from such frames. Then, during generation, if a frame similar to one of these identity conditions is generated, the model keeps generating the same frame repeatedly. Fig. 16 presents an example level generated using this approach. Using a $3:1$ ($width_{condition}:width_{output}$) ratio broke the symmetry and solved the issue for us. This approach led to our best performing architecture. An example level generated using this architecture is presented in Fig. 13.

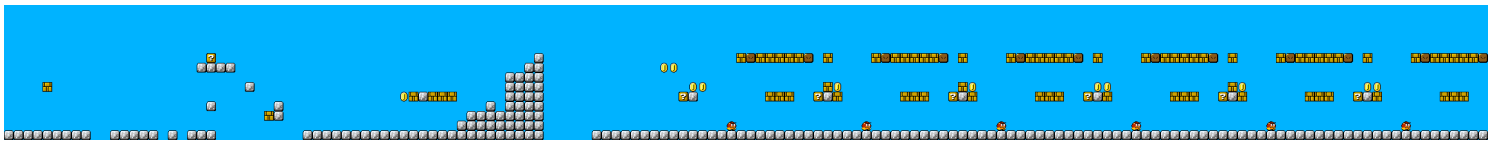


Fig. 16: Repeated pattern generation on using $1:1$ ($width_{condition}:width_{output}$) ratio.

6.3.3. Seed Image Condition as a Feature Selector

The design style of an image is a high-level feature that cannot be easily quantified. We have seen how conditioning our generations on previously generated frames can yield lesser structural anomalies and better structural flow across frames. This approach also has the added advantage of allowing us to condition our generations on the design style of seed images. From our experiments, we observed that the generated levels adopt the design style of the seed image. This is illustrated in Fig. 17. We can therefore generate different segments with different seed images and stitch them together to obtain a level with evolving design properties.

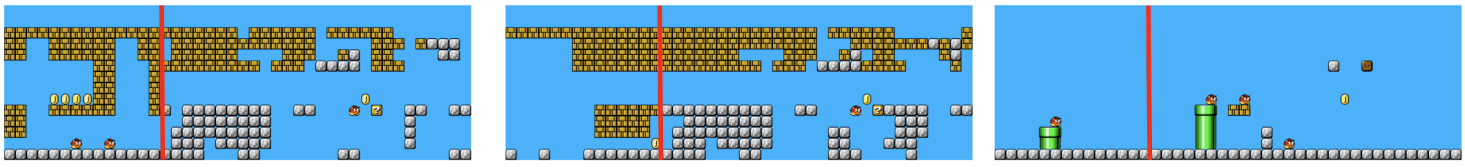


Fig. 17: Examples of Seed Image as a Feature Selector. For each image, on the left of the red line is the seed image and on the right are a couple of generated frames.

6.4. Latent Space Exploration

With the designed fitness functions for the two different tile distribution styles, we run CMA-ES with the standard deviation as 0.5 and the population size as 1000. For each run, we simulate five different conditional frames for generating different new frames (based on the design of our conditional DCGAN) for each population. The evaluation for each run is based on the mean of the fitness values of the five generated frames. As we are aiming to find the best solution for a 14 x 14 dimensional space, we derive the best solutions by keeping track of the results throughout running for 1000 iterations.

6.4.1. Results of hand-crafted fitness functions

The top two best solutions (two sizes 14 x 14 noise vectors) for the sky levels are selected for our level generation. The generated frames contain several floating tiles, with the ground tiles eliminated.

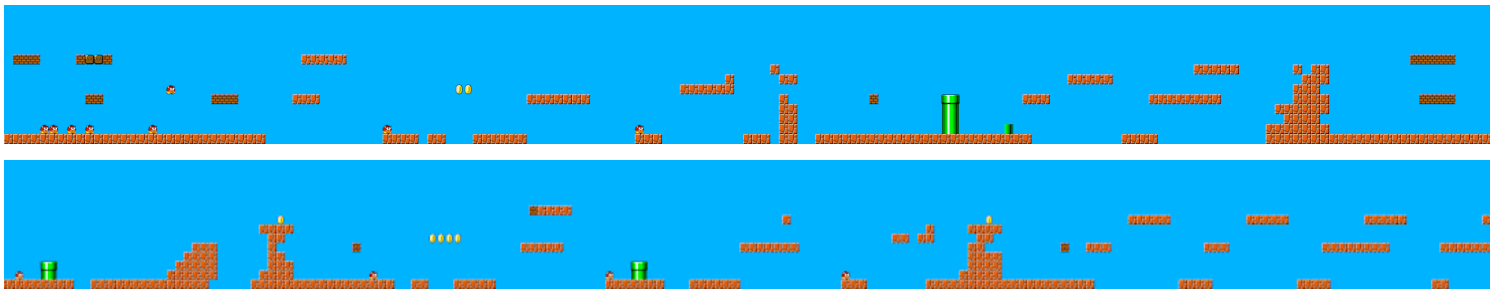


Fig. 18: Sky levels generated with the top noise vector explored by CMA-ES.

We also selected the top two noise vectors for generating a series of underground characterized frames for our underground game level. In the frames, we can see that the ceiling of the underground look is successfully generated along with the relatively increased amount of pipes seen.

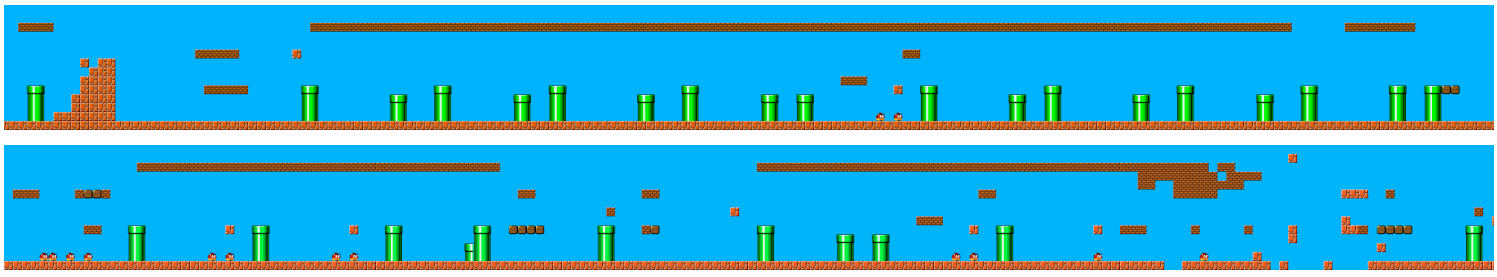


Fig. 19: Underground levels generated with the top noise vector explored by CMA-ES.

6.4.2. Results of Kullback–Leibler divergence based fitness function

We also experimented with fitness values that are created by calculating the KL divergence between the human-crafted sky level and the generated level. The

top image in the figure below is the human-designed sky level that we measure the KL distance with, and the generated level based on the found noise vector is shown in the bottom image of the figure below.

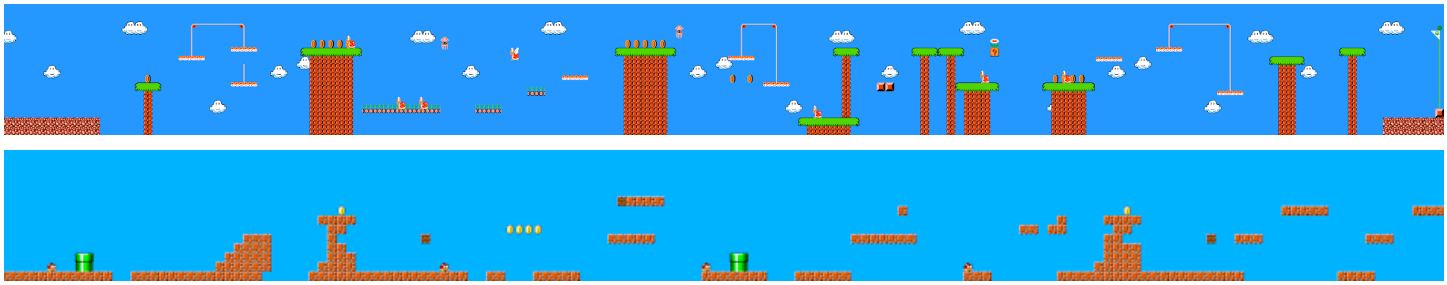


Fig. 20: Top: hand-crafted sky level; Bottom: sky level generated via noise vector found by CMA-ES using KL divergence as its evaluation metric.

6.4.3. Game Level Feature Orchestration

After performing LSE to map desired features with a set of noise parameters, we then experimented with dynamically changing the features while generating the level. As a result, a level can begin as an underground level and then change into a sky-based level or some frames with completely randomized features, as done originally. This setup demonstrates that our generator is capable of producing coherent levels whilst changing the structural theme of the level. The orchestration menu, where the user can select the theme of the generated frames, is shown in Fig. 21. A level that is generated by rotating the selections on the orchestration menu is displayed in Fig. 22.

Fig. 21: Orchestration menu with the option of creating sky level, underground level and random frames.

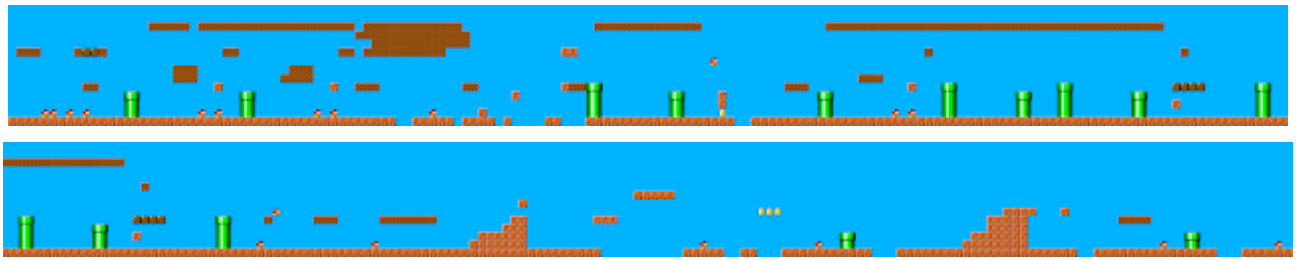


Fig. 22: Dynamic level (consists of underground frames, to sky frames, and to random noise frames) generated with the top noise members explored by CMA-ES.

7. Conclusion and Future Work

7.1. Pre-Midterm Progress Conclusion

We have shown some preliminary results that support our assumptions made before implementing the different GAN architectures. First of all, our conditional DCGAN was able to leverage the additional contextual information provided by the handpicked structural tiles of the previously generated frames and generate more natural-looking levels. The Multi-Stage GAN does not perform well in its current implementation, but we expect it to work significantly better with minor architectural improvements. As for our measurements, we have marked some action features performed in the games as our evaluation metrics and implemented a preliminary version with the results shown in the previous sections. With these results, we are now ready to explore in the direction of creating a game level that also contains desired attributes.

Our post-midterm plan is to further modify the GAN architectures that we have been working on before the midterm and get started with the implementation of the latent space exploration and other control methods. We will then work to integrate the generation control methods with the GAN architectures that we proposed. On top of this, we will be reviewing our evaluation metrics in order to measure the quality of our studied latent space exploration algorithms.

7.2. Post-Midterm Discussion and Future Work

During the latent space exploration, it is observed that the first one to two frames generated is generally more representative of the fitness functions. As more frames are generated, we can see that not all generated frames fit the desired feature description. The observed result may appear due to several different aspects, one of which we suspect is the limitation of the conditional DCGAN architecture. Our GAN is trained to generate frames by conditioning on only half a human-created game level frame. This design is intended to honor the tile structures and also preserve the natural stitching for side-scrolling games. However, the latent space search is evaluated based on the entire generated level, which may pose a gap between the amount of information used for training the GAN and searching for the noise vector. Thus, it is also valuable to explore training GAN with different amounts of conditional frames for future improvements. We also discussed how the seed image could be used as a high-level feature selector in

Conditional GAN architectures. We can generate different segments with different seed images and stitch them together to obtain evolving design properties. One problem is that the transition might not be smooth when we stitch these segments together. Training a model to interpolate frames between these segments could be one solution, which can also be considered a direction for potential future improvements.

The evaluation metrics used for evolving levels in this project can also be further explored in future work. This work mainly focuses on using static structure to compute fitness values; however, the correlation between these features and the difficulty of a level is only anecdotal. The other option for evaluating a difficulty fitness function is to determine metrics based on the performance of an agent at a certain level. This approach resolves the issue with low-impact metrics and generates metrics directly related to difficulty (e.g., time spent on level/level horizontal length). With evaluation metrics related to the performance of an agent, it is possible to generate more challenging levels that are even more tailored to players, which is an interesting next step for future work.

7.3. Post-Midterm Conclusion

We have approached our ultimate goal of auto-generating game levels with specific attributes by carrying out our post-midterm plans. Specifically characterized game level auto-generation is enabled via CMA-ES to find noise vectors that become inputs of the Conditional DCGAN. The results have shown that this project provides a rich and still natural looking game level. On top of that, when combined with Conditional DCGAN architecture, the Multi-Stage GAN architecture generates more loaded levels in terms of variation in structural patterns while maintaining its original advantages over the baseline.

8. References

[1] Volz, Vanessa, Jacob Schrum, Jialin Liu, Simon M. Lucas, Adam Smith, and Sebastian Risi. "Evolving mario levels in the latent space of a deep convolutional generative adversarial network." In *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 221-228. 2018.

[2] Togelius, Julian, Sergey Karakovskiy, and Robin Baumgarten. "The 2009 mario ai competition." In IEEE Congress on Evolutionary Computation, pp. 1-8. IEEE, 2010.

[3] Torrado, Ruben Rodriguez, Philip Bontrager, Julian Togelius, Jialin Liu, and Diego Perez-Liebana. "Deep reinforcement learning for general video game ai." In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 1-8. IEEE, 2018.

[4] Summerville, Adam James, Sam Snodgrass, Michael Mateas, and Santiago Ontanón. "The vglc: The video game level corpus." *arXiv preprint arXiv:1606.07487* (2016).

[5] Goodfellow, Ian, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. "Generative adversarial nets." In *Advances in neural information processing systems*, pp. 2672-2680. 2014.

[6] Schrum, Jacob, Jake Gutierrez, Vanessa Volz, Jialin Liu, Simon Lucas, and Sebastian Risi. "Interactive evolution and exploration within latent level-design space of generative adversarial networks." *arXiv preprint arXiv:2004.00151* (2020).

[7] Fontaine, Matthew C., Ruilin Liu, Julian Togelius, Amy K. Hoover, and Stefanos Nikolaidis. "Illuminating Mario Scenes in the Latent Space of a Generative Adversarial Network." *arXiv preprint arXiv:2007.05674* (2020).

[8] Lucas, Simon M., and Vanessa Volz. “Tile Pattern KL-Divergence for Analysing and Evolving Game Levels.” *Proceedings of the Genetic and Evolutionary Computation Conference*, 2019.

[9] Giacomello, Edoardo, et al. *DOOM Level Generation Using Generative Adversarial Networks*. 24 Apr. 2018.

[10] Wikipedia contributors, "Lode Runner," Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/w/index.php?title=Lode_Runner&oldid=982792890 (accessed October 11, 2020).

[11] Martin Arjovsky, Soumith Chintala, and Léon Bottou. 2017. Wasserstein Generative Adversarial Networks. In Proceedings of the 34nd International Conference on Machine Learning, ICML.

[12] Odena, Augustus. "Semi-supervised learning with generative adversarial networks." arXiv preprint arXiv:1606.01583 (2016).

[13] Mirza, Mehdi, and Simon Osindero. "Conditional generative adversarial nets." arXiv preprint arXiv:1411.1784 (2014).

[14] Radford, Alec, Luke Metz, and Soumith Chintala. "Unsupervised representation learning with deep convolutional generative adversarial networks." arXiv preprint arXiv:1511.06434 (2015).

[15] Zhang, Han, Ian Goodfellow, Dimitris Metaxas, and Augustus Odena. "Self-attention generative adversarial networks." In International Conference on Machine Learning, pp. 7354-7363. PMLR, 2019.