Evolving Mario Levels in the Latent Space of a Deep Convolutional Generative Adversarial Network

Vanessa Volz
TU Dortmund University
Dortmund, Germany
vanessa.volz@tu-dortmund.de

Jacob Schrum
Southwestern University
Georgetown, TX USA
schrum2@southwestern.edu

Jialin Liu
Queen Mary University of London
London, UK
jialin.liu@qmul.ac.uk

Simon M. Lucas
Queen Mary University of London
London, UK
simon.lucas@qmul.ac.uk

Adam Smith
University of California
Santa Cruz, CA USA
amsmith@ucsc.edu

Sebastian Risi
IT University of Copenhagen
Copenhagen, Denmark
sebr@itu.dk

ABSTRACT

Generative Adversarial Networks (GANs) are a machine learning approach capable of generating novel example outputs across a space of provided training examples. Procedural Content Generation (PCG) of levels for video games could benefit from such models, especially for games where there is a pre-existing corpus of levels to emulate. This paper trains a GAN to generate levels for Super Mario Bros using a level from the Video Game Level Corpus. The approach successfully generates a variety of levels similar to one in the original corpus, but is further improved by application of the Covariance Matrix Adaptation Evolution Strategy (CMA-ES). Specifically, various fitness functions are used to discover levels within the latent space of the GAN that maximize desired properties. Simple static properties are optimized, such as a given distribution of tile types. Additionally, the champion A* agent from the 2009 Mario AI competition is used to assess whether a level is playable, and how many jumping actions are required to beat it. These fitness functions allow for the discovery of levels that exist within the space of examples designed by experts, and also guide the search towards levels that fulfill one or more specified objectives.

KEYWORDS

Generative Adversarial Network, Procedural Content Generation, Mario, CMA-ES, Game

ACM Reference Format:
Vanessa Volz, Jacob Schrum, Jialin Liu, Simon M. Lucas, Adam Smith, and Sebastian Risi. 2018. Evolving Mario Levels in the Latent Space of a Deep Convolutional Generative Adversarial Network. In *GECCO '18: Genetic and Evolutionary Computation Conference, July 15–19, 2018, Kyoto, Japan.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3205455.3205517

GECCO '18, July 15–19, 2018, Kyoto, Japan

ACM ISBN 978-1-4503-5618-3/18/07…$15.00
https://doi.org/10.1145/3205455.3205517

1 INTRODUCTION

Procedural Content Generation (PCG) covers the creation of game content (e.g., game rules, levels, characters, background stories, textures and sound) by algorithms with or without help from human designers [23]. The history of digital PCG goes back to the 1980s, when the game *Elite*[1] was published. Due to the limited memory capacities of personal computers of the time, a decision was made to save only the seed to a random generation process rather than to store complete level designs. From a specified seed value, a generator would proceed to deterministically (pseudo-randomly) recreate a sequence of numbers which were then used to determine the names, positions, and other attributes of game objects. The adoption of PCG exploded during the 2000s when it was picked up in application to game graphics [5]. Since then, much work has sprung up around PCG in both the industry and academic spheres [17]. Additionally, various competitions have been organized in international conferences during recent years, such as the Mario AI Level Generation Competition[2], Platformer AI Competition[3], AI Birds Level Generation Competition[4] and the General Video Game AI (GVGAI)[5] Level Generation Competition [11]. The approach introduced here is an example of PCG via Machine Learning (PCGML; [20]), which is a recently emerging research area.

The approach presented in this paper is to create new game levels that emulate those designed by experts using a variant of a Generative Adversarial Network (GAN) [7]. GANs are deep neural networks trained in an unsupervised way that have shown exceptional promise in reproducing aspects of images from a training set. Additionally, the space of levels encoded by the GAN is further searched using the Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES) [9], in order to discover levels with particular attributes. The idea of *latent variable evolution* (LVE) was recently introduced in the context of interactive evolution of images [2] and fingerprint generation [3] but so far has not been applied to PCG of video game levels.

The specific game in this paper is *Super Mario Bros*[6], but the technique should generalize to any game for which an existing corpus of levels is available. Our GAN is trained on a single level

[1] https://en.wikipedia.org/wiki/Elite_(video_game)
[2] http://www.marioai.org/LevelGeneration
[3] https://sites.google.com/site/platformersai/LevelGeneration
[4] https://aibirds.org/other-events/level-generation-competition.html
[5] http://www.gvgai.net/
[6] https://en.wikipedia.org/wiki/Super_Mario_Bros.

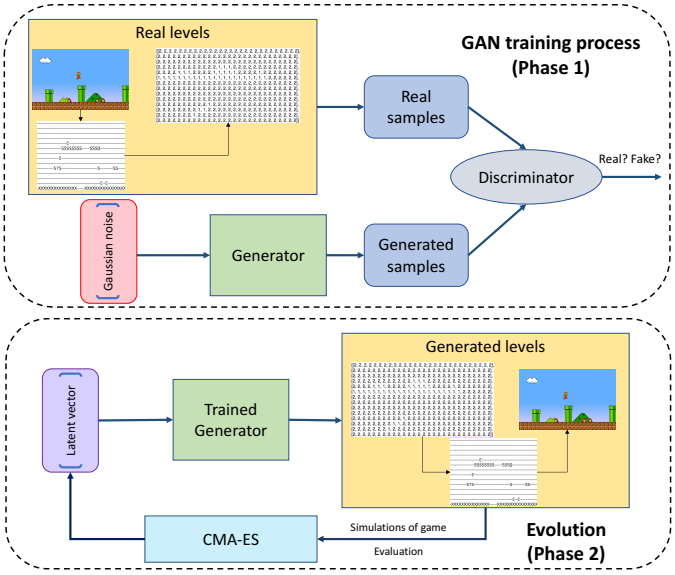


Figure 1: Overview of the GAN training process and the evolution of latent vectors. The approach is divided into two distinct phases. In Phase 1 a GAN is trained in an unsupervised way to generate Mario levels. In the second phase, we search for latent vectors that produce levels with specific properties.

from the original *Super Mario Bros*, available as part of the Video Game Level Corpus (VGLC) [21]. CMA-ES is then used to find ideal inputs to the GAN from within its latent vector space (Figure 1). During the evolution, the generated levels are evaluated using different fitness functions. This allows for the discovery of levels that exist between and beyond those sparse examples designed by human designers, and that also optimize additional goals. Our approach is capable of generating playable levels that meet various goals and is ready to be applied to level generation of other games, such as the games in the GVGAI framework. By training on only a single level, we are able to show that even with a very limited dataset, we can apply the presented approach successfully.

The rest of this paper is structured as follows. Section 2 introduces the background and related work. The main approach is described in Section 3. Section 4 details the experimental design. The experimental results are presented and discussed in Section 5. Section 6 then concludes the paper.

2 BACKGROUND AND RELATED WORK

In this section, Procedural Content Generation for games is discussed, followed descriptions of technical tools applied in this paper: GANs, latent variable evolution, and CMA-ES.

2.1 Procedural content generation

Togelius *et al.* [23] defined Procedural Content Generation (PCG) as the *algorithmic creation of game content with limited or indirect user input* [22, 23, 25]. Examples of game content include game rules, levels, maps/mazes, characters, weapons, vehicles, background stories, textures and sound. Automatic game level generation, with little or no human intervention, is a challenging problem. For some games, the levels are represented as maps or mazes [6]. Examples include *Doom*, *Pac-Man*, and *Super Mario Bros*, one of the classic platform video games created by *Nintendo*.

The first academic Procedural Content Generation competition was the 2010 Mario AI Championship [18], in which the participants were required to submit a level generator which implements a provided Java interface and returns a new level within 60 seconds. The competition framework was implemented based on *Infinite Mario Bros* [7], a public clone of *Super Mario Bros*.

The availability and popularity of the Mario AI framework has led to several approaches for generating levels for *Super Mario Bros*. Shaker *et al.* [16] evolved Mario levels using Grammatical Evolution (GE). In 2016, Summerville and Mateas [19] applied Long Short-Term Memory Recurrent Neural Networks (LSTMs) to generate game levels trained on existing Mario levels, and then improved the generated levels by incorporating player path information. This approach inspired a novel approach to level generation, in which new levels are generated automatically from a sketch of some desired path drawn by a human designer. Another approach that was trained using existing Mario levels is that of Jain *et al.* [10], which trained auto-encoders to generate new levels using a binary encoding where empty (accessible) spaces are represented by 0 and the others (e.g., terrain, enemy, tunnel, etc.) by 1. Though this approach could generate interesting levels, the use of random noise inputs into the trained auto-encoder sometimes led to problematic levels. Additionally, because of the binary encoding, no distinction was made between various possible types of tiles.

2.2 Generative Adversarial Networks

Generative Adversarial Networks (GANs) were first introduced by Goodfellow *et al.* [7] in 2014. Their training process can be seen as a two-player adversarial game in which a generator G (faking samples decoded from a random noise vector) and a discriminator D (distinguishing real/fake samples and outputting 0 or 1) are trained at the same time by playing against each other. The discriminator D aims at minimizing the probability of mis-judgment, while the generator G aims at maximizing that probability. Thus, the generator is trained to deceive the discriminator by generating samples that are good enough to be classified as genuine. Training ideally reaches a steady state where G reliably generates realistic examples and D is no more accurate than a coin flip.

GANs quickly became popular in some sub-fields of computer vision, such as image generation. However, training GANs is not trivial and often results in unstable models. Many extensions have been proposed, such as Deep Convolutional Generative Adversarial Networks (DCGANs) [15], a class of Convolutional Neural Networks (CNNs); Auto-Encoder Generative Adversarial Networks (AE-GANs) [13]; and Plug and Play Generative Networks (PPGNs) [14]. A particularly interesting variation are Wasserstein GANs (WGANs) [1, 8]. WGANs minimize the approximated Earth-Mover (EM) distance (also called Wasserstein metric), which is used to measure how different the trained model distribution and the real distribution are. WGANs have been demonstrated to achieve more stable training than standard GANs.

[7] https://tinyurl.com/yan4ep7g

At the end of training, the discriminator D is discarded, and the generator G is used to produce new, novel outputs that capture the fundamental properties present in the training examples. The input to G is some fixed-length vector from a latent space (usually sampled from a block-uniform or isotropic Gaussian distribution). For a properly trained GAN, randomly sampling vectors from this space should produce outputs that would be mis-classified as examples of the target class with equal likelihood to the true examples. However, even if all GAN outputs are perceived as valid members of the target class, there could still be a wide range of meaningful variation within the class that a human designer would want to select between. A means of searching within the real-valued latent vector space of the GAN would allow a human to find members of the target class that satisfy certain requirements.

2.3 Latent variable evolution

The first *latent variable evolution* (LVE) approach was introduced by Bontrager *et al.* [3]. In their work the authors train a GAN on a set of real fingerprint images and then apply evolutionary search to find a latent vector that matches with as many subjects in the dataset as possible.

In another paper Bontrager et al. [2] present an interactive evolutionary system, in which users can evolve the latent vectors for a GAN trained on different classes of objects (e.g. faces or shoes). Because the GAN is trained on a specific target domain, it becomes a compact and robust genotype-to-phenotype mapping (i.e. most produced phenotypes do resemble valid domain artifacts) and users were able to guide evolution towards images that closely resembled given target images. Such target based evolution has been shown to be challenging with other indirect encodings [26].

Because of the promising previous LVE approaches, in this paper we investigate how latent GAN vectors can be evolved through a fitness-based approach in the context of level generation.

2.4 CMA-ES

Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES) [9] is a powerful and widely used evolutionary algorithm that is well suited for evolving vectors of real numbers. The CMA-ES is a second-order method using the covariance matrix estimated iteratively by finite differences. It has been demonstrated to be efficient for optimizing non-linear non-convex problems in the continuous domain without a-priori domain knowledge, and it does not rely on the assumption of a smooth fitness landscape.

We applied CMA-ES to evolve the latent vector and applied several fitness functions on the generated levels. Fitness functions can be based on purely static properties of the generated levels, or on the results of game simulations using artificial agents.

3 APPROACH

The approach is divided into two main phases, visualised in Figure 1. First, a GAN is trained on an existing Mario level (Figure 2). The level is encoded as a multi-dimensional array as described in Section 3.1 and depicted in the yellow box. The generator (green) operates on a Gaussian noise vector (red) and is trained to output levels using the same representation. The discriminator is then employed to tell the existing and generated levels apart. Both the generator and discriminator are trained using an adversarial learning process as described in Section 2.2.

Once this process is completed, the generator network of the GAN, G, can be viewed as our learned genotype-to-phenotype mapping that takes as input a latent vector (blue) of real numbers (of size 32 in the experiments in this paper) and produces a tile-level description of a Mario level. Instead of simply drawing independent random samples from the latent space, we put exploration under evolutionary control (using a CMA-ES in this case). In other words, we search through the space of latent vectors to produce levels with different desirable properties such as distributions of tiles, difficulty, etc. Specific parts of the training process are discussed in the following.

3.1 Level representation

Mario levels have different representations within the Video Game Level Corpus (VGLC) [21] and Mario AI framework. Both representations are tile based. Specifically, each Mario level from the VGLC uses a particular character symbol to represent each possible tile type. However, it should be noted that this VGLC representation is primarily concerned with functional properties of tiles rather than artistic properties, and is thus incapable of distinguishing certain visually distinct tile types. The only exception are *pipes*, which are represented by four visually distinct tile types, despite all being functionally equivalent to an impassable ground block. Interestingly, the VGLC encoding ignores functional differences between different enemy types by providing only a single character symbol to represent enemies, which we choose to map to the generic *Goomba* enemy type.

To encode the levels for training, each tile type is represented by a distinct integer, which is converted to a one-hot encoded vector before being input into the discriminator. The generator network also outputs levels represented using the one-hot encoded format, which is then converted back to a collection of integer values. Levels in this integer-based format are then sent to the Mario AI framework for rendering. Mario AI allows for a broader range of artistic diversity in its tile types, but because of the simplicity of the VGLC encoding, only a simple subset of the available Mario AI tiles are used. The mapping from VGLC tile types and symbols, to GAN training number codes, and finally to Mario AI tile visualizations is detailed in the Table 1.

The GAN input files were created by processing a level file from the VGLC for the original Nintendo game *Super Mario Bros*, which is shown in Figure 2. Each level file is a plain text file where each line of the file corresponds to a row of tiles in the Mario level. Within a level all rows are of the same length, and each level is 14 tiles high. The GAN expected to always see a rectangular image of the same size, hence each input image was generated by sliding a 28 (wide) x 14 (high) window over the raw level from left to right, one tile at a time. The width of 28 tiles is equal to the width of the screen in Mario. In the input files each tile type is represented by a specific character, which was then mapped to a specific integer in the training images, as listed in Table 1. This procedure created a set of 173 training images.

While we could have used a larger dataset instead of this relatively small one, its use allows us to test the GAN's ability to

Figure 2: The Training Level. The training data is generated by sliding a 28×14 window over the level from left to right, one tile at a time.

Table 1: Tile types used in generated Mario levels. The symbol characters come from the VGLC encoding, and the numeric identity values are then mapped to the corresponding values employed by the Mario AI framework to produce the visualization shown. The numeric identity values are expanded into one-hot vectors when input into the discriminator network during GAN training.

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

learn from relatively little data, which could be especially important for games that do not offer such a large training corpus as Mario. Additionally, because of the smaller training set it is possible to manually inspect if the LVE approach is able to generate levels with properties not directly found in the training set itself.

3.2 GAN training

Our Deep Convolutional GAN (DCGAN) is adapted from the model in [1] and trained with the WGAN algorithm. The network architecture is shown in Figure 3. Following the original DCGAN architecture, the network uses strided convolutions in the discriminator and fractional-strided convolutions in the generator. Additionally, we employ *batchnorm* both in the generator and discriminator after each layer. In contrast to the original architecture in [1], we use ReLU activation functions for all layers in the generator, even for the output (instead of Tanh), which we found gave better results. Following [1], the discriminator uses LeakyReLU activation in all layers.

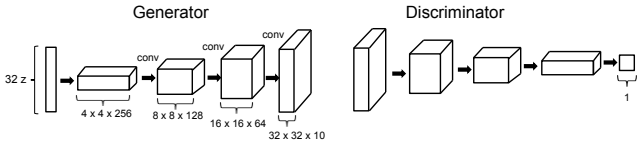


Figure 3: The Mario DCGAN architecture.

When training the GAN, each integer tile was expanded to a one-hot vector. Therefore the training inputs for the discriminator are 10 channels (one-hot across 10 possible tile types) of size 32×32 (the DCGAN implementation we used required the input size to be a multiple of 16 so the levels were padded). For example, in the first channel, the location of ground titles are marked with a 1.0, while all other locations are set to 0.0. The size of the latent vector input to the generator has a length of 32.

Once training of the GAN is completed the generator represents our learned genotype-to-phenotype mapping. When running evolution, the final $10 \times 32 \times 32$ dimensional output of this generator is cropped to $10 \times 28 \times 14$ and each output vector for a tile is converted to an integer using the argmax operator, resulting in a level that can be decoded by the Mario AI framework.

4 EXPERIMENTS

The approach of this paper is tested in two different sets of experiments that can be divided into representation-based and agent-based testing, which are described in more detail below. The experiments are intended as a proof of concept. To apply the proposed approach within a game, the employed fitness functions need to be designed more carefully to correspond to the intended purpose and required properties of the generated content. The whole project is available on Github[8].

4.1 Representation-based testing

In the representation-based scenarios we directly optimize for a certain distribution of tiles using CMA-ES. In more detail, we test (1) if the approach can generate levels with a certain number of ground titles, and (2) a combination of ground titles and number of enemies. The goal of the second experiment is to create a level composed of multiple subsections that increases gradually in difficulty. In all experiments, we seek to minimize the following functions.

Fitness in the first experiment is based on the distance between the produced fraction of ground titles g and the targeted fraction t:

$$F_{ground} = \sqrt{(g-t)^2}.$$

In the second experiment, we evolve five different subsections with 100% ground coverage for sections 1 and 2, and 70% coverage for sections 3–5. For the fourth and fifth subsection fitness is also determined by maximizing the total number of n of enemies:

$$F = F_{ground} + 0.5(20.0 - n).$$

This particular weighting was found through prior experimentation.

4.2 Agent-based testing

While being able to generate levels with exactly the desired number of ground tiles and enemies is one desirable feature of a level generator, a fitness function based entirely on the level representation has two inherent weaknesses:

[8] https://github.com/TheHedgeify/DagstuhlGAN

- Levels with maximal fitness value might not be playable, especially if they are optimized for a small number of ground tiles and/or a large number of enemies.
- The number of ground tiles and enemies does not necessarily affect the playthrough of a human or AI agent, and may thus not result in levels with the desired difficulty. E.g., the enemies might fall into a hole before Mario can reach them or there might exist an alternative route that avoids difficult jumps.

These problems can be alleviated by using an evaluation that is based on playthrough data instead of just the level representation. This way, playability can be explicitly tested and characteristics of a playthrough can be observed directly.

To this end, we implemented agent-based testing using the Mario AI competition framework, as there are a variety of agents already available [24]. The CMA-ES is used to find levels that optimize an agent-based fitness function described in the following. To evaluate a level, the latent vector in question is mapped to $[-1, 1]^n$ with a sigmoid function and then sent to the generator model in order to obtain the corresponding level. The level is then imported into the Mario AI framework using the encoding detailed in Table 1, so that agent simulations can be run.

While there are a variety of properties that can be measured using agent-based testing, for this proof-of-concept we chose to specifically focus on the two weaknesses of representation-based fitness functions mentioned above. As before, our use case is to find playable levels with a scalable difficulty.

Given that the A* agent by Robin Baumgarten[9] (winner of the 2009 Mario AI competition) performs at a super-human level, we use its performance to determine the playability of a given level. For an approximation of experienced difficulty, we use the number of jump actions performed by the agent. The correlation between the number of jumps and difficulty is an assumption, however, jumping is the main mechanic in Mario and is required to overcome obstacles such as holes and enemies. The fitness function we seek to minimize is:

$$F_1 = \begin{cases} -p & \text{for } p < 1 \\ -p - \#jumps & \text{for } p = 1, \end{cases}$$

where p is the fraction of the level that was completed in terms of progress on the x-axis.

In order to investigate the controllability of the level generation process via agent-based testing, we ran additional experiments where we sought playable levels with a minimal number of required jumps. The fitness function in this case is

$$F_2 = \begin{cases} -p + 60 & \text{for } p < 1 \\ -p + \#jumps & \text{for } p = 1, \end{cases}$$

where p is the fraction of the level that was completed in terms of progress on the x-axis. The offset of 60 for the incomplete levels was chosen after preliminary experiments so that unbeatable levels where the agent is trapped and repeatedly jumps are discouraged. As a result, passable levels will always score a higher fitness than impassable ones.

[9]https://www.youtube.com/watch?v=DlkMs4ZHHr8

Since the exact number of jumps is non-deterministic and can produce outliers if the agent gets stuck under an overhang, the actual fitness value in both cases is the average of 10 simulations.

4.3 Experimental parameters

For the non-agent testing we use a Python CMA-ES implementation[10]. Because Mario AI is implemented in Java, we use a Java implementation of CMA-ES for the agent-based testing[11] to evolve the latent vector passed to the trained Python generator.

For both Java and Python, the CMA-ES population size is $\lambda = 14$. For the non-agent based setting we set the initial point to 0, while we set it to a random point within $[-1, 1]^n$ for the more complex fitness function in the agent-based setting. Similarly, the standard deviation is initialized to 1.0 for non-agent and 2.0 for agent-based testing. The CMA-ES is run for a maximum of $1,000$ function evaluations.

A total of 20 runs were performed for the non-agent based experiments and 40 runs each for both (F_1 and F_2) of the agent-based CMA-ES experiments.

Our WGAN implementation is built on a modified version of the original PyTorch WGAN code[12]. Both the generator and discriminator are trained with RMSprop with a batch size of 32 and the default learning rate of 0.00005 for 5,000 iterations.

5 RESULTS

To get a better understanding of the GAN's suitability as a genotype-to-phenotype mapping we first tested for expressivity of the encoding and to what degree it has locality (i.e. small mutations resulting in small phenotype changes). Figure 4 shows examples of (a) a randomly sampled GAN and (b) samples around a particular latent vector generated by adding uniformly sampled noise in the range [-0.3, 0.3]. While some aspects (e.g. pipes) are sometimes not captured perfectly, the GAN is able to generate a variety of different level layouts that capture some important aspects of the training corpus (Figure 4). Additionally, mutations around a particular latent vector vary in interesting ways while still resembling the parent vector.

5.1 Representation-based testing

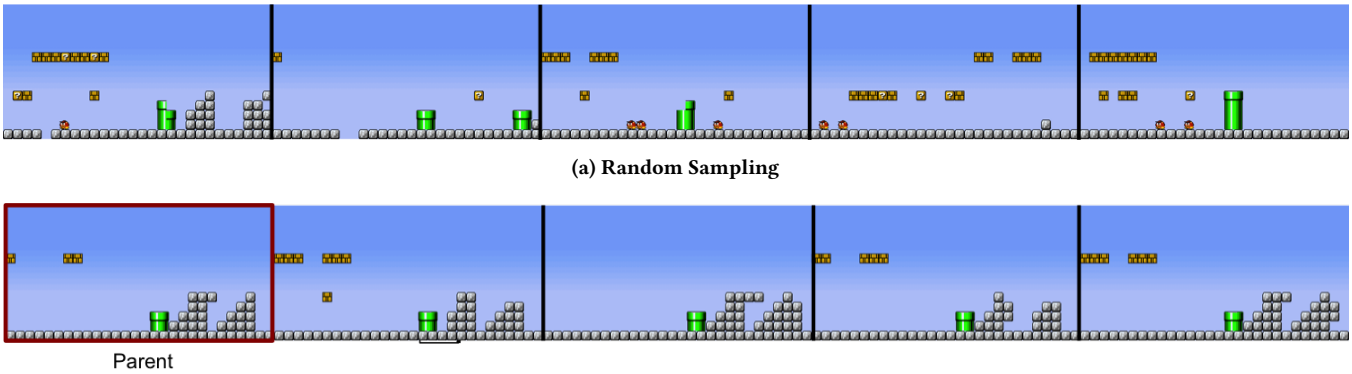
Figure 5 shows how close the approach can optimize the percentage of ground tiles towards a certain targeted distribution. The results demonstrate that in almost every run we can get very close to a targeted percentage.

Figure 6 shows a level that was created with increasing difficulty in mind: 100% ground coverage for sections 1 and 2, 70% coverage for sections 3–5, and maximizing the total number n of enemies for section 4 and 5. The approach is able to optimize both the ground distribution as well as the number of enemies.

5.2 Agent-based testing

Figure 7 shows some of the best and worst results obtained for both fitness functions. CMA-ES did discover some non-playable levels as depicted in Figure 7c. Among the best results for fitness function F_1 (i.e. playable levels with a high number of required jumps) are level

[10]https://pypi.python.org/pypi/cma
[11]https://www.lri.fr/~hansen/cmaes_inmatlab.html#java
[12]https://github.com/martinarjovsky/WassersteinGAN



(a) Random Sampling

(b) Mutations

Figure 4: Generated Examples. Shown are samples produced by the GAN by (a) sampling random latent vectors, and (b) randomly mutating a specific latent vector. The main result is that the generator is able to produce a wide variety of different level layouts, but varied offspring still resemble their parent.

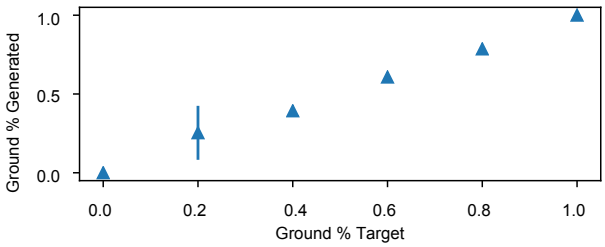


Figure 5: Optimized for different percentage of ground tiles. Mean values across 20 runs are shown along with one standard deviation. Except for a ground level fraction of 20% the approach is able to always discover the latent code that produces the desired target percentage of ground tiles.

sections with and without slight title errors (a and d). In the future, the representation of levels could be improved or directly repaired in such a way that the pipes are no longer a cause for visually faulty levels. The level depicted in (b) is one of the best examples found when optimizing for fitness F_2 (i.e. playable with a small number of required jumps). The level requires only one single jump over the enemy and is easy to solve.

Despite using a noisy fitness function, which is only an approximation of actual level difficulty, the optimization algorithm is able to discover a variety of interesting results. While we observe some individuals with a small fitness being generated even late into the optimisation process (Figure 8, top), the average fitness value of generated individuals decreases with increasing iteration (Figure 8, bottom). The overall decrease of fitness over time does suggest that the GAN-based level generation process is indeed controllable. It is likely that the low-scoring individuals in later iterations result from the fact that levels that require a high jump count and levels that are not playable are close in the search space. We suspect that further modification of the fitness function and using a more robust CMA-ES version intended for noisy optimization could further improve the observed optimization efficiency.

Overall, we show that we are able to create a variety of levels that translate to a plethora of different playthroughs. However, it is of course difficult to find a suitable fitness function, that (1) expresses the desired game qualities but (2) is also tractable for an optimization algorithm. Additionally, the noise of the function should be investigated in depth. Since the evaluation of the fitness function does take considerable time, one should probably also consider using other approaches, for example surrogate-based algorithms.

5.3 Discussion and Future Work

Although GANs are known for their success in generating photo-realistic images (composed of pixels with blendable color values), their application to discrete tiled images is less explored. The results in this paper demonstrate that GANs are in general able to capture the basic structure of a Mario level, i.e. a traversable ground with some obstacles (cf. Figure 4). Additionally, we are able to evolve levels that are not just replications of the training examples (compare Figures 2 and 6).

However, sometimes certain broken structures in the output of the GAN are apparent: e.g. incomplete pipes. In the future, this might be addressed by borrowing ideas from text (symbol sequence) generation models such as LSTMs [12]. In these models the discrete choice of symbol at each observable location is conditioned not only on the continuous output of a hidden layer but also the discrete choice of the immediately preceding symbol. This approach would combine the discrete context dependence of Snodgrass' Multi-dimensional Markov Chains (which accurately capture only local tile structures) with the global structure enforced by the upsampling convolutional layers used in our GAN.

An intriguing future possibility is to first train a generator off-line and then distribute the architecture and weights of this network with a game so that extremely rapid on-line level generation can be carried out with the model (perhaps to support evolving player-adapted level designs). Depending on the fitness function chosen, this could be employed for both dynamically adapting the difficulty of levels, but also for providing more exploration-focused content by adding more coins in places that are difficult to reach.

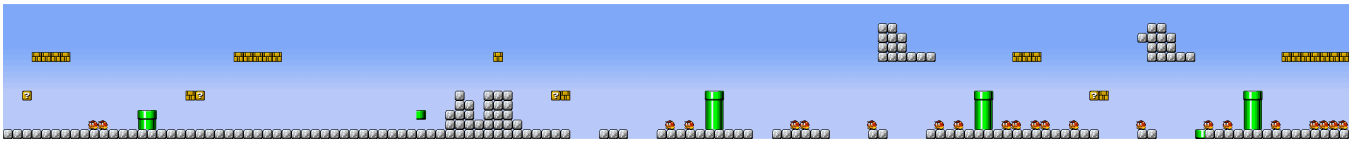
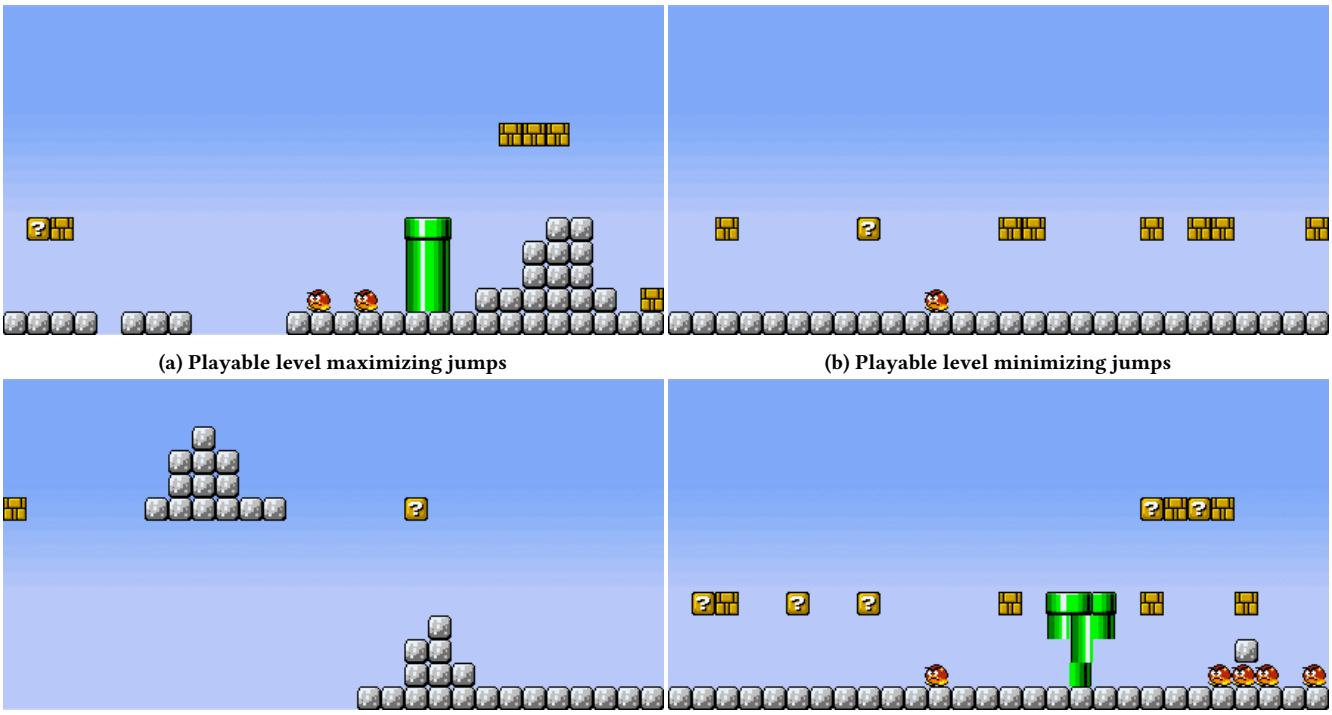


Figure 6: Level with increasing difficulty. Our LVE approach can create levels composed of multiple parts that gradually increase in difficulty (less ground tiles, more enemies). In the future this approach could be used to create a level in real-time that is tailored to the particular skill of the player (dynamic difficulty adaptation).



(a) Playable level maximizing jumps

(b) Playable level minimizing jumps

(c) Unplayable level

(d) Broken titles

Figure 7: Agent-based optimization examples. (a) and (b) show good examples of levels in which the number of jumps is maximized (F_1), and minimized (F_2), respectively. (c) shows an example of one of the worst individuals found (not playable, F_1). An example of an individual that reaches high fitness (maximizing jumps, F_1) but has broken titles is shown in (d).

Our generator focuses on recreating just the tile-level description of a level primarily because this is the data available in the Video Game Level Corpus. With a richer dataset capturing summaries of player behavior (which actions they typically took when their character occupied a given tile location), we could also train a network to output level designs along with design annotations capturing expectations about player behavior and experience for the newly generated level. Even if these annotation layer outputs go unused for generated levels, having them present in the training data could help the network learn patterns that are specifically relevant to player behavior, beyond basic spatial tile patterns. In general, training with a larger level corpus could allow the GAN to capture a greater variety of different Mario level styles.

One potential area of future work is the use of Multi-Objective Optimization Algorithms [4] to evolve the latent vector using multiple evaluation criteria. Many different criteria can make video game levels enjoyable to different people, and a typical game needs to contain a variety of different level types, so evolving with multiple objective functions could be beneficial. Given such functions, it would also be interesting to compare our results with other procedurally generated content, as well as manually designed levels, in terms of the obtained values. However, further work on automatic game evaluation is required to define purposeful fitness functions.

6 CONCLUSION

This paper presented a novel latent variable evolution approach that can evolve new Mario levels after being trained in an unsupervised way on an existing Mario level. The approach can optimize levels for different distributions and combinations of tile types, but can also optimize levels with agent-based evaluation functions. While the GAN is often able to capture the high-level structure of the training level, it sometimes produces broken structures. In the future this

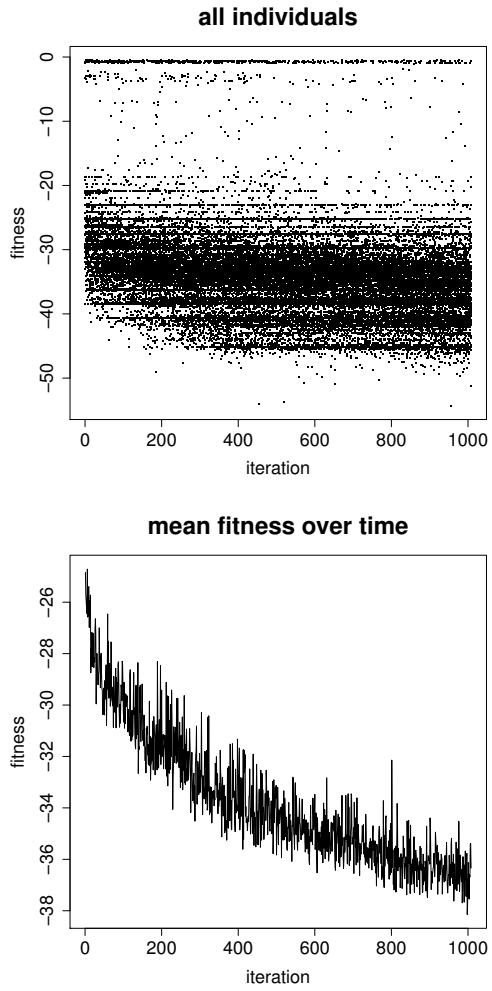


Figure 8: Agent-based fitness progression F_1: Top: Fitness of generated individual at CMA-ES iteration. Bottom: Average fitness of individuals generated at given iteration. Lower values are better.

could be remedied by applying GAN models that are better suited to the discrete representations employed in such video game levels. The main conclusion is that LVE is a promising approach for fast generation of video game levels that could be extended to a variety of other game genres in the future.

Acknowledgements

The authors would like to thank the Schloss Dagstuhl team and the organisers of the Dagstuhl Seminar 17471 for a creative and productive seminar.

REFERENCES

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

[2] Philip Bontrager, Wending Lin, Julian Togelius, and Sebastian Risi. 2018. Deep Interactive Evolution. *European Conference on the Applications of Evolutionary Computation (EvoApplications)*.

[3] Philip Bontrager, Julian Togelius, and Nasir Memon. 2017. DeepMasterPrint: Generating Fingerprints for Presentation Attacks. *arXiv preprint arXiv:1705.07386* (2017).

[4] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. 2002. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6 (2002), 182–197.

[5] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. 2002. *Texturing and Modeling: A Procedural Approach* (3rd ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[6] Alison Gazzard. 2013. *Mazes in Videogames: meaning, metaphor and design*. McFarland.

[7] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, and others. 2014. Generative Adversarial Nets. In *Advances in Neural Information Processing Systems*. 2672–2680.

[8] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron Courville. 2017. Improved Training of Wasserstein GANs. In *Advances in Neural Information Processing Systems 30 (NIPS 2017)*. 5769–5779. arXiv preprint arXiv:1704.00028.

[9] Nikolaus Hansen, Sibylle D Müller, and Petros Koumoutsakos. 2003. Reducing the Time Complexity of the Derandomized Evolution Strategy with Covariance Matrix Adaptation (CMA-ES). *Evolutionary Computation* 11, 1 (2003), 1–18.

[10] Rishabh Jain, Aaron Isaksen, Christoffer Holmgård, and Julian Togelius. 2016. Autoencoders for Level Generation, Repair, and Recognition. In *ICCC Workshop on Computational Creativity and Games*. arXiv preprint arXiv:1702.00539.

[11] Ahmed Khalifa, Diego Perez-Liebana, Simon M Lucas, and Julian Togelius. 2016. General Video Game Level Generation. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*. ACM, 253–259.

[12] Matt J Kusner and José Miguel Hernández-Lobato. 2016. GANs for Sequences of Discrete Elements with the Gumbel-Softmax Distribution. In *Workshop on Adversarial Training (NIPS 2016)*. arXiv preprint arXiv:1611.04051.

[13] Alireza Makhzani, Jonathon Shlens, Navdeep Jaitly, and Ian Goodfellow. 2016. Adversarial Autoencoders. In *International Conference on Learning Representations*. arXiv preprint arXiv:1511.05644.

[14] Anh Nguyen, Jason Yosinski, Yoshua Bengio, Alexey Dosovitskiy, and Jeff Clune. 2016. Plug & Play Generative Networks: Conditional Iterative Generation of Images in Latent Space. *arXiv preprint arXiv:1612.00005* (2016).

[15] Alec Radford, Luke Metz, and Soumith Chintala. 2016. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. In *International Conference on Learning Representations (ICLR)*. arXiv preprint arXiv:1511.06434.

[16] Noor Shaker, Miguel Nicolau, Georgios N Yannakakis, Julian Togelius, and Michael O'neill. 2012. Evolving Levels for Super Mario Bros Using Grammatical Evolution. In *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*. IEEE, 304–311.

[17] Noor Shaker, Julian Togelius, and Mark J Nelson. 2016. *Procedural Content Generation in Games*. Springer.

[18] Noor Shaker, Julian Togelius, Georgios N Yannakakis, and others. 2011. The 2010 Mario AI championship: Level Generation Track. *IEEE Transactions on Computational Intelligence and AI in Games* 3, 4 (2011), 332–347.

[19] Adam Summerville and Michael Mateas. 2016. Super Mario as a String: Platformer Level Generation via LSTMs. In *1st International Joint Conference of DiGRA and FDG*.

[20] Adam Summerville, Sam Snodgrass, Matthew Guzdial, and others. 2017. Procedural Content Generation via Machine Learning (PCGML). *arXiv preprint arXiv:1702.00539* (2017).

[21] Adam James Summerville, Sam Snodgrass, Michael Mateas, and Santiago Ontañón Villar. 2016. The VGLC: The Video Game Level Corpus. *Proceedings of the 7th Workshop on Procedural Content Generation* (2016).

[22] Julian Togelius, Alex J Champandard, Pier Luca Lanzi, and others. 2013. Procedural content generation: Goals, challenges and actionable steps. In *Dagstuhl Follow-Ups*, Vol. 6. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[23] Julian Togelius, Emil Kastbjerg, David Schedl, and Georgios N Yannakakis. 2011. What is procedural content generation? Mario on the borderline. In *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*. ACM, 3.

[24] Julian Togelius, Noor Shaker, Sergey Karakovskiy, and Georgios N. Yannakakis. 2013. The Mario AI Championship 2009-2012. *AI Magazine* 34, 3 (2013), 89–92.

[25] Julian Togelius, Georgios N Yannakakis, Kenneth O Stanley, and Cameron Browne. 2011. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games* 3, 3 (2011), 172–186.

[26] Brian G Woolley and Kenneth O Stanley. 2011. On the deleterious effects of a priori objectives on evolution and representation. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*. ACM, 957–964.