GAN-Based Game Level Generation

Sourish Banerjee[§]
sourishb@usc.edu
ID: 6316727744

David Chen[§]
chen749@usc.edu
ID: 1722561951

Ya-Chuan Hsu[§]
yachuanh@usc.edu
ID: 2705050725

Ujjwal Puri[§]
ujjwalpu@usc.edu
ID: 2176413563

***Abstract*— We propose a Conditional Deep Convolutional Generative Adversarial Network architecture, which feeds back the generated level frame to the next iteration. This enables the Generative Adversarial Network (GAN) to output long-running coherent levels with no apparent stitch boundaries, not commonly seen in previous work. The covariance matrix adaptation evolution strategy is implemented for exploring noise vectors to have GAN output specified structure features. Finally, a Multi-Stage GAN architecture is introduced and combined with our Conditional GAN for generating complex game structures. Our experiment results have shown to create levels with rich structures and natural-looking frames.**

I. Introduction

Our objective for this project is to generate game levels close to human-like design, while being able to control various attributes in the generated levels such as number of enemies, difficulty of jumps, or the theme of the level.

It is well known how essential level design is to a game; however, this process consumes a major chunk of man hours. Moreover, nowadays, the longevity of a video game is strongly dependent on the frequency and quality of content updates. With this motivation – to develop a method that can reduce level generation costs and, at the same time, maintain the quality of the generated game content, we propose to leverage generative adversarial networks (GANs) to create human-like game levels. For controlling the attributes of the generated game content, we will be exploring latent space search methods as well.

Our approach, generating games via GANs, is inspired by recent research [1] in game level generation. GAN is a machine learning method that uses two neural networks– generative network and discriminative network, to accomplish unsupervised generative model learning. It is mostly seen implemented for various purposes related to image processing, such as image classification, synthesis and upscaling. Volz et al [1] applied GANs on procedural content generation (PCG) with an additional enhancement on the generated quality by including the Covariance Matrix Adaption Evolution Strategy (CMA-ES).

Whilst Volz et al [1] were able to generate reasonable game levels, we observed that the levels contained sections with mismatched tiles and broken structures that were easily spotted. Some tile distributions were also seen to be too systematically laid out without variations. Hence, we would like to contribute the following via this project:

- *Stitching generated frames:* a conditional DCGAN architecture where we include previous generated frame as input to the generator such that generated frames stitch smoothly together.
- *Generate complex game maps:* a two-stage GAN architecture is proposed for generating both the structure and the details for a complex game map.
- *Explore and compare latent space search:* provide a comparison of various latent space search methods applied on game level generation.
- *Evaluation metrics:* since it is challenging to measure the quality of a generated game level, we explore different measurement methods for the games evaluated in this project.

II. Overview of Games

We aim to evaluate two types of games. One is a side-scrolling game where we can iteratively generate level frames, and the other is a complex game level that can be divided into the game skeleton and game details. Therefore, our choice of games, respectively, are the Super Mario Bros. and the Lode Runner.

A. Super Mario Bros.

Super Mario Bros. is a side-scrolling 2D platform game released by Nintendo. The player can run and jump across platforms and atop enemies in themed levels (as shown in Fig. 1).

In 2009, a Mario AI competition [2] was created and associated with the IEEE Games Innovation Conference and the IEEE Symposium of Computational Intelligence and Games. The initial purpose of the competition was to develop bots that were able to play Super Mario Bros. optimally. With the success of the competition debut, the organization included two other tracks the following year. One of three is a level generation track. This paper leverages the competition resources, such as game level database and play-through of generated levels.

B. Lode Runner

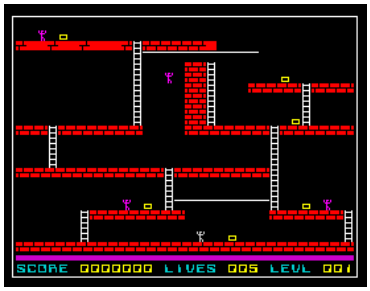
Lode Runner is created by Doug Smith and published by Broderbund in 1983 [3]. The player's objective is to collect gold pieces in a 2D puzzle-like map and reach the exit (at the top of the map) without being chased down by guards (see Fig. 2).

The game starts with the player having five lives and awards an extra life once a level is completed. If a guard

[§]Equal contribution

Fig. 1: A level example from Super Mario Bros.

catches the player, then one life is subtracted, and the current level restarts. Players can use ladders and suspended hand-to-hand bars to move around different sections in the game. There is also a wall/floor tile digging feature that allows the player to create holes as traps for or passages around guards.



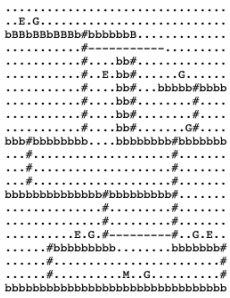


Fig. 2: Representation of the Lode Runner game map. Left: the white stick figure is the player, and the enemies are shown as pink stick figures. Gold pieces are drawn as gold blocks. Right: a 2D string representation of the Lode Runner game level.

C. Video Game Level Corpus

The Video Game Level Corpus (VGLC) [4] contains various games with multiple game levels. Both Super Mario Bros. and Lode Runner are included in VGLC. As the games are tile-based, the game corpus provided by VGLC represents levels as 2D strings where particular character symbols represent different tile types (see Fig. 2). For example, 'b' represents a brick tile and 'e' stands for an enemy. Thus, we input a game level corpus as a 2D array to our generative models.

III. Related work

A. Procedural Content Generation (PCG)

Procedural Content Generation can be thought of as an algorithmic method for generating assets on the fly. PCG has been used in games to generate whole levels (Rogue, Elite)[1][2], graphical assets (No Man's Sky, Star Citizen)[3], NPC models etc. since 1980s. Apart from the obvious savings in man-hours, PCG provides a way to add a large variety to the game assets, help save memory requirements leveraging on the fly generation, and adds an element of discovery to the content. This massive space of potential content is bound to contain assets that are straight-up broken and unplayable. Difficulty in controlling the generation is the biggest negative of relying solely on PCG. There has been previous work [1], [5], [6] done on exploring the latent space of Generative Adversarial Networks (GANs) to generate outputs with desired features.

[1]https://en.wikipedia.org/wiki/Roguelike
[2]https://en.wikipedia.org/wiki/Elite_(video_game)
[3]https://robertsspaceindustries.com/comm-link/transmission/15560-Procedural-Planets-V2-Demo

B. Generative Adversarial Networks

Generative Adversarial Networks is a generative model proposed by Goodfellow et al. [7]. A typical GAN architecture contains two sections 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 which seeks to create images that fools the discriminator. The generator takes a randomized input distribution $\mathbf{z} \sim p_{\mathbf{z}}$, 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 for the joint loss function for D and G,

$$\min_G \max_D \mathbb{E}_{\mathbf{x}\sim p_r} \log[D(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}} \log[1 - D(G(\mathbf{z}))] \quad (1)$$

Several types of GAN architecture-variants can be found in the literature. The original GAN uses a fully-connected neural network for both discriminator and generator. However, it does not generalize well on more complex images. Thus, different improvement methods were explored, such as including supervision learning, Semi-supervised GAN [8]; feeding class labels to both G and D networks and enhance discriminative abilities for D, Conditional GAN [9]; applying deconvolutional neural networks to G, Deep Convolutional GAN [10]; ensuring the large receptive field without sacrificing computational efficiency via the self-attention mechanism, Self-Attention GAN [11]; and up-scaling GAN training, Big GAN [12].

C. Generate Game Levels with Desired Attributes

To generate game levels to not only be playable but also exciting and challenging is a field of research that explores approaches for encoding specific attributes into generated game levels. Some researchers propose to hierarchically generate the game structure and distribution of the content details via evolutionary algorithms [13] and autoencoders [14].

In this project, we are particularly interested in constraining generated game levels to express certain desired attributes by controlling the noise input z while training our GANs. Through controlling the z, Bonteger et al. [15], [16] introduced the idea of using a Latent Variable Evolution (LVE) approach as to search for latent vectors via Covariance Matrix Adaption Evolution Strategy (CMA-ES) that optimizes for certain fitness functions – here desired attributes in the level. The LVE method was used for the desired game level content generation by Volz et al. [1]. Volz et al. where they divided the generation process into two distinct phases.

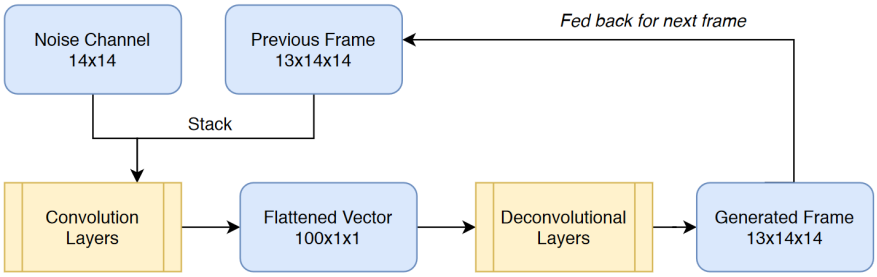


Fig. 3: Conditional DCGAN generator architecture. The generated frame is fed back as input for the next iteration.

The first phase is to produce a trained generator (G), and the second phase searches for latent vectors as input to G to have specific properties included in the game levels.

IV. Methodology

We extend DCGAN to improve frame stitching in the generated side-scrolling game frames. As for capturing interesting game content details, a two-layer GAN model is proposed and analyzed. Additionally, we explore the latent space for interesting game level structures and examine different evaluation metrics. The final goal is to form a comparison between the baseline methods [1] and our implemented improvements. The architecture details of the baseline, as well as all our implemented models are outlined in Table I.

A. GAN Architectures: Conditional DCGAN

As the input to a vanilla DCGAN generator is a random noise vector, the outputted consecutive frames show no structural theme flow. The idea of appending the features of the previous frame to the generator input enables the GAN to learn the dependencies between frames.

Some potential ways of passing the previous frame's feature as a condition to the generator are: 1) passing the whole frame as is, composed of all feature channels without minimal pre-processing; 2) handpicking some of the feature channels determined to define the structural aspect of the frame, e.g., a channel that contains the ground bricks; and 3) passing the previous frame through Convolutional layers as a way to extract a distilled representation of the features and pass that to the generator. We found that a mix of strategy 2), picking stable features – ground, pipes, breakable bricks, and 3) worked the best.

The channels get passed through multiple Convolutional layers without any max-pooling, which is then flattened into a 100x1x1 vector for the DCGAN generator. The discriminator's input is the condition frame stitched with the generated output. The motivation behind stitching the frames is to make the discriminator learn to distinguish between the flow of levels made by humans versus generators.

B. GAN Architectures: Multi-Stage GAN

The motivating factors for implementing a multi-stage GAN architecture are twofold: 1) break up the complex task of hallucinating game levels into more straightforward tasks for individual GANs to tackle, and 2) condition the GAN input on structural properties during training to preferentially generate levels with desired structure. We train two GAN architectures successively to achieve this. The training framework is illustrated in Fig. 4.

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 $\mathcal{N}(0,1)$ 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 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.

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

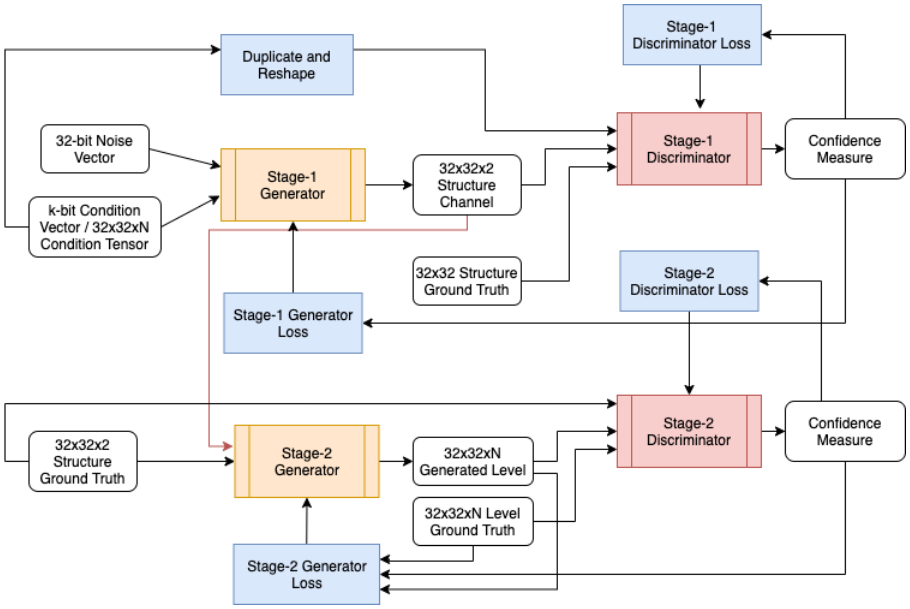


Fig. 4: Training workflow for the Multi-Stage GAN architecture. The red arrow shows that the output from the Stage-1 Generator is passed as condition to the Stage-2 Generator for the game-level generation after training is completed for both stages.

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 $\mathcal{N}(0,1)$ 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.

C. Evolving Levels

One of the goals of using GANs to generate game levels is to scale up the level difficulty as the player progresses. To do so, we need to explore the latent space of the generator, producing levels that have certain features such as high jump length/count, high enemy count and sparse ground tiles. Covariance Matrix Adaption Evolution Strategy (CMA-ES) is an evolutionary algorithm that we have chosen to use for latent space exploration. An open-source Python implementation of CMA-ES is incorporated together with custom fitness functions based on static features.

Static features are features that can be determined from the level representation alone. Thus, it makes the generation pipeline computationally efficient since levels can be evaluated without having agents to play the game. In this paper, we consider two different types of level features: the *sky level* and the *underground level*. The prior level consists mostly of floating tiles; the latter consists of tiles distributed on top of the frame as the ceiling (akin to the underground levels in mario) along with relatively more pipes.

1) Hand-craft fitness functions: For each level feature, we define a fitness function that captures the essence of the featured level.

$$f_{sky} = count_{\text{floating tiles}} + count_{\text{high floating tiles}} - count_{\text{ground tiles}}$$

$$f_{under} = 2 \cdot count_{\text{ceiling tiles}} - count_{\text{ground tiles}} \div 2 + count_{\text{enemies}}$$

[4]Input is 32-bit noise + 5-bit condition vector reshaped into 1x1x37 tensor. Input to S1 is 32x32xN condition channels.

[5]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.

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/Nodes (F-Layers)	256U [4], 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/Nodes (F-Layers)	S1 [5]: 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

TABLE I: GAN Architecture Details. 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.

Sky level fitness function f_{sky} is computed with high rewards for floating tiles and for ones that are positioned above half of the frame. The underground level fitness function f_{under}, focuses on creating ceiling tiles and add enemies to increase level difficulty. Both functions include a penalty for creating ground tiles.

2) Kullback–Leibler divergence: Besides manually crafting fitness functions, we also explore calculating the Kullback–Leibler (KL) divergence for comparing the generated levels with the ground-truth levels. 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 use a similar technique seen in [17]. With a sliding window of size N x N and capture windows across our levels and ground-truth levels, we compute these distinct N x N squares and their occurrences as the probability distribution.

V. Experiments and results

In this section, we first compare our proposed GAN architectures with the results in [1]. Next, a detailed analysis is conducted on the two GAN architectures by testing different parameter settings. Finally, the latent space is explored via CMA-ES, and sets of noise vectors are mapped to structural features. These vectors are then used to create levels with specified attributes.

A. Baseline

We compare the generated results from our models with those of the baseline architecture in [1]. These results are presented in Fig. 5 and Fig. 6. 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.

B. Conditional DCGAN

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 – ground tiles, brick tiles, pipe tiles, 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.

As seen in the comparison shown in the figs 7a and 7b, handpicking the features that depict the structure of the frame produce cleaner 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 and hunk of bricks.

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

This discovery motivates us further to explore hand tag-

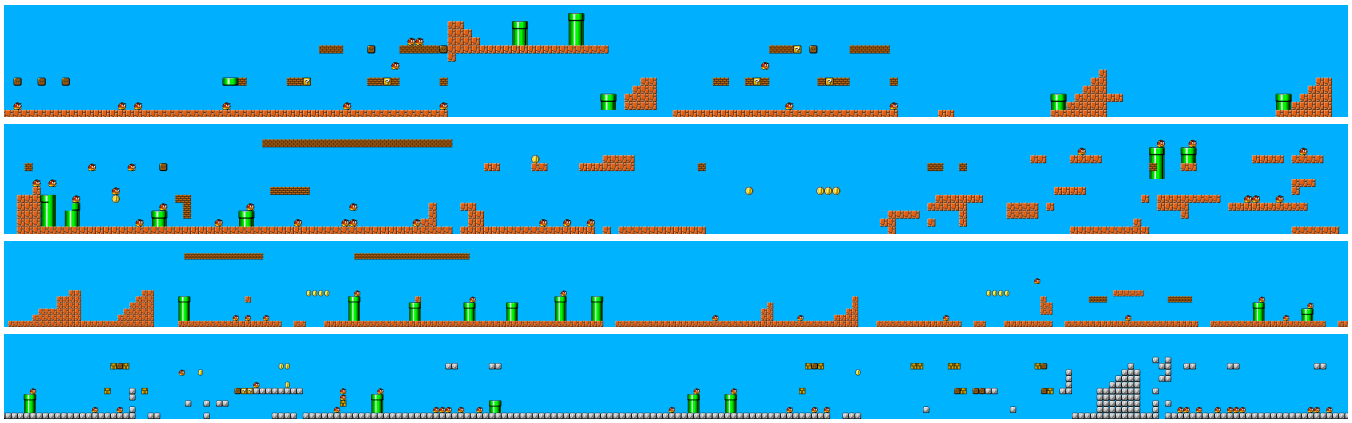


Fig. 5: From the Top: 1) Examples of Super Mario Bros. levels generated by the baseline trained on the original single level dataset; 2) Examples of Super Mario Bros. levels generated by the baseline trained on the full level dataset; 3) Examples of Super Mario Bros. levels generated by our Conditional DCGAN implementation; 4) Examples of Super Mario Bros. levels generated by our MSGAN (Combined) architecture using $width_{condition} : width_{output} = 3 : 1$

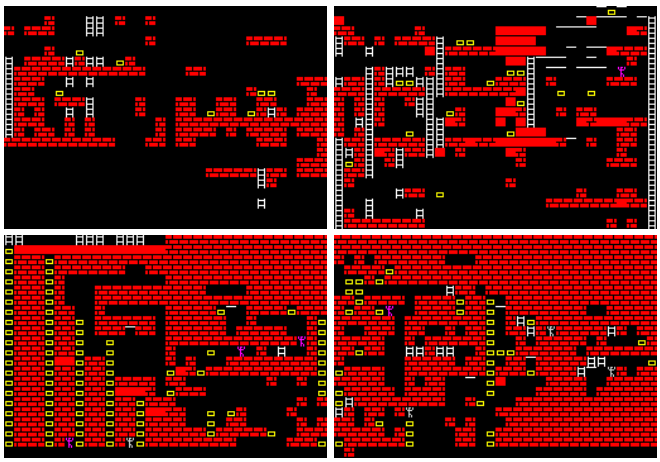
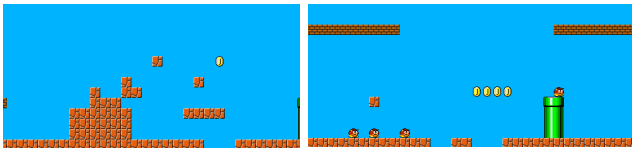


Fig. 6: Top: examples of Lode Runner levels generated by the baseline; Bottom: examples of Lode Runner levels generated by our Conditional DCGAN implementation.

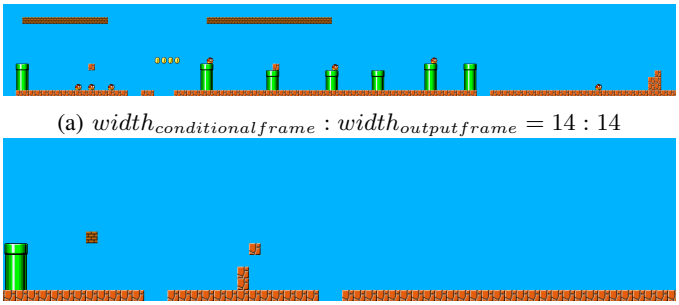
ging the frames with relevant tags such as the level type – sky, water and underground or with difficulty tags – the count of enemies, jump height, etc. to control the desired characteristics of the level.



(a) (b)

Fig. 7: (a) A generated level with all feature channels passed as conditions to the generator. (b) Result of only passing structural feature channels as conditions to the generator.

2) Slicing Ratio: We explored the different ratios– $width_{conditionalframe} : width_{outputframe}$, 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.



(a) $width_{conditionalframe} : width_{outputframe} = 14 : 14$

(b) $width_{conditionalframe} : width_{outputframe} = 24 : 4$

Fig. 8: The difference between the generated levels is quite stark. With an even ratio, the levels are more natural-looking, while the lopsided ratio of 24:4 generates uninteresting levels with only ground tiles.

This experiment tested out two viable extremes of the ratios. The first where the ratio $width_{conditionalframe} : width_{outputframe} = 1 : 1$, which for our implementation meant a tile ratio of $14 : 14$. The other part of the experiment is to reduce the width of the output frame to 4, which made the tile ratio $24 : 4$.

The experiments for different slicing ratios contradicts 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 figures 8a and 8b. Keeping the ratio near $14 : 14$ allows the generator to produce realistic levels while also providing enough contextual information from the previous frame.

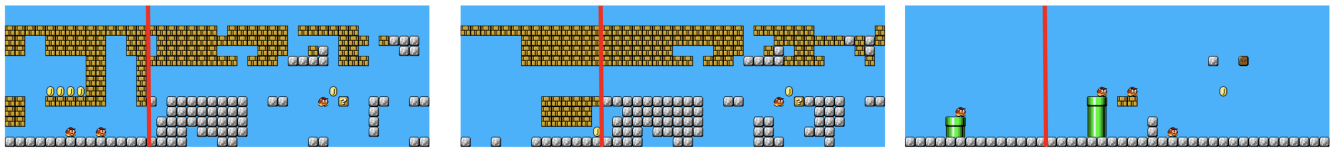


Fig. 9: Examples of Seed Image as a Feature Selector. For each image, on the left of the red line is the seed image and on the right are a couple of generated frames.

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

TABLE II: Summary of structural properties encoded in condition vector (Bit is 1 if statement is true, else 0).

C. 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 for conditioning the input to the Stage-1 Generator in order to achieve this.

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 II.

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 as 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 is presented in Fig. 10. In the top row, the condition is set to generate frames with pipes (failure). In the bottom row, the 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).

2) Using Previously Generated Frames as Condition: In this experiment, we combine our Conditional DCGAN architecture with our MSGAN architecture. Therefore, the condition to the Stage-1 GAN is 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 $1/4^{th}$ of a frame and where the condition to the Stage-1 GAN in each iteration is the output of the previous three iterations. A seed image drawn from the training data is used as the condition for the first iteration(s) in both cases.

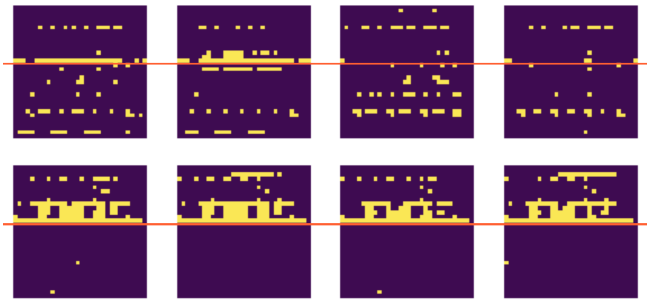


Fig. 10: Examples of frames generated by the Stage-1 GAN using condition vector as encoding.

Surprisingly, using $width_{condition} : width_{output} = 1:1$ did not work well for our MSGAN. We found that the generator would often fall into a loop and generate the same frame repeatedly. This is probably because our train 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. 11 presents an example level generated using this approach. Using $width_{condition} : width_{output} = 3:1$ 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. 5.

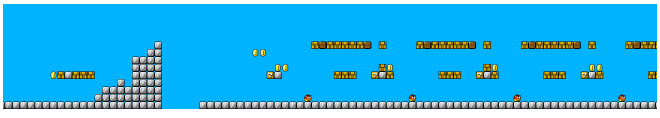


Fig. 11: Repeated Pattern Generation on Using $width_{condition} : width_{output} = 1:1$

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 conditioning our generations on the design style of seed images. From our experiments, we observed that the generated levels adopt the

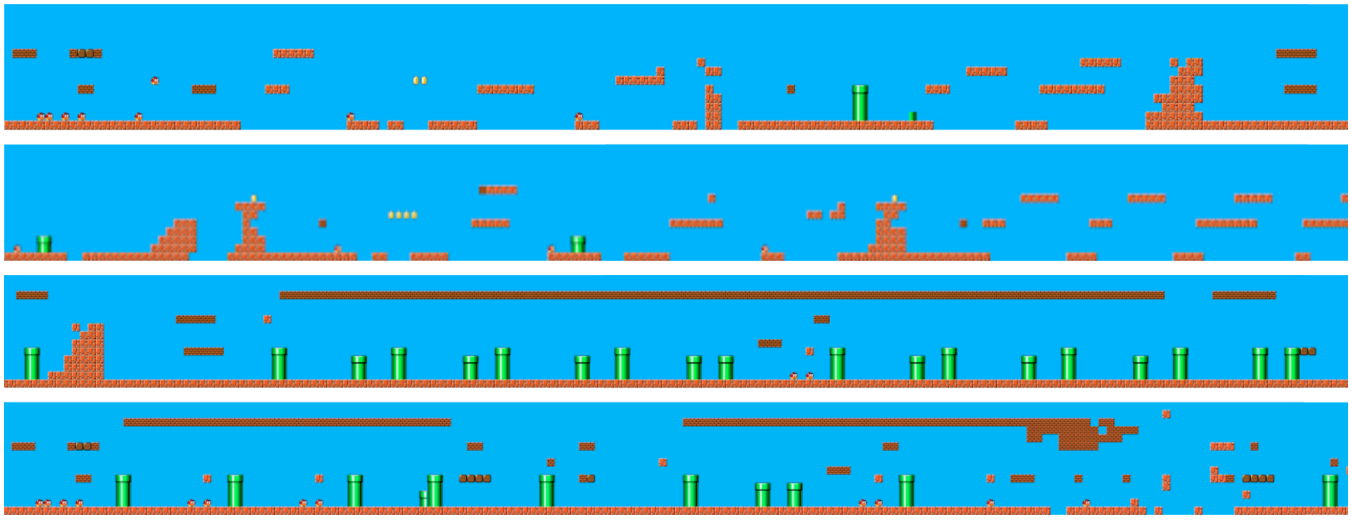


Fig. 12: With designed fitness functions used for finding noise vectors, we results in generated sky levels shown in the top two images and underground levels shown in the bottom two images.

design style of the seed image. This is illustrated in Fig. 9. Therefore, we can generate different segments with different seed images and stitch them together to obtain a level with evolving design properties.

D. Latent Space Exploration

We setup CMA-ES with the standard deviation as 0.5 and the population size as 1000. For each population, we simulated five different conditional frames and evaluated them by the mean of the fitness values of the generated frames. As we aimed to find the best solution for a 14×14 dimensional space, we kept track of the elites throughout the 1000 iterations.

The results of sky levels generated via searching with hand-crafted fitness functions are shown in Fig. 12. We can observe that floating tiles have largely increased and ground tiles have decreased. As for the generated underground levels (also see Fig. 12), the ceiling of the underground look is successfully generated along with the relatively increased amount of pipes seen.

Additionally, we examine sky levels generated by noise vectors that are found based on calculating the KL divergence as the fitness values while conducting the latent space exploration. In Fig. 13, the tile distribution represents both the floating and the ”island” tile positioning.

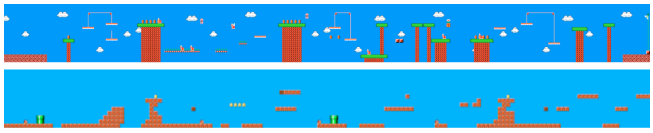


Fig. 13: Top: hand-crafted sky level; Bottom: sky level generated via noise vector found by CMA-ES using KL divergence as its evaluation metric.

VI. Conclusion and Future Work

Our ultimate goal, creating a pipeline for auto-generating game levels with specific attributes, is achieved in this work. We have proposed a Conditional DCGAN architecture to create natural-looking game levels for side-scrolling games. With CMA-ES finding noise vectors as inputs to our Conditional DCGAN, we have enabled the game level generator to characterize generated frames. The results have shown that this project provides the capability of creating rich yet natural-looking game levels. On top of that, when the Multi-Stage GAN architecture is combined with the Conditional DCGAN architecture, it generates more loaded levels in terms of variation in structural patterns while maintaining its original advantages over the original paper [1] that inspired us to start this project.

There are several potential directions for future work. For example, explore training GAN with different amounts of conditional frames for future improvements. Our GAN is trained to generate frames by conditioning on half a human-created game level frame, and the latent space search is evaluated based on the entire generated level. This forms a gap between the amount of information used for training the GAN and searching for the noise vector. Additionally, the evaluation metrics used for evolving levels in this project can also be further explored in future work by considering agent performances during the evaluation process. A more tailored game level can be generated once the player performance is considered.

References

[1] V. Volz, J. Schrum, J. Liu, S. M. Lucas, A. Smith, and S. Risi, “Evolving mario levels in the latent space of a deep convolutional generative adversarial network,” in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2018, pp. 221–228.

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

[3] Wikipedia contributors, "Lode runner," https://en.wikipedia.org/w/index.php?title=Lode_Runner&oldid=971631626, 2020, [Online; accessed 8-October-2020].

[4] A. J. Summerville, S. Snodgrass, M. Mateas, and S. Ontanón, "The vglc: The video game level corpus," *arXiv preprint arXiv:1606.07487*, 2016.

[5] M. C. Fontaine, R. Liu, J. Togelius, A. K. Hoover, and S. Nikolaidis, "Illuminating mario scenes in the latent space of a generative adversarial network," *arXiv preprint arXiv:2007.05674*, 2020.

[6] J. Schrum, V. Volz, and S. Risi, "Cppn2gan: Combining compositional pattern producing networks and gans for large-scale pattern generation," *arXiv preprint arXiv:2004.01703*, 2020.

[7] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in neural information processing systems*, 2014, pp. 2672–2680.

[8] A. Odena, "Semi-supervised learning with generative adversarial networks," *arXiv preprint arXiv:1606.01583*, 2016.

[9] M. Mirza and S. Osindero, "Conditional generative adversarial nets," *arXiv preprint arXiv:1411.1784*, 2014.

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

[11] H. Zhang, I. Goodfellow, D. Metaxas, and A. Odena, "Self-attention generative adversarial networks," in *International Conference on Machine Learning*. PMLR, 2019, pp. 7354–7363.

[12] A. Brock, J. Donahue, and K. Simonyan, "Large scale gan training for high fidelity natural image synthesis," *arXiv preprint arXiv:1809.11096*, 2018.

[13] J. M. Font, R. Izquierdo, D. Manrique, and J. Togelius, "Constrained level generation through grammar-based evolutionary algorithms," in *European Conference on the Applications of Evolutionary Computation*. Springer, 2016, pp. 558–573.

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

[15] P. Bontrager, W. Lin, J. Togelius, and S. Risi, "Deep interactive evolution," in *International Conference on Computational Intelligence in Music, Sound, Art and Design*. Springer, 2018, pp. 267–282.

[16] P. Bontrager, J. Togelius, and N. Memon, "Deepmasterprint: Generating fingerprints for presentation attacks," *arXiv preprint arXiv:1705.07386*, 2017.

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