Abstract

Machine learning's popularity has exploded in recent years with it being found to be being used in areas as varied as predictive texts to video generation with a number of places (6 Ways Machine Learning will be used in Game Development, 2018) tipping game development as the next area that would benefit from machine learning. This isn't surprising as models such as generative adversarial networks, commonly called gans, Have been shown (This Person Does Not Exist, 2021)to exhibit the sort of stochastic constrained creativity that would make it ideal for some aspects of game development such as level generation.

This project looks at utilizing said generative models and applying them to level generation. It takes inspiration from games such as Binding of Isaac (The Binding of Isaac on Steam, n.d.) and more generally from the roguelike genre which often feature forms of procedurally generated levels and ask where it's feasible to employ generative models such as generative adversarial networks. It also asks if it is feasible, what is the quality of those levels and how do they compare to existing systems. It does this by constructing a system utilizing 2 separate gans, one to generate an overall dungeon layout and another to control the interior layouts of each room. This system could obviously be hugely useful within games however beyond it's obvious use in games and entertainment it's not completely infeasible that the system described in the report could be useful a new approach in situations with similar types of constraints such as to generate such as architectural blueprints (Chaillou, 2021) or possibly as a tool underpinning city planning systems(Wang, Fu and Wang, 2021) as both the aforementioned use cases have similar constraints as the initial level generation systems namely that firstly they must have a certain level of connectivity across the generated objects and secondly they both rely on chaining models together to generate a final object meaning that with just a change of training data the system described in this report could be repurposed.

Project aims & objectives

In terms of measurable aims and objectives this has one overall aim is to investigate the feasibility of using machine learning to generate 3d levels for computer games. To do this the project will attempt to use gans to generate a level from a dungeon crawler style game, taking inspiration from a previous attempt at this (Giacomello, Lanzi and Loiacono, 2018).Should this be shown to be possible this project will go on further and put together a simple prototype roguelike game.

Literature review

When examining current literature of similar projects, it can be a struggle to extract any concrete information as traditionally developers haven't felt the need to publicise exactly what procedural generation system they employed. Since the first public use of procedural generation in video games in 1978s Beneath Apple Manor (Beneath Apple Manor, n.d.) till up until the early 2000s procedural generation was the domain of primarily films with tools being developed to manage things such as smoke and fire simulations. Beyond films the sole identifiable algorithms being employed with regards to level generation within games were concepts being co-opted from general computing and not explicitly tied to computer

games that the practice of procedural generation really took off and even then it required a fairly careful approach with developers working on a case by case basis to select algorithms such as Perlin noise originally developed in 1983 (Using Perlin Noise to Generate 2D Terrain and Water, 2021) as a method to generate more realistic terrain quickly for cgi in films it now found itself in 2011 being used to underpin the terrain generation of games such as Minecraft(Here's how 'Minecraft' creates its gigantic worlds, 2021)Another commonly reused technique is to use Binary space partitioning a technique that initially found its use in optimising 3d graphics by building a more performant representation of a given 3d scene. This representation would inevitably be structured as a tree and was promptly reused by games such as Nethack or Angband (BSP Room Dungeons - Roguelike Tutorial - In Rust, n.d.) to construct ascii dungeons. Binary space partitioning has the advantage that you're guaranteed a number of crucial facts about the level will be true; firstly that none of the rooms generated will overlap, secondly that every room will be connected these two facts alone made it the go to algorithm for decades of games. Of course borrowing and repurposing pre-existing concepts has been not without its drawback for video game developers in the recent decade as on one hand developers had this ready library of known good algorithms available to pickover and experiment with on the other hand it has completely stifled innovation and left a blank space for developers to develop new algorithms specifically targeted at level generation. This is only recently starting to be rectified with Wave function collapse (Gumin, M., 2016. *GitHub - mxgmn/WaveFunctionCollapse: Bitmap & tilemap generation from a single example with the help of ideas from quantum mechanics*. [online] GitHub)
this is a powerful new algorithm put forward by Maxim Gumin in 2016 that actually has numerous similarities to a deep learning approach without a few of the drawbacks. Similarities such as both options require some form of input data and both systems constrains the output based on the input data. Where the systems differ is perhaps to the detriment of deep learning with wave function collapse allowing for much more flexibility without the need to keep training models.

Beyond films and as computers became more prominent in everyday life procedural generation found itself at home in more and more places, of those the most relevant for this project is the computer demoscene subculture. A subculture dedicated to developing self-contained audio visual experiences often at extremely small file sizes. One such project was the game .kkrieger(.kkrieger: Chapter 1 (Windows), n.d.) released in 2004. It's a fairly bad first person shooter that noticeably generated absolutely every aspect of the game at runtime from the 3D art assets to the music, this gave the game a total file size of 96kb. This is noticeable as it's one of the earlier examples of procedural generation making the jump from cgi in films to computer games however moving on to deep learning it's long been theorised that deep learning could revolutionise game development (Statt, 2019) in fact we're start to see elements of that coming to pass with regards to level generation and not just through the use of gans With a collection of machine learning avenues being explored such as lstm based neural networks being applied to generating Mario levels or reinforcement learning based solutions being applied to generate Sokoban levels (Game Level Design with Reinforcement Learning, 2021) or ether Bayes networks being used to generate levels for one of the earlier Legend of Zelda games(J. Summerville, Behrooz, Mateas and Jhala, n.d.) the problem being proposed to mixed results. Generative adversarial networks themselves have been used already to generate levels with occasionally good results such as in

(Summerville, A. and Mateas, M., 2016. *Super Mario as a String: Platformer Level Generation Via LSTMs*. [online] arXiv.org. Available at: <https://arxiv.org/abs/1603.00930> [Accessed 14 September 2021].) where they showed that you could generate Super mario bros levels via gans from ownly one single example, they also showed it was possible to get incredible examples. Their method relies on generating a token based level and actually takes its inspiration from natural language processing by assigning the probability of a given token based on nlp concepts which wouldn't necessarily have been available to me given the 3 dimensions my system generates in vs their two dimensions. Perhaps the most most relevant is an attempt in 2018 to generate levels for the 1993 game Doom via Gans (Giacomello, Lanzi and Loiacono, 2018) in their attempt they utilized an existing repository of 9000 community made levels to train their system on which is obviously something this project is unable to replicate due to the this project not focusing on an existing game. By utilising an existing game their project required them to generate 4 separate maps to cover all aspects of a Doom level. These projects show that deep learning is starting to be experimented more as a tool for level generation with mixed results especially in the case of the Doom experiment which despite passing in terms of system performance the levels generated do not pass a human test as they all have a disjointed uncanny valley feel about them.

Methodology

In terms of methodology this system consists of a 5 stage process utilising two separate Gans that work in tandem to generate a fully playable level.

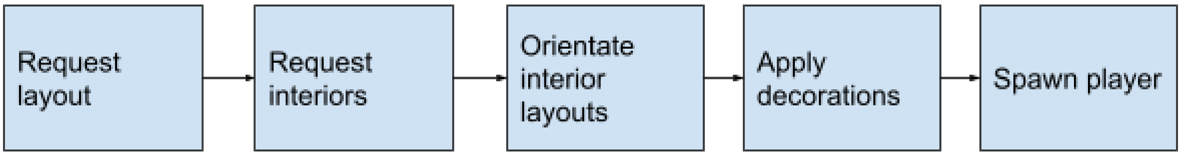
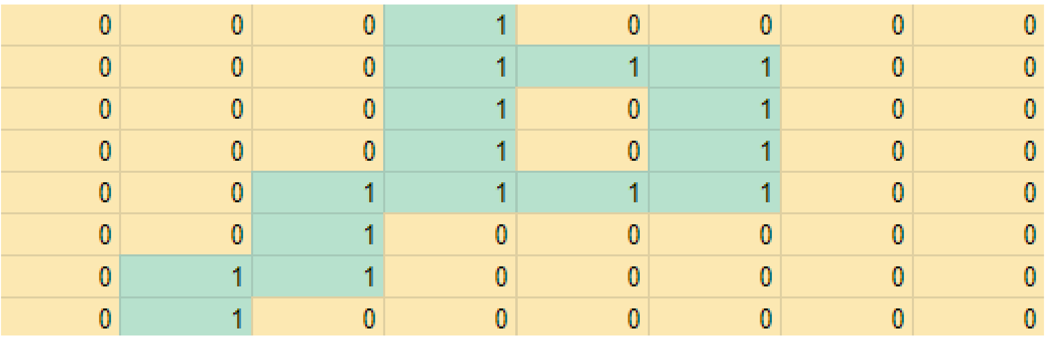


Fig - the five stages of level generation.

The first stage is for one the Gans to request the dungeon layout; this gan is a standard convolution neural network based Gan. It's job is to send the request for a new layout to a Flask web server running locally. Upon receiving a response the initial rooms are laid out and At this point the system prunes off any isolated single rooms as they're going to be unusable. Once This is complete each room sends a request to the Flask web server containing the rooms x and y location on the grid, from this the second Gan is capable of working out exactly how many exits are needed and generates an interior accordingly. Once the layout data is returned the room iterates over it generating the layout as best it can with some allowances for overlapping room objects. The next stage is orientating the rooms, this is crucial because the Gan has no knowledge of the orientation required to make each exit line up with another. The approach employed here is to brute force the problem by simply rotating each room a number of times recording the average distance from each exit to the closest exits on each of the rooms' neighbours along each cardinal direction. Before finally applying whichever rotation results in the smallest average distance.This approach is not perfect however it is good enough that minor failures to get the right rotation could be interpreted as a pleasant variation in the overall dungeon layout. The penultimate stage is a case of prettying up the dungeon, as specific spawnable objects that you'd expect to find additional decoration on are subsequently decorated, this includes stuff like bookshelves and tables, but really could be extended to any spawnable object. The final step is to spawn the player, ether this is done by taking the first and last room that was generated assigning one of them as start and placing the player in there while spawning an ending portal in the end room or the system as it stands allows the developer to assign rooms as start and end.

dataset overview.

In Terms of dataset used in this system in total there are 2200 samples across the two Gans, the layout Gan is trained using 1000 8 by 8 grids.



0	0	0	1	0	0	0	0
0	0	0	1	1	1	0	0
0	0	0	1	0	1	0	0
0	0	0	1	0	1	0	0
0	0	1	1	1	1	0	0
0	0	1	0	0	0	0	0
0	1	1	0	0	0	0	0
0	1	0	0	0	0	0	0

Fig - Example layout Gan data.

Structured in such a way that 1 means there's a room and 0 means there is an empty space.The secondary Gan that generates room interiors is a conditional Gan so naturally has a more complex dataset consisting of 1200 16 by 16 grids. This breaks down to be 200 samples per data class across 6 classes.

Table showing how the different data classes correspond to the number of exits.

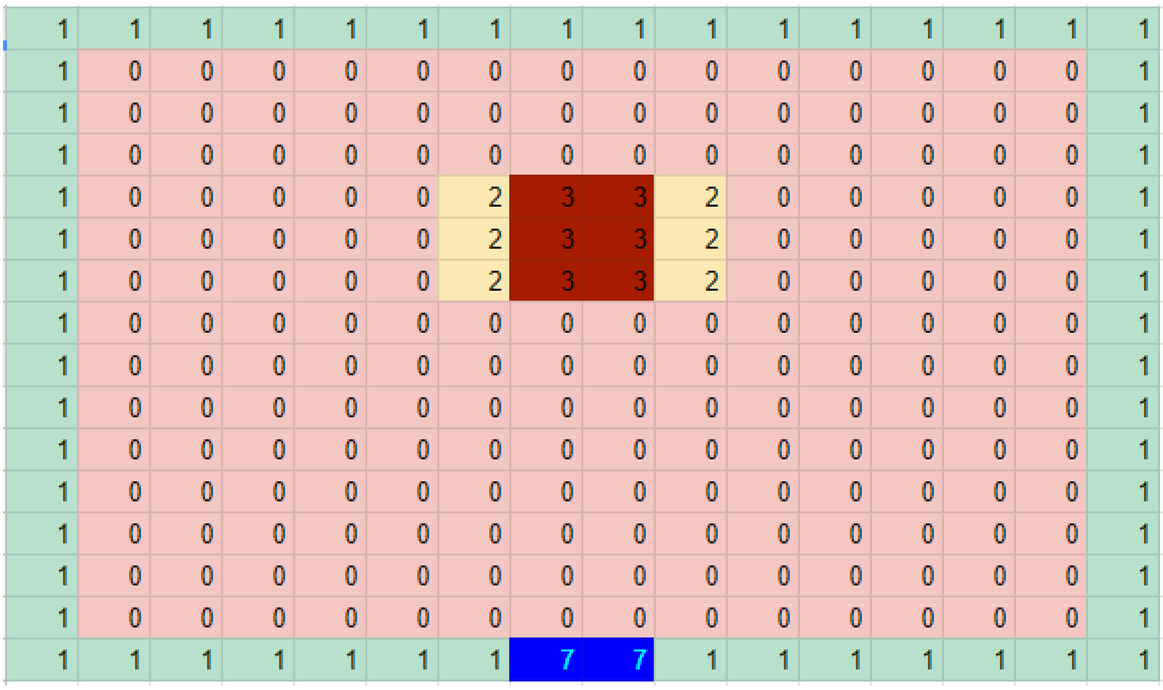
Class id	Number of exits.
1	1
2	2 - two exits across from each other.
3	3
4	4
5	2 - cornering left
6	2 - cornering right

In both cases additional data augmentation on account of was tried on account of the samples size being fairly low and to see if this in turn had a positive effects and to primarily exploit the lack of symmetry found in the samples however for reasons I will go in to in the results and conclusion section this was found to be sub optimal. As you can see from the table above the dataset accounts for all possible options this allows the system to be fairly flexible in terms of layouts and also guarantees that all parts of the level will be connected together. In terms of the actual dataset used in the second Gan it was agreed early on with my supervisor to limit myself to a total of 9 possible items and to assign different rules to some of them.

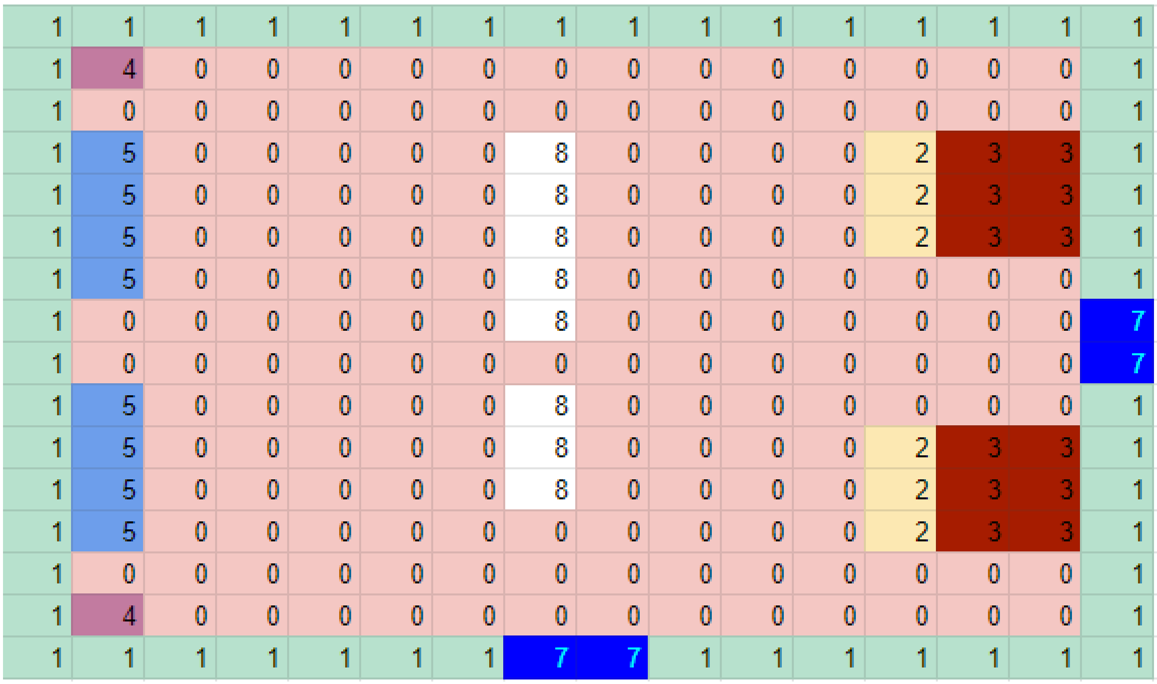
Table showing each possible item the system can place in the room layout & associated rule.

Item name	Id	Rule
Background	0	1 row of 1 square.
Wall	1	1 row of 1 square.
Bench	2	1 row of 3 squares
Table	3	2 rows of 3 squares
Collectable	4	1 row of 1 square
BookShelf	5	1 row of 4 squares
Torch	6	1 row of 1 squares
Door	7	1 row of 2 squares
Crate	8	1 row of 1 square.

Beyond the simple size rules a number of logical rules where employed to help better place the items and to hopefully give the levels an element of human design to them, rules such as when placing a bench it should ether be against a wall or against a table, because that's where you'd expect a human designed interior to place the benches. You can see below some examples of how these rules manifest themselves.



1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	0	0	2	3	3	2	0	0	0	0	0	1
1	0	0	0	0	0	2	3	3	2	0	0	0	0	0	1
1	0	0	0	0	0	2	3	3	2	0	0	0	0	0	1
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1	1	1	1	1	1	1	7	7	1	1	1	1	1	1	1



1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	4	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1	5	0	0	0	0	0	8	0	0	0	0	2	3	3	1
1	5	0	0	0	0	0	8	0	0	0	0	2	3	3	1
1	5	0	0	0	0	0	8	0	0	0	0	2	3	3	1
1	5	0	0	0	0	0	8	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	8	0	0	0	0	0	0	0	7
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	7
1	5	0	0	0	0	0	8	0	0	0	0	0	0	0	1
1	5	0	0	0	0	0	8	0	0	0	0	2	3	3	1
1	5	0	0	0	0	0	8	0	0	0	0	2	3	3	1
1	5	0	0	0	0	0	0	0	0	0	0	2	3	3	1
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1	4	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1	1	1	1	1	1	1	7	7	1	1	1	1	1	1	1

Fig - Example room interior Gan data.

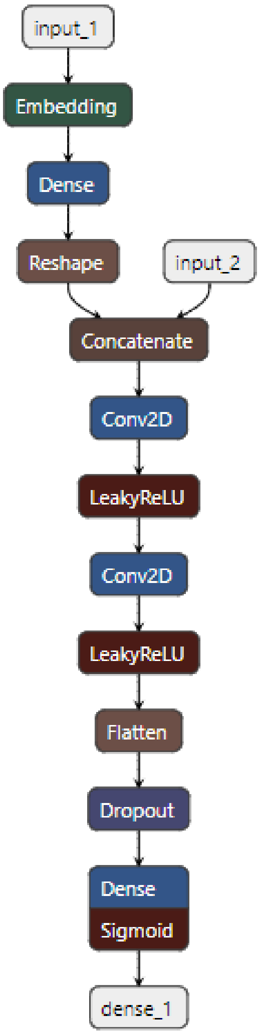
Network overview

In terms of network architecture I am indebted to the 'How to train your Gan' Github article (GitHub - soumith/ganhacks: starter from "How to Train a GAN?" at NIPS2016, 2016) more specifically the following tips;

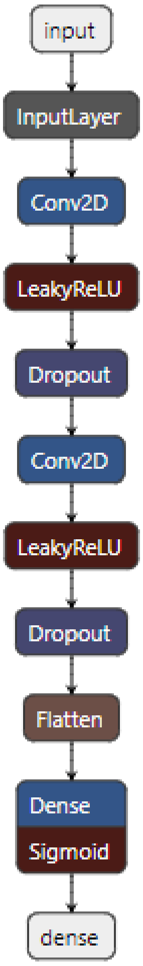
- Don't use a normal distribution when sampling
- Use Relu & Conv2D.
- Use Adam as an optimizer.

.

Using netron(Netron, 2021) we can see the inner workings of each of the four models within the system and more specifically we can see how tips such as to use Relu layers and Conv2D layers have been applied especially within the discriminators for both Gans.

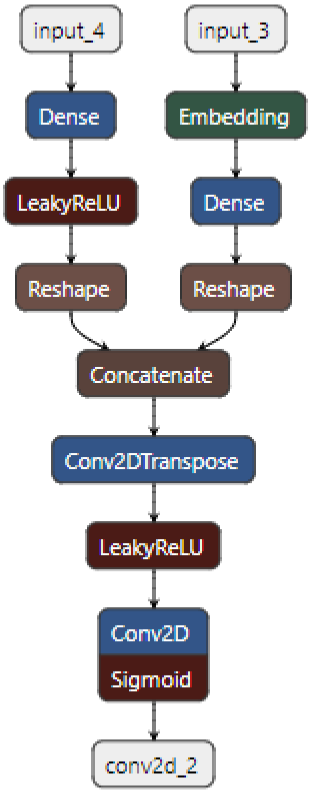


Interior discriminator model

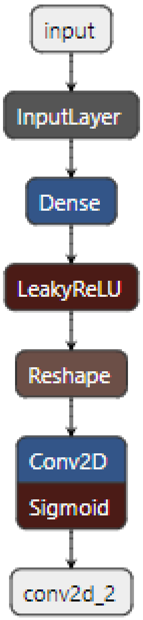


Layout discriminator model

The main goal of this combination is that these two layers together allow for the input image to be scaled down without running into the vanishing gradient issues that can happen. This is done via the Conv2D layer which downsamples the input gradually whilst the Leaky Relu layer is there because as (Singh, n.d.)and (A Practical Guide to ReLU, n.d.) note this type of activation function is noticeably faster to train a network then normal relu. It should be noted that the main reason for the Sigmoid activation function on the final layers of each of the discriminators is obviously because this is the discriminator within the gan so we need to scale the result to be either 0 or 1 as such sigmoid is the perfect function to use as it constrains the single output to be between those two values. When looking at the generators we can see that they have a lot of similarities with the discriminators only in this case they're using Con2DTranspose to up sample the inputs. The sigmoid activation function is also used for a number of reasons such as in the case of the layout generator we only have two possible options: either a given cell is a 0 and subsequently doesn't have a room or it is a 1 and it does. The room interior generator uses a sigmoid because as part of the data pre processing all 7 possible options are scaled to be between 0 because 0 is the lowest value used in the interior dataset and 1.



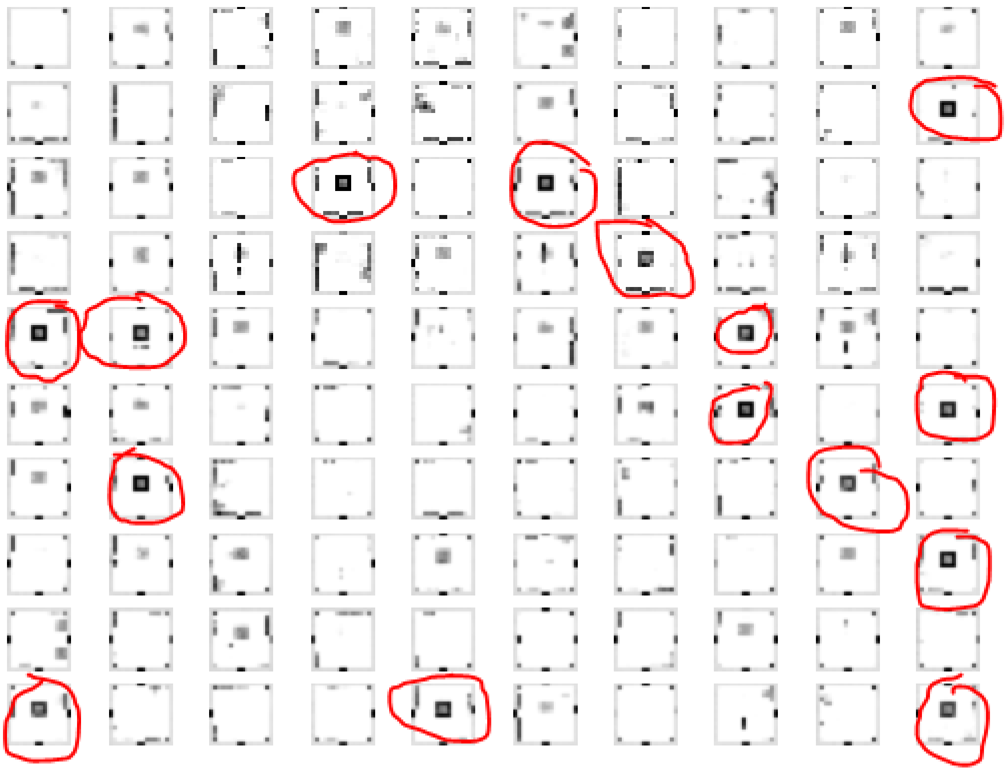
Interior generator model



Layout generator model

As you can see from the above diagrams beyond those useful tips none of the networks used are exactly new or innovative that's perhaps a good thing as with all machine learning it's possible to come up with systems and networks that are far too complex and complicated for a basic requirement which subsequently makes debugging an issue.

In addition to what worked during the development and training process the concept of using 'soft and noisy' data labels was tried as at one point all the images being generated during the train process began showing elements of model collapse almost immediately with it's fixation on squares.



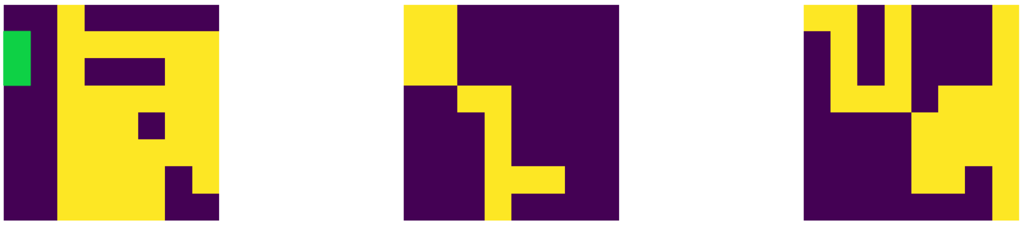
Early training example where the system started showing signs of model collapse.

Fortunately the resulting models when generated exhibited none of these traits as such these labels were removed owing to them also causing the Gans loss to always report as 0 regardless of what was actually being generated.

Validating dungeons

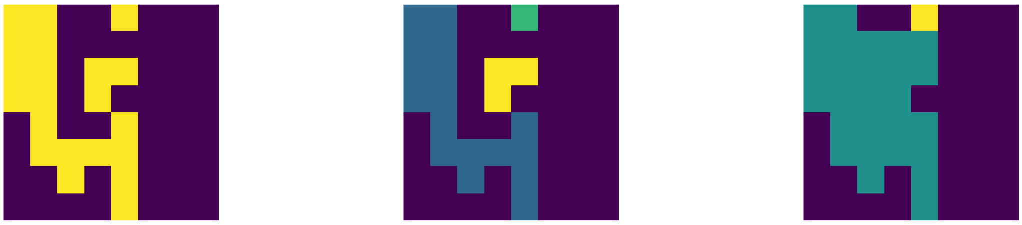
Unfortunately, due to the lack of samples the layout generator has a fatal flaw where it has a tendency to generate dungeons that fail to meet a minimum level of playability. Playability being defined in this case where ignoring all single isolated rooms as they will be removed later, all areas of the generated dungeon should be reachable by the player.

Table showing how occasionally the layout generator generated layouts that did not meet a minimum level of playability.



This issue happens roughly 20 times per 100 generated images as such it became fairly clear an extra step was required to make sure the layouts generated were actually playable. From discussions with my supervisor he suggested a two step solution, firstly isolate each of the islands via connected component labelling(Connected Component Labeling: Algorithm and Python Implementation, 2021) secondly to run some sort of pathfinding between all the islands, where if the path is greater than 0 fill it in.

The stages of the validation process.



Initial network generated output

Isolated components of the output

Final generated layout.

The connected component labelling is done in a one stage algorithm following a very basic algorithm. The system goes through each pixel in the image and checks if the value of the pixel is equal to 1. If it is the system then recursively grabs each of the pixels neighbours checks if those pixels are equal to 1, this continues until none of the neighbours contain a 1. Whereupon it assigns those pixels its checks to an array and assigns each of those pixels the value equal to the group's position within the array. The system then moves the initial starting point along by 1 and repeats.

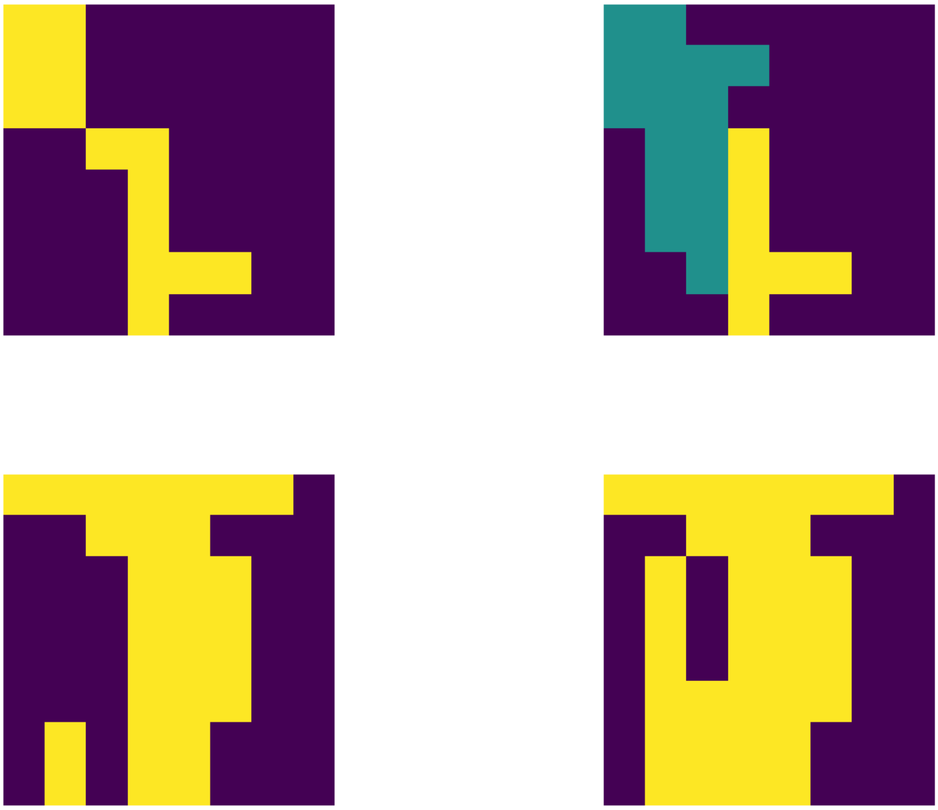
Upon isolating each of the components of the image we obviously need to run the pathfinding to connect up each of the islands, as for which cell of the island is connected, I have opted to pick randomly. The system iterates through each component that has been isolated picking a location in that component and another location that is in a different component. To generate a valid path from A to B the system starts with A and itteratiratively gets all of the neighbours for A and calculates a distance value for each neighbour using the equation found below.

$$|(Location\ to\ check\ x\ -\ goal\ location\ x)|\ +\ |(location\ to\ check\ y\ -\ goal\ location\ y)|$$

The system then moves to find the neighbour with the lowest distance value that hasn't already been checked before getting that cell's neighbours and calculating distance values for those cells; and so on until the get neighbors request contains B in which case the system has found it's path. We can take this brute force approach because the layout images have no obstacles on so we have no notion of actually having to find a path around anything we just need to connect a to b. As you can see from the table below the end result although not a precise fix however could be considered to at least be in the spirit of the initial image.

Initial image

Resulting image post validation



Unity

Connecting everything up.

The last stage of this system is to interpret the images generated and to actually create the levels. The game engine of choice for this was always going to be unity. Unity was picked for two reasons firstly because of development familiarity and secondly due to the programming language found within unity. The flexibility that c# provides is crucial because it allows the two halves of the system to communicate through the use of c#s native web request library on the c# side and a simple Flask api on the python side. As to what api requests are actually available, There are actually only two requests within the API. One to generate a layout and one to Request a room interior.

- /Layout

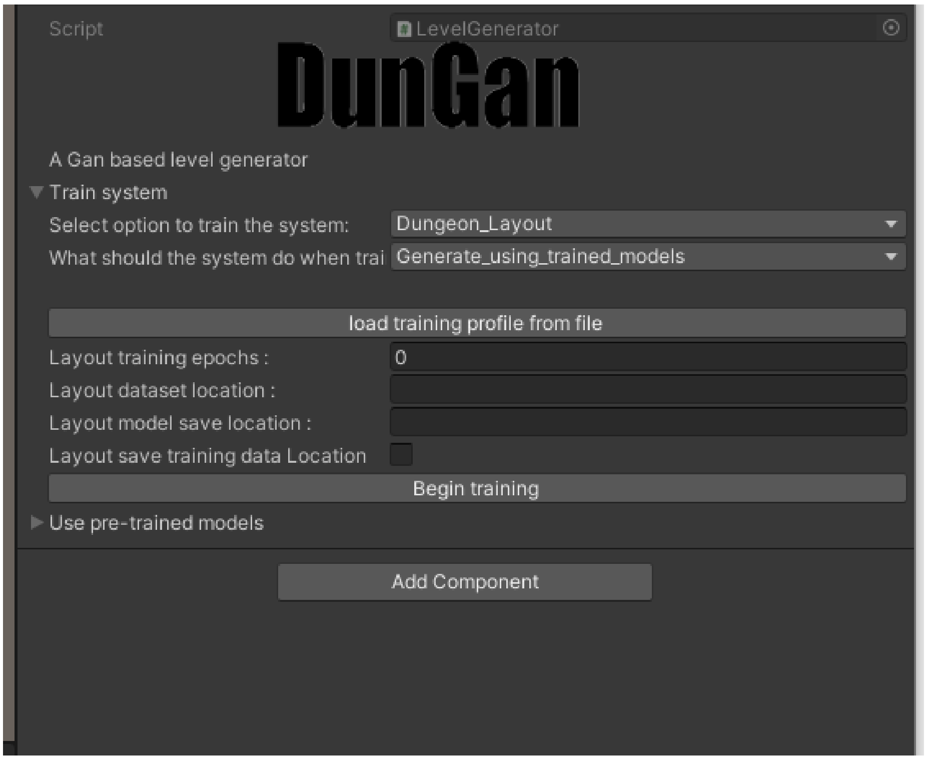
  This is the initial request that the system sends to start the process, the response from this request is a string of numbers that are used as the layout for the dungeon.

- /Room/<x>,<y>'

  This is the request that each of the rooms send to fill out their respective interiors, the response from this is again a string of numbers that get interpreted via a switch statement to spawn the correct object. The two arguments for this request are the x and y locations of the room, these are picked up from the initial request and get set upon the room being spawned, because of this the randomize room button found in one of the scenes is able to reuse this request.

Overview of what's been built.

In terms of what has actually been built to show off the system there are two scenes within unity. Firstly the system has been presented as an editor focused tool.



Script: LevelGenerator

A Gan based level generator

▼ Train system

Select option to train the system: Dungeon_Layout

What should the system do when trai Generate_using_trained_models

Layout training epochs : 0

Layout dataset location :

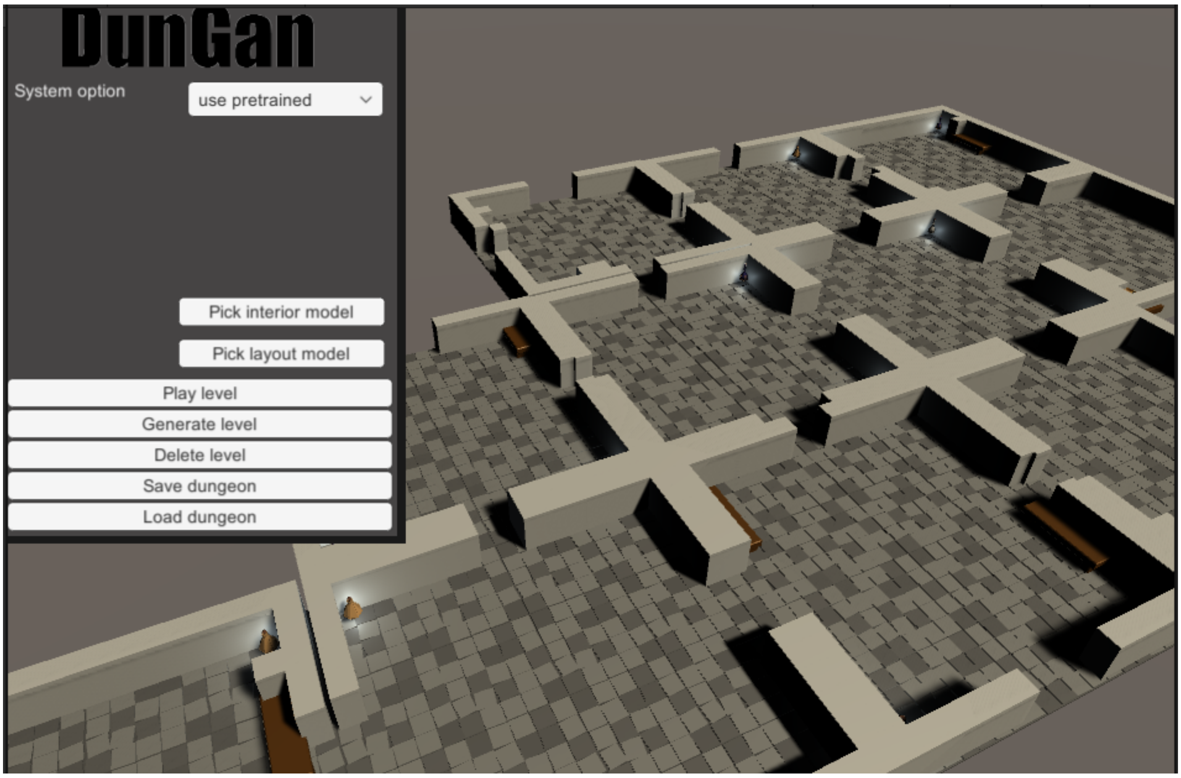
Layout model save location :

Layout save training data Location □

▶ Use pre-trained models

Fig - the systems editor tool set up.

This tool allows the user to train both gans within the system while providing flexibility in terms of what training data is used and where the trained model is saved and which models are actually used for generation. The second scene shows off the system as a replacement for the level generation systems that generate at run time such as bsp or Perlin noise. Providing the same basic controls as the editor tool whilst also expanding to show off the system as the basis for a full blow level editor by including the ability for the user to randomize any given room interior and save and load levels.



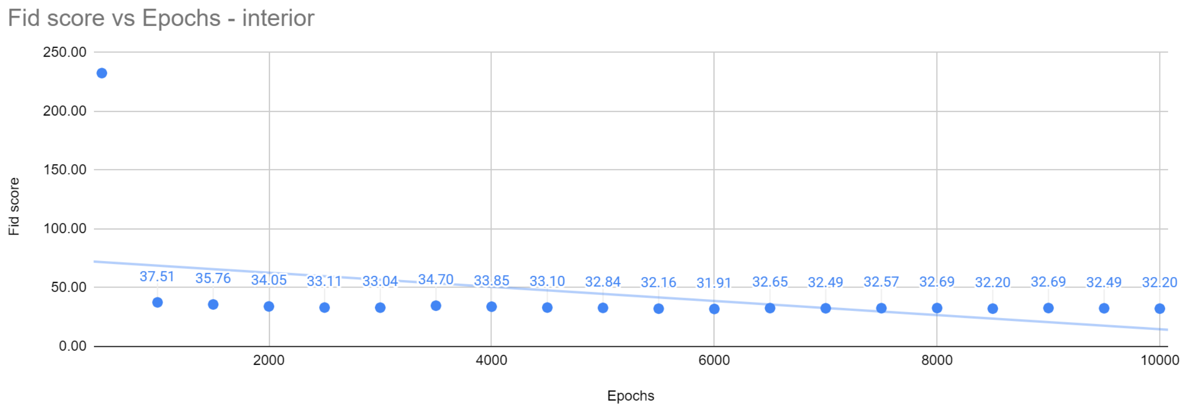
System option: use pretrained

Fig - Showing the system being used in editor.

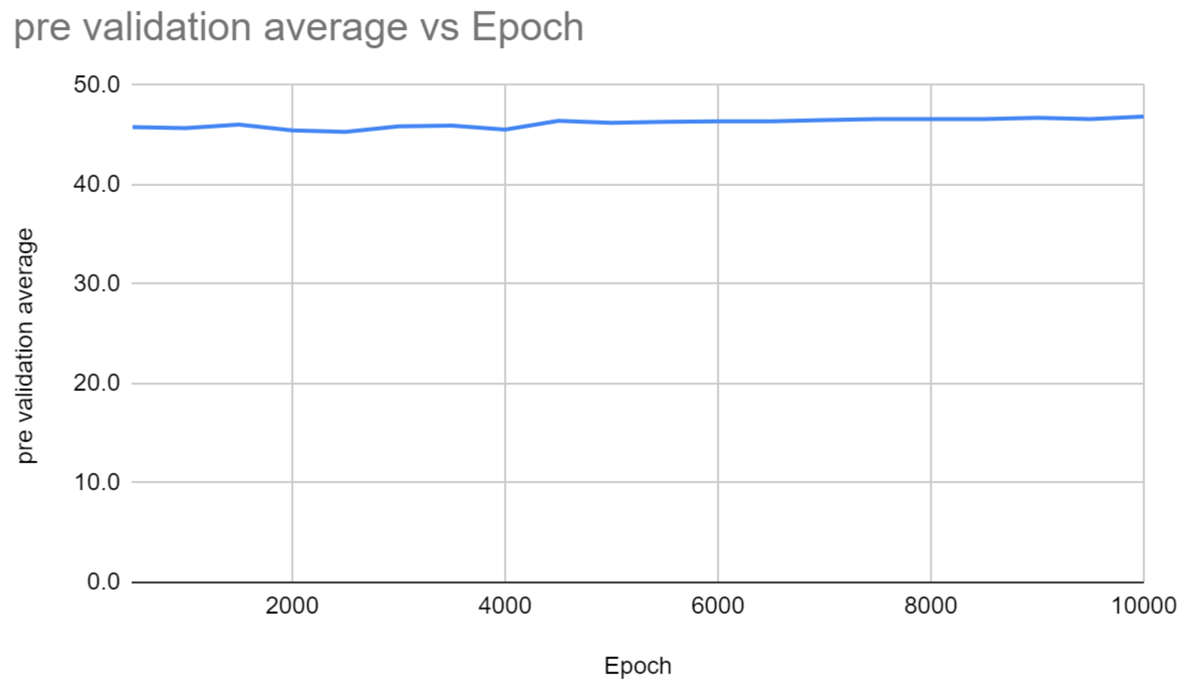
Results

Fid score

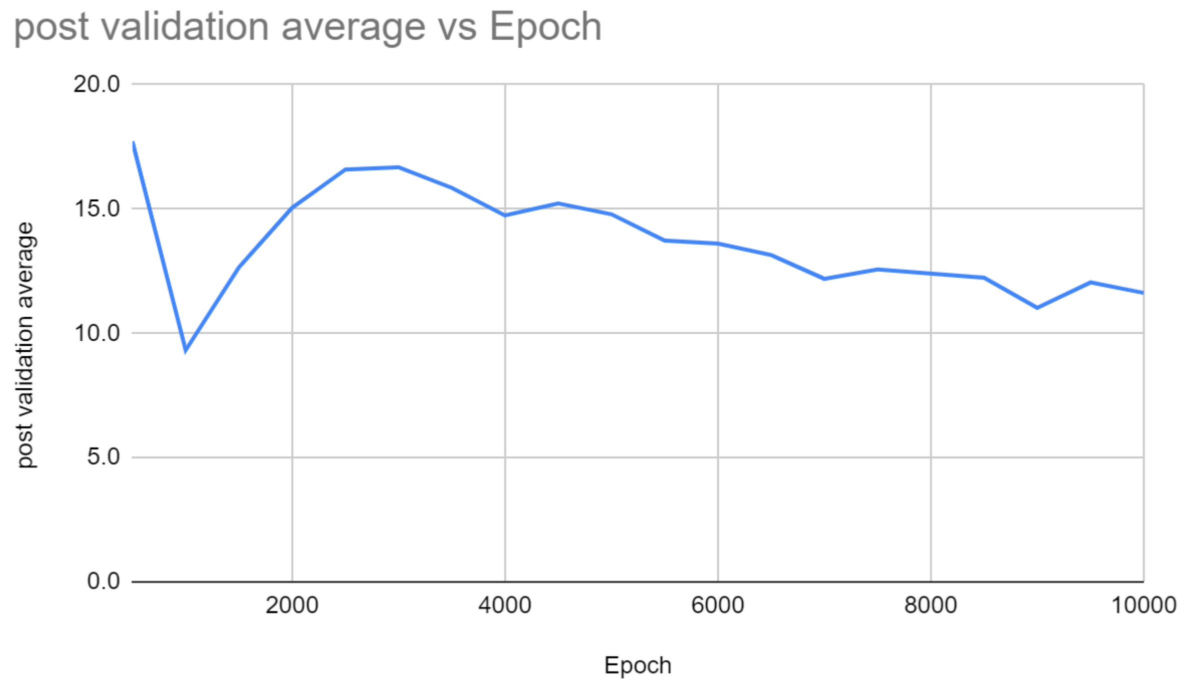
When analysing the system there are two points of reference. Firstly you can look at how the two gans perform numerically. Secondly it is important to example the quality of the final outputs actually within unity because it doesn't matter how the gans perform if the generated results are noticeably sub-par. In terms of analysing gan performance numerically a number of different metrics have been employed. Firstly I have used the Frechet Inception Distance or 'Fid' this is as close to an industry standard as there currently is. Fid measures the distance between two feature vectors, one containing real images and one containing images generated by the system so a lower Fid score means the system is generating images that are close to the real images, importantly this metrics has also been shown to correlate strongly with overall image quality.This project could be considered one of image generation as prior to being interpreted via Unity the system is generating valid images as such when looking at what constitutes a good Fid score the website Papers with code (Papers with Code - CIFAR-10 Benchmark (Image Generation), 2021) provides a good leaderboard for Fid scores calculated across a variety of image generation databases. I have opted to use the CIFAR-10 leaderboard purely down to it having the largest number of entries. In this leaderboard the Fid score ranged from 29 to an impressive 2.1 with that in mind we can at least say that the interior Gan displays a good level of similarity to the real samples



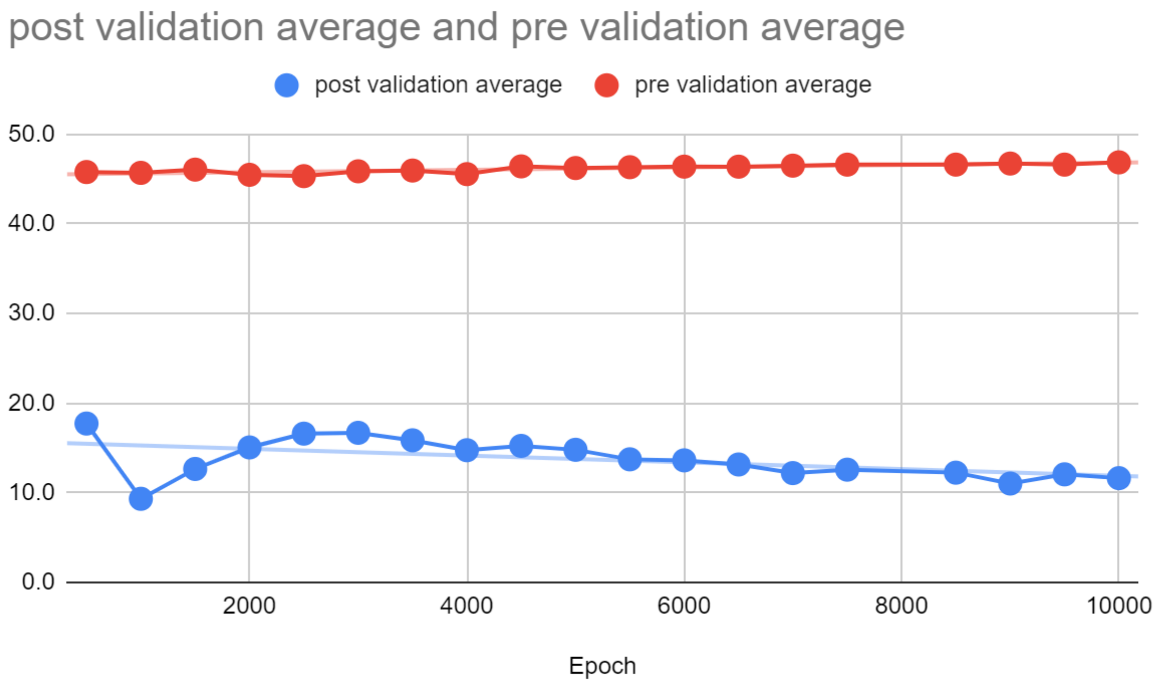
Obviously this is once you get beyond 500 epochs as a Fid score of in the 200s is akin to just random noise. Beyond that all scores are within the 30's and trending down suggesting that had training been carried on the samples would have gotten even closer to the real images. In terms of the dungeon layout, there's an interesting dilemma when generating quantitative data as to its performance. Because of the validation process resulting in dramatically different images vs initial output as such I've included both and a comparison.



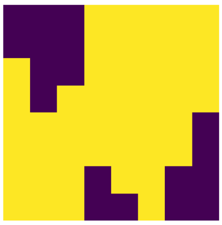
As you can see the pre validation fid score average of 5 separate runs across 10k epochs shows a fairly stable output which is interesting suggesting that issues that cause the dungeons to require validation are not in fact significantly impacting the overall image quality.

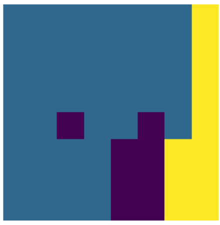


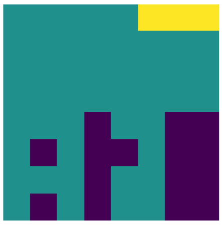
Impressively when looking at the fid score for post validation, again this was the average across 3 runs across 10k epochs. we can see that although it's more varied in terms of output the overall fid score is some 20 points lower than pre validation. Suggesting that the validation method is bringing the generated images closer to the real images.



The difference is especially clear when comparing the two.The dip at 1000 epochs is an interesting one. I think it's due to an effect of the validation process generating excess level geometry.





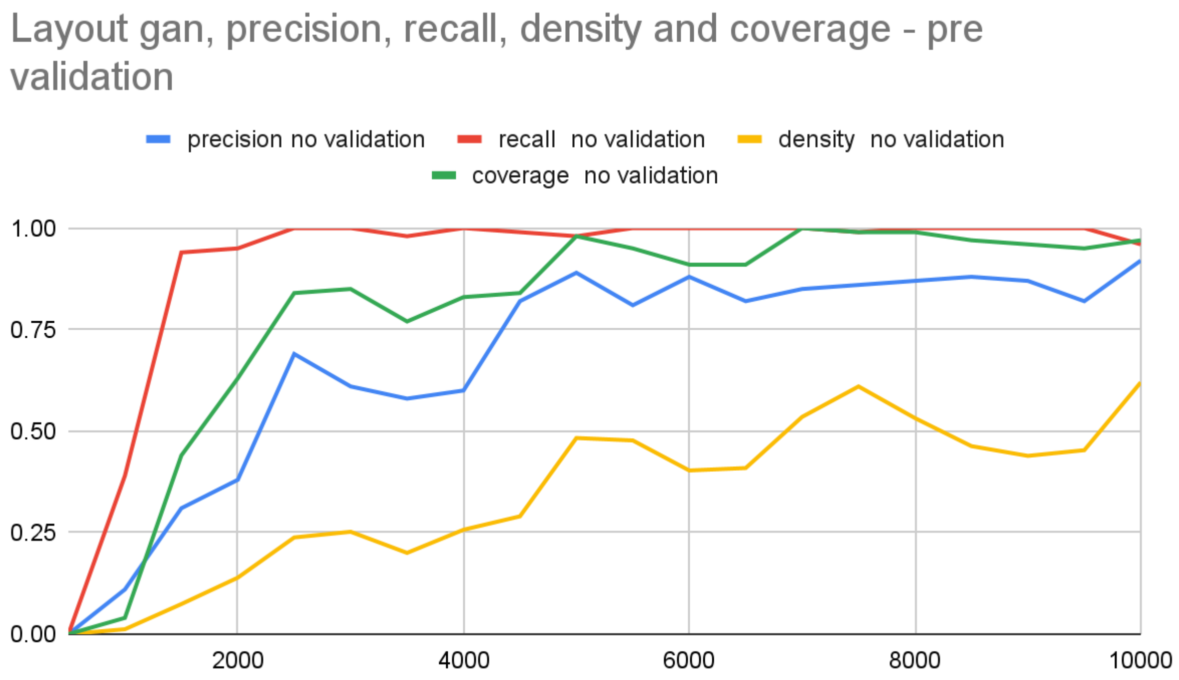


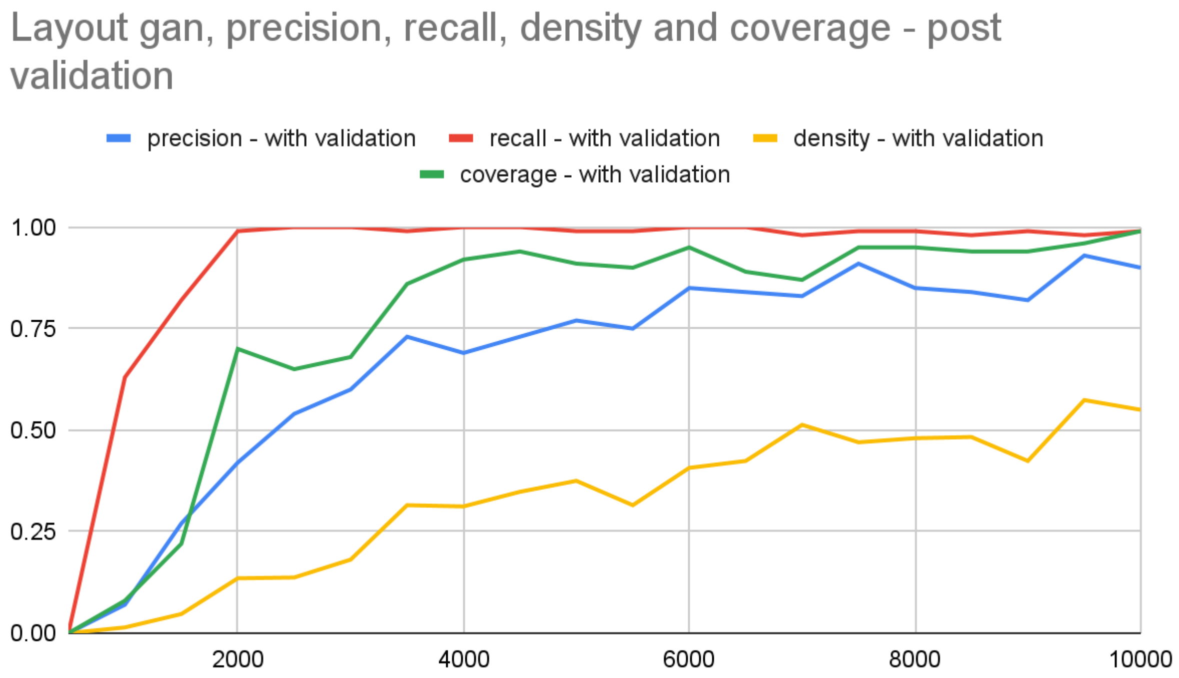
As these three examples show at 1000 epochs when the layout needs to be corrected via the validation process it tends to result in just flooding the grid with rooms.

Precision, recall, density and coverage.

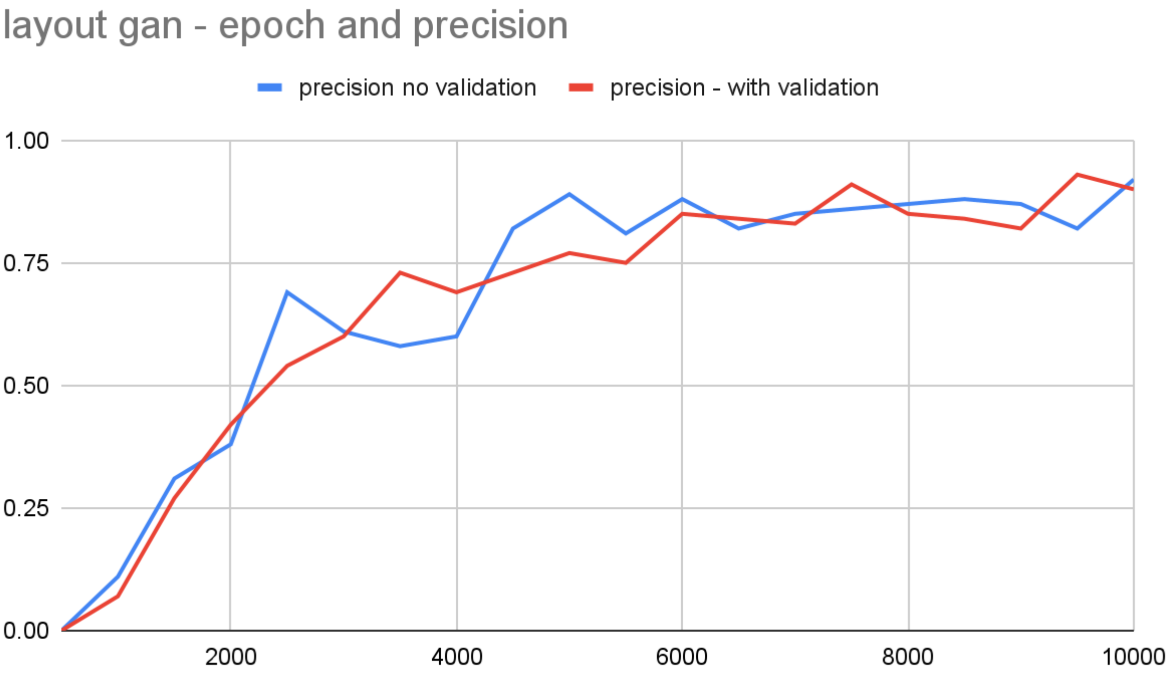
The fid score is ultimately an attempt to assign a numerical value to the generated images. Other types of metrics look at other aspects of the generative model. Metrics such as precision, recall, density and coverage. For these I am greatly indebted to this paper (Naeem et al., 2021) as it's comparative nature means that the code behind it allows for the measuring of all 4 metrics in one go. I am also greatly indebted to the paper's conference video (Naeem, 2021) for it's detailed explanation of each of these metrics.

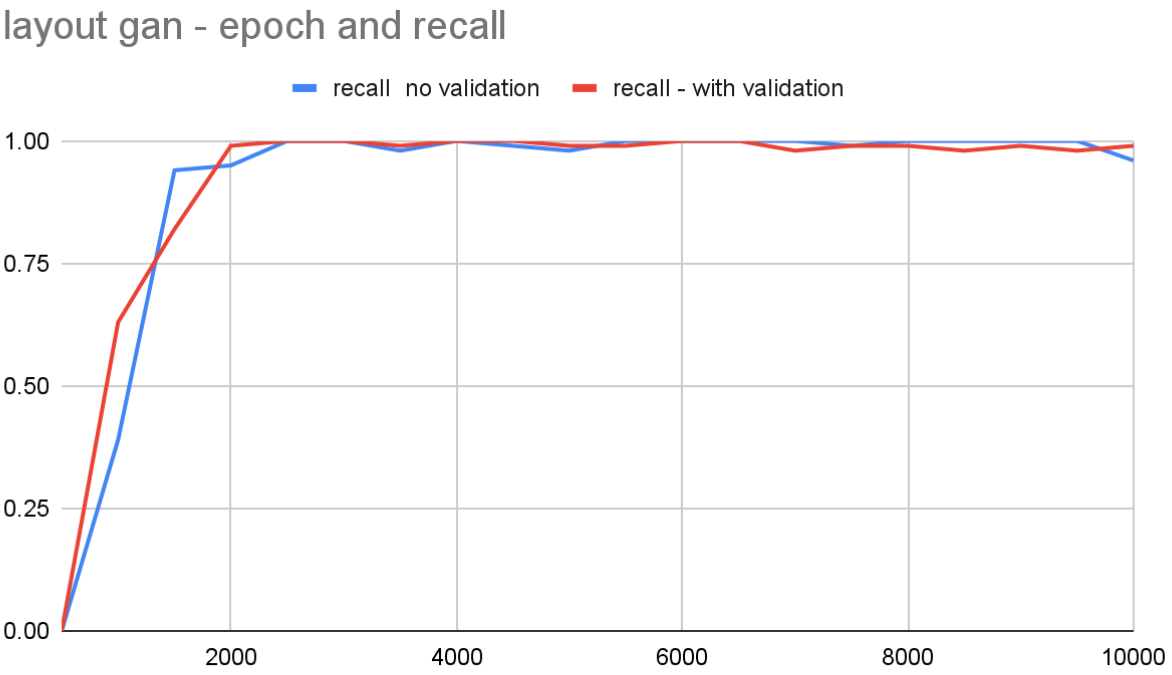
Layout gan

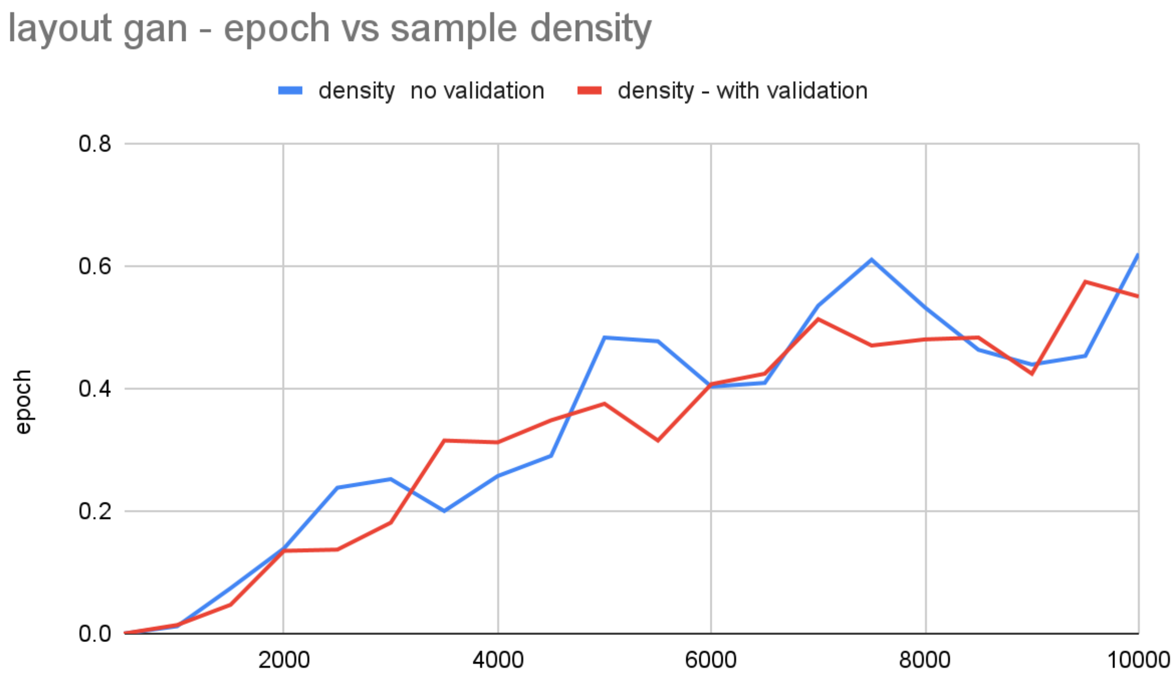


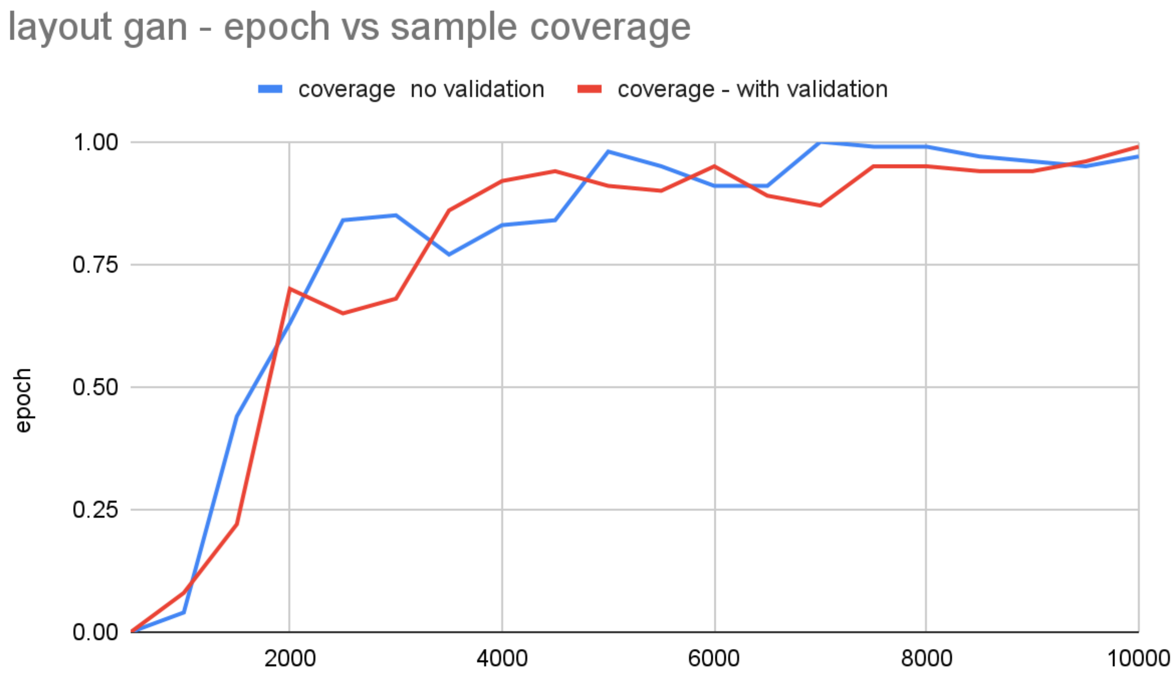


As you can see there's basically no difference between pre and post validation in terms of the scores for the layout gan suggesting that in this metric the validation process has almost zero impact.


layout gan - epoch and precision
precision no validation
precision - with validation
1.00
0.75
0.50
0.25
0.00
2000
4000
6000
8000
10000


layout gan - epoch and recall
recall no validation
recall - with validation
1.00
0.75
0.50
0.25
0.00
2000
4000
6000
8000
10000

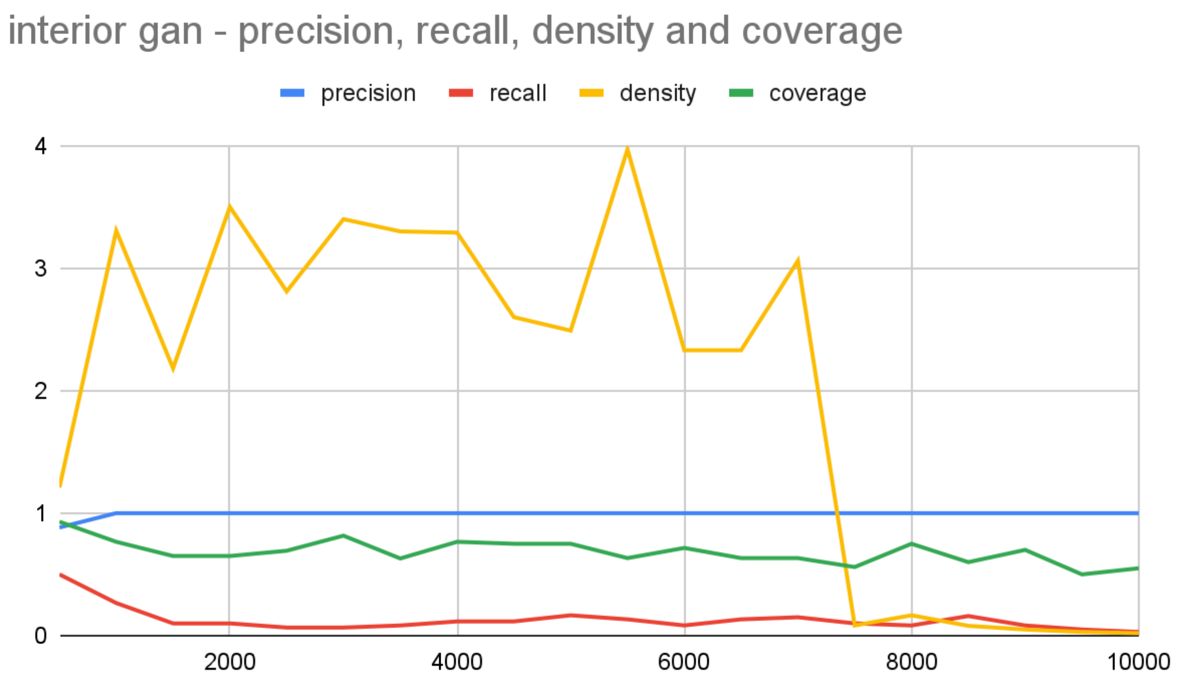




Ultimately the comparison shows that the layout gan is performing fairly similarly with or without validation applied to the output, however I do want to point out the recurring bump in the with validation line at around 4000 epochs. These are interesting, suggesting that it could be advantageous to stop the training early. In terms of looking at the fid score at that point it would definitely be trending down although it would still have a little way to go before it tailed off.

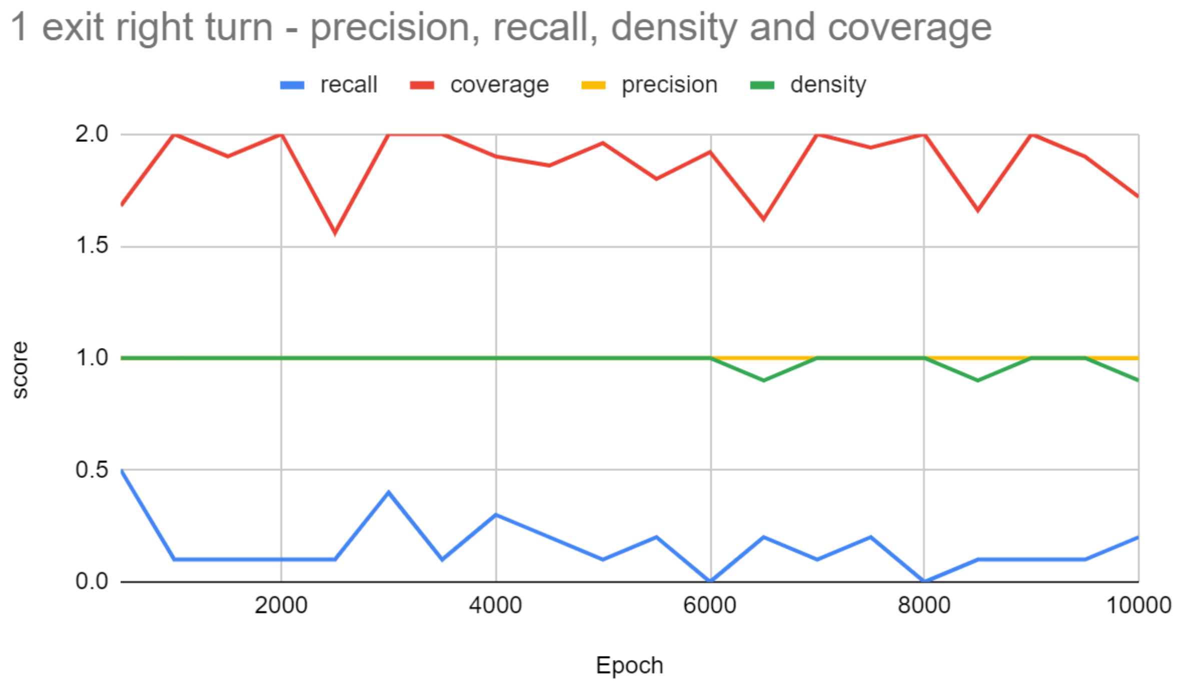
Interior gan

I've also ran this metric against the interior gan both against a generated dataset consisting of 20 of each data class for a total of 120 samples and against each room individually.



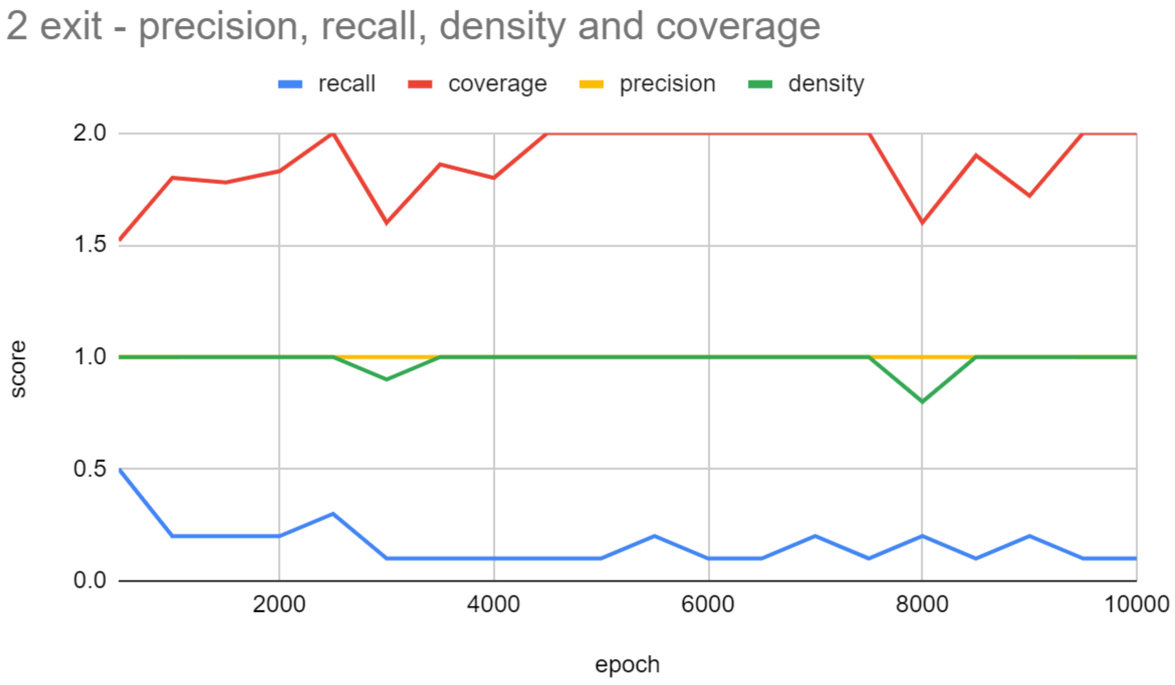
interior gan - precision, recall, density and coverage

1 exit

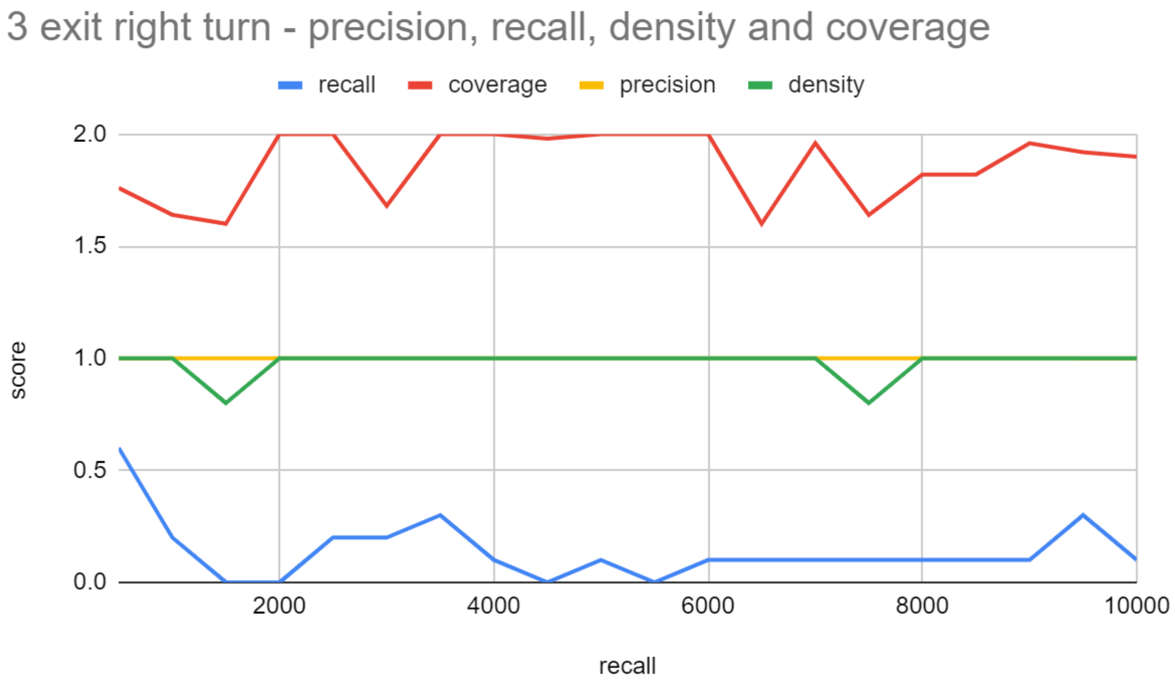


1 exit right turn - precision, recall, density and coverage

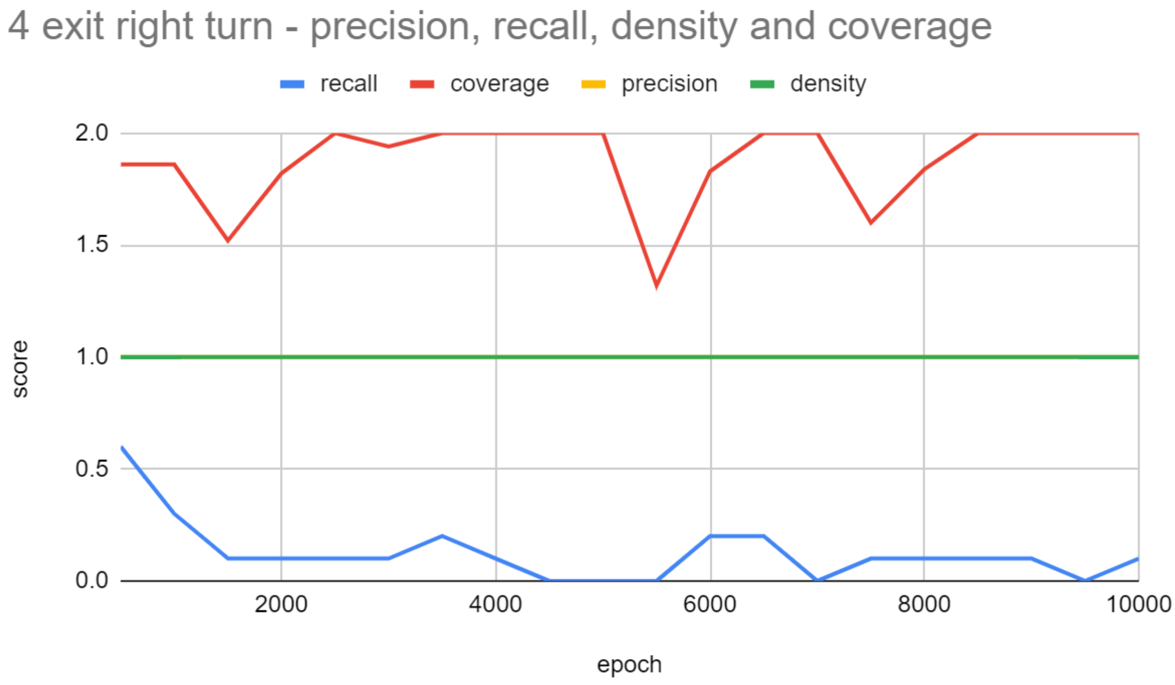
2 exit


2 exit - precision, recall, density and coverage
recall
coverage
precision
density
score
2.0
1.5
1.0
0.5
0.0
2000
4000
6000
8000
10000
epoch

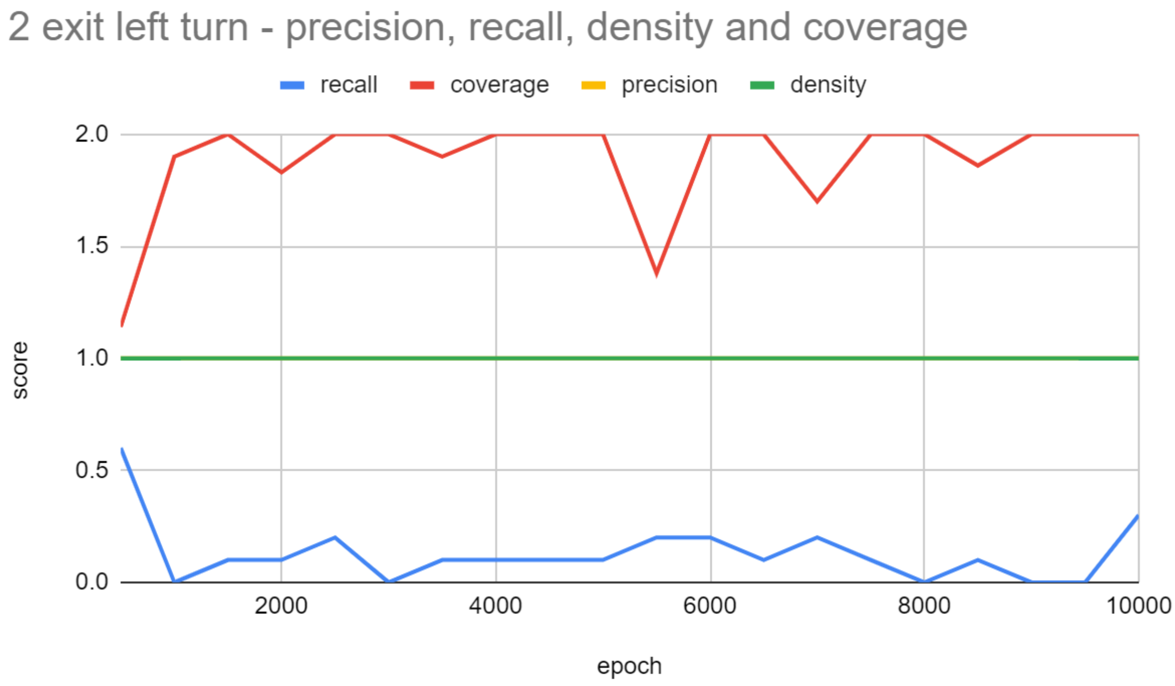
3 exit


3 exit right turn - precision, recall, density and coverage
recall
coverage
precision
density
score
2.0
1.5
1.0
0.5
0.0
2000
4000
6000
8000
10000
recall

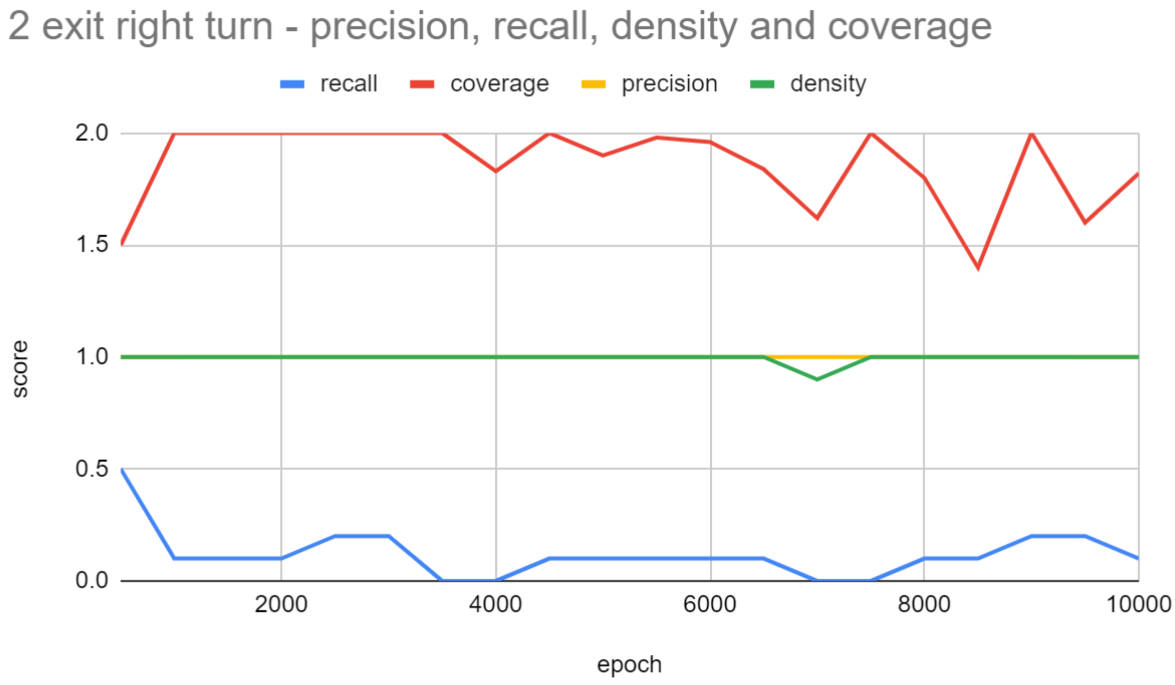
4 exit



2 exit corner left



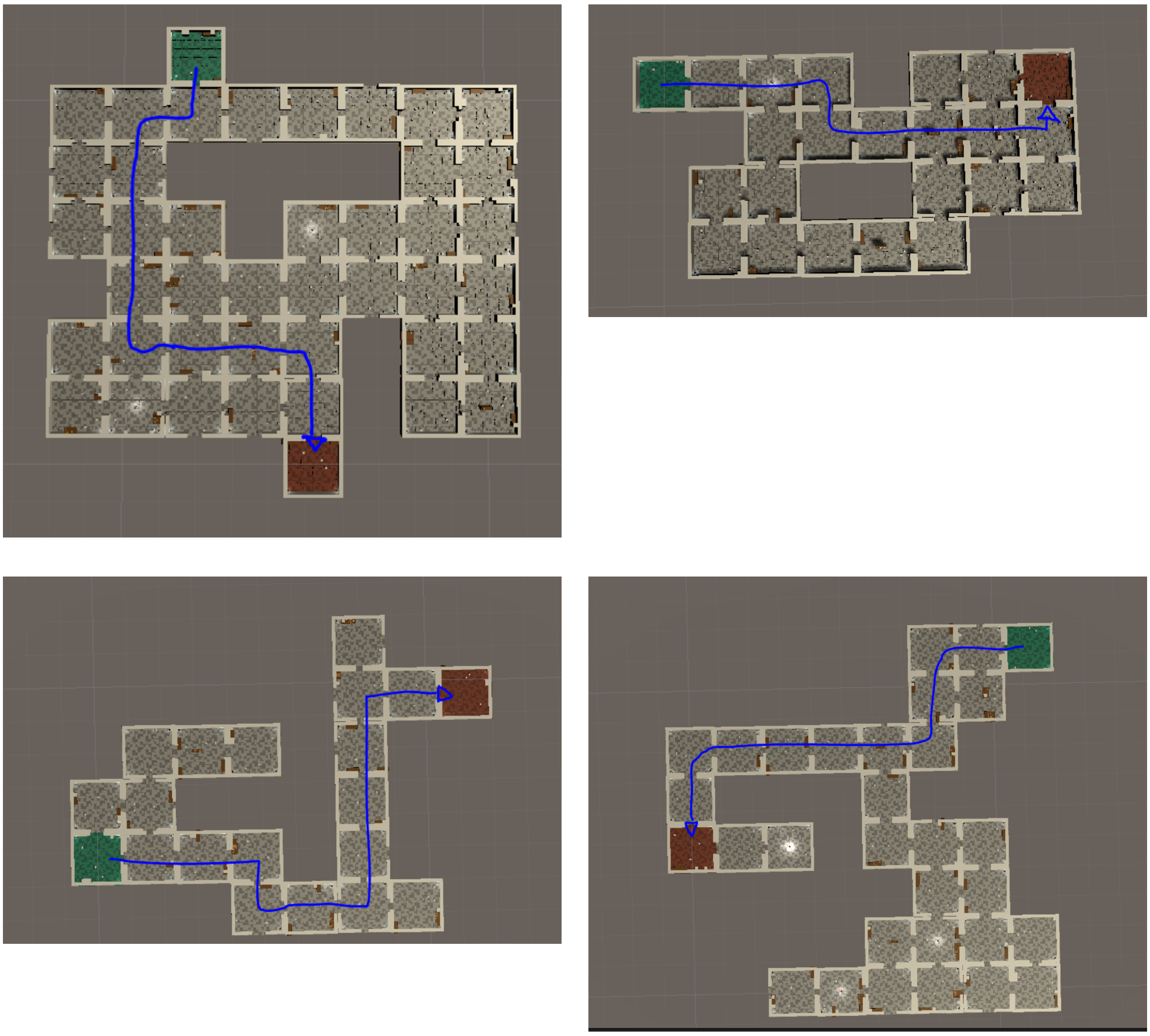
2 exit corner right



Unity examination

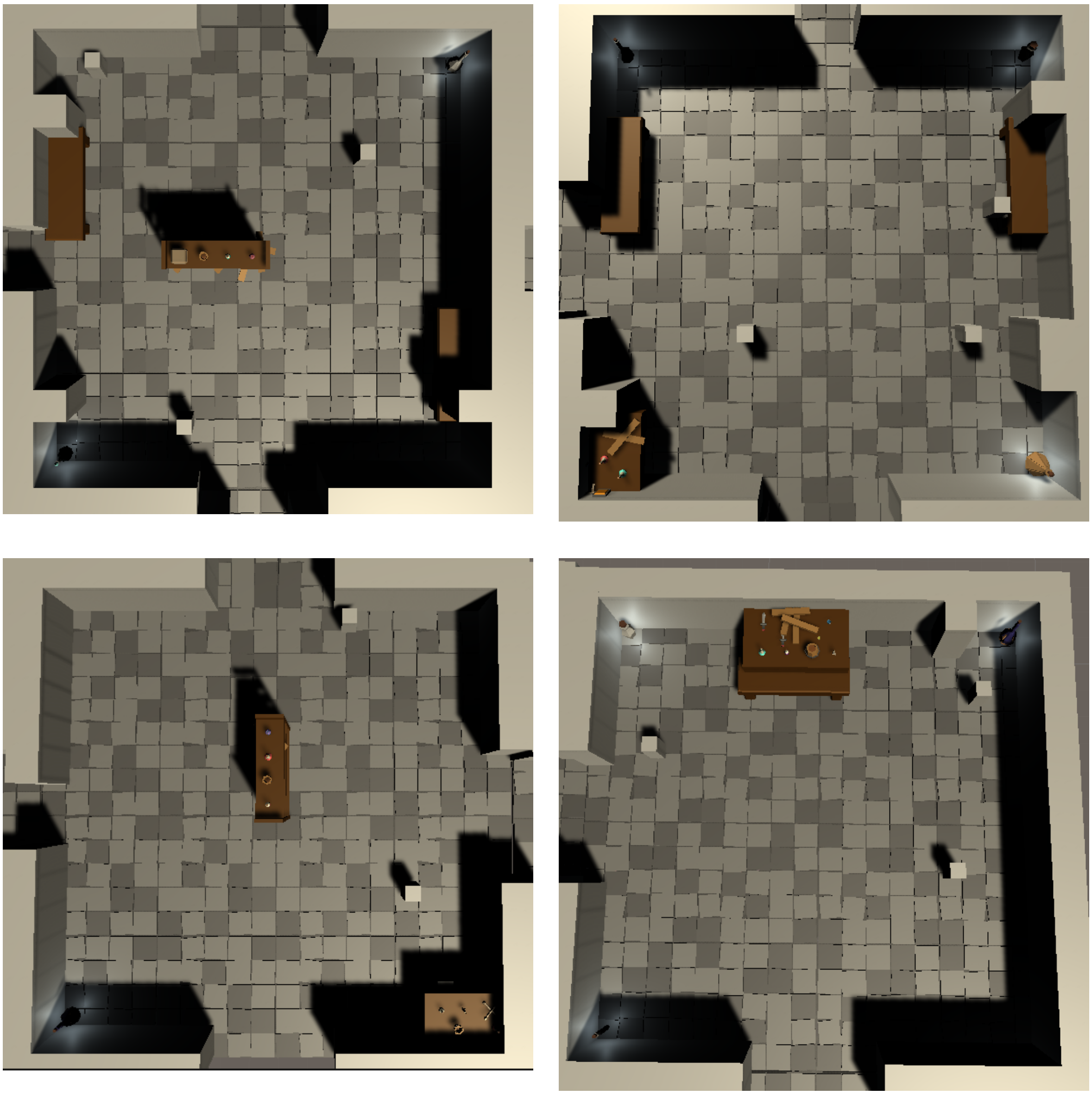
When discussing the end results of this project from within unity the main evaluative metric is the complexity and quality of the resulting dungeons. We can clearly see in terms of overall level layout the results are fairly mixed, on a purely technical level yes this system is generating valid layouts however as you can see from the images where i've drawn on the closest path from start to end in most of these layouts a majority of the layouts do not utilize a majority of the possible rooms, meaning that a player would have very little reason to explore in those locations. You can also see that perhaps a square grid is not ideal for the dungeon layouts because the inherited location forces a poor use of level geometry. A better option would be to spawn in the start room and to recursively spawn in extra rooms using something based on probability of a given room type appearing within the test data.

Examples of dungeon layouts with the path from start to end drawn on.



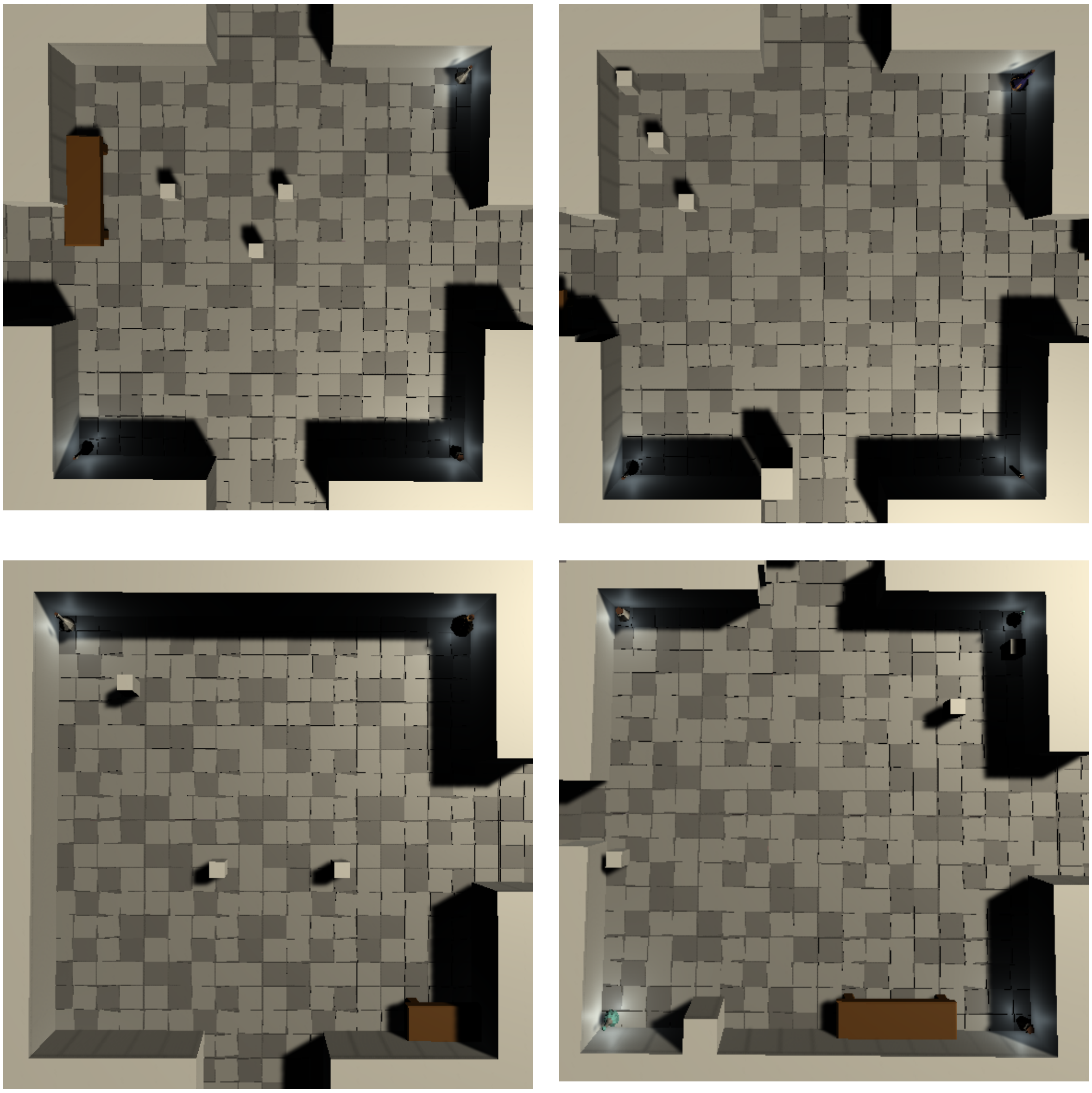
In terms of individual room generation the results are fairly mixed, on one hand there are definitely rooms that are generated containing a decent level of complexity and interesting layouts. It's a purely subjective metric but for me a good level of complexity is defined by having a good variety of interior objects. Should also be noted that a good level gives no accommodation to making sure the room objects are not clipping through the walls - this is because the generated images may be having issues sticking to the rules due to lack of samples.

Examples of rooms that were generated with good levels of complexity.



However these images were by far a minority at a rate of roughly 2 or 3 per level. A majority of these look fairly bare and uninteresting and often with things spawned in some odd locations.

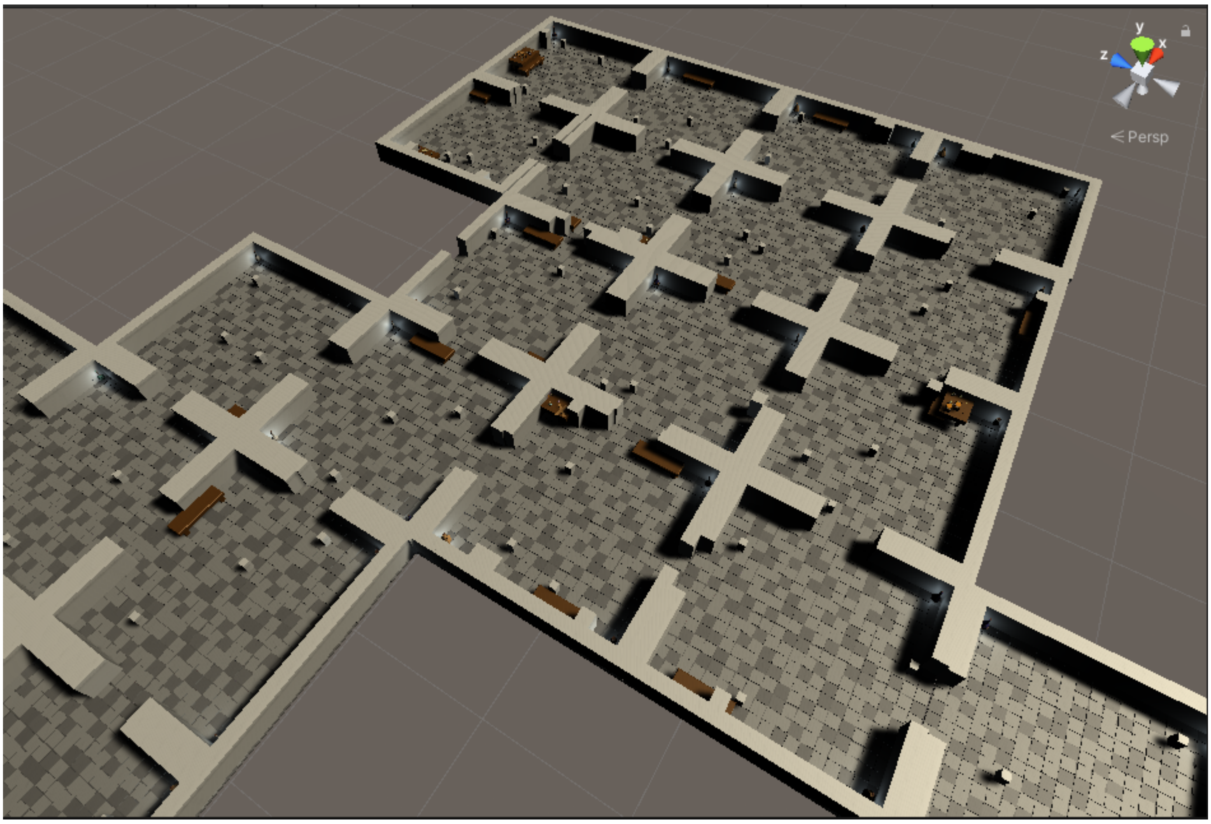
Examples of what most rooms looked like showing them to be bland and uninteresting.



conclusion

In conclusion it can be shown that combining a number of gans together can be used to generate game levels that meet a minimum level of playability; however, the nature of deep learning & gans mean that it's absolutely not ideal to go down this route as there are a multitude of better crafted solutions It would therefore be a better option to simply analyze what you're hoping to achieve and to then apply an existing algorithm be it Perlin noise or Binary space partitioning rather than trying to fit the projects situation in to the deep learning shaped hole.

Issues persist at almost every step of the way, starting with acquiring the training data as the first step with any deep learning system is acquiring or working out what the system is going to be trained on. The system presented in this report was trained on a total of 2200 grids of various sizes this took the best part of a week to put together for comparison the closest previous system to this is an attempt to use gans to generate levels for the original Doom game, this project used roughly 9000 examples (Giacomello, Lanzi and Loiacono, 2018) however they got around any setup time associated with such a large dataset by using a library of human created levels this is one option that could work in an academic situation where the developer is purely looking for a proof of concept, I believe it isn't remotely realistic for a vast majority of instances where a developer is looking to apply deep learning to something. Secondly the issue of actually training the system, obviously before you can use the models to generate you need them to be trained. As for how long you need to train the models for this obviously depends on a vast number of factors but during this project a stark increase in output quality was detected at around 1200 epochs for both the room interior gan and at 3950 epochs for the dungeon layout gan in terms of training time the layout gan was left training overnight for approximately 6 hours until an error with saving the results caused it to crash leaving the system with the most recent gan, the room interior gan took roughly 40 minutes to train. It should also be noted that this is a ludicrously low number of training epochs with the aforementioned comparable project looking at using gans to generate levels for the original Doom training their system for a total of 35k epochs. Presenting a huge delay between writing the system and generating any levels especially when you take into account that this must be performed after every change to the desired output making any deep learning based system extremely inflexible to quickly prototyping new gameplay mechanics.Thirdly, gan outputs traditionally lack the significant variety that a computer game level requires for it to be the sole underlying creative force to level generation. Lack of variety with level generation is a concern for computer game levels because often the fun and playability of a game can almost be directly tied to the variety of levels and should a player detect a lack of variety then their interest would naturally fall. One such cause of this lack of variety is model collapse this is defined as the point when the deep learning system focuses on specific aspects of the training dataset at the expense of others that perhaps are less prominent(Brownlee, 2019)This is catastrophic for a game that relies solely on a deep learning system to underpin the level generation especially when you consider the lost development time sunk in to training the system up until that point. Ultimately this speaks to the opaque 'closed black box' nature of deep learning. The lack of granular control that is inherent within such systems goes beyond the normal lack of control that are found within other procedurally generated system rendering development and debugging potentially even more complicated than if an alternative none deep learning based system was used this is compounded by the non-deterministic nature of deep learning meaning that unlike systems such as bsp there's no guarantee that you each generated dungeon will be of an equal quality owing to the innate nature of different room configurations being better than others. Lastly each room interior is lacking context in the grand scheme of the overall level, this is noticeable mostly when actually playing through a dungeon as game levels have a number of cohesive elements that are designed to inform the player about their progression through the level namely a difficulty curve that scales up as the player progresses through the level and gets closer to the end goal and thematic elements to show how the player is progressing and lastly a notion of pacing which helps the level unfold at an acceptable pace to allow all content in the level to be consumed by the player at a steady rate.



As it currently stands the system has absolutely no consideration for any of these aspects which leads to disjointed and uneven levels in terms of room content.This could potentially have been avoided if the system was designed in such away as to sequentially request rooms and incorporated more data such as the location of the room in relation to the expected player start and end therefore allowing a more cohesive narrative to be built around the level. At this point it's important to ask what benefits does applying machine learning to level generation actually have? The main benefit that can be seen is should during the development of the overall game the internal map data change drastically a machine learning based system would be noticeably more flexible to the required changes need nothing beyond retraining on some new dataset vs potentially being entirely scrapped where it an non machine learning based system. Beyond that benefit the conclusion is a fairly definitive yes that it is technically possible to generate game levels via deep learning, however due to the above mentioned issues such a process is in no way preferable to any of the myriad of already existing algorithm and system and that even though some of these issues are not insurmountable it's inescapable that almost by the design of deep learning the results of such a system will always lead to the conclusion that deep learning although technically capable is not suitable for generating game levels within this context, there are previous attempts at generating levels for different sorts of games such as the previously mentioned method of generating early super mario game levels via lstm based models which show immense promise.

references

Chaillou, S., 2021. *ArchiGAN: a Generative Stack for Apartment Building Design | NVIDIA Developer Blog*. [online] NVIDIA Developer Blog. Available at: <https://developer.nvidia.com/blog/archigan-generative-stack-apartment-building-design/> [Accessed 21 September 2021].

Wang, D., Fu, Y. and Wang, P., 2021. *Reimagining City Configuration: Automated Urban Planning via Adversarial Learning*. [online] Arxiv.org. Available at: <https://arxiv.org/pdf/2008.09912.pdf> [Accessed 21 September 2021].

2018. *6 Ways Machine Learning will be used in Game Development*. [online] Available at: <https://www.logikk.com/articles/machine-learning-in-game-development/> [Accessed 14 September 2021].
Steam.com. n.d. *The Binding of Isaac on Steam*. [online] Available at: <https://store.steampowered.com/app/113200/The_Binding_of_Isaac/> [Accessed 14 September 2021].

Thispersondoesnotexist.com. 2021. *This Person Does Not Exist*. [online] Available at: <https://thispersondoesnotexist.com/> [Accessed 14 September 2021].

Gpfault.net. 2021. *Using Perlin Noise to Generate 2D Terrain and Water*. [online] Available at: <https://gpfault.net/posts/perlin-noise.txt.html> [Accessed 14 September 2021].

Gumin, M., 2016. *GitHub - mxgmn/WaveFunctionCollapse: Bitmap & tilemap generation from a single example with the help of ideas from quantum mechanics*. [online] GitHub. Available at: <https://github.com/mxgmn/WaveFunctionCollapse> [Accessed 14 September 2021].

My Abandonware. n.d. *.kkrieger: Chapter 1 (Windows)*. [online] Available at: <https://www.myabandonware.com/game/kkrieger-chapter-1-cl1>.

Statt, N., 2019. How artificial intelligence will revolutionize the way video games are developed and played. [online] The Verge. Available at: <https://www.theverge.com/2019/3/6/18222203/video-game-ai-future-procedural-generation-deep-learning> [Accessed 14 September 2021].

Giacomello, E., Lanzi, P. and Loiacono, D., 2018. *DOOM Level Generation using Generative Adversarial Networks*. [online] arXiv.org. Available at: <https://arxiv.org/abs/1804.09154> [Accessed 14 September 2021].

Paperswithcode.com. 2021. *Papers with Code - CIFAR-10 Benchmark (Image Generation)*. [online] Available at: <https://paperswithcode.com/sota/image-generation-on-cifar-10> [Accessed 14 September 2021].

Engadget.com. 2021. Here's how 'Minecraft' creates its gigantic worlds.[online] Available at: <https://www.engadget.com/2015-03-04-how-minecraft-worlds-are-made.html?guccounter=1&guce_referrer=aHR0cHM6Ly93d3cuZ29vZ2xlLmNvbS8&guce_referrer_sig=AQAAAMqAX8Rw5PCdgfGTdPHK6H04wuKhZw2sEq-QYt4AX1vCCWM4b8zCxdxqZX8O5kAV9ysxiP0sK3bd8fucVOKSlykzHPhgsybZP6VfGij5yZwknH6LD3kxixO4kVdQPpjLicV-0x8U2godHnE9eXTKH59i9Ig2bSHkHo9uYj0-5p1G> [Accessed 14 September 2021].

Howtomakeanrpg.com. n.d. *Beneath Apple Manor*. [online] Available at: <https://howtomakeanrpg.com/r/l/g/apple-manor.html> [Accessed 14 September 2021].
Bfnightly.bracketproductions.com. n.d. *BSP Room Dungeons - Roguelike Tutorial - In Rust*. [online] Available at: <https://bfnightly.bracketproductions.com/chapter_25.html> [Accessed 14 September 2021].

Naeem, M., Oh, S., Uh, Y., Choi, Y. and Yoo, J., 2021. *Reliable Fidelity and Diversity Metrics for Generative Models*. [online] arXiv.org. Available at: <https://arxiv.org/abs/2002.09797> [Accessed 22 September 2021].

Summerville, A. and Mateas, M., 2016. *Super Mario as a String: Platformer Level Generation Via LSTMs*. [online] arXiv.org. Available at: <https://arxiv.org/abs/1603.00930> [Accessed 14 September 2021].

J. Summerville, A., Behrooz, M., Mateas, M. and Jhala, A., n.d. [online] Fdg2015.org. Available at: <http://www.fdg2015.org/papers/fdg2015_paper_04.pdf> [Accessed 14 September 2021].

Medium. 2021. *Game Level Design with Reinforcement Learning*. [online] Available at: <https://medium.com/deepgamingai/game-level-design-with-reinforcement-learning-52b02bb94954> [Accessed 14 September 2021].

GitHub. n.d. *GitHub - soumith/ganhacks: starter from "How to Train a GAN?" at NIPS2016*. [online] Available at: <https://github.com/soumith/ganhacks> [Accessed 14 September 2021].

Netron.app. 2021. *Netron*. [online] Available at: <https://netron.app> [Accessed 14 September 2021].

Singh, S., n.d. *Leaky ReLU as an Activation Function in Neural Networks*. [online] Deep Learning University. Available at: <https://deeplearninguniversity.com/leaky-relu-as-an-activation-function-in-neural-networks/> [Accessed 14 September 2021].

Medium. n.d. A Practical Guide to ReLU. [online] Available at: <https://medium.com/@danqing/a-practical-guide-to-relu-b83ca804f1f7> [Accessed 14 September 2021].

OpenGenus IQ: Computing Expertise & Legacy. 2021. *Connected Component Labeling: Algorithm and Python Implementation*. [online] Available at: <https://iq.opengenus.org/connected-component-labeling/> [Accessed 14 September 2021].

Brownlee, J., 2019. *How to Identify and Diagnose GAN Failure Modes*. [online] Machine Learning Mastery. Available at: <https://machinelearningmastery.com/practical-guide-to-gan-failure-modes/> [Accessed 14 September 2021].

Naeem, F., 2021. *Reliable Fidelity and Diversity Metrics for Generative Models - ICML2020*. [video] Available at: <https://www.youtube.com/watch?v=_XwsGkryVpk> [Accessed 22 September 2021].