Generating Playable Mario Levels from Agent Experience

Pranav Srivastava, Arije Mami, Andres Aranguren, Nguyen Quoc An, and Yousef Swed

Department of Computer Science, Leiden University
`p.srivastava@umail.leidenuniv.nl`, `a.mami.2@umail.leidenuniv.nl`, `a.aranguren@umail.leidenuniv.nl`, `s4561341@vuw.leidenuniv.nl`

Abstract. This project explores the intersection of procedural content generation and reinforcement learning within the domain of Super Mario Bros. We propose a dual pipeline framework where conditional generative models-specifically a DCGAN and a diffusion model-are trained to generate level segments conditioned on empirically derived difficulty scores. These scores are computed using performance metrics from reinforcement learning agents (Dueling DQN and PPO) trained to complete various levels in a custom Mario environment. The generative models are trained on symbolic level representations and output new, playable levels of controlled complexity. A quality function aggregates agent performance data to label level difficulty, forming the basis for conditioning. Experimental results show that diffusion-based models outperforms only the MLP in producing diverse and coherent level layouts, while PPO and Dueling DQN effectively learn to navigate these levels, enabling iterative level evaluation and generation. This work demonstrates a scalable approach to content generation in games by coupling learning agents with generative models for difficulty-aware level design.

1 Introduction

The field of artificial intelligence (AI) has seen remarkable advancements in recent years, particularly in fields such as Generative AI (GenAI) and Reinforcement Learning (RL)[4]. Generative AI is a subfield concerned with the artificial generation of data. Specifically, Generative AI focuses on the generation of new content by learning the underlying patterns within existing data. Implementations of GenAI such as Generative Adversarial Networks and Diffusion models are especially important in content generation such as images and videos [3].

In contrast, Reinforcement Learning focuses on decision-making and goal-based learning. This field involves agents that learn by interacting with their environment, and improve their decisions based on feedback and rewards from the environment. From robotics [10], to energy management [7], and healthcare [5], RL has seen significant applications in a variety of domains, due to it's strengths in sequential decision problems.

Particularly, both GenAI [6] and RL are applied in the domain of video games, which have become a cornerstone for research and testing [2]. Video games provide a controlled environment by stripping away the unpredictability of the real world, making content generation and agent learning easier. This controlled setting makes video games ideal for exploring how generative models can create new content, and how reinforcement learning agents can learn to navigate and interact with such environments. A popular video game for research is Super Mario Bros, where researchers have explicitly applied generative AI and reinforcement learning techniques.

A particular research paper of interest, is that of Volz, which explores the generation of Super Mario Bros levels using Deep Convolutional Generative Adversarial Networks, and the conditioning of the level difficulty using evolutionary algorithms [11].

This project aims to build on top of the work done by Volz et al. [11] by exploring Super Mario Bros level generation using GANs and diffusion models, and conditioning the difficulty of these levels using trained RL agents. The remainder of this report details our concept overview in relation to previous work, design and implementation, and the experiments and results of our work. Finally we discuss the implications of our research, and outline potential future work.

2 Concept Overview

2.1 Related Work

This project is based on the work proposed by [11], which trained a Deep Convolutional GAN (DCGAN) on Super Mario levels. To guide generation toward desired properties, they applied a Covariance Matrix Adaptation Evolution Strategy (CMA-ES) with a fitness function based on playability, using a scripted agent that follows a fixed, pre-defined algorithm and does not learn from interaction with its environment. This agent exhibits deterministic, rule-based, and non-adaptive behavior. While effective, this approach relies on hand-crafted fitness functions and a non-learning agent. In contrast, we propose a method that replaces the evolutionary search with a reinforcement learning (RL) agent trained to play the game. Additionally, we employ both conditional GANs and conditional diffusion models, the latter offering greater training stability and sample diversity—especially beneficial when dealing with structured data like game levels [1]. Our approach enables the generation of levels conditioned directly on empirically derived difficulty scores, producing content with controllable complexity.

2.2 Lecture Materials

Our project builds upon core concepts discussed during the Modern Game AI Algorithms course, and relates to at least three different lectures.

First, the project aligns with the principles introduced in **Lecture 1 on Procedural Content Generation (PCG)** concepts. The main use of PCG is to automatically generate game content with minimal human input. In our project we apply this idea by using GANs to autonomously generate new Super Mario Bros levels.

Second, the project relates to methods discussed in **Lecture 2 on Learning and Optimization**, where various machine learning techniques for gameplay were discussed. Specifically, we implement two reinforcement learning agents using Dueling Deep Q-Network, and PPO. The agents are trained to navigate the generated levels, thus serving both as a testbed for evaluating the quality of the generated levels, and as an example of how optimization algorithms can be applied in a game environment.

Finally, both the GAN-based level generator and CNN-based Mario agents are directly inspired by **Lecture 6 on Learning from Pixels**. These models were trained on symbolic representations extracted from each Mario level. Learning from the spatial relationships between the pixels (level tiles) allowed the GAN-based generator to generate synthetic levels, while also allowing the RL Mario agents to make good decisions based on the input pixels.

3 Methodology

For this project we propose a pipelines reinforcement learning and generative modeling to enable controllable Super Mario level generation based on gameplay complexity. First we start with an agent trained to play a set of Mario levels until it can complete a significant fraction of them. Once trained the agent, is evaluated on a separate validation set of levels using a custom quality score function that considers gameplay metrics, including total completion time, number of movements, and number of trial attempts required to finish each level. These metrics are used to compute a difficulty score for each level , creating a dataset formed by the tuples (level, score) pairs. This annotated dataset is then used to train and condition a generative model, such as a GAN or diffusion model, where the difficulty score conditions the latent gaussian noise space, allowing the generator to produce levels at specified difficulty level. The proposed model pipeline is describe in Figure 1.

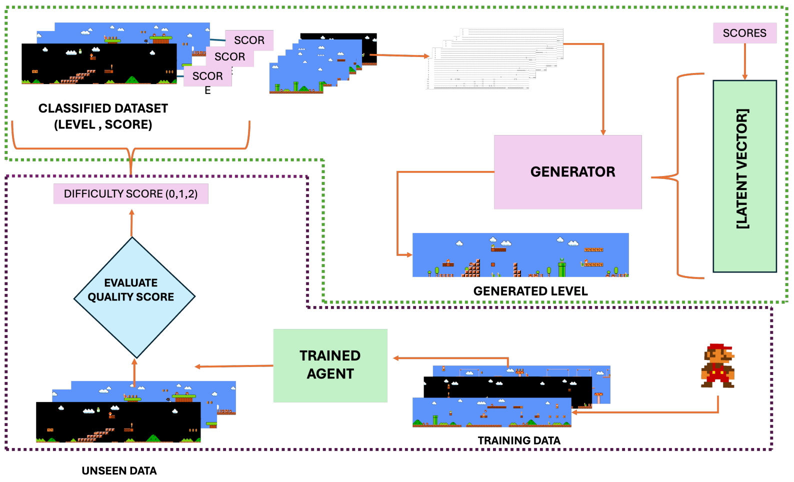

Fig. 1. Model architecture. Divided in two main sections Agent based learning used to train and evaluate level difficulty. Followed by level generation using difficulty conditioning, in generative models GAN or diffusion models. This framework anbles the generation of complexity based levels using empirical agent performance data

3.1 Levels processing

To enable training and evaluation of generative models on Super Mario levels, we implement a multi-stage symbolic processing pipeline that abstracts visual data into structured representations. Levels are initially stored as symbolic text files, where each character corresponds to a specific tile type (e.g., ground, enemy, pipe). These symbolic representations are then converted into identity matrices, which are then transformed into one-hot encoded vectors or dense embeddings. This abstract form is used to train conditional generative models such as GANs and diffusion models. After training, generated level outputs are decoded back from their latent vector representations into symbolic format and rendered as PNG images using a predefined tile set. These final rendered levels can be directly used by the reinforcement learning agent for evaluation and gameplay. Levels processing is depicted in Figure 2

Processing Steps:

1. **Load symbolic file:** A text-based Mario level is loaded where each character represents a tile type.
2. **Convert to identity matrix:** Each character is mapped to a unique integer ID using a predefined symbol-to-ID mapping.
3. **Embed or one-hot encode:** The ID matrix is either one-hot encoded or embedded into dense vectors for model input.
4. **Train generative model:** The abstract data is used to train a conditional generative model (GAN or diffusion).

5. **Train RL Mario agent:** The symbolic levels are used to train RL Mario agent (DuelDQN or PPO).
6. **Generate new samples:** The trained model outputs a tensor in latent space, representing a new level patch.
7. **Decode to symbolic:** The latent output is decoded back to the ID matrix and then to the original symbolic characters.
8. **Train generative model:** The trained Mario agent plays through the symbolic output of the generative models.
9. **Render to PNG:** The symbolic level together with the agent are rendered into a full-resolution PNG using tile sprites for visualization and gameplay.
10. **Render to Video:** The generated PNG frames are compiled into a video.

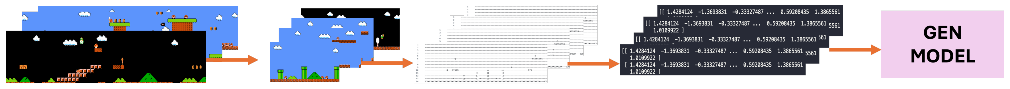

Fig. 2. Levels data processing for model training

3.2 Level Generation

Multi-Layer Perceptron (MLP) As a starting point for level generation, we implemented a simple MLP model. This model serves as a baseline, acting as a reference point to compare level generation quality with the more complex models. The MLP receives a latent noise vector as input, and outputs a flattened representation of a level patch. The patch is then reshaped into a tile-based grid, with each tile represented by unicode characters for which a mapping has been defined. While the MLP fails to take into account the spatial aspect of the data, it provides a minimal benchmark for generation using purely fully connected layers. Details of the architecture and implementation are provided in appendix section 9.1.

3.3 Generative Adversarial Network (GAN)

To build on the limitations of the simple MLP model, we implemented a Deep Convolutional Generative Adversarial Network (DCGAN). Similar to the MLP, the DCGAN generator takes a latent noise vector as input. However, instead of using fully connected layers, the DCGAN uses a series of transposed convolutional layers to iteratively transform the noise into a structured level patch. The benefit of using convolutional layers is that it introduces an inductive bias which helps preserve spatial relationships in the data. This is particularly important for level generation, where the placement of tiles are extremely important. This architecture allows the generator to create outputs where patterns are preserved, to resemble actual Mario levels. The architecture of the DCGAN is described in detail in the appendix section 9.1.

3.4 Diffusion based generative model

Diffusion models are a class of generative models that learn to reverse a gradual noising process applied to data. During training, noise is progressively added to input samples over a series of timesteps. The model is trained to predict and remove this noise, thereby learning how to generate new data from random noise. This denoising process is modeled as a sequence of conditional distributions that approximate the data distribution.

In our implementation, we use two core modules. The `SimpleDenoiseNet` is a U-Net-like neural network architecture that takes a noisy input along with a time embedding and predicts the added noise. It uses residual blocks and time-conditioned layers to capture spatial and temporal dependencies. The second component, `GaussianDiffusion`, carries the forward and reverse diffusion process. It handles the noise scheduling, sampling, and training logic using either standard diffusion. Together, these modules enable us to generate Super Mario level patches by sampling from a learned noise distribution and progressively denoising it to form coherent level structures. Figure 3, illustrates the diffusion model architecture

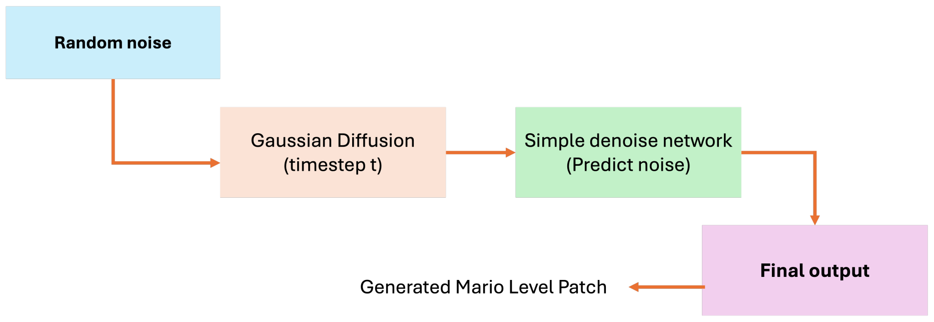
Random noise
Gaussian Diffusion (timestep t)
Simple denoise network (Predict noise)
Final output
Generated Mario Level Patch

Fig. 3. Diffusion model architecture

3.5 Level conditioning

To enable controllable level generation, we condition our generative model using difficulty labels derived from agent-based evaluations. Each training sample is represented as a tuple of the level and its corresponding difficulty score, which is discretized into classes (easy, medium, hard). During training, this difficulty label is embedded and concatenated to the noise vector. At generation time, a random noise vector is sampled and combined with the desired difficulty embedding, enabling the model to create the target level complexity. The process for conditioning the level generation is illustrated in Figure 4

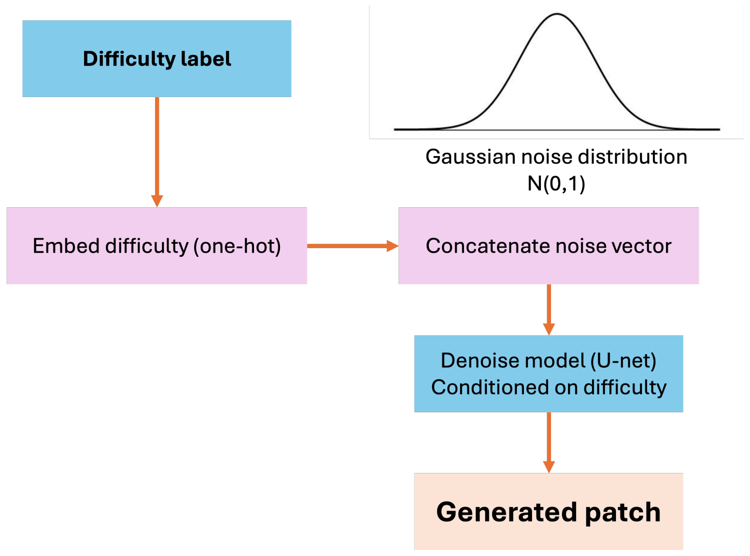

Fig. 4. Conditioning levels scheme for generative models

4 Level Interaction & Playability

4.1 Agent interaction with environment

With the output of the generators being in the form of text files, we needed a way for the Mario agent to interact with the environment. As we could not find an existing easy solution that would create an interactive environment from an external input, either by text or image files, a decision was made to hand code an environment, or engine, that resembles the Gym library by OpenAI. This custom environment allowed the agent to read level layouts from text files, interpret them as navigable game worlds, and receive observations and rewards.

The environment is a simple engine that roughly models core platformer physics and interactions. Gravity continuously influences the agent's vertical movement, while jumping provides an upward impulse, contingent on the agent being grounded. Collision detection is fundamental: the agent's movement is constrained by solid blocks, and specific interactions with enemies are handled. Falling onto an enemy eliminates it and yields a positive reward, whereas any other contact with an enemy results in a lethal outcome and a penalty. The reward system is engineered to guide agent behavior, offering substantial positive rewards for reaching the level's end or checkpoints, smaller rewards for progressive horizontal movement, and significant penalties for lethal encounters or falling off the map. Episode termination occurs upon goal achievement, death, or reaching a maximum step limit, with a reset mechanism preparing the environment for subsequent training episodes.

The agent is part of a rolling window (frame). Each time the agent takes an action, in other words an environment step, the environment makes all the necessary calculations and returns the next frame and rewards.

4.2 Qualtity function

To quantify the agent performance on the levels we defined a function that calculates a weighted score using different metrics to measure the agents score as an indicator of level of difficulty. For each test level, the agent was given 100 attempts. The maximum reward R, which is a combination of progression through level, number of enemies killed, penalty for moving back, penalty for falling off or dying to enemies, was recorded. The number of retries n to get to the maximum reward was also recorded. μ denotes the average reward for all the retries . The quality score is calculated as follows:

$$score = R - 0.3n + 0.8\mu \tag{1}$$

4.3 Models / Deep Mario DQN Agent

We experimented with two architectures to create our agent, namely Dueling Deep Q-Network and Promixmal Policy Optimisation.

Dueling Deep Q-Network (DQN) :

Dueling DQN is an extension of the Deep Q Network architecture, aimed at effective learning the values of states and the relative importance of action [12]. It achieves this by decomposing the Q-value into two separate streams: the state value function $V(s)$ and the action advantage function $A(s, a)$.

The core innovation of Dueling DQN lies in its network structure. Instead of a single sequence of layers directly outputting Q-values for each action, the Dueling DQN architecture features two parallel streams of fully connected layers after the initial convolutional (or other feature extraction) layers.

Value Stream: This stream estimates the scalar state value function $V(s; \theta, \beta)$, which represents how good it is to be in a particular state s. Here, θ denotes the parameters of the shared feature learning layers, and β denotes the parameters unique to the value stream.

Advantage Stream: This stream estimates the advantage $A(s, a; \theta, \alpha)$ for each action a in state s. The advantage function quantifies how much better or worse taking a specific action is compared to the average action in that state. Here, α denotes the parameters unique to the advantage stream.

These two streams are then combined to produce the final Q-values. A crucial aspect of this combination is ensuring identifiability – meaning that given Q, we cannot uniquely recover V and A. To address this, the advantage function is typically normalized. A common aggregation method is:

$$\begin{aligned} Q(s, a; \theta, \alpha, \beta) =& V(s; \theta, \beta) \\ &+ (A(s, a; \theta, \alpha) - \max_{a' \in \mathcal{A}} A(s, a'; \theta, \alpha)) \end{aligned} \tag{2}$$

where $|\mathcal{A}|$ is the number of possible actions. This formulation forces the average advantage to be zero, ensuring that the advantage estimates do not "compete" with the value estimate and improving learning stability.

The separation of value and advantage functions offers several benefits:

Learning State Value Independently: The Dueling DQN can learn the value of a state without needing to evaluate the effect of each action in that state. This is particularly useful in states where the choice of action has little or no impact on the environment or future rewards. In such scenarios, the advantage stream values will be small, and the Q-value will be primarily determined by the state value $V(s)$. Standard DQN, in contrast, would have to learn high Q-values for all actions that lead to a good state, or low Q-values for all actions that lead to a bad state, even if the actions themselves are inconsequential.

Improved Sample Efficiency: By decoupling the state value from the action advantages, the network can generalize learning about the state's value across multiple actions. If the network learns that a state is valuable (or not), this information directly updates $V(s)$, which then influences the Q-values of all actions in that state. This leads to more efficient learning from experience, especially when many actions have similar (or negligible) effects.

Proximal Policy Optimization (PPO) For this project we use the clipped surrogate objective variant of PPO introduced by Schulman et al. (2017) [9]. As an on-policy actor-critic algorithm, PPO maintains sample efficiency while avoiding excessive policy updates through a theoretically-grounded clipping mechanism that bounds the policy gradient step size. The method's stability and empirical performance make it particularly suitable for environments with sparse rewards and high-dimensional observations.

The agent is an actor-critic network with shared convolutional layers, which processes the environment's grid-like observations. The convolutional encoder extracts spatial features which are then fed into two separate fully connected heads:

- The actor outputs a categorical distribution over discrete actions.
- The critic estimates the state-value function $V(s)$.

This architecture allows the agent to simultaneously learn a policy and value function from high-dimensional input. During training, the agent interacts with the environment over fixed-length rollouts (2048 steps). At each step, an action is sampled by the policy using the categorical distribution, and the following data is stored:

- Observation
- Action
- Log-probability of the action
- Value estimate

– Reward and done flag

For training targets computation, we use Generalized Advantage Estimation (GAE), which reduces the variance of policy gradient estimates while introducing minimal bias [8], using the following formula:

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + (\gamma\lambda)^2\delta_{t+2} + \cdots \tag{3}$$

where $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$. The final returns used for the training of the critic are computed by adding the advantage estimates to the value predictions. The policy and value networks are updated using the clipped surrogate objective, which prevents excessively large updates.

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t\left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\right] \tag{4}$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the probability ratio between new and old policies. This objective encourages updates that improve performance while limiting deviation from the previous policy. The total loss is:

$$\mathcal{L} = L^{\text{CLIP}}(\theta) + c_1 \cdot \text{MSE}(V_\theta(s_t), R_t) - c_2 \cdot \mathcal{H}[\pi_\theta] \tag{5}$$

where c_1 and c_2 are coefficients for the value loss and entropy bonus, respectively. Entropy regularization encourages exploration by penalizing certainty.

Collected data is batched and used for multiple epochs (default is 4) of stochastic gradient descent. The loss is optimized using Adam optimizer, and gradients are clipped to a maximum norm of 0.5 to stabilize learning. Models are then evaluated using average episode return and periodically saved. The full pseudocode for our implementation is given in.

5 Experiments

5.1 Level complexity classification

After training the reinforcement learning agent to play Mario levels, we evaluated its performance on a separate set of unseen levels using a quality function. This function aggregates gameplay metrics such as completion time, number of movements, and the number of retries needed to complete a level. The resulting score serves as a proxy for the level's difficulty. To better interpret these results, we discretize the continuous scores into categorical difficulty labels: *Hard*(below -25), *Medium*(between -25 to 0), and *Easy*(above 0). Table 1 presents the evaluated levels, their corresponding quality scores, and the assigned difficulty levels.

5.2 Level generation

Once level complexity was evaluated as reported in Table 1, we used the classified subset to train the level generation models.

Table 1. Quality function scores and difficulty labels for evaluated levels

Level Name	Quality Score	Difficulty
level 1_1.txt	59.56	Easy
level 1_2.txt	-23	Medium
level 1_3.txt	-46.18	Hard
level 2_1.txt	9.04	Easy
level 3_1.txt	19.00	Easy
level 3_3.txt	-47.18	Hard
level 4_1.txt	-34.02	Hard

Table 2. complexity

DCGAN Before generating levels using the DCGAN, we needed to determine the optimal parameters in order to train the network. To do so, we ran an ablation study to determine the best configuration of hyperparameters. The hyperparameters we tested can be seen in Table 3.

Table 3. DCGAN Hyperparameter Search Space

Hyperparameter	Values Tested
Learning rate (Generator)	{0.0001, 0.0002, 0.0005}
Learning rate (Discriminator)	{0.0001, 0.0002, 0.0005}
Batch size	{16, 32, 64}
Latent dimension	{64, 100, 128}
Early stopping patience	{10, 15}
Minimum delta for early stop	{0.001, 0.005, 0.01}
Real label smoothing	{0.9, 1.0}
Fake label smoothing	{0.0, 0.1}
Input noise std. deviation	{0.0, 0.02}
Discriminator training frequency	{1, 2}
Generator training frequency	1 (fixed)

Considering the number of hyperparameters, it was infeasible to test every single combination in order to determine the optimal one. Specifically, for this set of hyperparameters, there exist 5184 combinations. Hence, we decided to test 50 configurations as a starting point, from which we could fine-tune and build upon to reach a configuration which would yield playable levels. The ablation results can be found in detail under the `/model/ablation_results` directory. Following the ablation study results, the best configuration was further fine-tuned in order to generate more realistic levels. The configuration of the final model can be found in appendix section 9.1.

Difussion model To train our final generative model, we used a diffusion-based architecture composed of two core modules: a denoising network and a diffusion controller. The denoising model, `SimpleDenoiseNet`, follows a U-Net-like architecture with an encoder-bottleneck-decoder structure. It incorporates residual blocks, group normalization, and SiLU activations, along with skip connections and time-step conditioning via sinusoidal embeddings. This allows the network to capture both spatial and temporal patterns necessary to reverse the diffusion process.

The diffusion mechanism itself is handled by the `GaussianDiffusion` class, which manages the forward noising process and reverse denoising sampling. It uses a linear or cosine noise schedule and supports both standard sampling and DDIM for faster inference.

Our training followed a progressive design approach, starting with a minimal version of the network and incrementally increasing complexity. We began with a shallow convolutional model to validate the data pipeline and loss convergence. We then gradually introduced deeper layers, skip connections, time conditioning, and normalized residual blocks. This progressive refinement allowed us to monitor stability and generalization at each stage while scaling up to our final architecture. Table 4 summarizes this process.

Table 4. Progressive architectural improvements to the denoising network

Version	**Changes Introduced**
V1	Simple 3-layer convolutional network, no skip connections, no time embedding.
V2	Added U-Net structure with encoder-decoder and skip connections.
V3	Introduced residual blocks and group normalization in each block.
V4	Added sinusoidal time-step embeddings and conditioned residual blocks on time.
Final	Deep U-Net with time-conditioned residual blocks, dropout, bilinear interpolation for upsampling alignment, and full integration with GaussianDiffusion and DDIM sampling.

The training loss for the final architecture is presented in Figure 9.

5.3 Agent training

Both Dueling DQN and Proximal Policy Optimisation agents were training following the same setup. All agents were trained on a subset of existing levels, for $2e6$ environment steps for each level. The score/reward increases as the agent progresses laterally through the levels, with bonus scores if the agent kills enemies, or penalties if the agent moves back or falls off the map or dies to enemies. The long training times as a result of the large state and actions spaces inherent to Mario levels put a constraint on the number of configurations we could test out. We tried out 2 architectures, namely Dueling DQN and PPO, and different network sizes for PPO, which are detailed in Table 5.

Table 5. DQN Mario agents

Hyperparameter	**Values**
Hidden layers (DuelDQN)	{1x512}
Hidden layers (PPO)	{1x256, 1x512}
Convolutional layer sizes (DuelDQN)	{16-32}
Convolutional layer sizes (PPO)	{16-32, 32-64-64}
Learning rate	{3e-4}
Batch size	{64}
Discount factor γ	{0.99}

5.4 Testing agent on generated levels

Once the generative model was trained with difficulty conditioning, we proceeded to evaluate the agent's performance on the generated levels. Specifically, the agent was tested on 5 levels generated by the DCGAN. For each level, we computed the quality function score and assessed whether it aligned with the difficulty level that was used as a conditioning input during generation. Results of the agent performance in terms of playability metrics are reported in **Results section**

6 Results

6.1 MLP

As explained in section 4.1, the MLP was an unsuitable model for training realistic, playable levels. The core issue lies in its inability to capture the spatial relationships within the data. Consequently, the generated level consisted of simply the floor tiles and few other tiles as obstacles. A sample level can be seen in Figure 5.

Fig. 5. Sample MLP generated level.

6.2 DCGAN Model Results

The results of the ablation study presented 4 comparable configurations. Figure 6 displays the generator and discriminator losses for the 4 best configurations.

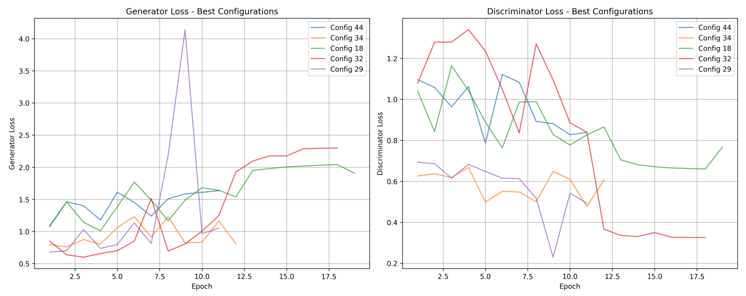

Fig. 6. Generator and discriminator losses for the 4 best configurations.

These configurations were quantified as the best based on the generator learning loss. The volatility of the losses can be attributed to several factors. GANs are notoriously difficult to train due to the adversarial aspect. Since we train two neural networks, with opposing goals, there is a significant challenge in balancing the quality of both networks. Since the networks learn simultaneously and affect each other's gradients, the losses can often fluctuate.

The instability prompted us to further improve the configuration. Configuration 34 9.3 had the smallest generator loss, and therefore was chosen to improve further upon. As seen in Figure 7, the generator and discriminator losses are a lot less volatile using the final configuration.

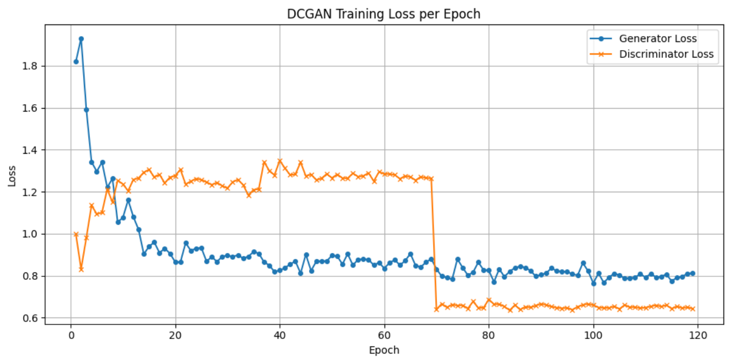

Fig. 7. The best hyperparameter configuration after fine-tuning.

The final model used a 3 phase training configuration, beginning with a relatively high learning rate, allowing it to converge quickly. In the second phase, we used the learning rates from configuration 34 of the ablation study, and in the final phase, we reduced the learning rates and increased patience to encourage the model to learn the finer details. Sample levels generated by the final DCGAN model can be seen in Figure 8, this generated levels were used to evaluate the agent performance.

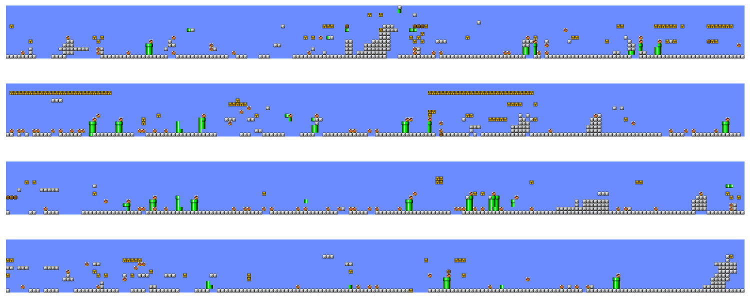

Fig. 8. Sample levels generated from the final DCGAN model.

6.3 Diffusion Model Results

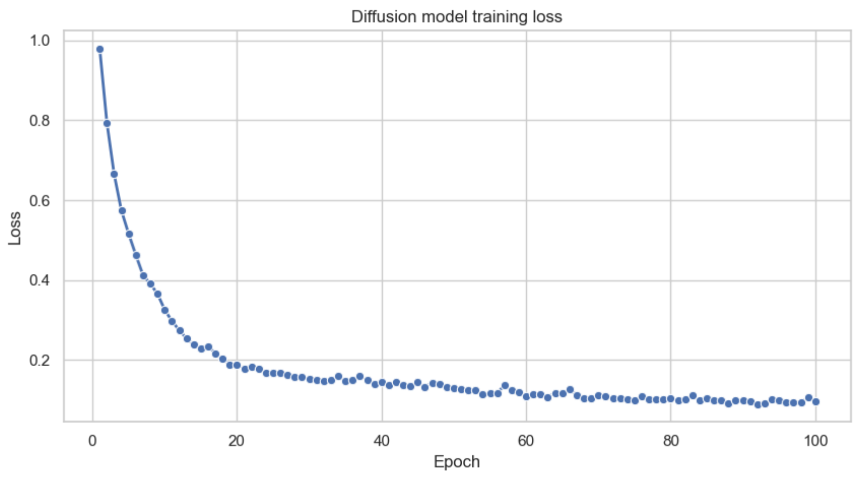

Fig. 9. Diffusion model training loss vs epochs

The training loss curve for the diffusion model demonstrates a clear downward trend, indicating effective learning over 100 epochs. Initially, the loss drops sharply, reflecting rapid improvements in the model's ability to denoise input samples. After approximately 20 epochs, the curve begins to flatten, which suggests slower convergence towards stable solution.

While the diffusion model demonstrated strong convergence during training as seen in Figure 9 the generated level text files exhibited poor structural coherence and failed to capture the spatial information of the input levels. Despite good performance in minimizing reconstruction loss, the model struggled to preserve layout constraints. A likely cause is the lack of an attention mechanism, which limits the model's ability to capture long-range spatial dependencies [13]. Additionally, there might be a problem during reconstruction step from the continuous embedding space back to discrete symbolic tiles, which may add inconsistencies.

6.4 Agent performance on generated levels

We decided to use the PPO agent as it was observed to perform better overall. The following results present the PPO agent's performance on five levels generated by the DCGAN. Each level was generated with a user-defined difficulty label. We then evaluated the agent's performance on these levels and computed the corresponding quality score to assess whether it aligned with the intended difficulty specified during generation. The agent managed to finish one of the easy generated levels and its playthrough was recorded in "Agent/playthrough". From the results we can see that the generator correctly generated 2 levels according to levels of difficulty while 3 were incorrectly generated.

Table 6. Evaluation of Agent Performance on Generated Levels

Generated Level	**Label During Generation**	**Agent Score / Difficulty**
level_1.txt	Easy	1080.64 (Easy, finished level)
level_2.txt	Medium	-46.19 (Hard => Mismatch)
level_3.txt	Hard	-79.44 (Hard)
level_4.txt	Easy	-4.12 (Medium => Mistmatch)
level_5.txt	Medium	5.07(Easy => Mismatch)

7 Conclusion

GANs proved to be the most effective model for level generation in our experiments, producing coherent and playable levels. The reinforcement learning agent, trained using PPO, was able to navigate and complete most of the generated levels successfully, validating the quality of the generated levels. Although the user-defined difficulty labels did not always align perfectly with the quality scores derived from agent performance, we successfully demonstrated conditional generation based on empirical gameplay data. This confirms the feasibility of using agent experience to guide content generation. In contrast, the diffusion model underperformed, likely due to the absence of attention mechanisms and challenges in accurately converting from continuous embeddings back to discrete symbolic representations.

8 Future Work

8.1 Level generation

The current diffusion model operates without an attention mechanism, which limits its capacity to capture long-range dependencies and spatial hierarchies within the level structure. Introducing attention layers such as self-attention or cross-attention could help the model better preserve important spatial relationships and contextual information during generation. This would likely result in levels that are more coherent, structurally sound, and visually consistent, especially in complex or large-scale levels.

Secondly, levels are rendered from embedding outputs, which caused incorrect symbolic representations. A proper one-hot encoding approach should be used for the diffusion model instead to ensure accurate tile placement and rendering. This method is more reliable, and is easier to convert back into the image file.

8.2 Mario agent

While our PPO agent with a CNN-based Actor-Critic architecture achieved stable training and reasonable performance on the symbolic Mario environment, several directions can be explore to improve efficiency of learning, generalization and semantic understanding in the future.
Our current convolutional architecture learns spatial patterns but ignores the symbolic and semantic nature of the level layout. An possible direction for future work is the integration of semantic information into the model. This would involve encoding symbolic meanings of the tiles (e.g., ”X”: [”solid”, ”ground”], ”o”: [”coin”, ”collectable”, ”passable”], etc.) into structured or learned embeddings that are fused into the CNN processing pipeline.
More sophisticated PPO variants could also be implemented such as PPO with adaptive KL penalties, which automatically balances policy updates and divergence [9].

–

References

1. Croitoru, F.A., Hondru, V., Ionescu, R.T., Shah, M.: Diffusion models in vision: A survey. IEEE Transactions on Pattern Analysis and Machine Intelligence **45**(9), 10850–10869 (2023)
2. Fan, X.: The application of reinforcement learning in video games. In: 2023 International Conference on Image, Algorithms and Artificial Intelligence (ICIAAI 2023). pp. 202–211. Atlantis Press (2023)
3. Goodfellow, I.J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., Bengio, Y.: Generative adversarial nets. Advances in neural information processing systems **27** (2014)

4. Hagos, D.H., Battle, R., Rawat, D.B.: Recent advances in generative ai and large language models: Current status, challenges, and perspectives (2024), `https://arxiv.org/abs/2407.14962`
5. Jayaraman, P., Desman, J., Sabounchi, M., Nadkarni, G.N., Sakhuja, A.: A primer on reinforcement learning in medicine for clinicians. NPJ Digital Medicine **7**(1), 337 (2024)
6. Mao, X., Yu, W., Yamada, K.D., Zielewski, M.R.: Procedural content generation via generative artificial intelligence. arXiv preprint arXiv:2407.09013 (2024)
7. Ponse, K., Kleuker, F., Fejér, M., Serra-Gómez, Á., Plaat, A., Moerland, T.: Reinforcement learning for sustainable energy: A survey. arXiv preprint arXiv:2407.18597 (2024)
8. Schulman, J., Moritz, P., Levine, S., Jordan, M., Abbeel, P.: High-dimensional continuous control using generalized advantage estimation (2018), `https://arxiv.org/abs/1506.02438`
9. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347 (2017)
10. Singh, B., Kumar, R., Singh, V.P.: Reinforcement learning in robotic applications: a comprehensive survey. Artificial Intelligence Review **55**(2), 945–990 (2022)
11. Volz, V., Schrum, J., Liu, J., Lucas, S.M., Smith, A., Risi, S.: Evolving mario levels in the latent space of a deep convolutional generative adversarial network. In: Proceedings of the genetic and evolutionary computation conference. pp. 221–228 (2018)
12. Wang, Z., Schaul, T., Hessel, M., Van Hasselt, H., Lanctot, M., De Freitas, N.: Dueling network architectures for deep reinforcement learning. arXiv preprint arXiv:1511.06581 (2016)
13. Yang, L., Zhang, Z., Song, Y., Hong, S., Xu, R., Zhao, Y., Zhang, W., Cui, B., Yang, M.H.: Diffusion models: A comprehensive survey of methods and applications. ACM Computing Surveys **56**(4), 1–39 (2023)

9 Appendix

9.1 MLPGenerator Architecture (`mlp_model.py`)

The `MLPGenerator` is a feedforward neural network implemented using PyTorch's `nn.Module`.

Initialization Parameters

- **patch_height**: The height of the level patch to be generated (in tiles).
- **patch_width**: The width of the level patch to be generated (in tiles).
- **n_tile_types**: The number of unique tile types in the dataset (e.g., ground, brick, question block), defining the depth of the one-hot encoding.
- **latent_dim** (default: 32): Dimensionality of the random input noise vector (latent space), serving as the seed for generation.
- **hidden_dim** (default: 256): Number of units in the first hidden layer. Deeper layers scale up based on this.

Network Structure The core of the generator is a sequential model (`self.main`) consisting of fully connected layers and non-linear activations:

- **Input Layer**: Takes a latent vector of size `latent_dim`.
- **First Hidden Layer**:
  - `nn.Linear(latent_dim, hidden_dim)`
  - `nn.LeakyReLU(0.2, inplace=True)`
- **Second Hidden Layer**:
  - `nn.Linear(hidden_dim, hidden_dim * 2)`
  - `nn.LeakyReLU(0.2, inplace=True)`
- **Third Hidden Layer**:
  - `nn.Linear(hidden_dim * 2, hidden_dim * 4)`
  - `nn.LeakyReLU(0.2, inplace=True)`
- **Output Layer**:
  - `nn.Linear(hidden_dim * 4, output_size)`
    where `output_size = patch_height * patch_width * n_tile_types`
  - `nn.Sigmoid()`

Forward Pass (`forward` method)

- **Input**: A batch of latent vectors `z` with shape `(batch_size, latent_dim)`.
- **Processing**: `z` is passed through `self.main`.
- **Output**: Output is reshaped to `(batch_size, patch_height, patch_width, n_tile_types)`, representing probabilities across tile types at each position.

Generation Helper Methods

- `generate_patch(batch_size, device)`: Generates random noise `z`, passes it through the generator, and returns the result as a NumPy array.
- `generate_symbolic_patch(processor, batch_size, device)`: Converts generated output into symbolic representation using a `ProcessDataSymbolic` instance.
- `generate_whole_level(level_tile_width, processor, device)`: Generates and stitches patches horizontally to form a larger level, then converts it to symbolic format.

9.2 DCGAN Architecture

The Deep Convolutional Generative Adversarial Network (DCGAN) consists of two primary components: the **Generator** and the **Discriminator**. The Generator learns to produce realistic game levels from random noise, while the Discriminator learns to distinguish between real and generated samples.

Generator Purpose: Generate synthetic game levels from a latent noise vector.
Architecture:

- **Input:** A latent vector sampled from a Gaussian distribution (size: *latent_dim*).
- **Fully Connected Layers:**
  - Three linear (dense) layers.
  - Each layer is followed by:
    - *Batch Normalization*
    - *ReLU* activation
    - *Dropout* for regularization
  - Final dense output is reshaped into a 2D feature map.
- **Upsampling via Transposed Convolutions:**
  - Series of transposed convolutional layers to progressively increase spatial resolution.
  - *ReLU* activations for hidden layers.
  - Final layer uses a *Sigmoid* activation to constrain values to $[0, 1]$.
- **Output:** A 2D game level. Output size is interpolated if needed to match target dimensions.

Discriminator Purpose: Classify inputs as real (from the dataset) or fake (from the generator).
Architecture:

- **Input:** A 2D game level sample (real or generated).
- **Convolutional Layers:**
  - Three convolutional layers for feature extraction.
  - Each layer uses:
    - *LeakyReLU* activation
    - *Dropout* for regularization
- **Fully Connected Layers:**
  - Flattened feature map is passed through two dense layers.
  - Final output layer uses a *Sigmoid* activation to produce a probability.
- **Output:** A scalar probability in $[0, 1]$ indicating the likelihood of the input being real.

9.3 Configuration 34

```
{
  "config_id": 34,
  "config": {
    "lr_g": 0.0002,
    "lr_d": 0.0002,
    "batch_size": 16,
    "latent_dim": 128,
    "max_epochs": 50,
    "patience": 10,
    "min_delta": 0.005,
    "real_label": 1.0,
    "fake_label": 0.1,
    "noise_std": 0.02,
    "d_train_freq": 2,
    "g_train_freq": 1
  },
  "metrics": {
    "final_g_loss": 0.808269522612608,
    "final_d_loss": 0.6059932652153547,
    "min_g_loss": 0.7582089817033538,
    "training_epochs": 12,
    "training_time": 50.866103172302246,
    "g_loss_std": 0.1702565866384097,
    "d_loss_std": 0.06346503201338247,
    "mode_collapse": false,
    "generation_success": true,
    "character_diversity": 4,
    "repetition_score": 0.8,
    "quality_score": 20.933978159318126
  },
  "g_losses": [
    0.7956755557392217,
    0.7582089817033538,
    0.8743282330187061,
    0.8017201482943153,
    1.0625834117961834,
    1.230973008053401,
    0.9125136924506743,
    1.2310042065462168,
    0.8208696834648712,
    0.8308456042144872,
    1.1614530908910534,
    0.808269522612608
  ],
  "d_losses": [
    0.626060162163988,
    0.6366631962076018,
    0.618229480483864,
    0.6677830694596979,
    0.4984484133841116,
    0.5512264699121064,
    0.5479290191131302,
    0.5002824637708785,
    0.6485213414777683,
    0.6097472921202455,
    0.4772930431969558,
    0.6059932652153547
  ]
```