VGL-GAN: Video Game Level Generation using Deep Convolutional Generative Adversarial Network

Abhinav Bohra*[1] Suryansh Kumar*[2] Maitreyi Swaroop*[3] Mihir Shrivastava*[4]

AI60201: Graphical And Generative Models For Machine Learning
Indian Institute of Technology, Kharagpur, India

[1] 18CS30049 [2] 18CS30043 [3] 19MA20065 [4] 18PH20013
* equal contribution

Abstract

Generating levels for video games using Machine Learning models instead of human designers is becoming increasingly common. Among generative models, GANs have shown to be very effective and efficient in generating a large number of levels which are **playable** by humans. In this paper, we explore an alternative GAN architecture applied to the creation of playable game levels with a focus on Super Mario games. We also compare latent space search techniques to optimise inputs to the GAN from within the latent vector space. Finally, we make our codes publicly available[1]

Keywords
Deep Learning, CNN, GAN, Procedural Content Generation, Evolutionary Search Strategies

1 Introduction

Generative adversarial networks (GANs) are becoming a ubiquitous approach to procedurally generate video game levels. While GAN-generated levels are stylistically similar to human-engineered examples, human designers often want to explore the generative design space of GANs to create interesting, yet feasible (playable by humans) levels. As procedural generation becomes increasingly popular for creating video game levels, novel machine learning techniques such as GAN can prove to be viable tools for game designers, not only as a method for automating the level generation process but also in the form of a unique gameplay feature. GAN-based level generation frameworks also help us appreciate the nuances of game-level design and bring into context the features that make a level interesting to play as we explore different methods of generating 'unique' and coherent levels.

[1]https://github.com/abhinav-bohra/VGL-GAN

2 Related Works

Procedural content generation (PCG) refers to creating game content algorithmically, with limited human input (Shaker et al., 2016). The game content can be any asset (e.g., game mechanics, rules, dialog, models, etc) required to realize the game for its players. Generating video game content that fulfils certain constraints (such as playability, variety etc) can be done by evolutionary solutions (Togelius et al., 2011) or constraint satisfaction methods (Smith and Mateas, 2011) and recently via machine learning (PCGML) (Summerville et al., 2018). Previous work in PCGML has enabled automatic generation of video game levels for Super Mario Bros. using LSTMs (Summerville and Mateas, 2016), Markov Chains (Snodgrass and Ontañón, 2014) and probabilistic graphical models (Guzdial and Riedl, 2016).

Studies by Volz et al. (2018) and Giacomello et al. (2018) have independently demonstrated the successful application of generative adversarial networks (GANs) to generate playable video game levels in an unsupervised way from existing video game level corpora. Volz et al. (2018) applied latent space search to find optimal Mario scenes using Covariance Matrix Adaptation Evolution Strategy (CMA-ES) (Hansen and Ostermeier, 2001).

Building up on that, Fontaine et al. proposed a hybrid technique, using CMA-ES and MAP-Elites with a directional variation operator (Covariance Matrix Adaptation MAP-Elites), to efficiently explore the latent space of a GAN to extract levels that vary across a set of specified characteristics.

3 Generative process

3.1 Generative Adversarial Networks

Generative adversarial networks or GANs (Goodfellow et al., 2014) are an approach to generative modelling using deep learning techniques such as convolutional neural networks. Generative mod-

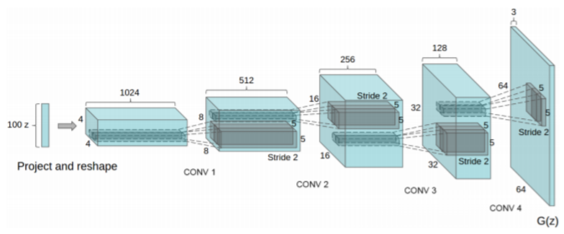

Figure 1: DCGAN Generator Model Architecture

elling involves identifying and learning the regularities or patterns in input data such that the model may be used to produce new examples that might have been reasonably derived from the original dataset. In order to produce new data points with some variances, all generative model types seek to learn the true data distribution of the training set. These models strive to simulate a distribution that is as close to the actual data distribution as possible because it is not always possible to learn the precise distribution of our data either intuitively or explicitly.

GANs offer an innovative way of training a generative model by framing it as a supervised learning problem with two sub-models—the generator model, which we train to create new instances, and the discriminator model, which tries to categorise examples as either real (from the domain) or fake(generated).

3.2 DC-GAN

Radford et al. introduced a class of CNNs called deep convolutional generative adversarial networks (DCGANs), which have certain architectural constraints, and demonstrated that they are a strong candidate for unsupervised learning. DCGAN uses convolutional and convolutional-transpose layers in the generator and discriminator, respectively. Here the discriminator consists of strided convolution layers, batch normalization layers, and LeakyRelu as activation function. It takes a 3x64x64 input image. As shown in figure 1, the generator consists of convolutional-transpose layers, batch normalization layers, and ReLU activations. A 100-dimensional uniform distribution Z is projected to a small spatial extent convolutional representation with many feature maps. A series of four fractionally-strided convolutions then convert this high-level representation into a 3 x 64 × 64 RGB image. Notably, no fully connected or pooling layers are used.

3.3 W-GAN

The Wasserstein Generative Adversarial Network, or W-GAN, proposed by Arjovsky et al., is an extension to the generative adversarial network that both improves the stability when training the model and provides a loss function that correlates with the quality of generated images. Instead of using a discriminator to classify or predict the probability of generated images as being real or fake, the WGAN changes or replaces the discriminator model with a critic that scores the realness or fakeness of a given image. This change is motivated by a theoretical argument that training the generator should seek a minimization of the distance between the distribution of the data observed in the training dataset and the distribution observed in generated examples.

The DCGAN trains the discriminator as a binary classification model to predict the probability that a given image is real. To train this model, the discriminator is optimized using the binary cross entropy loss function. The same loss function is used to update the generator model. Notably, WGAN critic provides very clean gradients on all parts of the space.

3.4 Proposed Model

In order to generate playable Super Mario levels, we designed and implemented a DCGAN using PyTorch framework whose architecture is outlined in fig 2 and fig 3. The labels for each layer describe the size of the output and the convolution kernel dimensions.

The generator is queried with a Gaussian vector of size 128 which is up-sampled to a 17-channel 64x64 output through successive de-convolution operations using *conv2d transpose* layers followed by batch normalization and a Leaky ReLU activation. The idea behind using 17 channels is the presence of 17 unique tiles in the Mario level tileset. The generator has cold-starting layers, where the first two *conv2d transpose* layers do not up-sample the input vector. We also added three dense, fully connected layers at the start before any convolutional layer to introduce sparsity in the input to the convolutional part of the generator. These two architectural choices help the model to avoid immediate mode collapse as well as prevent gradient blow-up.

The discriminator is provided with the output from the generator as well as from the level file of the first level of Super Mario Bros. present in the

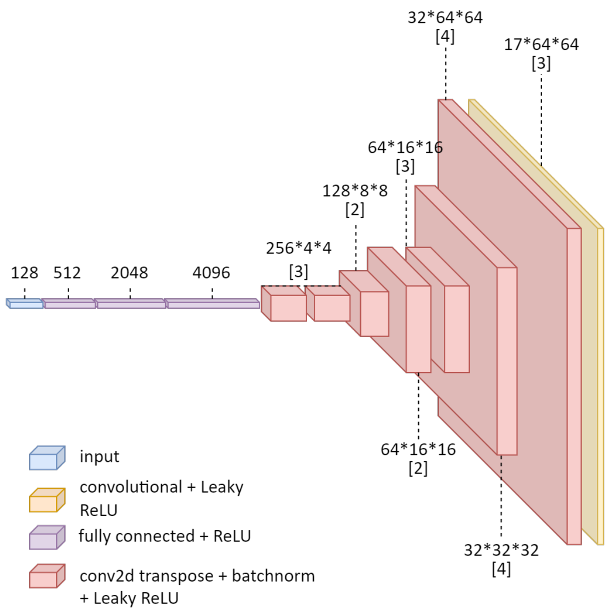

Figure 2: Proposed Generator Architecture

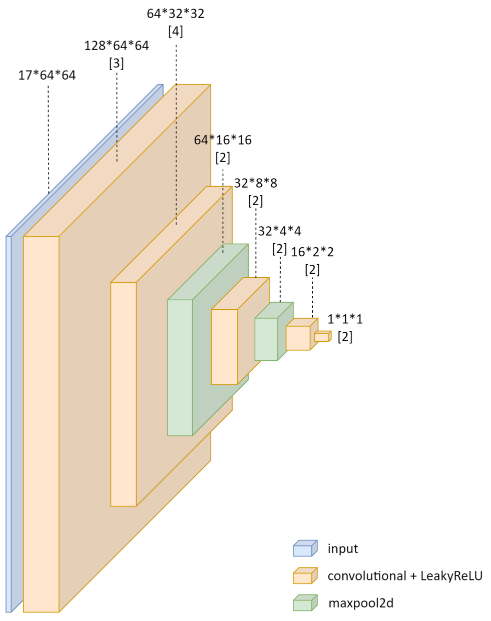

Figure 3: Proposed Discriminator Architecture

Component	LR	β_1	β_2
Generator	0.0001	0.65	0.999
Discriminator	0.00005	0.85	0.99

Table 1: Summary of optimizer hyperparameters used for training the GAN.

Mario-AI-Framework environment. The level is broken up into multiple pieces of length 64 which are then passed to the discriminator. The discriminator uses *conv2d* layers for downsampling the input to a scalar, binary output.

We train the generator and discriminator using Adam optimization schemes and Wasserstein loss metric. After experimenting with different architectures and optimizer hyperparameters, we converged to a paradigm where the levels generated were diverse in terms of the layout and tiles used but also playable by a human player. The parameters are summarized in table 1. The model was trained for 2500 epochs with a batch size of 32.

4 Enhancing Generated Levels through latent space exploration techniques

4.1 Evolutionary Latent Space Search Techniques

Evolutionary search techniques have the following main components:

1. A **population** of *individuals*, i.e. of potential solutions $\mathbf{x} \in \mathbb{R}^n$. Implicitly, this requires an efficient generative process for generating the population. The population is not static and *evolves* over iterations.

2. A **fitness function** $F(\mathbf{x})$, $F : \mathbb{R}^n \to \mathbb{R}$ which is a user-defined objective function. Inspired by evolution, these algorithms select and generate individuals in the population with the aim of maximising the population fitness (*survival of the fittest*). In common practice, most optimisation problems are stated as *minimisation problems*, thus we can always define an objective function $\min f(\mathbf{x})$, $f : \mathbb{R}^n \to \mathbb{R}$ that is equivalent to $\max F(\mathbf{x})$.

3. An evolution strategy (ES) is a set of rules dictating the evolution of the population through selection and variation strategies. That includes all the changes to the population, either through (non/) linear combinations of the members to give rise to new individuals, or rules to draw new individuals from an updated distribution. The ES is geared towards increasing overall population fitness while also allowing for diversity in the population.

A *behavioural characteristic* refers to a user-defined trait associated with an object. Here the object, i.e. our feature vectors $\mathbf{x}$ corresponds to the encoded game level. We consider the corresponding tile distribution, the number of jumps required etc to be the behavioural characteristics which together comprise the behaviour space.

We create a *map* of such vectors/solutions. Initially, the map contains uniformly sampled solutions from the search space, where each cell of the map contains at most one solution, which is the highest performing solution in that behavioural niche (an elite). The map is updated by generating new solutions (by perturbing the elites with Gaussian noise) and replacing the elite in its respective cell if the new solution has higher fitness. If the cell is empty the new solution simply fills the cell.

Below we define the notation that we will use throughout this paper while discussing the latent space search techniques.

Variable	Definition
g	Current generation (during evolution)
λ	Population size
$x_k^{(g)} \in \mathbb{R}^n$	kth individual of the population at iteration g

4.1.1 Guided Random Search Algorithm

Here we proceed by generating a population of a fixed size λ, initially from a standard normal distribution. Then μ of the fittest individuals are chosen and a new population is generated from a Normal distribution centred at the μ fittest values (a Gaussian-Mixture Model), with greater weight given to the fitter individuals. The variance remains as unity. This process is repeated till we obtain the fittest individual meeting the threshold criterion.

4.1.2 Covariance Matrix Adaptation Evolution Strategy (CMA-ES)

CMA-ES (Hansen, 2016) is an evolutionary algorithm used for optimisation via search over a non-smooth, non-convex solution space over a continuous domain. The CMA-ES is a gradient-free method, making it well-suited for such as problem. Through the step-size we can prevent premature convergence and a large population helps avoid local optima. Here, unlike the Random Search method and other evolutionary algorithms, we do not need a very large population. In our experiments, we use $\lambda = 100$ for Random Map-Elites and $\lambda = 25$ for CMA-ES and CMA-ME.

It is assumed that the individuals are drawn from a multivariate normal distribution $\mathcal{N}(\mathbf{m}, \Sigma) = \mathbf{m}^{(g)} + \sigma^{(g)}\mathcal{N}(\mathbf{0}, \boldsymbol{C}^{(g)})$, at a given generation (iteration) number g. Here the covariance matrix $\Sigma = (\sigma^{(g)})^2\boldsymbol{C}^{(g)}$. σ can be viewed as the overall deviation and can be interpreted as the step-size in the iterative algorithm.

We also define learning rates for updating of the covariance matrix (c_μ, κ...), among other parameters. Algorithm 1 gives a simplified overview of the updating rules for mean and covariance values.

Algorithm 1 CMA-ES Algorithm

```
Define: evolutionary search parameters $\lambda, \mu, \kappa_1, \kappa_2, \sigma, c_\mu$. Intialise: the mean vector $\mathbf{m}^{(0)}$ and covariance matrix $\boldsymbol{C}^{(0)} = \boldsymbol{I}$. $g \leftarrow 1$
while not converged do
    Sample search points $\{\mathbf{x}_k^{(g+1)}\}_{k=1}^{\lambda}$ for the given generation $g$.
        $z_k \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
        $\mathbf{y}_k = z_k \sim \mathcal{N}(\mathbf{0}, \boldsymbol{C})$
        $\mathbf{x}_k^{(g+1)} = \mathbf{m} + \sigma\mathbf{y}_k \sim \mathcal{N}(\mathbf{m}^{(g)}, (\sigma^{(g)})^2\boldsymbol{C}^{(g)})$
        $k = 1, ..., \lambda$
    (Here $\boldsymbol{B}, \boldsymbol{D}$ are obtained from the eigendecomposition of $\boldsymbol{C}$ by Spectral Theorem).
    Select $\mu$ points from the sample, in decreasing order of fitness (i.e. increasing order of objective function) as $\{\hat{\mathbf{x}}\}_{i=1}^{\mu}$. Here $f(\hat{\mathbf{x}}_1) \leq f(\hat{\mathbf{x}}_2) \cdots \leq f(\hat{\mathbf{x}}_\mu)$ Update the mean of the search distribution
        $\mathbf{m}^{(g+1)} = \mathbf{m}^{(g)} + \gamma \sum_{i=1}^{\mu} (w_i \hat{\mathbf{x}}_i^{(g+1)} - \mathbf{m}^{(g)})$
    where $\sum_{i=1}^{\mu} w_i = 1$, and $w_i > 0$ are arranged in non-increasing order. $\gamma$ is the learning rate.
    Further, the covariance matrix is updated by:
        $\boldsymbol{C}_\mu^{(g+1)} = (1+\kappa_1)\boldsymbol{C} + \kappa_2 + c_\mu \sum_{i=1}^{\lambda} w_i \mathbf{y}_{i:\lambda} \mathbf{y}_{i:\lambda}^T$
    where
    $g \leftarrow g + 1$
end while
```

4.1.3 Covariance Matrix Adaptation MAP-Elites (CMA-ME)

This algorithm (Fontaine et al., 2020) combines the CMA generation strategy and refines the Evolutionary Strategy by adding a notion of *directionality* to the update rules. When drawing a new individual,

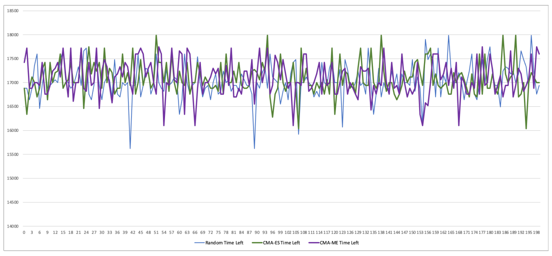

Figure 4: Results - remaining time vs iterations

we perturb an existing elite from the archive, as done in CMA-ES. In addition to that, a perturbation directed towards a second elite is also added. It can be viewed as emitting a new solution towards other good solutions so explore the search space better. The direction is determined by an emitter which decides which other solutions to prioritise.

5 Evaluation of generated levels

There are two requirements to be met by the generated content - firstly, the generated levels must be new while having similar properties to existing human-generated game levels. This is met by the objective function characteristic of the GAN architecture. The second goal is for the level to be playable by a human. Ideally, this will be answered by human annotators, but here that is infeasible, thus we adopt the convention to using standard methods such as playing against an A* agent. One common measure of playability is the amount of time left after the A* agent is done playing the game. Any non-zero value indicates that the agent was able to complete the level and the higher the value the easier the level is to finish.

We observe that all three of the search methods behave similarly over the iterations when searching for playable levels. The observed variation is to be expected as in any heuristic search, and is greatest for random search (in blue).

6 Data Sources

We use the *Mario-AI-Framework* as the data source for training our generative model, which is an open-source repository that helps with the seamless deployment of AI models with a version of Super Mario Bros. It has a java implementation of the classic Super Mario Bros., which can be used in conjunction with various AI agents provided with the framework to run real-time simulations of user-generated Super Mario levels. The framework fetches levels from text files with 17 possible level

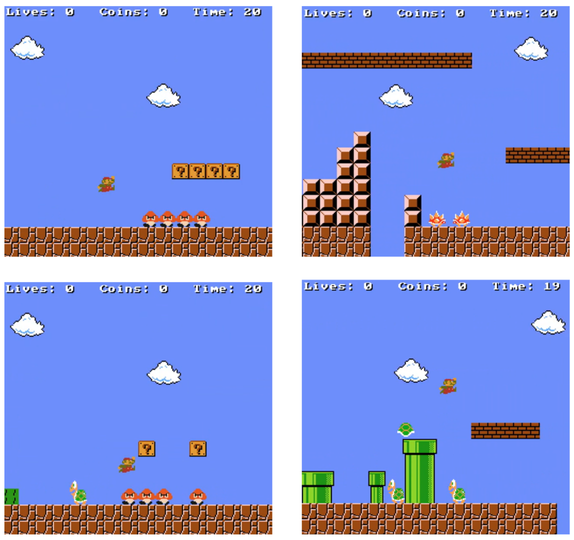
Lives: 0 Coins: 0 Time: 20
Lives: 0 Coins: 0 Time: 20
Lives: 0 Coins: 0 Time: 20
Lives: 0 Coins: 0 Time: 19

Figure 5: Some of the levels synthesized by the generator after querying with optimized latent vectors.

tiles encoded as ASCII characters. This provides a straightforward mechanism of running levels generated by our model by simply querying the generator with a latent vector optimized by a search technique and parsing the level to the required format. It also provides feedback from the agent which helps us to gather relevant parameters required for various latent space search techniques, such as level completion status, number of jumps, enemies killed, coins collected etc.

7 Conclusion

In this project, we have built a framework for generating human-playable levels for Super Mario Bros. using Deep Convolutional Generative Networks enhanced by latent space exploration techniques. We were able to deploy a DCGAN model, different from the earlier implementations that succeeded in efficiently generating new levels for the game. We applied different latent search exploration methods with the aim of enhancing the quality of generated levels by finding an optimized latent vector for querying to the generator. Through rigorous experimentation vis-à-vis the architecture and training parameters of the model as well as the mechanics of the latent space search models, we arrived at a framework that is able to generate diverse and human-playable levels without them being extremely contrived and unplayable.

Through the latent space search process, we observed that while different search techniques led to slight variations in the level design, the overarching design characteristics of the levels were similar across the different techniques. This can be attributed to the fact that the generator is able to synthesize high-quality levels without any latent space optimization.

References

Martin Arjovsky, Soumith Chintala, and Léon Bottou. 2017. Wasserstein gan.

Matthew C Fontaine, Ruilin Liu, Ahmed Khalifa, Jignesh Modi, Julian Togelius, Amy K Hoover, and Stefanos Nikolaidis. 2021. Illuminating mario scenes in the latent space of a generative adversarial network. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 5922–5930.

Matthew C. Fontaine, Julian Togelius, Stefanos Nikolaidis, and Amy K. Hoover. 2020. Covariance matrix adaptation for the rapid illumination of behavior space. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*. ACM.

Edoardo Giacomello, Pier Luca Lanzi, and Daniele Loiacono. 2018. Doom level generation using generative adversarial networks. In *2018 IEEE Games, Entertainment, Media Conference (GEM)*, pages 316–323. IEEE.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. In *Advances in Neural Information Processing Systems*, pages 2672–2680.

Matthew Guzdial and Mark O Riedl. 2016. Game level generation from gameplay videos. In *AIIDE*, pages 44–50.

Nikolaus Hansen. 2016. The cma evolution strategy: A tutorial.

Nikolaus Hansen and Andreas Ostermeier. 2001. Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2):159–195.

Alec Radford, Luke Metz, and Soumith Chintala. 2015. Unsupervised representation learning with deep convolutional generative adversarial networks.

Noor Shaker, Julian Togelius, and Mark J. Nelson. 2016. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer.

Adam M Smith and Michael Mateas. 2011. Answer set programming for procedural content generation: A design space approach. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):187–200.

Sam Snodgrass and Santiago Ontañón. 2014. Experiments in map generation using markov chains. In *FDG*.

Adam Summerville and Michael Mateas. 2016. Super mario as a string: Platformer level generation via lstms. *arXiv preprint arXiv:1603.00930*.

Adam Summerville, Sam Snodgrass, Matthew Guzdial, Christoffer Holmgård, Amy K Hoover, Aaron Isaksen, Andy Nealen, and Julian Togelius. 2018. Procedural content generation via machine learning (pcgml). *IEEE Transactions on Games*, 10(3):257–270.

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):172–186.

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 *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 221–228. ACM.