Hacettepe University

Computer Engineering Department

BBM479/480 End of Project Report

Project Details

Title	InDunGen (Infinite Dungeon Generator)
Supervisor	Assoc. Prof. Dr. Burkay Genç

Group Members

	Full Name	Student ID
1	Kayla Akyüz	21726914
2	Eren Kumru	21727518
3	Başak Şükran Melahat Çontu	21827282
4		

Abstract of the Project (/ 10 Points)

Explain the whole project shortly including the introduction of the field, the problem statement, your proposed solution and the methods you applied, your results and their discussion, expected impact and possible future directions. The abstract should be between 250-500 words.

Currently, the game development industry is increasingly adopting artificial intelligence technologies for a range of applications, from 3D model generation to image rendering. We share a strong interest in both the game development industry and artificial intelligence, and we are keenly aware of the exciting possibilities at the forefront of this technology frontier. As such, we aimed to familiarize ourselves with these cutting-edge, and ultimately, we aspired to develop a product that utilizes Generative Adversarial Networks (GANs) to generate procedurally generated maps for games.

GANs refer to a machine learning technique that has the ability to produce new outputs by utilizing a set of existing training examples [12]. This approach can be particularly useful in the context of Procedural Content Generation (PCG) [11] for video game levels, especially when there is an existing collection of levels that can be used as a reference for emulation.

Our project focuses on training game levels using GANs to generate new levels for NetHack, a popular rogue-like game. The main problem we address is the level design can be time-consuming and expensive. We wanted to see if we can train a model to generate levels for a game. Even though NetHack uses a procedural map generation algorithm which is appreciated for its ability to generate engaging and unpredictable levels, we wanted to test if we can achieve this with a GAN.

To overcome this challenge, we propose a solution that utilizes GANs to learn the patterns and structures of existing NetHack levels and generate new, diverse levels that maintain the game's complexity and unpredictability. We tried to train NetHack levels with different kinds of GAN models and compare their training time, accuracy and playability of the generated levels.

Our dataset of existing NetHack levels was crucial in training the GAN models, and we found that increasing the size of the dataset led to improved results in terms of level quality.

We experimented with different GAN architectures and hyperparameters. One of the best-performing models was the Nvidia StyleGAN2 [23]. One of the disadvantages of this model was that its training time took too long which is because it processes images as data and due to our lack of necessary hardware capabilities to work with it efficiently [13]. AdaIN GAN [24] was also better than other GAN models but generated levels were not as successful as Style GAN. It did not take as much time as Nvidia's StyleGAN2 because we were able to modify it to take the input data as an unsigned char array rather than images. Although generated levels were not good enough to play, it is showcased that with further research it is achievable to utilize Generative Adversarial Networks (GANs) to generate procedurally generated maps for games.

The GAN models had promising results for generating new levels for games but the generated levels for NetHack were not very playable and did not meet the wanted quality.

Despite the challenges, the potential impact of our work is significant, as it offers game developers a novel approach to level design that can enhance the replayability and

engagement of players. Furthermore, our methodology can open the door to a holistic new approach to Procedural Content Generation and can be extended to other game genres.

Introduction, Problem Definition & Literature Review (/ 20 Points)

Introduce the field of your project, define your problem (as clearly as possible), review the literature (cite the papers) by explaining the proposed solutions to this problem together with limitations of these problems, lastly write your hypothesis (or research question) and summarize your proposed solution in a paragraph. Please use a scientific language (you may assume the style from the studies you cited in your literature review). You may borrow parts from your previous reports but update them with the information you obtained during the course of the project. This section should be between 750-1500 words.

Introduction

The video game design and development industry has experienced significant growth in recent years, with players seeking out games that offer a unique and captivating experience. Key to achieving this is level design, which involves creating challenging and entertaining game environments for players. While traditional methods of level design, such as manual design or procedural generation algorithms, have been widely used, we wanted to harness the potentials of GANs and push the boundaries of traditional level design approaches. To address this issue, our project proposes the use of Generative Adversarial Networks to generate game levels that are diverse and complex, enhancing the overall gaming experience for the players.

Problem Statement

In video game design, the problem with using traditional manual level design or procedural generation algorithms is that they often result in levels that lack diversity and complexity, leading to repetitive gameplay and decrease in replayability. Manual level design can be time-consuming and expensive, while procedural map generation algorithms offer a more automated approach, generating levels based on existing rules or algorithms.

This problem is particularly relevant in games that rely heavily on level design, such as roguelikes, platformers and action games. In these types of games, the level design is a crucial aspect of the gameplay experience and can greatly impact the player's enjoyment of the game.

To address these issues, our project aims to leverage the power of GANs to generate game levels that are both diverse and complex, offering players a unique and engaging experience. By using GANs, we can generate levels that are more varied and unpredictable than those generated by traditional methods, resulting in improved player engagement and replayability.

Literature Review

In recent years, there have been many studies in using machine learning methods such as GAN.

In the paper which is called "Evolving Mario levels in the latent space of a deep convolutional generative adversarial network" (2018) [1], the authors proposed a method using a GAN-based approach for generating levels for the game Super Mario Bros. They trained the GAN on using the dataset of existing Mario levels consisting of images representing each tile in the level. The generator generated a new set of images which are corresponding to a new Mario level when a given random input vector. The discriminator network was trained to separate between the real

levels and the generated levels. The authors used a unique approach which is called novelty search to generate challenging and various levels. Novelty search algorithm measures the level's novelty by comparing how different it is from other existing levels rather than relying on feedback from the discriminator. One of the limitations of this research is that the computational requirements of the GAN-based approach may limit its applicability to some game development scenarios. In the end, results showed that levels which are generated from GAN were unique, challenging and better than existing procedural generation methods.

In another study "DOOM Level Generation Using Generative Adversarial Networks" (2018) [2], authors propose a GAN-based approach to generate levels for the game DOOM. First, they generated a dataset of DOOM levels and converted them to a matrix representation with pre-processing. To generate levels with respect to the game difficulty, they used a conditional GAN. They used the method called "GAN-2-PCG", which uses GAN to generate a population of levels, which are then evaluated using procedural content generation
metrics to select the best levels for the game. Similar to previous research, GAN-based approaches resulted much better than PCG algorithms in terms of quality and diversity of the generated levels. Limitation of this study is that the evaluation criteria utilized may not entirely reflect the gameplay experience, and there is a possibility to improve this aspect in future research by involving player feedback and assessments.

The article "Bootstrapping Conditional GANs for Video Game Level Generation" [3] introduces a novel approach to creating new video game levels using conditional generative adversarial networks (cGANs). The authors propose a bootstrapping technique to address the challenge of limited training data by utilizing an existing set of levels as a foundation for generating new ones.
The proposed approach involves three primary stages: preprocessing of existing levels, bootstrapping the cGAN with preprocessed levels, and generating new levels with the trained cGAN with the preprocessed levels, and generating new levels with the trained cGAN. During the preprocessing stage, the levels undergo conversion into a structured representation that can serve as input for the cGAN. Following that, the cGAN is trained on the preprocessed levels in the bootstrapping stage, then utilized to generate new levels in the generation stage, based on user-defined constraints.
The limitations of this approach are related to the availability, quality, and diversity of the pre-existing level set. Additionally, the generated levels may not always satisfy the user-defined constraints. Also, the approach is limited to generating levels within a specific game genre and may require substantial modifications to apply other genres.

The article "World-GAN: a Generative Model for Minecraft Worlds" [4] proposes a novel generative model called World-GAN for creating Minecraft worlds. The authors note that while previous work has used GANs to generate individual Minecraft structures, no one has yet created a model capable of generating entire Minecraft worlds with varying biomes, terrain, and structures. To address this problem, the authors propose World-GAN, which is designed to generate the full structure of a Minecraft world, including terrain, biomes, trees, water, and other environmental features.
The authors used a dataset of Minecraft worlds consisting of over 5000 images to train their model. World-GAN consists of two components: a terrain generator and a structure generator. The terrain generator produces the basic terrain, including elevation, biomes, and water bodies, while the structure generator generates structures such as trees, buildings and other features. These two components are trained separately and combined to generate final output.
They made experiments to evaluate effectiveness of their models and compared it to several baselines like a random generation model and a previous GAN-based method to generate

Minecraft structures. They found that World-GAN performed the baselines better in terms of realism of generated worlds and visual quality.
One limitation of this work is that they did not conduct user studies to evaluate playability of the generated worlds. Also, they note that this model may not be suitable for generating very large Minecraft worlds due to memory constraints.

The article “Illuminating the Space of Beatable Lode Runner Levels Produced By Various Generative Adversarial Networks” [5] researches using GANs to generate playable levels for the game Lode Runner. They aim to examine the capacity of different GAN models in generating levels that are not only visually appealing but also enjoyable and challenging to play. The proposed solution consists of training multiple GANs with a dataset of Lode Runner level. These trained GAN models are evaluated based on playability and quality of generated levels. They analyzed other various features like placement of enemies, presence of traps and overall difficulty.
One of the limitations of this work is the reliance on a specific game dataset, which may limit the generalizability of the findings to other games. Also, the measurement is subjective and based on authors’ own ratings of the levels.

The paper “Game Sprite Generator Using a Multi Discriminator” [6] focuses on generating high-quality game sprites, which are important in many video games. The proposed solution consists of training the GAN with data of game sprites. Generator learns to generate new sprites with respect to patterns and styles extracted from training data. The multiple discriminators are employed to define different aspects of the generated sprites like color, shape and texture. Feedback from the discriminator used to help the generator to generate sprites better. Generated sprites show diverse styles, shapes colors according to characteristics of the training data. But one limitation of this approach is the reliance on a constant dataset of game sprites for training which may limit the ability of the model to generate sprites with content different from the training set.

In paper “Toad-GAN: Coherent Style Level Generation from a Single Example” [7] explains generating consistent levels in video games with a single example as input. TOAD-GAN consists of a generator and discriminator network just like other GANs. The generator takes a single example level as input data and generates new levels that are consistent with the input. The discriminator determines the generated levels and sends feedback to improve the generator. The authors of this paper compare TOAD-GAN with other methods and show it has more advantages in terms of level consistency and quality. But TOAD-GAN has its limitations. Since the TOAD-GAN relies on one example level it can not generate unique levels. Also, again the evaluation of the generated levels focuses on visual consistency and there is no evaluation with respect to player experience.

In “A Case Study of Generative Adversarial Networks for Procedural Synthesis of Original Textures in Video Games” [8] presents using GANs for the procedural synthesis of original textures in video games. Proposed solution consists of training a GAN model with a dataset of textures and using this model to create new textures. It aims to create original and unique textures. The results show that generated textures using GAN are visually appealing, appropriate for game environments and unique. But it also states that this approach needs a large dataset to make sure that the model learns different visual patterns and styles. Another limitation is the computational complexity and need of resources to train GAN models.

In “Procedural 3D Terrain Generation using GANs” [9] focuses on using GANs for procedural generation of 3D terrains. The authors explain the challenge of creating unique and realistic

terrains for different applications like video games. Existing methods usually result in recurring or narrow variations in the generated terrains. GANs propose an approach to handle these limitations. The proposed solution involves training GAN with data of real life terrain samples. In the training process, the generator network improves the ability to generate realistic terrains which can fool the discriminator. Results show that a GAN-based approach can generate visually satisfied and unique terrains. But one limitation with this approach is the need for large and various datasets to train the GAN model.

In paper "A Dynamic Balanced Level Generator for Video Games Based on Deep Convolutional Generative Adversarial Networks" [10], they use deep convolutional generative adversarial networks for generating balanced levels in games. In the proposed solution, aim is to leverage the power of DCGANs to generate dynamic and balanced video game levels. They collect a dataset of game levels which include both balanced and imbalanced levels to train the DCGAN model. After the training process, the model can generate new levels with similar to balanced levels from the training dataset. Results show that the DCGAN-based approach was successful in generating dynamic and balanced levels which are more enjoyable gameplay experience compared to traditional methods. But one limitation is the computational complexity of training levels. Another limitation is the reliance on existing data of game levels which may limit the uniqueness of the new generated levels.

The paper "Generative Adversarial Network Rooms in Generative Graph Grammar Dungeons for The Legend of Zelda" [14] combines a GAN approach to generating individual rooms with a graph grammar approach to combining rooms into a dungeon. The GAN captures design principles of individual rooms, but the graph grammar organizes rooms into a global layout with a sequence of obstacles determined by a designer. Room data from The Legend of Zelda is used to train the GAN. This approach is validated by a user study, showing that GAN dungeons are as enjoyable to play as a level from the original game, and levels generated with a graph grammar alone. However, GAN dungeons have rooms considered more complex, and plain graph grammar's dungeons are considered least complex and challenging. Only the GAN approach creates an extensive supply of both layouts and rooms, where rooms span across the spectrum of those seen in the training set to new creations merging design principles from multiple rooms.

Overall, these studies present encouraging outcomes, but they have restrictions in their generalization to particular game types and do not consider the complex and unpredictable nature of some types of games. Moreover, GAN-based techniques demand significant computational resources and an extensive dataset for training.

Hypothesis

Our hypothesis is that GAN-based methods can create game levels that are varied and complex, which outperform current procedural generation techniques. To achieve this, we suggest utilizing GANs, which can learn from established game levels' patterns and structures to generate fresh levels that preserve the game's complexity and unpredictability.

Proposed Solution

Our proposed solution involves these following steps:

GANs Research:
Starting to investigate Generative Adversarial Networks and their categories. Reading relevant literature like papers and articles to amass knowledge and insights about GANs, their use cases or speeds. Cherry picking the ones that best fit for our project's aim.

GANs Test with Basic Inputs:

After selecting the GAN types, testing them with basic inputs before testing with game levels for evaluation of results and time measurements. These tests are helpful for analyzing the GANs and selecting the best ones for the project.

Data Collection and Preprocessing:
Next step involves gathering game levels from a dungeon game and preparing them to develop datasets which are suitable for training the GANs.

Training the GANs:
Training the GANs on the processed datasets to produce fresh game levels. This involves identification of appropriate hyperparameters, optimization algorithms and loss functions for the GANs. Analyze the GANs' performance during the training phase and make necessary tweaks to refine their output.

Importing Generated Levels to NetHack:
Ultimately, newly created levels are imported into NetHack to test their playability.

Methodology (/ 25 Points)

Explain the methodology you followed throughout the project in technical terms including datasets, data pre-processing and featurization (if relevant), computational models/algorithms you used or developed, system training/testing (if relevant), principles of model evaluation (not the results). Using equations, flow charts, etc. are encouraged. Use sub-headings for each topic. Please use a scientific language. You may borrow parts from your previous reports but update them with the information you obtained during the course of the project. This section should be between 1000-1500 words (add pages if necessary).

GANs Research:

Our project began with an extensive search where we looked into different types of GANs to gain understanding of the GAN models and their applications. This allowed us to explore different categories of GANs and select the ones most suitable for our project's objectives.

Here are the GAN models we have searched:

1) Vanilla GAN
   The Vanilla GAN is the original GAN architecture proposed by Ian Goodfellow in 2014. The generator takes random noise as input and generates samples while the discriminator tries to recognize real and fake samples. The generator tries to fool the discriminator and the discriminator tries to classify samples.

2) DCGAN
   DCGAN was introduced by Radford et al. in 2015, is an extension of the Vanilla GAN that utilizes deep convolutional networks (CNNs) in both the generator and discriminator. DCGAN improves stability and quality of generated samples compared to Vanilla GAN. It replaces fully connected layers with convolutional layers, employs batch normalization, and uses convolutional transpose layers for upsampling in the generator. These architectural choices lead to more stable training.

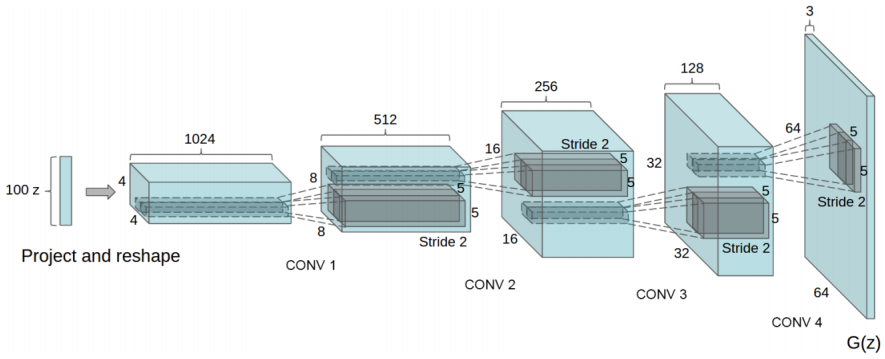
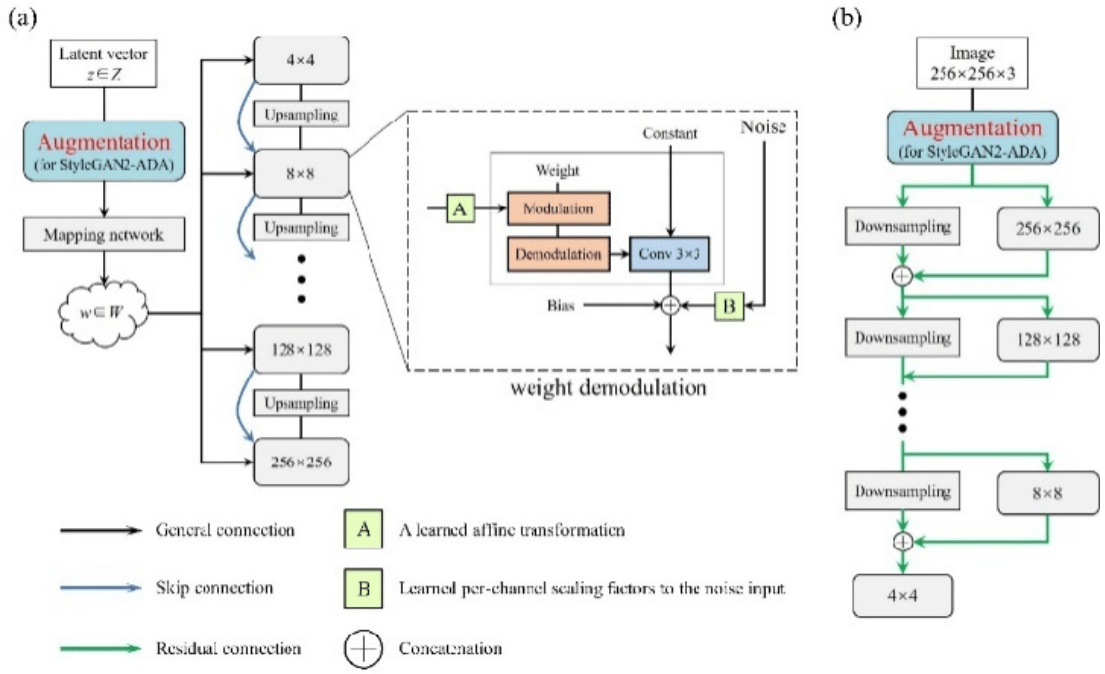


Figure 1: DCGAN generator used for LSUN scene modeling. A 100 dimensional uniform distribution Z is projected to a small spatial extent convolutional representation with many feature maps. A series of four fractionally-strided convolutions (in some recent papers, these are wrongly called deconvolutions) then convert this high level representation into a 64×64 pixel image. Notably, no fully connected or pooling layers are used.

3) AdaIN GAN
AdaIN GAN, proposed by Huang et al. in 2017, introduces adaptive instance normalization into the GAN framework. AdaIN is a normalization technique that aligns the mean and variance of feature maps to match those of a given style image. This allows the generator to transfer style characteristics from a style image to the generated samples. It achieves better control over the output appearance compared to DCGAN.

4) NVIDIA's StyleGAN2
StyleGAN2 is developed by Karras et al. 2019. It is a more advanced GAN model that addresses several limitations of previous ones. StyleGAN2 introduces the concept of disentangled representations, enabling separate control over different aspects of the generated images. StyleGAN2 employs progressive growing, where it gradually increases the resolution of generated images during training. It also consists of a new regularization technique which is path regularization. It helps generators to explore more different regions of the latent space.



PMGAs Research:

The project continued with research to find the optimal procedural map generation algorithm used in a well-known game. This allowed us to explore the methods and techniques used in procedural map generation and the differences between the resulting maps. We evaluated the PMGAs in terms of accessibility, performance and complexity, and the resulting maps in terms of diversity, quality, quantity, ease of access and being fun and selected the optimal one for our project's objectives.

Here are the PMGAs we have searched:

1) Enter The Gungeon

   Enter the Gungeon is a gunfight dungeon crawler game created and developed by Dodge Roll [16] and published by Devolver Digital [17]. The developers define the game as: “Enter the Gungeon is a gunfight dungeon crawler following a band of misfits seeking to shoot, loot, dodge roll, and table-flip their way to personal absolution by reaching the legendary Gungeon’s ultimate treasure: the gun that can kill the past.” [15]. Set in the firearms-themed Gungeon, gameplay follows four player characters called Gungeoneers as they traverse procedurally generated rooms to find a gun that can "kill the past".

   To explain without going into details, this is how procedural map generation in Enter the Gungeon works [18]: The individual rooms are pre-authored and populated with a wide variety of enemies, the general layouts called “flows” are also carefully pre-authored. There is a huge list of rooms – nearly 300 for only the first stage – and a handful of flows per stage – fewest with 4 and most with 8. The process starts with a randomly picked flow file which is a directed graph data structure that stores the relationships between the rooms but not the positioning. Then the flow file is transformed in a few different ways. Then, some extra nodes are “injected”. This feature is quite flexible and is used for all sorts of purposes. Each injection contains data specifying what and where to insert, probability of spawning and any pre-conditions that must be met. It is also at this point that the generator picks a specific room for each node and it avoids picking the same room twice. There are a few more tweaks and tricks to completing the flow and after the flow is completed the process moves on with generating composites and then assembling the final map.

   We decided not to use Enter the Gungeon’s procedural map generation method since the source code is not directly accessible and developing such a complex algorithm is quite time consuming.

2) Hauberk

   Hauberk is a roguelike, an ASCII-art based procedurally generated dungeon crawl game developed by Bob Nystrom.

   Starting with a stage of solid walls, the random dungeon generator works like so [19]:

   1. Place a number of randomly sized and positioned rooms. If a room overlaps an existing room, it is discarded. Any remaining rooms are carved out.

   2. Any remaining solid areas are filled in with mazes. The maze generator will grow and fill in even odd-shaped areas but will not touch any rooms.

3. The result of the previous two steps is a series of unconnected rooms and mazes. We walk the stage and find every tile that can be a "connector". This is a solid tile that is adjacent to two unconnected regions.

4. We randomly choose connectors and open them or place a door there until all the unconnected regions have been joined. There is also a slight chance to carve a connector between two already-joined regions, so that the dungeon isn't single connected.

5. The mazes will have a lot of dead ends. Finally, we remove those by repeatedly filling in any open tile that's closed on three sides. When this is done, every corridor in a maze leads somewhere.

The end result of this is a multiply-connected dungeon with rooms and lots of winding corridors.

There are many flaws in Hauberk's algorithm. It tends to produce annoyingly windy passages between rooms. You can tune that by tweaking your maze generation algorithm, but making the passageways less windy tends to make them wander to the edge of the dungeon, which has its own strange look. The fact that rooms and mazes are aligned to odd boundaries makes things simpler and helps fill it in nicely, but it does give the dungeon a bit of an artificially aligned look. Thus, we decided not to use Hauberk’s procedural map generation method.

3) Dungeonator

Dungeonator is a small C library for procedural roguelike dungeon generation [20]. It's written in pure C99 to allow it to be bound to basically every language that exists. The algorithm used is a variant of Hauberk's algorithm.

It performed better than Hauberk’s in small sized dungeons, but the resulting maps were not diverse enough thus we decided to not use it.

4) NetHack

NetHack is an open-source single-player roguelike video game, first released in 1987 and maintained by the NetHack DevTeam [21]. Its dungeon spans about fifty primary levels, most of which are procedurally generated when the player character enters them for the first time. The procedural map generation algorithm of NetHack is very complex. To put it simply, it starts by room creation, finding their positions and inner contents, then adds stairs – exactly one up stair and one down stair except the bottom level, then it connects rooms with adding corridors. Connecting rooms process is done by four consecutive loops each looping through the array of rooms.
First, it loops through the array of rooms, trying to join each room to the next one in the sequence. After each join, there’s a 2% chance that it’ll stop and go to the second step.

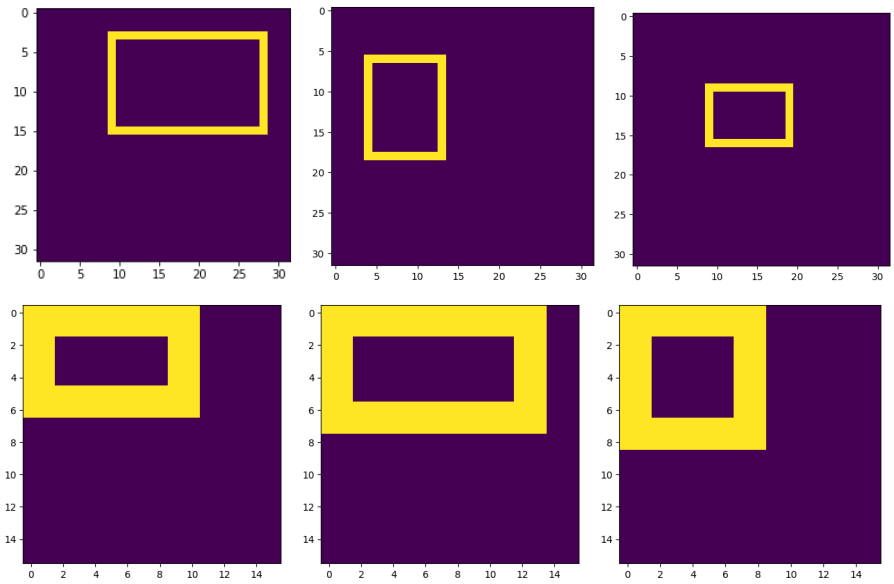
Second, it loops through the list again, this time trying to join each room to the room two steps down the array.
Third, it goes through the list of rooms, trying to join each room to every other room, but stopping once it runs out of rooms that aren't the current room.
Finally, if there are more than two rooms, it picks a random number based on the number of rooms, adds 4 to it, and then counts down, each loop attempting to link two random rooms.
As it can be seen, it attempts to add connections between nearly every room, so what is important here is the order the joins are attempted. "Join" process tries to find places to put doors and then tries to dig corridors between them. It doesn't always succeed, which the algorithm takes advantage of. Enough corridors are placed to result in NetHack's characteristic look of rooms suspended in a sea of tunnels [22].

NetHack's resulting maps were diverse, mass-producible, easily accessible and playable, and very fun to play. Since it is an open-source project the procedural generation method is completely available and accessible. Also, we could add the generated levels back to the game and test them. For these reasons, we chose NetHack's procedural map generation method as the optimal PMGA for our project.

GANs Test with Basic Inputs:

Before training the GANs models with NetHack level data, we tested them with basic inputs which are different types of generated rectangles such as ones with thick border, colored, filled and with multiple rectangles. Initial tests were conducted on Vanilla GAN, DCGAN, Style GAN and AdaIN GAN. These tests helped us to explore the performance and capabilities of each GAN model in generating outputs.

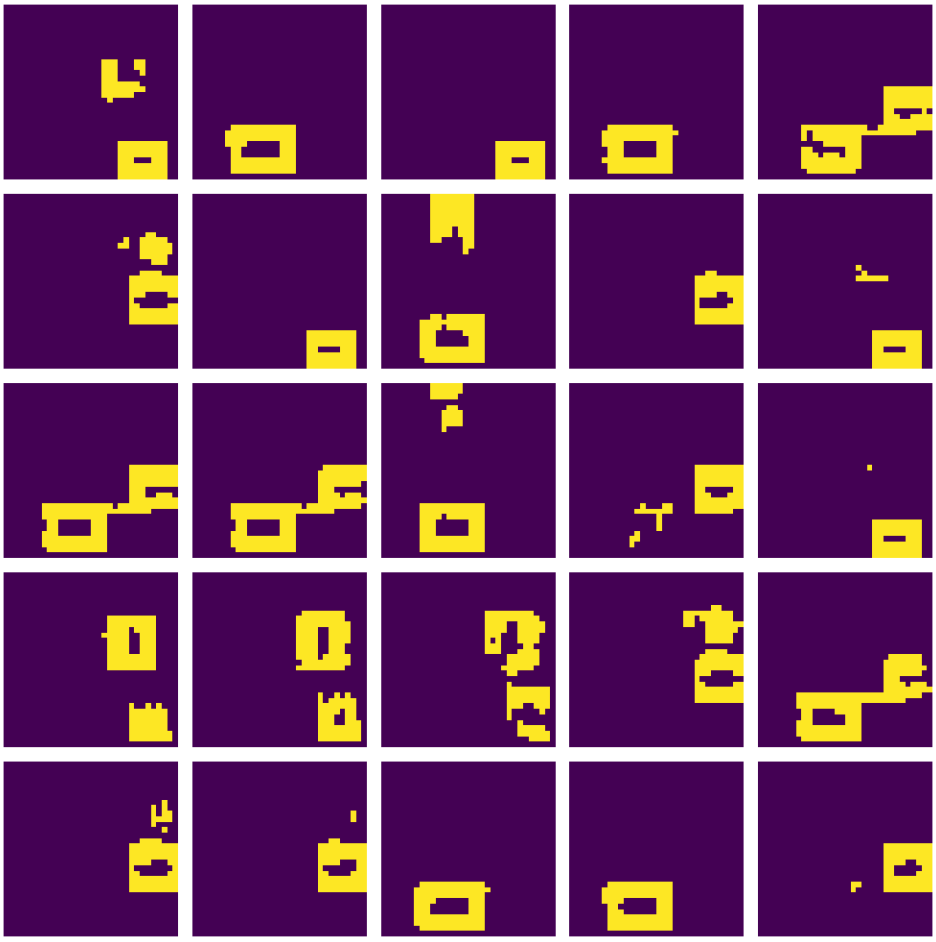
Here are some of the training data with different sizes and thickness off rectangles:



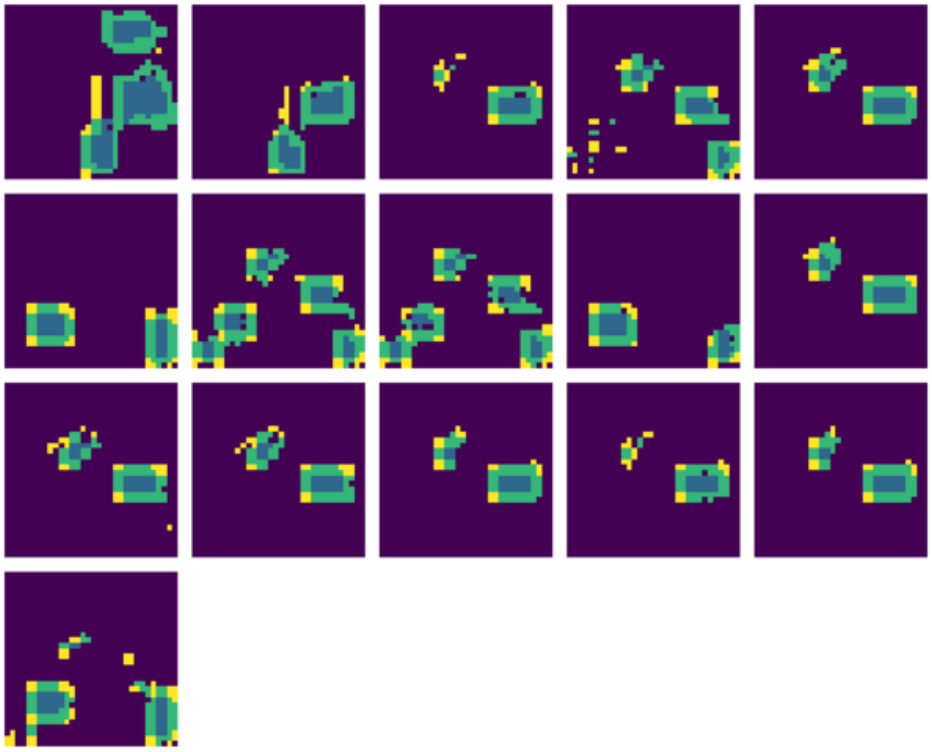
During the testing phase, we observed that Vanilla GAN and DCGAN did not have very satisfying results. They were very slow in training compared to Style GAN and AdaIN GAN thus we eliminated these models.

However, StyleGAN and AdaIN GAN didn't have very successful results. The Generated rectangles had jagged edges. This led us to question why models like Style GAN, known for their ability to recognize complex patterns like human faces, struggled with such a seemingly simple task of learning rectangular shapes.

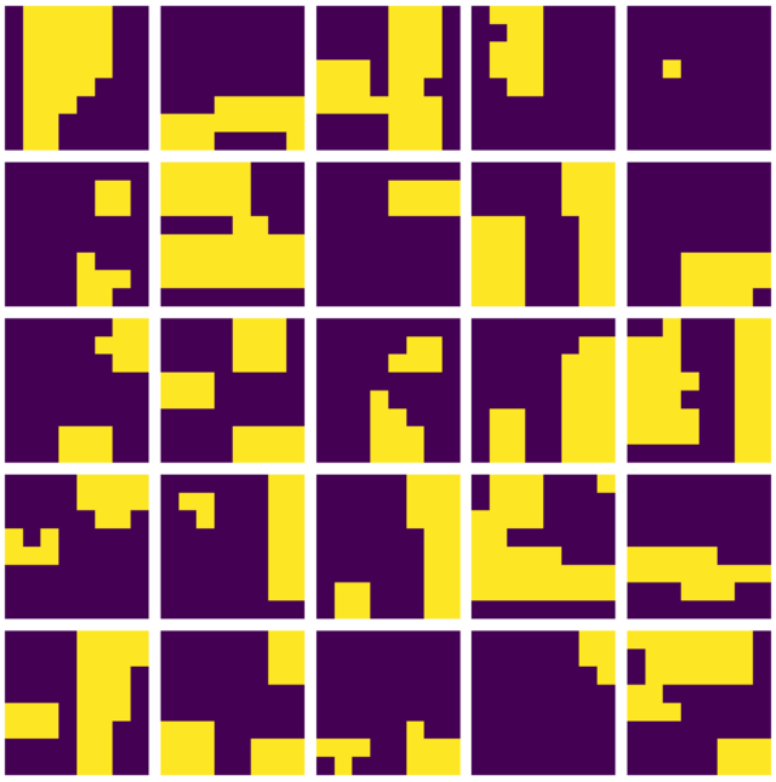
Here are some of the generated empty rectangles with thick borders from AdaIN GAN:



Results from colored filled rectangles from AdaIN GAN:

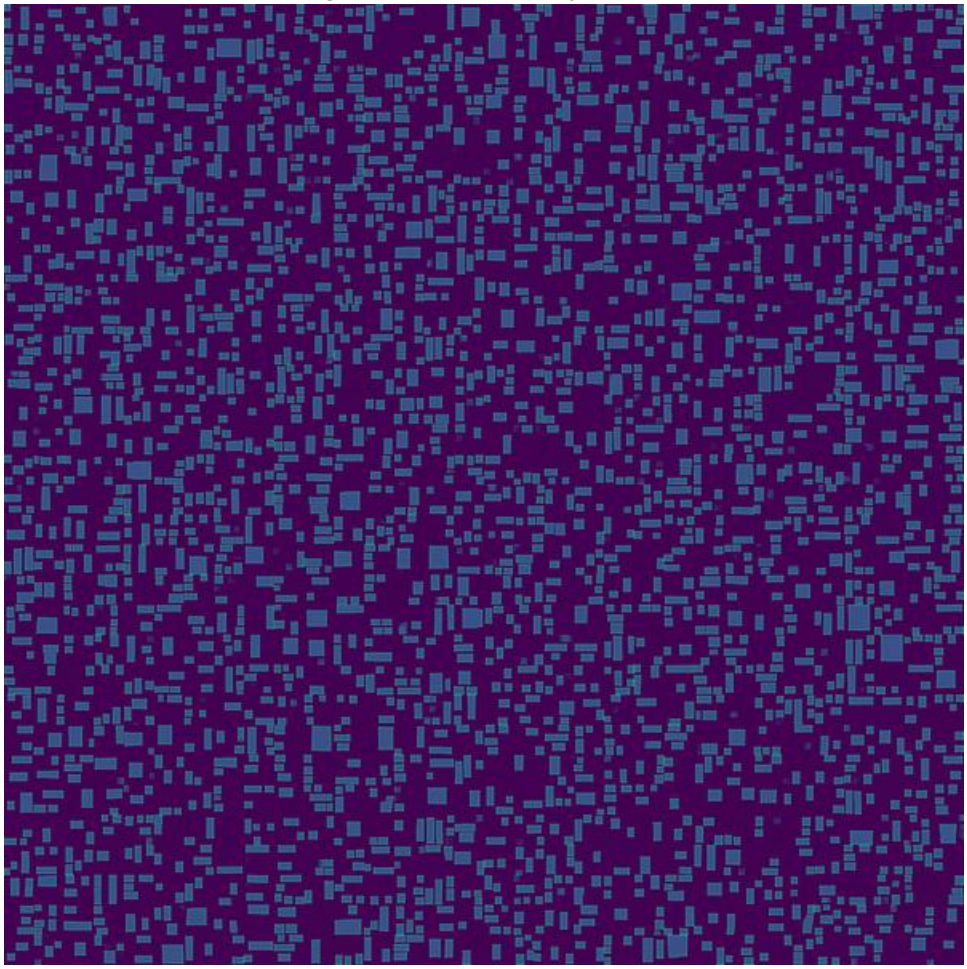


Results from filled rectangles from StyleGAN:



To address this issue we tried to test using the raw implementation of StyleGAN2-ADA from NVlabs. Remarkably, this implementation significantly improved the quality of results. The rectangles generated by StyleGAN2-ADA were much smoother, visually pleasing, and aligned with our expectations for playability.

Results from filled rectangles from Nvidia StyleGan2-ada:



Data Collection and Preprocessing

To train the GAN models for level generation in NetHack, we needed a dataset of existing levels of NetHack. We modified the source code of NetHack to export the level data and preprocessed it for training.

Modifying NetHack's source code allowed us to capture the structure and content of each game level and save it as a binary file. In NetHack's source code the function called **mklev** for generating and populating a dungeon level. The function initializes the data structures for the level. Thus we modified this function to export a large number of NetHack level data.

```
1297    void
1298    mklev(void)
1299    {
1300        reseed_random(rn2);
1301        reseed_random(rn2_on_display_rng);
1302
1303        init_mapseen(&u.uz);
1304        if (getbones())
1305            return;
1306
1307        gi.in_mklev = TRUE;
1308
1309        for (int i = 0; i < 10000; i++) {
1310            makelevel();
1311            char filename[20] = "";
1312            sprintf(filename, "levels/level_data_%d.bin", i);
1313            write_array_to_file(filename);
1314        }
1315
1316        level_finalize_topology();
1317
1318        reseed_random(rn2);
1319        reseed_random(rn2_on_display_rng);
1320    }
```

After the makelevel function is executed, the write_array_to_file function is called which we implemented to export binary level data to the files. We wanted to export 10000 levels thus they are called 10000 times. Here is the implementation of **write_array_to_file** function:

```
920     void
921     write_array_to_file(char *filename)
922     {
923         FILE *fp = fopen(filename, "wb");
924         struct rm *lev;
925         for (int y = 0; y < ROWNO; y++) {
926             for (int x = 0; x < COLNO; x++) {
927                 lev = &levl[x][y];
928                 fwrite(&levl[x][y].typ, sizeof(unsigned char), 1, fp); // Write each element to the file
929             }
930         }
931         fclose(fp);
932     }
933
```

This function stores each tileset of the generated level to the binary file with a given name.

Level data read from these binary files and reshaped to the desired dimensions and padded to make row and column size equal. Normally NetHack level data size is 21 rows and 80 columns. But for the AdaIN GAN model, it needs to take the same sizes of rows and columns. Thus we padded the arrays to make them suitable for training.

Training the GANs:
We decided to train NetHack levels with StyleGAN2 and AdaIN GAN since they were more successful than other GAN models with basic inputs.

We modified the AdaIN GAN code [24] for training with Nethack levels. To run the model we need to install TensorFlow with GPU support in Jupyter Notebook. We used Anaconda for this and created a virtual environment. After the initial setup and library imports. We begin by defining the components of the ADAIN GAN model. The generator network is responsible for generating NetHack level, while the discriminator network learns to distinguish between real and generated levels. These networks work together during the training process to improve the generator's ability to create levels.

To prepare the NetHack levels for training, we implemented data preprocessing steps. Since levels were stored as binary files we implemented a method to read and extract the data using Python.The binary data was then converted into unsigned char arrays, ensuring compatibility with the AdaIN GAN model. Additionally, we transformed the data into float32 format, which is a suitable representation for training the GAN.

Hyperparameters:
The hyperparameters for the training process, like the learning rate, batch size and number of iterations were adjusted based on results that were explained in the results and discussion section. Learning rate determines the step size at which the optimization algorithm adjusts the weights of the GAN's neural network during training. Batch size refers to the number of samples that are processed together in a single forward and backward pass during training. Number of iterations represents the total number of times the GAN model has gone through the entire training dataset.

AdaIN:
The AdaIN function in the code normalizes the input tensor by subtracting the mean and dividing by standard deviation. It also applies scale and shift operations using the gamma and beta parameters.

The Gradient Penalty Loss:
The gradient penalty loss function calculates the gradient penalty for enforcing the Lipschitz constraint on the discriminator.

Generator Model:

```
Model: "model_1"
_________________________________________________________________________________________________
Layer (type)                 Output Shape              Param #     Connected to
=================================================================================================
input_1 (InputLayer)         (None, 100)               0
_________________________________________________________________________________________________
dense_1 (Dense)              (None, 512)               51712       input_1[0][0]
_________________________________________________________________________________________________
dense_2 (Dense)              (None, 512)               262656      dense_1[0][0]
_________________________________________________________________________________________________
dense_3 (Dense)              (None, 25600)             13132800    dense_2[0][0]
_________________________________________________________________________________________________
reshape_1 (Reshape)          (None, 10, 10, 256)       0           dense_3[0][0]
_________________________________________________________________________________________________
conv2d_1 (Conv2D)            (None, 10, 10, 128)       524416      reshape_1[0][0]
_________________________________________________________________________________________________
dense_4 (Dense)              (None, 128)               65664       dense_2[0][0]
_________________________________________________________________________________________________
dense_5 (Dense)              (None, 128)               65664       dense_2[0][0]
_________________________________________________________________________________________________
lambda_1 (Lambda)            (None, 10, 10, 128)       0           conv2d_1[0][0]
                                                                   dense_4[0][0]
                                                                   dense_5[0][0]
_________________________________________________________________________________________________
activation_1 (Activation)    (None, 10, 10, 128)       0           lambda_1[0][0]
_________________________________________________________________________________________________
up_sampling2d_1 (UpSampling2D)  (None, 20, 20, 128)    0           activation_1[0][0]
_________________________________________________________________________________________________
conv2d_2 (Conv2D)            (None, 20, 20, 64)        131136      up_sampling2d_1[0][0]
_________________________________________________________________________________________________
dense_6 (Dense)              (None, 64)                32832       dense_2[0][0]
_________________________________________________________________________________________________
dense_7 (Dense)              (None, 64)                32832       dense_2[0][0]
_________________________________________________________________________________________________
lambda_2 (Lambda)            (None, 20, 20, 64)        0           conv2d_2[0][0]
                                                                   dense_6[0][0]
                                                                   dense_7[0][0]
_________________________________________________________________________________________________
activation_2 (Activation)    (None, 20, 20, 64)        0           lambda_2[0][0]
_________________________________________________________________________________________________
up_sampling2d_2 (UpSampling2D)  (None, 40, 40, 64)     0           activation_2[0][0]
_________________________________________________________________________________________________
conv2d_3 (Conv2D)            (None, 40, 40, 32)        32800       up_sampling2d_2[0][0]
_________________________________________________________________________________________________
dense_8 (Dense)              (None, 32)                16416       dense_2[0][0]
_________________________________________________________________________________________________
dense_9 (Dense)              (None, 32)                16416       dense_2[0][0]
_________________________________________________________________________________________________
lambda_3 (Lambda)            (None, 40, 40, 32)        0           conv2d_3[0][0]
                                                                   dense_8[0][0]
                                                                   dense_9[0][0]
_________________________________________________________________________________________________
activation_3 (Activation)    (None, 40, 40, 32)        0           lambda_3[0][0]
_________________________________________________________________________________________________
up_sampling2d_3 (UpSampling2D)  (None, 80, 80, 32)     0           activation_3[0][0]
_________________________________________________________________________________________________
conv2d_4 (Conv2D)            (None, 80, 80, 16)        8208        up_sampling2d_3[0][0]
_________________________________________________________________________________________________
dense_10 (Dense)             (None, 80, 80, 1)         17          conv2d_4[0][0]
=================================================================================================
Total params: 14,373,569
Trainable params: 14,373,569
Non-trainable params: 0
_________________________________________________________________________________________________
```

Input Layer: Receives a noise vector of size 100.
Dense Layers: Transform the input noise into a higher-dimensional representation.
Reshape Layer: Reshapes the output of the dense layers into a 4D tensor.
Convolutional Layers: Apply convolutions to capture features and patterns.
Dense Layers: Introduce additional information to the model.
Lambda Layers: Combine the outputs of previous layers.
Activation Layers: Introduce non-linearity to the model.
UpSampling2D Layers: Upsample the feature maps to increase resolution.
Convolutional Layers: Further process the upsampled feature maps.
Dense Layers: Introduce additional information to the model.
Lambda Layers: Combine the outputs of previous layers.
Activation Layers: Introduce non-linearity to the model.
UpSampling2D Layers: Upsample the feature maps further.
Convolutional Layer: Further process the upsampled feature maps.
Dense Layers: Introduce additional information to the model.
Lambda Layers: Combine the outputs of previous layers.
Activation Layers: Introduce non-linearity to the model.
UpSampling2D Layers: Upsample the feature maps further.
Convolutional Layer: Further process the upsampled feature maps.
Dense Layer: Output the final generated image.
The model has a total of 14,373,569 trainable parameters, which are learned during training to generate NetHack level.

Discriminator Model:

```
Model: "model_2"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_3 (InputLayer)            (None, 100)          0
__________________________________________________________________________________________________
input_2 (InputLayer)            (None, 80, 80, 1)    0
__________________________________________________________________________________________________
model_1 (Model)                 (None, 80, 80, 1)    14373569    input_3[0][0]
__________________________________________________________________________________________________
sequential_1 (Sequential)       (None, 1)            1337273     input_2[0][0]
                                                                 model_1[1][0]
==================================================================================================
Total params: 15,710,842
Trainable params: 1,337,273
Non-trainable params: 14,373,569
__________________________________________________________________________________________________
```

The discriminator model takes as input both the generated level from the generator model and a real image from the dataset. It consists of two main parts:
Input Layers:
Input Layer 1: Receives the noise vector of size 100.
Input Layer 2: Receives the real image of shape (80, 80, 1).

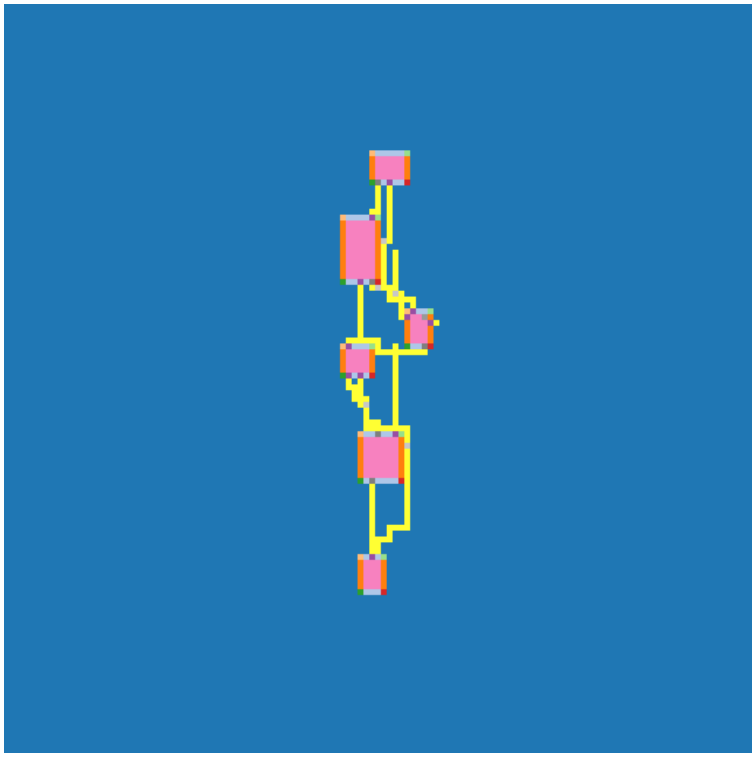
Model Layers:
Model Layer 1: The generator model takes the noise vector as input and generates a new level.
Sequential Layer: Combines the original level and the generated level for discrimination.

The discriminator model has a total of 15,710,842 parameters. But only 1,337,273 of these parameters are trainable, meaning they are updated during training to optimize the discriminator's ability to distinguish between original and generated levels.

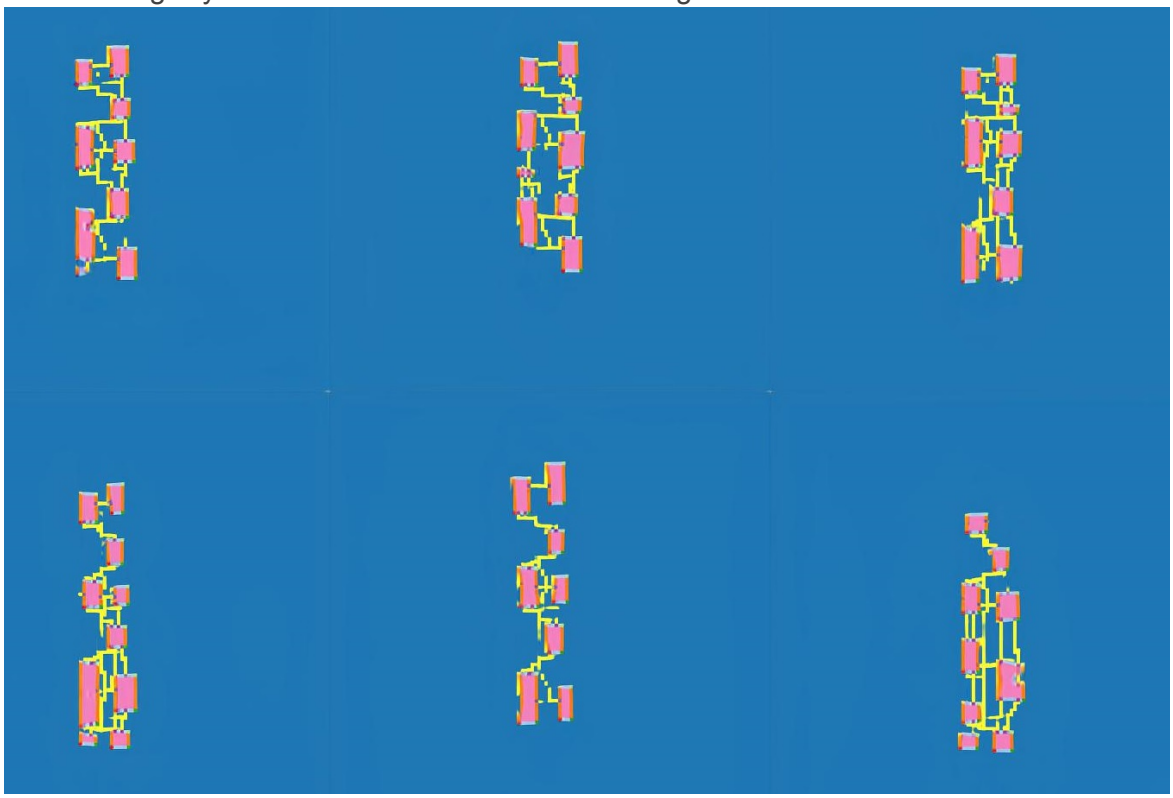
StyleGAN2
There are 4 StyleGAN versions published by Nvlabs which are StyleGAN, StyleGAN2, StyleGAN2-ADA, StyleGAN3. Each being better implementations of the previous solution with bug fixes and performance increases. StyleGAN3 addresses issues for animations and animated videos while StyleGAN2-ADA focuses on little training data. Since this project is not about generating animations but a focus on little training data would benefit, We chose StyleGAN2-ADA. [23]

We used Google's Colab to train StyleGAN2-ADA because StyleGAN2-ADA depends on Nvidia's cuda software, GPUs and Tensorflow which is accessible in Google Colab. We enabled GPU in the notebooks settings. StyleGAN2-ADA uses Tensorflow 1.14 and it doesn't support Tensorflow 2.x. But Google Colab stopped supporting Tensorflow 1 thus we had to manually install Tensorflow 1.x. The StyleGAN2-ADA training dataset should be in TFRecords thus we converted the NetHack levels which were matplot images to TFRecords dataset. During training, network pickles are exported. Google Colab sometimes disconnects because of network usage thus we had to save the model. Thus when it disconnected we resumed from the last saved .pkl file.

Also when training the StyleGAN2-ADA model, the choice of colormaps had a significant impact on the generated level images. Colormaps determine how the model maps latent space values to colors, influencing the overall appearance and visual appeal of the generated levels. Default colormap did not bring out details thus we adjusted the colormap for NetHack level images.



After training StyleGAN2-ADA here are some of the generated levels.



As we can see, the results are good but cannot say they are perfect.

We also saved the trained generator models from both AdaIN GAN and StyleGAN2-ADA to reuse them in the future for generating NetHack levels whenever desired. Saving the model ensures that the efforts put into training are preserved, enabling quick and efficient level generation without the need for training again.

Importing Generated Levels to NetHack:

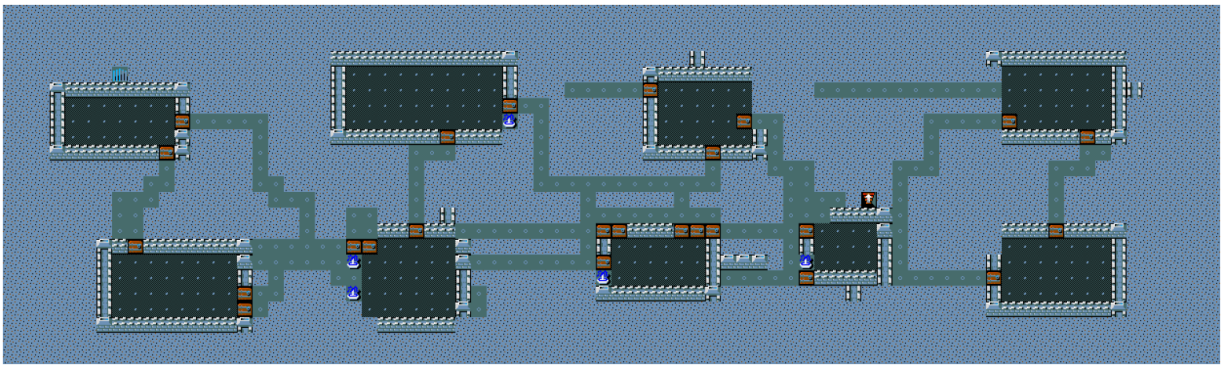
We wanted to measure the generated levels playability by visually checking them in the NetHack game environment. To able to do this we imported the generated levels into NetHack source code with this function:

```
static
void
mymakelevel()
{
    FILE *fp = fopen("output50.bin",
                     "rb"); // Open the file for reading in binary mode
    unsigned char buffer[ROWNO * COLNO];
    fread(buffer, sizeof(unsigned char), ROWNO * COLNO,
          fp); // Read the data from the file into the buffer
    fclose(fp);

    struct rm *lev;
    for (int y = 0; y < ROWNO; y++) {
        for (int x = 0; x < COLNO; x++) {
            lev = &levl[x][y];
            lev->typ = buffer[y * COLNO + x]; // Create new tiles (rm) based
                                              // on the unsigned char values
            //lev->glyph = 9608;
            lev->glyph = cmap_to_glyph(lev);
        }
    }

}
```

In the code the file named **output50.bin** is the generated file from the GAN model. It reads the data into a buffer of unsigned char values. Each element represents a position in the level. The loop iterates over each position in level, a pointer to the corresponding tile (rm) in NetHack game structure is obtained. The tile's type is assigned based on the corresponding value in the buffer. Also **with cmap_to_glyph** function, the code sets visual representation of each tile. With this function, we imported generated levels into the NetHack which helped us to observe the appearance of the levels during the gameplay.

Here is one of the imported level that is generated by AdaIN GAN:

And generated by StyleGAN2-ADA



Results & Discussion (/ 30 Points)

Explain your results in detail including system/model train/validation/optimization analysis, performance evaluation and comparison with the state-of-the-art (if relevant), ablation study (if relevant), a use-case analysis or the demo of the product (if relevant), and additional points related to your project. Also include the discussion of each piece of result (i.e., what would be the reason behind obtaining this outcome, what is the meaning of this result, etc.). Include figures and tables to summarize quantitative results. Use sub-headings for each topic. This section should be between 1000-2000 words (add pages if necessary).

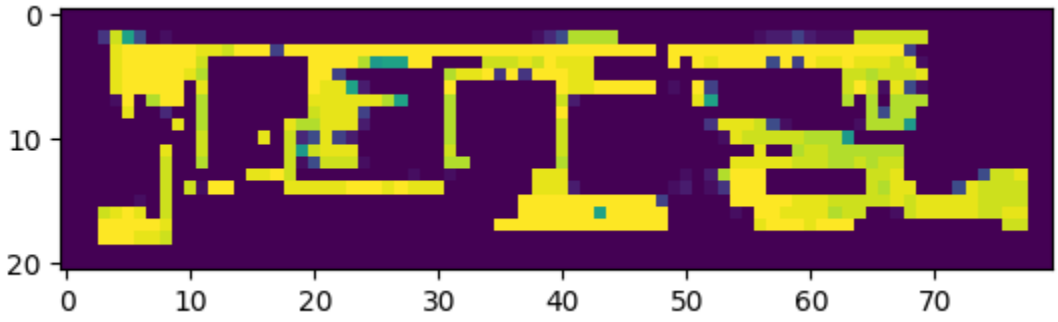
We made a detailed analysis of the results obtained from training Generative Adversarial Network models for level generation in NetHack. We tried experiments with different numbers of training iterations, batch sizes, and learning rates to assess their effect on the performance and quality of the generated levels. Here is analysis for training AdaIN GAN model:

Training Iterations Analysis

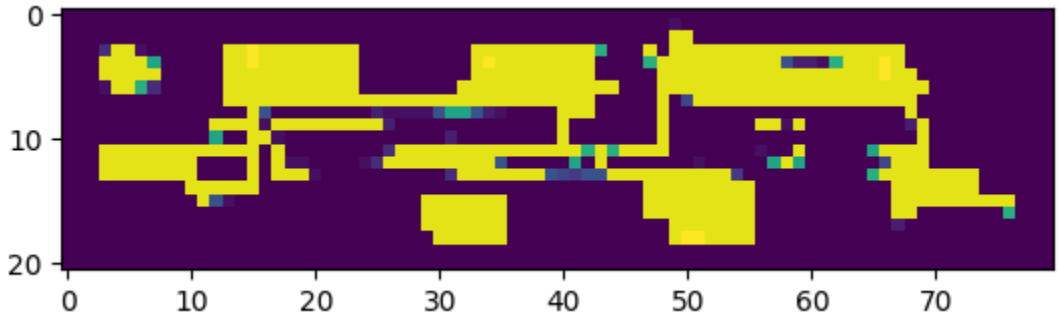
We trained the GAN model for different numbers of iterations which are 10000, 20000, 25000 and 50000. The number of iterations during training affects how well the model learns and refine their level generation capabilities. Based on our analysis, we observed that increasing the number of training iterations generally improved the quality and complexity of the generated levels. The models became better at learning the complex details and patterns that are in the training dataset. The increased training time allowed the GANs to converge to more optimal solutions, resulting in levels that exhibited greater variety and accuracy to the real NetHack gameplay. However, we also noticed a diminishing return with higher numbers of iterations, where the improvements became less significant beyond a certain point. After 50 thousand iterations, we trained the model for another 50 thousand iterations which resulted in much worse than 10 thousand iterations. The rooms were not definable thus levels were not playable.

Here are some of the generated levels with different number of iterations with same learning rate (0.0002) and batch size (64):

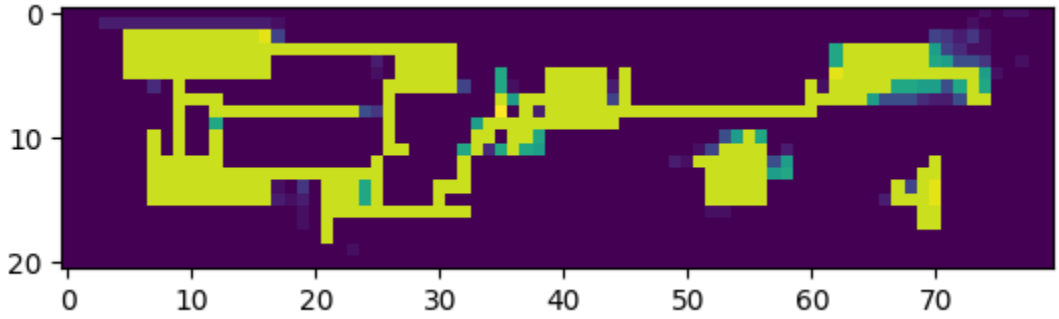
10000 iterations:



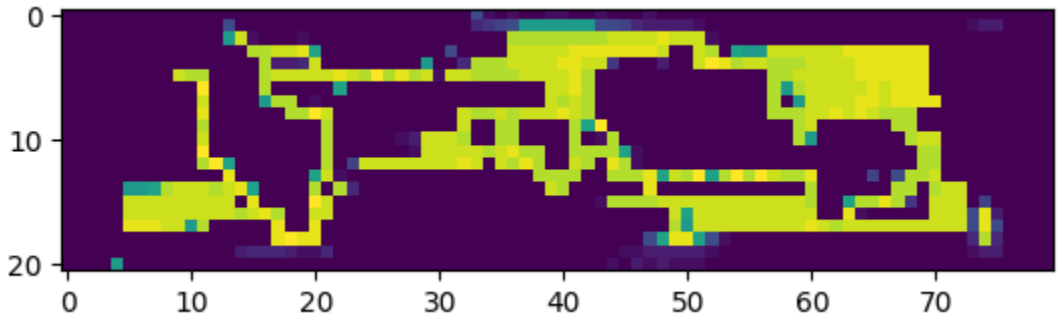
20000 iterations:



25000 iterations:

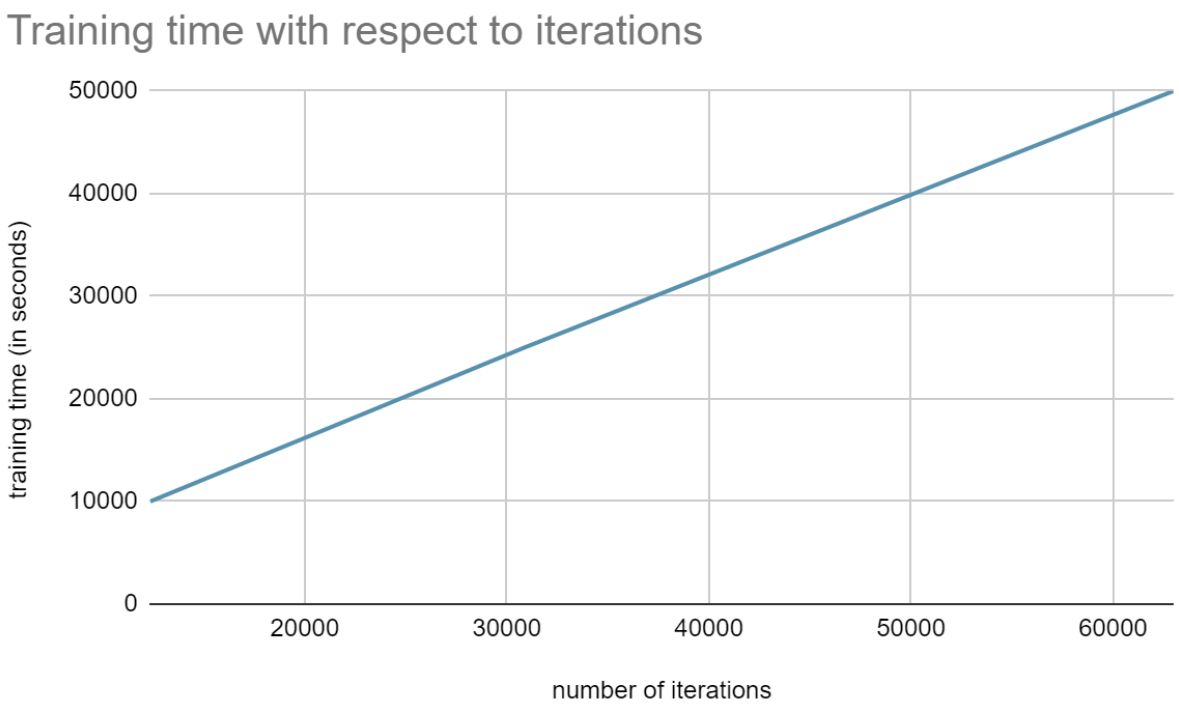


500000 iterations:



As we can see from these results, 50000 iterations resulted worse than others which is because of overfitting. When models are overtrained, they effectively memorize the training examples and fail to capture the underlying patterns. As a result models become too sensitive to noise or variations in the training dataset and struggle to generate new and diverse outputs. For this experiment we can say that 20000-25000 iterations were better than other ones.

Here is the training time change with respect to the number of iterations. As expected, training time increased linearly. It is important to consider the trade-off between training time and the desired level of model convergence.While training for a larger number of iterations may lead to better model performance, it also requires more computational resources and time.

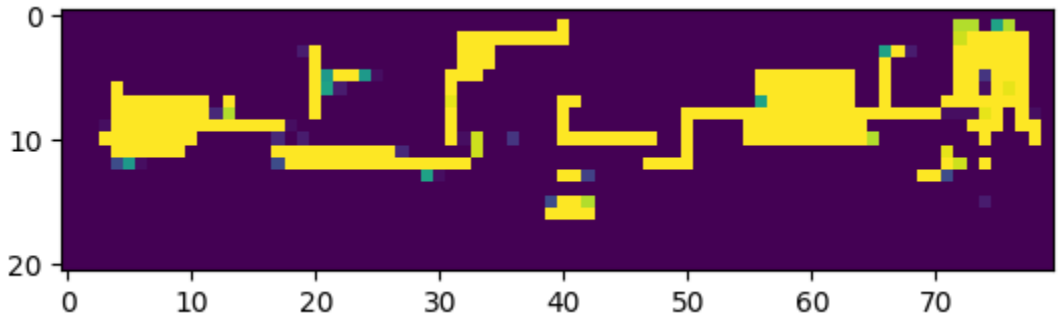


Batch Size Analysis

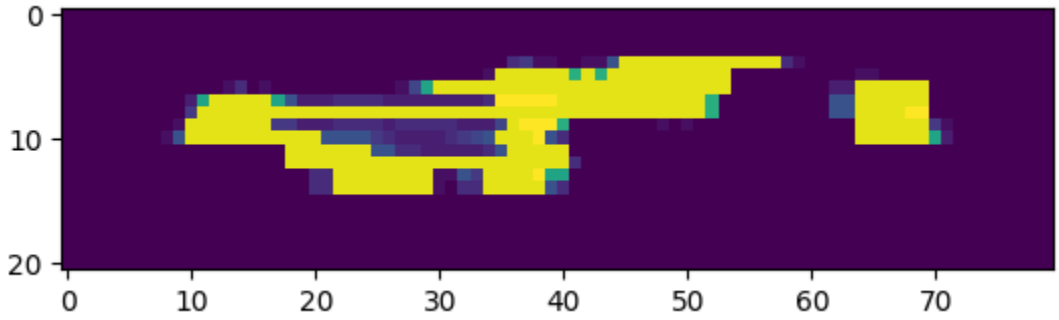
We experimented with different batch sizes during training which are 64, 32 and 16. The batch size determines the number of samples processed in each training iteration and affects the stability and convergence of the GAN models.

Our analysis showed that smaller batch sizes, like 16, led to slower convergence and less stable training. The models struggled to capture the underlying distribution of the training data and exhibited more variability in the generated levels. On the other hand, larger batch sizes, such as 64, provided more stable training and produced better levels. Here are the generated levels with different batch sizes but with the same learning rate (0.001) and equal number of iterations (10000).

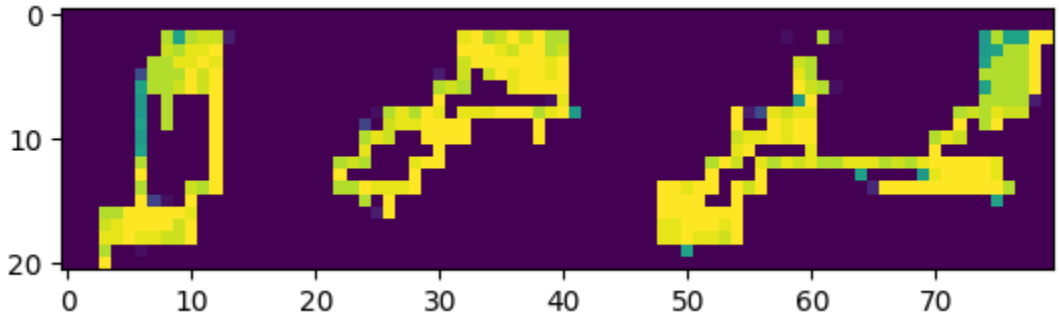
Batch size = 64:



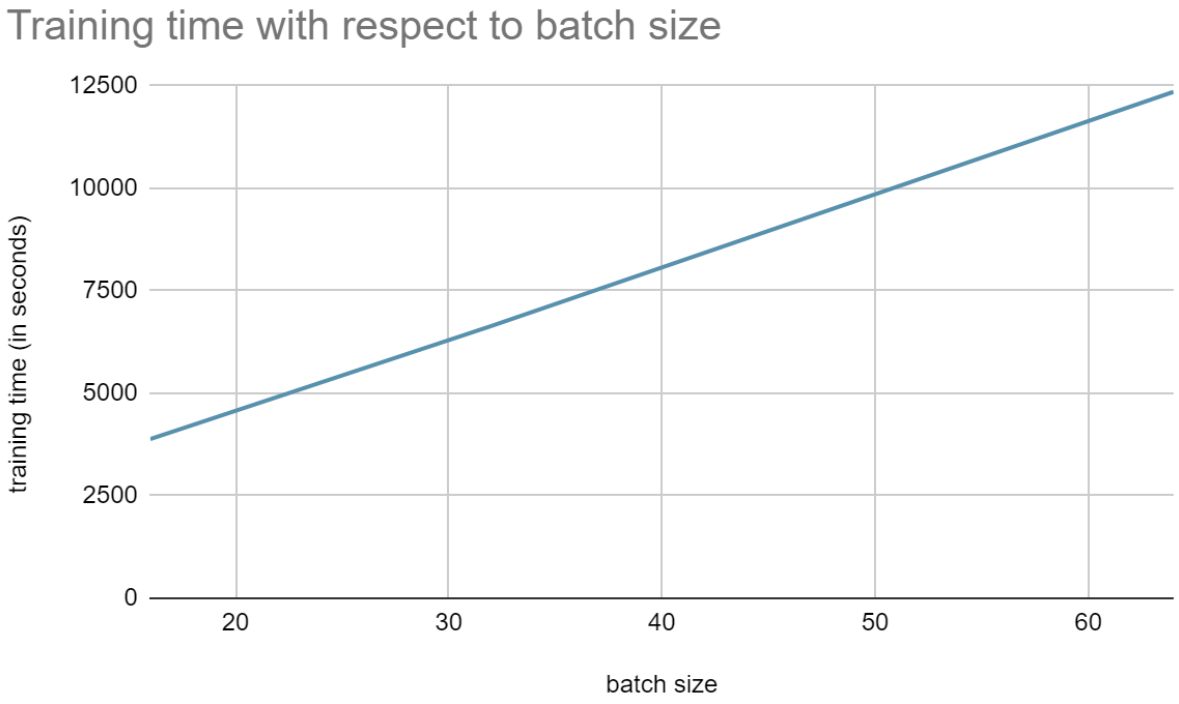
Batch size = 32:



Batch size = 16:



During experiments we observed that a batch size of 64 provided better balance between training time and model performance for generating levels. For a batch size of 16, the training time was reduced but it also led to a slight degradation in the quality of the generated levels.



We realized that smaller batch size resulted in less training time. When training a model with smaller batch size each update step involves a smaller number of samples. As a result the model can converge faster and reach a desired level of performance in fewer iterations.

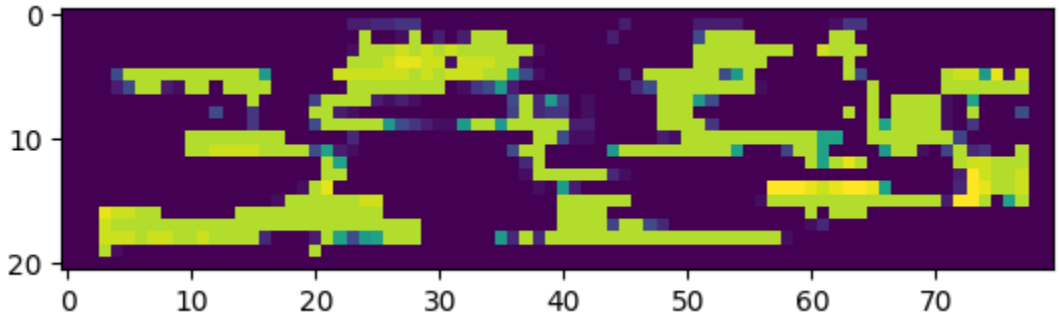
Learning Rate Analysis

We tried to train the model with different learning rates which are 0.001, 0.002 and 0.0002. The learning rate determines the step size at each iteration during the optimization process and affects the speed and quality of convergence.

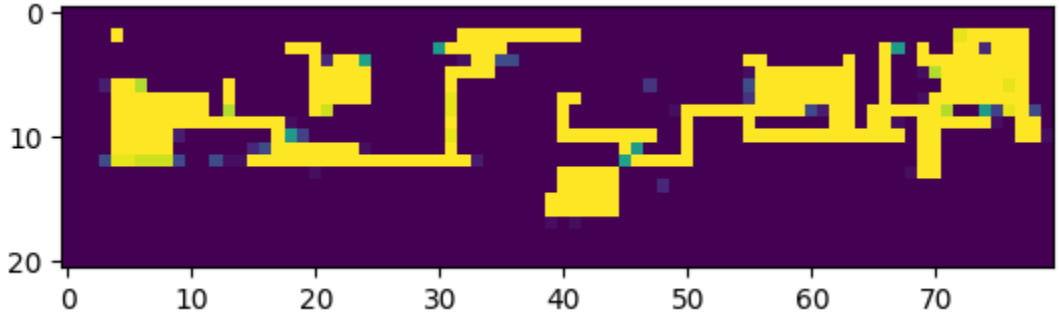
We found that a learning rate 0.0002 had the best results in terms of level quality and convergence speed. Higher learning rates, such as 0.002 resulted in unstable training and generated levels with more artifacts and inconsistencies. Lower learning rates like 0.001 led to slower convergence and lower quality levels.

Here are some of the generated levels with different learning rates:

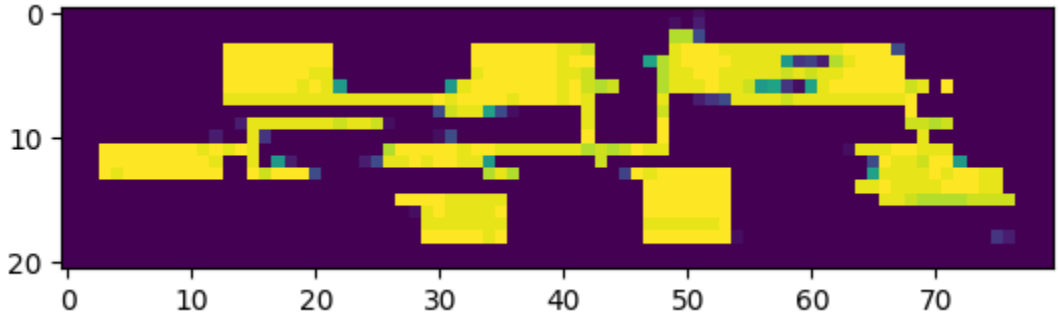
Learning rate = 0.002



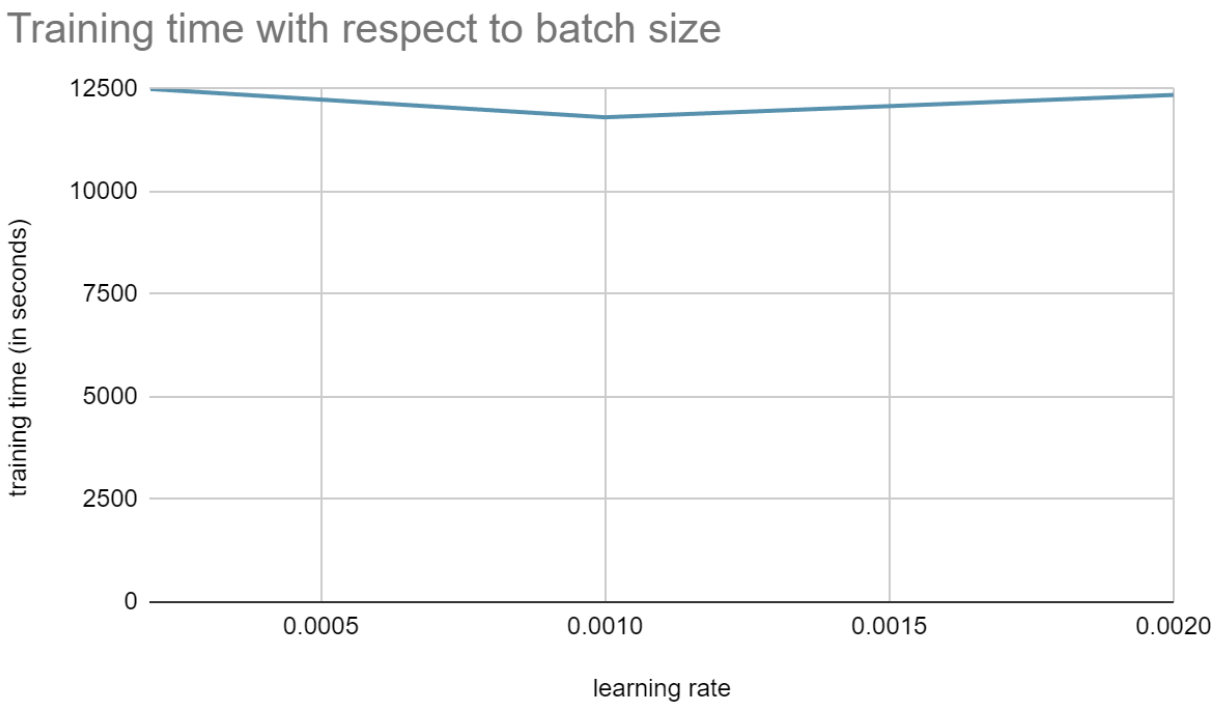
Learning rate = 0.001



Learning rate = 0.0002

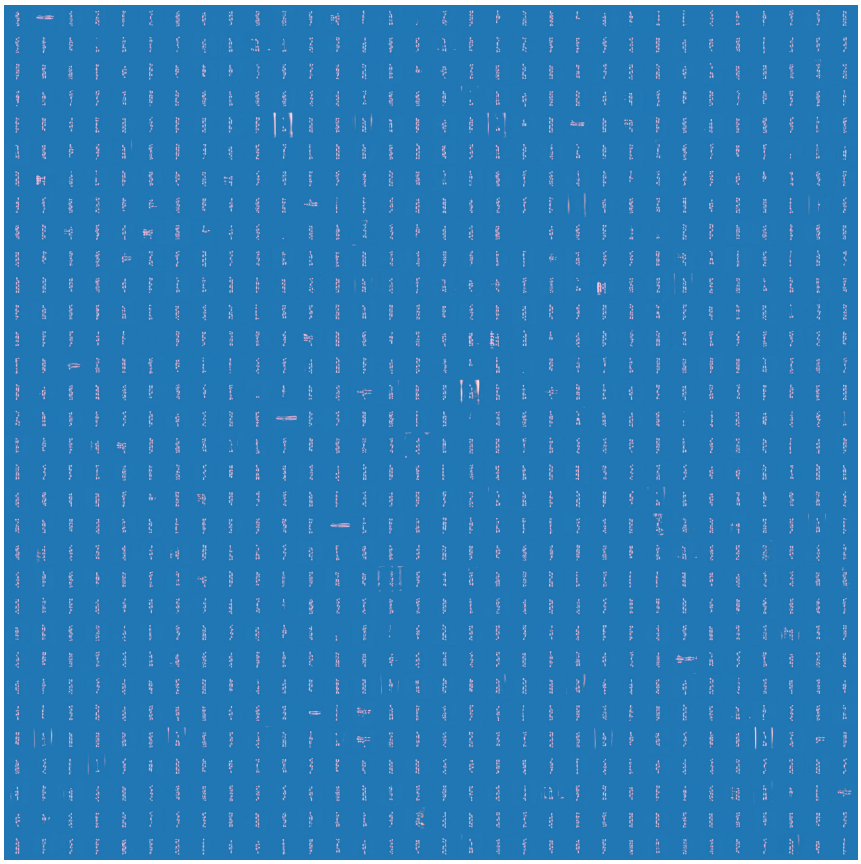


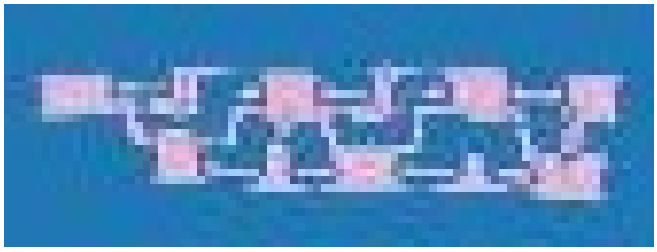
As we can see with generated levels 0.0002 yielded better results. In the model that trained with 0.002 learning rate, its parameter updates became too large. This can lead the optimization process to overshoot the optimal solution and result in unstable training. But lower training rate allowed for more gradual and controlled updates to the model's parameters. This helps the model to converge smoothly and reach better results.



As it can be seen from the graph, the learning rate did not have an impact on training time. It shows that the learning rates did not cause big fluctuations in the convergence behavior of the model.

We trained the StyleGAN2-ADA model with default learning rate and batch sizes which are 0.002 and 4. StyleGAN2-ADA's training time was much longer than AdaIN GAN even though using the same dataset. StyleGAN2-ADA has a more complex model. The increased complexity of the model requires more computational resources and longer training time to converge. Also StyleGAN2-ADA requires more powerful hardware like high-end GPUs. But after one week of training, we can say that StyleGAN2-ADA had more promising results than AdaIN GAN. Here are the generated levels from the StyleGAN2-ADA model.





Performance Evaluation and Comparison

We analyzed generated levels and unfortunately the results were not so promising in AdaIN GAN model. The generated levels exhibited several shortcomings and did not consistently capture the complexity and unpredictability of the original NetHack levels. The levels often contained disjointed sections, disconnected pathways, or abrupt changes in terrain. This compromised the playability of the generated levels.

While the model showed some capability in learning basic patterns and structures, it failed to capture small details and fine-grained features that are in original NetHack levels. The generated levels lacked room designs, hidden passages and diverse enemy placements which contribute to the engaging gameplay experience of NetHack. Comparing the results to the state-of-the-art approaches in NetHack level generation, it became evident that our trained AdaIN GAN model fell short in terms of level quality, it did not match the complexity by procedurally generated NetHack levels. While StyleGAN2-ADA's results were better, its generated levels still needed adjustments to make them playable.

In conclusion, although our AdaIN GAN model demonstrated some potential in generating NetHack levels, the results fell short of our expectations. The generated levels exhibited structural incoherence, lacked fine-grained details, and failed to match the complexity and diversity of manually designed or procedurally generated Nethack levels. These limitations indicate the need for further research and improvement in the application of GANs for Nethack level generation.

The Impact and Future Directions (/ 15 Points)

Explain the potential (or current if exist) impacts of your outcome in terms of how the methods and results will be used in real life, how it will change an existing process, or where it will be published, etc. Also, explain what would be the next step if the project is continued in the future, what kind of qualitative and/or quantitative updates can be made, shorty, where this project can go from here? This section should be between 250-500 words.

Future directions for this project include, but not limited to, exploring the use of reinforcement learning techniques to further improve the quality and diversity of the generated levels and developing tools that allow game developers to customize and fine-tune the generated levels according to their preferences and goals. Overall, our project reveals the potential of GANs in Procedural Content Generation and lights the way for further research and potential innovation in this field.

We expect these potential innovations to emerge first as a development tool and impact the game development industry in 3 stages:

1) Pre-Development Stage

In the Pre-Development Stage, newly developed tools will come to light and will have a minor impact on the game development industry. Developers and designers will meet and play with those tools to discover what they can and can not do. The tools are expected to be used for big open world map – such as Minecraft maps [4] [9], dungeon map – such as The Legend of Zelda dungeons [14], environment – such as Doom environments [2] [4], level – such as Super Mario Bros levels [1] [3] [5] [7] [10] and game content and asset [6] [8] [9] generation. Since the tools will still be in development and in dire need of improvement, they will have many flaws, from being very costly and exhaustive to generating unsatisfying results. Hence, it will be necessary to do post-processing and even hand-crafted (manual) adjustments on the generated results.

2) Development Stage

In the Development Stage, more improved tools will be developed to be used in development of a commercial game. These improved tools will have very few defects in terms of their cost, efficiency, usability and reliability, and will generally generate satisfactory results which can be used in a final product. Furthermore, the tools will offer advantages in terms of map and content generation speed, variety and quality. They will be reusable and customizable, making them much more advantageous for rapid prototyping. However, they will still require more data and fine-tuning to take the place of more common and accepted procedural generation methods that are used at the time this paper is written.

3) Post-Development Stage

In the Post-Development Stage, developed tools will be very advanced and complete. These advanced tools will have almost no defects and will be extremely optimized. The tools will be universally accepted and widely used as the new procedural content generation method in the game development industry. Specialized tools will also begin to be developed that focus only on generating one aspect of the game or interacting with other game systems or fields. Moreover, research and tool development will begin in different game development fields such as

storytelling, in-game dialogues, game economics, balancing systems, online matchmaking and more complex systems.

References

[1] V. Volz, J. Schrum, J. Liu, S. M. Lucas, A. M. Smith and S. Risi, "Evolving Mario levels in the latent space of a deep convolutional generative adversarial network", *Proc. Genet. Evol. Comput. Conf.*, pp. 221-228, 2018

[2] E. Giacomello, P. L. Lanzi and D. Loiacono, "Searching the Latent Space of a Generative Adversarial Network to Generate DOOM Levels," 2019 IEEE Conference on Games (CoG), London, UK, 2019, pp. 1-8, doi: 10.1109/CIG.2019.8848011.

[3] R. Rodriguez Torrado, A. Khalifa, M. Cerny Green, N. Justesen, S. Risi and J. Togelius, "Bootstrapping Conditional GANs for Video Game Level Generation," 2020 IEEE Conference on Games (CoG), Osaka, Japan, 2020, pp. 41-48, doi: 10.1109/CoG47356.2020.9231576.

[4] M. Awiszus, F. Schubert and B. Rosenhahn, "World-GAN: a Generative Model for Minecraft Worlds," 2021 IEEE Conference on Games (CoG), Copenhagen, Denmark, 2021, pp. 1-8, doi: 10.1109/CoG52621.2021.9619133.

[5] K. Steckel and J. Schrum, "Illuminating the Space of Beatable Lode Runner Levels Produced by Various Generative Adversarial Networks," in *GECCO Companion*, ACM, 2021, pp. 111-112.

[6] "Game Sprite Generator Using a Multi Discriminator GAN," *KSII Transactions on Internet and Information Systems*, vol. 13, no. 8, pp. [page numbers], Korean Society for Internet Information (KSII), Aug. 31, 2019.

[7] M. Awiszus, F. Schubert, and B. Rosenhahn, “TOAD-GAN: Coherent Style Level Generation from a Single Example”, *AIIDE*, vol. 16, no. 1, pp. 10-16, Oct. 2020.

[8] A. Fadaeddini, B. Majidi and M. Eshghi, "A Case Study of Generative Adversarial Networks for Procedural Synthesis of Original Textures in Video Games," 2018 2nd National and 1st International Digital Games Research Conference: Trends, Technologies, and Applications (DGRC), Tehran, Iran, 2018, pp. 118-122, doi: 10.1109/DGRC.2018.8712070.

[9] E. Panagiotou and E. Charou, "Procedural 3D Terrain Generation using Generative Adversarial Networks," in Proceedings of the IEEE Conference on Games (CoG), Galway, Ireland, 2019, pp. 1-8, doi: 10.1109/CIG.2019.8848077.

[10] M. Rajabi, M. Ashtiani, B. Minaei-Bidgoli, and O. Davoodi, "A dynamic balanced level generator for video games based on deep convolutional generative adversarial networks," in Proceedings of the 2nd National and 1st International Digital Games Research Conference: Trends, Technologies, and Applications (DGRC), Tehran, Iran, 2018, pp. 118-122, doi: 10.1109/DGRC.2018.8712070.

[11] Noor Shaker, Julian Togelius and Mark J Nelson, “Procedural content generation in games”, Springer, 2016.

[12] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio. “Generative Adversarial Networks”, 10 Jun 2014 https://arxiv.org/abs/1406.2661v1 , Accessed 23 May 2023.

[13] Nefi Alarcon, “Synthesizing High-Resolution Images with StyleGAN2”, 17 Jun 2020, https://developer.nvidia.com/blog/synthesizing-high-resolution-images-with-stylegan2/

[14] Jake Gutierrez, Jacob Schrum, “Generative Adversarial Network Rooms in Generative Graph Grammar Dungeons for The Legend of Zelda”, 19 Apr 2020, https://arxiv.org/abs/2001.05065 , Accessed 24 May 2023.

[15] Enter the Gungeon Official Webpage, https://enterthegungeon.com/

[16] Dodge Roll Official Webpage, https://dodgeroll.com/gungeon/

[17] Devolver Digital Official Webpage, https://www.devolverdigital.com/

[18] Boris, “Dungeon Generation in Enter The Gungeon”, 28 July 2019, https://www.boristhebrave.com/2019/07/28/dungeon-generation-in-enter-the-gungeon/

[19] Bob Nystrom, “Rooms and Mazes: A Procedural Dungeon Generator”, 21 December 2014, https://journal.stuffwithstuff.com/2014/12/21/rooms-and-mazes/

[20] Benjamin Hinchliff, “Dungeonator”, https://github.com/BenjaminHinchliff/dungeonator

[21] NetHack Official Webpage, https://nethack.org/

[22] “NetHack Level Generation Part 1-2-3”, https://procedural-generation.tumblr.com/post/157578222372/nethack1, https://procedural-generation.tumblr.com/post/157870872607/nethack-level-generation-part-2-making-rooms, https://procedural-generation.isaackarth.com/2017/03/08/nethack-level-generation-part-3-were-back-in.html

[23] NVlabs StyleGAN2-ADA. [Online]. Available: https://github.com/NVlabs/stylegan2-ada

[24] Manicman1999, "AdaIN-GAN: An implementation of AdaIN GAN," GitHub. [Online]. Available: https://github.com/manicman1999/AdaIN-GAN/blob/master/AdaIN_GAN.ipynb

[25] A. Radford and L. Metz, "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks," in Proceedings of the 4th International Conference on Learning Representations (ICLR), San Diego, CA, USA, May 2016.

[26] Situ, Zuxiang & Teng, Shuai & Liu, Hanlin & Luo, Jinhua & Zhou, Qianqian. (2021). Automated Sewer Defects Detection Using Style-Based Generative Adversarial Networks and Fine-Tuned Well-Known CNN Classifier. IEEE Access. PP. 1-1. 10.1109/ACCESS.2021.3073915.