Maastricht University

Master Thesis

Utilizing generative adversarial networks for stable structure generation in a physics-based simulation.

by

Frederic Marvin Abraham
i6262598

DKE: Dr. Matthew Stephenson

University of Maastricht
Data Science & Knowledge Engineering
Maastricht, November 10, 2022

Contents

Declaration of independence

I hereby declare that I have produced the present work independently and by my own hand, without the unauthorized assistance of others and exclusively using the sources and aids listed.

The independent and self-contained production assures in lieu of oath:

Maastricht, the November 10, 2022

.....

Signature

Abbreviations

AI Artificial Intelligence

ML Machine Learning

NN Neural Network

CNN Convolutional Neural Network

DL Deep Learning

CV Computer Vision

WS WebSocket

RMSP Root Mean Squared Propagation

EM Earthmover Distance

MLP Multilayer Perceptron

LSTM Long Short-Term Memory

RNN recurrent neural network

VAE Variational Autoencoder

GAN Generative Adversarial Network

CGAN Conditional GAN

WGAN Wasserstein GAN

WGAN-GP Wasserstein GAN with gradient penalty

DCGAN deep convolutional generative adversarial network

MC mode collapse

PCG Procedural Content Generation

PCGML Procedural Content Generation via machine learning

MNIST Modified National Institute of Standards and Technology database

VGLC Video Game Level Corpus

LVE latent variable evolution

CMA-ES Covariance Matrix Adaptation Evolutionary Strategy

LeakyReLU leaky rectified linear units

RRD Recursive Rectangle Decoding

CD Confidence Decoding

Abstract

Procedural Content Generation via machine learning (PCGML) using deep generative models, such as GANs, has attracted attention as a technique to automate level generation. GANs are the construct of two adversarial networks, training one another to differentiate between real and generated and are said to be the most interesting idea in the last ten years in Machine Learning. (LeCun 2016) Exploring applications of GANs on various data representations reduces the required transfer learning cost and can reduce the development cost when used to generate video game content.

Previous applications of GANs-based level generation are mostly limited to game domains with tile-based level representations. This thesis proposes various ways to encode a 2D physics-based real-valued block structure in combination with their respective decoding algorithms in order to train two distinct GAN-models with the original and state-of-the-art training algorithm to generate stable block structures. The most suitable combination of data representation and GAN-model is evaluated by searching the proposed decoding algorithm's parameter space. Using the best parameter set, an extensive simulation is done to generate stable structures with specific characteristics.

The main results of this thesis are the encoding and decoding algorithms that work with imperfect generated structure representations and the gained insight into the relationship between data representations in combination with GAN models.

1 Introduction

A majority of techniques that utilize Artificial Intelligence (AI) use supervised learning techniques, while unsupervised training is a relatively unsolved research area (Jabbar et al. 2020). The latest impressive achievements in the task of text-conditional image synthesis by the DALL-E 2 model (Ramesh et al. 2022), (Examples attached in Figure A.1) resulted in one of the highest media attention when it comes to generative machine learning models. Some went so far as to worry about the existence of creative careers when a generated artwork (Figure A.2) won first place at the Colorado State Fair's fine arts competition[1].

Even so, the latest advancements of art generation with AI use the same Machine Learning (ML) models, and the requirements of generated content in video games are usually higher and reach in purpose from visual assets to generating the main gameplay loop of the game. The field of Procedural Content Generation (PCG) has a long history in video game development and was initially used to compress game data (Amato 2017) to extend the amount of content a game could contain. PCG is defined as the creation of content automatically through algorithmic means (Yannakakis and Togelius 2011) and Procedural Content Generation via machine learning (PCGML) uses machine learning models trained from existing video game content and has received increasing research attention recently (Liu et al. 2020).

Different types of algorithms have been used to generate content. Forbs[2] states that one of the most promising ML models are the GANs which is supported by scientist when comparing the amount of published research that is related to GAN. GANs are the construct of two adversarial networks, which play a min-max game of one network that generates new content and a second that discriminates the generated content to differentiate between real and generated (Goodfellow et al. 2014). The generated content can use almost any underlying data structure; therefore, the two networks can be of various architectures depending on the data structure. Due to the simplicity of having two Neural Network (NN) train one another, many

1. Colorado State Fair's fine arts competition
2. The Next Generation Of Artificial Intelligence

advancements can be made in almost every aspect of Generative Adversarial Network (GAN). The architecture of each network, the controllability, the training process, scaling, adaptation and application in different domains are a few research directions that are investigated.

The adaptation of GANs in video game content generation is less researched due to the usually high unreliability of the generated content. While GANs have promising capabilities, they were previously only used in discrete domains, for example, tile-based games such as Super Mario with no physical constraints (Volz et al. 2018) and grid-like positioning of blocks. The physics-based simulation domain is a 2D physics-based real-valued block simulation in the form of structures in science birds.

1.1 Motivation

There are two main motivations for exploring content generator neural networks in new and difficult domains. The first is of monetary nature, as a significant portion of the budget in video games is spent to create media used in every aspect of a game. The development of better generators able to aid content creators will improve their output and consequently reduce the cost of video game development Amato (2017).

The second motivation for researching GANs is their adaptability for new tasks and domains by changing the kind of data structures and networks used. The fact that they are unsupervised learning methods that do not require the laborious task of labelling data. The range and variety where GANs are already applied in general and in particular in video game content generation, such as textures, terrain, faces and, to some extent, levels, motivates the research to reduce the required transfer work required.

The domain mentioned above with the physical constraint differentiates itself from the usual application environment of GANs. Therefore, exploring new domains reduces the transfer work required in applications relying on a similar data structure or having similar external constraints. For example, the 2D physics-based real-valued block simulation could be extendible into the third dimension or more complex shapes.

Lastly, game-playing agents that require a lot of training variety can also benefit from having a good content generator. A GAN that can produce solvable and challenging levels could help train the AI to better adapt to a higher variety of challenges. At

the same time, an improved AI could enhance the level generator in an adversarial manner with better validation capabilities.

1.2 Research Goals

In this thesis, in the related work chapter, an overview of how GANs function, explained with their initial training process, their difficulties and how the state-of-the-art training algorithms work and circumvent previous problems is given. Followed by their application in content generation and the algorithm that generate structures in the same science birds domain with different approaches.

The main contribution of this thesis is the exploration of different structure encodings, that are able to capture the real-valued domain and investigate their capabilities to function as the basis for training GANs. In order to recreate structures from the generated structure representation, two decoding algorithms have been developed with both their advantages and disadvantages.

In total 11 GAN networks have been trained on various datasets with the goal to explore the different characteristics of the used GAN architectures, training methods, structure representations and decoding methods. To evaluate produced structures, the open-source science bird implementation is extended to allow for fast structure evaluation and data collection. Also, an application was implemented to interact with various aspects of the thesis.

1.2.1 Research Questions

The resulting research questions are as follows:

- In the domain of a 2D physics-based real-valued block simulation, how can a structure be encoded into a data representation capable of being used in the training process of a GAN?
- Given the encoding of a structure, how can it be decoded into a usable structure representation?
- Are GANs suitable for generating stable structures in a 2D physics-based block domain.
- What GAN architectures, training algorithms, encoding and decoding parameter compositions are usable for training and generating stable structures.

1.3 Science Birds

The physics-based simulation, which is the subject of this thesis, is represented in the domain of physics-based puzzle games similar to the popular game Angry Birds.

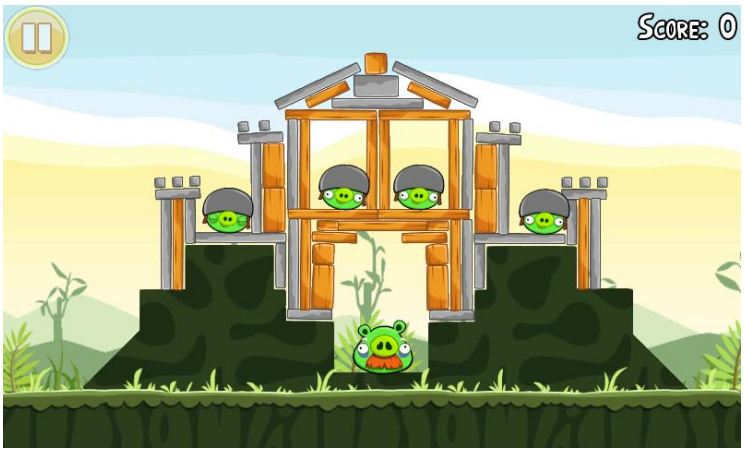


Figure 1.1: An example structure from angry birds.[3]

A level of Angry Birds consists of one or more 2D structures built out of various blocks in which pigs are placed. An example structure is given in figure 1.1. The player's objective is to shoot birds with a slingshot at these structures in order to kill the pigs by utilizing either the instability of the structure or by aiming at the pigs directly.

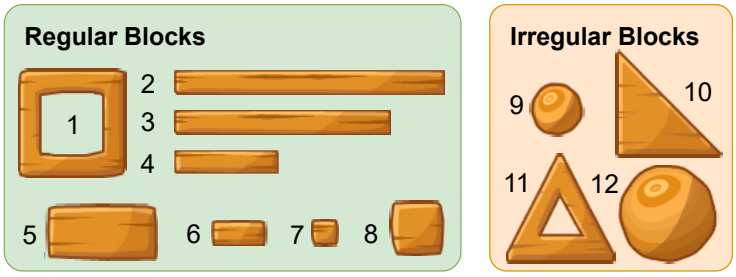


Figure 1.2: The available blocks in Science Birds are separated into regular and irregular blocks.

Figure 1.2 shows different available blocks, the used names throughout the thesis are listed in Table A.1. The blocks come in 3 different materials, ice, wood and stone, each with higher durability than the previous one. Irregular blocks are usually not part of the structural integrity of the structure and are usually used in a decorative manner. As the focus of this thesis is stable structure generation, the irregular blocks are not considered in this thesis.

3. Source of the structure image.

```
<?xml version="1.0" ?>
<Level width="2">
    <Camera x="0" y="2" minWidth="20" maxWidth="30"/>
    <Birds>
        <Bird type="BirdRed"/>
        <Bird type="BirdRed"/>
        <Bird type="BirdRed"/>
    </Birds>
    <Slingshot x="−8" y="−2.5"/>
    <GameObjects>
        <Platform type="Platform" material="" x="−5.69" y="−3.19" rotation="0.0"/>
        <Platform type="Platform" material="" x="−5.07" y="−3.19" rotation="0.0"/>
        <Block type="RectTiny" material="wood" x="−4.87" y="−2.665" rotation="90.0"/>
        <Platform type="Platform" material="" x="−4.45" y="−3.19" rotation="0.0"/>
        <Platform type="Platform" material="" x="−3.83" y="−3.19" rotation="0.0"/>
        <Block type="RectTiny" material="stone" x="−4.14" y="−2.665" rotation="90.0"/>
        <Block type="RectMedium" material="stone" x="−4.14" y="−2.34" rotation="0.0"/>
        <Platform type="Platform" material="" x="−3.21" y="−3.19" rotation="0.0"/>
        <Block type="RectTiny" material="stone" x="−3.41" y="−2.665" rotation="90.0"/>
        <Platform type="Platform" material="" x="−2.59" y="−3.19" rotation="0.0"/>
    </GameObjects>
</Level>
```

Listing 1.1: Level description

An example level XML representation is given in listing 1.1. The level is described by what birds are used, the slingshot position, and a list of game objects. A game object, classified by the type, is either one of the aforementioned blocks or a platform, which is an indestructible object not affected by physics or a block of TNT. A game object is further defined by its real-valued x and y coordinates, its rotation, and, if applicable, the material.

Procedural level generation research usually uses the Science Birds game[4] (Ferreira and Toledo 2014), a clone of Angry birds developed in Unity that provides an interface for remote AI execution. The procedural level generation field in this game domain is active, and a level generation competition is held yearly (Stephenson et al. 2019).

The science bird domain has two main difficulties when developing a level generator. Firstly the high degree of freedom in the level/structure design. The real-valued positioning of each block allows for a large number of varied structure designs. Combined with the second challenge, that a small error in the placement of each block can lead to the collapse of the structure immediately after the game start, which makes the level unplayable. The generator needs to consider if blocks overlap one another, which would result in unpredictable behaviour by the physics engine, or if they are placed sufficiently close in order to not cause an impact when simulated.

4. Science Birds Source Code: https://github.com/lucasnfe/science-birds

2 Related Work

This chapter reviews related work in the field of PCG with a focus on the ML approaches. Starting with a brief introduction on the origin of PCG and an overview of PCGML techniques. The main focus of this master thesis is to utilize generative adversarial networks to generate levels in a real-valued block world. The following section gives an insight into the main concepts of GANs, their development, difficulties and different advancements to combat these problems. Closing this chapter with examples of PCGML using GANs followed by an overview of the previous PCG for angry birds.

2.1 History of PCG

PCG refers to the creation of content automatically through algorithmic means (Yannakakis and Togelius 2011). Over the years, the field of PCG became populated with various algorithms aiming to achieve different goals. The original problem PCG addressed was memory limitations for storing larger video game levels on computers in the 1980s (Fontaine et al. 2020). One of the earliest games that incorporated PCG to generate adventures and levels was "Beneath Apple Manor" in 1978 (Doull 2015) followed by the more famous and highly influential game Rogue (Doull 2016) in 1980, which is the progenitor of following games in the "Rouguelike" category. The original dungeon generation algorithm created a level by generating several rooms connected with corridors in a 2d grid-based environment. When modern game development began focusing more on realistic graphics in the 1990s, many procedural modelling algorithms were developed for generating environmental aspects, such as textures, plants and terrain. (Smelik et al. 2014).

The master thesis focuses on the more recent area of PCGML, which utilises the latest advancements in ML to generate new content. PCGML is defined as the generation of game content using machine learning models trained on existing content (A. Summerville et al. 2017).

2.2 PCGML

The application of ML, in particular Deep Learning (DL), led to increased capabilities and application methods to learn from a large amount of data and has won numerous contests in pattern recognition. (Schmidhuber 2015) The major difficulty with many of the problems DL excels at is a vast number of features with different levels of importance. Goodfellow et al. (2016) describe DL as the solution to the central problem in representation learning by introducing representations expressed in terms of other, simpler representations.

In PCGML, various models are used for content generation. A. Summerville et al. (2017) surveyed machine learning methods for content generation such as long short-term memory (LSTM) networks, autoencoders, deep learning and generative adversarial networks. The other category of content generation methods falls into the search-based methods, for example, evolutionary algorithms, and solver-based methods, which try to maximize an objective while preserving a specific constraint, and constructive methods using grammars. Other machine learning models used in PCG that are not based on neural networks are Markov-Models, one-dimensional n-gram models, clustering, and matrix factorization (A. Summerville et al. 2017).

2.3 Generative Adversarial Networks

The chief AI Scientist at Facebook LeCun (2016) described Generative Adversarial Network (GAN) as “the most interesting idea in the last ten years in Machine Learning”. The central concept of GANs is the adversarial idea, which, from a game-theoretical point of view, is to define the task as a game between two opposing systems trained in an adversarial manner to reach a zero-sum Nash equilibrium (Moghadam et al. 2021). It has been successfully applied in many areas, such as machine learning, artificial intelligence, computer vision and natural language processing. (Gui et al. 2021) Another public example of how an adversarial system exceeded previous achievements is when the AlphaGo model (Silver et al. 2016) defeated the top human Go player. Parts of AplhaGo utilized two networks that were trained by playing against itself.

2.3.1 GAN Introduction

Goodfellow et al. (2014) introduced generative adversarial networks as a framework for estimating generative models via an adversarial process in which two models are trained simultaneously. He described GANs as frameworks consisting of two models, which can be any kind of network, instead of one coherent model architecture. This change is done in recent literature and can be referred to interchangeably. Figure 2.1 visualizes the general structure of a GAN framework consisting of the two models.

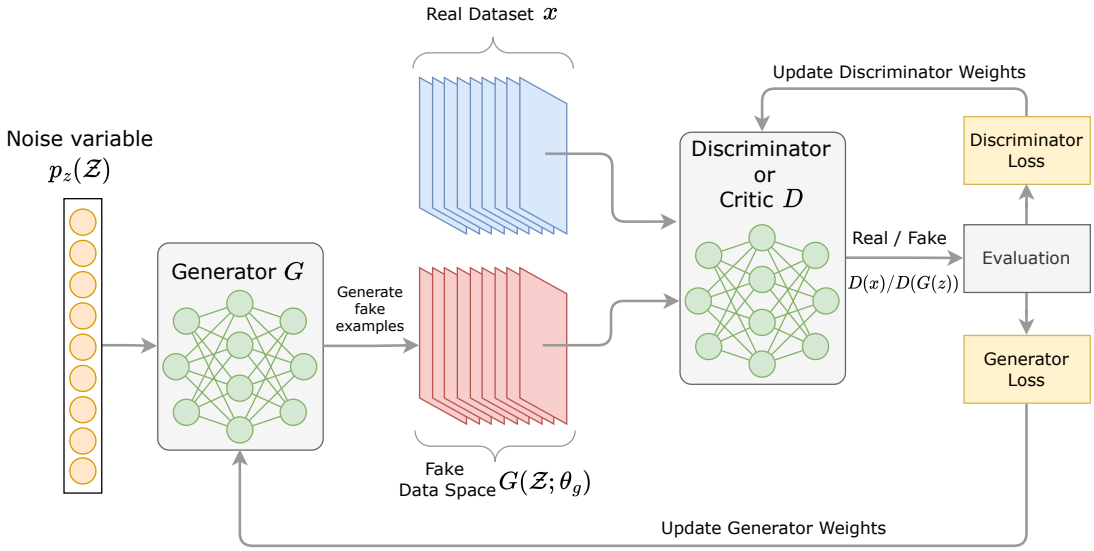


Figure 2.1: Overview of a generative adversarial network

The two models mentioned above are the generator that tries to capture the data distribution of the training data and the discriminative model, also called the critic, which tries to differentiate between samples drawn from the training data and samples generated by the generator. The generator is trained to maximize the probability that the discriminator mistakes its generated example as drawn from the actual distribution. The GAN can be trained through backpropagation if the generator and the discriminator are Multilayer Perceptron (MLP).

Goodfellow et al. (2014) describe the training process, visualized in Figure 2.1, as a two-player minimax game in the following steps:

1. Create a noise variable $p_z(\mathcal{Z})$ which functions as input to the generator.
2. The generator, which tries to learn the distribution p_g over the data x, is defined as a mapping from the random input space into the data space $G(\mathcal{Z};\theta_g)$. G is a differentiable function due to being an MLP with parameters θ_g.

3. The discriminator is defined as a function $D(x;\theta_d)$ that maps from the data space to a single scaler. The scaler $D(x)$ is defined as the probability of x being drawn from the real data distribution.
4. The discriminator is trained to classify real and fake data to maximize the probability of assigning the correct label. The error in assigned labels is the loss used to update the weights of the discriminator.
5. The generator tries to minimize $\log(1 - D(G(z))$. In other words, given the noise variable z, the generated example shall receive a probability close to 1 from the real dataset. The loss for the generator is the amount of images correctly identified as generated which is used to update its weights.

The description of the training process accumulates into the aforementioned two-player minimax game with the value function $V(G, D)$:

$$\min_G \max_D V(D, g) = \mathbb{E}_{x \in p_{data}(x)}[logD(x)] + \mathbb{E}_{z \in p_z(z)}[\log 1 - D(G(z))] \tag{2.1}$$

Solving this problem and "finding the Nash equilibrium is a very difficult problem. [As the] [...] cost functions are non-convex, the parameters are continues, and the parameter space is extremely high-dimensional." (Salimans et al. 2016)

2.3.2 Difficulties in GAN Training

Training a GAN has been repeatedly stated to be a challenging problem. (Arjovsky and Bottou 2017; Salimans et al. 2016; Gui et al. 2021) The main challenges that can be observed when training a GAN are mode collapse (Goodfellow et al. 2014), the discriminator loss converging quickly to zero and therefore does not provide a sufficient update to the generator (Arjovsky and Bottou 2017) and difficulties in making the generator and discriminator converge. (Radford et al. 2015)

2.3.2.1 Mode Collapse

The primary reason for failure of GANs is the mode collapse (MC) problem, in which the generator collapses to a parameter setting in which it learns a mapping of different input z values to the same output point. (Salimans et al. 2016; Goodfellow 2017) Jabbar et al. (2020) describe it as the most crucial topic of GAN training. MC comes in different levels of severity and is therefore classified into partial or complete mode collapse.

1. The MNIST database of handwritten digits

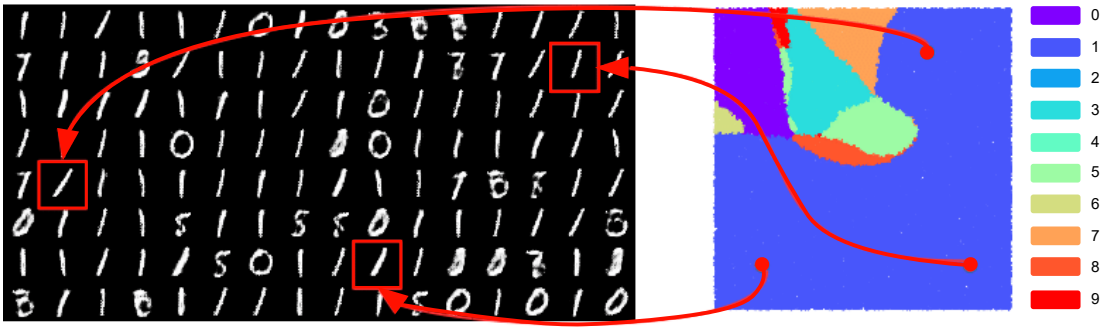


Figure 2.2: A partial mode collapse of a GAN trained on the MNIST[1]dataset, which is a hand drawing dataset of numbers. Visualized on the right is a two-dimensional projection of the input latent space $\mathcal{Z}$, which shows that a majority of the latent space vectors are mapped to the drawing of a one. The graphic is created by Tran et al. (2018).

A partial mode collapse, visualized in Figure 2.2, is the case where some diversity remains in the generator. In contrast, the complete mode collapse only produces a single data point. It can be observed that when a mode collapse is about to occure that all gradients of the generator point in similar directions. (Salimans et al. 2016) In the original GAN architecture, no mechanism enforces diversity, and there exists no coordination between the gradients of each example of the discriminator. Thus the discriminator calculates the gradients for each example independently and points the generator to the point that it believes to be the most probable. After the generator collapses and produces only a single or few distinct outputs, the gradient descent algorithm used to train the models can not separate the outputs resulting in a GAN that can not converge to a correct distribution.

The solution to the mode collapse problem has many different approaches. The first simple solution is to associate examples with each other, typically achieved through batch normalization (Ioffe and Szegedy 2015) as firstly done in the deep convolutional generative adversarial network (DCGAN) (Radford et al. 2015), which is further discussed in section 2.3.3.1. Batch normalization modifies the input for each layer to have zero mean and unit variance over the whole batch. Radford et al. (2015) describe batch normalization as critical for getting deep generators to begin to learn while preventing mode collapse. This approach is criticized by Gulrajani et al. (2017) to change the problem of the discriminator's training. Instead of learning a mapping from a single input to a single output, it is changed to a mapping from an entire batch of inputs to a batch of outputs, reducing the quality of the outputs. They recommend layer normalization (Ba et al. 2016) as a drop-in replacement for batch normalization.

Another approach is to modify the loss function to encourage the generator to be more diverse. This is done in various ways, with the most adopted one being the Wasserstein Gan discussed in 2.3.3.2. A more direct approach by Tran et al. (2018) is to introduce a latent-data distance constraint which tries to enforce compatibility between the latent sample distances and the corresponding data sample distances.

2.3.2.2 Vanishing Gradients

The vanishing gradient problem is a general deep learning problem of NN (Basodi et al. 2020). It results in the problem that the generator does not improve in producing good quality images.

The main issue is that the gradients required to train the generator become vanishingly small in the initial layers of the network. Combined with the problem that minimizing $\log(1 - D(G(z))$ results in the issue that if the discriminator is too confident in its prediction, the probability becomes $D(G(\mathcal{Z})) \approx 0$ and the gradient diverges to zero. Goodfellow et al. (2014) proposed to maximize $D(G(\mathcal{Z}))$ instead, which provides stronger gradients in the early stages of the learning process but introduces a larger variance of gradience, making the training less stable.

Another common practice, even in the original training algorithm, is training the discriminator more than the generator. The goal is to keep the discriminator close to its optimum for the intermediate generator state. Arjovsky and Bottou (2017) argue that the gradients become more reliable with a more trained discriminator. This becomes even more important in architectures that use the Wasserstein distance due to its stronger gradients, which will be discussed in section 2.3.3.2.

2.3.3 Development of GANs

The described difficulties in training GANs and their promising capabilities in generating a variety of content led to a variety of research into improving the stability in training, measurability and overall results.

Saxena and Cao (2020) categorize the different approaches for improving a GAN in three distinct ways.

1. Re-engineering the overall network architectures.
2. Proposing a new objective function for the generator and or the discriminator.

3. Developing new optimization algorithms for the generator and or discriminator.

Plenty of other optimizations which are non-GAN specific, such as an model ensamble of multiple generators (Tao et al. 2018) or training multiple discriminators (Y. Wang et al. 2016) have been investigated but are not subject of this thesis.

2.3.3.1 DCGAN

The aforementioned deep convolutional generative adversarial network (DCGAN) falls into the first category of reengineering the architecture of the generator and discriminator. It is the first Convolutional Neural Network (CNN) based GAN architecture that employs a continuous training process and has been shown to perform well in image generation tasks. (Jabbar et al. 2020)

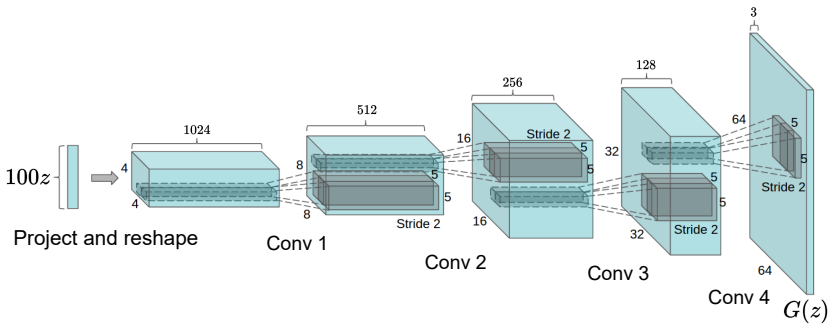


Figure 2.3: The architecture of a DCGAN generator that creates a 64x64 image. Visualization of the network is created by Radford et al. (2015)

Figure 2.3 visualizes the architecture of the generator developed by Radford et al. (2015). Compared to the original GAN architecture that only used fully connected layers and pooling layers, the proposed architecture uses transposed convolutions to create the required size. No fully connected or pooling layers are used. The use of fractionally-strided convolutional layers, also called transposed convolution, improved the stability of GAN training significantly. (Jabbar et al. 2020) Similarly, the discriminator uses only convolutional layers to derive its prediction from a given image.

The generator uses the ReLU (Nair and Hinton 2010) activation function between the layers and a Tanh function at the output layer, similarly to the original implementation by Goodfellow et al. (2014). In contrast, the discriminator does not use the Maxout function but a leaky ReLu (Xu et al. 2015).

2.3.3.2 Wasserstein GAN

The Wasserstein GAN (WGAN) developed by Arjovsky et al. (2017) is an improvement of the training of GANs that falls into the second category of proposing a new objective function. The objective function they adapt for training the Wasserstein GAN is the Earthmover Distance (EM) [2] or Wasserstein-1 distance.

The original discriminator objective function determines whether an example is drawn from the fake or real distribution and designates a value between 0 and 1, respectively. As Goodfellow et al. (2014) pointed out, this leads to vanishing gradients if the two distributions do not overlap, resulting in slow generator training. The EM distance is the distance between two probability distributions over a region D. It is intuitively described by visualizing both distributions as different ways to pile an amount of earth with the task of transforming one pile into the other. The EM describes the minimal cost of transformation, where the cost is the amount of earth times the distance by which it moved.

Compare to Wikipedia to discuss if cool

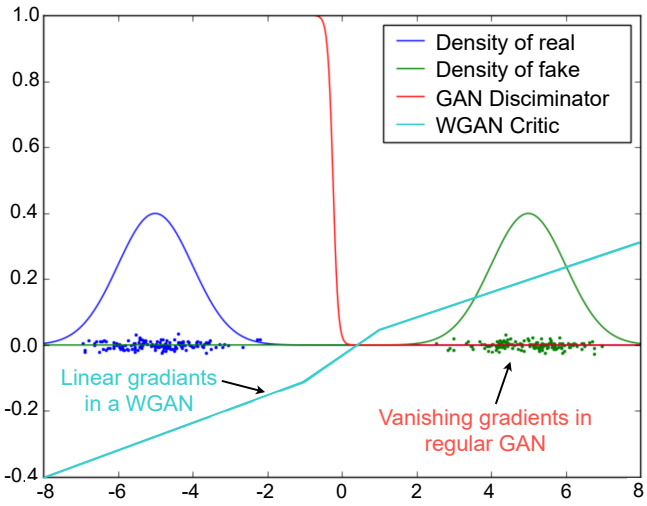


Figure 2.4: A artificial example in which two optimal trained discriminators/critics, trained to differentiate two Gaussians, calculate gradients.

Figure 2.4 gives insight into how the gradients behave for a WGAN-Critic using the Wasserstein-Distance / earth mover distance and the original GAN discriminator. The GAN discriminator's gradients saturate and result in vanishing gradients, while the WGAN critic provides clean gradients on all parts of the space (Arjovsky et al. 2017).

Incorporating the Wasserstein Distance into the gan training results in the gradients

2. Formal definition: The Earth Mover's Distance

	Discriminator/Critic	Generator
GAN	$\nabla_{\Theta_d} \frac{1}{m} \sum_{i=1}^{m} [log(D(x^{(i)})) + log(1 - D(G(z^{(i)})))]$	$\nabla_{\Theta_g} \frac{1}{m} \sum_{i=1}^{m} log(D(G(z^{(i)})))$
WGAN	$\nabla_w \frac{1}{m} \sum_{i=1}^{m} [f(x^{(i)})) - f(G(z^{(i)}))]$	$\nabla_\Theta \frac{1}{m} \sum_{i=1}^{m} f(G(z^{(i)})))$

Table 2.1: Comparing the gradients for discriminator/critic and generator for the original GAN and Wasserstein GAN (Hui 2018).

shown in table 2.1. When applying the Wasserstein Distance, the WGAN critic f becomes a function that has to be a 1-Lipschitz function, which is a strong form of uniform continuity. In other words, a Lipschitz continuous function is limited in how fast it can change, which results in stable gradients, that point towards the right direction, even far away from the actual distribution.

The constraint is enforced by clipping the weights of the critic f by an extra hyperparameter c. The WGAN algorithm describes the weight update of the critic by using the Root Mean Squared Propagation (RMSP) (Hinton et al., n.d.) update and weight clipping:

$$w \leftarrow w + \alpha \cdot RMSProp(w, g_w)$$
$$w \leftarrow clip(w, -c, c)$$

Arjovsky et al. (2017) show multiple benefits over the original GAN training when using the Wasserstein optimisation function.

- Does not require a careful balance between the discriminator and generator.
- Less restricted in the architecture design of the NN.
- Mode collapse becomes less likely because of more stable gradients.
- The EM Distance correlates well with the sample quality.

While noting the progress that WGAN makes toward stable training of GANs, Gulrajani et al. (2017) criticise the use of weight clipping to enforce the Lipschitz constraint on the critic. They claim this could lead to low-quality samples or that the training is less likely to converge. They propose an alternative to weight clipping by penalising the norm of the critic's gradient with respect to its input. They enforce the 1-Lipschitz constraint by penalising the gradient by adding the penalty given in equation 2.2.

$$gp = \lambda \mathbb{E}_{\hat{x} \sim \mathbb{P}_{\hat{x}}} [(\|\nabla_{\hat{x}} D(\hat{x})\|_2 - 1)^2] \tag{2.2}$$

The variable $\hat{x}$ is an interpolation between a generated sample and a sample drawn from the real distribution. Given that interpolation, the norm of the gradient of the critic at the interpolated point should be 1. They prove that an optimal critic contains interpolations with a gradient of 1.

They claim that the proposed Wasserstein GAN with gradient penalty (WGAN-GP) performs better than the standard WGAN with higher quality samples and more stable training through a wider variety of architectures with less hyperparameter tuning. A downside to this approach is that calculating the interpolations for each training step is an expensive operation.

2.3.4 Applications of GAN

GANs have been applied in various task besides PCGML which is further described in section 2.4.1. As two neural networks are trained, one to differentiate real from fake data and one to generate new data, they both can be used in their respective tasks. Generally, what can be achieved with a GANs depends on what kind of data that can be encoded. Different data structures have been investigated to be used with GANs: H. Wang et al. (2017) developed the GraphGAN, used to learn graph representation and Yu et al. (2016) developed the SeqGan for token sequences.

Figure 2.5: Different GAN architectures in the task of face synthesis.
From left to right: (1) The Original GAN Paper (Goodfellow et al. 2014). (2) First use of Deep Convolution Networks (Radford et al. 2015). (3) Using a joint distribution training task (Liu and Tuzel 2016). (4) Progressive training of Generator and Discriminator (Karras et al. 2017). (5) Incorporates style transfer architecture in generator (Karras et al. 2018).

For the various applications of the GANs generator, the image synthesis task is the most well-studied one. (Huang et al. 2018) Figure 2.5 shows the progress of GAN capabilities from the year 2014 to 2018 in the task of face synthesis.

Another well-studied application of GANs is anomaly detection due to their ability to learn data unsupervised. Outlier detection models are typically trained on large amounts of annotated data (Schlegl et al. 2017), and the effort of labelling this data, which usually requires expert knowledge, limits the applicability of such approaches. The discriminator is able to detect outliers in image data as done in the AnoGan architecture developed by Schlegl et al. (2017). It is trained on healthy medical images and is able to detect imaging markers relevant to disease progression. Xia et al. (2020) used GANs in their LogGAN model to detect anomalies in sequential log data.

2.4 Videogame level Generation via machine learning

With the basis of machine learning and GANs covered in section 2.3, this section reviews the application in the context of Procedural Content Generation via machine learning (PCGML), primarily video game level generation. PCGML can be separated into the two fields of data representation and training method. A. Summerville et al. (2017) organize PCGML in their taxonomy of methods techniques into these two categories.

The underlying data representation used defines how the data is encoded in order to be used in training and the generation process. In their taxonomy, they consider three distinct data representations: (1) Sequences, (2) Grids, (3) and Graphs. One piece of information is not constrained to only one representation but can be encoded in many different ways. For example the level of a platformer has been defined in all three representations (Summerville and Mateas 2016; A. Summerville et al. 2017). The Video Game Level Corpus (VGLC) (A. J. Summerville et al. 2016) contains 428 levels from 12 games in all three representations.

The training methods reviewed in A. Summerville et al. (2017) taxonomy are Back Propagation, Evolution, Frequency Counting, Expectation Maximization, and Matrix Factorization. The main focus of this section is level generation through GANs, which primarily use NN in their underlying architecture and are mainly trained through backpropagation.

2.4.1 General Level generation with GANs

Transferring the knowledge of image synthesis into level generation is not a trivial task. Primarily finding an encoding of the level that can be decoded even if the outcome of the GAN is noisy or imperfect is difficult.

One of the first to apply GANs in the context of level generation is Giacomello et al. (2018) in their work to generate DOOM levels based on human-designed content.

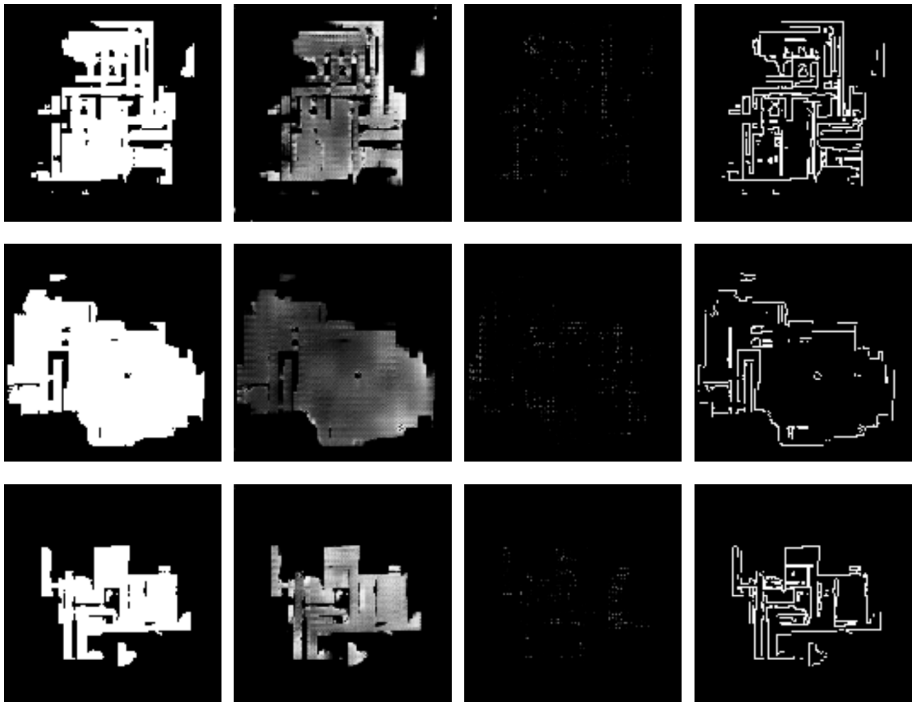


Figure 2.6: The layers required for recreating a doom level. From left to right: the FloorMap, HeightMap, ThingsMap, and WallMap of the generated levels.

They extract six images for each level representing the game level file to train the GAN. Figure 2.6 visualizes the first four layers from left to right.

- A floor map represents the space a player can walk on or not walk on.
- A height map that defines the vertical height location of the floor.
- The things map includes one-pixel representations of locations where items are placed. Different items are represented through different values.
- A wall map visualizes the locations of walls through one-pixel borders.
- The trigger map encodes the location of triggers, e.g. doors or elevators.
- Lastly, a RoomMap segments a level into different rooms.

They describe their preliminary results as a good starting point for researching the viability of GANs compared to classical PCG. The generated levels contained DOOM typical features and are supposedly interesting to explore, even though they

could not decode the generated data into levels, and therefore the playability was not tested.

Volz et al. (2018) utilized GANs to generate complete Mario levels. They use Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES) to search the latent space of the GANs generator to influence the outcome based on different metrics over the generated levels. The first approach is to optimize different block distributions. For example, fewer stone blocks could lead to an air level with greater difficulty. In their second approach, they utilize a Mario AI (Togelius et al. 2013) that can produce playthrough data of their generated levels. They focused on optimizing toward playable levels with a scalable difficulty. The idea of using latent variable evolution (LVE) to explore the generator's latent space was firstly introduced by Bontrager et al. (2017) in their works to match generated fingerprints to as many real fingerprints as possible. Evolving the latent space to gain control over the output stands in contrast to Conditional GANs (CGANs) (Mirza and Osindero 2014), which utilize a condition vector combined with the noise vector as input to the generator to produce a controllable output.

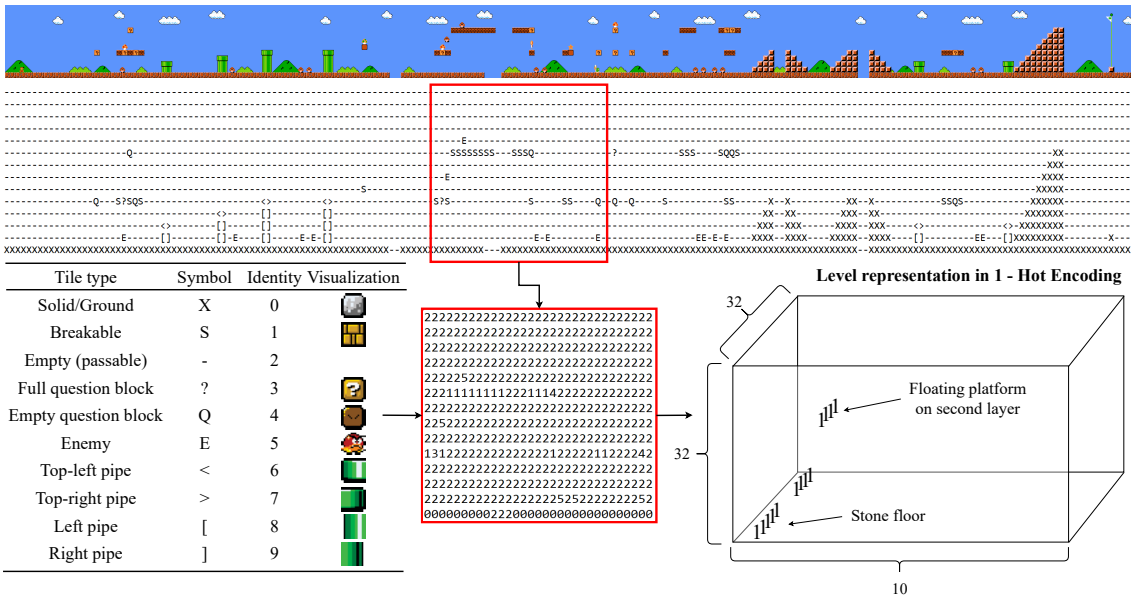


Figure 2.7: Process of transforming a Mario level into a One-Hot, multi-dimensional level representation used by MarioGAN. The $32 \times 32 \times 10$ matrix, on the bottom right, is filled with zeros except x, y coordinate on the layers of the blocks visible in the selected window.

Figure 2.7 visualizes how a Mario level is transformed into a representation that can be put into the discriminator. The ASCII representation provided by the VGLC (A. J. Summerville et al. 2016) is mapped into the identity matrix, which is put into a one-hot encoded matrix. In other words, each x, y coordinate becomes a vector

of size 10 with a one at the location of the visible block. The advantage of this number-based representation is that the generated image can be decoded into a playable level through an argmax operation over the layers of the one hot encoded matrix.

They conclude that GANs can capture high-level structures of the training level even though they sometimes produce broken elements such as incomplete pipes and structures. Their main conclusion is that LVE is a promising approach for fast level generation that can be adapted to other game genres.

2.4.2 Angry Birds Level Generation

Machine learning-based level generation using neural networks and backpropagation for Angry Birds has only recently been investigated by Tanabe et al. (2021). Conventional level generation approaches for Angry Birds are designed using domain knowledge, and search-based approaches (Tanabe et al. 2021).

2.4.2.1 Genetic Algorithm

Ferreira and Toledo (2014) used a genetic algorithm to find a stable combination of blocks and predefined structures.

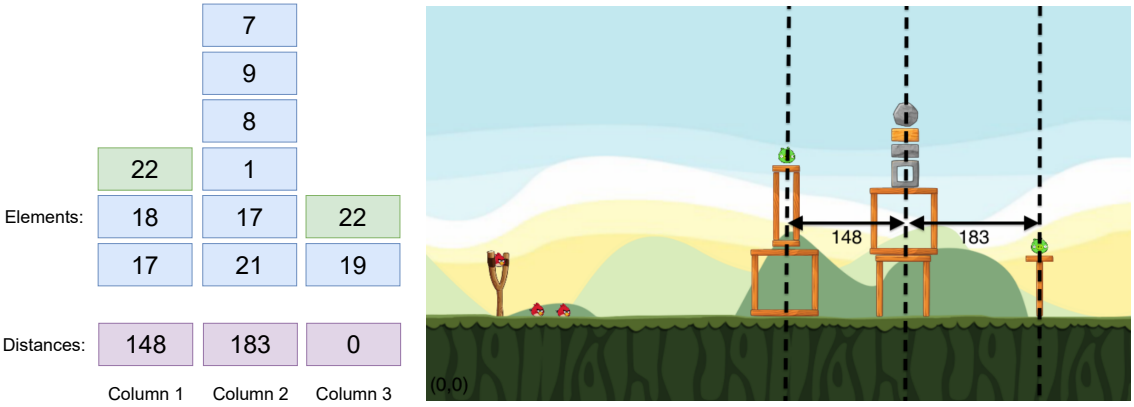


Figure 2.8: The level encoding defined by Ferreira and Toledo (2014) used blocks and predefined structures in arrays of columns. For example, the block ID 22 represents a pig.

Figure 2.8 shows how a level with three structures is represented as an individual used in the genetic algorithm. The initial population is generated using a stochastic selection that defines the likelihood of a block appearing at the bottom, middle or top. Crossover operators are used only between whole columns, and mutation operators change individual blocks.

2.4.2.2 Search-based Approach

The Winning entry for the 2017 and 2018 AIBIRDS level generation competitions[3] is the search-based approach by Stephenson and Renz (2017), which can create complex stable structures with various different elements. The proposed level generator is built upon and improves their previous iterations. (Stephenson and Renz 2016a, 2016b)

Structures generated using the original algorithm are made up of rows, each consisting of a single block type. It operates by recursively adding rows of blocks to the bottom of the previous row. After the block type of each row is selected, the previous row is divided into subsets based on the distance of the blocks in the row. For each subset, the block placement of putting the subsequent blocks in the centre, at the edge or in both locations is created and checked for validity (overlap and support are checked). Of all possible valid block placements, one is selected at random.

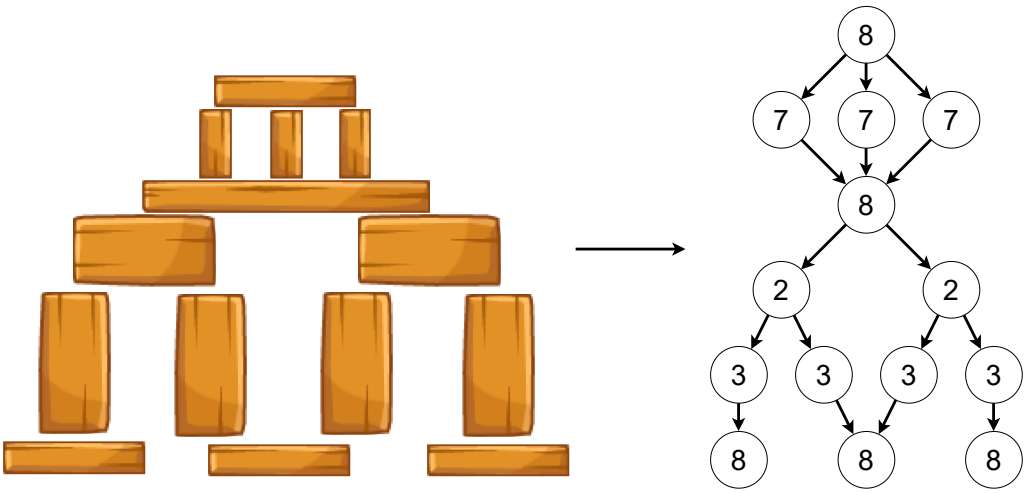


Figure 2.9: A structure generated by Stephenson and Renz (2016a) which shows the block placements and how the structure can be represented as an acyclic graph.

Figure 2.9 shows the three different kinds of block placements in different subsets incorporated into a generated structure. The pig placement is done by analysing the structure and searching for free space above the centre or corners of each block. This algorithm is extended in (Stephenson and Renz 2016b) to include irregular blocks after the pigs have been placed.

This approach's main limitation is limiting each row's block type, which reduces the variety of possible structures. The last iteration (Stephenson and Renz 2017)

3. https://aibirds.org/other-events/level-generation-competition.html

extends the generator to allow multiple block types in one row by swapping blocks with other block types of the same height.

2.4.2.3 VAE-LSTM model

The aforementioned NN-based approach by Tanabe et al. (2021). uses VAE (Kingma and Welling 2013) in combination with LSTM (Hochreiter and Schmidhuber 1997). VAEs are trained on recreating the input at the output, and LSTMs are a version of recurrent neural networks (RNNs) and are usually used in text processing tasks.

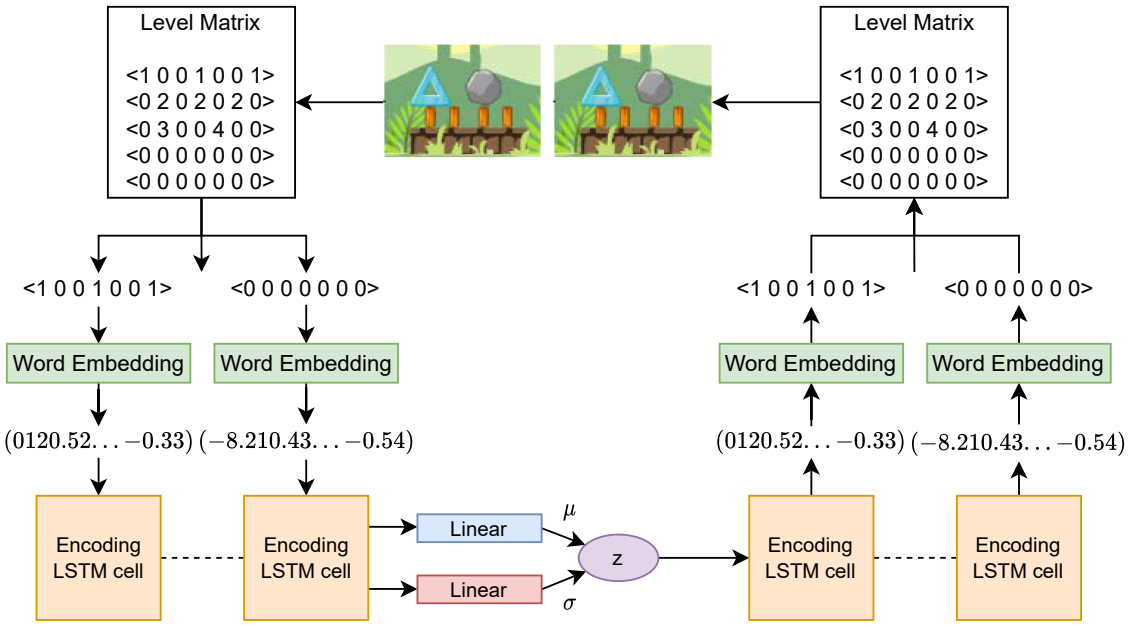


Figure 2.10: Flowchart of the proposed approach by Tanabe et al. (2021). The chart is a recreation based on their provided Flowchart.

Figure 2.10 describes how a level is encoded into a level matrix. The level matrix encodes a level as a data sequence capturing the level as if each block is dropped individually from the top onto the next block or platform below. The matrix is then processed as if it were a natural language sentence, with each row representing a word. The word embedding is put into the VAE-LSTM model with the training goal of recreating the level. Creating new levels is archived through LVE in between the autoencoders encoder and decoder.

3 Concepts

This chapter addresses the concepts used in the different approaches that form the structure generation with GANs. An overview of the concepts, and the process flow of encoding and decoding with the individual design decisions, is given in Figure 3.1.

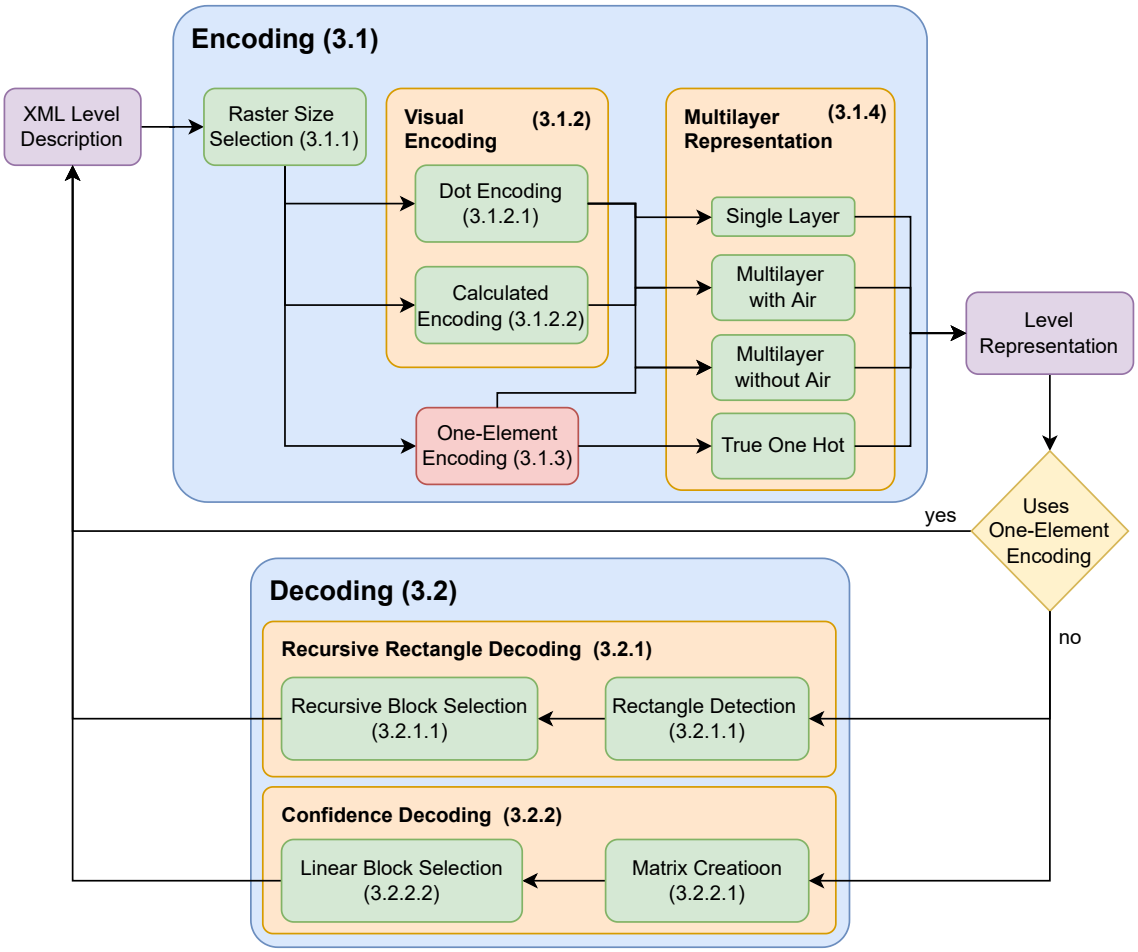


Figure 3.1: Flow chart of the encoding decoding process with all possible design decisions.

Input to the encoding-decoding process is the XML-level description containing the structure encoded into a level representation. The level representation is required to be processed by the GAN. After the encoding raster size is decided, described

in Section 3.1.1, two different ways to represent a block are possible. The “Visual Encoding”, covered in Section 3.1.2, encodes the level as if a picture is taken of the level and is further divided into the “Dot Encoding” (3.1.2.1) and “Calculated Encoding” (3.1.2.2). On the other hand, the “One-Element Encoding” inspired by the Tanabe et al. level encoding represents each block into one single pixel, further described in Section 3.1.3. With each block represented in the encoding, there are multiple ways how they can be split up over multiple layers, further discussed in Section 3.1.4.

With the level representation, a decoding algorithm is required to recreate the XML-level description, covered in Section 3.2. The One-Element encoding does not require an elaborate decoding algorithm and can be recreated into the XML description directly. For the visual encodings, two different approaches have been implemented. The “Recursive Rectangle Decoding” Recursive Rectangle Decoding in Section 3.2.1 and the “Confidence Decoding” in Section 3.2.2.

Each path from the XML-level description to the level representation in the encoding block of Figure 3.1 is a different way a dataset can be created. The last concept Section 3.3 describes the process of how different GAN architectures can be trained with the different training methods using the respective datasets.

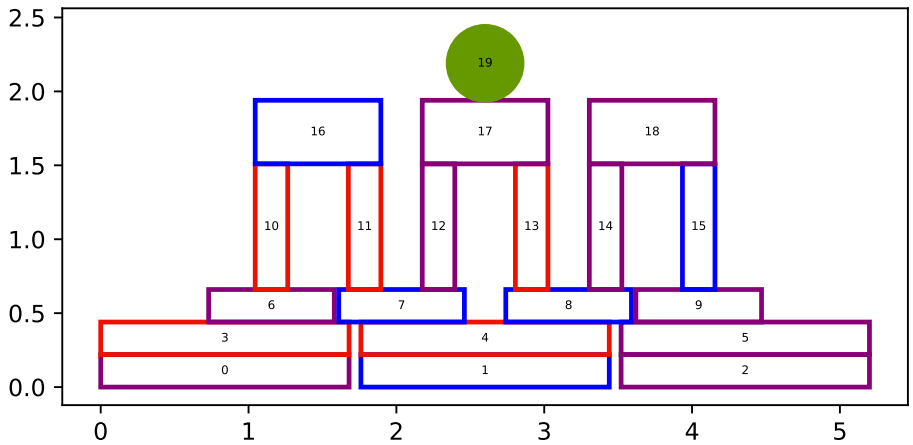


Figure 3.2: Original test structure as wireframe coloured by material.

Over the whole concepts chapter, the structure shown in Figure 3.2 is the “test structure”, used as an example for the different processes. This representation is the original, with no regard for raster size or encoding method to visualize the original structure as close as possible. The colours represent the different materials, ice, wood and stone, with the circle representing a pig.

3.1 Encodings

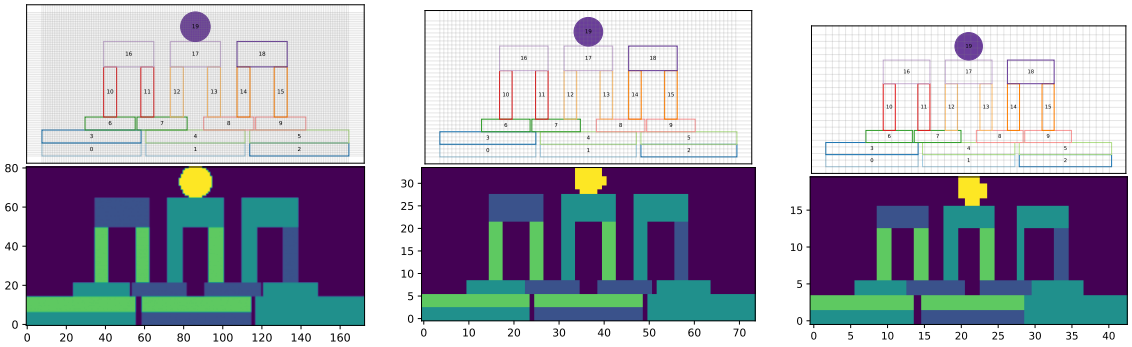
As previously described in section 2.4, a structure can be encoded in various data structures, namely as a graph, sequence or grid. Most previous attempts on level generation with GANs use image synthesis, which is also the most researched task of GANs. This thesis investigates different grid-based structure representations to expand on that knowledge. The encoding describes how the level, given as an XML file 1.1, is parsed into a structure representation.

The different design decisions, visualised in Figure 3.1, have to be made in order to create the structure representation. Both encodings, the visual and one-element encoding, require a rasterization process. The main parameter of this process is the tile size of the raster. Section 3.1.1 describes the selection process of this parameter. The following sections describe the two ways a block is represented in the structure representation. The visual encoding described in Section 3.1.2 has the decision of what tiles a block exactly covers, which introduces the problem that a block can be right on the border of a tile creating uncertainty if the tile should be coloured or not. Two different ways how this problem is solved are described in Section 3.1.2.1 and Section 3.1.2.2. While the One-Element encoding of Section 3.1.3 presents an alternative solution to how a block can be encoded. As previously explained, the representations can be split over multiple layers. The different ways to separate the blocks to create the multilayer representations are discussed in Section 3.1.4.

3.1.1 Raster Size selection

One major challenge in this domain is the real-valued positioning and dimensions of the game objects. In order to rasterize the level, the parameter of the grid dimension has to be chosen. The tradeoff in this selection is that a smaller grid size better represents the irregular blocks and their positioning but increases the dimensionality of the final img.

Figure A.6 visualizes how different grid sizes affect how the level is encoded. It can be seen that the smaller grid size, shown in Figure 3.6a, results in a representation with a larger dimension, but small gaps and imperfect positionings are better captured. Conversely, a bigger grid size results in a smaller encoded output representation where only a few tiles represent a block. This comes with a loss of finer details and wrongly represented block sizes. Imprecisions in the encoding, where the same block type gets encoded with different dimensions, become more significant in the bigger grid size. This can become problematic in the decoding process, where a distinct block and position have to be chosen.



(a) Resolution 0.03 (81, 173) (b) Resolution 0.07 (34, 74) (c) Resolution 0.12 (22, 43)

Figure 3.3: Level visualization with different raster sizes.

A good tradeoff is the biggest grid size that results in a whole number when dividing the block's dimension. Table A.2 lists possible grid sizes and the quotient of each distinct block dimension divided by the respective size. The grid size 0.07 results in quotients with only a small remainder and has been chosen for all remaining visualizations and tests.

3.1.2 Visual Encoding

As mentioned above, the "Visual Encoding" encodes the structure as if a picture was taken of the visualized structure. Each science bird block's dimension is directly encoded into the width and height of the tiles in the structure representation. Two different ways to determine what tiles a block should cover have been implemented. The "Dot Encoding" encodes the structure as a whole through a grid of dots, while the "Calculated Encoding" encodes each block individually in the structure representation.

3.1.2.1 Dot Encoding

The "Dot Encoding" uses a grid of dots to determine the value of each tile. Each dot of the grid represents a tile in the final structure representation. The dimension of the grid is determined by the outermost blocks and the aforementioned raster size.

The tile's value is determined by checking if the dot intersects with any block. The intersected blocks' material determines the tile's value, with zero being air and one to three being wood, stone and ice, respectively. The original test level with the grid of dots is visualized in Figure 3.4.

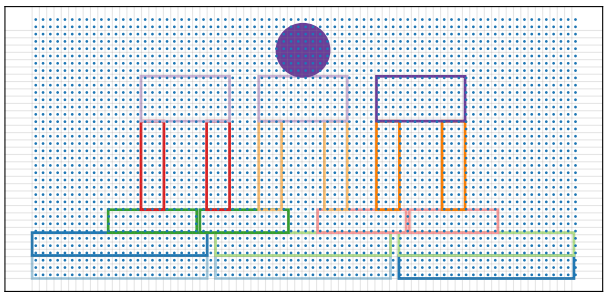
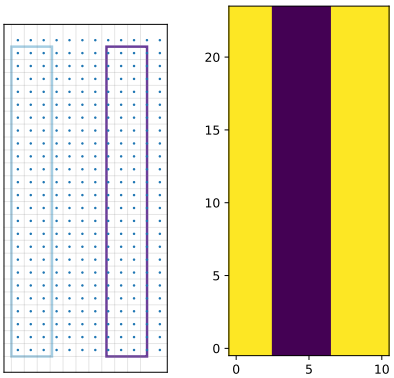


Figure 3.4: Encode the test structure using a grid of dots to determine the value of each tile.



(a) Original (b) Encoded

Figure 3.5: The same block encoded in two different dimensions.

The problem is that an individual block can be placed on the grid in such a way that it covers different amounts of dots. Figure 3.5a shows the block outline of the same block covered with a grid of dots. The right block is positioned on the grid to cover four columns of dots instead of three. The "structure" encoded in Figure 3.5b shows that the right block is wider than the left one, even so, they are the same block.

3.1.2.2 Calculated Encoding

The "Calculated Encoding" approach does not create a grid over the whole structure but sets each block independently into the image. For each block, the horizontal and vertical start and end position gets calculated. Ths positions are calculated by taking the given centre position of the block and adding/subtracting the respective half of the width/height. For both directions, the values in between are interpolated using real-valued positions. To transform the positions into the grid dimension indices, each value is divided by the grid size and rounded to the next integer.

Doing only this comes with the same problem that the "Dot Encoding" has. The possibility that a block is positioned on the grid in such a way that it would be encoded too short. To solve this problem a size check is introduced. It compares the encoded dimension of each block with an apriori calculated block size so that every same science bird block is encoded with the same dimensions.

3.1.2.3 Block Encoding Comparision

The main difference between the two visual encoding approaches is one encodes true to the structure while the other encodes true to the structures blocks. True to the structure means that an individual's block's dimensions aren't preserved in the encoding and are discarded over the structural dimension.

(a) Calculated Encoding (b) Dot Encoding (c) Subtraction of both encodings

Figure 3.6: Comparing the two structure encodings and visualising their difference.

Figure 3.6 shows the original test structure encoded once using the "Dot Encoding" and the "Calculated Encoding". By taking the difference of the encoded structure representations it can be seen that the dot encoding encodes multiple blocks in both orientations differently.

The fact that the block encodings are not consistent over a structure representation when using the dot encoding becomes a problem in the decoding process. Another downside to the dot method is that the time to generate the image encoding is comparatively high as it takes more time to calculate the intersection for each dot.

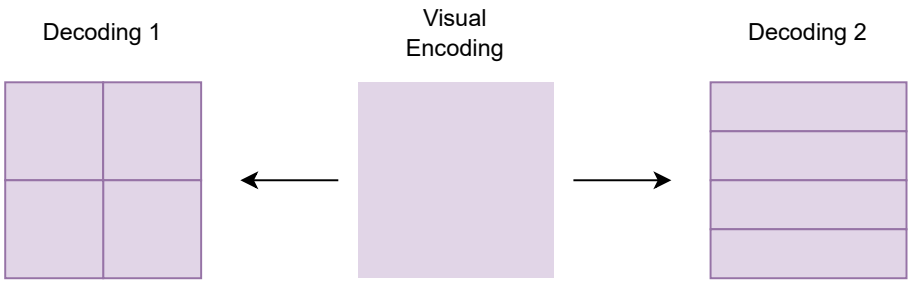


Figure 3.7: Visualizing the ambiguity of the visual encoding.

One overall issue with the "Visual Encoding" is that it is ambiguous in its encoding. An example of an ambiguous encoding is given in Figure 3.7. The centre image represents a chunk of encoded pixels and can be made up of different block configurations. Either to the left as four smaller square blocks or to the right as a stack of four squares.

3.1.3 One-Element Encoding

To address this ambiguous encoding problem, the one-element encoding represents the dimension of each individual block not through the number of til but through the value of the pixel in the centre of the block which is inspired by Tanabe et al.'s level encoding, which encodes the structure as a data sequence explained in Section 2.4.2.3. Figure 3.8 shows how the testing structure is represented in the one-element encoding. Each individual block type with its material becomes only one pixel that is located at the centre of the block.

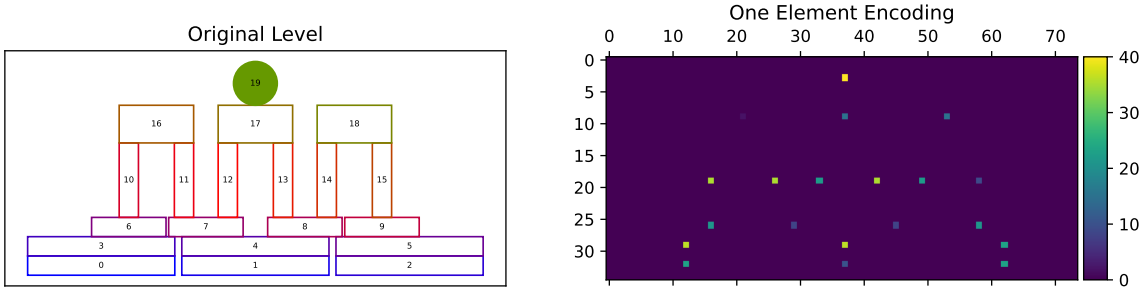


Figure 3.8: The test level in one element encoding.

The problem is that the rotation of a block is not directly represented in the tile's value. To solve this problem, each 90-degree rotation of the rectangular blocks becomes its own block value. This limits the rotation of each block to 90 degrees increments. This results in 13 individual block values from the original nine different science bird blocks listed in Figure 1.2. Each block can be either of wood, stone or ice and therefore results in the value range of $[1, \ldots, 39]$ while a pig gets encoded as a 40 and the air is still zero.

As indicated in the overview Figure 3.1 the "One-Element encoding" does not require a decoding algorithm as every pixel has a defined meaning so that every structure representation can be decoded into a collection of blocks directly. This covers only the decodability and not the structural integrity of the decoded structure. The problem is that every coloured pixel could come too close to one another, and the represented blocks would intersect if decoded directly. The issues of overlapping blocks get highlighted in the results chapter. While this problem doesn't exist in the "visual encoding", which on the other hand has the problem that a set of pixels can not resemble any type of block.

3.1.4 Multilayer Representation

GANs data representations are usually not only two-dimensional. For example, face synthesis requires three channels for the RGB-Color space, and Volz et al. (2018)

used ten channels in a one-hot encoding, one for each Mario block. Therefore there are multiple distinct ways a level representation can be split up into different layers.

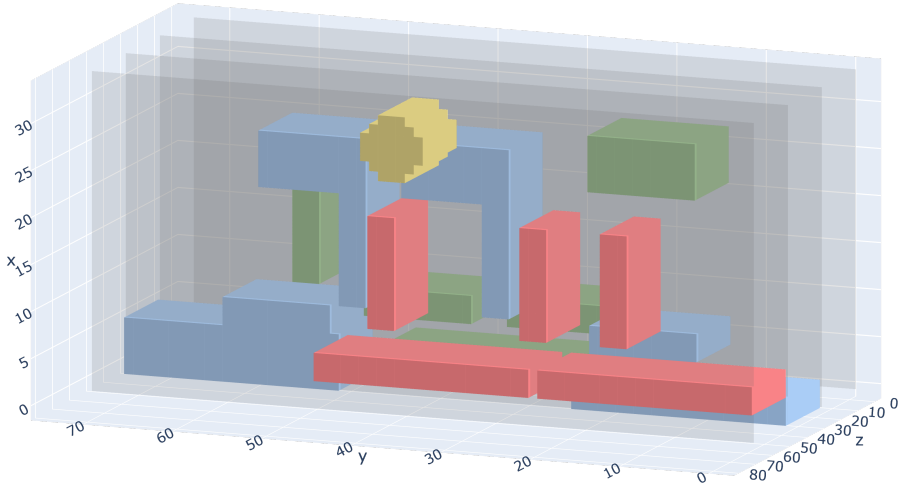


Figure 3.9: Expanding the test level to encode over multiple layers. The z-axis represents the layers and is scaled for visualization purposes.

Figure 3.9 shows the "Multilayer without Air" encoding, which splits the representation up in such a way that blocks of one material are on the same layer. A reason to do this is that having multiple semantic meanings in one pixel makes the training more difficult. Letting the GAN simultaneously decide about the positioning of the elements and their correct material in only one layer is difficult. By moving the materials to different layers, the decision is reduced to if an element is present at a specific position. Another problem of using a different value range removes the possibility of interpreting the value as confidence in the pixel. A higher value does not mean more confidence in this position but different material. The distance to the closest integer could be used instead. A value right in between two values has less confidence than a value which is spot on.

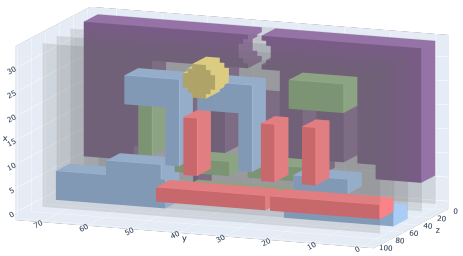
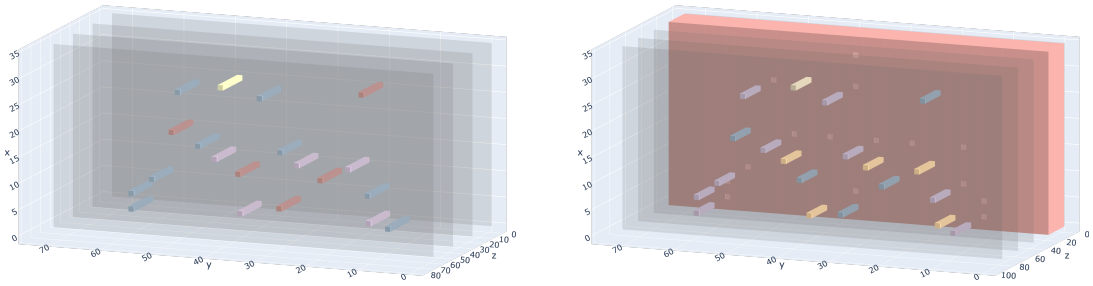


Figure 3.10: Explicitly encoding air in the level.

When using this multilayered encoding, the GANs output is a real-valued matrix in which a value closer to one means there should be a block. To recreate the visual representation given the multilayered output, a flattening process is required. With the encoding of Figure 3.9 a real-valued threshold is required. This leaves the final decision of where the air should be to this threshold. Adding a layer in which the air is explicitly encoded, shown in Figure 3.10,

moves all decisions to the GAN. A flat image is recreated by using the argmax operator.



(a) Without air layer (b) With air layer

Figure 3.11: One Element encoding in multiple layers.

The same process of creating a multi-layer representation can also be applied to the one-element encoding. Once without air (Figure 3.11a) and once with air (Figure 3.11b) in which little hols can bee seen in the last layer. In the representation without air, the value range for an element goes from the previous $[0, \ldots, 40]$, which represents every block type in every material, to $[0, \ldots, 13]$ for each block type of one material. The pig layer is still one hot encoded.

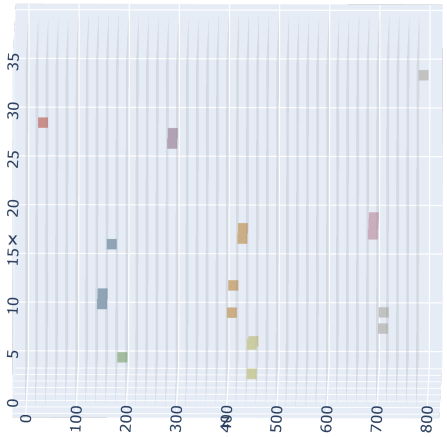


Figure 3.12: Expanding the test level to encode over multiple layers.

The last tried method to split the encoding over multiple layers is to move every possible block type and material combination to an individual layer, creating a full one-hot encoding. Figure 3.12 shows the side view of the test level in the one-hot encoding. Out of the 39 distinct blocks and the enemy position, 40 layers are created. The x, y coordinate of the pixel represents the position equal to the one-element encoding.

3.2 Decoding

One major component of generating levels with GANs is a decoding process. It describes how the encoded level is interpreted and used to recreate a level. Usually, PCGML research uses an encoding that allows one-to-one decoding, which means that no extra steps are required to create the level. As previously mentioned, this can lead to broken elements, such as a broken pipe that consists of multiple tiles (Volz et al. 2018), which was solved by encoding the whole group of tiles like the one-element encoding. The decoding process can also include a healing section as post-processing that fixes errors and tests for playability and repairs the level accordingly.

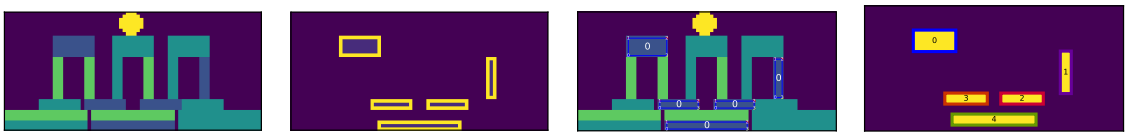
Of the two introduced encoding methods, only the visual encoding requires a more complicated decoding algorithm because one block is encoded into multiple tiles. The one-element encoding can be decoded one-to-one as each pixel represents a block, but the block elements can overlap without any post-processing.

Two different algorithms have been developed to decode the visual representation. One approach, the "Recursive Rectangle Decoding" (RRD) described in Section 3.2.1, decodes perfectly based on the silhouette of the structure representation, and the other, the "Confidence Decoding" (CD) described in Section 3.2.2, decodes through the confidence and probability of the gan output. The RRD assumes a perfect structure representation and is prone to fail if a block is missing a corner or if the representation is noisy. While the CD greedily selects the block that covers the most contour area and fits the best regardless of the combination of blocks.

3.2.1 Recursive Rectangle Decoding (RRD)

The RRD is separated into the "Rectangle Detection" part and the "Recursive Rectangle Selection" part. The first step (rectangle detection) describes the process of how the level representation is preprocessed to detect possible rectangles in the contour of the structure representation. While the "Recursive Rectangle Selection" recursively chooses these rectangles to fit science bird blocks in order to cover the respective siluet. To aid the explanation for this decoding method, all steps in the decoding process are exemplarily done on the testing structure.

To initially separate the block combination and speed up the recursive selection algorithm, further discussed in Section 3.2.1.1, all processing steps are done on each material independently. Figure 3.13 shows the whole decoding process on the wood



(a) Original (b) Border following (c) Found rectangles (d) Fitted blocks

Figure 3.13: The images show the trivial case of the whole decoding process of the first wood layer visualized in purple.

layer. In this material, all blocks are separated and do not form a combined structure making the decoding case straightforward.

The first step of the decoding process, visualized in Figure 3.13b, starts with removing all pixels not belonging to the current layer, followed by finding the contours of the blocks with OpenCV (Bradski 2000) using a border following algorithm. (Suzuki and be 1985) Each contour is represented as a list of corners that can be of any length bigger than three. With a list of corners, all rectangles that are inside the contours are put into a list as shown in Figure 3.13c. As not all rectangles that can be found in a contour have the dimensions of a science bird block, a filtering step removes all impossible rectangles. In the last step, rectangles are recursively selected to fill the contour area, further clarified in Section 3.2.1.2.

3.2.1.1 Rectangle Detection

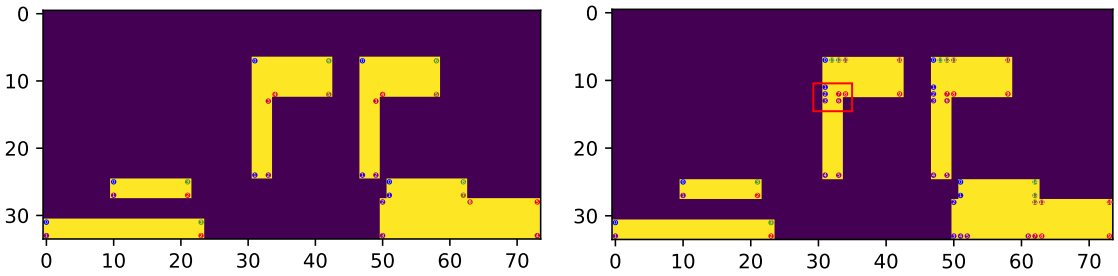
The rectangle detection is done for each individual contour. A contour with only four corners is a trivial rectangle, and the step is omitted. The rectangle detection algorithm is split into two sections: (1) the first section, given in Algorithm 3.1, searches for extra possible rectangle corners in the contour border and (2) the second section, given in Algorithm 3.2, selects four dots and checks if they form a rectangle inside of the contour.

Algorithm 3.1: Dot extending Algorithm

```
1  input: list of corner dots
2  output: extended list of corners
3  begin
4      // Inner corners
5      for A, B ← consecutiveBorder(i)
6          if A, B do not share a corner and A, B are perpendicular
7              intersection p1 ← getIntersection(A, B)
8              addCorner(p1)
9
10     dotDistance c ← closest vertical or horizontal distance between each dot
11
12     // Search for intersections between non−consecutive line pairs
13     for A ← line on contour starting at index i
14         for B ← line on contour starting at index i + 2
```

```
15
16              // Check intersection and check for integrity
17              intersection p1 ← getIntersection(A, B)
18              if p1 exists
19                    or p1 is on corners of A, B
20                    or p1 is not on A, B
21                    or p1 exists on contour list
22                  continue
23
24              // Add intersection and pairs to the contour line that it is on
25              C ← select A or B on which p1 is on
26              p2, p3 ← points along C next to p1 with distance c
27              addCorner([p1, p2, p3])
28    end
```

The input to the algorithm is the corners found by the border following algorithm (Suzuki and be 1985), which detects inside and outside corners of the contour, as shown in Figure 3.14a. The main part, lines 12 to 22, uses each consecutive pair of corners to form a line, which is used to search for intersections between those lines if they would extend infinitely in both directions. A requirement for an intersection is that it needs to fall on a line. For each found intersection, two dots are added along the line to serve as possible rectangle corners. The first section, in lines 4 to 9, adds a dot for each inside corner as these would not fall on a line but are required to detect a rectangle with only inside corners. An example of the need for the inside corners is given in Appendix A.3.



(a) Original Dots (b) Added Dots

Figure 3.14: Steps in finding possible rectangles.

The input (Figure 3.14a) and result (Figure 3.14b) of Algorithm 3.1 is visualized in Figure 3.14. The first image shows the corners given by the border following algorithm (Suzuki and be 1985), and the second image shows the contours with dots that form possible rectangles. From the intersection marked with a red rectangle, two dots that were added in this location are required to reconstruct the original test level.

Algorithm 3.2: Rectangle Selection Algorithm given a list of dots.

```
1   input: List of dots dots on the contour
```

```
output: List of rectangles
begin
    rectangleList ← new list
    // rectangle in order along the contour
    for p1, p2, p3, p4 ← combination of four (dots)
        if ||√|p1 − p3| − |√|p2 − p4|| > ϵ // unequal diagonals
              or |(p1 − p2) × (p2 − p3)| > ϵ // Unorthogonal corners
              or |(p3 − p4) × (p4 − p1)| > ϵ
            continue outer loop

        // Rectangle inside contour
        for center_point ← [(p1 + p2)/2, (p2 + p3)/2, (p3 + p4)/2, (p1 + p4)/2]
            if center_point is not inside Contour
                continue outer loop

        add Rectangle([p1, p2, p3, p4]) to rectangleList
    return rectangleList
end
```

With all necessary corners given, the Algorithm 3.2 selects each 4-corner combination and checks if it spans a rectangle inside the contour. A rectangle is defined by checking if the diagonals are equal in length and the opposite corners are perpendicular. To check that the rectangle is not outside the contour, each edge of the new rectangle is checked if the centre point is inside of the contour.

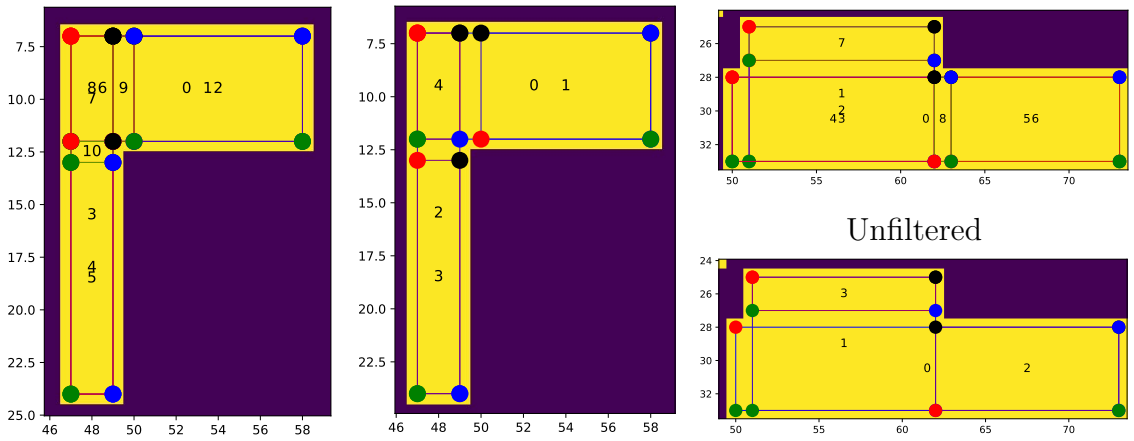


Figure 3.15: Selected rectangles through the Algorithm 3.2 and their filtered version for eligibility.

The original results of the rectangle selection can be seen in the unfiltered images of Figure 3.15. Multiple of these rectangles are not in the shape of any block or block combination to be possible candidates. A filtering step is added to remove all rectangles that are either too small or whose dimensions are not possible, given all possible stacked block combinations. To do this step, the width and height of all possible block combinations limited by four stacked blocks are calculated and put into a list.

The filtering process simply checks if the dimension of the rectangle appears in both lists. The result of this filtering process is given in Figure 3.15.

3.2.1.2 Recursive Block Selection

This section describes the algorithm which recursively selects the block-shaped rectangles given by the Algorithm 3.2.

Algorithm 3.3: Block selection algorithm.

```
1  input: list of rectangles, required area, occupied area ← 0, selected blocks ← []
2  output: List assigned blocks
3  begin
4      // stop condition
5      if occupied area = required area
6          return selected blocks
7
8      // Filter rectangles
9      foreach possible rectangle
10         foreach allready selected block:
11             if possible rectangle overlapps selected block
12                 or possible rectangle to close to selected block
13                 remove rectangle
14
15     // Main part
16     foreach possible rectangle
17         if block exists that fits the rectangle perfectly
18             selected block ← block that fits in rectangle perfectly
19
20             return recursive call (
21                 list of rectangles ← remaining rectangles,
22                 required area ← required area,
23                 occupied area ← occupied area + area of selected block,
24                 selected blocks ← selected blocks + selected block,
25             )
26
27     // To big rectangles
28     foreach possible rectangle
29         for stack direction ← [vertical, horizontally]
30             for n ← n blocks in combination from 2 to 5
31
32                 block combination ← combination of n blocks with same seconday dimension
33
34                 if block combination do not fit in stack direction perfectly
35                     continue
36
37                 if block combination are to small in secondary direction
38                     add remaining space as rectangle to list of rectangles
39
40                 return recursive call (
41                     list of rectangles ← remaining rectangles,
42                     required area ← required area,
43                     occupied area ← occupied area + area of block combination,
44                     selected blocks ← selected blocks + block combination,
45                 )
46
```

```
47      return No combination found
48  end
```

The Algorithm 3.3 selects blocks until the contour area is covered. The first step of the recursive algorithm is to remove all rectangles from the list of possible rectangles that interfere with the previous selection. In the first iteration, no rectangles are removed as there are no previous block selections. This includes any rectangles that overlap the selected block or share an edge with it. The actual block selection is split into two parts: (1 - Line 16 to 25) Of all valid rectangles with the same dimensions as an existing block, the biggest is selected, and the algorithm is recursively called with the remaining rectangles. (2 - Line 28 to 45) If no block is big enough, then the rectangle is made out of a combination of blocks. The possible blocks can be stacked either vertically or horizontally. Firstly we check if a block is stacked vertically because any horizontal separation results in worse stability. The second iteration goes over the number of stacked blocks. Each stacked block needs to have the same secondary dimension. Otherwise, the combination of blocks wouldn't be a rectangle. If the combination of blocks fits perfectly, the big rectangle gets removed from the list of possible rectangles, and the block selection gets called recursively. If the stacked blocks do not fill the secondary direction completely, a new rectangle is created in the remaining space, which needs to be filled in a deeper iteration. An example in which this case is required is visualized in Figure A.4. The new rectangle is also checked for validity. Otherwise, this rectangle would create many unnecessary recursive calls.

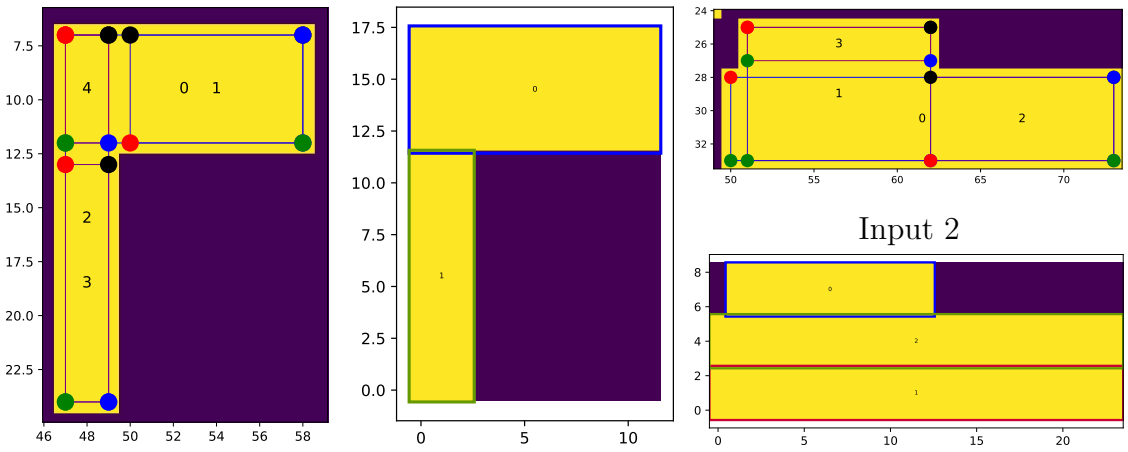


Figure 3.16: The result of the more complex shapes in the test level.

Figure 3.16 shows what rectangles are selected in the two more complex shapes of the previous example. The input 1 can be directly decoded without any block-stacking,

while input 2 needs two stacked blocks. More examples of the decoding process are attached.

The recursive block selection can be visualized as a tree, and the algorithm represents the depth-first search. Each wrong selection that can not produce a solution adds a lot of iteration to the search. Every early termination is required to make the algorithm more performant. An implemented speed-up is to check each iteration if the remaining rectangles could possibly fill the area, which stops unsuccessful paths earlier. Another speed-up can be achieved by limiting the blocks which are used in the search for a fitting block combination. Only blocks that actually fit into both dimensions of the rectangle are considered in the combination. An example with a bit more selection depth is given in Figure A.5.

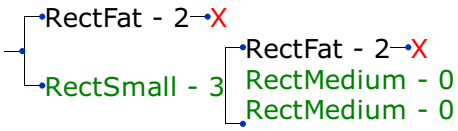


Figure 3.17: Graph of the recursive algorithm for shape in result 2.

The last step of the decoding process is to determine the pig positions. This is done using an erosion operation (Soille 2004) which is used in morphological image processing. It removes each pixel if the sum of its neighbouring pixels given by a structuring element is less than a minimum. The used structuring element is a pig-sized circle, which reduces each encoded pig to one pixel that determines the position.

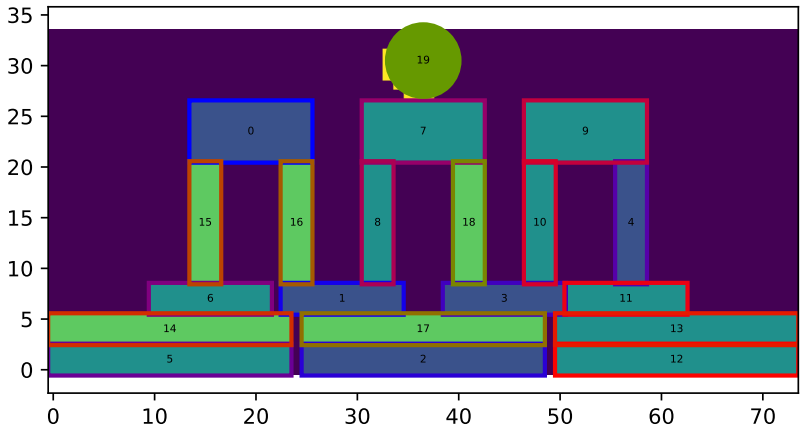


Figure 3.18: Finished decoding result.

All algorithms combined for each layer produce the result in Figure 3.18.

3.2.2 Confidence Decoding CD

As previously explained, the "confidence decoding" is compared to the "recursive rectangle decoding", the greedy linear approach for decoding a level. The main idea of the algorithm is to use convolution of block-shaped filters over the level representation. The algorithm can be done either on each material separately or on one layer combined. It is important that the value of the pixel is in the range of zero and one, representing the confidence that this pixel belongs to a block. The steps of the algorithm are visualized with the previous test level.

3.2.2.1 Matrix creation

The first step is to use convolution to create two matrices over the level. The first one ("Hit Probabilities Matrix") represents how well a block type fits into any specific position, and the second ("Size Ranking Matrix") represents a ranking of how many pixels would be covered if this block type is put into that position. A combination of both matrices creates a ranking over each pixel which is used to select blocks in a greedy manner.

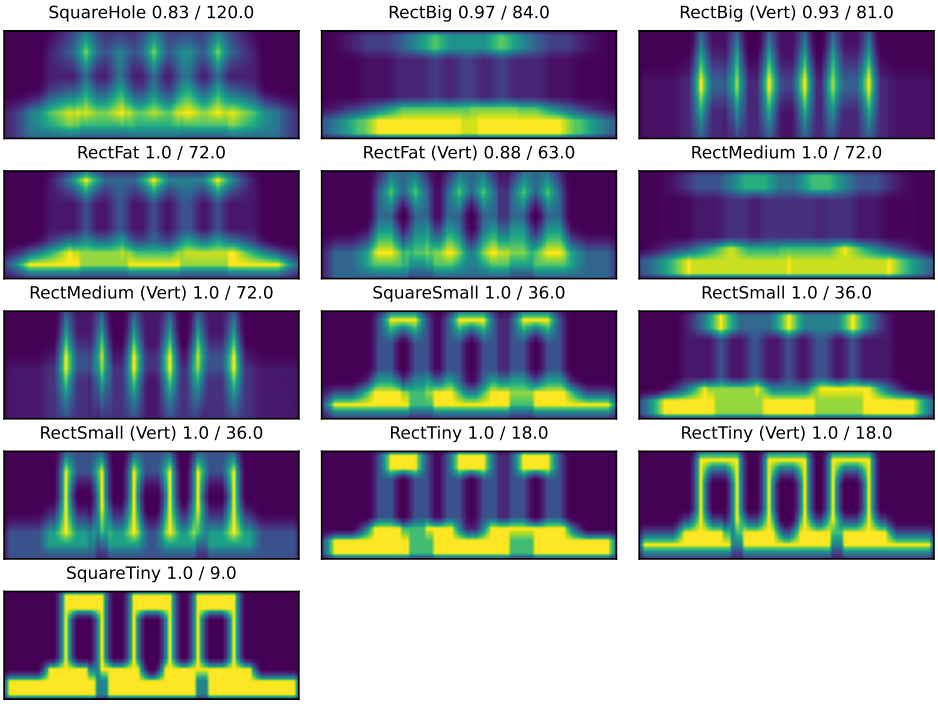
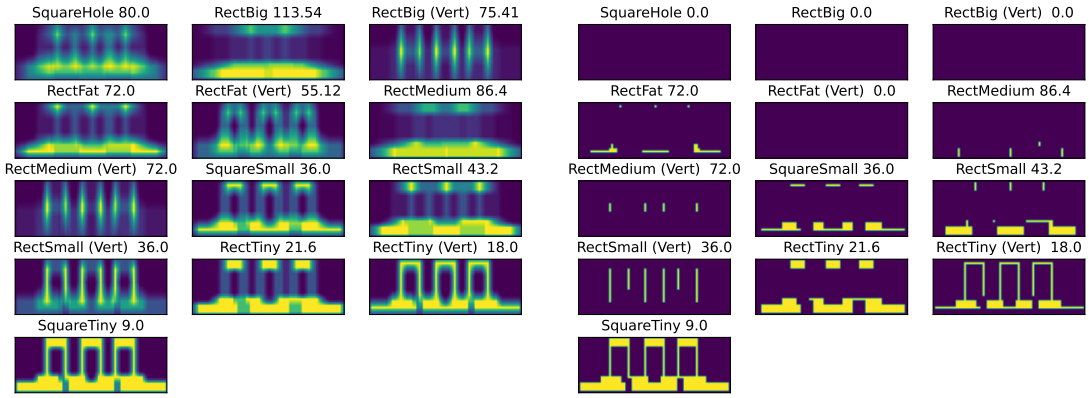


Figure 3.19: Matrix (each image represents a layer) represents the hit probability and size ranking. The first value in the tile of an image represents the highest hit probability while the second value is the highest size ranking of the respective block. "Vert" marks the block type rotated.

The “Hit Probabilities” matrix is created using multiple gaussian kernels in the shape of each block type. It can be seen that the smaller blocks in the lower rows fit perfectly, represented by a probability of one, while the bigger blocks that do not appear in the level have a lower hit probability, as they do not cover any location completely. If only the highest hit probabilities were used in the selection process, any small imperfections in the representation would dismiss any bigger block that’s intended to be there, and a group of smaller blocks that fit perfectly would be used instead. The “Size Ranking” matrix uses a sum kernel that sums the value of each pixel.



(a) Selection Ranking (b) Clipped at 0.98

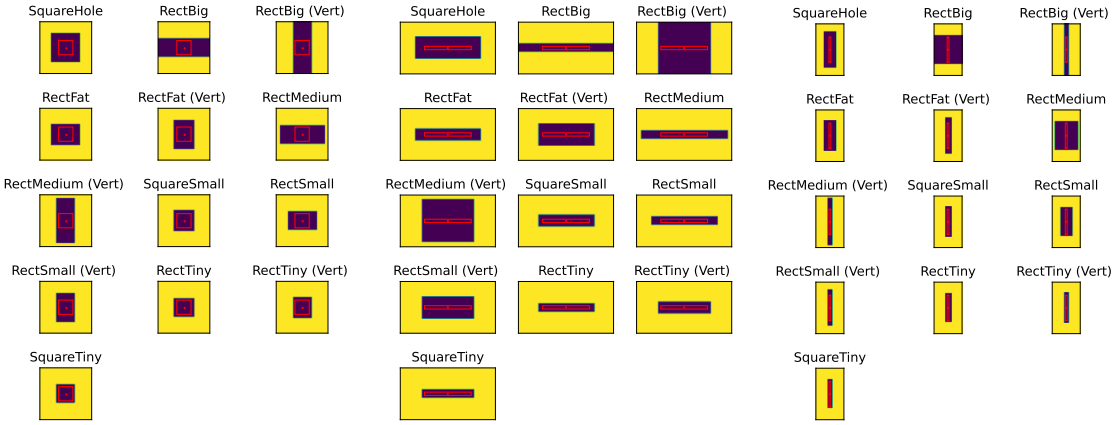
Figure 3.20: Layer-wise multiplication of the Hit Probability matrix and Size Ranking matrix creates the selection ranking. A clipping parameter p controls how well each block has to fit into the space and creates the clipping matrix with valid pixels to choose from in the block selection.

Figure 3.20 shows the combination of the two initial matrices through layer-wise multiplication. This is done to prevent the previously mentioned problems when using only one of the two matrices. It can be seen in Figure 3.20a that the horizontal RectBig still has the highest selection probability even though it crosses gaps in the encoding. By clipping at a high hit probability, which is only possible due to this example being a perfect encoding, the block types that cross gaps are eliminated. The matrix that gets passed into the block selection algorithm is the clipping matrix in Figure 3.20b.

3.2.2.2 Linear Block Selection

The block selection is a straightforward algorithm that chooses the highest value of the clipping matrix and removes all pixels that become invalid through this selection.

Invalid means any pixel that would produce a block that would overlap with the previous selection.

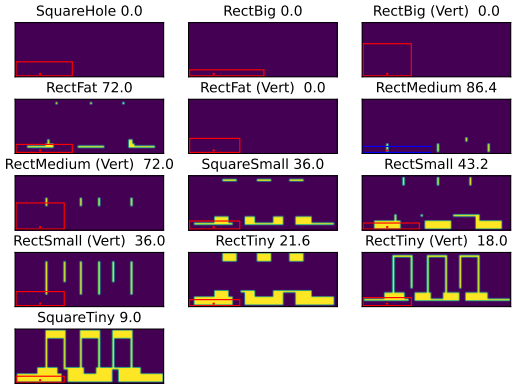


(a) Square Hole (b) Rect Big (c) Vertical Rect Big

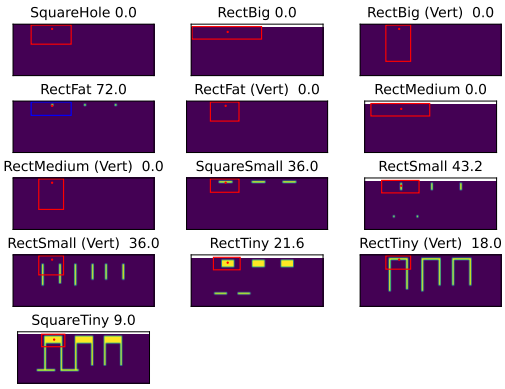
Figure 3.21: Exemplary three matrices that show the delete matrix used to remove the invalid pixels. The selected block is marked in the centre.

To do this efficiently, a matrix is created for every possible block type before running the algorithm. This matrix gets multiplied with the input matrix at the position of the selected block. The matrices are initially identity matrices, and the pixels that belong to the selected block and the pixels in which a neighbouring block would overlap are set to zero. Figure 3.21 shows the delete matrix for three block types and the region that is deleted by the matrix. It can be seen how the centre block shape and outside block shape influence the removed region. By precalculating the deletion matrices, the operation is done with a singular matrix multiplication instead of calculating the affected ranges for each block combination individually each time.

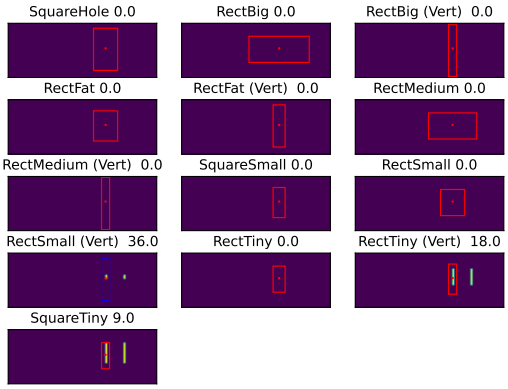
The selection process for the test level has 18 iterations in which a block is chosen. Figure 3.22 visualizes the first, the eighth and second last iteration to show the process of how the available block locations become fewer over each step. The blue rectangle marks the block location which was chosen, and the rectangle marks where pixels are removed. The block selection stops if every pixel is removed from the image. The last step of the decoding process is to determine the pig positions by using a circular kernel, followed by the same selection process until every pixel that is above an apriori threshold is selected.



(a) Iteration 1

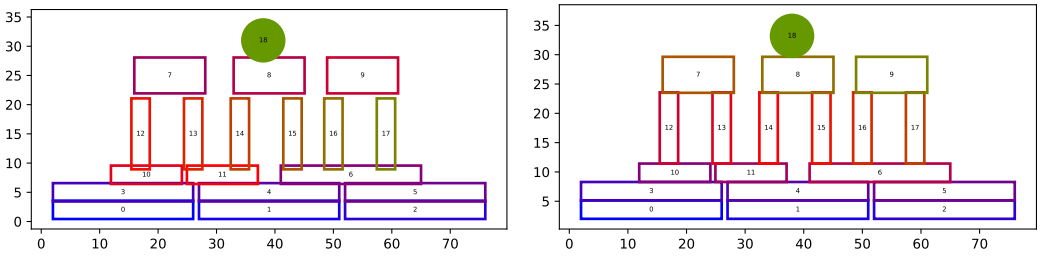


(b) Iteration 8



(c) Iteration 17

Figure 3.22: Three selected iterations in the selection process and which pixels are removed in the process.



(a) Without Recalibration (b) With recalibration

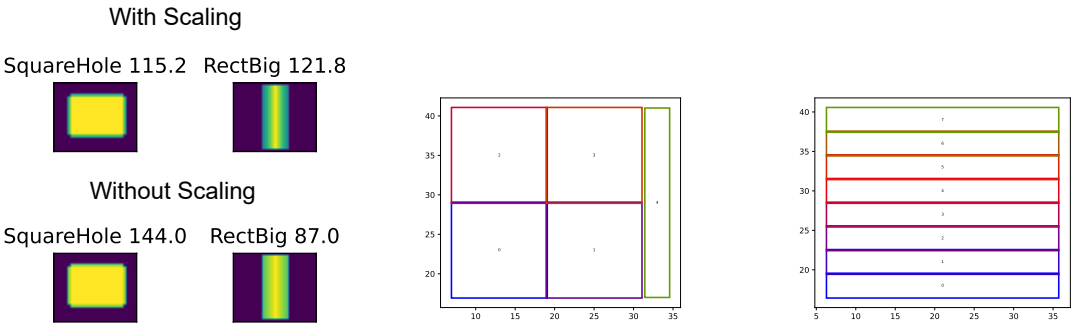
Figure 3.23: Result of the decoding process (a) without recalibrating and (b) with recalibrating.

The finished result of the decoding process can be seen in Figure 3.23a. A problem with using only the centre position as the block position is that a block with an uneven dimension is put off-centre. This can be fixed by adding a recalibration step that

moves each block up and to the right until it is out of the blocks that are further down and left. The results of this step can be seen in Figure 3.23b.

3.2.2.3 Parameter

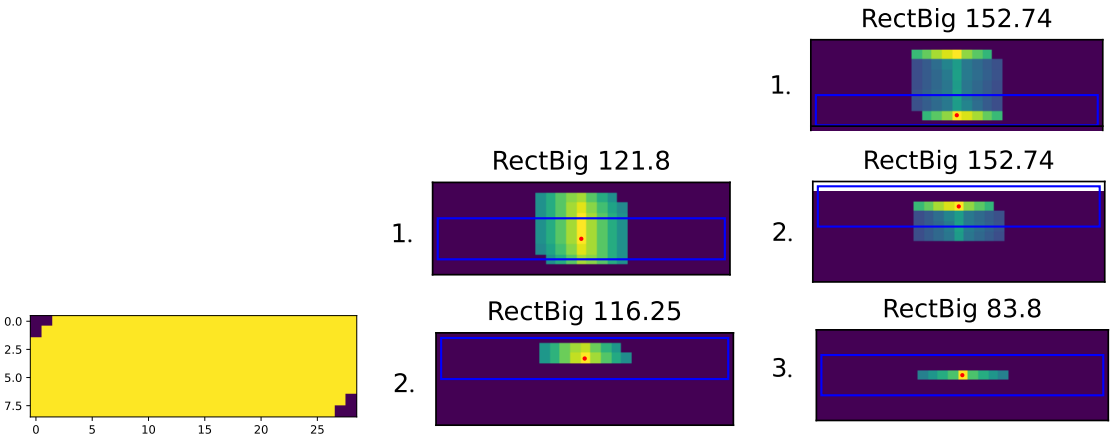
Different parameters that control the decoding process and result in different results are implemented. One way to influence which block should be used is by modifying the sum kernel with a scaler depending on the block type.



(a) Selection Matrix (b) without Scaling (c) With Scaling

Figure 3.24: The effect of using block type scaling on the sum kernel.

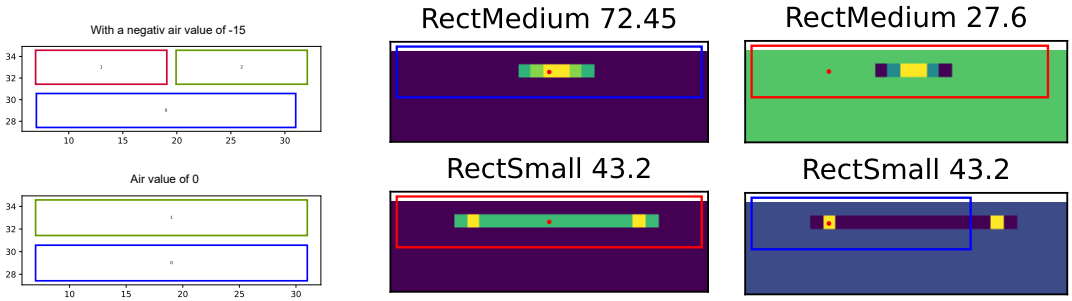
Figure 3.24 shows the effect of **block scaling** on a selected example. The "Square-Hole" block covers a lot of pixels and produces a high selection rank. By giving blocks that are more horizontal a bigger scaler, and vertical blocks and squares a smaller scaler, the selection of the blocks are changed. This can affect the stability of a level as more vertical block combinations are less stable than one bigger horizontal one.



(a) Original Contour (b) Without -1 border (c) With -1 border

Figure 3.25: The effect of using a minus one border around the sum kernel.

Another way to influence the decoding method is to encourage the selection of blocks closer to the edge of a contour with the **minus one border** parameter. Without any special method, the algorithm simply selects the block with the highest value, and if two pixels have the same value, the one that comes earlier is selected. This puts the selection at risk of floating point imprecisions and chooses, for example, a centre block that would block the better block selection. By adding a -1 border around the sum kernel, a block that is in the centre of a contour receives a smaller value. Figure 3.25 shows an example in which a contour with impurities produces only two blocks at the centre of the contour while using the minus one border selects the border blocks first and doesn't overwrite the centre pixels.



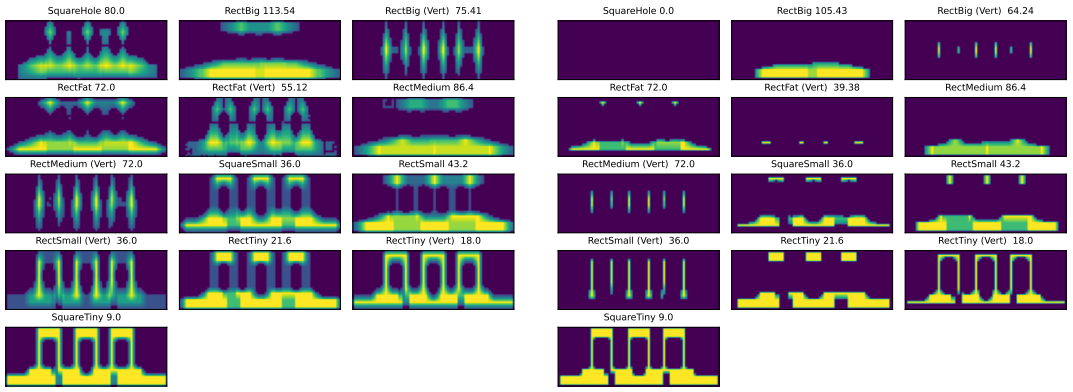
(a) Decoded result (b) Without negative air (c) With negative air

Figure 3.26: Using a negative air value to encourage better block positioning.

The **air value**, which is by default zero, can be set to any negative value. This only affects the size ranking and not the precision ranking and encourages a better positioning of blocks that do not overlap any gaps. Figure 3.26 shows the effect of this value once with a negative air value of 15 and once without. It can be seen how the selection in Figure 3.26b chooses the "rect medium" as the value is higher, while in Figure 3.26c, the individual blocks receive a higher value as they do not cross any negative value. The effect of the minus one border parameter gets amplified when using a negative value for the air.

Changing the **clipping parameter** influences how well a block has to fit into a given space. Figure 3.27 visualizes the selection matrix of the test structure clipped once at 0.4 (3.27a) and once at 0.85 (3.27b). It can be seen that the SquareHole (Top-Left) does not fit into any block and is removed if the selection ranking is clipped at 0.85 while it is still visible when clipped at 0.4. Compared to the clipping of 0.95 in Figure 3.20b, the RectBig is still visible and would be selected even so the resulting decoding and overlap gaps in the structure representation.

The last two parameters for this algorithm control the gan output rather than the decoding itself. The first option (**combine layers**) controls if all output layers of the



(a) Clipped at 40 (b) Clipped at 85

Figure 3.27: Clipping the size ranking matrix at different hit probability values.

GAN should be combined into a single layer. The second parameter (**Round Integer**) controls whether the pixel values should be rounded to the closest integer value. If all layers are added together, the decoding has to assign a material to the selected block by checking which layer had the most influence over the selection. In other words, the material is assigned by checking which layer had the highest confidence a block should be at the position of the selected block.

3.3 Model Training

With the encoding and decoding of a level representation covered in the previous sections. This section describes how the data presentation is processed and used for training.

Figure 3.28 describes the abstract training process of any gan model. The given interchangeable components are the specific GAN model and the optimizer function labelled Train Stepper. Given a dataset of n level representations, the first selection is what GAN model is loaded. The chosen level representation defines the dimension of the model's layers, as the generated output has to match that of the level representation.

The train stepper is responsible for training the discriminator and generator over one batch. It also encapsulates the previously mentioned objective functions used to train the networks. Two implemented training algorithms have been tried. The original approach described in Section 2.3.1 and the Wasserstein-GAN with gradient penalty, explained in Section 2.3.3.2 as a state-of-the-art training algorithm.

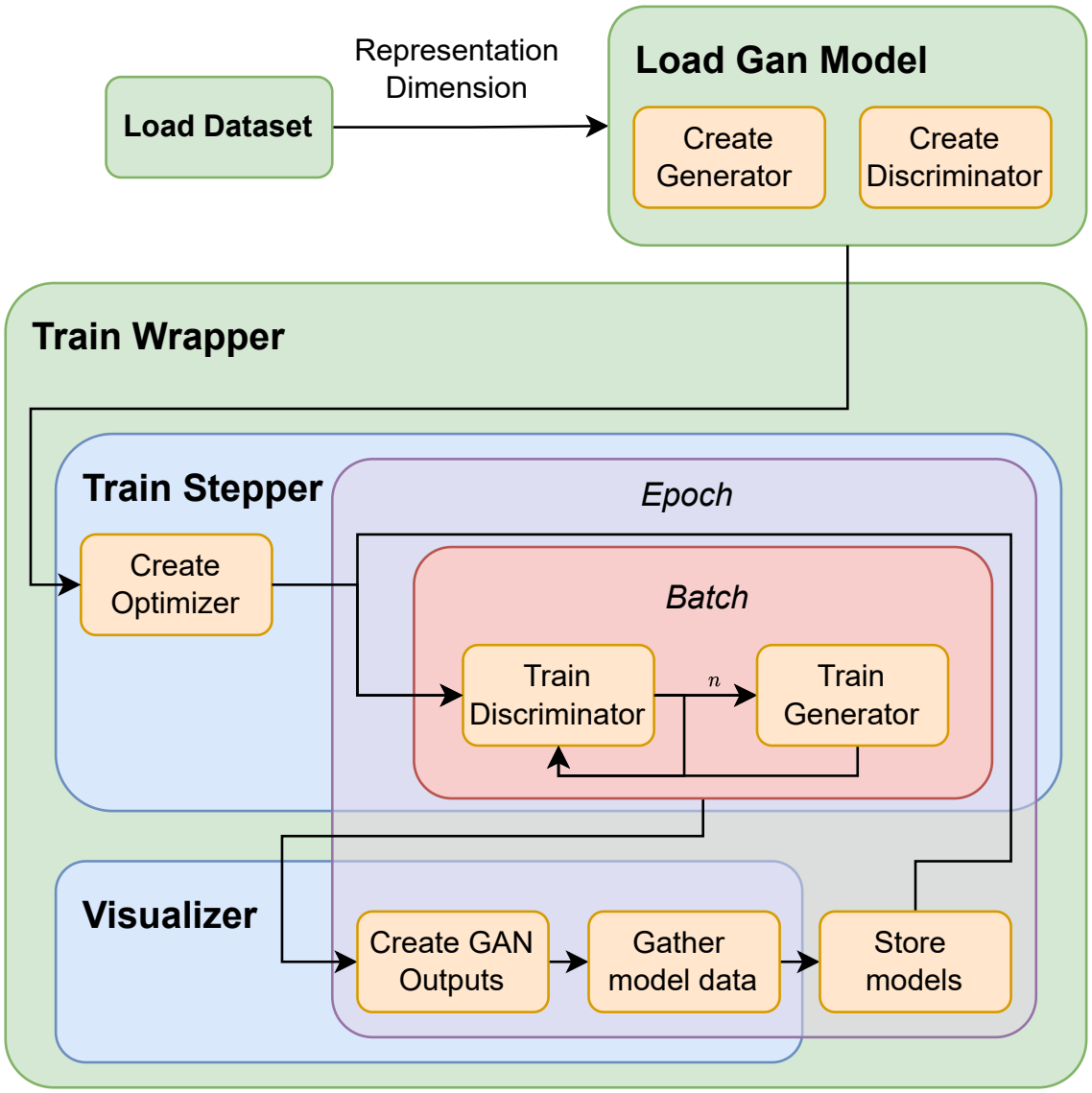


Figure 3.28: Flow diagram of the training algorithm with interchangeable GAN Model and Train Stepper.

4 Approach

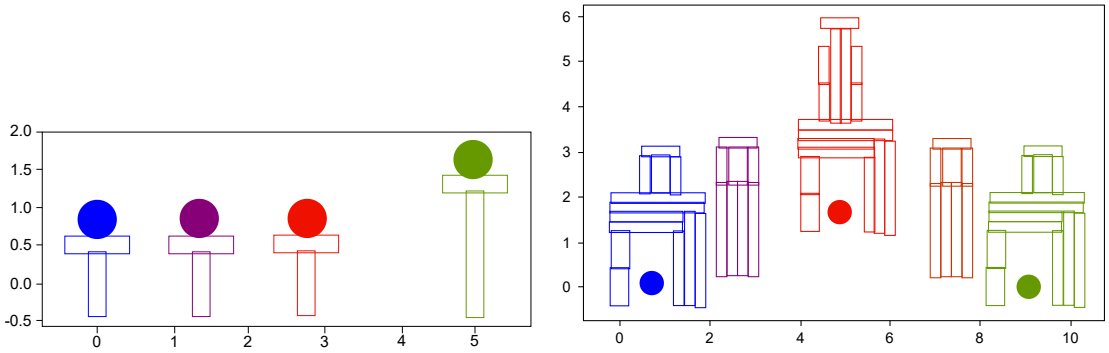
With all foundations covered in the previous chapters, this chapter puts the pieces together into individual tests. This includes the different created datasets used in various training runs. Each combines the data creation process and the source of the structures, the encoding method for each structure and any further preprocessing step. The data creation includes the simulation of the structure in the Science Birds game, and if enabled the playing of the level with AI. The WebSocket interface of the game was modified to allow interaction with the structure generation process to retrieve the data and still allow for Science Bird AI to interact with the game through a separate WebSocket connection.

One training run is defined by a combination of the dataset, model architecture and training method and hyperparameters. The specific models used in the experiments are grouped in two sets, and their differences are explained in Section 4.2. After the training has been completed, the quality of the results can be visually evaluated by plotting the output and quantitatively evaluated by reviewing the amount of stable decoded levels. Finally, the created application is presented in Chapter 4.4, which is used to interact with every previously mentioned aspect of the thesis.

4.1 Data Creation

The dataset is one of the most crucial parts of training a machine learning model, as it defines what the model tries to create. There are two options for creating a dataset, human-made levels and computer-generated ones. Only a few human-made Angry Birds levels are available, and their design makes it difficult to parse into a block representation. The limitation of only using 90-degree block rotations removes a chunk of available levels. Of the 100 levels that were ported into Science Birds, only 50 contain 90-degree increment rotation in their design. And also, for training a machine learning model to create a variety of levels, 50 is too little. Due to these reasons, all training datasets were created using Stephenson and Renz (2017) level generator, presented in Section 2.4.2.2.

The generator can be tuned to create levels with specific widths and heights, predefined block types and amounts of structures. It also only uses 90-degree rotations in its creations. Even so, the generator can produce levels that contain only one structure the data creation process was built with every kind of level in mind.



(a) Loading GAN Models (b) Storing GAN Models

Figure 4.1: Hand-created level with multiple structures and their grouping by colour.

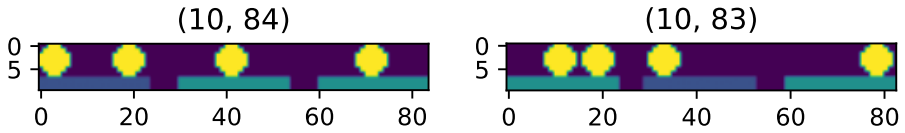
Given a level with multiple structures, the data creation process filters the slingshot platform and groups each block into a structure. The algorithm iterates over every block and searches for the closest structure. If no structure is found, a new structure group is created, while if only one group is found, the block gets assigned to it. If more than one structure is found, the block functions as a binding piece and they get merged together. While the structure representation could incorporate multiple structures, this would increase the size of the structure representation. In Section 4.2 is further discussed how the GAN model limits structure representations dimensions. On top of that, a composition of multiple structures can be combined into one level after the structures are generated.

Each structure's metadata, namely start- and end position in both axis and the resulting width and height, the number of blocks grouped by special block, material and pigs, and simulation data like stability and AI score, are collected and stored alongside the structure representation. More on the simulation in Section 4.1.1. Collecting AI data was only done on smaller datasets due to the reason that for this thesis, it is not used and would take a lot of time. If no structure-simulation data is collected, the creation process can be parallelized.

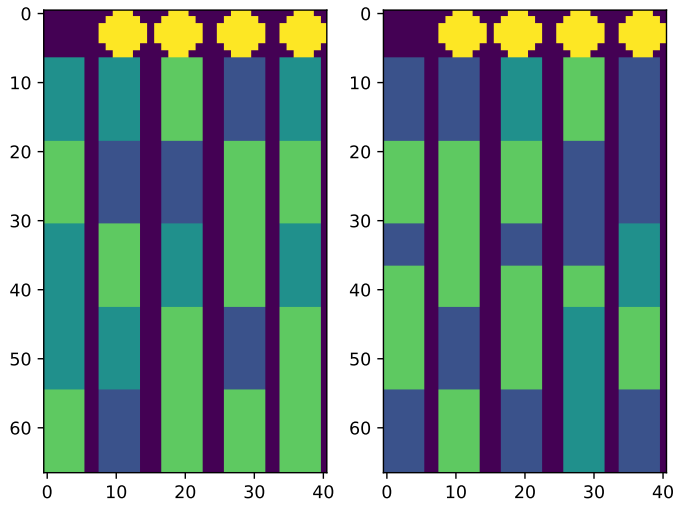
Three different structure data sets were generated. The first and smaller one contained 200 levels with multiple structures and no block-type restrictions. A bigger one, with 5000 levels containing only one structure, but whose width and height are smaller than in the previous dataset. This is also the dataset used in a majority

of training runs. The last created dataset restricted the allowed materials to only one, allowing every multilayer representation, which scales by the number of materials, to become smaller or become a one-hot encoding resulting in better semantic representations, as previously discussed.

Also previously discussed is the prowess of GANs to mode collapse, which is circumvented by carefully balancing the training of the generator and discriminator but also balancing the diversity in the dataset. Originally datasets of, for example, face synthesis tasks only rarely include the same faces, so it becomes a problem. On the other hand, using a generator to create many structures can result in several repeating structures, which would increase the probability of a mode collapse. One approach to circumvent this problem is to filter the datasets to include more diversity. The two filters that have been implemented search once for levels with the same metadata and unify the number of levels with similar dimensions.



(a) Metadata Match



(b) Same Shape

Figure 4.2: Filtering by searching for similar levels

The first filter uses the previously collected metadata and filters every structure with the same amount of blocks per material combined with having the same width and height with a 0.1 margin. The two levels of Figure 4.2a have the same metadata, and one is removed. While the two levels in Figure 4.2b use different blocks and do not have the same metadata, they have the exact same structure, and one is also removed.

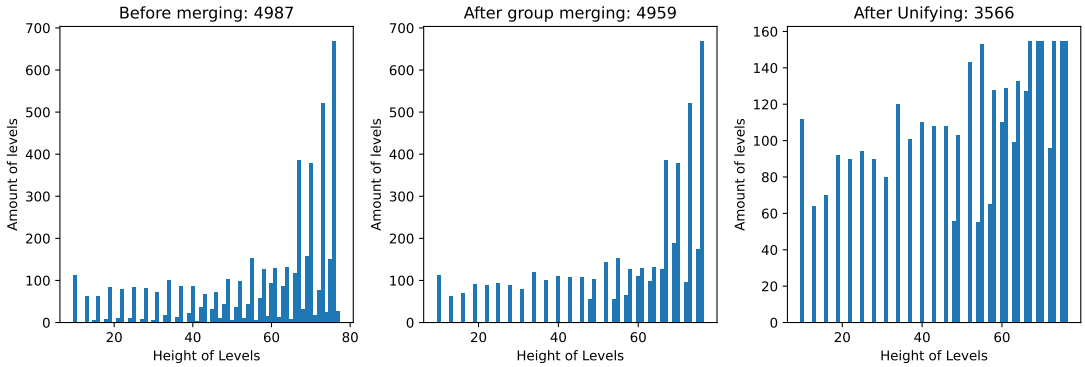


Figure 4.3: Unifying the structure diversity and creating a more equal amount of levels with different heights in the second dataset.

The main goal of the second filter is to equalize the number of levels per height with the intent that the GAN produces diverse levels. It groups every structure by its height and merges the closest groups with less than 50 levels into buckets. Each step is visualized in the bar chart in Figure 4.3. The average of the number of levels in each bucket gets calculated, and every bucket gets trimmed by this average. From the 5000 levels of the second dataset, 3566 levels remain while 3957 remain in the second dataset. The last step is to convert the dataset into a TensorFlow format to load them easily in the training process. On loading a dataset, the structure representation gets normalized between -1 and 1 and transformed into tensors.

4.1.1 Simulation modifications

The open-source Science Birds game has an interface designed to play with AI. It originally provided only the functionality to create screenshots of the whole level and send them encoded to the AI, which uses Computer Vision (CV) to analyze the structure and plan its move. While this step is part of the AI competition, it is not beneficial for collecting data about the generated levels, and more functionality is built into the simulation to help collect data. To receive the simulation data, the generator framework implemented a WebSocket (WS) server to which the simulation can connect to.

The WS functionality was extended with a few features, and already existing features were modified: (1) To be able to connect simultaneously to the Science Bird games, a second WS is added, which waits for a connection with the generator while still allowing an AI to connect normally. It uses the port provided as a command line argument which allows to run multiple instances next to one another which accelerates the level simulation. (2) The WS call that selects a level now answers collected

data of the loaded level. It comes with the capability to stop the game time to review the initial block positioning and to better screenshot a collapsing structure and the option to wait for the structure to become stable to have more data in the answer. A function that iterates over all present levels and collects their stability data. (3) A few minor added features include screenshotting only the structure instead of the whole level, a callback which waits till a level is finished loading, enabling the AI WebSocket, and receiving the data of played levels.

The second main extension was to add data collection to the playing or simulating of a level, depending on whether the game is actually played or not. The data that gets collected, from the moment the level is loaded, is (1) the number of dying pigs or (2) broken blocks grouped by material, (3) the initial and cumulative damage done to blocks, (4) whether the structure is stable based on the total block velocity and if the level is played, (5) the amount used birds and (6) if it was won. The initial damage represents the damage after the blocks stop moving after the initial level loading. The metadata can be requested at any point and does not require the level to be played.

4.2 Gan Models

With the data creation process covered, the concrete models used to create the structures are explained in this section. As mentioned in Section 3.3, two training algorithms and two model types have been tried. Both versions are implemented in different variations depending on the used dataset. For example, a multilayer structure representation comes with more input layers, and a model made for a single-layer representation could not handle the representation.

4.2.0.1 Simple GANs

The design of the first set of GANs are based on the used dataset and is overall shallower and uses only two deconvolution blocks compared to the second set. It is also primarily used with the original GAN training algorithm described in Section 2.3.

Figure 4.4 visualizes the first version of a generator from the first set. Each layer is grouped into a functional block that combines a layer that uses trainable weights, a normalization layer and an activation function. The values above each block show the respective output shape or resolution of the last layer of the block. It can be seen that “based on the used dataset” translates to the last output layer matching

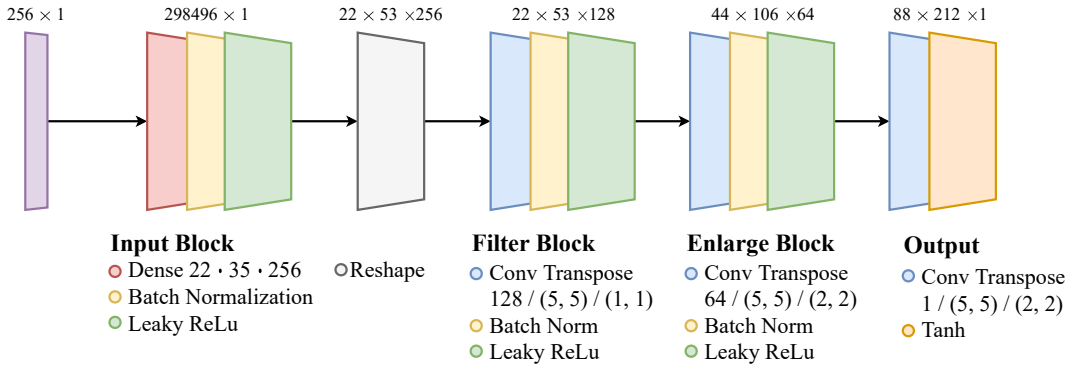


Figure 4.4: Layers of a generator from the first set of GANs.

the max dimension of the structure representations of the used dataset. To reach this dimension, two transposed convolutions are used with a five-wide kernel and a stride of two, doubling the previous resolution. The initial noise vector with a size of 256 gets put into a dense layer, creating the required amount of outputs for the following filter block. Overall layer blocks, the generator uses batch normalization as recommended by Goodfellow et al. (2016) and Radford et al. (2015) with leaky rectified linear units (LeakyReLU) in their activation except for the last layer, which uses Tanh to recreate the datasets dataspace.

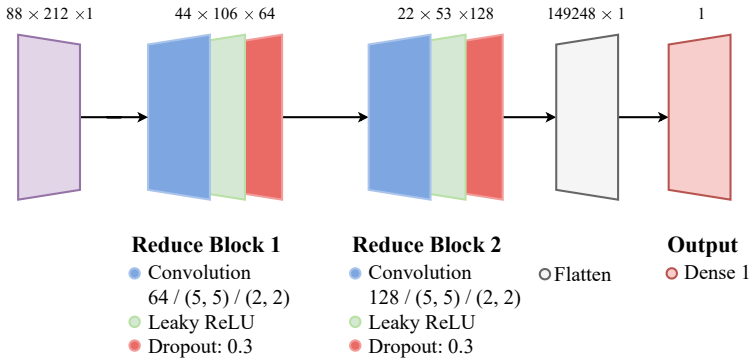


Figure 4.5: Layers of a discriminator from the first set of GANs.

The discriminator, described in Figure 4.5, takes the structure as input and uses normal convolution layers with LeakyReLU and dropout as reduction blocks to come to its conclusion. With a stride of two, the reduction blocks half the resolution each time. To produce a single scaler, a dense layer is used, which takes the flattened output of the last reduction block.

This set's remaining GANs differ only in the final output resolution. This is archived by changing the number of output neurons in the initial dense layer, followed by using the reshaped layer. The implemented GANs have an output resolution of 100×112

and 100 × 116, which is smaller than the first GAN, removing the need for a big initial dense layer. The reduction was made possible by removing the exceptionally large structures from the test dataset.

4.2.0.2 Convoluiton GANs

The second set of GANs is based on the DCGAN explained in Section 2.3.3.1.

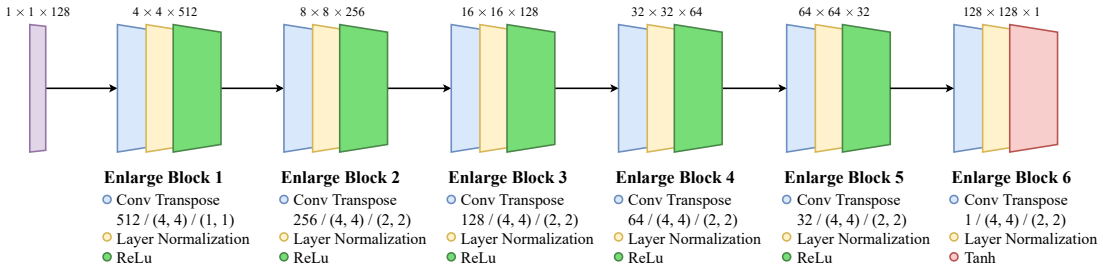


Figure 4.6: Fully convolutional generator based on the DCGAN.

Figure 4.6 describes the deeper generator model. Compared to the first set, the GANs of this set uses exclusively transposed convolutional layers to enlarge the image resolution. The stride of two doubles the resolution with each layer block, limiting the possible resolution to a power of two. As described in Section 2.3.2.1, the recommended batch normalization layers were replaced by layer normalization. Another reason to avoid batch norm is that a WGAN assumes independence between samples in a batch, which would be introduced through batch normalization.

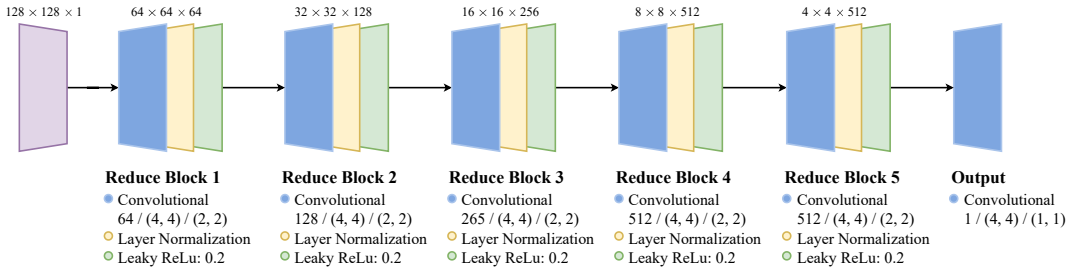


Figure 4.7: Critic that only uses convolutional layers

The discriminator or Critic is visualized in Figure 4.7. While the generator only uses transposed convolution, the Critic only uses convolutional layers to reduce the img size to arrive at a decision. Instead of normal ReLU activation, it uses leaky LeakyReLU as recommended by Radford et al. (2015) in their architecture guidelines for stable Deep Convolutional GANs.

The architecture for the multilayer representation simply uses more filters in the generator's last layer and the discriminator's first layer. This allows the generation of a four or five-layered output depending on the provided structure representation. Based on Volz et al. (2018) recommendation, the last model that was tested uses a different activation function in the last layer of the generator. Instead of the original Tanh, they used a ReLU, which improved their results.

4.3 Evaluation and Training

This section reviews the concrete evaluation methods used to validate various aspects of the thesis. It briefly reviews the used training hardware and capabilities that mainly limited the amount of tested model variety.

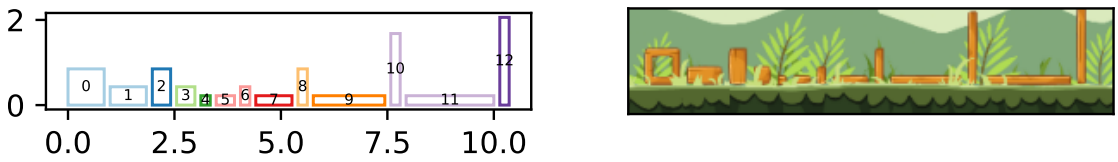
4.3.1 Evaluation

Various aspects of this thesis can be evaluated, such as the model's capability, the encoding and decoding algorithms, and their respective capabilities to decode the output of a model. Evaluating a GANs models capabilities usually requires a human expert to review the results manually. If the GAN mode collapsed and produces only a small variety of outputs or if the produced images are not usable can be easily seen. This visual inspection is mostly enough to compare the GANs performances to create a stable structure because most do not create a decodable structure representation or, in the case of the one-element encoding, create non-playable structures.

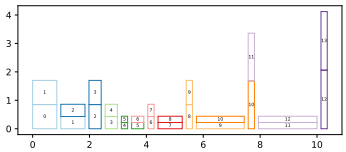
4.3.1.1 Encoding Decoding

The encoding and decoding can be tested by how reliable the algorithm can decode the input structure. Reliability in encoding/decoding can be interpreted in a few ways: (1) if it recreates the structure with the original block selection, (2) if the positioning of the blocks is equally close, and (3) if the stability is affected in any way. The first two points can be evaluated with a fabricated test structure that uses a variety of block combinations and reviews the decoded structure.

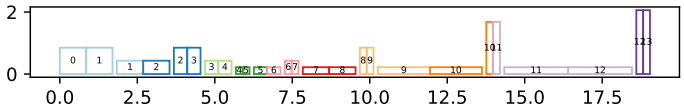
The test structure is visualized in Figure 4.8. It simply uses all used block types and puts them next to one another. Figure 4.8b and 4.8c show that the stacking direction can be modified, and the spacing of the blocks can be increased in such a way that it varies between each block type.



(a) Original with screenshot



(b) Verticaly stacked



(c) Horizontally stacked

Figure 4.8: Test structure used to evaluate decoding behaviour of different decoding algorithms.

To validate the effect on the stability of the structure, an evaluation algorithm simulates the original structure and its encoded and decoded counterpart and compares the collected data. This can be compared to the encoding and decoding approach introduced by Tanabe et al. (2021) in their VAE-LSTM model.

4.3.1.2 Quantitative Evaluation

A quantitative evaluation of the GAN with the most promising capabilities to generate stable structures is done. To do this, a grid search over the parameter space of the confidence decoding algorithm with a fixed set of generated structure representations has been implemented. Using the found parameter, a quality search over 4400 generated structures has been done to search for stable structures with desirable characteristics such as block variety, block amount, or structure dimensions.

For the grid search, a total of 200 structure representations are initially generated for this test. Each gets decoded with every parameter combination in the search space, followed by simulating the resulting structure. For every structure, its metadata and simulation data are collected. The parameter space is created through the parameters of the "Confidence Decoding" explained in Section 3.2.2.3. The boolean type parameters, namely the (1) round to next int, (2) use custom kernel scale, (3) use of minus one border and (4) if all layers should be combined, can be in two states. These alone create $2 * *4$ parameter sets. As the number of parameter combinations grows exponentially, only a few values of the continuous value-based parameter can be chosen. The five negative values $[-10, -5, -2, -1, 0]$ are tested for the negative air parameter. For the clipping parameter, only the four values $[0.1, 0.5, 0.8, 0.95]$

are tried. This creates $2 \cdot 2 \cdot 2 \cdot 2 \cdot 5 \cdot 4 = 320$ different sets of parameters that are tested.

Doing an exhausting grid search that includes decoding and simulating all structures with every parameter combination is a lengthy process. To speed up this process, the decoding of all 200 structures is parallelized to fully utilize the computing power, which reduces the time it takes to create the structures. Similarly, multiple instances of Science Birds are started for the simulation, and each receives an equally sized set of decoded structures to collect the simulation data.

4.3.2 Training

Write a few words for training.

Training a GAN is relatively expensive as we train two models, and due to the lower learning rate, a WGAN requires the overall training time is longer. The initial training of the first set of GANs was executed on a Jetson AGX Xavier card. Its low power requirement meant that while having much GPU memory, the performance was this not suited to train the deeper convolutional GANs. Most of the convolutional GANs training is done on the RWTH High-Performance Computing cluster, which uses NVIDIA Volta 100 GPUs, reducing the training time from 7 days to 2 days.

4.4 Testing application

An application has been developed to test the different aspects of the thesis. This includes the encoding and decoding algorithms with an interface to add parameters to the decoding process to review their effect on an encoding. Another aspect of this application is to review the output of the different GAN models and store results for later use.

Figure 4.9 is a screenshot of the application, which is separated into four regions. Left is the drawing area, the top row of buttons are general controls, on the right is a location for figure visualization, and the bottom row handles the loading and storing of GAN images. It can be seen that the test structure is loaded into the drawing area and that its decoded result is visualized on the right. Therefore a feature is the loading and visualization of test structures stored in a predefined folder using encoding selected in the top right.

To test different edge cases of the decoding algorithms, the drawing area allows the creation of a structure in the visual and one-element structure representation. It is

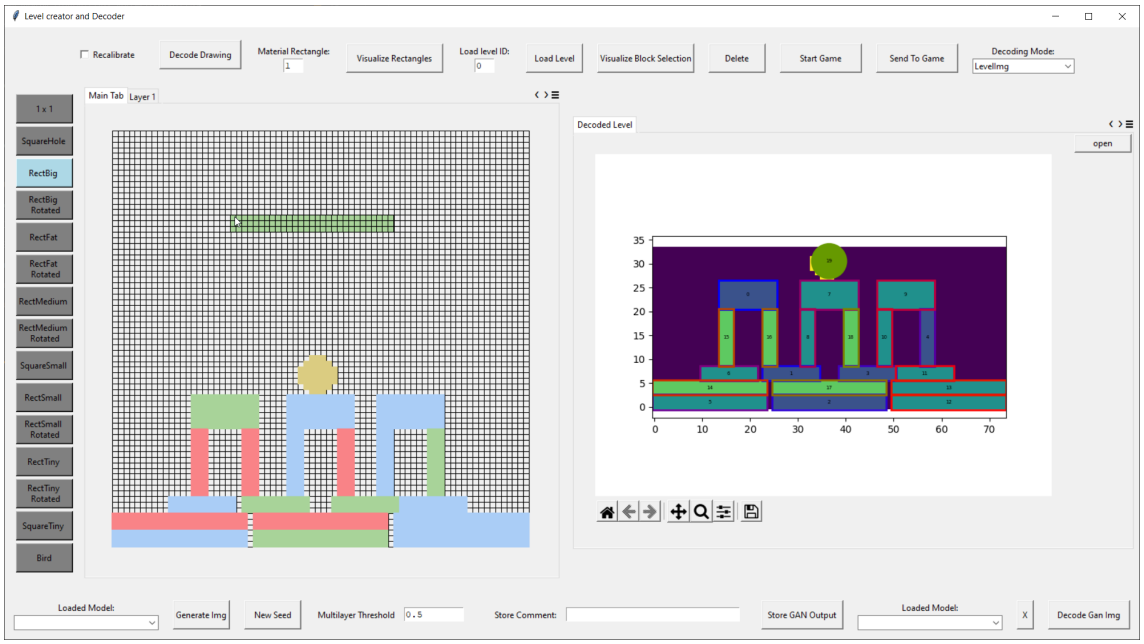


Figure 4.9: The whole testing application.

also used to display the, into one layer reduced, GAN output. If the representation uses multiple layers, the respective output layer is visualized and rounded to its closest integer. Figure 4.10 shows the buttons that are used for the block-type selection. The drawing area functions like a usual painting area where the pencil is in the shape of the selected block type. The drawn smiley shows that any arbitrary shape can be created. Calling the decoding algorithm opens a parameter window, as seen in Figure 4.11. This uses reflections to set the parameter in the decoding object and calls the decoding function with the input selected. The smiliy decoded is attached in Figure A.7. If the decoding mode is set to one element, the drawn image colours only the centre block of the selected block type (Figure A.7).

Each available model gets listed, seen in Figure 4.12a and can be called to see their output. Internally the used normalizing, encoding and flatting methods are stored that get loaded alongside the model. After a model is loaded, a random seed is created, and its output is displayed. The output can be labelled, stored and loaded at a later time, as seen in Figure 4.12b.

Each layer of the output in the multilayer encoding is displayed in a separate tab. Figure 4.13 shows the second layer and its output if rounded to the closed integer. The remaining feature allows you to start the science birds game and to send a decoded structure to the game.

Figure 4.10: A custom shape is drawn in the structure drawing area.

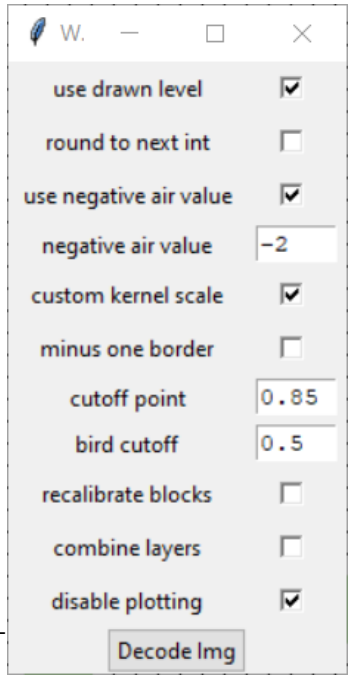
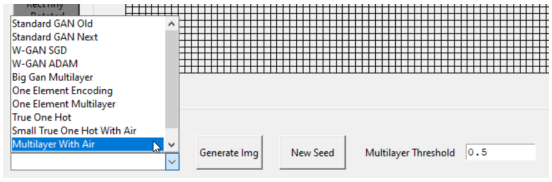


Figure 4.11: Decoding Parameters



(a) Loading GAN Models

(b) Storing GAN Models

Figure 4.12: Bottom row responsible for loading and calling a GAN model with the persistent storing of data.

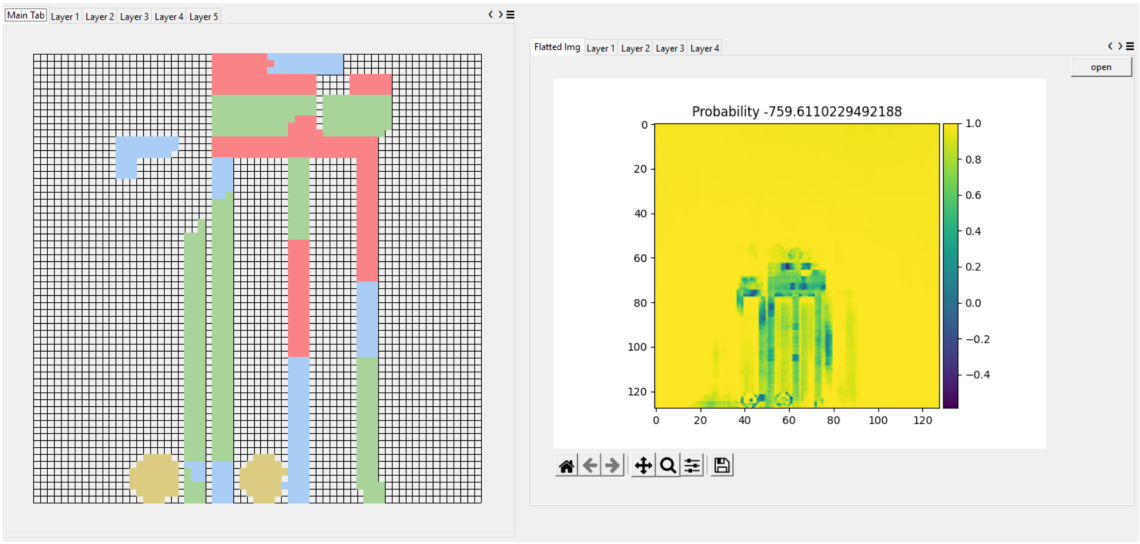


Figure 4.13: Reviewing multiple layers of the output in the raw version (right) and rounded to the next integer in the drawing area.

5 Results

The result chapter reviews the individual training instance and experiments to evaluate a model's performance with a decoding algorithm. In this chapter, only a few exemplary outputs are shown for each test, with more generated examples are online available.

Add a link to G-Drive

5.1 Visual Review

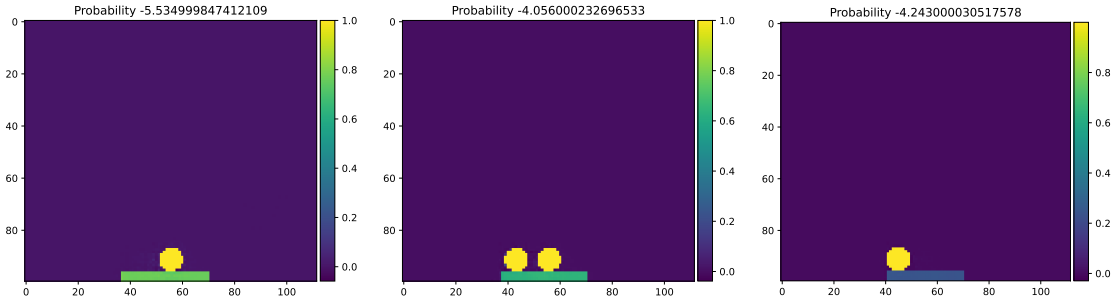
As previously mentioned, the majority of GAN models are only reviewed visually, as further inspections would be unwarranted in most cases. Each individual model training is defined by the design decisions of the encoding algorithm, used data set, model architecture, training algorithm and training hyperparameters.

5.1.1 Simple GANs

The first reviewed results are from the set of simple GAN architectures described in Section 4.2.0.1. Four different runs are done with the first two training instances using the original training algorithm, while the following two have been trained using the W-GAN training algorithm. The dataset uses the "Visual Encoding" with the dot method for encoding the blocks with a single-layer representation.

5.1.1.1 Original GAN Training

Figure 5.1 shows the first results with the original training algorithm and the Simple GAN architecture. It can be seen that the model mode collapsed into generating only simple structures. All generated structures contain only one rectangle at the bottom and either one or two pigs sitting on top with no further variation. Further

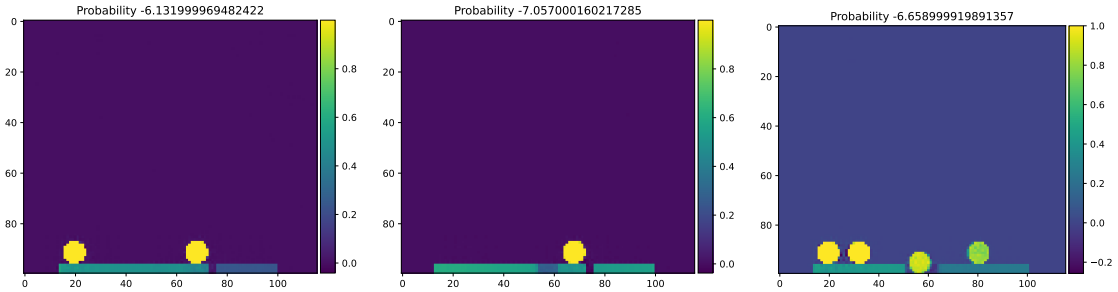


(a) Structure 1 (b) Structure 2 (c) Structure 3

Figure 5.1: Structure generated with the Simple GAN architecture.

in Figure 5.1c can be seen that the created rectangle is in value closer to air than the next block value which is visualized in Figure A.8.

The dataset used in the first training iteration wasn't filtered for repeating structures described in Section 4.1. Training the same model but with a filtered dataset produces the results shown in Figure 5.2.



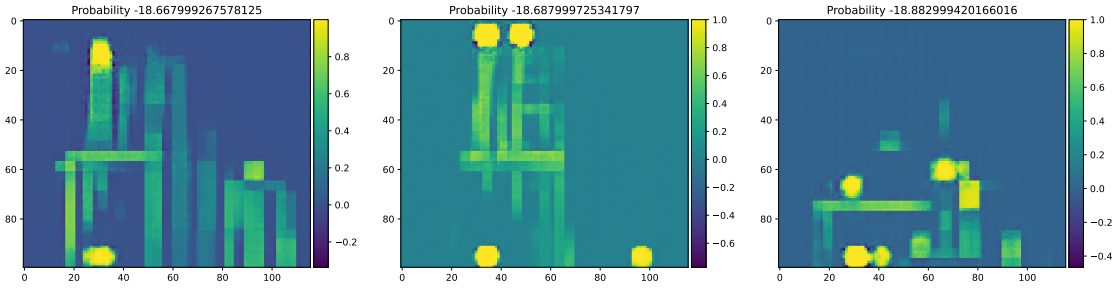
(a) Structure 1 (b) Structure 2 (c) Structure 3

Figure 5.2: Structures generated with the Simple GAN architecture and a filtered dataset.

It can be seen the variety in generated structure representations is higher compared to the first training iteration. Now it contains two blocks instead of one and also contains a greater variety in the number of used pigs. This shows the importance of balancing the dataset when using GANs.

5.1.1.2 WGAN Training

As mentioned in the Mode Collapse Section 2.3.2.1, the WGAN training algorithm was one of the major factors in combating the mode collapse problem. Using the same GAN architecture but trained with the WGAN algorithm on the filtered dataset produces the results of Figure 5.3.

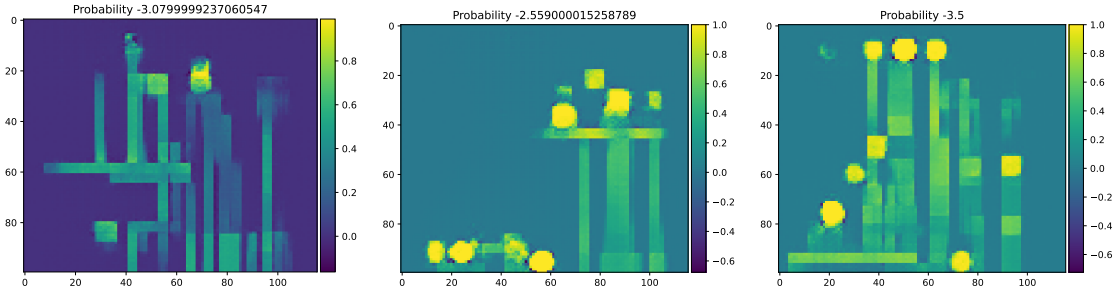


(a) Structure 1 (b) Structure 2 (c) Structure 3

Figure 5.3: Structures generated with the Simple GAN architecture, filtered dataset and the WGAN training algorithm with a stochastic gradient descent optimizer.

The generated structure variety is more diverse compared to the training iterations using the original training algorithm, but the quality of the generated structures is worse than before. The representations contain more uncertainty which can be seen in the unclear block boundaries and the overall blurry appearance of most of the blocks. While the perceived structure in Figure 5.3a looks almost stable, the generated structure in Figure 5.3b has floating block elements and can not be stable at all. More generated structures are attached in Figure A.9.

To remove the probability of not training long enough, instead of training for 5000 epochs, the following training iteration is over 15000 epochs. Also, instead of the stochastic gradient descent optimizer (Robbins 2007), the adam optimizer (Kingma and Ba 2014) as recommended by Radford et al. (2015) was used. Furthermore, the dataset with 3000 structures encoded using the Calculated Visual encoding was used to increase the structure variety.



(a) Structure 1 (b) Structure 2 (c) Structure 3

Figure 5.4: Structures generated with the Simple GAN architecture, filtered dataset and the WGAN training algorithm with a Adam optimizer over 15000 epochs.

The generated structures, visualized in Figure 5.4, have the same perceived quality compared to the previous training iteration. It seems to be the case that the GAN

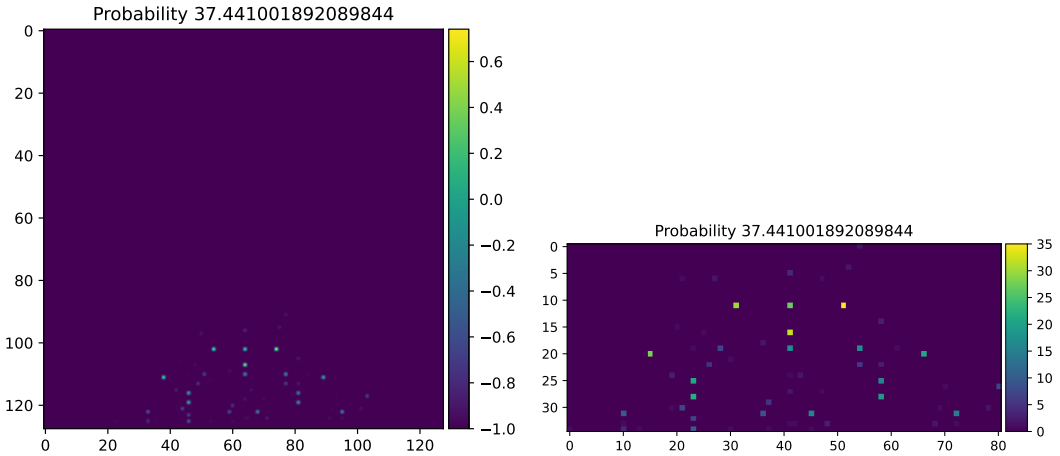
is not limited by the training length. More generated structures are attached in Figure A.10.

5.1.2 One Element Encoding

Decoding these generated structure representations with the mentioned uncertainty which visualizes itself in the blurry blocks is difficult. A more promising approach to create a decodable structure generation is given with the always decodable "One-Element" encoding. The models used in this approach are the Convolutional Models described in Section 4.2.0.2. Every training iteration over 15000 epochs, with the WGAN training algorithm

5.1.2.1 Single-Layer

The first iteration of using the One-Element encoding uses the single-layer representation. This means the value range of the structure representation is between $[0, \ldots, 40]$ and every non-zero pixel represents a block.



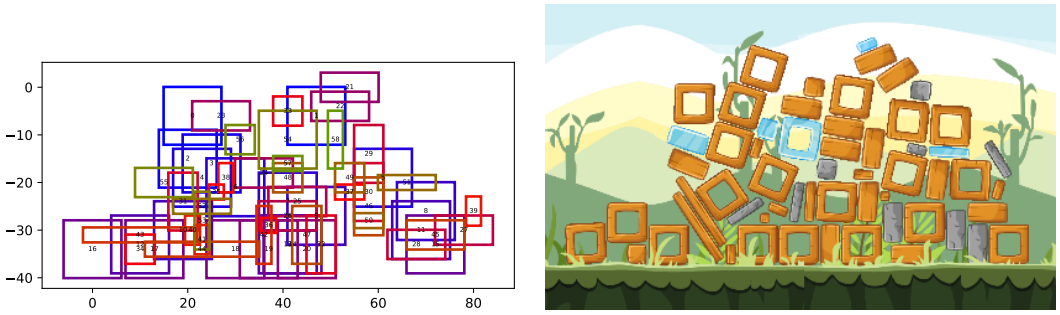
(a) Original generated structure representation

(b) Readjusted and rounded to the next integer

Figure 5.5: The generated structure representation trained on the One-Element encoding once in the normalized data space (a) and once in the original data space (b) rounded to the next integer.

The originally generated structure representation in the normalized data space of $[-1, \ldots, 1]$ is visible in Figure 5.5a. Moving the values into the original dataspace and trimming all the air results in Figure 5.5b on the right. It can be seen that a few encoded blocks form lines, and an overall structure can be made out. The problem in this representation is that every uncertainty or noise in the encoding becomes an

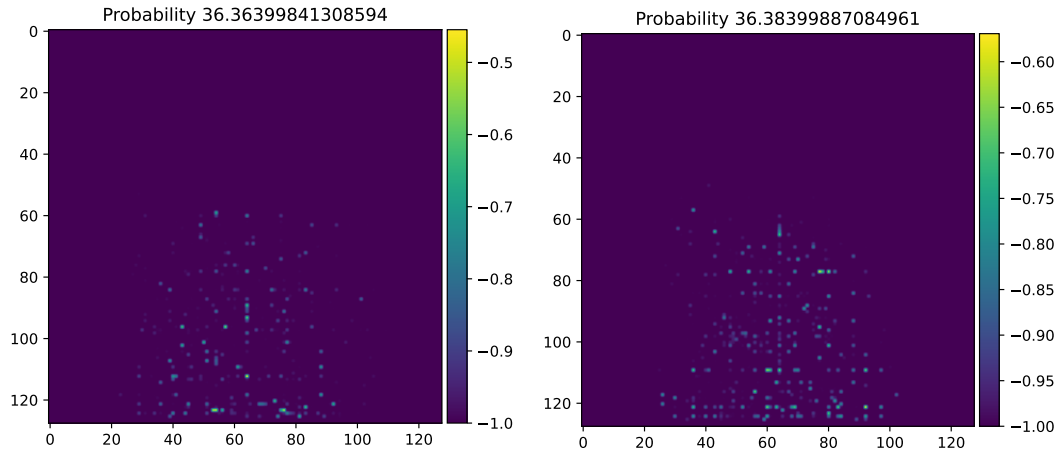
encoded block with no way to differentiate if the pixel is due to noise or if it is an intended block placement. As the representation is still decodable with all the noise, the resulting structure is given in Figure 5.6a.



(a) Decoded into a structure (b) Send to the game and simmulated

Figure 5.6: The structure of Figure 5.6 decoded into individual blocks.

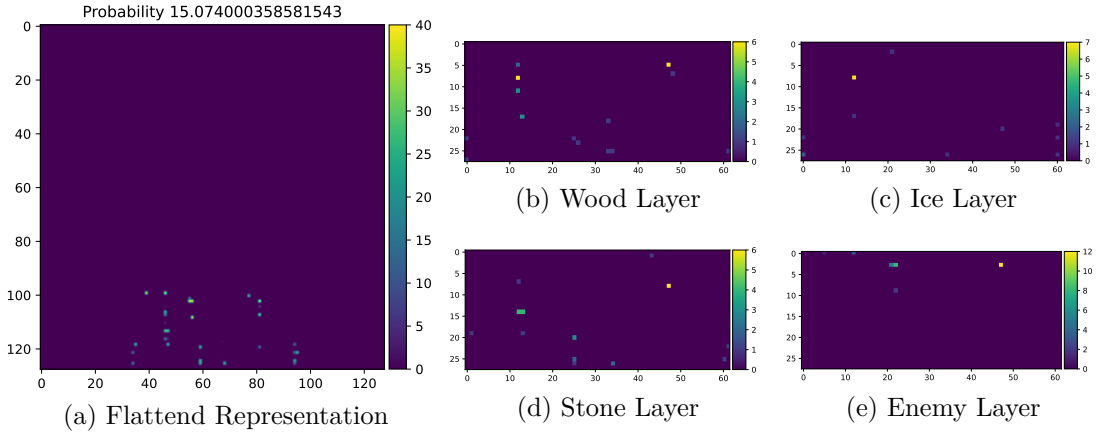
Besides the fact that there is no block that doesn't overlap another block, there also does not exist an enemy in the structure. When moving the blocks to the simulation, as seen in Figure 5.6b, the game automatically pushes each block outside of one another, resulting in a big pile of blocks.



(a) Structure 2 (b) Structure 3

Figure 5.7: Generating more examples of the Single-Layer One-Element Encoding.

When looking at other generated structures of the One-Layer encoding, two are exemplarily visualized in Figure 5.7, it can be seen that the initially selected structure is comparatively small. The value ranges of the two structures indicate further that they only contain blocks of the first wood material instead of various materials indicating that the GAN is unable to produce singular pixels with high values surrounded by zeros. The decodings of the two structure representations are attached in Figure A.11.



(a) Flattend Representation (b) Wood Layer (c) Ice Layer (d) Stone Layer (e) Enemy Layer

Figure 5.8: A structure representation generated in the One-Element multilayer encoding.

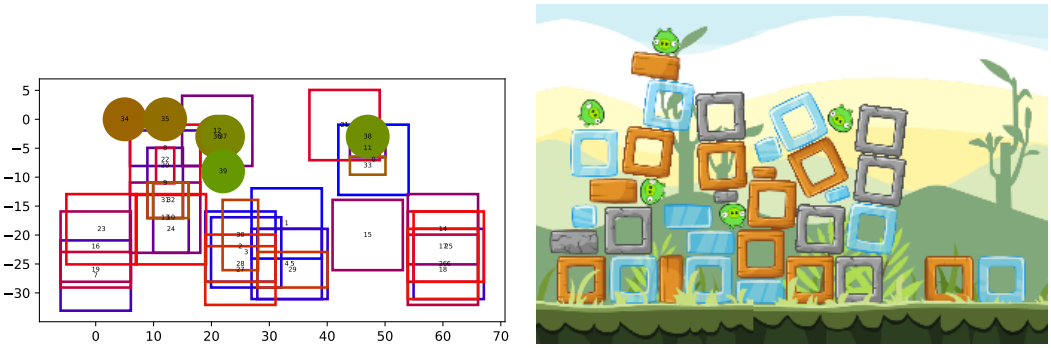
5.1.2.2 Multi-Layer

As described in Section 3.1.4, moving each material onto its own layer can alleviate the problem of high values to a certain degree as each layer now uses a value range of $[1, \ldots, 13]$. Therefore the next trained iteration uses the One-Element multilayer representation.

An example of a generated structure is given in Figure 5.15. The Figures 5.15b to 5.15e, on the right, represent each layer of the multilayered encoding. The Figure 5.8a on the left is the flattened reconstruction of the Single-Layer representation used in the decoding process. The reconstruction method uses an argmax operator over the layers to decode from which layer a block comes. As this encoding does not use a designated air layer, the flattening process adds a threshold layer with a value of 0.5 to the beginning of the matrix. The argmax operator creates a two-dimensional matrix of indices that are used to select the value of the respective layer. The maximum layer value of 13 is added depending on the layer from which the block comes.

The previously described problems of existing noise that results in unwanted blocks is not solved with this representation. Most of the generated structures have overlapping blocks that interfere with one another resulting in an unstable structure, exemplarily shown in Figure 5.9, the decoding the One-Element multilayer representation of Figure 5.15.

One major problem of the encodings that use a value range on each individual layer is that the values do not represent how certain the GAN is but rather which block type should be placed.

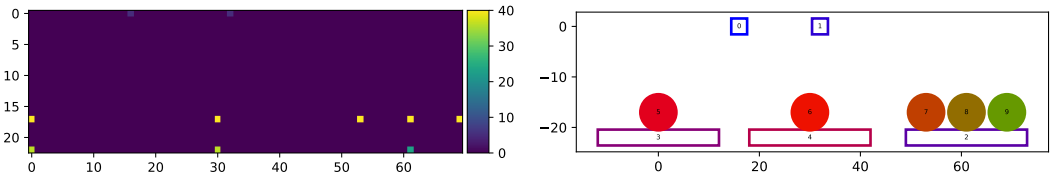


(a) Structure 2 (b) Structure 3

Figure 5.9: Visualizing the decoding and simulation of the One-Element Multilayer representation of Figure 5.15

5.1.3 True-One-Hot Encoding

This problem can be solved by moving each block type to its own layer as described in the One-Hot encoding of Section 3.1.4. This means a value closer to 1 translates into a higher certainty of the model that a block should be at the location. Any kind of noise is supposedly lower in value and can be filtered by removing any pixel below a certain threshold.

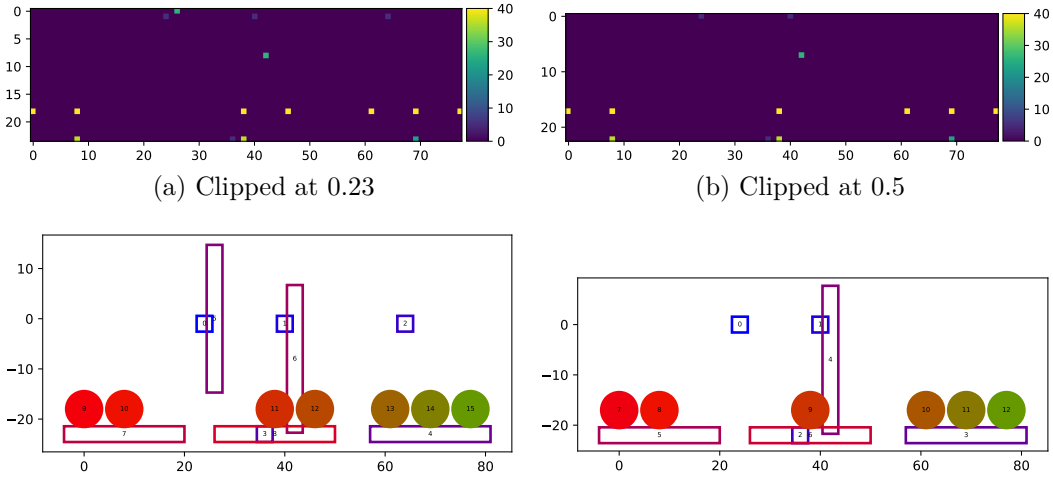


(a) Flattend into Single-Layer (b) Decoded

Figure 5.10: A structure generated in the True-One-Hot encoding using the convolutional GAN with the WGAN training method. The layers are is clipped at 0.7.

Figure 5.10a shows the flattened version of a structure representation in the True-One-Hot encoding. The noise is removed by clipping each layer and removing every value below 0.7, which is impossible in the previous One-Element representations. It can be seen in the decoded structure visualized in Figure 5.10b that the model placed the enemies perfectly on top of the rectangular blocks on the bottom. This shows that the model learned to reconstruct structure formation to a certain degree.

A problem with this model can be seen when reviewing the same structures at different clipping thresholds. Figure 5.11 shows the previously generated structure

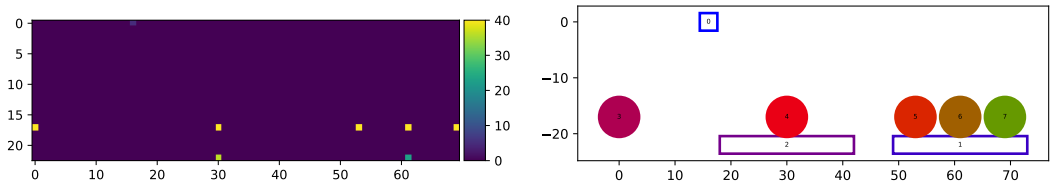


(a) Clipped at 0.23 (b) Clipped at 0.5

(c) Decoding of 0.23 (d) Decoding of 0.5

Figure 5.11: The same structure representation of Figure 5.10 clipped at different value thresholds

of Figure 5.10 clipped at different thresholds. It can be seen that the same problems of overlapping blocks occur as in the previous One-Element encoding. The version with the lower threshold, in Figure 5.11a has evidently more compared to Figure 5.11b.

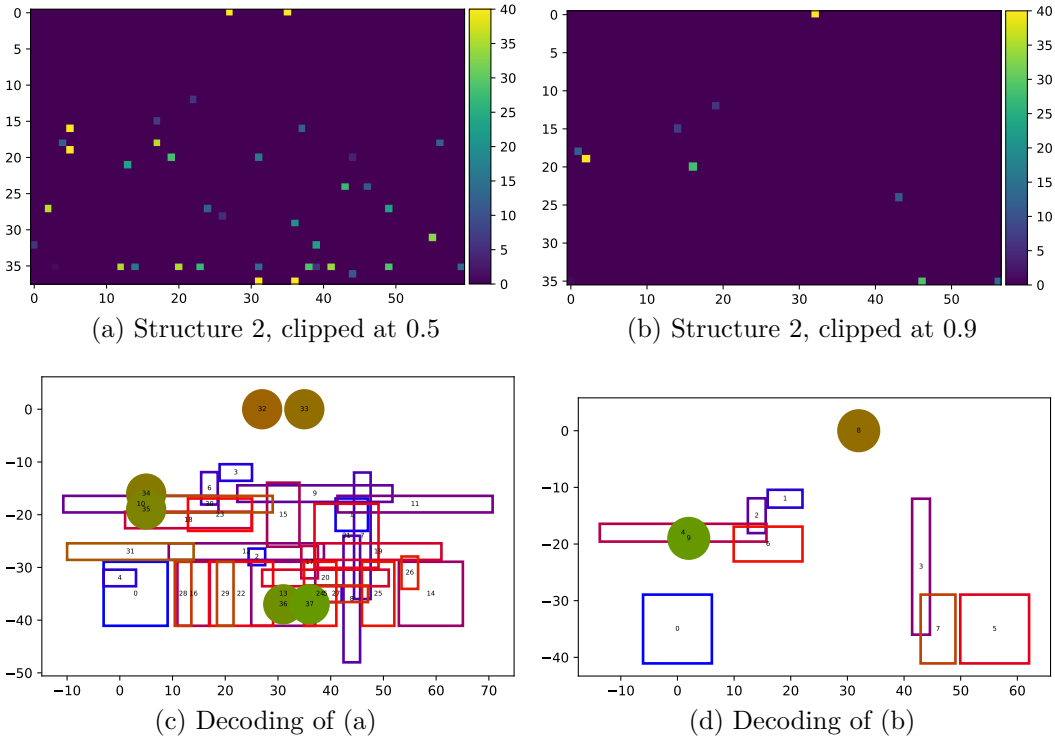


(a) Clipped at 0.95 (b) Decoding of the 0.95 clipped

Figure 5.12: Visualizing the problem with a high clipping parameter.

One could think a solution to this problem is simply setting the threshold to a high value to remove every block which contains any uncertainty. Figure 5.12 visualizes that it could be the case that structural blocks required for the stability of the structure are removed. Even so, in this example, a high threshold is not crucial for the overall structure, more examples emphasize this problem.

It seems the GAN using this encoding method is able to generate structures, but most of the bigger structures suffer from the same overlapping block issue or are floating sporadic blocks resulting in an unstable structure. The generated structure representation, visualized in Figure 5.13, has the aforementioned problems. Even clipped at a higher threshold leaving only certain blocks behind, contains overlapping and floating blocks meaning.



(a) Structure 2, clipped at 0.5

(b) Structure 2, clipped at 0.9

(c) Decoding of (a)

(d) Decoding of (b)

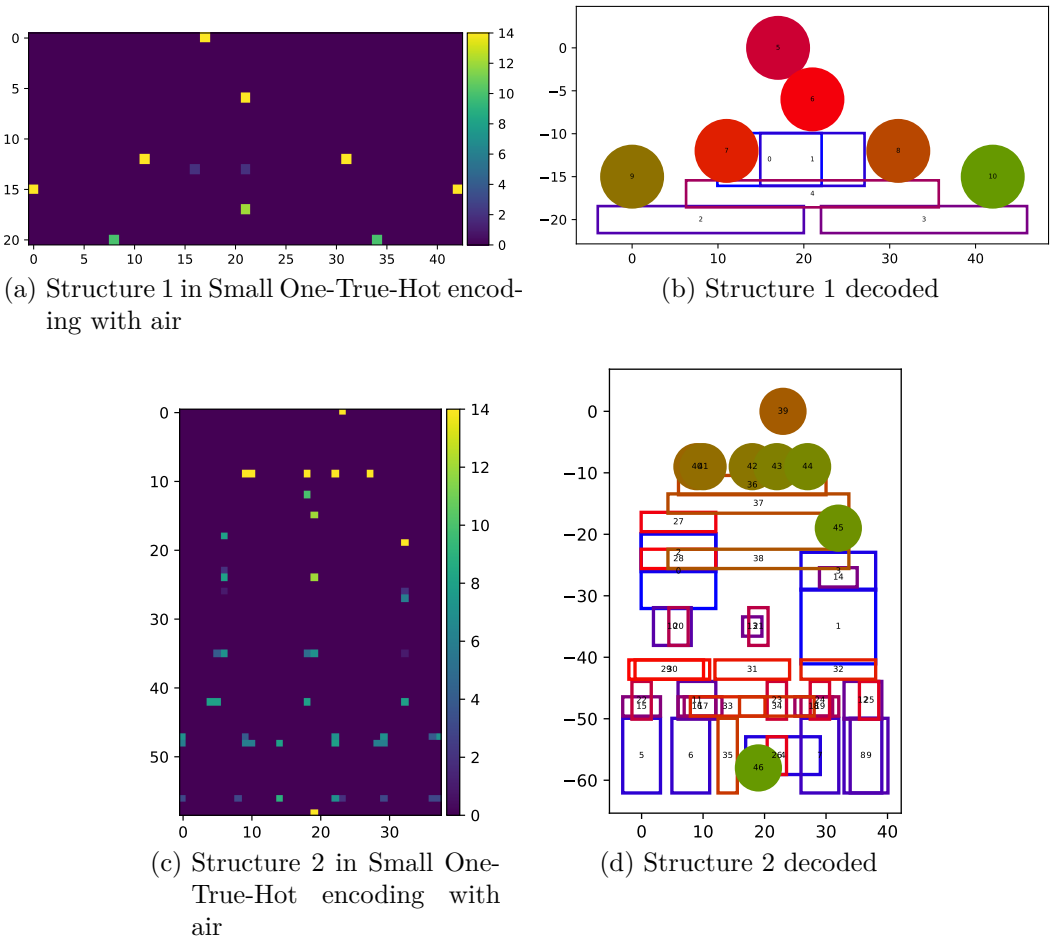
Figure 5.13: Another generated structure in the True-One-Hot encoding, clipped at two different thresholds

Adding an air layer to the representation, as previously explained in the Multi-Layer Section 3.1.4, removes the requirement of a threshold as the decision where the air should go is entirely encoded into the representation. A new True-One-Hot encoding was trained with the air layer in the representation, but training the One-Hot model is relatively expensive. The training of the previous model took 84h on the RWTH cluster compared to the usual 48h with the 4-layered representation due to the size difference of the One-Hot encodings representation.

A smaller version of the One-Hot encoding is trained on smaller structures with only wooden blocks. This means this smaller version uses only a 15-layered representation, one air layer, 13 block layers and one enemy layer.

Two more structures with the smaller True-One-Hot encoding are visualized in Figure 5.14. It can be seen that even with an extra air layer, the encoding does not solve the overlapping block problem. The second structure in Figure A.12d, while still containing a lot of noise and overlapping blocks, has more defied structural integrity and fewer floating blocks compared to the 40-layer True-One-Hot structures. More examples of this encoding are appended in Figure A.12.

It can also be seen that in the representation of Figure A.12c, a “coloured” pixel



(a) Structure 1 in Small One-True-Hot encoding with air

(b) Structure 1 decoded

(c) Structure 2 in Small One-True-Hot encoding with air

(d) Structure 2 decoded

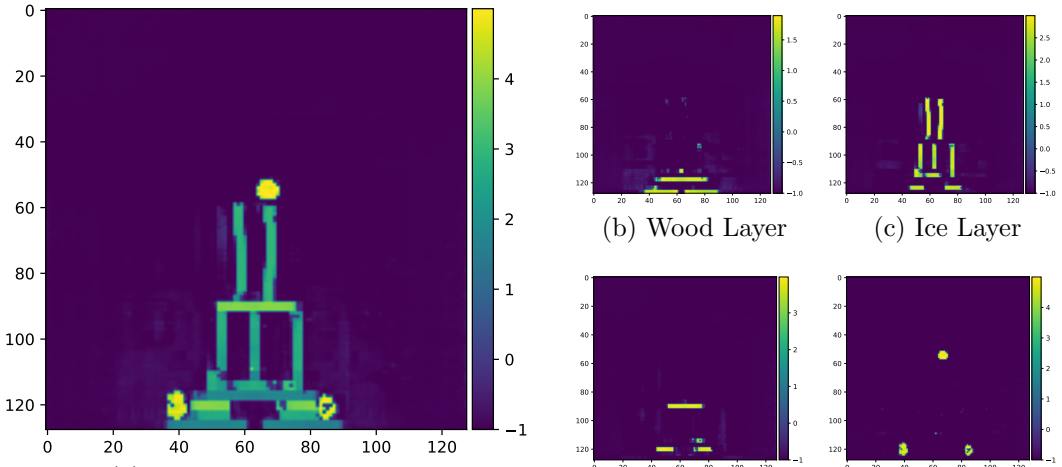
Figure 5.14: Using the small true one hot encoding with the extra air layer.

is usually right next to one another, which consequently results in an overlapping block. This phenomenon hints at a fundamental problem when using convolutional GANs and the One-Element representation, which is further discussed in Section 6.1.2. Another approach to fix this problem could be a more elaborate decoding algorithm, similar to the "confidence decoding", that selects the most confident block and removes every surrounding pixel that would result in an overlapping block.

5.1.4 Visual Multilayer Encoding

Combining the knowledge of the previous results, this training iteration uses the convolutional GAN trained over 15000 epochs with the WGAN algorithm. The encoding is the same calculated visual encoding as in Section 5.1.1.2 but in this iteration, not on a single layer but separated into the multilayer representation once with air and

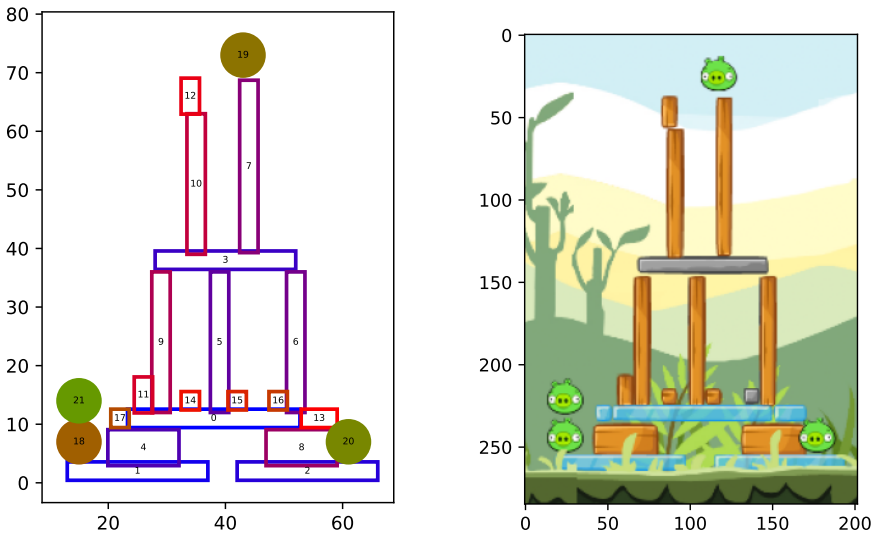
once without. Two versions have been trained once with an extra air layer and once without.



(a) Flattend Representation (b) Wood Layer (c) Ice Layer (d) Stone Layer (e) Enemy Layer

Figure 5.15: A structure in the multilayer visual representation and its layers.

Figure 5.15 shows the first more promising generated structure that resembles a more complex structure that seems to be stable. The separate layers in Figure 5.15b to 5.15e show that the blocks are well separated over the layers and don't interfere in most cases. Even so, there is noise on every layer, a threshold of 0.5 for the air removes most of it.

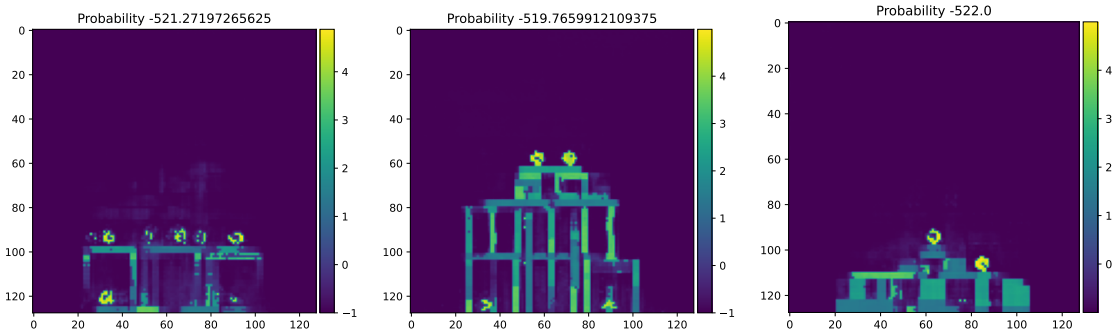


(a) Decoded Structure (b) Simulated

Figure 5.16: The generated structure of Figure 5.15 decoded using the confidence decoding and send to the simulation.

The structure of Figure 5.15 decoded with the confidence decoding is visible in Figure 5.16. The decoded structure simulated shows a stable structure that contains

structural blocks and added blocks with no structural purpose. While this approach seems promising, it still produces structures of low quality.

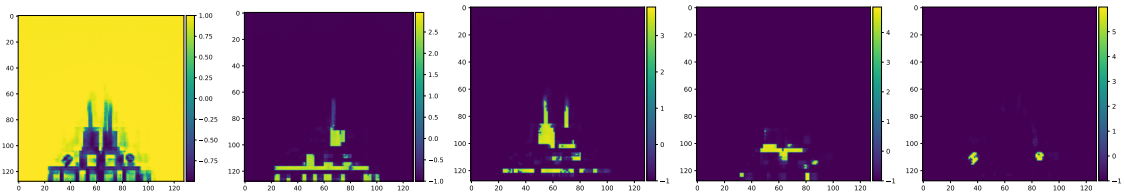


(a) Structure 2 (b) Structure 3 (c) Structure 4

Figure 5.17: Three structures in the multilayer visual representation with lower quality.

Three more generated structures are shown in Figure 5.17. It can be seen that the structures have a lower quality which is shown in the blurriness of the blocks. All of these structures are flattened using an air threshold of 0.5. Another fully visualized example of a structure generated using this encoding is appended in Figure A.13.

While the previous training iteration seems promising, the extra threshold parameter causes more problems. If the value is too low, a lot of noise is added to the structure, hindering the decoding process, while a value that is too high can add holes in the representation, also creating problems in the decoding process. In the previous One-Element encoding, adding the air layer helped the gan to better differentiate between air and block positioning, reducing the overall noise in the structure representation. The last training iteration uses the same setup but with the added air layer described in Section 3.1.4.



(a) Air Layer (b) Wood Layer (c) Ice Layer (d) Stone Layer (e) Enemy Layer

Figure 5.18: The layers of a structure in the representation that uses an explicit air layer.

All layers of a structure generated by this model are visualized in Figure 5.18. It can be seen that the Air Layer (5.18a) is the negative of the remaining layers that

create the structure. The highest value on each layer is used to recreate a single-layer representation. As this is the model used in the quantitative and qualitative search for stable structures, more examples of structures generated through this GAN are shown in Section 5.3.

5.2 Quantitative Evalutation Results

The quantitative evaluation, as previously explained, is done on the convolutional GAN with the calculated visual encoding in the multilayer version with air. First, the results of the grid search are displayed, followed by using the found parameters to search for structures with different characteristics.

5.2.1 Grid Search Results

The grid search results are the combination of the decoded structure's metadata and their simulation results. There are two main ways to evaluate the data of the search. The first one searches for the parameter set that resulted in the desired structure characteristic, and the second compares the effect when changing a parameter over all of the accumulated data.

5.2.1.1 Characteristic Search

The searched parameter set are the ones that result in (1) the lowest block damage, which indicates stable structures, (2) the percentage of stable structures calculated by the simulation, (3) the number of destroyed blocks when simulated, (4) the overall amount of blocks and (5) the average highest structures height and (6) width. The resulting parameters for each characteristic are given in Table A.3.

The search characteristics that are visualized in Figure 5.19 aim at the stability of the structures. Blue is the one that minimized the received damage, red is the stable percentage, and green is the number of destroyed blocks. All parameter sets share that they use the **custom kernel scale**. As a reminder, the custom kernel scale influences the probability that a rectangular block is selected. This evidently influences the stability of the generated structure positively, as a long horizontal block adds more support than a narrow vertical one. Another shared parameter is the **negative air** value of -1. The parameter set that maximises the stability of the structure is

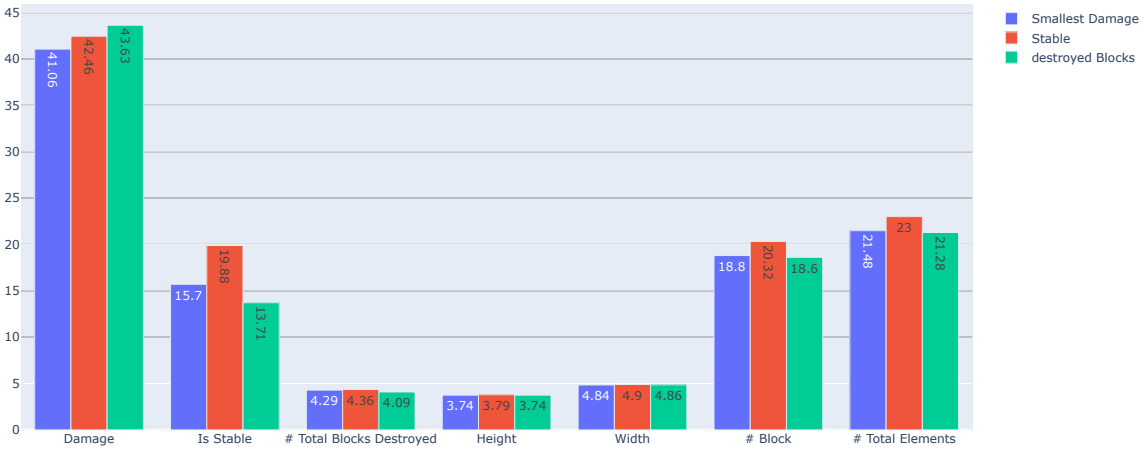


Figure 5.19: Collected data of the parameter sets that optimize for the aforementioned data (1), (2).

almost equal to the one that minimizes the received damage. Only the **Cutoff Point** is 0.5 compared to the 0.8 of the smallest damage.

The parameter set that minimizes the total amount of destroyed blocks doesn't necessarily correlate with the stability of the structure. The accumulated damage of this set is the highest, and the percentage of stable structures is the lowest. This is supported by the fact that it has the lowest amount of blocks compared to the other sets.

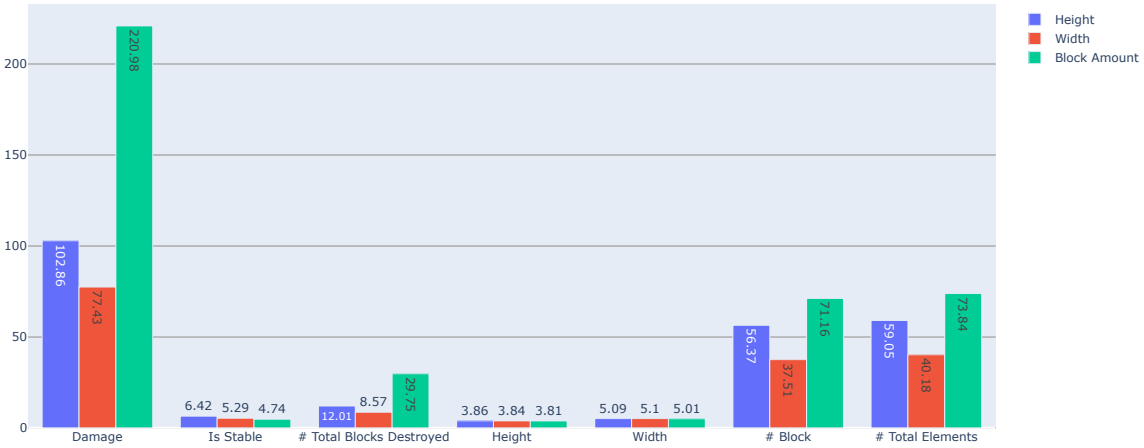


Figure 5.20: Each colour represents the parameter set that optimizes one of the aforementioned characteristics.

The effect of optimizing towards specific structure characteristics is visualized in Figure 5.20. The optimized characteristics are the structure dimensions and the total block amount of the structure. Most notable is that with more blocks, the average amount of damage and destroyed blocks rises significantly. This effect is explained later when looking at the "combine layers" parameter. Also relatively noticeable

is that the parameter set that optimizes for width uses the **custom kernel scale** compared to the set that optimizes for height which results in the less destroyed blocks.

5.2.1.2 Parameter Compare

The second approach averages all collected data of all parameter sets with the respective parameter set to the respective value. The evaluation application visualized in the appended Figure A.14 provides the feature to search for the structure that maximises the difference of a selected data point when it is decoded with the respective parameter turned on or off.

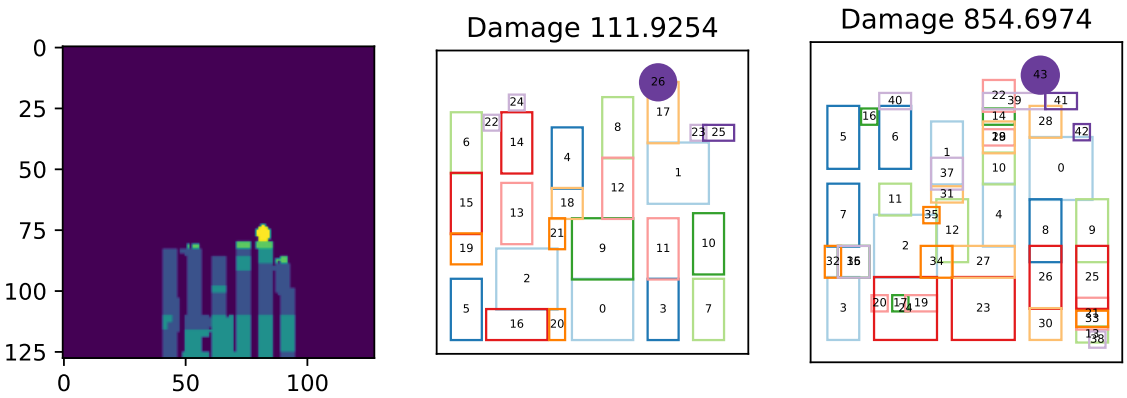


Figure 5.21: The structure that maximises the damage difference when the **combine layers** parameter is turned **on** (left) and when the **combine layers** parameter is turn **off** (right).

For example, Figure 5.21 visualizes the structure that maximises the damage when the **combine layer** parameter is turned on and when it is turned off. It becomes clear that there are structures that have overlapping blocks in different layers, which result in a lot of damage when the layers are decoded independently. Reducing the layers into one layer results in a more stable structure as the decoding algorithm prohibits overlapping blocks that result in complications.

Averaging the data of all tested structures, displayed in Figure 5.22, also shows that the flattened decoding process produces more stable structures than decoding each layer independently. This also means that the uncertainty of the GAN has to be compensated through the decoding process.

The previous assumption that the "custom kernel scale" positively influences the stability of a structure is supported by reviewing the direct comparison in Figure 5.23. Less damage has been done, and a higher percentage of structures is stable.

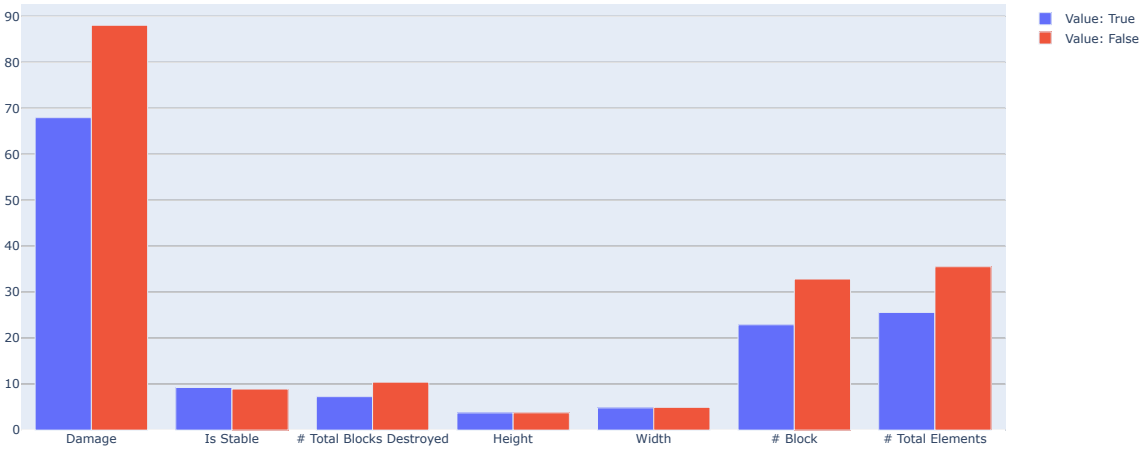


Figure 5.22: Graph that compares the data with both states of the “combine layers” parameter.

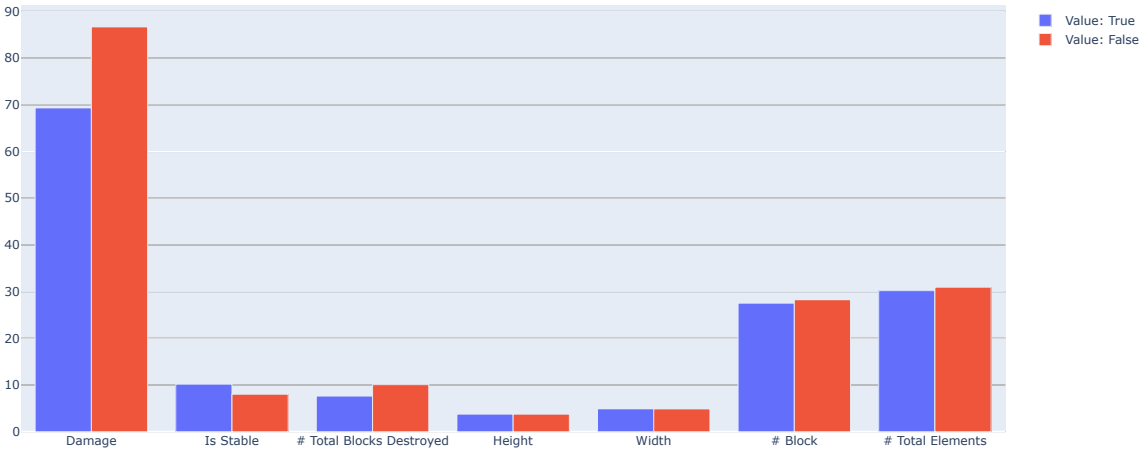


Figure 5.23: Graph that compares the data with both states of the “custom kernel scale” parameter.

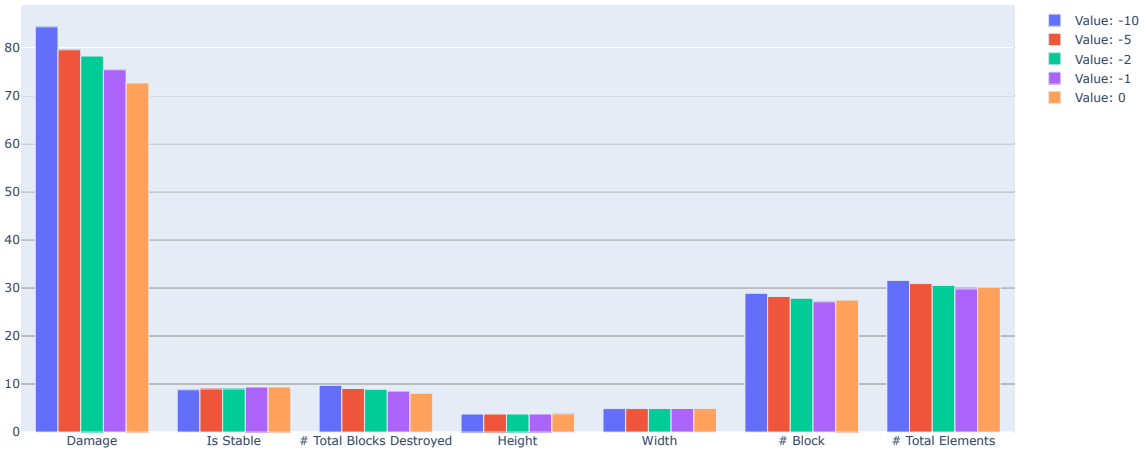


Figure 5.24: Graph that compares the data created with the different states of the “negative air value” parameters.

Reviewing the effect that different values of negative air have in Figure 5.24 shows that having no negative air results in less damage. As shown in the section where the individual parameter was introduced, having negative air results in more decoded blocks. Also, no negative air and a negative air of -1 resulted in equal amounts of stable structures.

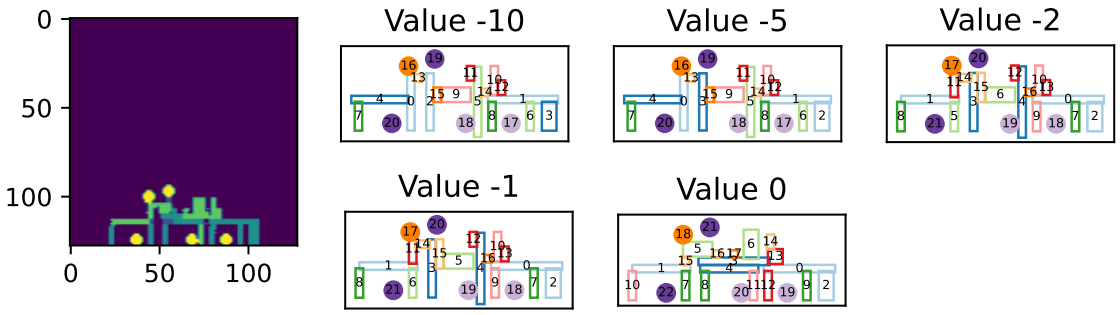


Figure 5.25: A structure decoded with different negative air values.

Having a higher negative "air value" means that bigger blocks that would not cover the contour as well as smaller blocks, are less likely to be chosen. In Figuer 5.25, it can be seen that the decoding with a "air value" of 0 uses a big block to bridge the gap resulting in a more stable structure compared to the decodings where better fitting blocks were selected.

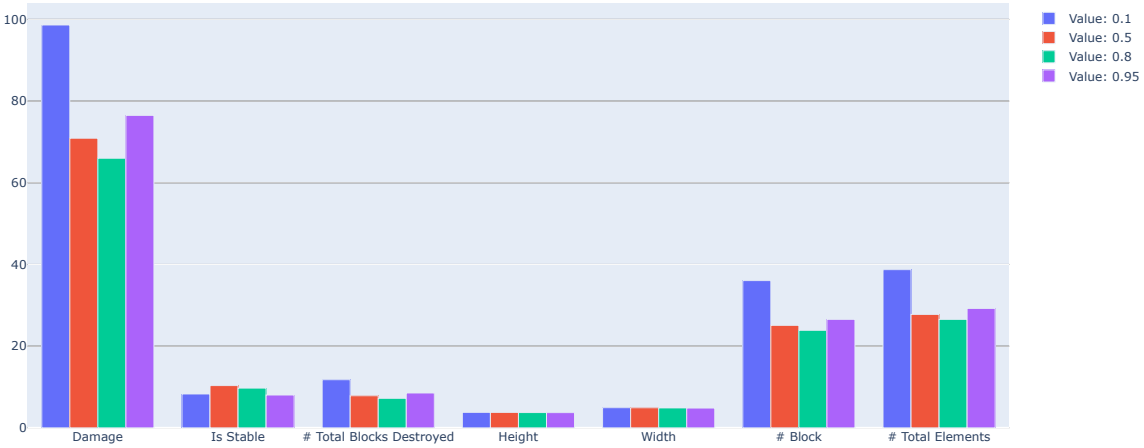


Figure 5.26: Graph that compares the data created with the different states of the "cut of point" parameters.

Comparing the different cutoff values visualized in Figure 5.26 shows that the "cutoff value" of 0.1 leaves more blocks but at points that do not help with the stability of the structures. Removing too many possible block positions, as a high value of 0.95 does, also resulted in more damage and less stable structures. While a parameter set with a cutoff value of 0.8 resulted in the most stable levels, overall other

parameter sets, the parameter sets with 0.5, had more stable levels, even so, they had more damage.

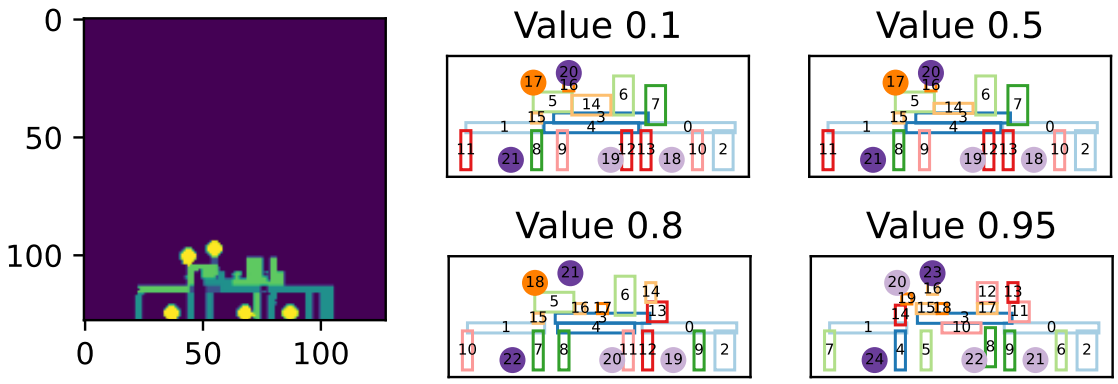


Figure 5.27: A structure decoded with different cutoff points.

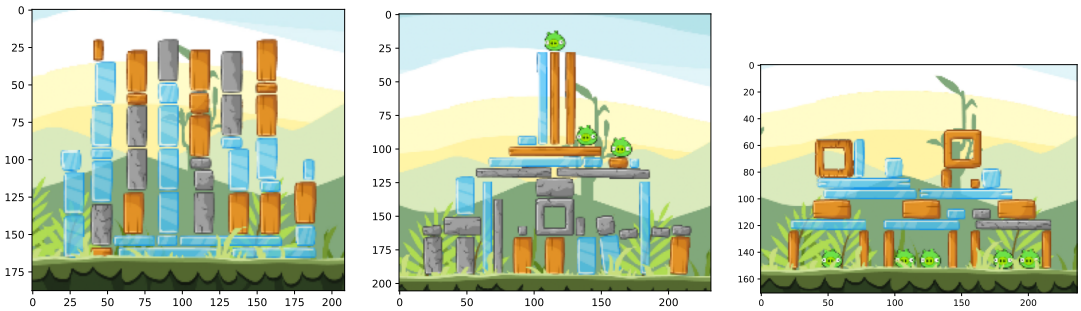
Almost the same argument, why a smaller **negative air** parameter has a positive influence on the stability, can be made by the different cutoff points. A strict cutoff at high values removes the probability of bigger blocks. In Figure A.16, it can be seen that the value of 0.95 results in many small blocks that are unsupported. Moving down towards lower cutoff values introduces more supporting blocks.

The results of the last two parameters, the **round to next integer** and the use of a **minus-one border**, are attached. Rounding to the next integer produced more stable structures while using a minus-one border in less stable structures.

5.3 Quality Search

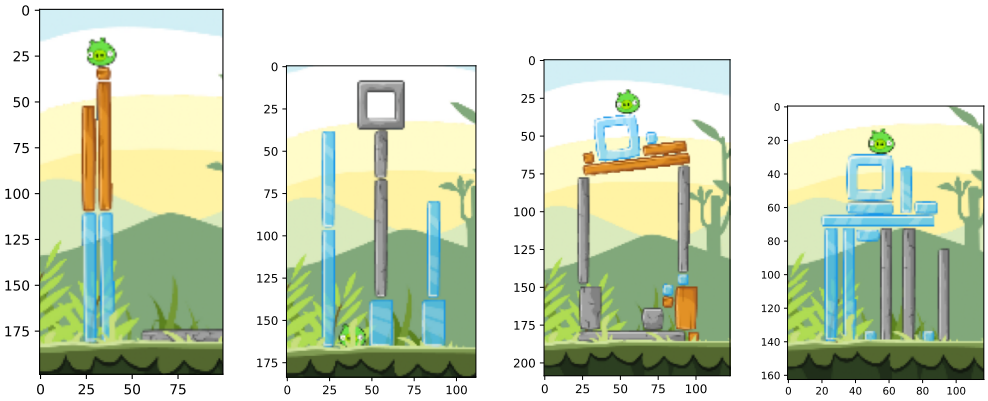
This section visualizes various generated, stable structures and reviews some accumulated data over the generated structures. Overall, 4400 structures are generated and simulated in this experiment. The parameter set that has been used is the one that maximizes the stability of the structures. Of the 4400 structures, only 455 are stable, as detected by the velocity method in the simulation, while 3934 are unstable.

The first set of generated structures, visualized in Figure 5.28, are stable based on the simulation data and are the ones with the most used blocks. It was not filtered whether the structure contains any enemies, resulting in



(a) Structure 1 (b) Structure 2 (c) Structure 3

Figure 5.28: Generated structures based on maximising the block amount.



(a) Small Tower 1 (b) Small Tower 2 (c) Small Tower 3 (d) Small Tower 6

Figure 5.29: Generated structures based on maximising the structure height while minimising the structure width.

the structure of Figure 5.28a, with no enemies and would therefore not be playable.

The second set of generated structures, visualized in Figure A.18, are also stable and maximize the height of the structure while minimizing the width, which results in tower-like structures. Different structures optimized for various parameters are appended in Section A.6.2.

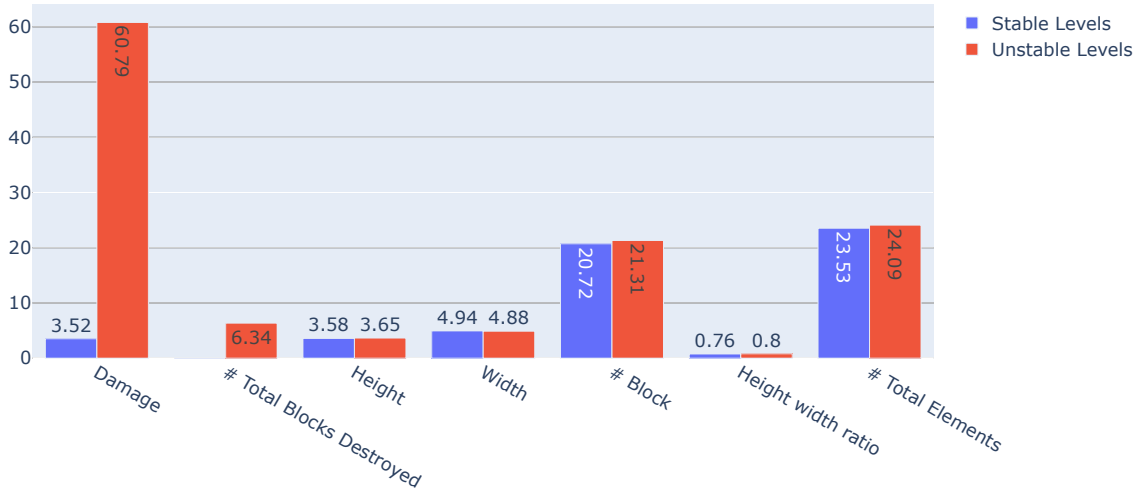


Figure 5.30: Comparing the collected structure data of stable structures vs unstable structures.

When comparing stable structures versus unstable structures, visualized in Figure 5.30, it can be seen that even the stable structures received a little bit of damage, but no blocks were destroyed. One could assume that most stable structures are shallower, such as the structures in Figure A.17, compared to the unstable structures, which is evidently only very little the case. The unstable structures are only a little higher and more narrow than the stable structures. This can also be seen in the height/width ratio, which shows that most stable structures are wider.

6 Discussion & Conclusion

In this chapter, the individual design decisions and how suitable they were towards generating stable structures are reviewed in the discussion section 6.1. Followed by summarising the work of this thesis in the conclusion 6.2 section with an outlook on possible future work which could extend the generating capabilities.

6.1 Discussion

From the first step of generating a structured data set to evaluate the last step of decoding a generated structure representation, many design decisions have to be made. We have seen in the results section that some resulted in a better working structure generation while others produced big block piles. The four most crucial parts are the GAN architecture and its training algorithm, the structure encoding and the decoding algorithm.

First, a brief overview of the capabilities and difficulties in the selection of GAN architectures and the training algorithm is given in this section. Then a review of the implemented encoding and decoding methods and their advantages and disadvantages with insight into why the One-element encoding yielded no results is given. Finally, a brief discussion of the implemented decoding algorithms is given.

6.1.1 GAN architectures

We have seen that different GAN architectures in combination with the training algorithm resulted in significantly different results. Training the shallower Simple-GAN architectures, even though they have more trainable parameters compared to the deeper Convolutional-GANs, resulted in no usable structure generator.

As mentioned in Section 2.3.2, a major challenge of GAN training is the possibility of mode collapse, which both models that were trained using Goodfellow et al. (2014) training algorithm did. On the other hand, only switching to the state-of-the-art WGAN-GP training algorithm, without changing the GANs architecture, produced no good results. The diversity of the produced representations went up while the quality respective quality went down.

It is also the case that the choice of the encoding method, or to be more precise, the data representation itself, should influence the architecture of the GANs generator and training algorithm. This can be seen in every training iteration that used the One-Element encoding and is further discussed in the next Section 6.1.2.

With all these factors that need to be considered, the fact that there are two neural networks that require a different composition of layers and activation functions and the overall difficulty of training the two networks make GAN a difficult choice for content generation. This is enhanced by the content's reliance on a perfect representation. Any small imperfection, for example, overlapping blocks in different layers with the Multilayer-Visual encoding or too closely positioned blocks in any One-Element encoding, results in an unstable level.

Another factor that comes on top of this is that training a GAN requires a lot of hardware resources as they rely on a small learning rate (Radford et al. 2015) and that the generator is trained through the discriminator indirectly. This limits the number of doable training iterations that can be done in a given time frame. Only the access to the high-performance computing cluster enabled the training of the 11 different models.

6.1.2 Encoding

All introduced encoding methods, depending on the used grid size, can represent a stable level. They differ functionally only in their capability to work as a training base to be recreated by the GAN. On the one hand, the encoding methods can be grouped into the usage of different value ranges and, on the other hand, how the information is encoded.

In the first group, the different value ranges refer to either using the One-Hot value range of $[0, ..., 1]$, which represents either on or off, or an arbitrary value range in the Single-Layer encoding, in which every different value represents a different kind of information. Using the One-Hot encoding, when it comes to encoding information which needs to be persistent and reliable in its local vicinity, is more practical, as it

can be interpreted as a confidence value. A higher value means more confidence in the result of the model, while in the Single-Layer approach, a higher value probably means different information. This is most important in the decoding process, where consistent data is required.

The other differentiation, how the information is encoded, aims at the Visual and One-Element encoding types. It has been shown that CNN are good at image processing due to their positional invariance and other capabilities to learn abstract structural information. When using the One-Element encoding, a transposed convolutional filter had to learn the task of placing a high-value pixel into a position that is surrounded by only zeros. With the way that transposed convolutional filters work, this task seems to be difficult. Adding the fact that the DCGAN uses only two-strided transposed convolutional layers means the layer had to interpolate between pixels and isolate a single pixel in the same step.

With the large amount of zero space in the One-Element representations, it could be the case that more filters learned to remove the values. This, and the previous problem, would explain why models that trained on the One-Element encodings that did not use the One-Hot encodings mainly produced blocks in the first material range and only rarely any higher values. Adding the previously made observation that the models that used the One-Element/One-Hot encoding mostly had high confidence values right next to one another supports the claim that transposed convolution cant isolate and interpolate pixels in the same layer.

6.1.3 Decoding

As previously discussed, encoding all information into one layer results in problems with noise and confidence problems. Using multiple layers and decoding them independently evidently has its own problems, as shown in the grid search results in Section 5.2.1.2. The encoded pixels overlap in different layers and interfere with one another in the decoding process. Adding more correction steps into the decoding process can be seen as a tradeoff between the direct expressivity of the model and a more stable structure.

The two implemented decoding algorithms have both their advantages and disadvantages. In its raw form, the Recursive-Rectangle algorithm assumes a perfect structure representation which is almost completely unusable with the noisy struc-ture representation of the GANs. While the Confidence-Decoding greedily selects blocks linearly that fit into the contour of the level representation. It has been shown that this approach only produces the encoded level representation with a

very specific set of parameters that are not applicable to the generated representations.

Again a combination of both algorithms could improve the overall stability. The "recursive block selection" that ensures the given encoded silhouette is filled completely in combination with the confidence-based block positions, which can, given a noisy encoding, find plausible possible block positions.

6.2 Conclusion

Concluding the thesis with a brief summary and thoughts on the main research goals of this thesis, whether GANs can be utilized for stable structure generation in a physics-based simulation. Reviewing the research questions and answering them directly and ending on an outlook of what future research on this topic could include.

Initially, an overview of different techniques in procedural content generation, in particular the ones that use machine learning, is given in the related research chapter. As this thesis focused on content generation utilizing the GAN networks, the first section explains how GANs function with their initial training process, their difficulties and how the state-of-the-art training algorithms work and circumvent previous problems. The second related research section focuses on the content generation that uses GANs and the algorithm that generate structures in the same science birds domain with different approaches.

The main contribution of this thesis is the developed encoding and decoding methods for the difficult domain of real-valued science bird structures. When combining the two different Visual encodings and the One-Element encoding with the different Multilayer representations, a total of 10 distinct encodings can be created. Further, two decoding algorithms for the ambiguous visual encoding have been introduced, with both having their respective advantages and disadvantages. Lastly, the proposed training framework can combine different GAN architectures with different training algorithms.

In order to train 11 GAN networks, various datasets have been created with different characteristics and using a structure filtering process. The open-source science bird implementation had to be extended to allow for fast structure evaluation and data collection required in the parameter search for the decoding algorithm and the quality-based search for stable structures. Also, an application was implemented to interact with the GAN models and decoding algorithms to test various edge cases.

In the result Section, it has been shown that a careful selection of GAN architecture, training algorithm, encoding representation and decoding parameters resulted in a generator capable of creating stable and diverse structures. The decoding parameters have been tested using a grid search, and the results provided insight into the advantages of the different representations. Even though the generator is able to generate stable structures, it mostly generates unstable structures that fall over when put into the simulation. This is due to errors in the decoding algorithm or noise in the generated encoding.

Overall using GANs to generate structures that require the perfect positioning of blocks to be stable is doable but difficult. In order to see the results of a minute change requires a full training iteration, and the associated time and computing capacity, combined with the number of parameters distributed over the whole process, lengthens the development process using GANs.

6.2.1 Future Work

The latent space of the generator could be further analysed either through the aforementioned LVE, similarly to Volz et al. (2018) work, or by incorporating the discriminator's output into the structure selection process. While the output of the discriminator is, in most cases, not usable without a reference frame. This could be created by reviewing the stability properties of generated structures in combination with the discriminator's output.

Regarding the implemented encodings, various aspects of how information can be represented could be done differently. With the difficulty of the One-Element encoding to isolate single pixels, using more pixels would presumably already help with this problem. A combination of the One-Hot encoding with the Visual-encodings, where multiple pixels represent a block, removes any uncertainty about which block is placed in any given location. Using multiple pixels to represent a block would still require a proper block selection in the decoding algorithm that combines pixels belonging to the same block.

Instead of appending on the rasterized, image synthesis-based encodings, completely different data representations could be investigated. As discussed in the different GAN applications, different data structures can be generated, even though image synthesis is the most researched. This means different encodings could be investigated that use other data structures that work better with the real-spaced positioning compared to the difficulty of rasterising in the proposed visual and One-Element encodings.

Lastly, because GAN work with every kind of NN, future research can be done with every application that uses deep neural networks. Various different compositions of, for example, convolutional layers, activations, normalising layers or the use of residual connections in both the generator and discriminator, can be tested. Further, the use of entirely different GAN architecture, for example, the Conditional GAN, that uses an extra parameter to gain control over the GAN output, could be evaluated.

List of Figures

List of Tables

Bibliography

Amato, Alba. 2017. "Procedural Content Generation in the Game Industry." In *Game Dynamics: Best Practices in Procedural and Dynamic Game Content Generation,* edited by Oliver Korn and Newton Lee, 15–25. Cham: Springer International Publishing. ISBN: 978-3-319-53088-8. https://doi.org/10.1007/978-3-319-53088-8_2.

Arjovsky, Martin, and Léon Bottou. 2017. "Towards Principled Methods for Training Generative Adversarial Networks." *arXiv.org,* https://doi.org/10.48550/arXiv.1701.04862.

Arjovsky, Martin, et al. 2017. *Wasserstein GAN.* https://doi.org/10.48550/ARXIV.1701.07875.

Ba, Jimmy Lei, et al. 2016. *Layer Normalization.* https://doi.org/10.48550/ARXIV.1607.06450.

Basodi, Sunitha, et al. 2020. "Gradient amplification: An efficient way to train deep neural networks." *Big Data Mining and Analytics* 3 (3): 196–207. https://doi.org/10.26599/BDMA.2020.9020004.

Bontrager, Philip, et al. 2017. *DeepMasterPrints: Generating MasterPrints for Dictionary Attacks via Latent Variable Evolution.* https://doi.org/10.48550/ARXIV.1705.07386.

Bradski, G. 2000. "The OpenCV Library." *Dr. Dobb's Journal of Software Tools.*

Doull, Andrew. 2015. "Procedural Content Generation Wiki Beneath Apple Manor." Accessed August 9, 2022. http://pcg.wikidot.com/pcg-games:beneath-apple-manor.

———. 2016. "Procedural Content Generation Wiki Rogue." Accessed August 9, 2022. http://pcg.wikidot.com/pcg-games:rogue.

Ferreira, Lucas, and Claudio Toledo. 2014. "A Search-based Approach for Generating Angry Birds Levels." In *Proceedings of the 9th IEEE International Conference on Computational Intelligence in Games.* CIG'14. Dortmund, Germany.

Fontaine, Matthew C, et al. 2020. "Illuminating Mario Scenes in the Latent Space of a Generative Adversarial Network." *arXiv.org,* https://doi.org/10.48550/arXiv.2007.05674.

Giacomello, Edoardo, et al. 2018. *DOOM Level Generation using Generative Adversarial Networks.* https://doi.org/10.48550/ARXIV.1804.09154.

Goodfellow, Ian. 2017. *NIPS 2016 Tutorial: Generative Adversarial Networks.* https://doi.org/10.48550/ARXIV.1701.00160.

Goodfellow, Ian, et al. 2014. "Generative Adversarial Nets." In *Advances in Neural Information Processing Systems,* edited by Z. Ghahramani et al., vol. 27. Curran Associates, Inc.

Goodfellow, Ian, et al. 2016. *Deep Learning.* Http://www.deeplearningbook.org. MIT Press.

Gui, Jie, et al. 2021. "A Review on Generative Adversarial Networks: Algorithms, Theory, and Applications." *IEEE Transactions on Knowledge and Data Engineering,* 1–1. https://doi.org/10.1109/tkde.2021.3130191.

Gulrajani, Ishaan, et al. 2017. "Improved Training of Wasserstein GANs." *CoRR* abs/1704.00028.

Hinton, Geoffrey, et al. n.d. "Neural Networks for Machine Learning Lecture 6a Overview of mini–batch gradient descent."

Hochreiter, Sepp, and Jürgen Schmidhuber. 1997. "Long Short-term Memory." *Neural computation* 9 (December): 1735–80. https://doi.org/10.1162/neco.1997.9.8.1735.

Huang, He, et al. 2018. *An Introduction to Image Synthesis with Generative Adversarial Nets.* https://doi.org/10.48550/ARXIV.1803.04469.

Hui, Jonathan. 2018. *GAN — Wasserstein GAN & WGAN-GP - Jonathan Hui - Medium,* June.

Ioffe, Sergey, and Christian Szegedy. 2015. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.* https://doi.org/10.48550/ARXIV.1502.03167.

Jabbar, Abdul, et al. 2020. "A Survey on Generative Adversarial Networks: Variants, Applications, and Training." *arXiv.org,* https://doi.org/10.48550/arXiv.2006.05132.

Karras, Tero, et al. 2017. *Progressive Growing of GANs for Improved Quality, Stability, and Variation.* https://doi.org/10.48550/ARXIV.1710.10196.

Karras, Tero, et al. 2018. "A Style-Based Generator Architecture for Generative Adversarial Networks." *CoRR* abs/1812.04948.

Kingma, Diederik P, and Max Welling. 2013. *Auto-Encoding Variational Bayes.* https://doi.org/10.48550/ARXIV.1312.6114.

Kingma, Diederik P., and Jimmy Ba. 2014. *Adam: A Method for Stochastic Optimization.* https://doi.org/10.48550/ARXIV.1412.6980.

LeCun, Yann. 2016. "What are some recent and potentially upcoming breakthroughs in deep learning?" Accessed August 13, 2022. https://www.quora.com/What-are-some-recent-and-potentially-upcoming-breakthroughs-in-deep-learning.

Liu, Jialin, et al. 2020. "Deep learning for procedural content generation." *Neural Computing and Applications* 33, no. 1 (October): 19–37. https://doi.org/10.1007/s00521-020-05383-8.

Liu, Ming-Yu, and Oncel Tuzel. 2016. *Coupled Generative Adversarial Networks.* https://doi.org/10.48550/ARXIV.1606.07536.

Mirza, Mehdi, and Simon Osindero. 2014. *Conditional Generative Adversarial Nets.* https://doi.org/10.48550/ARXIV.1411.1784.

Moghadam, Monireh Mohebbi, et al. 2021. "Game of GANs: Game-Theoretical Models for Generative Adversarial Networks." *arXiv.org,* https://doi.org/10.48550/arXiv.2106.06976.

Nair, Vinod, and Geoffrey E. Hinton. 2010. "Rectified Linear Units Improve Restricted Boltzmann Machines." In *Proceedings of the 27th International Conference on International Conference on Machine Learning,* 807–814. ICML'10. Haifa, Israel: Omnipress. ISBN: 9781605589077.

Radford, Alec, et al. 2015. *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks.* https://doi.org/10.48550/ARXIV.1511.06434.

Ramesh, Aditya, et al. 2022. *Hierarchical Text-Conditional Image Generation with CLIP Latents.* https://doi.org/10.48550/ARXIV.2204.06125.

Robbins, Herbert E. 2007. "A Stochastic Approximation Method." *Annals of Mathematical Statistics* 22:400–407.

Salimans, Tim, et al. 2016. *Improved Techniques for Training GANs.* https://doi.org/10.48550/ARXIV.1606.03498.

Saxena, Divya, and Jiannong Cao. 2020. "Generative Adversarial Networks (GANs): Challenges, Solutions, and Future Directions." *CoRR* abs/2005.00065.

Schlegl, Thomas, et al. 2017. *Unsupervised Anomaly Detection with Generative Adversarial Networks to Guide Marker Discovery.* https://doi.org/10.48550/ARXIV.1703.05921.

Schmidhuber, Jürgen. 2015. "Deep learning in neural networks: An overview." *Neural Networks* 61 (January): 85–117. https://doi.org/10.1016/j.neunet.2014.09.003.

Silver, David, et al. 2016. "Mastering the game of Go with deep neural networks and tree search." *Nature* 529, no. 7587 (January): 484–489. https://doi.org/10.1038/nature16961.

Smelik, Ruben M., et al. 2014. "A Survey on Procedural Modelling for Virtual Worlds." *Comput. Graph. Forum* (Chichester, GBR) 33, no. 6 (September): 31–50. ISSN: 0167-7055. https://doi.org/10.1111/cgf.12276.

Soille, Pierre. 2004. "Erosion and Dilation." In *Morphological Image Analysis,* 63–103. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-662-05088-0_3.

Stephenson, Matthew, and Jochen Renz. 2016a. "Procedural generation of complex stable structures for angry birds levels." In *2016 IEEE Conference on Computational Intelligence and Games (CIG),* 1–8. https://doi.org/10.1109/CIG.2016.7860410.

———. 2016b. "Procedural Generation of Levels for Angry Birds Style Physics Games." In *Proceedings of the Twelfth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment.* AIIDE'16. Burlingame, California, USA: AAAI Press. ISBN: 978-1-57735-772-8.

———. 2017. "Generating varied, stable and solvable levels for angry birds style physics games." In *2017 IEEE Conference on Computational Intelligence and Games (CIG),* 288–295. https://doi.org/10.1109/CIG.2017.8080448.

Stephenson, Matthew, et al. 2019. "The 2017 AIBIRDS Level Generation Competition." *IEEE Transactions on Games* 11 (3): 275–284. https://doi.org/10.1109/TG.2018.2854896.

Summerville, Adam, and Michael Mateas. 2016. *Super Mario as a String: Platformer Level Generation Via LSTMs.* https://doi.org/10.48550/ARXIV.1603.00930.

Summerville, Adam, et al. 2017. "Procedural Content Generation via Machine Learning (PCGML)." *arXiv.org,* https://doi.org/10.48550/arXiv.1702.00539.

Summerville, Adam James, et al. 2016. *The VGLC: The Video Game Level Corpus.* https://doi.org/10.48550/ARXIV.1606.07487.

Suzuki, Satoshi, and KeiichiA be. 1985. "Topological structural analysis of digitized binary images by border following." *Computer Vision, Graphics, and Image Processing* 30 (1): 32–46. ISSN: 0734-189X. https://doi.org/https://doi.org/10.1016/0734-189X(85)90016-7.

Tanabe, Takumi, et al. 2021. "Level Generation for Angry Birds with Sequential VAE and Latent Variable Evolution." In *Proceedings of the Genetic and Evolutionary Computation Conference,* 1052–1060. GECCO '21. Lille, France: Association for Computing Machinery. ISBN: 9781450383509. https://doi.org/10.1145/3449639.3459290.

Tao, Chenyang, et al. 2018. "Chi-square Generative Adversarial Network." In *Proceedings of the 35th International Conference on Machine Learning,* edited by Jennifer Dy and Andreas Krause, 80:4887–4896. Proceedings of Machine Learning Research. PMLR, June.

Togelius, Julian, et al. 2013. "The Mario AI Championship 2009-2012." *AI Magazine* 34, no. 3 (September): 89–92. https://doi.org/10.1609/aimag.v34i3.2492.

Tran, Ngoc-Trung, et al. 2018. *Dist-GAN: An Improved GAN using Distance Constraints.* https://doi.org/10.48550/ARXIV.1803.08887.

Volz, Vanessa, et al. 2018. "Evolving Mario Levels in the Latent Space of a Deep Convolutional Generative Adversarial Network." In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2018).* Kyoto, Japan: ACM, July. https://doi.org/10.1145/3205455.3205517.

Wang, Hongwei, et al. 2017. *GraphGAN: Graph Representation Learning with Generative Adversarial Nets.* https://doi.org/10.48550/ARXIV.1711.08267.

Wang, Yaxing, et al. 2016. "Ensembles of Generative Adversarial Networks." *CoRR* abs/1612.00991.

Xia, Bin, et al. 2020. "LogGAN: a Log-level Generative Adversarial Network for Anomaly Detection using Permutation Event Modeling." *Information Systems Frontiers* 23, no. 2 (June): 285–298. https://doi.org/10.1007/s10796-020-10026-3.

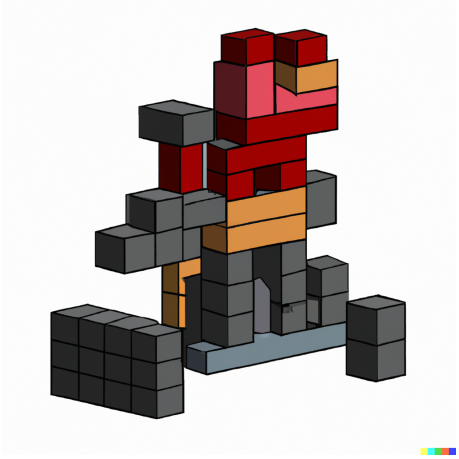
Xu, Bing, et al. 2015. *Empirical Evaluation of Rectified Activations in Convolutional Network.* https://doi.org/10.48550/ARXIV.1505.00853.

Yannakakis, G. N., and J. Togelius. 2011. "Experience-Driven Procedural Content Generation." *IEEE Transactions on Affective Computing* 2, no. 3 (July): 147–161. https://doi.org/10.1109/t-affc.2011.6.

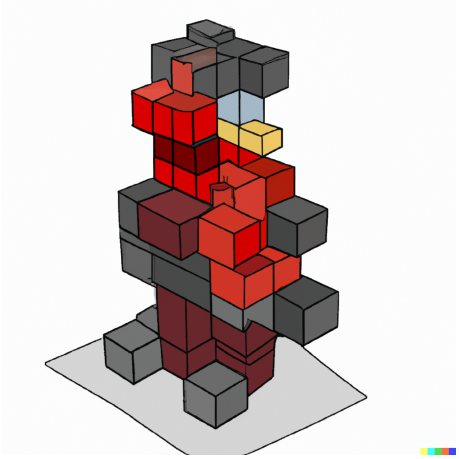
Yu, Lantao, et al. 2016. *SeqGAN: Sequence Generative Adversarial Nets with Policy Gradient.* https://doi.org/10.48550/ARXIV.1609.05473.

A Appendix

A.1 Introduction



(a) A simple two dimensional block structure, as seen from the side, that's structurally stable cartoon



(b) Variation of (a)

(c) A render of burger as a planet with onion rings as ring in space infront of a galaxy with cats

(d) A foto of a cut girl standing in a flower field looking at a stary night sky, abstract art

Figure A.1: Art generated by me using the DALL-E 2 (Ramesh et al. 2022) model.

Figure A.2: Generated Art created using Midjourney[1]by Jason M. Allen that won the Colorado State Fair Fine Art competition in the digital arts category.

A.1.1 Sciencebirds

Regular Blocks		
1		SquareHole
2		RectBig
3		RectMedium
4		RectSmall
5		RectFat
6		RectTiny
7		SquareTiny
8		SquareSmall
Irregular Blocks		
9		CircleSmall
10		Triangle
11		TriangleHole
12		Circle

Table A.1: Table that lists each science bird block and their names.

1. Midjourney AI art generation

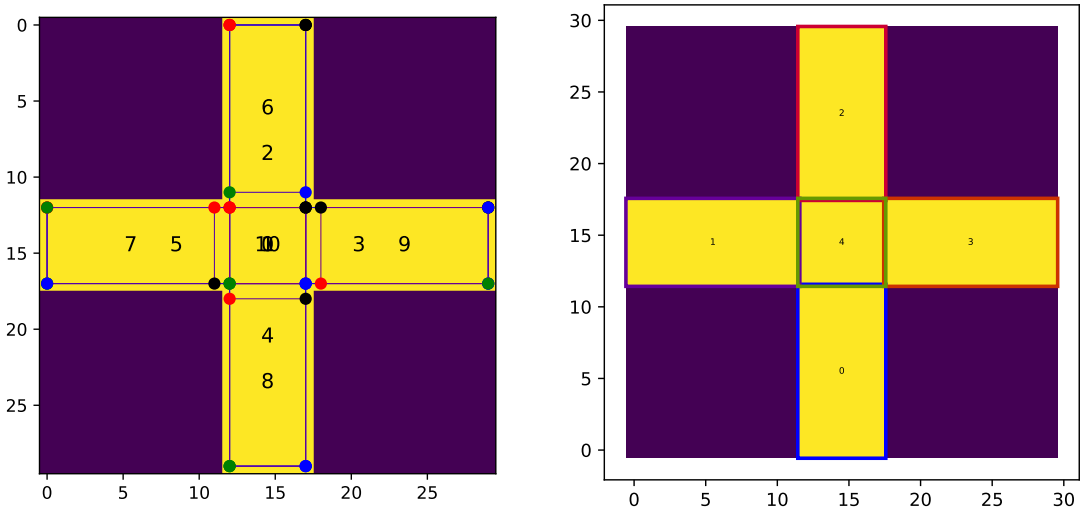
A.2 Resolution table

Table A.2: The table shows different grid sizes and the quotient of each distinct block dimension divided by the size. The max / average remainder is the respective value of the numbers behind the dot. The row with a grid size of 0.07 shows that the divisions are closest to the lower integer, indicated by a good average and maximum.

									Remainder	
Width	SquareHole	RectFat	SquareSmall	SquareTiny	RectTiny	RectSmall	RectMedium	RectBig	Max	Avg
0.01	85.0, 85.0	85.0, 43.0	43.0, 43.0	22.0, 22.0	43.0, 22.0	85.0, 22.0	168.0, 22.0	206.0, 22.0	0	0
0.02	42.5, 42.5	42.5, 21.5	21.5, 21.5	11.0, 11.0	21.5, 11.0	42.5, 11.0	84.0, 11.0	103.0, 11.0	0.5	0.16
0.03	28.33, 28.33	28.33, 14.33	14.33, 14.33	7.33, 7.33	14.33, 7.33	28.33, 7.33	56.0, 7.33	68.67, 7.33	0.67	0.17
0.04	21.25, 21.25	21.25, 10.75	10.75, 10.75	5.5, 5.5	10.75, 5.5	21.25, 5.5	42.0, 5.5	51.5, 5.5	0.75	0.2
0.05	17.0, 17.0	17.0, 8.6	8.6, 8.6	4.4, 4.4	8.6, 4.4	17.0, 4.4	33.6, 4.4	41.2, 4.4	0.6	0.15
0.06	14.17, 14.17	14.17, 7.17	7.17, 7.17	3.67, 3.67	7.17, 3.67	14.17, 3.67	28.0, 3.67	34.33, 3.67	0.67	0.12
0.07	12.14, 12.14	12.14, 6.14	6.14, 6.14	3.14, 3.14	6.14, 3.14	12.14, 3.14	24.0, 3.14	29.43, 3.14	**0.43**	**0.08**
0.08	10.62, 10.62	10.62, 5.38	5.38, 5.38	2.75, 2.75	5.38, 2.75	10.62, 2.75	21.0, 2.75	25.75, 2.75	0.75	0.26
0.09	9.44, 9.44	9.44, 4.78	4.78, 4.78	2.44, 2.44	4.78, 2.44	9.44, 2.44	18.67, 2.44	22.89, 2.44	0.89	0.31
0.1	8.5, 8.5	8.5, 4.3	4.3, 4.3	2.2, 2.2	4.3, 2.2	8.5, 2.2	16.8, 2.2	20.6, 2.2	0.8	0.23
0.11	7.73, 7.73	7.73, 3.91	3.91, 3.91	2.0, 2.0	3.91, 2.0	7.73, 2.0	15.27, 2.0	18.73, 2.0	0.91	0.31
0.12	7.08, 7.08	7.08, 3.58	3.58, 3.58	1.83, 1.83	3.58, 1.83	7.08, 1.83	14.0, 1.83	17.17, 1.83	0.83	0.15
0.13	6.54, 6.54	6.54, 3.31	3.31, 3.31	1.69, 1.69	3.31, 1.69	6.54, 1.69	12.92, 1.69	15.85, 1.69	0.92	0.3
0.14	6.07, 6.07	6.07, 3.07	3.07, 3.07	1.57, 1.57	3.07, 1.57	6.07, 1.57	12.0, 1.57	14.71, 1.57	0.71	0.1
0.15	5.67, 5.67	5.67, 2.87	2.87, 2.87	1.47, 1.47	2.87, 1.47	5.67, 1.47	11.2, 1.47	13.73, 1.47	0.87	0.32
0.16	5.31, 5.31	5.31, 2.69	2.69, 2.69	1.38, 1.38	2.69, 1.38	5.31, 1.38	10.5, 1.38	12.88, 1.38	0.88	0.26
0.17	5.0, 5.0	5.0, 2.53	2.53, 2.53	1.29, 1.29	2.53, 1.29	5.0, 1.29	9.88, 1.29	12.12, 1.29	0.88	0.15
0.18	4.72, 4.72	4.72, 2.39	2.39, 2.39	1.22, 1.22	2.39, 1.22	4.72, 1.22	9.33, 1.22	11.44, 1.22	0.72	0.25
0.19	4.47, 4.47	4.47, 2.26	2.26, 2.26	1.16, 1.16	2.26, 1.16	4.47, 1.16	8.84, 1.16	10.84, 1.16	0.84	0.24

A.3 Decoding Examples

This appendix section includes more examples of the decoding process.



(a) Corners (b) Decoded

Figure A.3: The specific shape shows how inner corners, as calculated in lines 4 to 9 of the Algorithm 3.1, are required in the decoding process. If this specific step were not present, the block in the centre would be found or decoded.

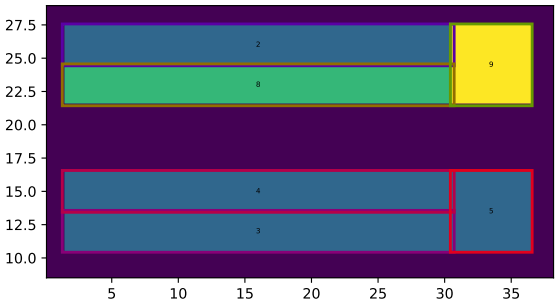


Figure A.4: Decoding case in which the original rectangle of the lower structure is so big that the two rectangular blocks don't fill it fully. The two blocks are created and an extra rectangle is added to the side.

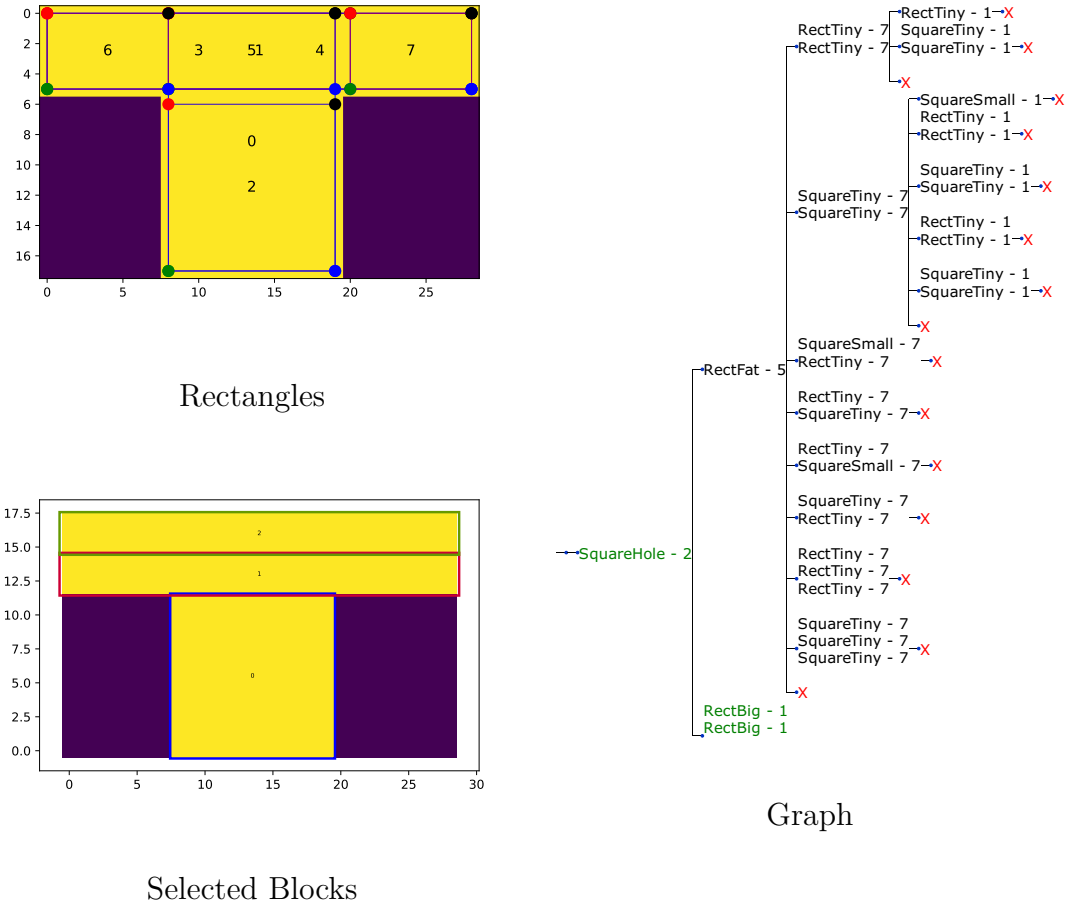


Figure A.5: A bit more complex example of the selection process. It can be seen that the selection selects big blocks in the centre first, which results in wrong paths.

A.4 Confidence Decoding Examples

This section presents a few examples that were created but not decoded in the thesis. Also, a few examples that do not appear but are interesting are added.

A.4.1 Application

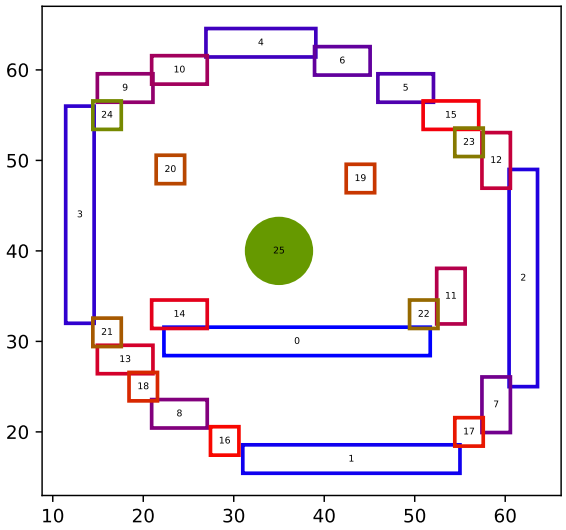


Figure A.6: The decoding of the smiley used to visualize the drawing capabilities of the Level Drawer application.

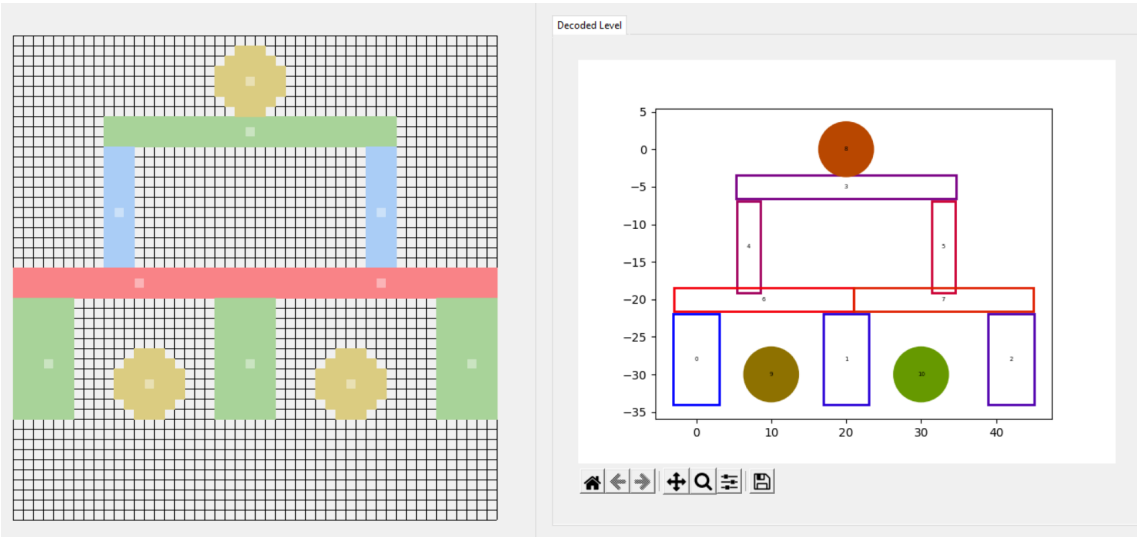


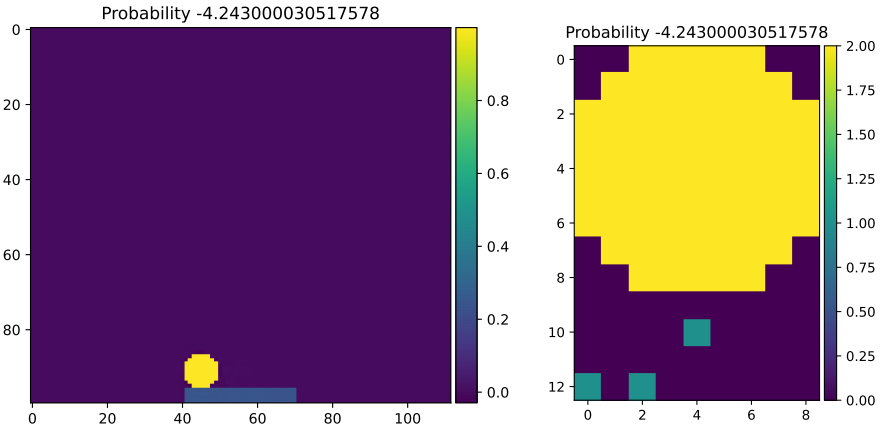
Figure A.7: One element drawing example

	Round To Next Int	Custom Kernel Scale	Minus One Border	Combine Layers	Negative Air Value	Cutoff Point
Smallest Damage	True	True	False	True	-1	0.8
Stable	True	True	False	True	-1	0.5
Destoyed blocks	False	True	True	True	-1	0.8
Height	False	False	False	False	0	0.1
Block Amount	True	False	True	False	-10	0.1

Table A.3: Table that visualizes the best parameter set for the respective characteristic.

A.5 More Results

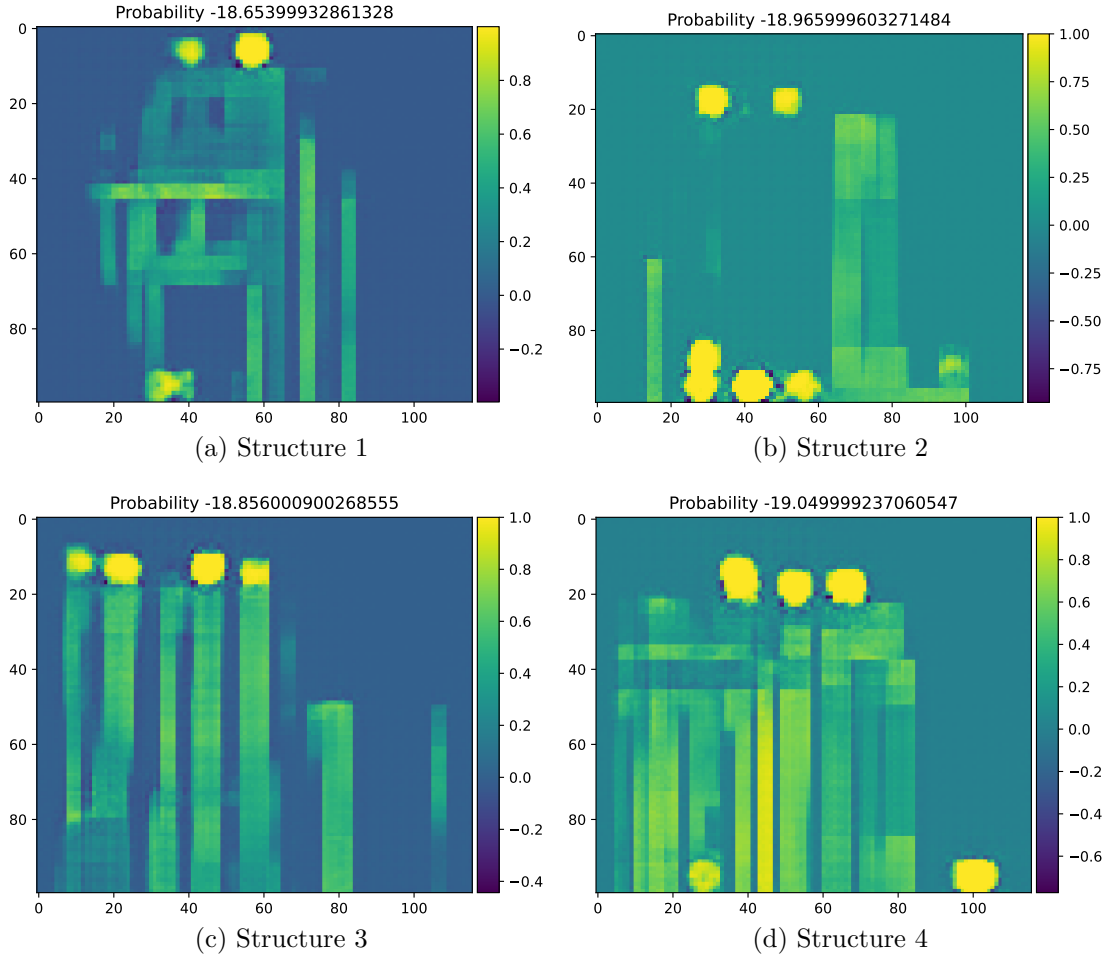
A.5.0.1 Simple GAN - 1



(a) Level 3

(b) Rounded to the next Integer

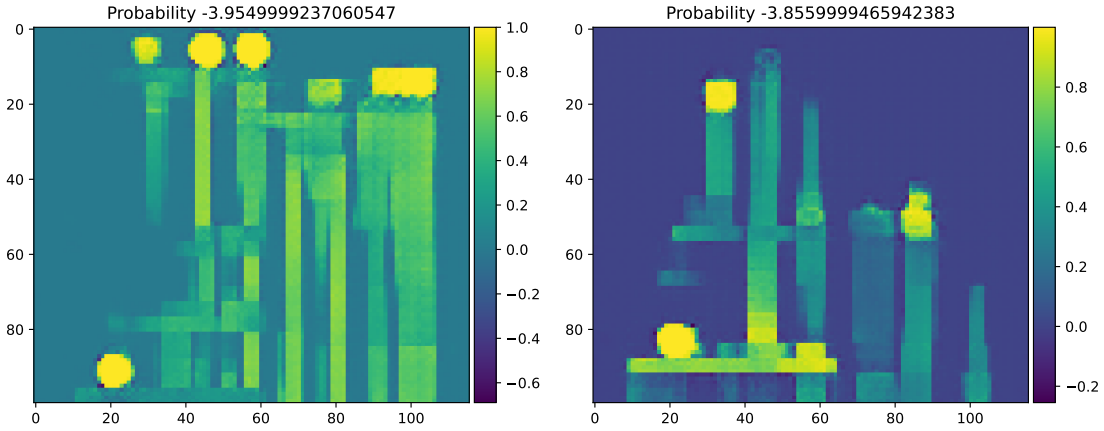
Figure A.8: The confidence in the block below is not high enough so that rounded to the next integer it is closer to zero and removed from the rounded representation.



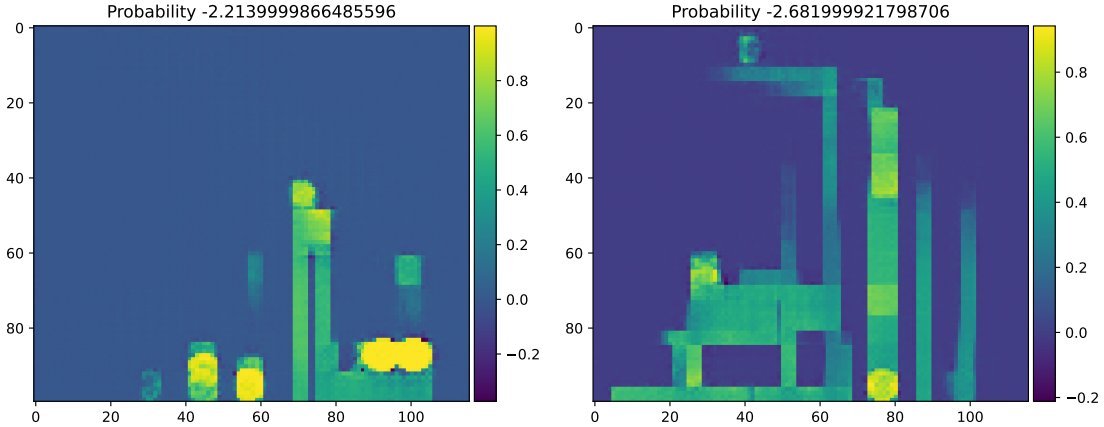
(a) Structure 1 (b) Structure 2

(c) Structure 3 (d) Structure 4

Figure A.9: Levels generated with the Simple GAN architecture and a filtered dataset with the WGAN training algorithm.



(a) Structure 1 (b) Structure 2



(c) Structure 3 (d) Structure 4

Figure A.10: Levels generated with the Simple GAN architecture, filtered dataset and the WGAN training algorithm with a Adam optimizer over 15000 epochs.

A.6 Quantitative Evalutation Results

A.6.1 Grid Search

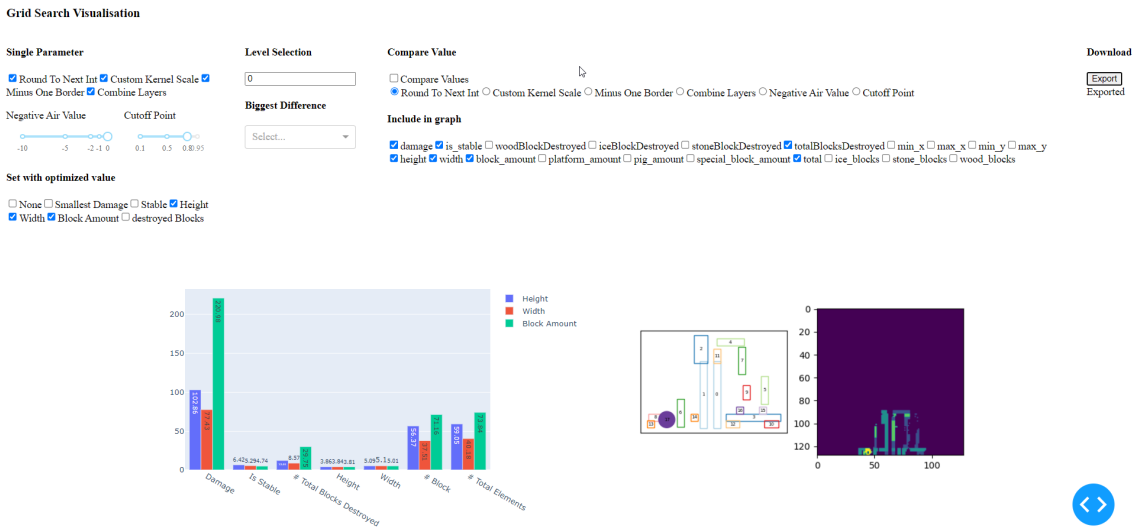
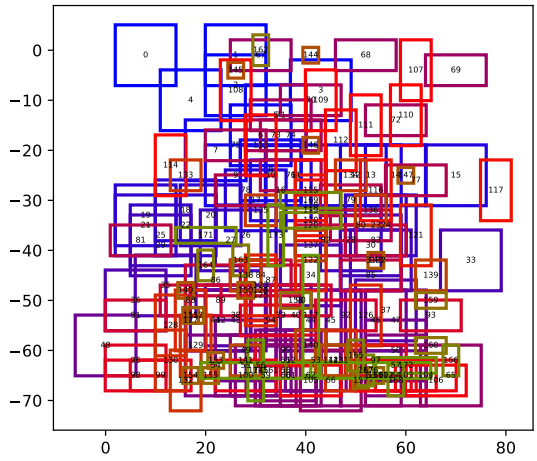
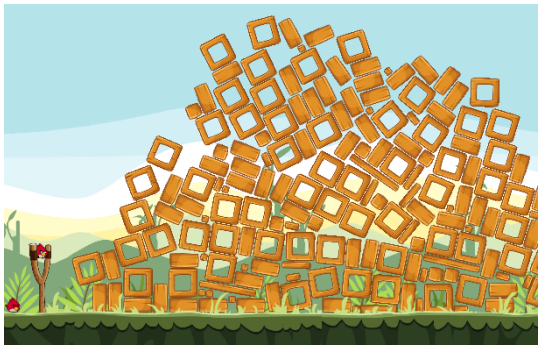


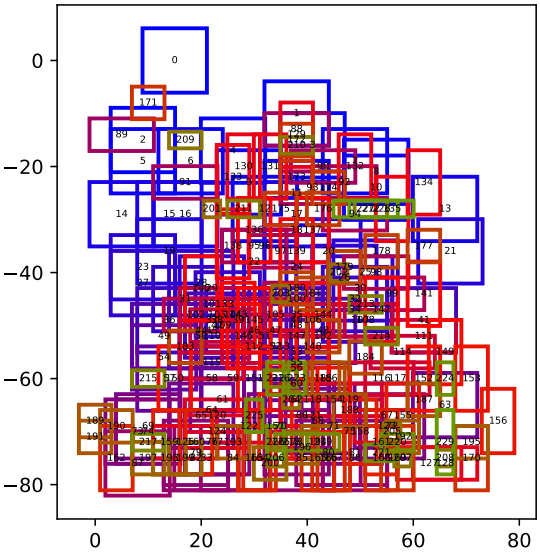
Figure A.14: Application written for evaluating the grid search data.



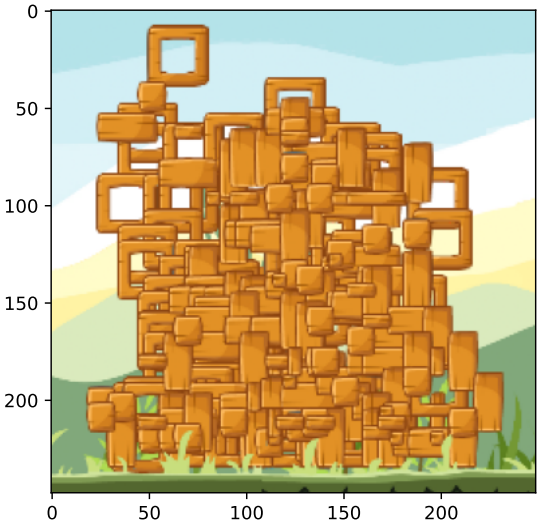
(a) Decoded Structure 2



(b) After simulation resumes



(c) Decoded Structure 3



(d) Before simulation resumes

Figure A.11: The decoded structures of the Single-Layer One-Element encoding shown in Figure 5.7a and Figure 5.7b

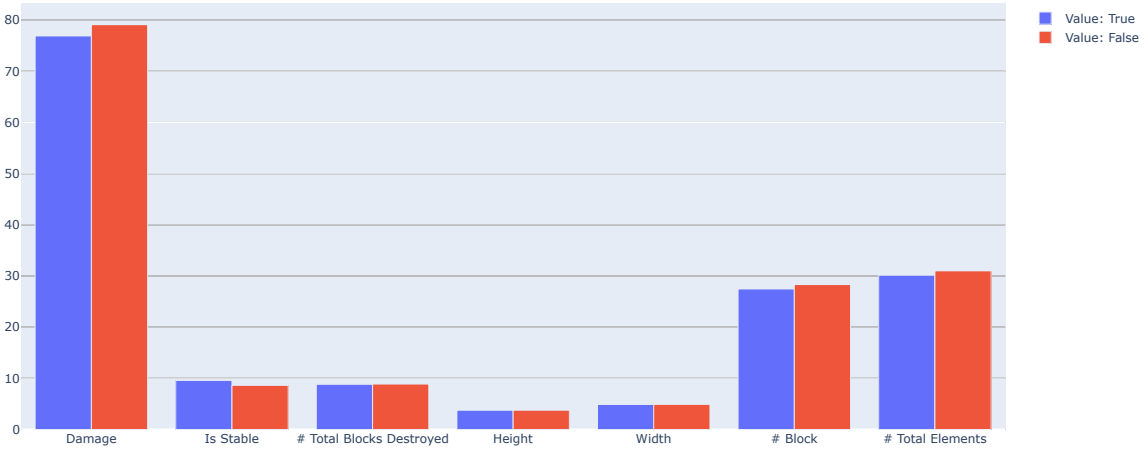
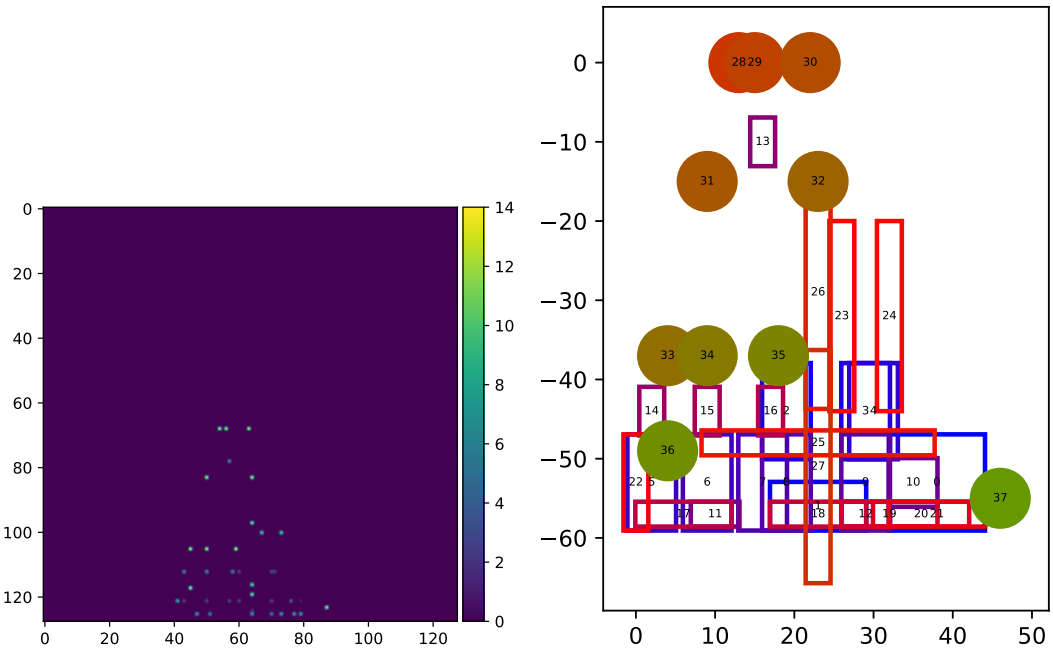
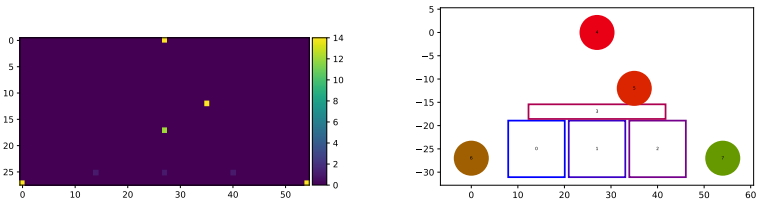


Figure A.15: Graph that compares the data with both states of the "round to next integer" parameter. Using rounded values results in a more stable level.



(a) Structure 3 in Small One-True-Hot encoding with air

(b) Structure 3 decoded



(c) Structure 4 in Small One-True-Hot encoding with air

(d) Structure 4 decoded

Figure A.12: More examples of the model using the smaller One-True-Hot encoding trained on the convolutional WGAN architecture.

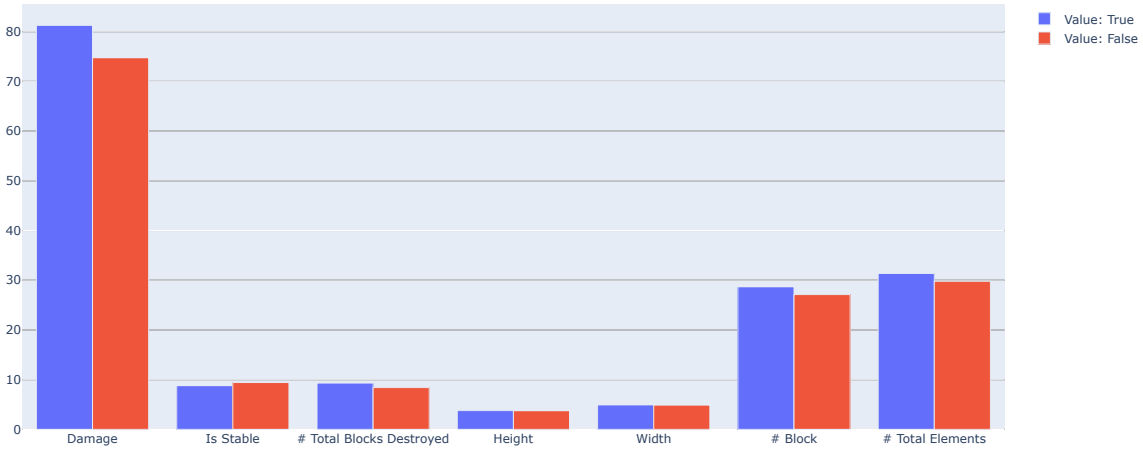
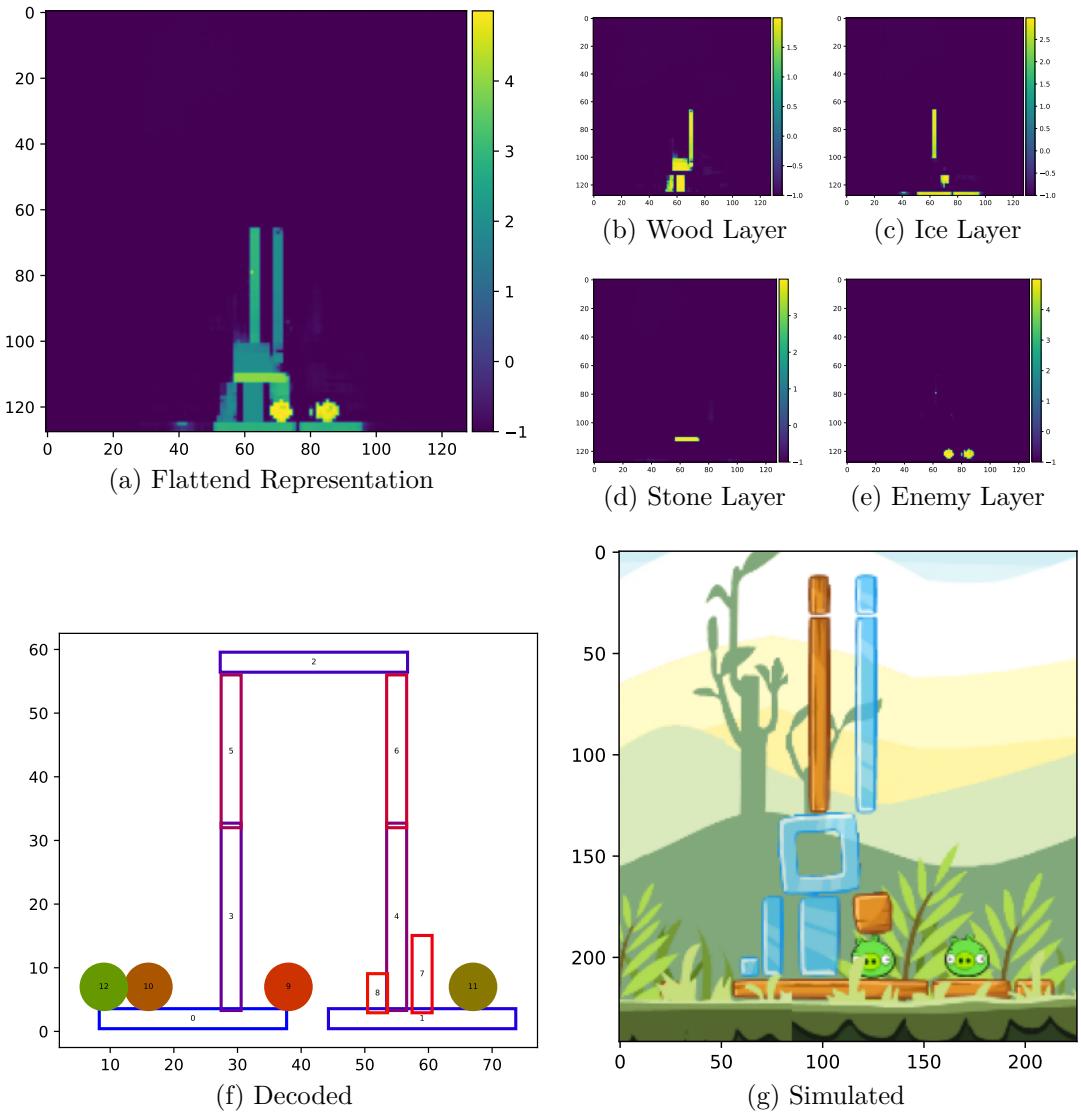


Figure A.16: A structure decoded with both states of the MinusOneBorder. Using a minus one border results in less stable levels and more damage.

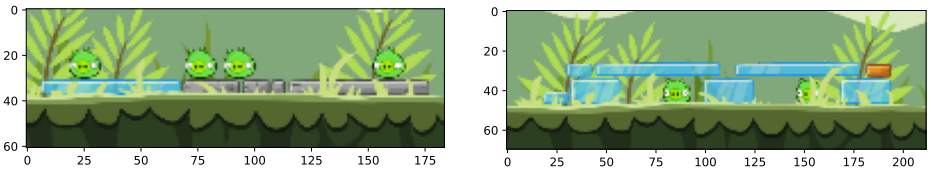


(a) Flattend Representation (b) Wood Layer (c) Ice Layer (d) Stone Layer (e) Enemy Layer (f) Decoded (g) Simulated

Figure A.13: More examples of the multilayer structure without air

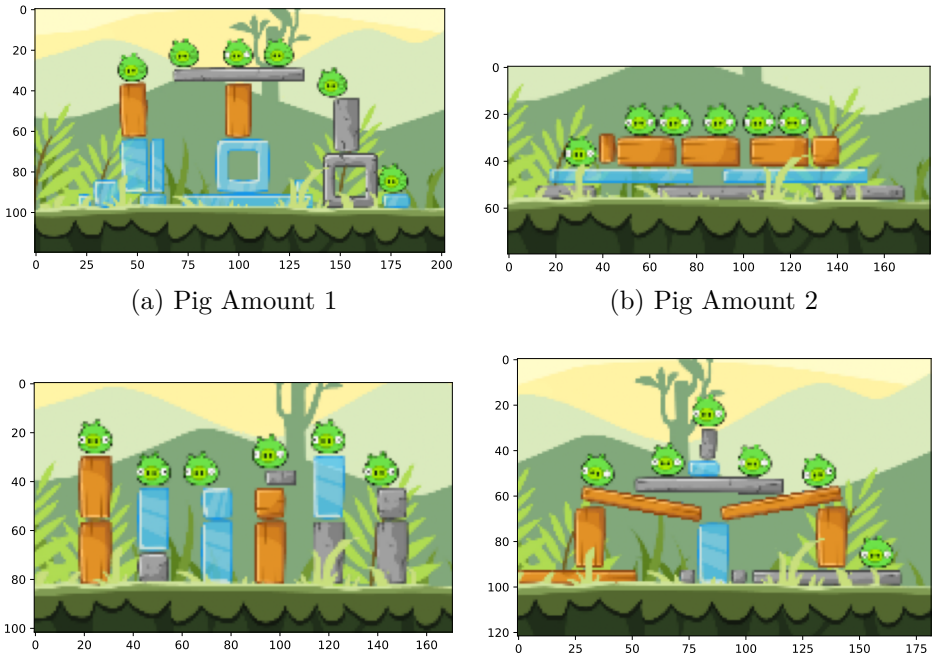
A.6.2 Quality Search

Add more examples



(a) Low Profile 1 (b) Low Profile 2

Figure A.17: Generated stable structures based on minimizing the structure height.



(a) Pig Amount 1 (b) Pig Amount 2

(c) Pig Amount 3 (d) Pig Amount 6

Figure A.18: Generated stable structures based on maximising the amount of pigs.