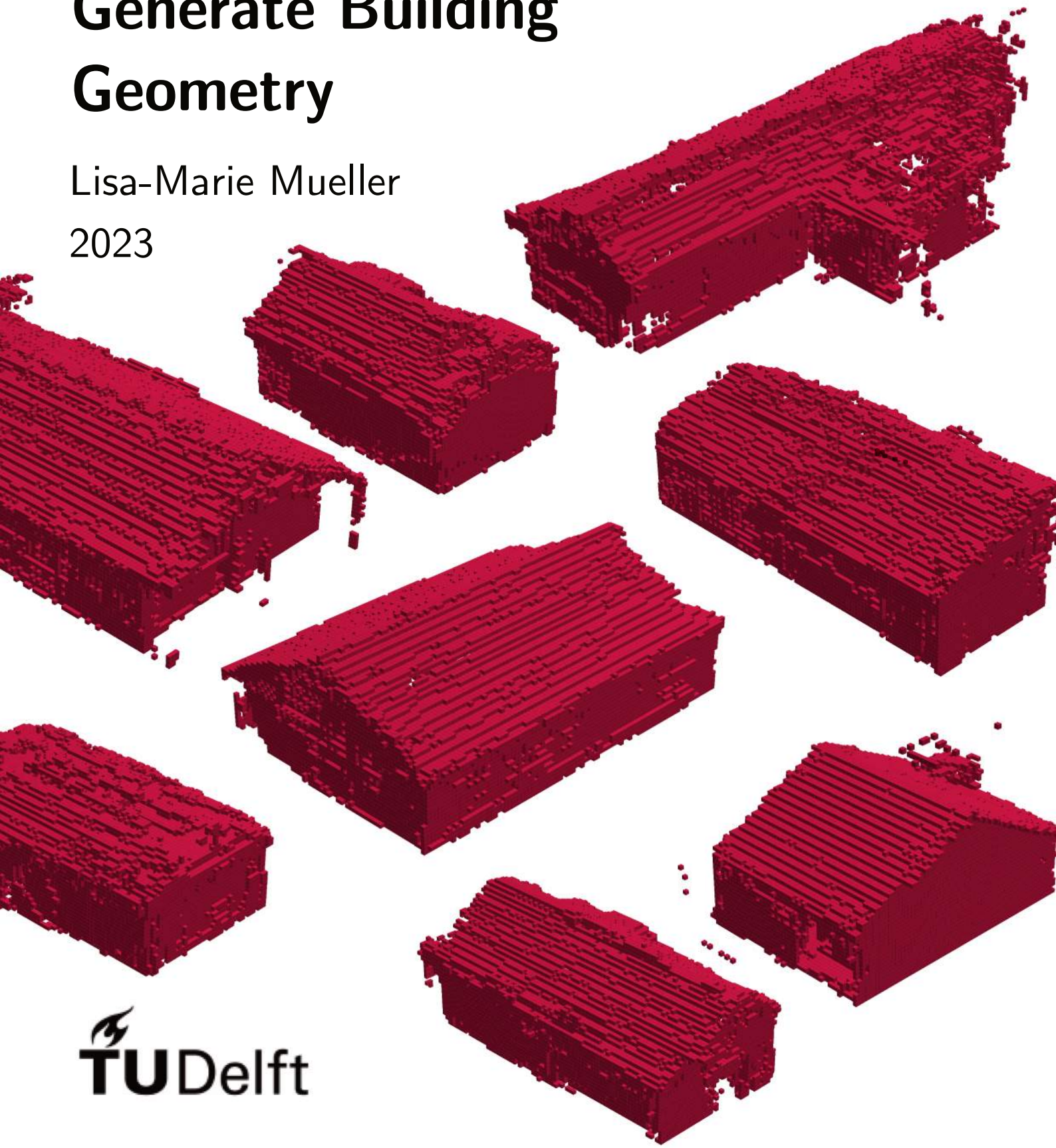
MSc Thesis in Building Technology

3D Generative Adversarial Networks to Autonomously Generate Building Geometry

Lisa-Marie Mueller

2023



TU Delft

MSc Thesis in Building Technology

3D Generative Adversarial Networks to Autonomously Generate Building Geometry

Lisa-Marie Mueller

June 2023

A thesis submitted to the Delft University of Technology in partial fulfillment of the requirements for the degree of Master of Science in Architecture, Urbanism, and Building Sciences

The work in this thesis was carried out in the:

Building Technology Track
Delft University of Technology

Supervisors:	Dr. Michela Turrin
	Dr. Charalampos Andriotis
External Examiner:	Dr. Olindo Caso

Abstract

Across the world, countries are facing housing shortages and the Netherlands is no different. The increasing demand for new housing exceeds the growth rate of the architecture, engineering, and construction industry. Current solutions remain small in scale and therefore unsustainable. Multi-family housing is the optimal typology to address the housing shortage but the industry cannot design and build these projects fast enough. Automation can help. In 2016, Kevin Kelly [2016] reframed the conversation about automation, by stating "our most important mechanical inventions are not machines that do what humans do better, but machines that can do things we can't do at all." Humans cannot address the housing crisis alone, but automation through the application of deep learning models can bring well-designed spaces to everyone.

Since the introduction of computers, architects have looked for ways to automate menial tasks. Some researchers even imagine a future where the machine becomes a partner to architects and designers, contributing to the design process. This ambition requires that the algorithms train themselves. Innovations in the field of deep learning have made this possible by allowing algorithms to train themselves through the use of artificial neural networks. When it comes to applying deep learning to generative design tasks, however, there is little research. The studies that have been done generate geometry that is small in size (64 x 64 x 64 voxels) and focuses on objects like chairs, not on buildings. 3D generative adversarial networks show promise for generating building geometry. By automating design, it is possible to apply expert knowledge on good design to all projects so everyone has access to well-designed buildings.

This research aims to develop the architecture for a generative adversarial network that produces feasible building geometry. An important first step was identifying and pre-processing a data set that could be used for this purpose. The data set is released with the publication of this thesis so it can be used for further research. Through this thesis, the Improved 3D Wasserstein Generative Adversarial Network architecture has also been developed and documented. The research found that using a combination of Wasserstein loss with gradient penalty, Leaky ReLU activation functions in the generator and the critic, and the RMS Prop optimizer results in an architecture with stable training and outputs that are similar in size, shape, and proportion to the training data with minimal noise in the output. Performance of 3D Wasserstein generative adversarial networks with these hyperparameters was improved even further when using ten layers and a larger number of channels. The experiments concluded that generative adversarial networks can be used to generate building geometry and can be an area of continued research to improve generative design tools and support the automation of architectural design.

Acknowledgements

My sincere gratitude goes out to everyone who supported me on the journey to submitting my Master's Thesis.

My deepest appreciation goes to my supervisors, Dr. Michela Turrin and Dr. Charalampos Andriotis, for your invaluable guidance and feedback throughout my thesis work. Thank you for helping me to pursue a challenging topic and championing me through the process. Your encouragement and expertise helped me complete my research, ensured I learned so much along the way, and made certain that I had a wonderful time doing so.

I am grateful to TU Delft for providing me with the opportunity to conduct my research and for all of the resources and support provided.

I am also thankful to the researchers who created and maintain BuildingNet. Especially, Pratheba Selvaraju, for answering my questions and providing supplemental information for the data set.

Thank you to my family. This endeavour would not have been possible without my incredible wife and partner, Andrea. Thank you for making possible the decision to move 5,450 miles (8,770 km) across the world to pursue my dreams. Thank you for being there through all the challenges and successes along the way. Thank you for your unwavering support in everything I do. I wouldn't be here without you. Thank you also to my parents, Andrea and Georg, and brother, Marius, for your unconditional love, support, and understanding and for always believing in me.

I am so grateful to have an amazing support system here in Delft. Special thanks to Frederik for being my rock and my rubber duck. Thank you for believing in me even when I felt defeated, exchanging ideas with me, and making this year so great. I am so lucky to have you in my life. I am so thankful for my incredible friends who have brought so much joy and encouragement these past two years. Thank you especially Sebastian for always being there to work through challenges and exchange ideas. Thank you Alex, Carmen, Eliana, Ifrah, Iiris, Juan, Kyujin, and Marije for your encouragement and for always being there to celebrate each other's accomplishments throughout the Master's program.

Finally, I would like to thank all of my professors and my classmates throughout the program for their time and willingness to share their expertise and experiences. It has been a wonderful journey and I am so grateful to have had the opportunity to learn from so many talented individuals.

I am grateful to everyone who has supported me throughout this process. This thesis would not have been possible without your help and guidance.

Contents

List of Tables

List of Algorithms

Acronyms

2D	two-dimensional
3D	three-dimensional
3D GAN	three-dimensional generative adversarial networks
3D GAN	three-dimensional generative adversarial network
3D WGAN	3D Wasserstein generative adversarial network
ANN	Artificial Neural Networks
AI	artificial intelligence
AEC	architecture, engineering, and construction
batch norm	batch normalization
BIM	building information model
CAD	computer-aided design
CFD	computational fluid dynamics
CNN	Convolutional Neural Networks
CRAFT	Computerized Relative Allocation of Facilities Technique
DCGAN	deep convolutional generative adversarial network
DL	deep learning
DHPCC	Delft High Performance Computing Centre
ELU	Exponential Linear Unit
EM	Earth-Mover
FEA	finite element analysis
GAN	generative adversarial network
I3D WGAN	Improved 3D Wasserstein Generative Adversarial Network
IID	independent and identically distributed
IWGAN	Improved Wasserstein Generative Adversarial Network
JS	Jensen-Shannon
KL	Kullback-Leibler
LOD	level of detail
lrDecay	learning rate decay
ML	machine learning
MLP	multi-layer perceptrons
NN	Neural Network
ReLU	Rectified Linear Unit
RMSProp	root mean squared propagation
WGAN	Wasserstein generative adversarial network

Definitions

architecture in the study of the built environment, architecture refers to the art or practice of designing and constructing buildings.

architecture in the field of machine learning, architecture refers to the definition of the neural network and includes its structure and hyperparameter settings.

building model (also 3D model(s)) in the study of the built environment, a model is a type of scaled representation made to study aspects of an architectural design or to communicate design intent. It can be digital or physical but in this thesis, it refers to three-dimensional digital representations. Although frequently just called model, this thesis will refer to these models as 'building models' or 'three-dimensional models' to distinguish them from the term 'model' used in the field of machine learning.

classification in machine learning, the kind of model prediction in which a class label is anticipated for a specific example of input data. An example is predicting if an email is spam or not spam.

critic in a generative adversarial network, the critic learns features of the data set through a convolutional neural network and scores the realness or fakeness of the generator's output through an output of a scalar between $-\infty$ and ∞.

data set a collection of information where each data point in the set contains information that a computer can read.

discriminator in a generative adversarial network, the discriminator learns features of the data set through a convolutional neural network and then tries to identify if the geometry created by the generator is real or fake. Its goal is to maximize the objective loss function by correctly categorizing the generator's output.

fidelity describes the gap between what a model learns and the ground truth learned by humans. High fidelity means there is a small gap between learned and ground truth.

generator in a generative adversarial network, the generator has an input of random noise and creates an output of geometry within the geometry space. When paired with a discriminator or critic, its goal is to minimize the objective loss function by ensuring the output is realistic and can fool the discriminator or critic.

geometry space the three dimensional, voxel space that the generator outputs with the produced geometry.

gradient of a function, the collection of all the function's partial derivatives into a vector. It denotes the direction of greatest change.

gradient descent an optimization algorithm which randomly initializes and then takes steps in the direction opposite the gradient of the function. It is the most common method for training machine learning models.

latent space an abstract multi-dimensional space that encodes a meaningful internal representation of externally observed events. Similar samples are near each other in the latent space.

model in the field of machine learning, a model is a file that has been or can be trained to recognize certain types of patterns in training data.

norm the length or distance of a vector or matrix.

objective function the mathematical representation of the goal of an optimization problem and represents how good the solution to the problem is.

regression in machine learning, the kind of model prediction where the predicted output is a continuous numerical value. An example is predicting the value of homes based on their features.

space continuity space continuity means that the latent space can be represented by a continuous function. This is important because then it is possible to take the derivative of the latent space which is necessary for the optimization function used for training neural networks.

voxel a representation of a single sample, or data point, on a regularly spaced, three-dimensional grid.

1. Introduction

1.1. Motivation and Problem Statement

As we enter an era where the world population has exceeded 8 billion people [Victor, 2022], we can see the impact of housing pressure around the world. This is also causing more adults to suffer from mental health problems [Shelter, 2017]. A staggering number of homes must be built every year to help [CapitalValue, 2022].

In the five years between 2014 and 2019, the architecture industry grew only 2.1% [IBISWorld, 2021]. At the same time, the demand for new construction has increased at a faster rate. The demand particularly hits the housing sector, as countries around the world are experiencing housing shortages. In the Netherlands specifically, the housing shortage reached 279,000 homes in 2022, which is 3.5% of the housing stock. The shortage is expected to increase as there is a projected 4.7% increase in the number of households between 2022 and 2026 [CapitalValue, 2022]. The increase in demand for just housing exceeds industry growth. These demand estimations do not consider other construction needs such as community centers, libraries, and schools that need to support new housing. Dutch government officials have already discussed that the architecture, engineering, and construction (AEC) industry will not be able to keep up with this demand [NLTimes, 2022].

To keep up with the growing demand for AEC services, the industry must radically rethink the design, planning, and construction process.

These pressures are leading to a scarcity of houses, but also to a scarcity of design services. History has shown that people who are economically disadvantaged are the first to lose access when demand increases and supply decreases. Affordable housing is therefore more at risk of becoming a catalog of poorly designed buildings. Current solutions that have been proposed and implemented include shipping container homes that are too cold to sleep in and rust within weeks of occupancy [Rippingale, 2014] and tiny home "communities" that journalists have described as more like slums than neighborhoods [McManus, 2022]. These solutions demonstrate a lack of understanding of the complex needs of people and provide small-scale, unsustainable approaches to the problem. To address the needs of all, architectural services need to change from a service for the elite to an option for all. To make this shift, multifamily housing should no longer be designed by architects. The design of multifamily housing should be automated through the use of deep learning (DL).

Automating the architectural design process would allow for equal access to well-designed, customized solutions.

When news reports discuss automation taking over another sector of jobs, these sources are quick to confirm that an architect's job is not at risk of being replaced. There are a number of different reasons stated to support the position. Frequently, reports discuss that computers cannot think creatively [Davis, 2015] and cannot adapt to real-world problems [Chahine, 2019]. Despite these objections, engineers have already trained machine learning (ML) models to generate novel images and art - which also requires the same considerations. Philosophically, it is still possible to debate whether ML

models are creative, but it has been shown that they can perform creative tasks [Marr, 2020]. Within the AEC industry the applications of ML models have been based on generating two-dimensional (2D) images like generative floor plans [Keshavarzi and Rahmani-Asl, 2021; Carta, 2021], novel site layouts [Spacemaker, 2023; TestFit, 2023], and context-responsive building massing envelopes [Vesely, 2022]. When we look at other industries, however, innovation does not stop at 2D. In computer science research, there are also applications in 3D. For example, Wu et al. [2016] developed 3D GAN architecture for generating 3D objects like chairs and cars, and Smith and Meger [2017] improved on GANs for 3D geometry by changing the loss function.

Despite these improvements, little research has been extended to generate 3D geometry with respect to the design of buildings. Research in this area has been done by private companies with little published information. Autodesk has hinted at working on such tools at a keynote at Autodesk University in 2014 [Davis, 2015] and a parallel to this research can be seen through their released software Project Dreamcatcher [Autodesk, 2016]. Google had also done some research that resulted in Flux, a start-up that has since closed [Schwartz, 2016]. One company that is currently active and provides advertising services for automated building design is Augmenta. Augmenta promises clients the ability to design sustainable, code compliant, and fully constructible buildings in hours [Augmenta, 2022]. However, their website currently only offers electrical raceway design with other products "coming soon".

There are opportunities to automate design through the applications of deep learning within the architecture, engineering and construction industry.

Because existing research has been completed by various private companies, there is a lack of published information in the area of automating the design of buildings. These gaps present opportunities to expand industry knowledge about how to automate the design process. The goal of automating design is to provide access to thoughtful design to larger groups of people. This thesis takes a step forward to help close this knowledge gap.

The research conducted in this thesis develops strategies to automate the design process. It also supports the expansion of the generative design tools used today. Currently, generative design tools propose simple massing, site layouts, and floor plans. This research lays the foundation for these tools to also propose 3D system designs. Expanding generative design tools to produce novel 3D geometries can lead to time savings and optimizations throughout the design process.

1.2. Objectives, Deliverables, and Questions

This thesis aims to explore one step in the larger challenge of automating design by investigating the opportunities of applying deep learning models to the design of buildings. Initially, the goal was to apply an existing GAN architecture called 3D GAN [Wu et al., 2016] to a new data set and build on this existing research. When using an existing architecture, it would have been possible to expand on the existing research by incorporating multiple building features like doors and windows, and by generating site-responsive designs. After some initial research described in Chapter 7, I determined that the current state-of-the-art architecture was unstable when applied to the building data set. Therefore, I pivoted the focus of my research and proposed a method to develop a new Neural Network (NN) architecture. The new model is trained to generate building geometry.

To test the current state-of-the-art solutions and to develop a new architecture, it was necessary to select and modify a training data set. The selected training data set BuildingNet [Selvaraju et al., 2021] was modified for the purposes of this thesis and can also be used for future applications, including other research related to the generation of 3D building geometry. BuildingNet [Selvaraju et al., 2021]

contains labeled 3D building models which through this thesis have been pre-processed to remove site geometry and relabeled to use only 'wall', 'window', 'door', and 'roof' labels.

After pre-processing the data set, the next step was to develop a GAN architecture that will generate feasible building geometry based on the training data. The largest obstacle was developing a GAN architecture that was stable. GANs are notoriously challenging to train as is further discussed in Chapter 3. To maintain a stable training process and achieve a desirable output, many hyperparameters need to be tested and tuned. A key step in developing a more successful architecture was quickly identifying any problems and testing many different hyperparameter options. The steps to address the research objectives are summarized as follows:

1. Pre-process data set
2. Test 3D GAN [Wu et al., 2016] architecture on training data set
3. Test different loss functions
4. Tune specific hyperparameters like activation functions and learning rate
5. Refine architecture by tuning use of the optimizer, normalization, and weight clipping
6. Test the impact of adjusting the network depth, width, and kernel size
7. Refine the inputs by testing different input options
8. Identify the best performing architecture

There were two check-points leading up to the final deliverable:

1. completing HouseNet which consists of a sub-set of the existing BuildingNet data set pre-processed for the application of generating 3D building models with detailed information in Appendix A.4
2. training a model with fewer data points that can generate a result that is approximately the correct size, shape, and scale

The final deliverable is a trained model that can generate a feasible building geometry considering the training data set.

The research question for this thesis is:

How can a GAN model be trained to produce 3D building geometry given 3D building models of single-family houses as an input?

The following sub-questions are therefore relevant:

- What are key aspects of a training data set that are required for generating 3D building designs?
- What are the key network architecture hyperparameters that lead to a stable GAN training when generating 3D geometry?
- How does network depth, width, and kernel size impact GAN training when generating 3D geometry?
- How do network inputs impact GAN training when generating 3D geometry?
- What are the challenges and benefits of using GAN for architectural design?

1.3. Scope

This thesis focuses on the applications of DL models to the challenge of generating single-family house designs. As such, there will be no design project as a component of the thesis. Instead, the project is focused on delivering a modified data set tailored to the task of generating building designs and a DL model trained for this application. Since GANs are an emerging model type in the field of DL, research is interdisciplinary as it spans the fields of architecture and computer science. This presents an opportunity for an interdisciplinary research approach that will review the expertise developed in this adjacent field and apply the concepts within the AEC industry.

1.4. Framework

The research framework for this thesis focuses on developing two key areas, the data set and the NN architecture of the GANs. This framework is visualized in Figure 1.1. The topics explored in this thesis are based on a literature review discussed in Chapter 2 and Chapter 3 and followed by research by design described further in Section 1.5.

The initial steps will require pre-processing the existing data set. This involved:

- selecting a building typology: Each building model in the data set comes with a label for the building typology and the building type. The five typologies are commercial, military, public, religious, and residential. The fifteen building types are castle, cathedral, church, city hall, factory, hotel building, house, monastery, mosque, museum, office building, palace, school building, temple, and villa. Chapter 4 and chapter 5 discuss how a relevant building typology was selected.
- evaluating and editing the geometry of the building models: Most building models in the data set contain site geometry that is not relevant to the research. This excess geometry must be removed before the building models can be used for training, as discussed in Chapter 5.
- determining the resolution of the training data: When the training data set is used as an input in the GAN, it needs to be formatted in a way that is useful to the algorithm. The algorithm cannot work with point clouds, so instead the data needs to be encoded into an array. Encoding is the process of reformatting the data to make them machine readable. When a point cloud is encoded into an array, each axis of the 3-dimensional array represents the x, y, and z coordinate. Therefore, when the array is visualized, it is a voxel 3D model of the building geometry as shown in Figure 1.2. The voxel visualization tool by Brake [2023] was used for all visualizations of the encoded arrays throughout this thesis.
- selecting the class labels: Class labels are human-readable information that must be mapped to numerical representations so that the algorithm can use this information. This process is called label encoding. The data set being used starts with 31 class labels including the 'wall', 'door', and 'window' labels. This thesis only uses two labels for training, 0 to indicate that a voxel is empty and 1 to represent that it is filled. However, future research would allow for the incorporation of multiple labels in the generated geometry. Chapter 5 describes the process of selecting relevant labels.

The above-mentioned preparation all results in a data set that can be used for training.

Additional steps are required to develop and test the NN architecture. This involved:

- testing hyperparameters: To find a NN architecture that works well for the problem, it is necessary to research, select, and test different hyperparameters alone and in combination. These tests allow the best-performing hyperparameter settings to be identified.

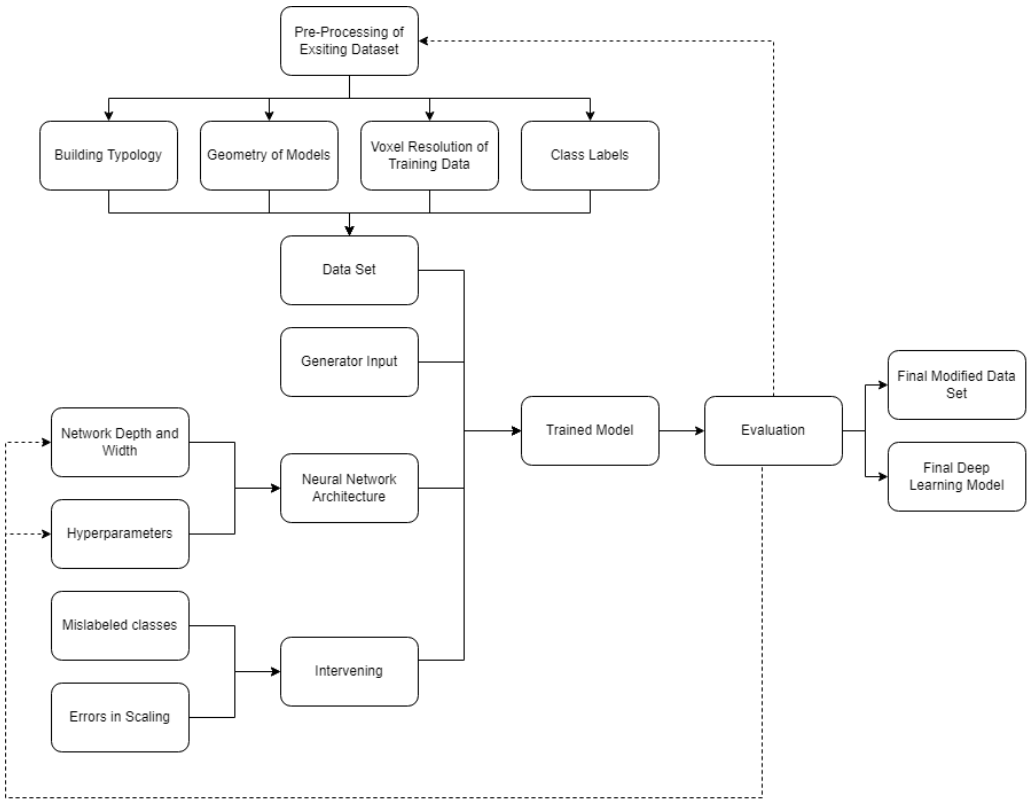


Figure 1.1.: The research framework of the thesis for modifying a data set, creating a GAN architecture, and training a model to generate building geometry.

- testing network depths and widths: The number of layers (depth) and the number of channels (width) affect the performance of the NNs because these impact how well the network learns patterns in the data and also how long the network takes to train.
- testing network inputs: Changes to generator input and training data also have an impact on how well the network trains.

1.5. Methodology

To address all parts of the primary research question, a selection of methodologies is required for each part. The focus of the research is on modifying the existing data set and training a model to generate feasible building designs. This will mainly involve exploratory research and the application of design science methodology [Simon, 1970]. As Simon [1970] notes, it is important to recognize that, unlike studying the natural world, in design research, the result can be different given different goals. Therefore, I strove to document the goals and reasoning of each step to clarify why the research took the path it did.

What are key aspects of a training data set that are required to generate 3D building designs? A key step for training a GAN model to generate feasible building designs is the data set. Deep learning models require large data sets as discussed further in Section 3.2. The proposed data set, BuildingNet v0.1 [Selvaraju et al., 2021], was originally developed to label building features in 3D building models.

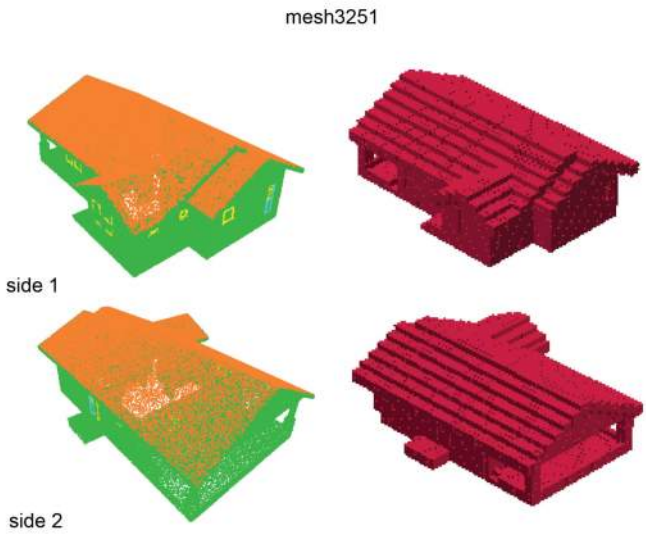


Figure 1.2.: The images on the left show the point cloud and the images on the right visualize the matrix that represents the encoded point cloud as voxels.

Because the data set will now be used to train a DL model that generates buildings, the data set needs to meet requirements different from the purpose for which it was originally created. This research aims to determine what changes are required in the data set and to implement these changes. This sub-question will be addressed through a literature review and by experimenting and testing.

What are the key network architecture hyperparameters that lead to a stable GAN training when generating 3D geometry? GANs have a complex structure that allows tuning of the network architecture by changing many different hyperparameters. This also means that many options need to be tested. Some examples of hyperparameters that need to be tuned include the loss function, activation functions, normalization methods, and others. It is important to research how each parameter is used and how it can impact the output to be able to apply this knowledge when developing a new network architecture. This sub-question will be addressed through a literature review and through exploratory research by experimenting and testing.

How does network depth, width, and kernel size impact GAN training when generating 3D geometry? In addition to hyperparameters, there are other factors of GAN architecture which must be reviewed. Network depth, width, and kernel size all have an impact on the performance, output, and computational resources required to train. Finding a reasonable balance between these parameters and a successful training result is important. This sub-question will be addressed through a literature review and through exploratory research by experimenting and testing.

How do the network inputs impact GAN training when generating 3D geometry? GANs have multiple inputs. The input to the critic is the training data set, and the input to the generator is noise. There are options to adjust these inputs, such as increasing the size of the training data set or having the random noise input of the generator be more similar in size and shape to the expected output. All of this impacts training time and quality of the output. This sub-question will be addressed through a literature review and through exploratory research by experimenting and testing.

What are the challenges and benefits of using three-dimensional generative adversarial networks (3D GAN) for architectural design? Based on all aspects that the research will explore, the overall challenges and benefits will be reviewed. The results will be observed and evaluated to determine what additional progress can be made and whether the goals of the research were achieved.

1.6. Verification and Validation

For GAN, the loss cannot be used as a metric to objectively assess the progress of training or the quality of the model. Borji [2022] reviewed a number of metrics for assessing GAN models and determined that the measurements should favor GAN models which:

- generate high fidelity samples
- generate diverse samples
- have disentangled latent spaces and space continuity
- have well-defined lower, upper, and chance bounds
- can handle data set distortions and transformations which means it is robust to imperfect data and translated data
- align with people's rankings of models
- have low sample and computational complexity which means it minimizes the use of computational resources during and after training

[Borji, 2022]

Of the metrics discussed by Borji [2022], some were focused on GAN models trained for classification problems that have training and test data. These metrics were not considered because they are not applicable. Some other methods were only applicable for 2D GANs so these methods were also excluded.

After a review of the various metrics, two qualitative metrics were selected:

- nearest neighbors: generated samples are reviewed next to their nearest neighbors in the training set
- visualization and evaluation of the output: the output of each training iteration will be visualized and compared between different architectures and to the training data

The performance of the trained model also validates the data set and framework. The above mentioned performance metrics will be used to evaluate the different models trained during this research.

2. Design Automation

The thesis builds upon existing research in the fields of computer science, design, and engineering. The main areas of literature research include reviews focused on two key areas: the automation of architectural design and the exploration of DL models, specifically GANs. This chapter further discusses the former: the automation of architectural design.

2.1. Literature Research Methodology

The literature review investigates sources that discuss design automation, multifamily building typologies, and design tools. The search looked at the title, abstract, and keywords of the literature in the JSTOR, ResearchGate, Science Direct, Scopus, and WorldCat Discovery databases. The keyword search used to find information and applicable case studies for architectural theory included the terms deep learning for architecture/design, generative adversarial networks for architecture/design, design automation, automation in construction, and generative design. Furthermore, sources cited by articles in peer-reviewed journals and conference proceedings were reviewed. From this list of sources, a select number of additional sources were chosen based on those that were frequently referenced. The articles reviewed came primarily from the fields of architecture and design.

2.2. The Need for Automation

Automation within the AEC industry remains an important area of research. Automation increases efficiency and, therefore, scalability in the industry. Design automation research is a topic that building technologists are uniquely positioned to undertake. Building technologists can approach research with a multidisciplinary lens to explore both the design and the informatics perspectives. Research into design automation is beneficial for the industry because it allows architects to complete deliverables faster and more efficiently. This has the potential to decrease design service costs, project costs, project timelines, and design timelines. Through these benefits, the industry can increase its productivity and provide design services to a larger group of people.

The architecture industry needs to improve productivity. Statistics in the years leading up to the pandemic show that demand for construction and design services continued to increase, while growth in the industry has been fairly minimal. Between 2014 and 2019, the architecture industry increased only 2.1% [IBISWorld, 2021] or 0.42% per year. The demand for construction has been increasing at a faster rate. The demand for housing alone has been increasing as the gap in housing availability increases, and the demand is expected to increase by 4.7% [CapitalValue, 2022] in 4 years or 1.175 % per year. Currently, the industry cannot keep up with the rising demand.

If not addressed, the challenge people will face is that a higher demand for design services will lead to higher prices for these services. This prices out clients who need these services but also have low budgets, such as those providing affordable and low-income housing. These projects need to be completed with cost efficiency in mind and in short timelines. To provide everyone with a safe and healthy place to live, the gap between supply and demand needs to be addressed by providing

more services faster. Automation can help. Design automation combines two key concepts: generative design and the digital tools that support these ideas.

2.3. Generative Design

Concepts around generative design have been published about since the 1970s. Research simultaneously explored computerized methods such as the Computerized Relative Allocation of Facilities Technique (CRAFT) program developed by Delon [1970] and manual implementations of generative design systems such as the Pallidian Villas shape grammar by Stiny and Mitchell [1978]. The CRAFT program developed a methodology for automating the size, adjacency, and location requirements of the building program [Delon, 1970]. It also incorporated optimization of the layout based on the time it takes to walk between the building programs and the cost. A flow chart of the logic used in CRAFT can be seen in Figure 2.1. Another early example of generative design included applying concepts to designs manually. Stiny and Mitchell [1978] developed a shape grammar for the Palladian Villas which they then followed to generate floor plans. An example of the grammar and a generated floor plan can be seen in Figure 2.2.

Since the topic generative design was first discussed in publications, a number of different techniques have emerged. There are four techniques that are most commonly used:

- shape grammars [Delon, 1970]
- L-systems [Lindenmayer, 1968]
- cellular automata [Wolfram, 2002]
- genetic algorithms [Holland, 1975]

[Kasmarik, 2023]

Shape grammars [Delon, 1970] describe a set of rules that are used to generate geometry. This concept is probably the most similar to the structure of a programming language, although initially the rules were executed manually. Both shape grammars and programming languages use basic terms or components and a grammar to combine them. Using shape grammars to generate designs can be done computationally or manually. To define a shape grammar, there must be a start rule, at least one transformation rule, and a termination rule. However, most applications expand upon these requirements. When executed computationally, a generation program selects and processes which rules should be applied. Shape grammars can be used to create new designs or they can be developed to understand existing designs. When describing existing designs, the shape grammar can also be used to generate new designs similar to the original. An example of shape grammar is shown in Figure 2.3. This is the shape grammar that Koning and Eisenberg developed for Frank Lloyd Wright's prairie houses [Koning and Eizenberg, 1981]. Shape grammars originated in the architecture industry but have since been used across other industries. [Cagan and Antonsson, 2001]

L-systems, also known as Lindenmayer systems, were originally introduced to describe plant cells and plant growth. This is also the system by which self-similar fractals are generated. An L-system consists of three parts. An alphabet defines the set of replaceable elements called variables and the non-replaceable elements called constants. The start value defines the state of the system upon initialization. The set of production rules defines how the variables can be replaced with constants and other variables. [Lindenmayer, 1968]

Cellular automata were developed in the 1940s by Stanislaw Ulam and John Von Neumann [Cosentino et al., 2013]. They consist of a regular grid of any size and dimension that contains cells. Each cell has a finite number of states, and the neighboring cells are defined relative to a specific cell. A cellular

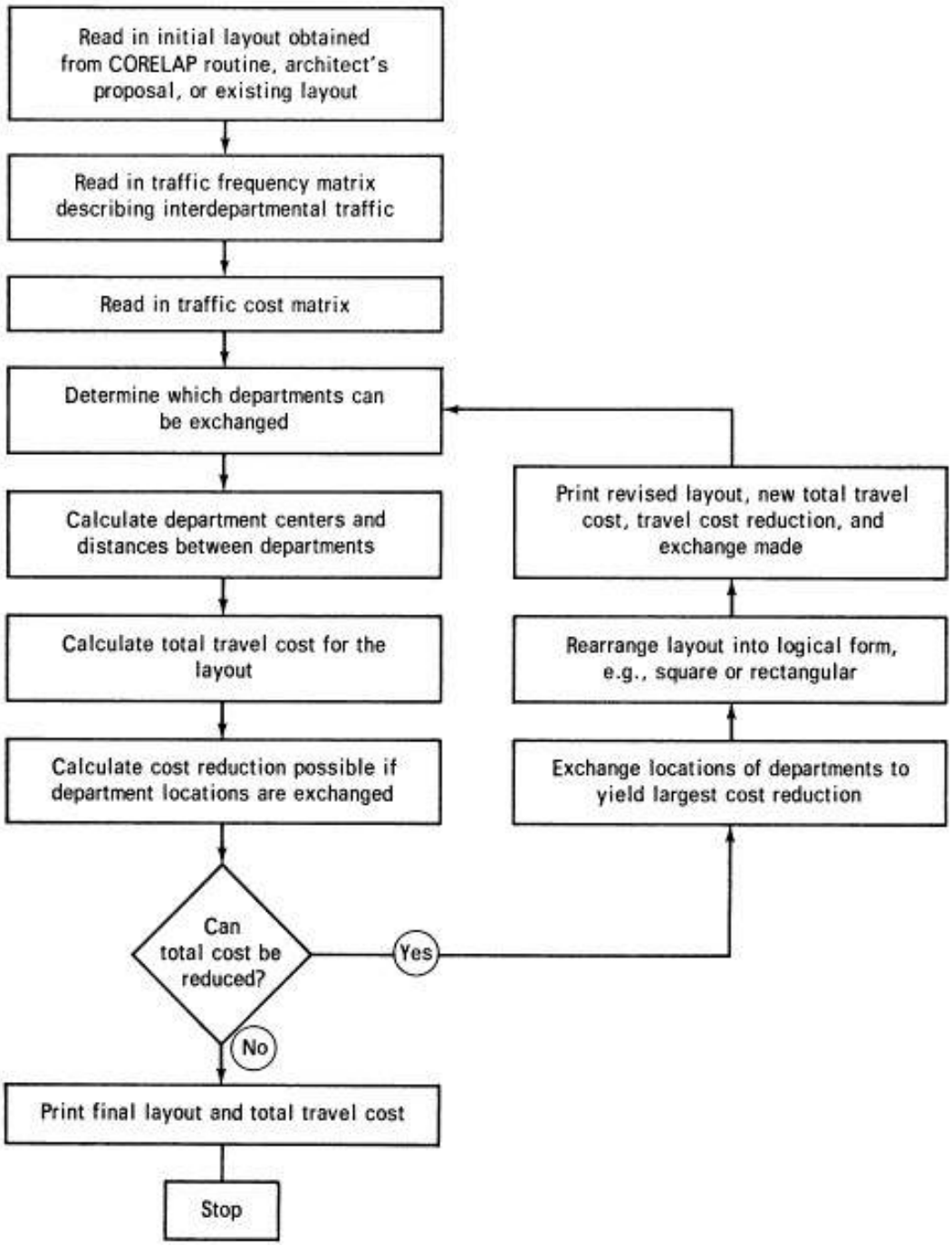


Figure 2.1.: A flow chart of the logic used in CRAFT [Delon, 1970].

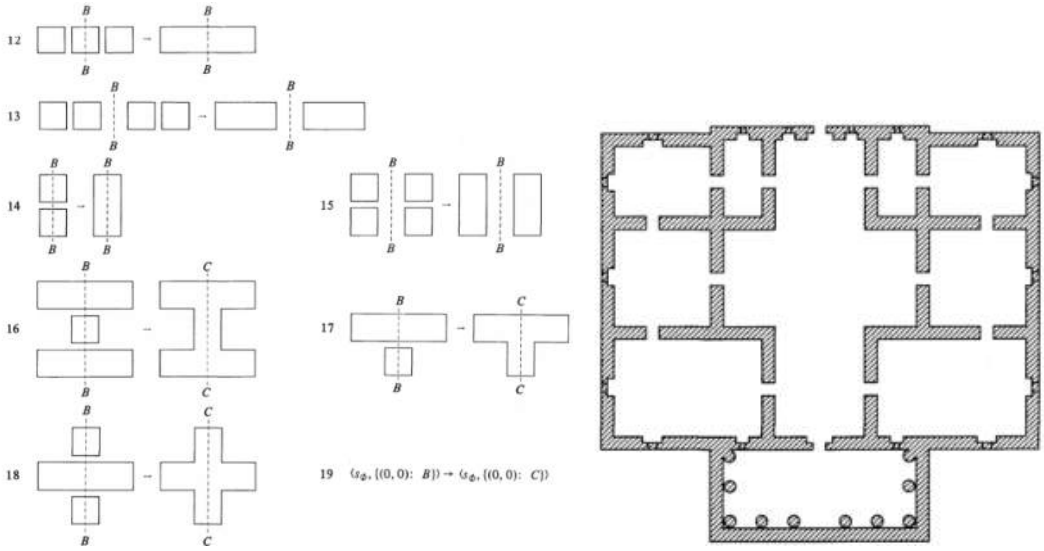


Figure 2.2.: The rules for room layouts (left) in the shape grammar developed for the Pallidian Villas and an example of a generated floor plan (right) using all the rules from the shape grammar [Stiny and Mitchell, 1978].

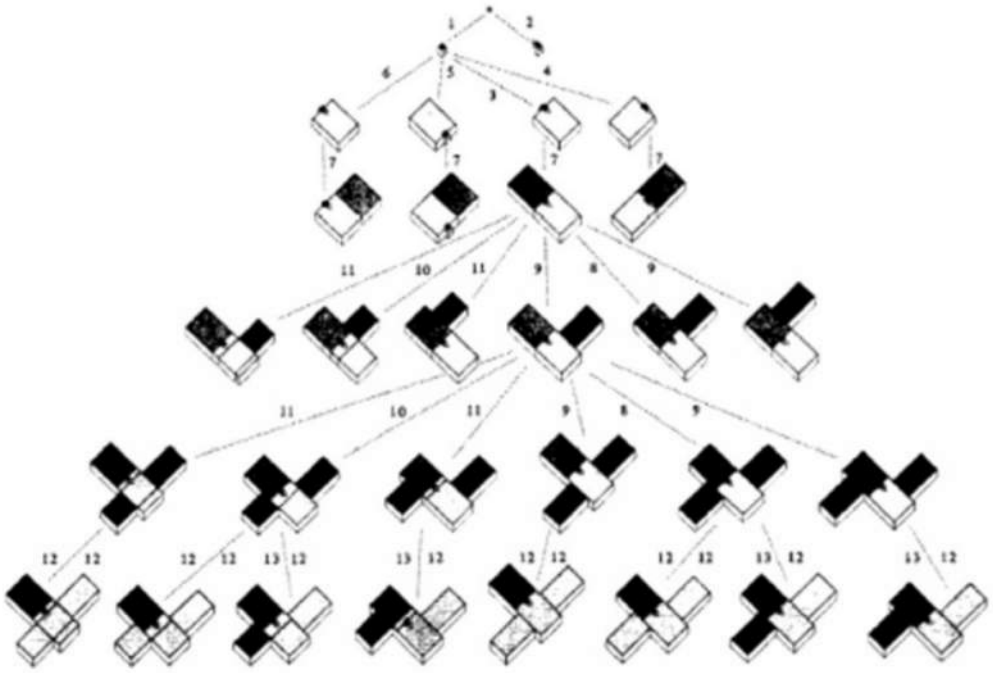


Figure 2.3.: An example of shape grammar developed by Koning and Eizenberg [1981] for Frank Lloyd Wright's prairie houses.

automaton has an initial state and, for each generation, the new state of each cell is based on the current state of the neighboring cells. The rule applied to a cell does not change and all cells are updated simultaneously. [Wolfram, 2002]

Genetic Algorithms are metaheuristics, or functions that tune a heuristic with a focus on quickly finding many solutions to a problem. A heuristic is an underlying function that provides a solution to an optimization problem even when the problem definition is incomplete. Genetic Algorithms are inspired by the process of natural selection, generating solution options based on operators that include mutation, crossover, and selection. [Holland, 1975]

In architectural practice, generative design using genetic algorithms was quickly popularized once the hardware advanced and allowed the implementation of such ideas without coding knowledge.

2.4. Digital Design Tools

Since the computer's first applications in architecture in the 1960's, its use has continued to expand. After the era of 2D drafting, computer-aided design (CAD) software programs like AutoCAD extended their capabilities to allow for 3D modeling as well. Soon after this change, the concept of a building information model (BIM) was conceptualized, developed, and published in the 1970s and 1980s [Ruffle, 1986]. The first BIM program was AEC CAD, released in the mid 1970s and developed by Charles M. Eastman [Eastman, 1975]. BIM programs allow users to model buildings accurately in 3D but instead of representing geometry as volumes, the geometry has embedded information. A wall is modeled as an element that is tagged as a wall and contains information on thickness, finishes, insulation, construction, materials, and other characteristics. This information can then be used to complete simulations, analyses, and to generate construction documentation. At the turn of the century, programs like ArchiCAD and Autodesk Revit popularized the concept of BIM. But innovation did not stop there. As hardware advanced, computers finally had the capacity to process large amounts of data making the implementation of methodologies such as generative design possible.

The early 2000s brought an era of scripting programs. Visual scripting allows users to customize workflows through a graphical interface. The nodes in the script each have specific functions that users can connect to create a variety of results. In 2007, Robert McNeel & Associates released Explicit History, which is now known as Grasshopper. It is a visual scripting language that is integrated with McNeel's CAD software Rhinoceros 3D. Figure 2.4 shows an example of a Grasshopper script that shows how nodes that complete specific functions are connected to generate a result. Before this release, architects and designers had to know how to code to customize their workflow and process, but now they could do so by scripting. A few years later, in 2011, Autodesk, a multinational software company that provides software across numerous industries, released their graphical programming interface called Dynamo for the AEC industry. Figure 2.5 shows an example of a Dynamo script where, similar to Grasshopper, nodes that perform specific functions are connected to automate a task.

The release of these tools allowed architects to automate a variety of tasks necessary to complete their projects. Scripts could be as simple as renumbering construction document sheets or as complex as designing an entire building from site constraints. As users pushed the envelope of what scripting programs could do, independent developers began releasing open-source plug-ins to add functionality. It was through these open-source projects that generative design began to be integrated into the design workflow and accessible to anyone willing to learn Grasshopper or later Dynamo. Early examples include evolutionary solvers like Octopus, released in 2012 [Vierlinger, 2012] and shown in Figure 2.6, and other plug-ins that soon followed.

In 2016, Autodesk released the beta version of a generative design tool called Project Fractal, later renamed Project Refinery. Autodesk's tool brought a more tailored generative design tool that incorporates an evolutionary solver to the masses. These generative design programs were integrated with

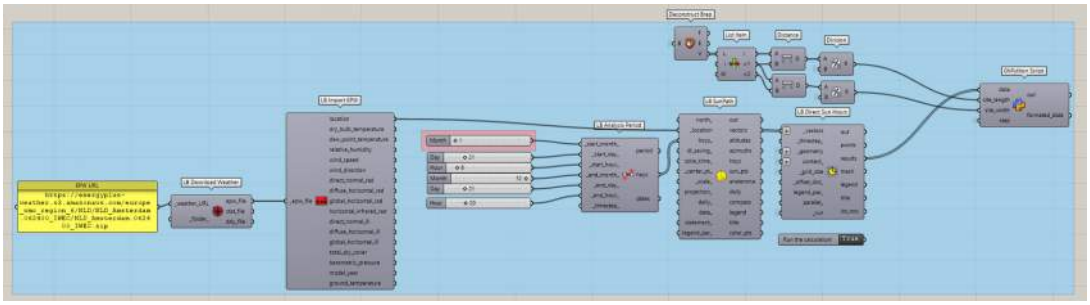


Figure 2.4.: Example of a Rhino Grasshopper script.

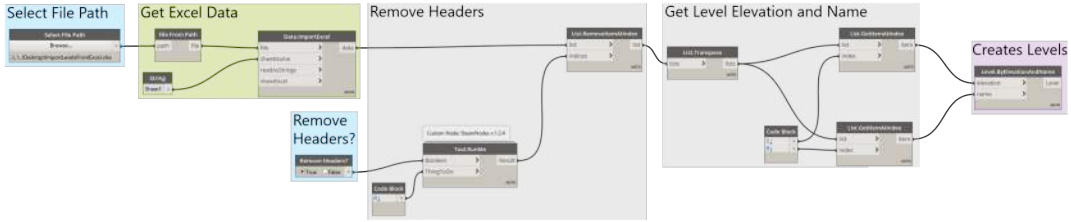


Figure 2.5.: Example of a Revit Dynamo script.

Dynamo, so users still had to know Dynamo to be able to use the tool, but this made generative design significantly more accessible than before. This software allowed architects and engineers to take automation into their own hands. Instead of having to learn to code entire programs or waiting for software companies to develop a product that addressed their problem, firms could develop their own tools with minimal coding knowledge. Research into automation has always been of interest and has so far led to many advances in the AEC industry.

However, generative design still has a number of limitations. It follows a system of rules-based design coupled with parameter tuning. Not only does the problem need to be defined in significant detail within the framework of the existing parametric nodes, there is also a limitation to the number of parameters that can be optimized. Scripting an entire building with defined architectural features and interior layouts can mean a large and extensive script that is time consuming to write and manage.

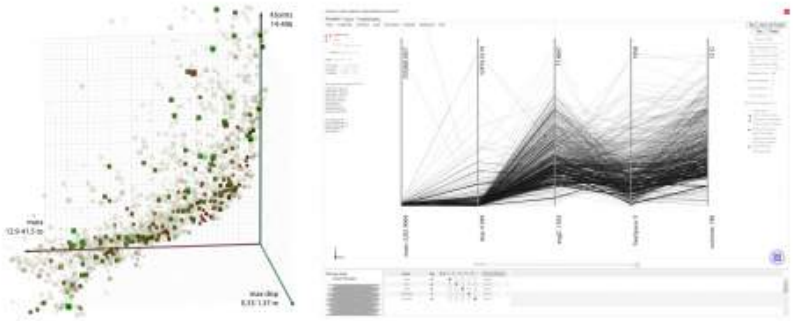


Figure 2.6.: Octopus is an evolutionary solver that allows designers to optimize various design problems. [Vierlinger, 2012]

2.5. Machine Learning in Architecture

Alongside these advances, computational design researchers continue to strive to use the computer for more than just executing explicit instructions. Researchers envision a future where the machine becomes a partner for architects and designers, contributing to the design process. This ambition requires that the algorithms train themselves. Although machine learning has brought some advancements, there have been limitations to the creative tasks these algorithms can perform. DL, however, introduced the concept of having algorithms train themselves through the use of NN. Within the field of DL, GANs have shown promise for generating 3D geometry.

A GAN [Goodfellow et al., 2014] trains two Artificial Neural Networks (ANN) models simultaneously to contest against each other in a zero-sum game. One of these NN is called the generator, and it creates geometry based on a randomized input. The other NN is called the discriminator, and it learns features from the training data set and then classifies the generator's output as real or fake. The generator learns to create believable geometry that fools the discriminator and the discriminator learns to identify fake geometry. Examples of a use of GANs are the early text-to-image generators. The details about how these innovations have been implemented are discussed further in Chapter 3.

Newton [2019] noted that until 2019, when his paper was published, GAN research had been used primarily for 2D outputs (images). The only studies of 3D GAN were with small geometry spaces with a maximum output dimension of 64 x 64 x 64. GAN research originates in the field of informatics and was implemented with the ShapeNet data base, which consists of 3D models of objects such as chairs, cars, and others. One of the deep learning models Newton [2019] reviewed was Improved Wasserstein Generative Adversarial Network (IWGAN). IWGAN developed by Smith and Meger [2017] was an improvement of existing 3D GAN research by Wu et al. [2016]. IWGAN implemented a new loss function for GAN and trained the models on ShapeNet to generate 3D objects like chairs. Newton [2019], the author of the survey paper, implemented IWGAN using an architectural data set; however, there were many limitations. The model was trained on a data set consisting of New York City skyscrapers and generated a geometry geometry space of 32 x 32 x 32 as seen in Figure 2.7. The data set and implementation were not published except for the reference in his survey of GAN use in architecture, so limited information is available. The low resolution output meant that the generated geometry was not meaningful. Buildings that were hundreds of stories high were represented as 32 voxels high, so one voxel represented many stories. With this, Newton [2019] acknowledged that 3D GAN could be useful for a number of other 3D generation tasks in architecture and design if it could produce results with higher resolution. However, increasing resolution poses its own challenges. Due to the fact that geometry is generated in 3D, increasing the size of the geometry space has a significant impact on computational resources. Increasing the geometry space from 32 x 32 x 32 to 64 x 64 x 64 increases the resources required eightfold.

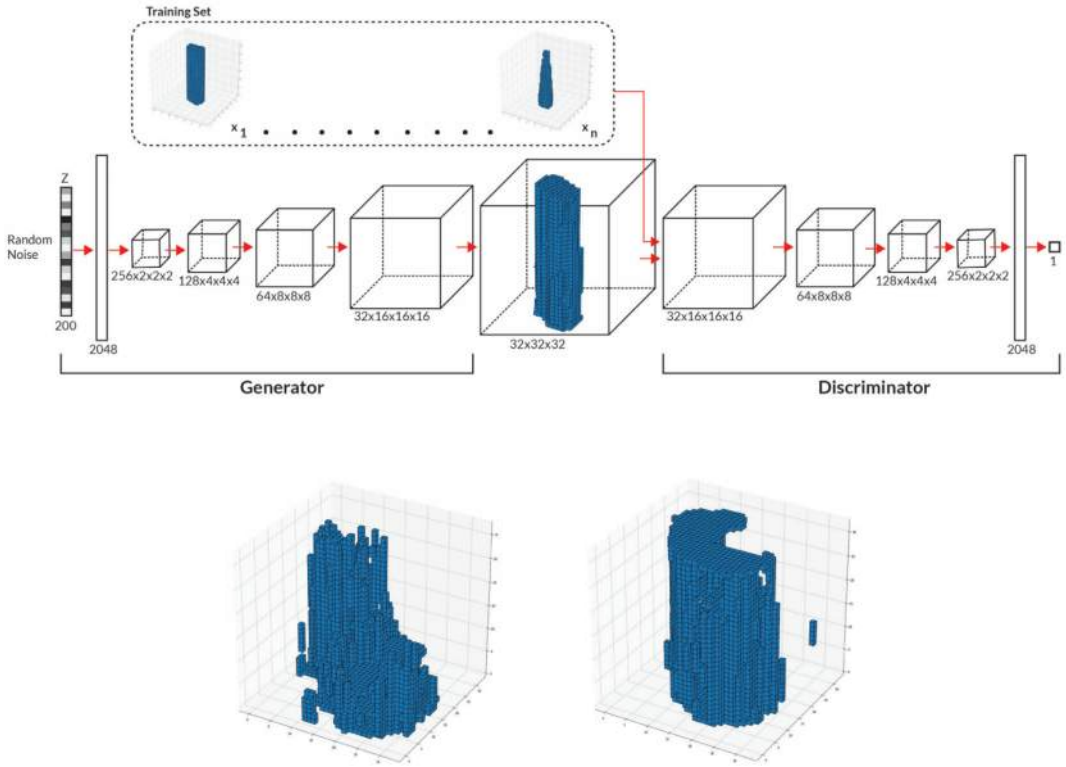


Figure 2.7.: Newton [2019] trained a GAN to generate skyscrapers with a data set consisting of sky scrapers from New York City. The geometry space used was 32 x 32 x 32 voxels.

This survey identifies a gap that still exists. In the four years since Newton [2019] published their survey, minimal additional research has been done in this regard. Another survey by Wu et al. [2022] published in 2022 found that only a handful of additional papers regarding GAN application to the built environment were published between 2019 and 2022. These include topics such as generating terrain maps from semantic masks of street views [Kim et al., 2020], improving the level of detail of the generated 3D models [Kelly et al., 2018; Du et al., 2022], and generating synthetic data sets for building performance studies [Chokwitthaya et al., 2020]. Although all of these research papers all use GANs for 3D architectural applications, their research does not cover automating the design process. The rest of the research surveyed was 2D and therefore used images not 3D geometry. This demonstrates that there is no research on the use of GANs to generate high-resolution 3D building geometry. This is an opportunity to expand the research and investigate the application of DL for building design.

3. Deep Learning

As is described in Section 2.1, the literature review covered two primary topics and extends here with the theoretical background and related work on deep learning.

The literature review investigates sources that discuss DL models for classification, image generation, and generating 3D designs with and without optimization. The search looked at the title, abstract, and keywords of the literature in the JSTOR, ResearchGate, Science Direct, Scopus, and WorldCat Discovery databases. The keyword search used to find information and applicable case studies for DL included the terms deep learning, generative adversarial networks, 3D generative adversarial networks, generative adversarial networks for design, and generative 3D design. Furthermore, sources cited by articles in peer-reviewed journals and conference proceedings were reviewed. From this list of sources, a select number of additional sources were chosen based on those that were frequently referenced. Furthermore, articles published in journals and conference proceedings that pioneered or first documented specific DL topics were reviewed. The articles reviewed come primarily from the fields of architecture, engineering, and computer science.

3.1. Artificial Intelligence

The Oxford English Dictionary [OED, 2022] describes artificial intelligence (AI) as "the capacity of computers or other machines to exhibit or simulate intelligent behavior." This can include any technique that allows a computer to mimic human intelligence. Some examples of AI include self-driving cars, chatbots, and automated investing tools. AI can be narrow or general. Narrow AI focuses on training a model for a specific task on which this thesis will focus. ML is a sub-field of AI which applies statistical techniques and methods that allow computers to improve the output of a task with experience. Some examples of ML include speech recognition, medical diagnosis from scans, and smart assistants. This thesis focuses on a subset of ML called DL. DL consists of algorithms that allow software to train itself to perform tasks and generally require large data sets [LeCun et al., 2015]. Some common methods used in DL include multi-layer perceptrons (MLP) [LeCun et al., 1998; Rumelhart and McClelland, 1987; Werbos and John, 1974], Convolutional Neural Networks (CNN) [Bengio and Lecun, 1997], GAN [Goodfellow et al., 2014], and autoencoders [Kramer, 1991]. Figure 3.1 visualizes the relationship between these related disciplines.

3.2. Deep Learning

As mentioned, DL describes algorithms that allow software to train itself through a NN with more than one hidden layer. DL has already been applied in the field of architecture and engineering. Applications have focused on predicting the results of simulations, but generative design tasks have been less commonly explored. A field of study in which deep learning is applied within the AEC industry is for computational fluid dynamics (CFD) [Wang and Wang, 2021]. This includes studies of predicting wind and heat transfer to circumvent the long and resource-intensive physics simulations. Research aims to reduce the time it takes for simulations to run by predicting the results of CFD and other analyses instead of completing the physics simulation. Additionally, researchers have used DL

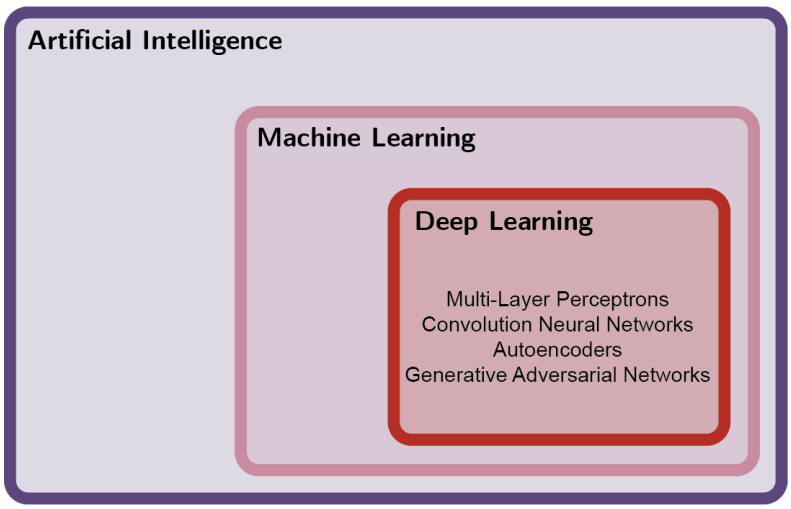


Figure 3.1.: Deep learning is a sub-discipline of machine learning and includes a number of different methods.

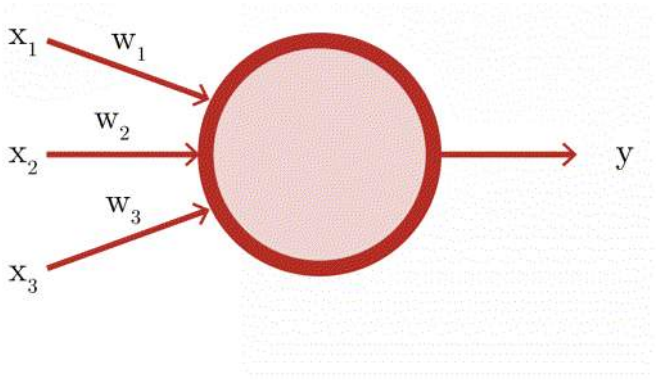


Figure 3.2.: Perceptron Model [Rosenblatt, 1958; Minsky and Papert, 1969]

for topology optimization. Several researchers, including Oh et al. [2019] and Banga et al. [2018], apply finite element analysis (FEA) through DL. These varied applications show how DL has become an imperative tool for simulations and optimizations. Although less commonly researched, it also presents opportunities to further apply DL models in generative design applications. To understand how DL can be used to generate building designs, which will be discussed further in Section 3.5, it is important to first understand how NNs work.

3.3. Neural Networks

NNs are machine learning models which were first conceptualized in 1943 and were inspired by the function of the brain [McCulloch and Pitts, 1943]. This is also why some of the terminology overlaps with neuroscience. A NN is a network made up of components called neurons or nodes and their connections.

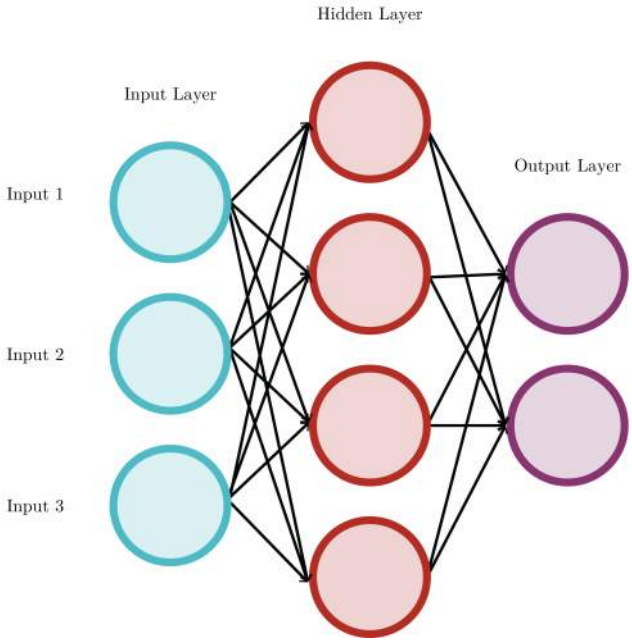


Figure 3.3.: Multi-Layer Perceptron Model [LeCun et al., 1998; Rumelhart and McClelland, 1987; Werbos and John, 1974]

PERCEPTRONS Originally, each node in a NN was a perceptron which was first described by Rosenblatt [1958] and is shown in Figure 3.2. Perceptrons, now sometimes referred to as single-layer perceptrons, are not DL models, but they form the building blocks of contemporary NNs. A perceptron is a threshold function that takes an input and maps it to a binary output value. This makes perceptrons linear classifiers. A simplified notation for the threshold function of a perceptron is:

$$f(x) = 0 \text{ if } w \cdot x + b \leq 0, \text{ and } f(x) = 1 \text{ if } w \cdot x + b > 0. \tag{3.1}$$

where w is the weight and b is the bias. The weight determines the strength of a neuron's output, and the bias value determines how easily a neuron will fire. A neuron fires when it returns an output of a value of 1. The weights and biases are determined when training the network.

MULTI-LAYER PERCEPTRONS In a MLP [LeCun et al., 1998; Rumelhart and McClelland, 1987; Werbos and John, 1974], perceptrons are combined in a minimum of three layers. Each neuron in a layer is connected to all neurons in the next layer, making MLPs fully connected networks. The structure is shown in Figure 3.3, and the layers are an input layer, a hidden layer, and an output layer. When there are two or more hidden layers, the network is called an ANN. Contemporary ANNs still have nodes, but each node is no longer a perceptron. Remember that a perceptron can only output a binary value. In practice, this makes tuning the weight and bias parameters challenging because even a small change in the weights and biases can drastically change the output from 0 to 1 or vis versa. It is more optimal to have a small change in the output when there is a small change in the weights and biases. This is why activation functions are now used instead at each node. They output a value that is typically between 0 and 1 or between -1 and 1.

ACTIVATION FUNCTIONS An activation function is a non-linear function that maps an input to an output value between 0 and 1 or -1 and 1. Common activation functions are shown in Figure 3.4 and include: sigmoid, TanH, ReLU [Nair and Hinton, 2010], Leaky ReLU [Maas et al., 2013], ELU [Clevert et al., 2015], and SoftPlus [Glorot et al., 2011]. In practice, ReLU [Nair and Hinton, 2010] is a commonly used activation function.

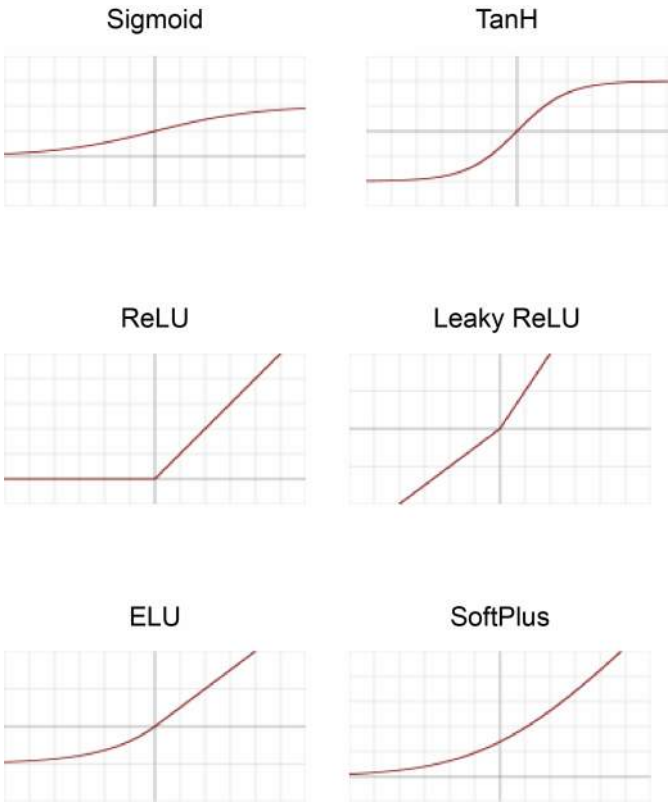


Figure 3.4.: Sigmoid, TanH, Rectified Linear Unit (ReLU) [Nair and Hinton, 2010], Leaky ReLU [Maas et al., 2013], Exponential Linear Unit (ELU) [Clevert et al., 2015], and SoftPlus [Glorot et al., 2011] activation functions. Graphs from Wikipedia [2023].

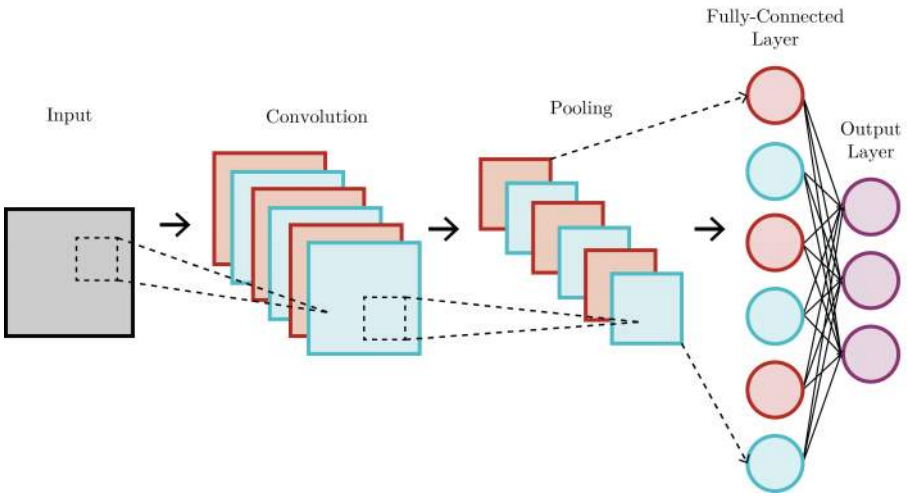


Figure 3.5.: Convolutional Neural Network model [Bengio and Lecun, 1997].

Each of the three different layer types in an ANN uses a different activation function. The input layers use a linear function. The hidden layers typically use ReLU [Nair and Hinton, 2010] and all nodes in the hidden layers use the same activation function. The output layer usually uses a different activation function and this can vary based on the application.

LEARNING At the connection between nodes, the output of the previous layer is multiplied by a weight and a bias is added. The result is the input for the next layer of nodes. ANNs learn by changing the weights and biases based on comparing the error in the output against the expected result. The training process is carried out through a process called backpropagation, a supervised learning technique.

BACKPROPAGATION Backpropogation [Rumelhart and McClelland, 1987] is an algorithm that starts at the last layer of the ANN and works backwards one layer at a time to determine changes in weights and biases. A loss function compares the predicted output values with the target values. Using the chain rule, the algorithm calculates the gradient of the loss function with respect to each weight and bias. There are options of which loss function to use and the selection also depends on whether the model is used for regression or classification.

3.4. Convolutional Neural Networks

CNNs [Bengio and Lecun, 1997] shown in Figure 3.5 are a class of ANNs which are specifically used for analyzing images, and similar principles have been extended to the analysis of 3D objects. CNNs regularize by taking advantage of patterns in the data. Each hidden layer detects increasingly complex patterns. They then assemble these complex patterns into smaller and simpler patterns in their filters, also known as kernels. Just like ANNs, CNNs have an input layer, a number of hidden layers and an output layer. Unlike the general ANN architecture discussed in Section 3.3, CNNs have convolutional layers, pooling layers, and fully connected layers.

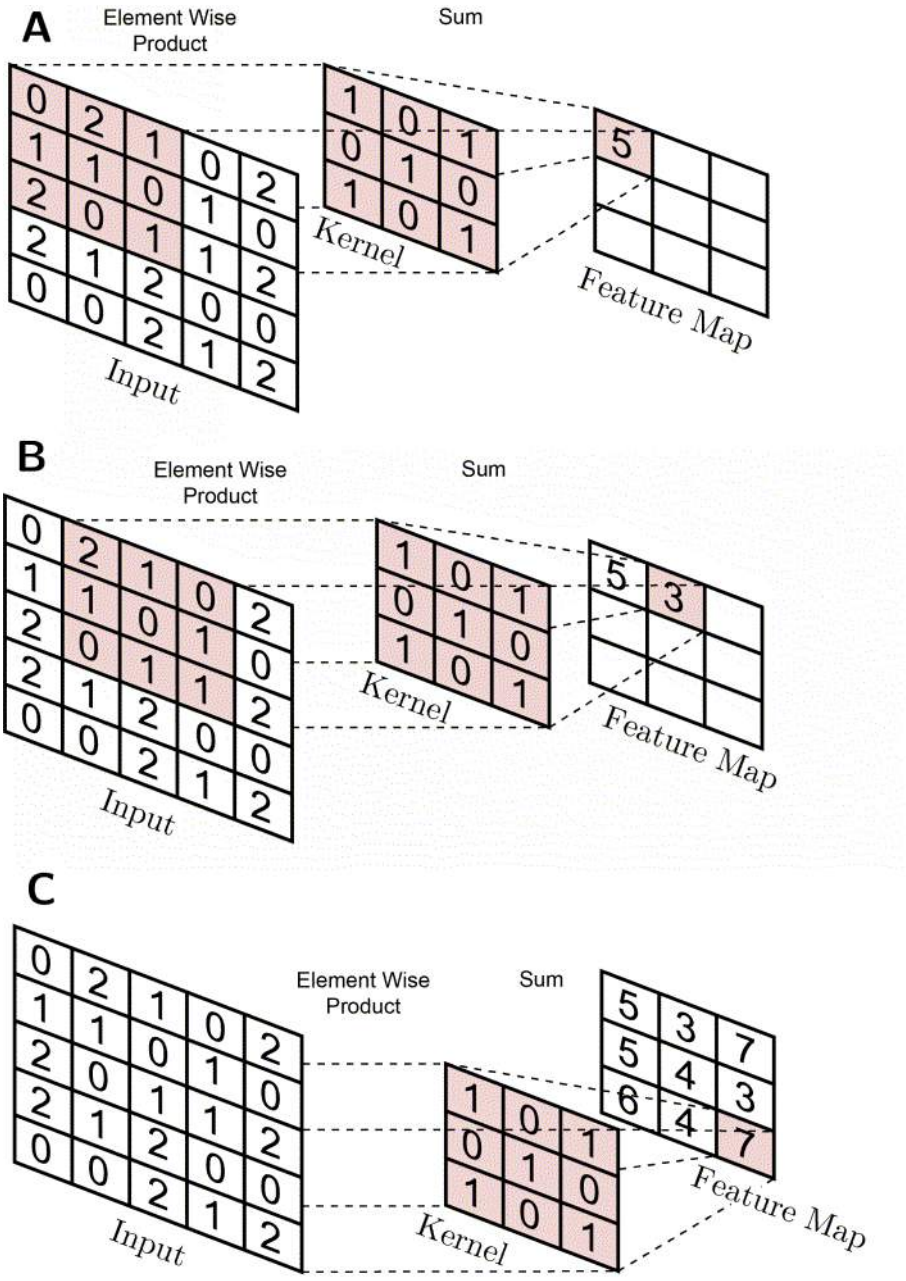


Figure 3.6.: An example of convolution operation in (A) through (C). The kernel size is 3 x 3, there is no padding, and the stride is 1.

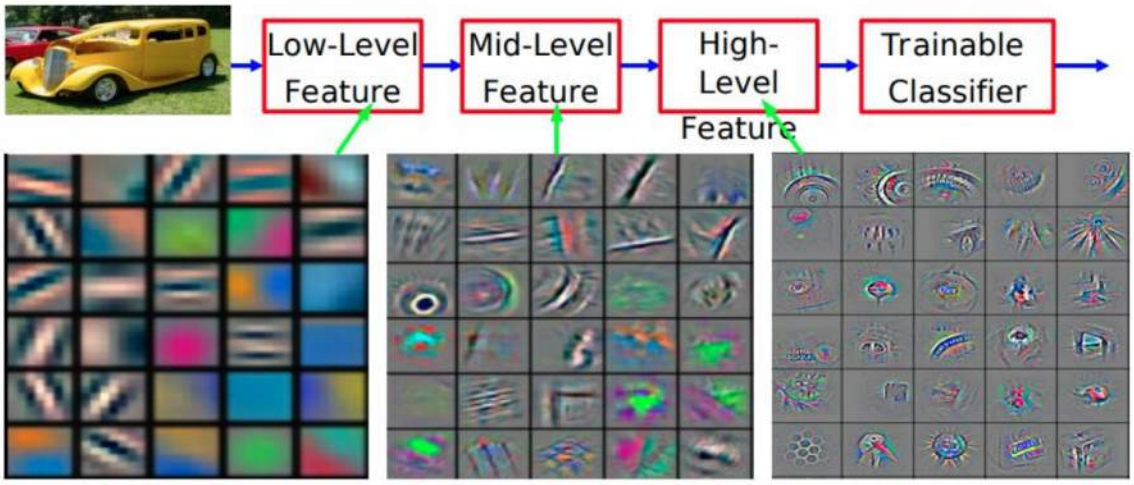


Figure 3.7.: Examples of features that a CNN learns and how these features become more detailed as the network is deeper. Image by Ibrahimli [2022] created using images from Zeiler and Fergus [2013].

CONVOLUTIONAL LAYERS Convolutional layers abstract the input (usually images) into a feature map which shows the locations and strengths of features in that input. The CNN performs the convolution by taking the dot product of the input data matrix with a kernel, sometimes also called a filter, as shown in Figure 3.6. The kernel is a 2D array of weights. The kernel is smaller than the input and is systematically moved across the input so that the feature can be detected anywhere in the image. Each value of the feature map is then passed through a non-linearity like ReLU [Nair and Hinton, 2010]. A convolutional layer learns between 32 and 512 kernels, allowing it to learn specific details of the training data. Multiple convolutional layers allow for the features learned to be higher-level features as the layer is located further down in the network. The higher-level features are larger and more descriptive, as can be seen in Figure 3.7. For example, when learning how to recognize a car, the first convolutional layer may learn a low-level feature, such as a kernel that identifies that the black color of the tire is next to the background color of the image. The third convolutional layer may then learn a high-level feature like a kernel that identifies a quarter of the shape of the wheel.

POOLING LAYERS In 2D applications, a pooling layer follows a convolutional layer and is used to reduce the dimensionality of the data. This is done by taking the outputs of a cluster of neurons of one layer and combining them as an input into a single neuron in the following layer. Pooling can be local or global and uses one of the two common types of pooling, max pooling [Yamaguchi et al., 1990; Ciregan et al., 2012], or average pooling. Local pooling combines small clusters, and global pooling combines all neurons of the feature map. Pooling layers are not used when working with 3D geometry.

FULLY CONNECTED LAYERS To classify inputs, the flattened matrix passes through a fully connected layer. The fully connected layers are the same as in a traditional MLP.

THREE-DIMENSIONAL CONVOLUTIONAL NEURAL NETWORKS 3D CNNs work similarly with a few small differences. Instead of using 2D kernels 3D CNNs use 3D kernels and the kernels are applied by moving the kernel in three directions. Additionally, as mentioned previously, pooling layers are not used.

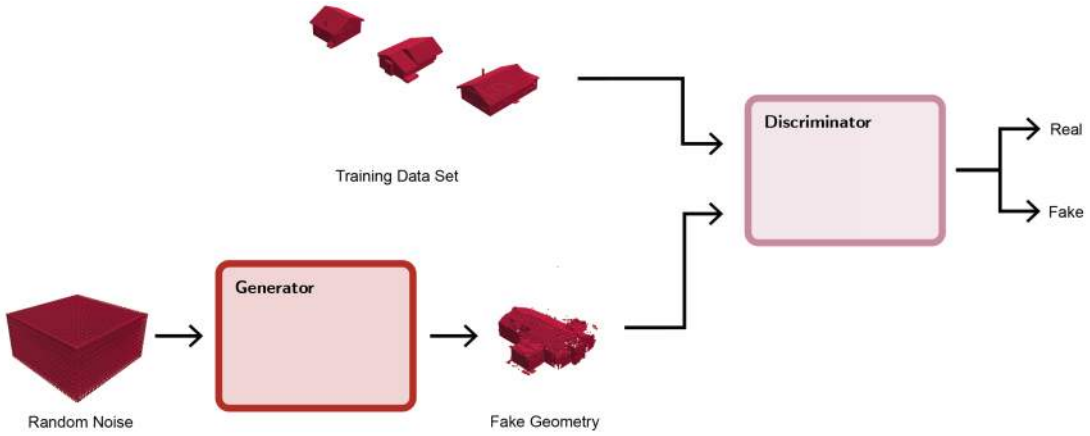


Figure 3.8.: An overview of a typical structure for a GAN [Goodfellow et al., 2014] model.

3.5. Generative Adversarial Networks

A GAN [Goodfellow et al., 2014] is an architecture for ANN models where two different ANNs are trained to contest against each other in a zero-sum game. A zero-sum game describes a game where the advantage won by one player is equal to the loss of the other player [CambridgeDictionary, 2022]. The *generative network* generates candidates based on the data set, while the *discriminatory network* evaluates if the candidate is generated or is from the data set. In other words, the discriminator determines if the input is *real*, realistic enough to be part of the data set, or *fake*, not realistic enough and a generated image or object. Figure 3.8 demonstrates a typical GAN structure. The generator has only one input, randomized noise, and outputs fake geometry. The discriminator has two inputs, the training data set and the fake geometry created by the generator. The discriminator learns characteristics of the data set and uses this information to output the probability that the fake geometry is real or fake. The generator is rewarded when it fools the discriminator and the discriminator is rewarded when it correctly classifies the geometry. Some examples of GANs include text to image generators like DALL-E with images shown in Figure 3.9 (A) [OpenAI, 2021], this person does not exist with images shown in 3.9 (B) [ThisPersonDoesNotExist, 2023], and sketch to image generators like Picsart SketchAI [Picsart, 2023].

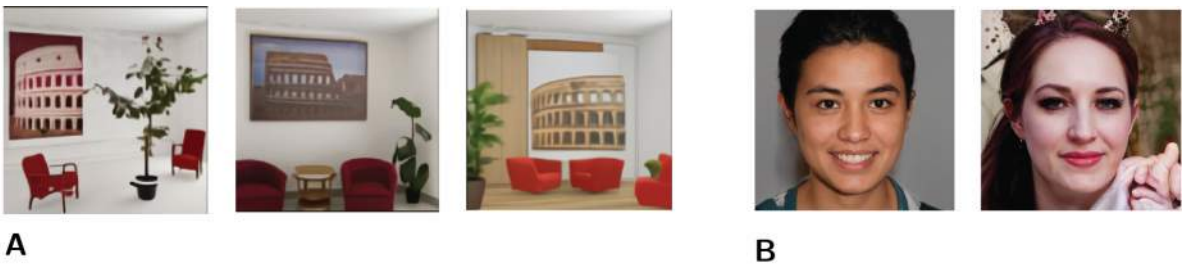


Figure 3.9.: (A) Three images generated by OpenAI [2021] for the prompt "A living room with two red armchairs and a painting of the Colosseum. The painting is mounted behind an indoor plant." (B) Two images of people generated by This Person Does Not Exist [ThisPersonDoesNotExist, 2023].

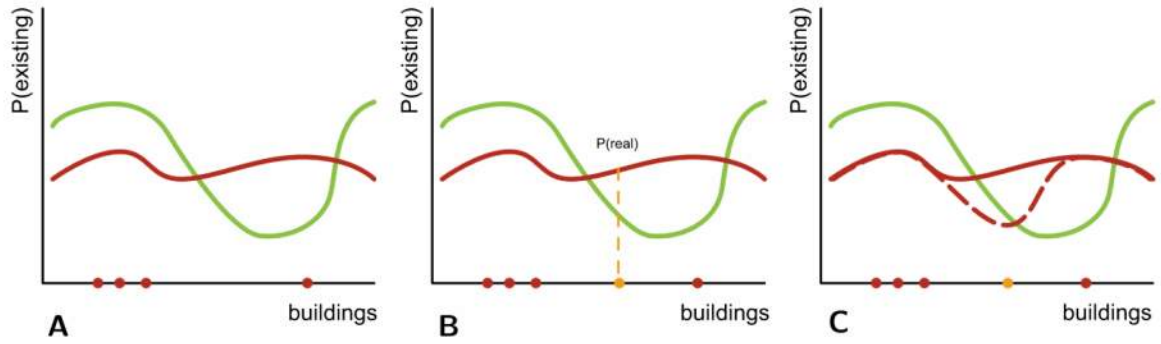


Figure 3.10.: There exists a distribution (green line) of buildings that exist. The discriminator uses its assumption of this distribution (red line) to evaluate if data samples produced by the generator (yellow) are real or fake and then adjust its distribution based on the new data point (dashed red line).

A GAN game is defined by a probability space with the game state (Ω), the probability measure of the training data (μ_{ref}), and two players which are the *generative network*, called the generator, and the *discriminative network*, called the discriminator. The generator's strategy is to generate new outputs, which is noted by the strategy set $P(\Omega)$ which is the set of all probability measures (μ_G) on (Ω). The discriminator's probability measure (μ_D) is the state of the game (Ω) as it impacts the probability measure of the state being real or fake $P[0,1]$. The GAN game is a zero-sum game with the objective function:

$$L(\mu_G, \mu_D) := \mathbb{E}_{x\ \mu_{ref}, y\ \mu_D(x)}[\ln y] + \mathbb{E}_{x\ \mu_G, y\ \mu_D(x)}[\ln(1-y)] \tag{3.2}$$

The goal of the generator is to minimize the objective, and the discriminator aims to maximize the objective. The goal of the generator is to approach $\mu_G \approx \mu_{ref}$, that is, to match the reference distribution as closely as possible with its output distribution. The discriminator outputs a value close to 1 when the input appears to be from the reference distribution and a value close to 0 when the input appears to be from the generator distribution.

This objective function is also called a loss function and the standard loss function for GANs as described by Goodfellow et al. [2014] is the Min-Max loss function. To better understand how the training with the Min-Max Loss function works, note the following scenario. As shown in Figure 3.10 (A), there exists a distribution, represented by the green line, of all buildings that exist. Although it is not possible to know this distribution, there are data points available that show which buildings are more likely to exist. The discriminator has assumed what this distribution is based on the data set it has seen. The assumed distribution of the discriminator is represented by the red line. When the generator creates a new building geometry, it is located somewhere along this distribution as shown in Figure 3.10 (B). When presented this new sample, the discriminator uses its distribution to determine whether the sample is real or fake and provides this information to the generator. The generator then tries to improve its next output based on the feedback. From the feedback, it now knows that samples near the geometry that was labeled 'fake' will most likely also be fake. It keeps trying different geometry options to try to create geometry that gets identified as real. This is how the generator learns. The discriminator also takes the actual value of this new data point to adjust its distribution as shown in Figure 3.10 (C). This is how the discriminator learns.

Thus, training of a GAN is indirect as the discriminator trains to be able to distinguish whether the output of the generator is realistic given the training data. Therefore, the generator is not trained to minimize a loss function, but rather to fool the discriminator. The two models are trained together to ensure an equilibrium. This is contrary to most DL models which usually train with a loss function and continue training until the loss function converges. This also presents some challenges, as there is no objective way to assess the progress of the training or the quality of the model, both relative and absolute quality from the loss [Salimans et al., 2016].

Another challenge when training GANs is that the learning rate between the discriminator and the generator tends to be different. The discriminator tends to learn much faster than the generator. This is possibly because it is more difficult for the generator to generate objects in a 3D voxel space than it is for the discriminator to determine whether the object is real or fake [Goodfellow et al., 2014]. Because the discriminator trains faster, it will always label inputs as fake with high confidence. This does not provide enough feedback for the generator to improve. To mitigate this challenge, adaptive training strategies may need to be employed which keep the networks learning at a similar pace. There have been different loss functions that have been developed that can help stabilize training.

3.6. 3D Generative Adversarial Networks

When focusing on the application of GANs in 3D, there are some additional differences and considerations. To be able to apply existing 3D GAN architectures to a new problem type, it was important to understand how the state-of-the-art systems work.

Wu et al. [2016] developed the architecture for 3D GAN which was used to generate novel 3D objects. 3D objects have long been explored within computer vision, but research about generating 3D objects has been more limited. Most methods focus on generating novel shapes based on the retrieval and combination of selected parts from a database of options or using supervised training techniques [van Kaick et al., 2011; Tangelder and Veltkamp, 2004; Carlson, 1982]. Wu et al. [2016] pioneered research on generating objects from voxels through learned features instead of using predefined parts with an unsupervised training method.

NETWORK ARCHITECTURE As introduced by Goodfellow et al. [2014] and discussed in Section 3.5, GANs consist of a generator and discriminator. In the case of 3D GAN described by Wu et al. [2016], the generator, (G), maps a 200-dimensional latent vector, (z) that is randomly sampled from a probabilistic latent space to a 64 x 64 x 64 cube. This cube represents an object ($G(z)$) in the 3D voxel space. The discriminator (D) outputs a confidence value ($D(x)$) that describes whether an input of a 3D object (x) is real or synthetic. Wu et al. [2016] followed Goodfellow et al. [2014] and used binary cross-entropy as the classification loss. Therefore, the overall adversarial loss function, the loss function of the entire architecture that combines the loss of the generator and the discriminator, as described by Wu et al. [2016] is

$$L_{3DGAN} = \log D(x) + \log(1D(G(z))), \tag{3.3}$$

where x is a real object in a 64 × 64 × 64 space, and z is a randomly sampled noise vector from a distribution $p(z)$. Additionally, each dimension of z is an independent and identically distributed (IID) uniform distribution over [0, 1].

Wu et al. [2016] also implemented a number of key changes that were proposed by Radford et al. [2016]. These changes allowed for the scaling of GANs using CNN architecture. Radford et al. [2016] discuss three changes resulting from existing research and two additional changes. They published

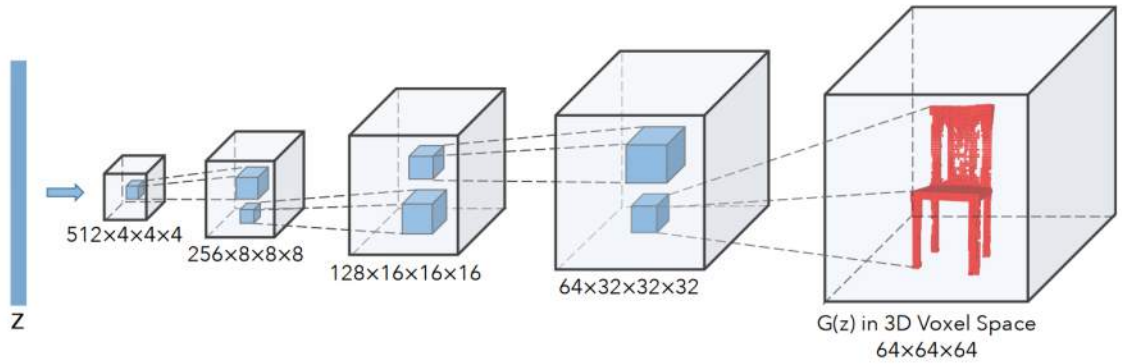


Figure 3.11.: The architecture of the generator in 3D GAN. The discriminator mostly mirrors the generator. Image by Wu et al. [2016]

the following guidelines for stable deep convolutional generative adversarial network (DCGAN) implementation:

- Replace any pooling layers with strided convolutions in the discriminator and fractional strided convolutions in the generator [Springenberg et al., 2014]. Strided convolutions are convolutional layers where the stride is equal to two or more, and fractional-strided convolutional layers pad each pixel by placing a row and column of zeros between each pixel in an image. This means that when the kernel moves across the image, the stride is less than 1 which basically decreases the size of the kernel. Fractional-strided convolutions are also called transposed convolutions.
- Use batch normalization (batch norm) in both the generator and the discriminator [Ioffe and Szegedy, 2015]. batch norm normalizes each of the layers' inputs by recentering and rescaling them.
- Remove fully connected hidden layers for deeper architectures [Mordvintsev et al., 2015].
- Use ReLU [Nair and Hinton, 2010] activation in generator for all layers except the output, which uses Tanh.
- Use Leaky ReLU [Maas et al., 2013] activation in the discriminator for all layers [Radford et al., 2016].

Based on these principles, Wu et al. [2016] designed an all-convolutional ANN architecture that generates 3D objects. The structure is shown in Figure 3.11.

In this architecture, the generator consists of five volumetric fully convolutional layers. The convolutional layers are of kernel size 4 x 4 x 4 with strides of 2. The batch normalization and ReLU [Nair and Hinton, 2010] layers are in between the convolutional layers. The final layer is a Sigmoid layer. The discriminator mirrors the generator, but instead of using ReLU [Nair and Hinton, 2010] layers, it uses Leaky ReLU [Maas et al., 2013] layers. The network does not contain pooling layers or linear layers. [Wu et al., 2016]

As discussed in Section 3.5, the discriminator tends to learn much faster than the generator which can lead to the discriminator always labeling inputs as fake with a high confidence, therefore not providing enough feedback for the generator to improve. The adaptive training strategy employed by Wu et al. [2016] includes updating the discriminator only if the accuracy in the last batch is not greater than 80%. Additionally, the learning rate of the generator was set at 0.0025 and the learning rate of the discriminator was set at 10^{-5} and the authors used a batch size of 100. The optimization method used was ADAM [Kingma and Ba, 2014] with $\beta = 0.5$. ADAM is a first-order gradient-based optimization of stochastic objective functions [Kingma and Ba, 2014]. Gradient-based optimization

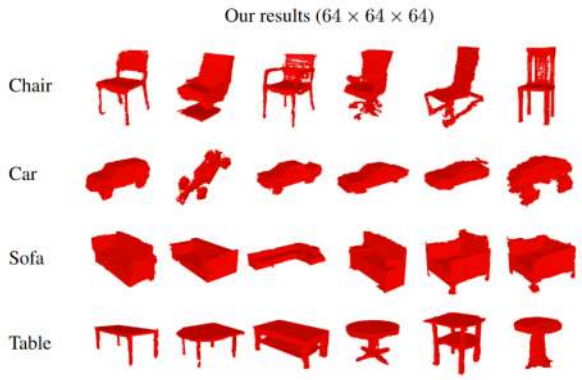


Figure 3.12.: A selection of 3D objects generated by Wu et al. [2016] with their 3D GAN model.

methods are methods in which the search directions are defined by the gradient of the function at the current point.

TRAINING STABILITY GANs are notoriously challenging to train because they are very unstable. As discussed in previous sections, the discriminator and generator need to find a balance with each other, however, the tasks each is completing vary in complexity. It is significantly easier for the discriminator to learn if geometry is real or fake than it is for the generator to create real-looking geometry to fool the discriminator. When the discriminator outperforms the generator too quickly, neither network continues to learn, and training fails. This issue is referred to as vanishing gradients. The gradients of the loss function that provide information to the generator about how to learn are so small that the generator cannot learn.

A second problem that can occur concerns the output of the generator. It is expected that, for each new random input the generator receives, it produces a new and different output. Sometimes, however, this does not happen. This common problem with GANs is called mode collapse. Mode collapse occurs when the generator begins to produce only the same geometry, even when different random inputs are provided. The problem occurs because the generator learns to create one output that fools the discriminator and then produces only that one output. The discriminator and generator get stuck in a loop where the generator is over-optimizing to the specific discriminator.

These two common GAN issues are important to watch out for, recognize, and address if they arise during training. Since they are common problems, they have been well researched. There are existing strategies that involve adjusting the loss functions and that can address these failures.

GENERATING NOVEL 3D OBJECTS Wu et al. [2016] used the 3D GAN architecture that they developed and trained a model that generates novel 3D objects. The outputs of their architecture are shown in Figure 3.12. The researchers trained a separate 3D GAN according to each category of objects such as chair, car, and table. Based on this approach, it may be necessary to also train separate models for each type of building. This will be an aspect to explore in future research, as building types still have more overlap of features than objects of different classes such as those used in Wu et al. [2016]. However, this thesis will focus on learning one building typology, which is selected to be single family houses. The model trained by Wu et al. [2016] sampled 200-dimensional vectors following an IID uniform distribution over [0, 1] and the authors rendered the largest connected component of each of the generated objects. The result produced higher resolution objects that had more detailed geometry compared to a model synthesizing objects from a probabilistic space [Wu et al., 2015] and a volumetric autoencoder [Girdhar et al., 2016; Sharma et al., 2016].

One concern that needs to be addressed when generating 3D objects is if the model is memorizing the objects that are already in the training data. The method implemented by Wu et al. [2016] to confirm

that this was not happening was to compare the models to their nearest neighbors in the training set. The output of the last convolutional layer in the discriminator was used to retrieve features. The authors concluded that the generated objects are similar to the nearest examples in the training set, but not identical. This will also be a concern that needs to be addressed in my own research to ensure that the generated buildings are not the same as the training set. The methods discussed in Section 1.6 will be used to address this concern.

3.7. Conclusions

As discussed in the above review of related work, research with a focus on DL has been conducted primarily in the computer science field. There has been some work within the AEC field, but this research focuses on simulations and topology optimization. One study translated GAN applications to generate building geometry based on sky scrapers from New York City [Newton, 2019]. Unfortunately, the geometry space was very small and minimal detail is provided to be able to reproduce the study. Within DL, GANs have the potential to be applied more widely to the generation of 3D geometry in the AEC industry. Research published on 3D GANs shows that applying similar principles to generating novel 3D buildings is feasible. Based on the literature review, key gaps in existing research include generating high-resolution geometry and applications on building geometry. This thesis aims to address these shortcomings by developing a new GAN architecture to generate high resolution 3D building geometry.

4. Application and Limitations

4.1. Deep Learning for Design

As mentioned, DL can help solve design problems in a more time-efficient manner by allowing algorithms to train themselves to perform tasks in a human-like manner. Some of the advantages of deep learning include the ability of neural networks to discover hidden patterns in data and the fact that training can use a variety of data types. Additionally, after a network is trained, it can complete the task over and over to the same level of detail and in a significantly shorter time than it would take a person to complete the same task. All of these features make it well suited to be applied to design tasks. Existing surveys of DL applications within architecture and design [Newton, 2019; Wu et al., 2022] and the literature review in this thesis have concluded that there is a lack of research regarding GAN applications to the generation of 3D building geometry. Despite the lack of existing research, these methods show promise for the application of design.

There are a number of advantages to using ML, but it is also important to carefully formulate the task to which the algorithm is applied. With the potential benefits, there are also some disadvantages to keep in mind. Because DL models train themselves, they generally require large data sets for training [LeCun et al., 2015] so these data sets need to be available. When training themselves, algorithms also make decisions without providing an explanation for the reasoning of reaching the result. Despite these challenges, deep learning allows professionals in the AEC industry to go beyond automating menial and tedious tasks. It presents the opportunity to automate design tasks without having to define all of the parameters.

An important motivation for automating design is to raise the bar for providing good design. Currently, well-designed buildings are not evenly distributed. And with the housing crisis, the risk that 'good design' will become a scarce commodity is even greater. When clients can pay more money for an architect, their project receives more skilled design attention. Those with fewer resources are left with unsustainable solutions. Other industries have found methods to automate creative tasks, and the AEC industry can draw on these examples. It is not necessary to limit the experts' attention to a few projects when DL can model the expert team's knowledge and generate well-designed buildings for everyone.

4.2. Innovation to Solve the Housing Problem

Industry professionals frequently describe automation as a way to let machines do the repetitive tasks to free up time and allow architects and designers to do the creative tasks which they are best at. In 2016, Kevin Kelly [2016] changed the conversation. He discusses "our most important mechanical inventions are not machines that do what humans do better, but machines that can do things we can't do at all." [Kevin Kelly, 2016]. This leads to the question:

What can machines do for the housing crisis that people cannot?

When contemplating the current housing crisis, there are a number of aspects that people are not able to address, or are not able to address fast enough. DL models could:

- deliver a code-compliant building design without an architect
- provide safe housing solutions in hours rather than months or even years
- increase productivity in the AEC industry
- efficiently address the increasing housing needs

As generative DL models have gained popularity in recent years, these methods provide innovative solutions to a long-established problem.

4.3. Optimal Building Typology

Generating building geometry through ANN models is not feasible for all types of buildings. Iconic buildings, for example, are generally designed by world-renowned architects, and owners pay a premium to have a well-known name attached to their project. Multifamily housing, on the other hand, requires fast solutions to keep up with the current demand. It also requires good design solutions that provide code-compliant, safe, and healthy places to live. This is why multi-family housing is the typology that provides the most opportunity for algorithmic generation and optimization. Multi-family housing:

- is a sustainable solution to the housing crisis
- can provide low-income and affordable housing options
- provides opportunity for construction standardization
- is not an iconic building typology
- has repeating patterns that can be learned by an algorithm

These characteristics and the benefit that automation can provide for this typology make midsized, multifamily housing the ideal typology for automation.

Based on detailed research of data set options, there were no data sets that contained enough buildings of the desired typology. Because the thesis focuses on a DL model, the data set must be substantial with a minimum of 100 building geometries and ideally a few hundred building geometries. Due to the lack of available data, this thesis will instead train on building geometries that represent primarily single-family homes. This is discussed in further detail in Chapter 5.

When enough data is available for the optimal typology, then it is possible to implement the steps documented in this thesis and the algorithms written for this thesis with the new training data set.

5. Training Data Set

To select a data set, a critical consideration was the level of detail (LOD) of the building models. The LOD determines what building features are visible in a building model. As shown in Figure 5.1, LOD 0 describes a building in 2D as a massing outline, LOD 1 describes the 3D massing of a building, LOD 2 describes the envelope of the building and the form of the roof, and LOD 3 describes a building with its features such as windows and doors. Since the goal of this thesis is to generate building geometry, the building form that is generated should be more detailed than its initial massing. Recognizable building features include windows, doors, and a more developed building form, which can include changes in the facade and roof overhangs. Models that have these characteristics have an LOD of 3.1. This also means that the data set must have a minimum LOD of 3.1 because the network will need to train on a data set that contains the same level of information.

Another requirement for a data set for any DL task is the size. THe data set needs to contain a minimum of 100 models, but the larger the data set, the better it is for training. After determining these primary requirements and searching for available data sets, it was determined that one data set met the LOD requirements and was large enough to train a deep learning model.

The data set used for training is a large-scale data set of just under 2,000 building models called BuildingNet v0.1 [Selvaraju et al., 2021]. The data set was developed to train a graph ANN that is used to label building meshes by analyzing relationships between spatial and structural geometric primitives [Selvaraju et al., 2021]. The data set is well suited for this application and provides the opportunity for other applications. Because the data set was recently released, the research will use version 0.1 with the opportunity to retrain the GAN model once version 1 of the data set is released. After detailed review of the BuildingNet v0.1 data set [Selvaraju et al., 2021], there are a few key changes that must be made to prepare the data set for the application of generating buildings. Section 5.1 discusses the content of the existing data set and the changes that must be made in more detail.

5.1. 3D Model Data Set

BuildingNet v0.1 [Selvaraju et al., 2021] is a data set released in 2021 that consists of 1,938 buildings with consistently labeled exteriors. Buildings were selected from SketchUp Warehouse, labeled with exterior classes, and labeled with building typology classes. The data set contains for each building the mesh (.OBJ file format), rendering textures (.jpg file format and .mtl file format), labels for each mesh surface (.JSON file format), a point cloud consisting of 100,000 points generated from the mesh (.ply file format), and labels for each point (.JSON file format). The label files includes a nominal classification for each point consisting of 31 different classes: undetermined, wall, window, vehicle, roof, plant/tree, door, tower/steeple, furniture, ground/grass, beam/frame, stairs, column, railing/baluster, floor, chimney, ceiling, fence, pond/pool, corridor/path, garage, dome, road, entrance/gate, parapet/merlon, buttress, dormer, lantern/lamp, arch, awning, shutters, and balcony/patio. The 3D models are also classified by what will be called typology and building type. The five typologies are commercial, military, public, religious, and residential. The fifteen building types are castle, cathedral, church, city hall, factory, hotel building, house, monastery, mosque, museum, office building, palace, school building, temple, and villa. Table 5.1 lists each label and indicates the number of 3D models that contained this label before and after cleaning. Table 5.2 lists each building typology and building

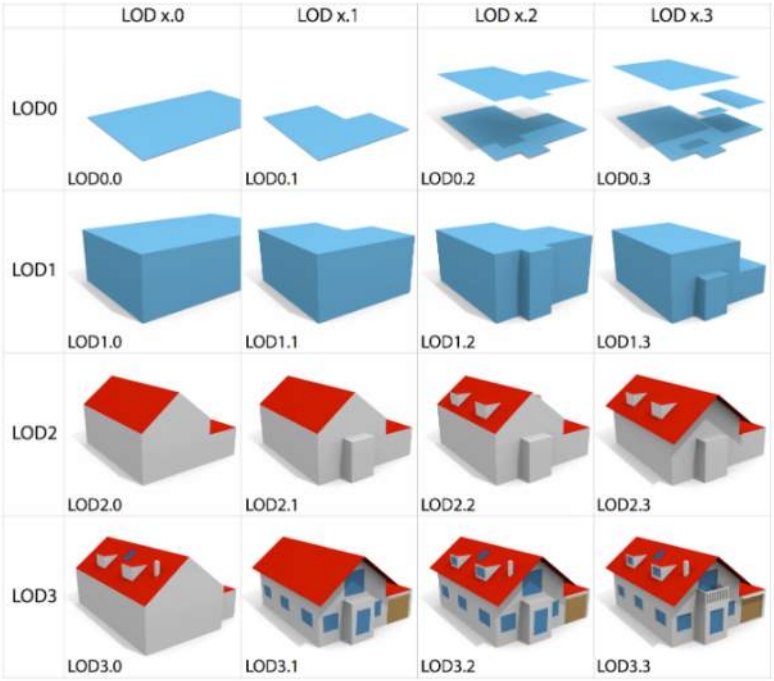


Figure 5.1.: A visual representation of the levels of detail and their sub-levels. Image from Biljecki et al. [2016].

type and indicates the number of 3D models that belong to that building typology and building type before and after cleaning.

After reviewing the data set, several key attributes were identified that are not suitable for training a GAN model to generate buildings. These attributes needed to be rectified in order to use the data for this thesis:

- The 3D models are different scales and have been normalized so that the maximum dimension of each 3D model is equal to one unit regardless of the real-world dimension of the building. There are some buildings where the maximum dimension is not normalized to one unit but the maximum dimension is instead smaller than one unit.
- The 3D models contain geometry that is related to the site. For example, some models contain plants, vehicles, and roads.
- The labels cover a wide range of classes, but all 3D models also contain 'undetermined' points. For some 3D models, the 'undetermined' points belong to one of the labeled classes. For example, some 3D models have walls labeled as 'undetermined.'
- The size of the voxels affects the resolution and detail of the generated 3D models and also depends on the available memory.

	before cleaning	after cleaning
0 undetermined	1,938	1,247
1 wall	1,930	1,247
2 window	1,688	1,247
3 vehicle	268	202
4 roof	1,860	1,247
5 plant, tree	400	269
6 door	1,358	1,247
7 tower, steeple	389	154
8 furniture	301	236
9 ground, grass	859	558
10 beam, frame	385	307
11 stairs	687	487
12 column	523	370
13 railing, baluster	362	293
14 floor	871	599
15 chimney	382	330
16 ceiling	591	463
17 fence	284	203
18 pond, pool	320	213
19 corridor, path	521	394
20 balcony, patio	458	364
21 garage	359	300
22 dome	172	53
23 road	356	249
24 entrance, gate	124	84
25 parapet, merlon	173	108
26 buttress	60	20
27 dormer	120	99
28 lantern, lamp	62	54
29 arch	81	41
30 awning	68	49
31 shutters	52	43
total	1,938	1,247

Table 5.1.: Total number of 3D models that contain each type of label.

	before cleaning	after cleaning
Building Typology		
commercial	126	51
military	65	16
public	57	20
religious	447	183
residential	1,243	977
total	1,938	1,247
Building Type		
castle	68	19
cathedral	34	11
church	271	133
city hall	16	5
factory	10	7
hotel building	31	12
house	780	629
monastery	7	3
mosque	71	23
museum	34	16
office building	110	53
palace	44	20
school building	27	20
temple	45	11
villa	390	284
total	1,938	1,247

Table 5.2.: Total number of 3D models that belong to each typology and building type.

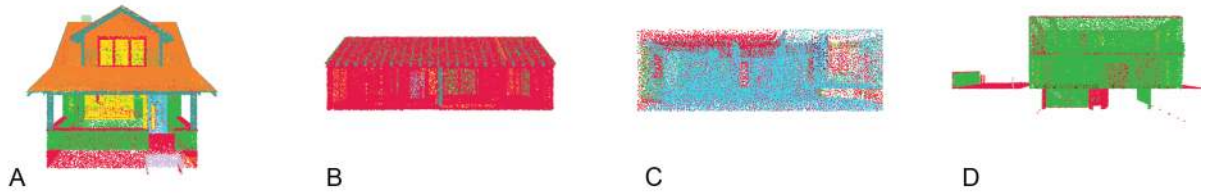


Figure 5.2.: Examples of the building point cloud visualizations with each label represented by a color. (A) shows an acceptable 3D model and (B), (C), and (D) show examples of 3D models that were removed because they contained too many 'undetermined' points, incorrectly labeled points, or unclear building features.

5.2. Cleaning Data Set

To analyze the necessary cleaning steps, each building model was visualized as a point cloud with the points labeled and color coded according to their class label. From this analysis, the following steps were observed as necessary to clean the data set and then implemented.

1. 3D models with an acceptable number of labels and generally good accuracy of the labels, as shown in Figure 5.2 (A) should remain in the data set.
2. 3D models with a significant amount of points labeled as 'undetermined' (indicated by the color red) as shown in Figure 5.2 (B) should be removed. This was evaluated by counting the number of 'undetermined' labels that were present in each 3D model. Then the top 10% of the data was calculated and the upper limit (equal to the top 10% plus 1.5 times the top 10 percent minus the bottom 10%) was calculated. Any 3D models that contained more undetermined labels than the upper bound were removed from the list.
3. 3D models with a significant number of incorrectly labeled points, for example, walls labeled as doors as shown in Figure 5.2 (C), need to be removed. This was evaluated by counting the number of points for each of the remaining labels that were present in each 3D model. Then the top 10% of the data was calculated and the upper limit (equal to the top 10% plus 1.5 times the top 10 percent minus the bottom 10%) was calculated. Any 3D models that contained more labels than the upper bound were removed from the list.
4. 3D models where the building characteristics are unclear or missing, as shown in Figure 5.2 (D) must be removed. This was evaluated based on whether 3D models contained at a minimum the labels 'undetermined', 'wall', 'window', 'roof', and 'door.' 3D models that did not contain all five of these labels were removed.

An overview of this process is described in Algorithm 5.1. The 691 3D models that met criteria 2, 3, and 4 were removed from the data set, leaving 1,247 3D models. These 1,247 3D models are then used for the pre-processing steps. Table 5.1 lists each label and shows the difference in the number of 3D models that contained this label before and after cleaning. Table 5.2 lists each building typology and building type and indicates the difference in the number of 3D models that belong to that building typology and building type before and after cleaning.

Algorithm 5.1: An overview of how the models are evaluated for incorrect labels.

Input: the data set, D, containing point clouds, *pcd*
Output: the revised data set containing remaining point clouds, D_r

labeltrackingtable is an empty table where each model is represented by a row and each column represents a label
for *pcd in D* **do**
    load label array L ;
    for *label in L* **do**
        ;
    count total instances of *label* ;
    save to *labeltrackingtable*[*pcd*, *label*] ;
for *column c in labeltrackingtable* **do**
    d_1 = calculate the lowest decile, the 10th percentile ;
    d_9 = calculate the highest decile, the 90th percentile ;
    $upperlimit = d_9 + 1.5 * (d_9 - d_1)$
for *pcd in D* **do**
    if *labeltrackingtable[pcd, label]* $>$ $upperlimit$ **then**
        remove *pcd* from D
    if *labeltrackingtable[pcd, wall] == 0* ***or*** ;
    labeltrackingtable[pcd, window] == 0 ***or*** ;
    labeltrackingtable[pcd, roof] == 0 ***or*** ;
    labeltrackingtable[pcd, door] == 0 **then**
        remove *pcd* from D
return D

Figure 5.3.: A number of examples of the house building type (top row) and villa building type (bottom row).

5.3. 3D Model Pre-Processing

Based on the analysis in Section 5.1, the remaining 3D models needed to be pre-processed to meet the needs of the research. This generated a revised data set consisting of pre-processed 3D models that are a subset of the BuildingNet v0.1 [Selvaraju et al., 2021] 3D models which then meet the needs of training a GAN to generate 3D building forms. The steps for pre-processing the data set are outlined in the following sections, and the pre-processed data set is released with this research so it can be used for future research.

THREE-DIMENSIONAL MODEL TYPOLOGY SELECTION As discussed in Chapter 4, automating the design of buildings is not suited for unique or monumental building types. Therefore, the following building types were removed: castle, cathedral, church, city hall, factory, hotel building, monastery, mosque, museum, palace, school building, temple. After removing these building types, the following types remained: house, office building, villa.

Further analysis outlined in Chapter 4 explains that mid-sized multifamily housing with a height of 8-12 stories is the ideal typology for automation. However, there were not many data sets that contained this typology. Therefore, single-family homes were used for training. Based on the information recorded in Table 5.2, the BuildingNet v0.1 [Selvaraju et al., 2021] data set had many buildings that matched the single-family home typology and even after cleaning the data set, there were many 3D models left that matched the criteria. Therefore, the following building types remain in the revised data set: house and villa. Examples of these two building types are shown in Figure 5.3. After selecting the typology, 913 3D models remained in the training data set.

REMOVING SITE GEOMETRY The models that remained still contained unnecessary geometry. Geometries labeled as vehicle, fence, furniture, ground, gate, lighting, pool, road, and plant/tree were removed since these elements are either too detailed in scale or are part of the site, not part of the building. After removing these labels, it became clear that some 3D models also have site geometry

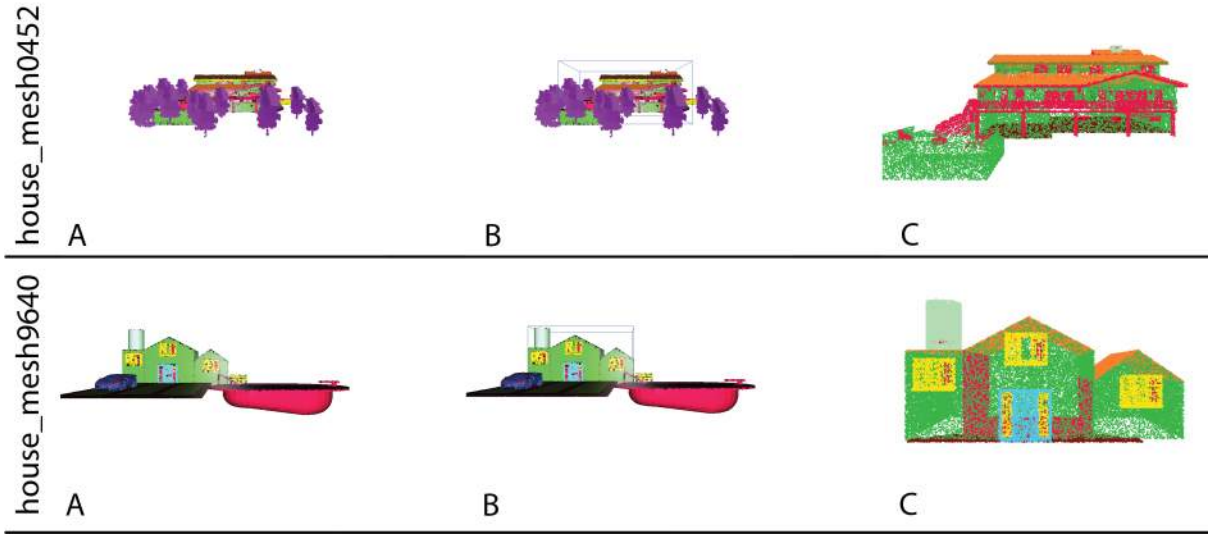


Figure 5.4.: (A) shows the original 3D model with each label represented by a different color. (B) shows the bounding box that was created around the wall, door, and roof geometry. (C) shows the resulting 3D model after removing the site geometry based on labels and based on geometry that was located outside the bounding box.

that is labeled 'undetermined' or site site that is incorrectly labeled as building geometry. To address this, all points of the 3D model belonging to the labels 'wall', 'door', and 'roof' were collected and a bounding box was placed around these points. After removing the site geometry that belonged to the labels mentioned above, any additional geometry located outside the bounding box was also removed. This helped to clean up most of the site geometry that was 'undetermined' and some of the site geometry that was incorrectly labeled, as shown in Figure 5.4.

THREE-DIMENSIONAL MODEL SCALING (not implemented) After reviewing the models, it became evident that the scale of the buildings was mostly normalized so that the maximum dimension of each 3D model was one unit. Some 3D models did not have normalized heights and instead had heights less than 1 unit. The range of sizes is shown in Figure 5.5. The first 3D model is a two-story house and it is smaller than 1 unit in width and height (shown) and depth (not shown). The second 3D model is a three-story house and it is 1 unit wide because the width is its largest dimension, and it is just under 1 unit tall. The third 3D model is an approximately 30-story building and is 1 unit tall. To use these 3D models for training, the height of the buildings should be normalized in relation to a set scale so that additional 3D models can be added to the data set. Furthermore, having a set scale would mean that all 3D models are scaled relative to each other.

There were multiple options for scaling the models that were reviewed, but due to time constraints, they were not implemented. Some methods were tested that were based on scaling the model according to known features such as door heights. Unfortunately, the resulting point clouds were not uniformly scaled because the doors were not consistently labeled. Scale information was also requested from the authors of the data set, but unfortunately this data was not available. Therefore, the most feasible route was determined to be downloading the original 3D models from Google SketchUp Warehouse and scaling 3D models this way. This method is outlined in detail below, but it was not implemented due to the time restraints of this thesis.

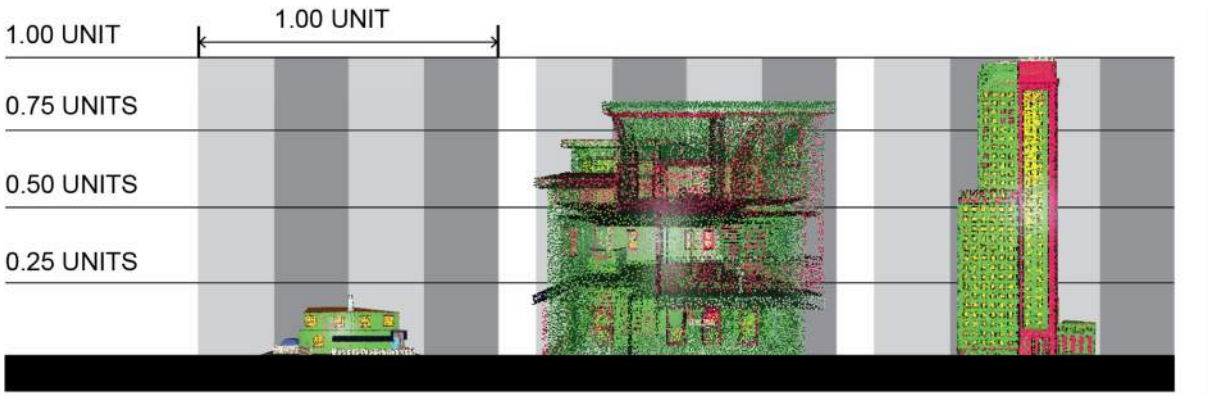


Figure 5.5.: The 3D models in the BuildingNet data set are mostly normalized to have the maximum dimension equal to 1 unit regardless of building height and some 3D models have a smaller maximum dimension.

In order to scale the 3D models, the following steps must be taken:

1. Download the original 3D model from Google SketchUp Warehouse
2. Convert the file type from a SKP file to an OBJ file
3. Load the original 3D model file
4. Create a bounding box around original 3D model
5. Load the point cloud from the data set
6. Create a bounding box around the point cloud
7. Scale the point cloud model up to original 3D model size based on the bounding box heights of each model
8. Save the scaled point cloud

The automated portion of these steps are summarized in Algorithm 5.2. At this time, downloading the original models from the SketchUp Warehouse and converting the file from SKP to OBJ needs to be done manually due to licensing requirements. Due to the large amounts of manual work involved in scaling more than 900 3D models, and considering the time limitations of this thesis, the data set that is released with this thesis does not contain scaled models.

5.4. Selecting Labels for Training

The data in the data set is also already formatted to include only the labels contained in all building models. These labels include 'wall', 'window', 'door', and 'roof'.

To select which labels could be used for further research, I analyzed the 31 existing labels. Of these labels, there are many labels that appear in very few 3D models as shown in Table 5.3. This means that there is not enough data to train models on these labels. Furthermore, some labels do not provide enough information. Since the 'undetermined' label encompasses a large collection of geometry and some geometry that is labeled 'undetermined' actually belongs to another label like 'wall', this label does not provide any additional information. Instead, it is better to use labels that are present in all 3D

Algorithm 5.2: An overview of how the models are scaled when the original model is available as an OBJ file.

Input: the original OBJ file, *obj*; the point cloud file, *pcd*
Output: the scaled point cloud file, pcd_{scaled}

create bounding box, bb_{obj}, around *obj*;
create bounding box, bb_{pcd}, around *pcd*;
h_{obj} = height of bb_{obj} ;
h_{pcd} = height of bb_{pcd} ;
`// select a revised scale, in this case 1 unit is 5 meters`
$scale = h_{obj}$ / (h_{pcd} * 5) ;
pcd_{scaled} = scale *pcd* by *scale* ;
return pcd_{scaled}

Figure 5.6.: Select examples of 3D models from the pre-processed data set.

models and those that are labeled fairly consistently. Based on how many 3D models contained each label, it was determined that the labels 'wall', 'window', 'roof', and 'door' had the most 3D models available and a combination of these labels can be used for training.

5.5. Processed Data Set Information

The pre-processed data set released with this report can be found on 4TU.ResearchData at `https://doi.org/10.4121/4d82052e-650c-4775-8bd9-623df68991b6.v1`.

Based on the above-mentioned changes, the data set released with this thesis contains labeled point clouds of 'house' and 'villa' buildings. Each model contains only points of label 'door', 'roof', 'wall', and 'window'. Some examples of models from the data set are shown in Figure 5.6. The data set was narrowed down to relevant building types, and site geometry has been removed. The data set has not been scaled due to the time limitations of this thesis. The revised data set contains a total of 913 3D models. Table 5.3 shows how many 3D models contain each label before being simplified to the four labels and Table 5.4 shows the number of 3D models of each typology and building type. Additional information on the released data set can be viewed in Appendix A.4.

	number of models
0 undetermined	913
1 wall	913
2 window	913
3 vehicle	0
4 roof	913
5 plant, tree	0
6 door	913
7 tower, steeple	2
8 furniture	0
9 ground, grass	0
10 beam, frame	266
11 stairs	357
12 column	267
13 railing, baluster	253
14 floor	460
15 chimney	297
16 ceiling	373
17 fence	0
18 pond, pool	0
19 corridor, path	0
20 balcony, patio	338
21 garage	281
22 dome	0
23 road	0
24 entrance, gate	0
25 parapet, merlon	70
26 buttress	0
27 dormer	83
28 lantern, lamp	0
29 arch	0
30 awning	43
31 shutters	40
total	913

Table 5.3.: Total number of 3D models that contain each type of label before the labels were removed. The released data set only contains the labels highlighted in the table.

	number of models
Building Typology	
commercial	7
military	1
public	2
religious	1
residential	902
total	913
Building Type	
house	629
villa	284
total	913

Table 5.4.: Total number of 3D models that belong to each typology and building type in the released data set.

6. Generative Adversarial Network Training Data

Chapter 5 discussed changes to the data set that would make it more suitable for automating the design of buildings. To use the data set for this thesis, additional adjustments needed to be made to format each 3D model correctly for training a GAN. These changes include limiting the number of labels and converting the data from a point cloud data format to an array.

6.1. Selecting Labels for Training

Although the training data is uploaded with four different labels, the training for this thesis was simplified and no labels were used. GANs can only generate a binary output. Therefore, they can only produce an output that states if a voxel exists or does not exist. Future research can explore labeling the voxels based on the available classes for different combinations of labels. An example of different label combinations can be seen in Figure 6.1. The architectures tested during this thesis were trained on 3D models that were simplified to one label only. The label determines whether the voxel is visible or not.

6.2. Point Cloud to Voxel Grid

After trying a number of different Python libraries without success, I wrote my own script for converting point clouds into an array. This allowed me to skip the intermediate step of using an external library to first generate voxels from the 3D models and instead meant that the data could be encoded directly from a point cloud to an array. The number of voxels is an important consideration. It not only impacts the resolution of the 3D models, as indicated in Figure 6.2, but it also affects the required

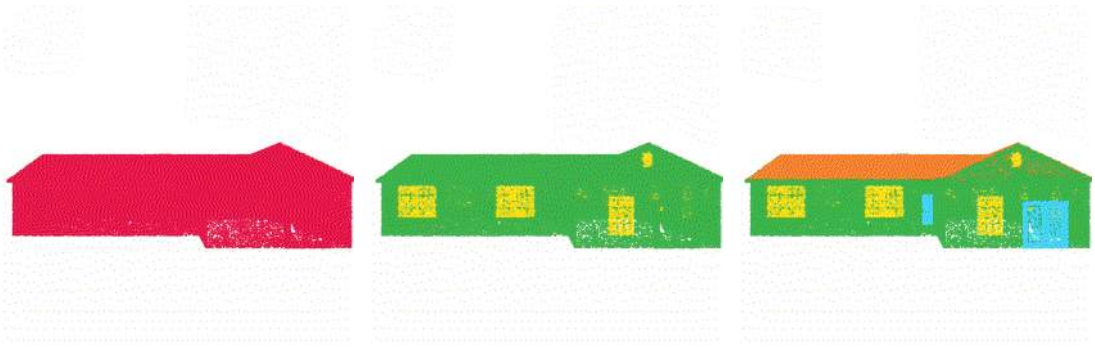


Figure 6.1.: Examples of a 3D model with one label, two labels ('wall' and 'window'), and four labels ('wall', 'window', 'roof', and 'door' labels.

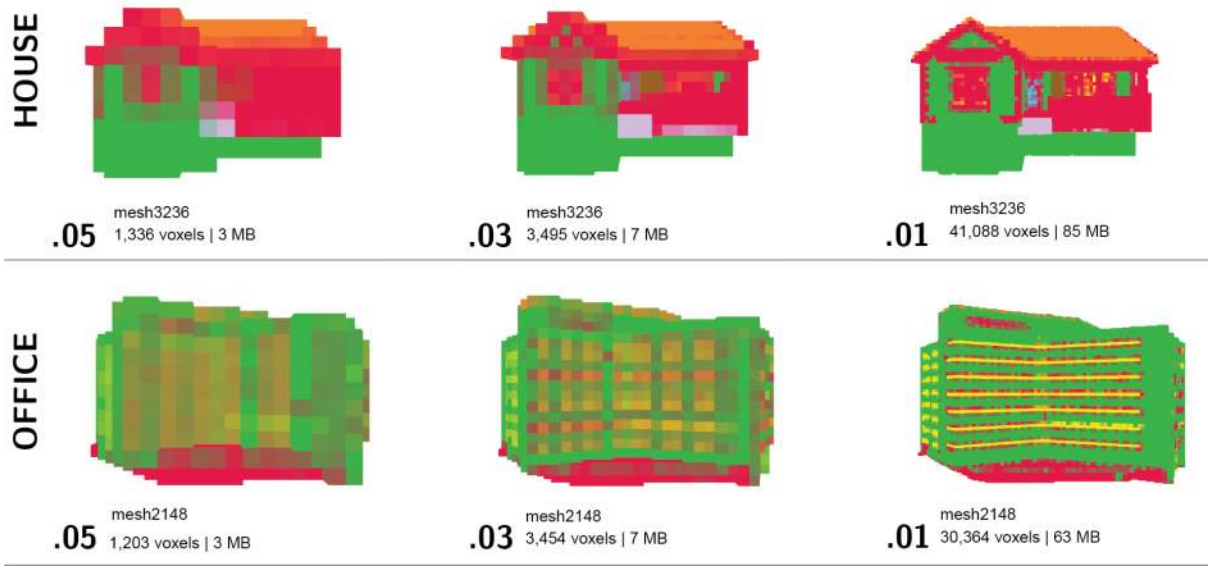


Figure 6.2.: Examples of three different scales of voxels and how much memory is then required to store the 3D model. This images shows two 3D models before the 3D models have been scaled relative to each other.

amount of memory, which is an important consideration. As found in existing research, the creation of high-resolution models is a critical next step in 3D GAN research. Balancing the need for high-quality geometry is a limit of the amount of memory needed to load the models for training.

MEMORY The maximum resolution to allow the training to run on a laptop depends on the available memory. Assuming that each voxel will be stored in 4 bytes of memory, the maximum resolution is equal to the amount of memory available divided by the number of channels multiplied by four bytes.

$$V_{max} = M_{available} / (C_{total} * 4 \text{ bytes}) \tag{6.1}$$

Where $M_{available}$ is the amount of memory available, V_{max} is the maximum number of voxels, and C_{max} is equal to the total number of channels. This means that the maximum size of the building models in the training set is dependent on the total number of channels and the available memory. For this thesis, I had two resources available, a laptop and a high performance computing cluster. More details on the hardware are available in Section 7.1.

The minimum number of channels used was a total of 86 channels. This meant that the maximum voxels could range between 11.6 million and 93.0 billion voxels per building model, which includes the empty voxel space. This gives a maximum size geometry space of 226 x 226 x 226 when training on a laptop and 453 x 453 x 453 when training on the Delft High Performance Computing Centre (DHPCC) assuming a cubic shape. However, through the experiments, different numbers of channels were also tested, with the maximum number of channels being 724 channels in total (architecture **17R** described in Table 9.3). Some even larger options with up to 1,924 channels were also tested. When using 724 channels, the maximum geometry space is 111 x 111 x 111 on a PC and 222 x 222 x 222 on the DHPCC.

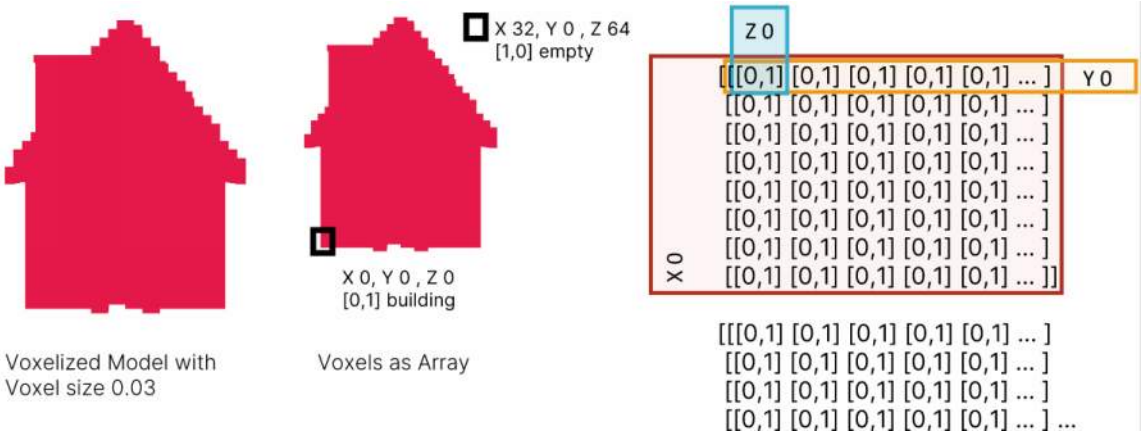


Figure 6.3.: An example of how a 3D model with two labels is encoded.

For the largest number of channels, 1,924, the maximum geometry space is 80 x 80 x 80 on a computer and 160 x 160 x 160 on the DHPCC.

However, when selecting the geometry space size, there were also other factors to consider, such as the number of training data points that are loaded at once, the padding, stride and kernel sizes, and the size of the building models. The overall size was ultimately based on the size of the building models and how the number of layers and stride impacted the size. For example, with a stride of two, the dimensions of the geometry space are doubled for each convolution in the generator (and halved in the discriminator or critic). This means that with five layers, the minimum size of each axis of the geometry space can be 1 * 2 * 2 * 2 * 2 or 16. This, of course, is too small to fit any of the building models, but it demonstrates how many factors impact the size of the geometry space. The selected size is discussed in further detail in Chapter 7.

ENCODING TO ARRAY After the steps are taken to clean and pre-process the data set, it is necessary to encode the voxelized 3D models to an array which the GAN can read. This is done through one-hot encoding. The pseudo-code that describes the process of encoding the point cloud in an array is described in Algorithm 6.1. Each point in the point cloud is transferred to the correct location in the array and the label is encoded as a tensor. If two labels are used as with the training in this thesis (empty and building geometry), then the encoding is [1,0] for empty voxels and [0,1] for filled voxels. This can be seen in Figure 6.3. Additionally, the values of the labels are encoded into a label array, which is saved separately. This is set up so that future research can explore training a ANN to label generated geometry with more than two labels.

The final data structure of the encoded array is a 4-D matrix where the first axis describes the x location, the second axis describes the y location, the third axis describes the z location of a voxel, and the fourth axis describes the state of the voxel (filled or empty). Figure 6.4 shows a comparison by visualizing the images of the point clouds from the data set above and the voxel representation of the matrix after the point cloud is encoded.

6.3. Filling 3D Model Shells

The 3D models from the data set are hollow, but throughout the experiments it became evident that it was important to also test the network architecture with solid building models. After recognizing this, I wrote a script to fill in the building models. The pseudo-code that describes the process of filling the building model is described in Algorithm 6.2. The script starts with the bottom layer of the building

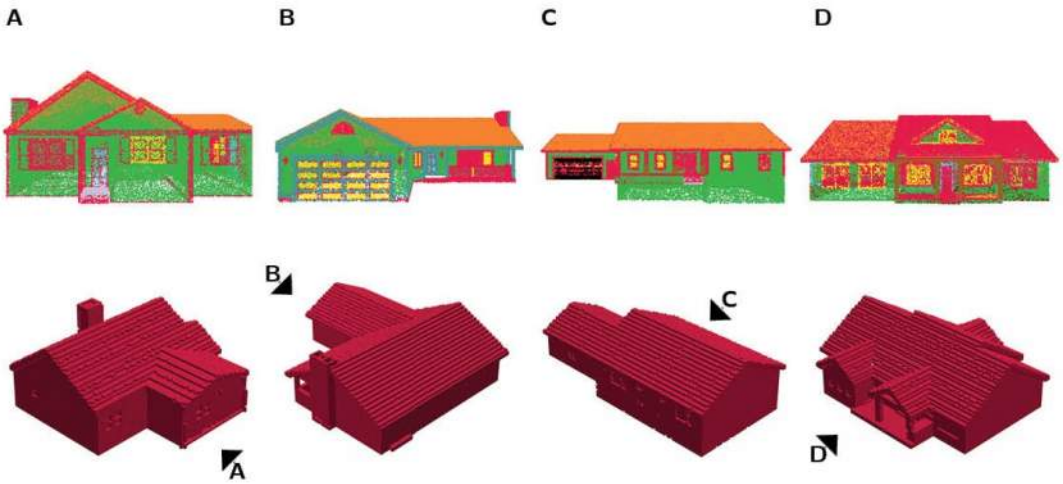


Figure 6.4.: A comparison showing the original point cloud model and a voxel representation of the matrix after the 3D model has been encoded.

Algorithm 6.1: An overview of how the point clouds and labels are encoded to an array

Input: the point cloud *pcd*, the labels for each point *label*, the voxel size vox_{size}, the maximum bounds of the geometry space $(x_{max}, y_{max}, z_{max})$

Output: P: the encoded array of the point cloud, L: the encoded array of the labels

generate P: a new, empty matrix of size [x_{max}, y_{max}, z_{max}, 2]
generate L: a new, empty matrix of size [x_{max}, y_{max}, z_{max}, 1]
calculate the translation t between the center of the bottom layer of the matrix and the center of the bottom of the point cloud
for *i, p in pcd* **do**
    $x = p[0]$ // $vox_{size} + t[0]$
    $y = p[1]$ // $vox_{size} + t[1]$
    $z = p[2]$ // $vox_{size} + t[2]$
    $P[x, y, z]$ = [0,1]
    $L[x, y, z] = label[i]$
return P, L

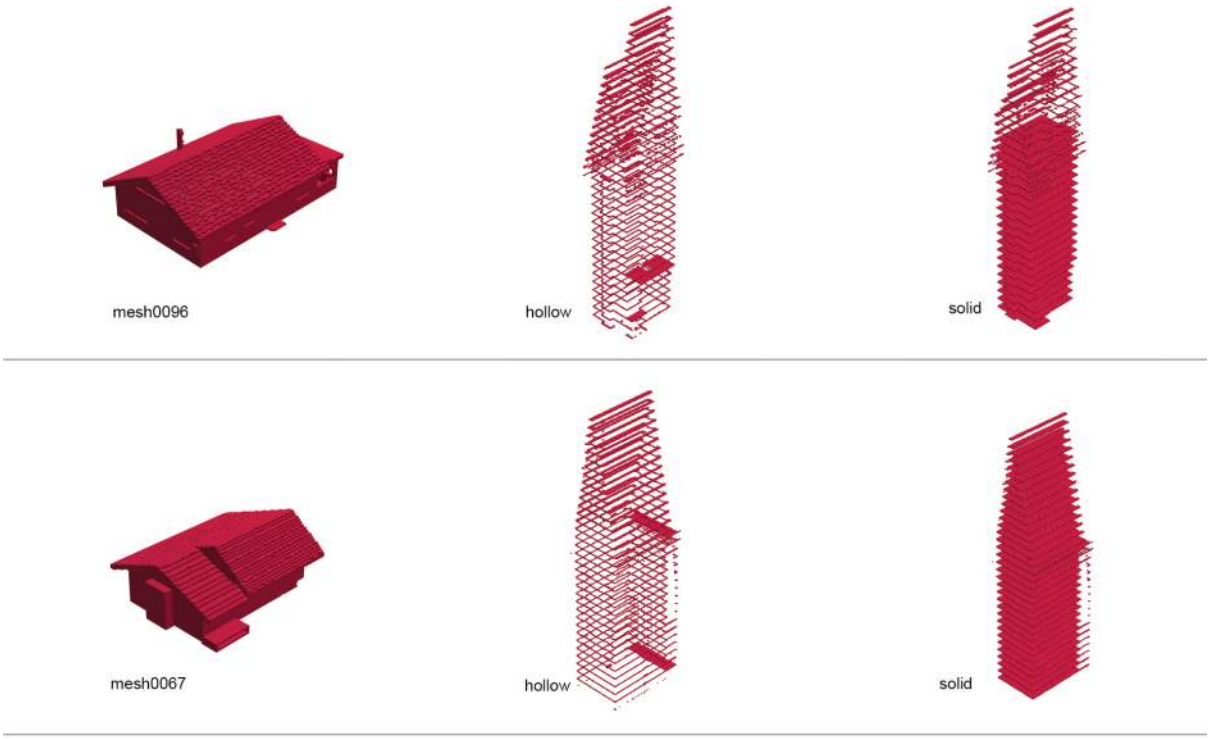


Figure 6.5.: The original 3D model, a pulled apart view of the 3D model showing the hollow internal structure, and a pulled apart view of the 3D model showing the solid internal structure after filling the interior.

model, selects the center point, and checks if it is filled in. If it is not filled, it fills the center point and then adds all the adjacent points to a list to be checked as well. When it gets to a point that is already filled, the neighbors do not get added to the list, so the fill stops at the edges. There are also checks in place for layers that have holes and for layers where the center point is outside of the exterior boundary. The script does not perfectly fill all 3D models, but it fills the majority of the layers and works fairly well. Figure 6.5 shows an example of how it successfully fills one building model and fills another fairly well, although not perfectly. If more time were available to continue this research, the script should be adjusted to catch more edge cases, which would allow it to fill all models perfectly.

Algorithm 6.2: An overview of how the interior points and labels are filled in the array

Input: the encoded array of the point cloud P, the encoded array of the labels L, the maximum height h, the maximum bounds of the geometry space $(x_{max}, y_{max}, z_{max})$

Output: P: the filled encoded array of the point cloud, L: the filled encoded array of the labels

```
center = (x_max // 2, y_max // 2)
for layer in range(h) do
    points = [(center[0], center[1], layer)]  while length of points is greater than 0 do
        if point (x, y, z) in points is within the maximum bounds and P[x, y, z] is empty then
            // fill the voxel, [0, 1] represents a filled voxel
            P[x, y, z] = [0, 1] // 50 is the label for interior voxels
            L[x, y, z] = 50
            neighbors = [ (x-1, y, z), (x, y-1, z), (x, y+1, z), (x+1, y, z) ]
            append neighbors to points
return P, L
```

7. Generating Geometry with 3D Deep Convolutional GANs

Once the data is properly encoded, it can be used to train a GAN model that generates building geometry. Initially, the architecture was based on 3D GAN documented by [Wu et al., 2016]. However, after testing this architecture and adjusting the hyperparameters, it became clear that this architecture was not going to work for the application of generating buildings. Therefore, it was necessary to develop new architectures to generate 3D geometries and test these different architectures.

7.1. System Details

For the computation time mentioned in the rest of the thesis, the data processing was completed on one of two set-ups.

A Windows PC, which will be referenced as PC, with following specifications:

1. CPU: Intel(R) Core(TM) i7-10750H @ 2.60GHz (6 cores)
2. GPU: NVIDIA Quadro T1000 with Max-Q Design (4 GB GDDR6 dedicated)
3. RAM: 16 GB

The DHPCC [Delft High Performance Computing Centre , DHPC], which will be referenced as DHPCC with the following specifications:

1. CPU: AMD EPYC 7402 24C @ 2.80 GHz (2 cores)
2. GPU: NVIDIA Tesla V100S 32 GB
3. RAM: 2GB - 8GB requested [64 GB available total]

7.2. Testing State-of-the-Art 3D GANs

As discussed in Section 3.5, GANs consist of two ANN, a generator G and a discriminator D. The generator learns how to map a random noise vector z to a 3D geometry x, $G(z) = x$. The discriminator distinguishes between the geometry generated by G and the ground truth. Therefore, given x, D should be able to tell if x is real or fake by providing an output that is the probability value $D(x) = P(real)$. The two networks compete against each other in a zero-sum game. The generator is trying to produce 3D geometry as close to the ground truth, while the discriminator becomes better at identifying if the geometry is fake.

Initially, I implemented an architecture structure based on 3D GAN [Wu et al., 2016]. This structure is a DCGAN where the discriminator has a deep convolutional network to stabilize the training, but the output is 3D instead of 2D. With the goal of rapidly prototyping network designs, I made adjustments to the architecture starting with fewer layers and channels to accelerate training, as shown in Figure 7.1.

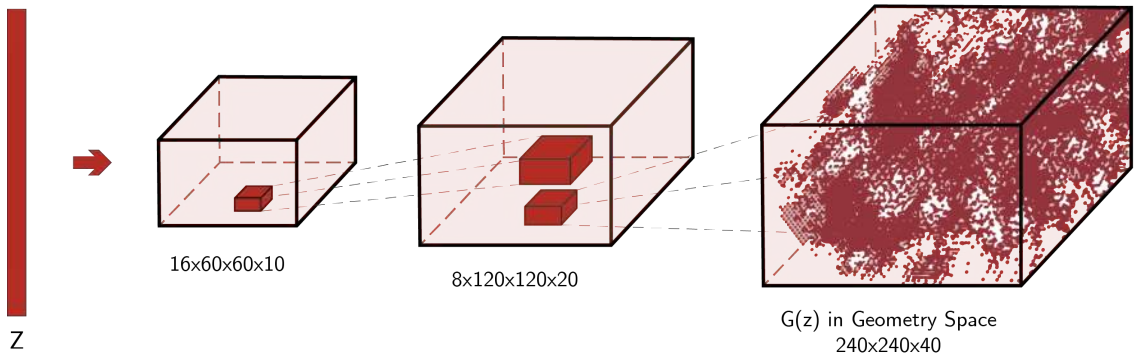


Figure 7.1.: The generator in **v00** base architecture based on 3D GAN [Wu et al., 2016]. The discriminator mostly mirrors the generator. It takes $G(z)$ as an input and outputs a real number in [0, 1]. Additionally, the discriminator uses Leaky ReLU, instead of ReLU, as the activation function.

I also initially used only one training data point to minimize training time and get results faster. Having a small training data set means that the model is more prone to overfitting, but I would still expect the model to produce a meaningful output that allows me to evaluate if it is training properly. This allowed the architectures to be tested quickly on a PC with 6,000 epochs of training completing in a few hours. Additional information on which models were included in the small data set and visualizations of some models can be found in the Appendix A.1.

More detailed information on the GAN architecture is provided later in this section. Using this network architecture, after several thousand epochs, the generator should produce geometry similar to the data set. After implementing this architecture, the training was very unstable when training on the data set which contains the 3D building geometry.

Table 7.1 shows the different hyperparameter adjustments that were initially tested to stabilize training. With these architectures, the network width, learning rates, and initializers were adjusted.

NETWORK WIDTH Network width refers to the number of channels used. Remember that each channel is one learned feature of the 3D model. Therefore, it is important to balance the number of channels so that the model learns enough features to generate accurate geometry, but not too many to unreasonably increase the amount of time it takes to train. It is also important to limit the network width so that the training procedure does not exceed memory limits.

LEARNING RATES The learning rate determines how much of an impact a single epoch's results have on the update to the weights and biases of the network. Changing the learning impacts how well the network learns and the balance between the generator and the discriminator. The initial rates were based on the learning rates Wu et al. [2016] used which are 0.025 in the generator and 0.00001 in the discriminator.

ID	factors
v00	base architecture
v01	learning rate G: 0.025 D: 0.00001
v02	kernel 4x4x4
v03	learning rate G: 0.025 D: 0.000001
v03	learning rate G: 0.025 D: 0.001
v03	learning rate G: 0.025 D: 0.00015
v04	4 layers 48, 24, 12, 2
v05	4 layers 56, 28, 14, 2
v05	4 layers 56, 28, 14, 2, initializer RandomNormal
v05	4 layers 56, 28, 14, 2, initializer GlorotNormal
v05	4 layers 56, 28, 14, 2, initializer GlorotUniform

Table 7.1.: A list of the experiments that were run to asses the existing 3D GAN architecture.

INITIALIZERS Initializers define the way in which the initial random weights of layers are set. Keras offers many initializer options, but for GANs, HeNormal, RandomNormal, GlorotNormal, and GlorotUniform are the most commonly used.

For each version of the architecture, the hyperparameter options were tested based on the changes noted in Table 7.1. A 240 x 240 x 40 geometry space was selected because the largest model in the data set would fit inside this space. Training with a data set of one model also meant that this larger geometry space did not cause any memory problems. The initial base architecture of v00 is described below.

GENERATOR v00 [BASE] The generator consists of three fully convolution layers with numbers of channels {32, 16, 8, 2}, kernel sizes {3, 3, 3}, and strides {2, 2, 2}. The kernel and stride sizes are the same in all three dimensions. The activation function used is ReLU (Figure 3.4) between convolutional layers, and a Softmax (Figure 3.4) layer at the end. The kernel initializer used is HeNormal. The input is a 200-dimensional vector, and the output is a 240 × 240 × 40 matrix with values in [0, 1]. The generator uses ADAM [Kingma and Ba, 2014] with Learning Rate=1e-4 and Beta=0.5.

DISCRIMINATOR v00 [BASE] The discriminator is a mirrored version of the generator, it takes an input of a 240 × 240 × 40 matrix, and outputs a real number in [0, 1]. The discriminator consists of three volumetric convolution layers, with numbers of channels {8, 16, 2}, kernel sizes {3, 3, 3}, and strides {2, 2, 2}. The kernel and stride sizes are the same in all three dimensions. There are Leaky ReLU (Figure 3.4) layers with the alpha parameter set to 0.2 in between the convolutional layers, and a Sigmoid (Figure 3.4) activation function at the end. The discriminator uses ADAM [Kingma and Ba, 2014] with Learning Rate=1e-4 and Beta=0.5.

All the tests resulted in the generator producing noise and training remained unstable. Despite adjusting the parameters, the generator and discriminator did not balance each other. This is a typical challenge faced when training GANs. Figure 7.2 shows some results from this early training. The visualized output is noise and the training does not converge or stabilize. Each improvement the generator makes causes a sharp impact on the discriminator and they are not able to find an equilibrium. When looking at the different aspects of the training, it became clear that the models had the common problem of vanishing gradients.

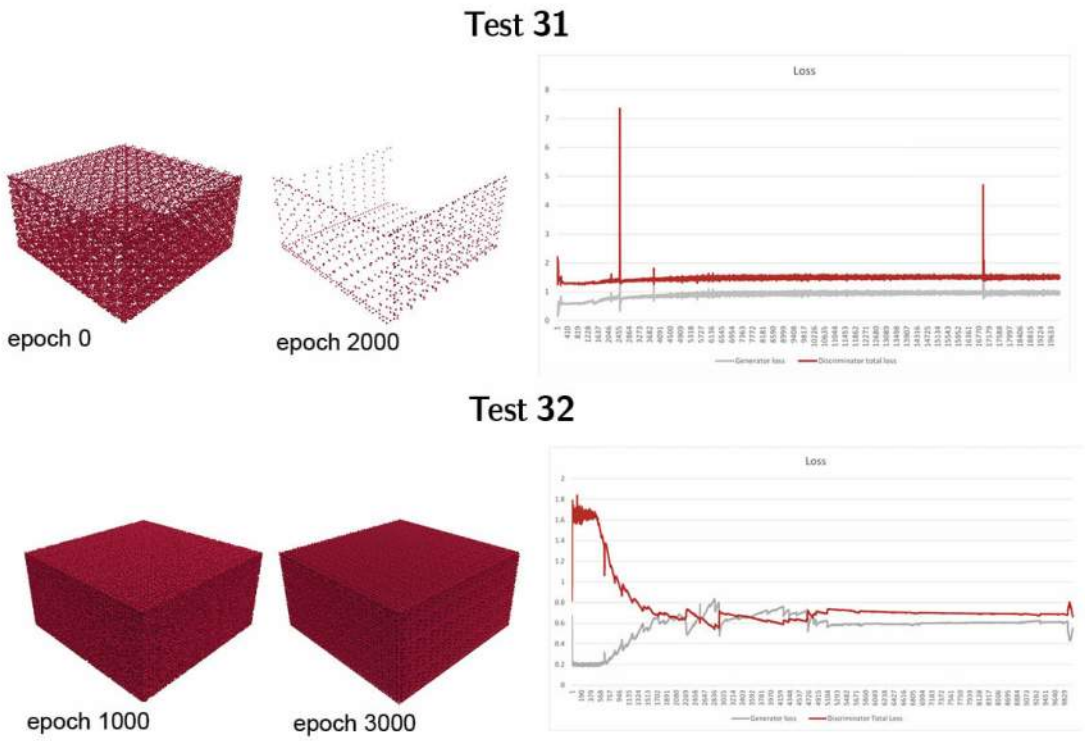


Figure 7.2.: Early training results showed a very unstable training process and the output remained noise or geometry that does not resemble a building.

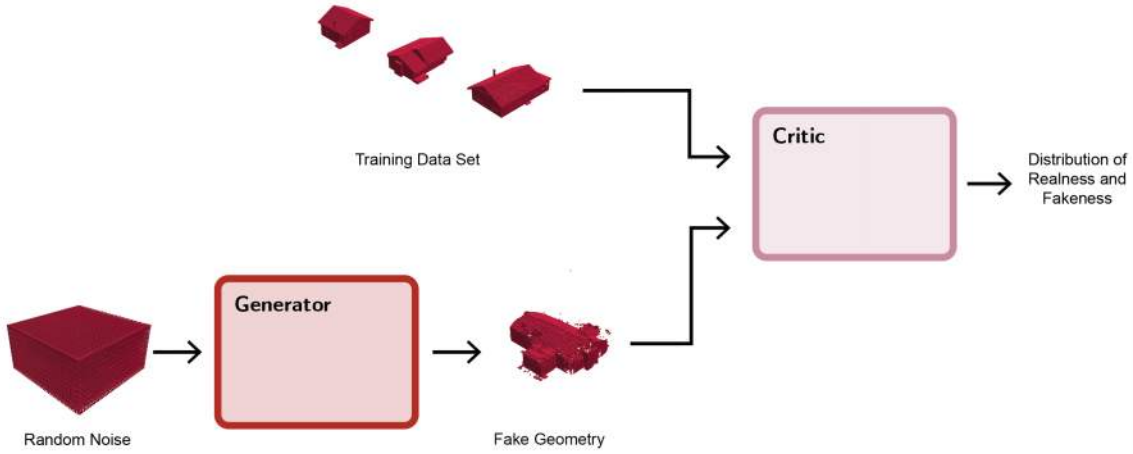


Figure 7.3.: When using Wasserstein Loss [Arjovsky et al., 2017], a Critic (C) is trained with the training data and the Critic evaluates the fake geometry and outputs a scalar representing the distribution of realness between $-\infty$ and ∞.

7.3. GAN Loss Functions

After testing a number of different parameter adjustments, the primary issue that made the training unstable was vanishing gradients. Unstable training led me to realize that instead I would need to develop a new GAN architecture to address the problem of generating building geometry.

To address the issue of vanishing gradients, I implemented Wasserstein Loss [Arjovsky et al., 2017]. The expectation was that using Wasserstein Loss would stabilize the training by solving the problem of the vanishing gradients. This would then allow the generator to have a more successful training procedure and generate geometry that was more similar to the training data. The results of the experiment showed that this primary change helped stabilize the training and provided results that aligned with the initial expectations. This loss function replaces the commonly used min-max loss function. To understand how changing the loss function improved the training, it is necessary to understand the algorithm of the loss function.

Min-Max Loss Function Recall that the loss function used by GANs is called the standard GAN loss function or the min-max loss function. This loss function was first described by Goodfellow et al. [2014] when they first described GANs.

$$\min_G \max_D \mathbb{E}_{x\sim p_{\text{data}}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[1 - \log D(G(z))] \tag{7.1}$$

The loss function is minimized by the generator, and it is maximized by the discriminator. Although mathematically, the min-max loss formulates the loss effectively, in practice it does not work as well. The generator easily falls behind the discriminator and, therefore, stops learning if it cannot catch up because the generator cannot gain any valuable information from the discriminator. The loss function can be examined in relation to the discriminator and the generator.

Discriminator Loss The discriminator trains by classifying real data and fake data that is output by the generator. The discriminator receives a penalty for incorrectly classifying the data. For example,

by classifying a real data point as fake or a fake data point as real. The gradient is calculated by maximizing the function:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} [\log D(x^i)] + \log (1 - D(G(z^i)))] \tag{7.2}$$

where $\log D(x^i)$ describes the probability that the discriminator classifies the real data points correctly. Maximizing $\log (1 - D(G(z^i)))$ provides the reward for correctly labeling fake data points as fake.

Generator Loss To train, the generator samples random noise and generates an output. The output is then classified by the discriminator as real or fake. The generator's loss function rewards the generator for tricking the discriminator when the discriminator labels the data point as real. If it does not trick the discriminator, it is penalized. The gradient is calculated by minimizing the function:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} [\log (1 - D(G(z^i)))] \tag{7.3}$$

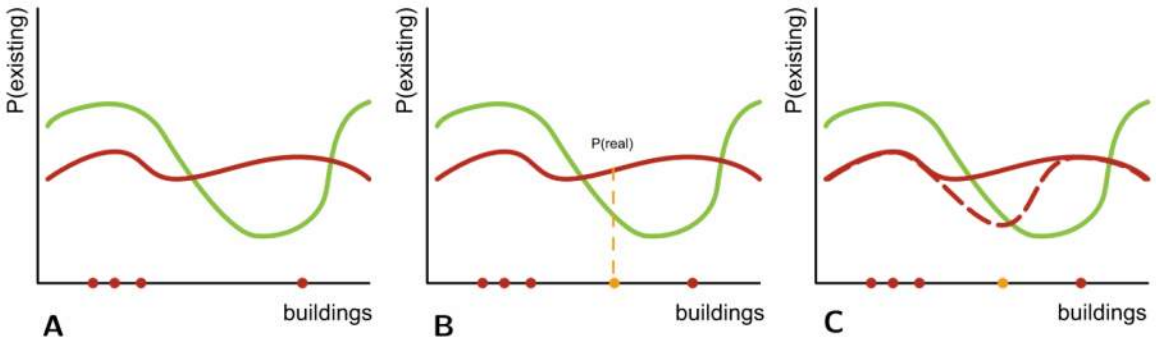


Figure 7.4.: There exists a distribution (green line) of the buildings that exist. The discriminator uses its assumption of this distribution (red line) to evaluate if data samples produced by the generator (yellow) are real or fake and then adjust its distribution based on the new data point (dashed red line).

To better understand the Min-Max Loss function, note the following scenario. As shown in Figure 7.4 (A), there exists a distribution, represented by the green line, of the buildings that exist. It is not possible to know this distribution, but there are available data points that show which buildings are more likely to exist. There is another distribution, represented by the red line, that the discriminator has assumed based on the data set it has seen. When the generator creates a new building geometry, it is located somewhere along this distribution, as shown in Figure 7.4 (B). When presented this new sample, the discriminator uses its distribution to determine whether the sample is real or fake and provides this information to the generator. The generator then tries to improve its next output based on the feedback, working towards geometry that gets identified as real. This is how the generator learns. The discriminator also takes the actual value of this new data point to adjust its distribution as shown in Figure 7.4 (C). This is how the discriminator learns. When using the Min-Max Loss Function, the generator receives very little information from the discriminator.

Wasserstein Loss Function To address the issue of vanishing gradients, I instead implemented Wasserstein Loss [Arjovsky et al., 2017]. This change helped stabilize the training and provide results that were consistent with the initial expectations. When implementing Wasserstein loss, the structure of the network changed because instead of a discriminator D training the generator G, the discriminator is replaced by a critic C. Figure 7.3 gives an overview of a typical Wasserstein generative adversarial network (WGAN) structure. Instead of only providing a probability value to the generator, the critic scores the realness or fakeness of a given 3D geometry produced by the generator. The critic outputs a scalar between $-\infty$ and ∞. Therefore, given x, C should be able to produce a scalar s, representing the distribution of realness and fakeness, $C(x) = s$. The loss function for Wasserstein loss is as follows:

$$\min_G \max_C \mathbb{E}_{z\sim p_z(z)}[C(G(z))] - \mathbb{E}_{x\sim p_{\text{data}}(x)}[C(x)] \tag{7.4}$$

Changing the loss function to improve performance is motivated by the process of training GANs. In all GANs, the generator trains by minimizing the distance between the distribution of the training data and the distribution of the generated geometry. There are a number of distance measures that can be used which include: Kullback-Leibler (KL) divergence, Jensen-Shannon (JS) divergence, and Earth-Mover (EM) distance, also known as the Wasserstein distance. Min-Max Loss effectively uses JS divergence and minimizes it. To implement Wasserstein loss, the critic is trained to approximate the EM or Wasserstein distance instead, and also provides this additional information to the generator. Since the Wasserstein distance is continuous and differentiable, it will provide a linear gradient to the generator even after it is well trained. This is contrary to a discriminator model, which once trained may no longer provide useful gradient information to the generator. The use of a critic allows the GAN model to seek convergence by lowering the generator loss instead of stabilizing by finding stability between a generator and discriminator. Arjovsky et al. [2017]

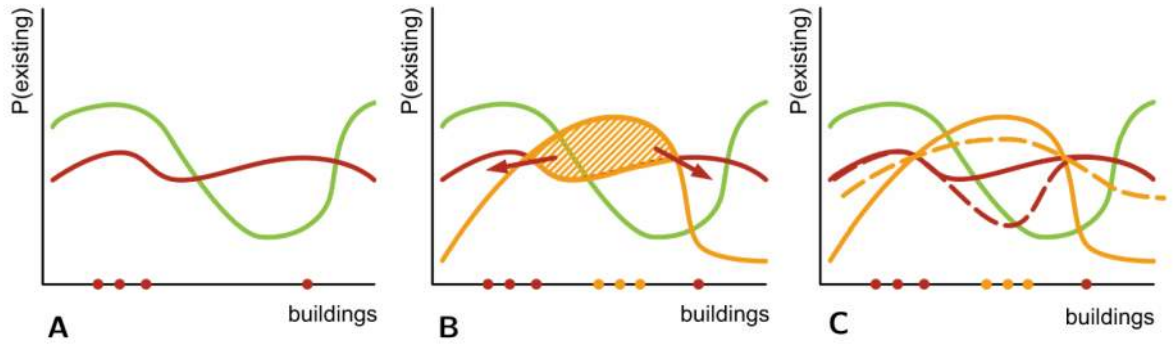


Figure 7.5.: There exists a distribution (green line) of buildings that exist. The critic uses its assumption of this distribution (red line) to evaluate how the distribution produced by the generator (yellow line) is different than the critic's distribution. The generator then adjusts its distribution based on information received from the critic (dashed yellow line) and the critic updates its distribution based on the new data points (dashed red line).

To better understand the Wasserstein loss function, remember the scenario set up in the Min-Max loss section. As shown in Figure 7.5 (A), there exists a distribution, represented by the green line, of buildings that exist and another distribution, represented by the red line, that the critic has assumed based on the data set it has seen. With Wasserstein loss, the generator also randomly guesses this distribution shown with the yellow line in Figure 7.5 (B). The generator creates multiple samples for

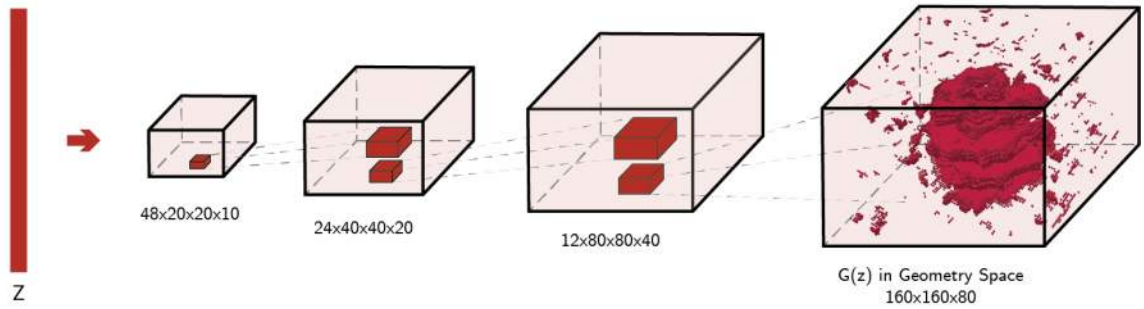


Figure 7.6.: The generator in the base architecture for DCGAN and WGAN. The discriminator and critic mostly mirror the generator.

the critic to review, as shown with the yellow points in Figure 7.5 (B). Based on these points, the critic is able to guess what the generator's distribution most likely is, and it compares this distribution to its own distribution. The critic then determines how the generator's distribution needs to change based on the critic's own distribution using the Wasserstein distance. This is done by taking the area under the curve and moving it as indicated in Figure 7.5 (B). The generator is then able to update its distribution based on the information it has received from the critic, as shown by the yellow dashed line in Figure 7.5 (C). This is how the generator learns. The critic can then also update its distribution based on the new data points it received, as shown by the red dashed line in Figure 7.5 (C). This is how the critic learns. When seeing these graphs, it becomes clear that when using Wasserstein Loss, more information is exchanged between the generator and the critic. This helps the generator learn faster because it does not have to work by trail and error as with the Min-Max Loss Function.

7.4. Experimental Set Up

[completed on PC]

After changing the loss function, I also explored some other hyperparameters of the network architecture that could be adjusted. I completed an ablation study to determine which aspects of the network architecture had an impact on stabilizing the training. An ablation study separates out hyperparameters and tests these to determine which parameters impact a successful architecture. These changes were tested on both the original DCGAN and the new WGAN. Starting with the base architectures described below and visualized in Figure 7.6, subsequent versions incorporated different hyperparameters into the architecture until the training stabilized. Due to having to test many different factors, it was important to be able to test quickly. Therefore, the training data set still contained only 1 3D model to allow fast training. More information on the 3D model used can be found in Appendix A.1. The base architecture started with four layers and the maximum number of channels was 48, since this architecture performed better than the network with three layers. The 160 x 160 x 80 geometry space was selected because most of the data set fit within this smaller geometry space. The model performs better when the generated geometry takes up most of the geometry space. Additionally, when another layer was added to the network, it was necessary to increase the height of the geometry space. To do this without increasing the required memory by too much, the width and depth were reduced. The base architecture for 3D GAN and WGAN is described below.

3D DCGAN

This architecture is based on the precedent of the 3D GAN architecture [Wu et al., 2016].

LOSS FUNCTION Min-Max GAN loss function

GENERATOR The generator consists of four fully convolution layers with a number of channels of {48, 24, 12, 2}, kernel sizes {4, 4, 4, 4}, and strides {2, 2, 2, 2}. The kernel and stride sizes are the same in all three dimensions. The activation function used is ReLU between convolutional layers, and a Softmax layer at the end. The input is a 200-dimensional vector, and the output is a 160 × 160 × 80 matrix with values in [0, 1]. The generator uses ADAM [Kingma and Ba, 2014] as the optimizer with Learning Rate=0.0025 and Beta=0.5.

DISCRIMINATOR The discriminator is a mirrored version of the generator, it takes an input of a 160 × 160 × 80 matrix, and outputs a real number in [0, 1]. The discriminator consists of four volumetric convolution layers, with a number of channels of {12, 24, 48, 2}, kernel sizes {4, 4, 4, 4}, and strides {2, 2, 2, 2}. The kernel and stride sizes are the same in all three dimensions. There are Leaky ReLU layers with the alpha parameter set to 0.2 in between the convolutional layers, and a Sigmoid layer at the end. The discriminator uses ADAM [Kingma and Ba, 2014] as the optimizer with Learning Rate=0.00001 and Beta=0.5.

3D WGAN

This architecture takes DCGAN as a starting point, but with a revised loss function.

LOSS FUNCTION Wasserstein loss function [Arjovsky et al., 2017]

GENERATOR The generator consists of four fully convolution layers with a number of channels of {48, 24, 12, 2}, kernel sizes {4, 4, 4, 4}, and strides {2, 2, 2, 2}. The kernel and stride sizes are the same in all three dimensions. The activation function used is Leaky ReLU between the convolutional layers and a Tanh layer at the end. The input is a 200-dimensional vector, and the output is a 160 × 160 × 80 matrix with values in [0, 1]. The generator uses ADAM [Kingma and Ba, 2014] with Learning Rate=0.0025 and Beta=0.5.

CRITIC As a mirrored version of the generator, the critic takes as input a 160 x 160 x 80 matrix, and outputs a scalar between $-\infty$ and ∞. The critic consists of four volumetric convolution layers, with a number of channels of {12, 24, 48, 2}, kernel sizes {4, 4, 4, 4}, and strides {2, 2, 2, 2}. The kernel and stride sizes are the same in all three dimensions. There are Leaky ReLU layers added between the convolutional layers of parameter 0.2, and no activation function in the final layer. At the end, there is a flatten layer followed by a dense layer with one output and no activation. The critic uses ADAM [Kingma and Ba, 2014] with Learning Rate=0.00001 and Beta=0.5.

7.5. Wasserstein and Min-Max Loss Experiments and Results

To improve the quality of the output and stabilize the training, a number of hyperparameters were considered. Some hyperparameters can only be applied to DCGANs or WGANs while others can be applied to both DCGANs and WGANs. Select results are visualized and shown in this chapter to demonstrate specific training outcomes. Visualized results from all architectures that were tested starting with the base architectures can be seen in the Appendix B.

LOGITS (DCGAN ONLY) Logits is only relevant for DCGANs. This is because it is a parameter of the binary cross entropy, which is not used in WGANs. When the logits parameter is set to true, the loss function takes a value between negative infinity and infinity. It then uses sigmoid to transform the input to a value between 0 and 1. Another way of accomplishing this is to set logits to false and then

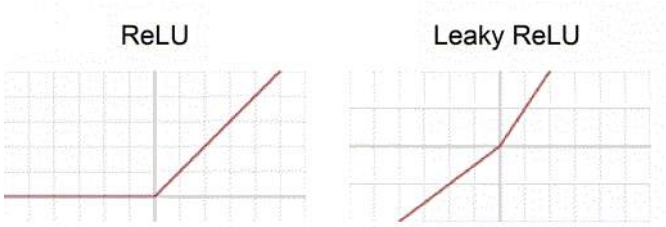


Figure 7.7.: ReLU [Nair and Hinton, 2010] and Leaky ReLU activation functions. Graphs from Wikipedia [2023].

add a sigmoid layer at the end of the discriminator. When logits is set to false, then the loss function expects to take as an input the probability of real and fake. The sigmoid at the end of the discriminator would transform the output to this probability; however, this can lead to more floating point errors. Using logits instead reduces the floating-point errors and is therefore a better option.

LABEL SMOOTHING (DCGAN ONLY) As a standard, the discriminator outputs a label of 1 to represent real geometry and a label of 0 to represent fake geometry. Label smoothing randomly varies these values so that real geometries receive a value between 0.7 and 1.2 and fake geometries receive a value between 0.0 and 0.3 as described in Algorithm 7.1. This can have a regularizing effect on model training. [Salimans et al., 2016]

Algorithm 7.1: An overview of how label smoothing is implemented

Input: the original label l
Output: the label after smoothing

```
// random returns a random value between 0 and 1
if l == 1 then
    return l - 0.3 + random * 0.5
if l == 0 then
    return l + random * 0.3
```

ACTIVATION FUNCTIONS Activation functions are used after each convolutional layer and at the end of the generator. Additionally, there are different layer structures that can be used at the end of the discriminator or critic. For the activation function after each convolutional layer, it is common to use ReLU in the generator and Leaky ReLU in the discriminator regardless of which loss function is used. ReLU however cuts off all negative values by setting them equal to zero. Leaky ReLU, on the other hand, allows some values less than zero, as shown in Figure 7.7. This is why some researchers have implemented Leaky ReLU in both the generator and the discriminator. The slope of the line mapping negative numbers can be specified. In TensorFlow, the slope is adjusted by the parameter *alpha* and 0.2 is a good starting point. This is why the experiments tested architectures and compared using ReLU and Leaky ReLU in the generator.

WGANs also use different activation functions at the end of the generator and the critic. At the end of the generator, they use Tanh or Sigmoid. At the end of the critic, there are no activation functions, but there is a layer to flatten and an additional dense layer. DCGANs use Softmax at the end of the generator and no activation function and no additional layers at the end of the discriminator. Therefore, I also tested whether the Tanh activation function at the end of the generator and the flatten and additional dense layer at the end of the discriminator would impact the performance of DCGAN for this training problem.

LEARNING RATE DECAY Using learning rate decay (lrDecay) means that the algorithm starts with an initial learning rate and decays the learning rate as training continues. For the experiments, the learning rate started at 0.0001 with a decay factor of 0.9999600016 based on commonly used values. The learning rate decreased as training went on but never below a minimum of 0.000001.

Research has shown that lrDecay works based on the optimization analysis in stochastic gradient descent. The large initial learning rate helps the algorithm escape false local minima and the rate is then decreased (decayed) so that the algorithm will locate a true local minimum and not oscillate around it [LeCun et al., 1990; Kleinberg et al., 2018]. There are also other theories of why it works. Others have argued that starting with a higher initial learning rate keeps the network from memorizing noisy data and therefore the decay helps improve the network's ability to learn complex patterns [You et al., 2019]. Although the exact reasoning is not agreed upon, lrDecay is cited as a factor that improves GAN training and was therefore a hyperparameter that was tested in the experiments.

EXPERIMENTS The parameters described above each provide benefits that can stabilize GAN training. To properly understand how each parameter impacts the training and output of the GAN, a number of experiments were run with different variables. The experiments not only tested individual parameters, but they also tested different combinations of parameters and how they work together. The experiments and results are described in Table 7.2.

This first round of experiments was completed on the PC and involved a training data set of 1 3D model. The goal was to try to overfit the network to the training data set so that the initial results could be visualized. A small training data set was also used because the experiments needed to run on a PC in a reasonable time frame. Training with the 1 3D model kept the training time below three hours. The results generated by the base architectures are visualized in Figure 7.8. These tests were not successful and did not produce any building geometry. The output was only noise. Figure 7.9 shows the training results of architectures that did not train well. Figure 7.10 shows the initial results of the **G** architecture which performed the best in the experiments. Initial tests were useful in identifying which parameters were key to stabilize the training and how the loss function had an impact on the training.

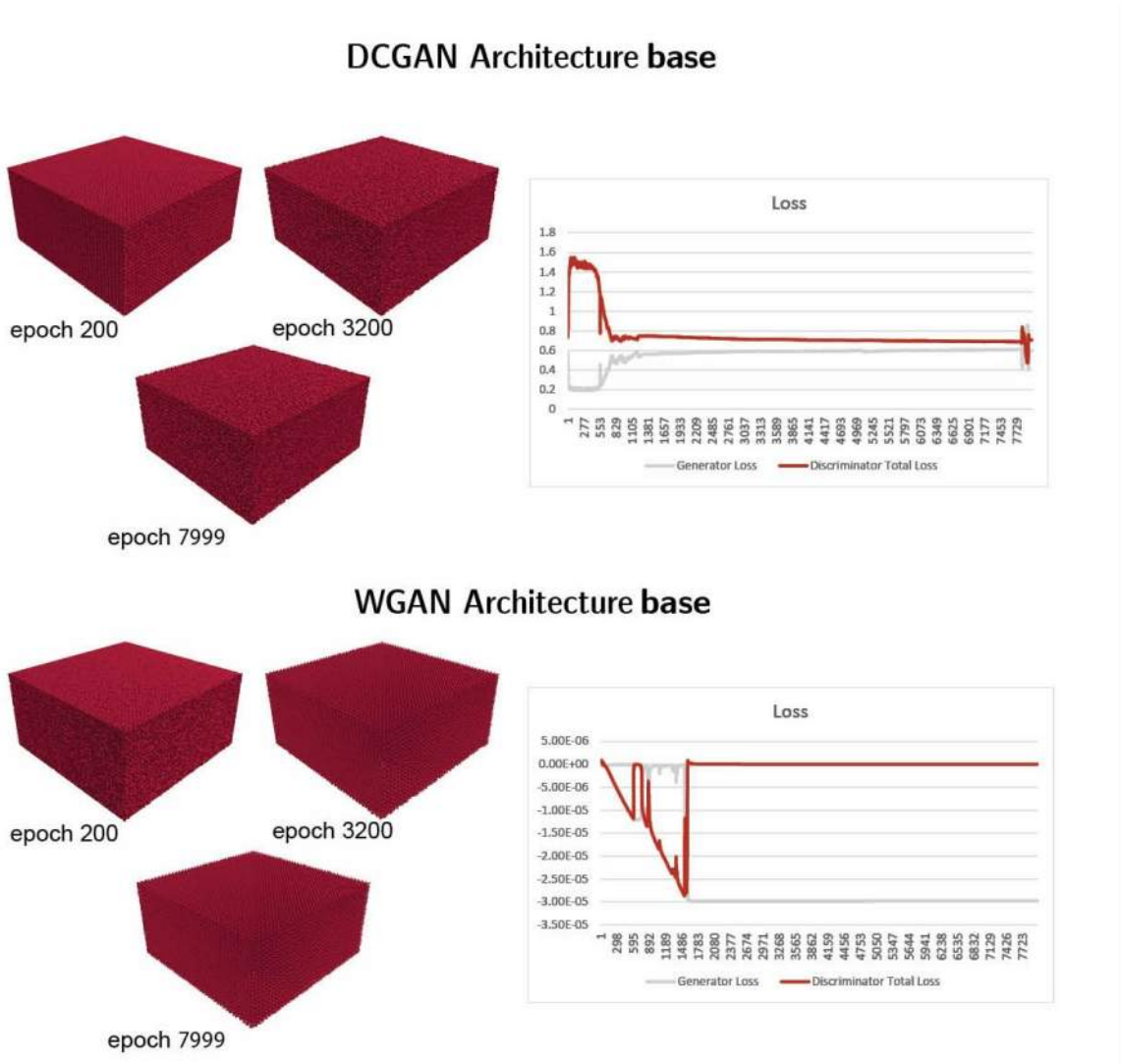


Figure 7.8.: Images showing the results of the base architectures from DCGAN (top) and WGAN (bottom) and graphs showing their respective loss.

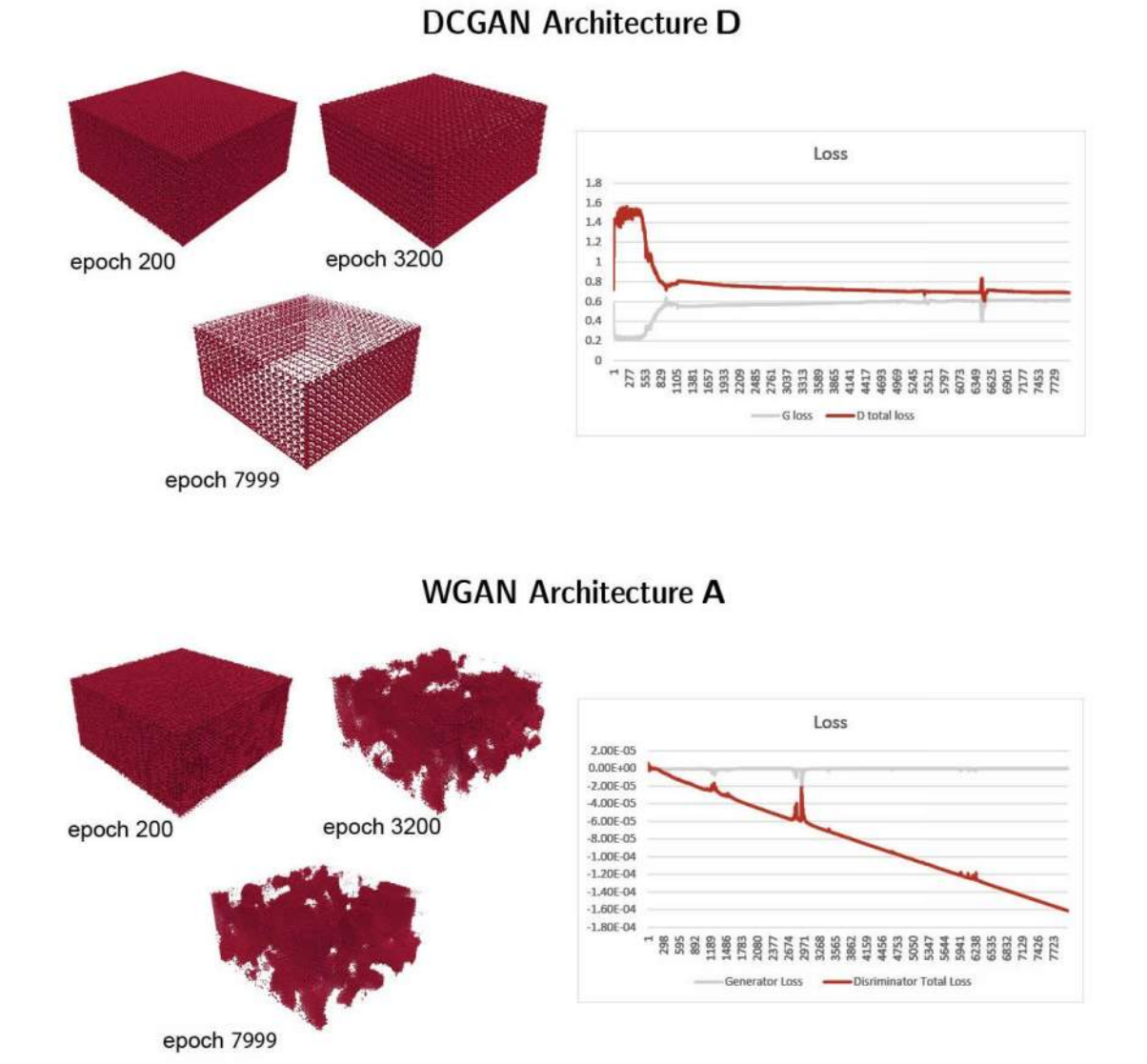


Figure 7.9.: Images showing generated geometry that resembles noise from DCGAN (top) and WGAN (bottom) and graphs showing their respective loss.

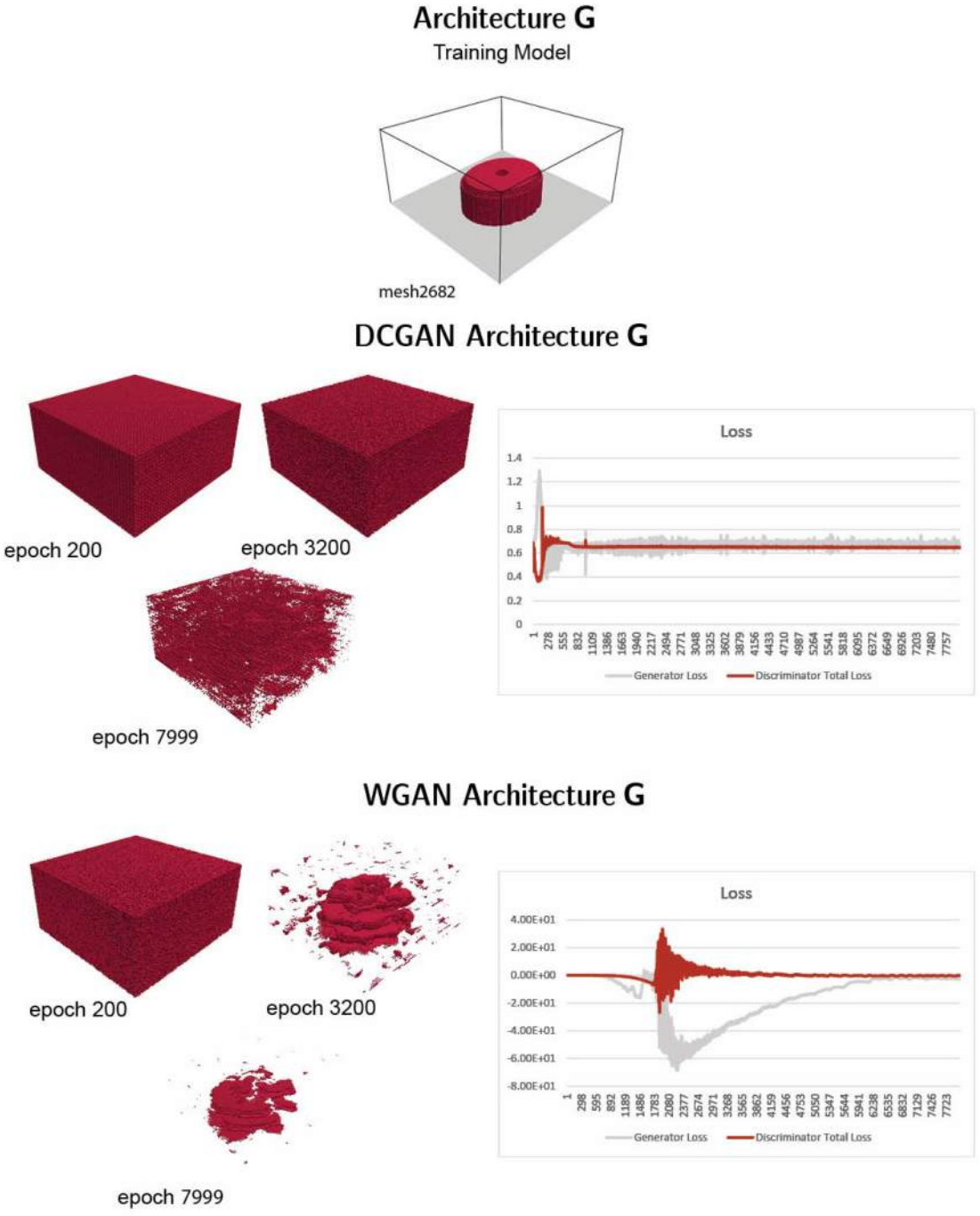


Figure 7.10.: Images showing the results of the **G** architecture with DCGAN on the top and WGAN on the bottom and graphs showing their respective loss. The models were trained with 1 training data point shown above the results.

ID	factors	3D DCGAN		3D WGAN	
		train	output	train	output
BASE	base architecture	[U]	[1]	[U]	[1]
A	Leaky ReLU	[x]	[x]	[U]	[1]
B	lrDecay	[x]	[x]	[U]	[1]
C	using logits	[U]	[1]	[x]	[x]
D	using label smoothing	[U]	[1]	[x]	[x]
E	using logits + Leaky ReLU	[U]	[1]	[x]	[x]
F	using logits + lrDecay	[U]	[1]	[x]	[x]
G	using logits (DC only) + Leaky ReLU + lrDecay	[U]	[1]	[S]	[4]
G-A	G w/ Tanh activation (end of G)	[U]	[1]	[x]	[x]
G-B	G w/ Tanh activation (end of G) and flatten (end of D)	[U]	[1]	[x]	[x]
H	using logits + Leaky ReLU + lrDecay + label smoothing	[U]	[1]	[x]	[x]

Table 7.2.: List of experiments run to asses GAN architecture with depth and width of **11** per Table 9.1. [x] indicates this combination was not tested because the parameters cannot be applied to the noted architecture [U] indicates that the training was unstable [S] indicates that the training was stable. [1] indicates the output was noise as shown in Figure 7.9 [2] indicates that the output size, shape, and/or proportions are significantly different than the training data [3] indicates that the output is correct in size or proportion but the shape is not accurate [4] indicates that the output is noisy but roughly the correct size, shape, and proportions [5] indicates the output is roughly the correct size, shape, and proportions and has less noise [6] indicates the geometry is very similar to resolution of the training data. The highlighted row indicates the best performing architecture.

7.6. Experiment Analysis and Conclusions

The results of the experiments are indicated in Table 7.2 and the generated geometry from the best performing architecture shown in Figure 7.10.

Based on the completed experiments, DCGANs did not train well. The results showed that even after the various hyperparameters were adjusted, the generator only produced noise. Additionally, the training was very unstable, with vanishing gradients being the primary issue. The different parameters tested on DCGANs did not have an impact on stabilizing the training and the generator continued to produce only noise. The switch to Wasserstein loss was key to stabilize the training.

ACTIVATION FUNCTIONS On its own, using Leaky ReLU in both the generator and the critic did not have an impact on the training, but when also using lrDecay, it helped to ensure that the WGAN produced geometry that had roughly the correct size, shape and proportions.

LEARNING RATE DECAY When implemented alone with the base architecture, lrDecay did not benefit the training. However, using lrDecay in combination with Leaky ReLU helped to ensure that the WGAN produced geometry that had roughly the correct size, shape, and proportions.

Based on the ablation study, it was determined that a 3D implementation of WGAN with some additional modifications allowed for stable training. I was unable to reproduce a stable training of the 3D GAN model architecture when it was trained on the building geometry data set. The key factors that impacted the training were using Wasserstein Loss [Arjovsky et al., 2017], using Leaky ReLU as the activation function in the generator and critic, and implementing lrDecay in the generator and critic. After these experiments, the most promising 3D GAN architecture was **G**. Architecture **G** produced geometry that was noisy but it had roughly the correct size, shape, and proportions as the training data model.

After developing the initial 3D WGAN architecture, it was necessary to test the architecture with more data points. Initially, tests were run on several of the architectures with a data set containing 10 models. The 10 models used are described in Appendix A.1. Then, it was necessary to scale up even more to test on at least 100 models. Training on 10 models on the PC was slow, and training took almost an entire day. Therefore, training with 100 models was also not feasible on the PC since the training time was too long. To continue training, I set up access to DHPCC so that I could run experiments using the larger training data set of at least 100 3D models. The data set is described in Appendix A.2. The first test of architecture **G** with a 10 and then 100 model data set resulted in some interesting results. The resulting geometry no longer looked like the building shapes of the training data, so more exploration was necessary to improve the GAN architecture.

8. Refining 3D Wasserstein GANs

The results of the initial tests were promising, but it was necessary to test on larger data sets. After testing the WGAN architecture **G** with 100 data points, the results were less promising than the tests on 1 data point. Additional information on which models were included in the 100 model data set and visualizations of some models can be found in Appendix A.2. When architecture **G** was trained on one data point, it was overfitting, but the geometry it generated was accurate in size, shape, and proportion. However, when the same architecture was trained on 100 models, the generator created geometry that was not accurate in size and proportion. Figure 8.1 shows how the results changed when using a larger training data set. This led to another round of experiments that studied additional hyperparameters. Only select visualizations are shown in this chapter, but the results of all architectures that were tested can be seen in Appendix B.

8.1. Experiment Set Up

[completed on DHPCC]

From the initial experiments discussed in Chapter 7, the generator output 3D geometry, but this geometry did not look like building models. After determining that DCGAN remained unstable despite parameter adjustments, I began testing additional changes using only WGAN. The **G** architecture was used as the base architecture with a slight modification of using Sigmoid instead of Tanh as the activation function at the end of the generator. This is because sigmoid outputs a value between 0 and 1 which aligns with the goal of the generator outputting values between 0 and 1 for each voxel to represent whether the voxel is visible or not. Furthermore, the training data set consisted of 100 hand-selected training building models that were similar in size, shape, and height. A list of the training building models used can be found in Appendix A.2. These models were scaled so that the height of all the models was equal to one unit.

8.2. Experiments and Results

Through an additional literature review, a new set of WGAN hyperparameters were identified. These were all parameters that various researchers noted would improve WGAN performance. When these new hyperparameters were added, it was also important to assess their compatibility with existing hyperparameters selected after the first round of experiments. lrDecay, a feature that was found to be helpful with architecture **G**, did not always generate good results when used with the additional hyperparameters that were tested. Therefore, some architectures used lrDecay and others did not. When learning rate decay was not used, the learning rate was LR = 0.00005 for both the generator and the discriminator.

BATCH NORMALIZATION batch norm normalizes the input of the next layer of an ANN by centering and scaling each entry by the population mean and variance across a batch. The original idea of batch norm was to accelerate training and improve stability by reducing the covariant shift [Ioffe and Szegedy, 2015]. However, this has been disputed, and there is no consensus among DL practitioners

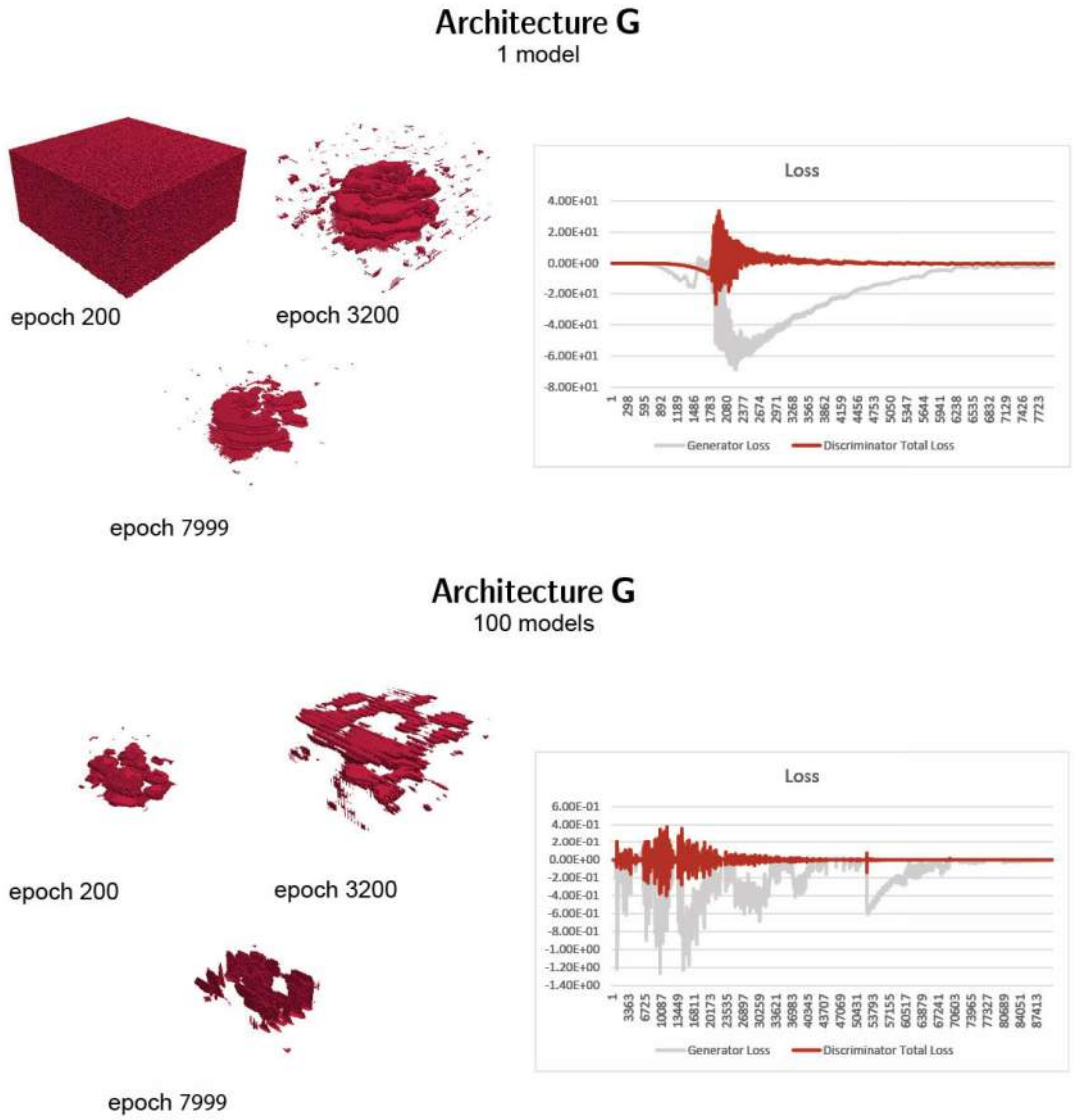


Figure 8.1.: Images showing the results of the **G** architecture of WGAN. The model was trained with 1 and 100 training data points respectively.

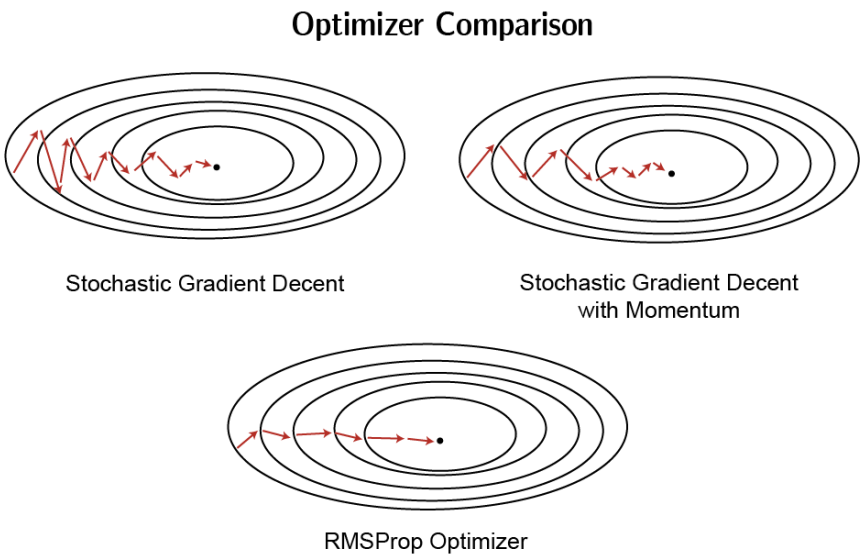


Figure 8.2.: A visual representation of the trajectory of three different optimizers.

about the usefulness of batch norm [Santurkar et al., 2019], especially for GANs Xiang and Li [2017]. Salimans et al. [2016], for example, discuss implementing a modified version of batch norm that they introduced called virtual batch normalization. Salimans et al. [2016] found that batch norm makes the output highly dependent on several other inputs. Therefore, they instead used a virtual batch which is chosen once and fixed at the start of training. However, this method is computationally expensive because it requires forward propagation on two batches. Xiang and Li [2017] also found that batch norm can decrease the stability, efficiency, and quality of GAN compared to their alternative normalization method, weight normalization. Since existing research conflicts, it was worth implementing batch norm to understand the impact as it pertains to generating 3D geometry.

ADAM OPTIMIZER The ADAM optimizer [Kingma and Ba, 2014] extends an adaptive version of stochastic gradient decent. There are two adaptive gradient decent algorithms which ADAM is based on, AdaGrad and root mean squared propagation (RMSProp). AdaGrad has a learning rate per parameter, which helps increase the learning rate when there are few parameters. Therefore, it performs better when there are sparse gradients. RMSProp, described in more detail in the next section, was developed to speed up mini batch learning. ADAM combines these two concepts in its implementation. It is currently the most popular optimization algorithm.

RMS PROP Like ADAM, RMSProp, is based on the concepts of stochastic gradient decent. The RMSProp optimizer Hinton [2018] is a gradient-based adaptive learning rate method, which means that it uses the gradient to minimize the objective function. RMSProp tracks a running sum of the average of the gradients squared. The learning rate is then adapted by dividing the learning rate by the average of the gradients squared Hinton [2018]. This helps prevent the algorithm from getting stuck on saddle points. It is currently the second most popular optimization algorithm.

When comparing RMSProp with ADAM with the same learning rate, RMSProp has a high momentum and reaches further before changing direction, which means it is faster, but it can sometimes get stuck at saddle points in scenarios where ADAM does not. A visual example of this can be seen in Figure 8.2. There is no clear consensus on which is better. Both optimizers are commonly used. In the research discussed in previous chapters, architectures **A** through **H** all used the ADAM optimizer, but for architectures **J** through **W**, both ADAM [Kingma and Ba, 2014] and RMSProp Hinton [2018] were tested. The use of RMSProp is noted in the table. When no optimizer is noted, ADAM is used since this was described in the base architecture.

WEIGHT CLIPPING As Sohrab [2003] described, "there exists a real number such that, for every pair of points on the graph of this function, the absolute value of the slope of the line connecting them is not greater than this real number". The smallest bound that this real number can be is called the Lipschitz constant. Due to the theory of Wasserstein metrics, WGANs must maintain a Lipschitz constant for the discriminator that is less than 1 to train. To ensure that the GAN loss function has a Lipschitz constant bounded by 1, weight clipping is introduced. Weight clipping clips the weights if their norm is too large. When weight clipping is implemented, it keeps the Lipschitz constant under control.

GRADIENT PENALTY Despite having been commonly applied, weight clipping causes a number of problems when applied to WGANs. To understand these problems, remember that for WGANs, the critic learns a value function that is then able to output the value representing the distribution of realness. The value function estimates how good it is to take an action given a specific set of observed features. The value surface is the representation of a multidimensional value function. Gradient penalty and weight clipping both impact the value function.

Weight clipping results in weights that are close to the boundary conditions as shown in the top right graph in Figure 8.3. This results in a pathological value surface, which means that the resulting value surface is contrary to intuition [Gulrajani et al., 2017; Chen and Tong, 2017]. The impact is that the value function that the critic learns has a pathological surface, meaning that it cannot represent the true distribution of buildings well. Instead of clipping the weights, Gulrajani et al. [2017] proposed a penalty to the norm of the gradient of the critic. This penalty is calculated with respect to the critic's input and, therefore, encourages the satisfaction of the Lipschitz constraint without the issues caused by weight clipping. The equation for the gradient penalty added to the critic is as follows:

$$\min_G \max_C \mathbb{E}_{z\sim p_z(z)}[C(G(z))] - \mathbb{E}_{x\sim p_{\text{data}}(x)}[C(x)] + \lambda \mathbb{E}_{x\sim p_{\text{data}}(x)}[(\|\nabla_x C(x)\|_2 - 1)^2] \tag{8.1}$$

where λ is the regularization rate and controls the magnitude of the penalty by weighing it against the regular Wasserstein loss. The benefits of weight clipping can be seen in the experiments run by Gulrajani et al. [2017] on the Swiss Roll data set. Figure 8.3 generated by Gulrajani et al. [2017] shows how the gradient penalty kept the gradient norms consistent during training and kept the weights distributed across a normal distribution.

GRADIENT PENALTY AND NORMALIZATION It is important to note that implementing gradient penalty does not work with batch norm in the critic. This is because batch norm changes the way the critic maps geometries to a scalar value. Without batch norm, the critic maps a single input to a single output. With batch norm, the critic maps a whole batch of inputs to a batch of outputs Ioffe and Szegedy [2015]. Gradient penalty is no longer valid because the penalty calculated is the norm of the critic's gradient with respect to each independent input. Gulrajani et al. [2017] instead recommends layer normalization, which does not introduce a correlation between samples. According to this recommendation, architectures **Q-A** and **S-A** use layer normalization in the critic and batch norm in the generator. Using layer normalization leads to the generator only producing noise throughout the training. Therefore, it was also important to test using no normalization in the discriminator. Architectures **Q-B** and **S-B** use no normalization in the critic and batch norm in the generator.

EXPERIMENTS All the parameters discussed above are mentioned in research papers as methods that help stabilize GAN training and help clarify the output. The second round of experiments continued using hyperparameters that were found to be successful from the first round of experiments (Wasserstein loss, Leaky ReLU, and lrDecay) and built on these. The experiments continued to look for changes that would stabilize training and help the generator produce more clear outputs. As noted, lrDecay did not always produce favorable results and, therefore, was only used in select architectures as indicated in Table 8.1.

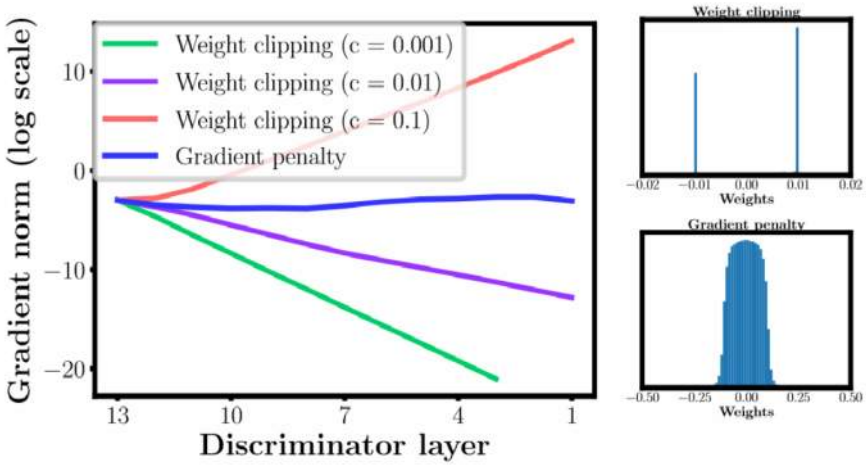


Figure 8.3.: Gulrajani et al. [2017] shows how gradient penalty was able to keep the gradient norms consistent during training while weight clipping caused them to explode or vanish. Weight clipping (top, right) pushes the weights towards two values which are the extremes of the clipping range while gradient penalty (bottom, right) keeps a non-degenerate distribution. Image by Gulrajani et al. [2017].

The experiments were carried out on the DHPCC with a training data set of 100 3D models and each training of 8,000 epochs ran approximately six to eight hours. The experiments and the results are described in Table 8.1. Figure 8.4 visualizes the training of one of the unsuccessful architectures, architecture **M**. Figure 8.5 shows the results of the **R** architecture which had the most successful training outcomes with width and depth of **11** per Table 9.1.

8.3. Experiment Analysis and Conclusions

After reviewing a number of additional hyperparameters, it became evident that certain parameters had a larger impact. The results are indicated in Table 8.1 and the geometry generated from the architecture with the best performance shown in Figure 8.5. Overall, there was a benefit to using RMSProp over ADAM as the optimizer. Furthermore, the gradient penalty was more beneficial than weight clipping, which aligns with current research [Gulrajani et al., 2017]. After completing the research, I found a survey about different deep learning applications in design. My findings were validated by the notes made by Newton [2019] in their survey, where they implemented a WGAN architecture with gradient penalty to generate skyscrapers.

BATCH NORMALIZATION batch norm had no consistent impact on training. It improved some architectures. For example, adding batch norm to an architecture that used Leaky ReLU and lrDecay improved the output. However, batch norm did not help improve the output when this same architecture was tested with RMSProp instead of ADAM as the optimizer. Furthermore, normalization did not work when gradient penalty was introduced, even when normalization was used properly to avoid the impact on gradient penalty as described in Section 8.2. Based on these findings and tests, it remains inconclusive if batch norm on its own is beneficial for generating 3D building geometry with GANs.

Based on the trade-off that when batch norm is used, gradient penalty cannot be used, I would argue that batch norm is not beneficial for this problem. The benefits of gradient penalty discussed in the associated subsection outweigh benefits seen from applying batch norm.

ID	factors	3D WGAN	
		train	output
G	Leaky ReLU + lrDecay	[S]	[3]
J	Leaky ReLU + lrDecay + batch norm	[S]	[2]
K	Leaky ReLU + RMSProp optimizer + lrDecay	[S]	[2]
L	Leaky ReLU + RMSProp optimizer + lrDecay + batch norm	[S]	[2]
M	Leaky ReLU + RMSProp optimizer	[U]	[2]
N	Leaky ReLU + RMSProp optimizer + batch norm	[U]	[3]
P	Leaky ReLU + lrDecay + gradient penalty	[U]	[1]
Q-A	Leaky ReLU + lrDecay + layer normalization + gradient penalty	[U]	[1]
Q-B	Leaky ReLU + lrDecay + batch norm (G) + gradient penalty	[S]	[2]
R	Leaky ReLU + RMSProp optimizer + gradient penalty	[S]	[4/5]
S-A	Leaky ReLU + RMSProp optimizer + layer normalization + gradient penalty	[U]	[1]
S-B	Leaky ReLU + RMSProp optimizer + batch norm (G) + gradient penalty	[S]	[3]
T	Leaky ReLU + lrDecay + gradient clipping	[S]	[2]
U	Leaky ReLU + lrDecay + batch norm + gradient clipping	[S]	[2]
V	Leaky ReLU + RMSProp optimizer + gradient clipping	[S]	[2]
W	Leaky ReLU + RMSProp optimizer + batch norm + gradient clipping	[S]	[2]

Table 8.1.: List of experiments run to asses GAN architecture with depth and width of **11** per Table 9.1. When learning rate decay was not used, the learning rate was LR = 0.00005. [U] indicates that the training was unstable [S] indicates that the training was stable. [1] indicates the output was noise [2] indicates that the output size, shape, and/or proportions are significantly different than the training data [3] indicates that the output is correct in size or proportion but the shape is not accurate [4] indicates that the output is noisy but roughly the correct size, shape, and proportions [5] indicates the output is roughly the correct size, shape, and proportions and has less noise [6] indicates the geometry is very similar to resolution of the training data. The highlighted row indicates the best performing architecture.

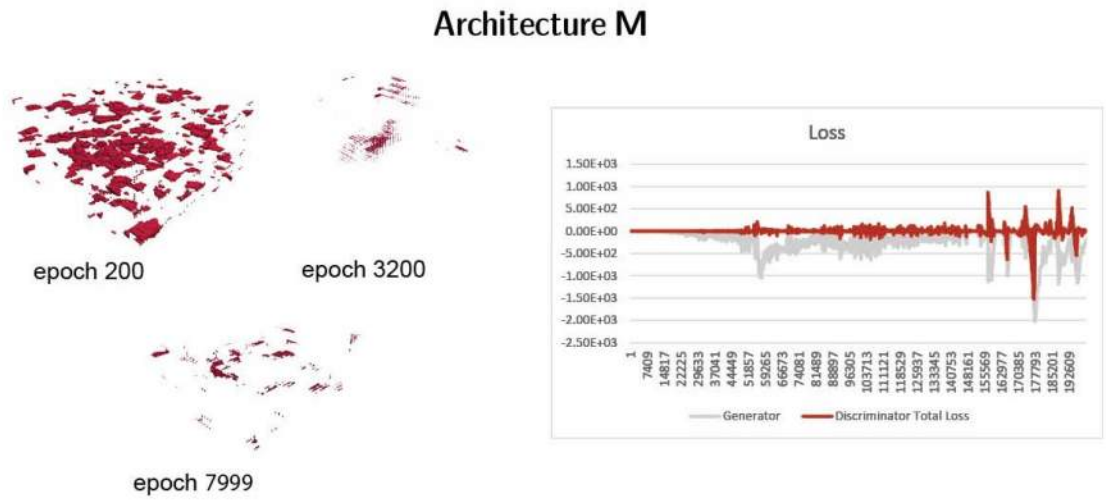


Figure 8.4.: Images showing the results of the **M** architecture and graphs showing its loss which show the unstable training and unsuccessful results. The model was trained with 100 training data points.

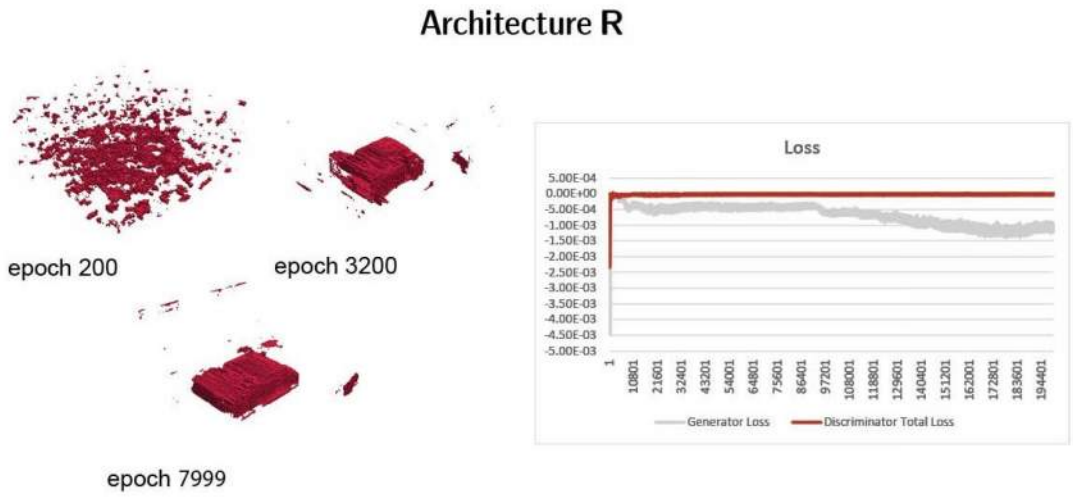


Figure 8.5.: Images showing the results of the most successful, **R** architecture, and graph showing its loss. The model was trained with 100 training data points.

RMS PROP When comparing architectures that used the ADAM and RMSProp optimizers, results show that RMSProp brought benefits to GAN performance when combined with other hyperparameters. The most successful architecture uses RMSProp. An interesting result is that ADAM works best with lrDecay while RMSProp works best with a set learning rate. Due to time limits, I was only able to test RMSProp with the recommended learning rate of 0.00005. An opportunity for future research would be to further test and tune the learning rate to see if this can bring even better results.

WEIGHT CLIPPING AND GRADIENT PENALTY Based on existing research, [Gulrajani et al., 2017], I expected that weight clipping would not be beneficial and that gradient penalty should be used instead. This aligned with the results of the experiment. Gradient clipping did not show a positive impact on training. Implementing gradient penalty did have a positive impact, and the architecture that produced the best results used gradient penalty.

Of the different architectures tested, architecture **R** generated the best results. The result was geometry of a size, shape, and proportion similar to the training data. It also incorporated features of the training data, such as fairly flat walls and sloped roofs, which can be seen in Figure 8.5. This is based on experiments using architecture **11** (Table 9.1) which is an architecture with 4 layers with {48-24-12-2} channels.

9. Network Depth and Width

9.1. Experiment Set Up

[completed on DHPCC]

All architectures described in Chapter 7 and 8 were tested with architecture **11**. Architecture **11** has 4 layers with {48-24-12-2} channels. Then only the most promising architectures were tested with different depths, widths, and kernels. These were architectures **G**, **J**, and **R**. Only select visualizations are shown in this chapter, but the results from all architectures that were tested can be seen in Appendix B.

9.2. Experiments and Results

The initial network architecture was intentionally shallow and narrow. It was selected for the first tests to ensure that these models trained quickly. These early tests used a network depth of 3 with a maximum of 24 channels. After these initial tests, the network size was increased to 4 layers and {48-24-12-2} channels because this generated better results while still running in a relatively short period of time. This larger network was successful when testing different hyperparameter adjustments. This four-layer network structure was the 11th structure tested, so it was labeled architecture **11** and all networks (**A** through **W**) were tested with this structure. The results of tests from **A** through **G** are discussed in Table 7.2, the rest are shown in Table 9.3. Then, only the ones that were the most successful were tested with additional structures. The additional structures are summarized in Table 9.1.

KERNEL SIZES In the field of computer vision, it is generally agreed that it is best to use smaller kernel sizes. This has resulted in many CNN architectures for 2D applications using 3x3 kernels. The 3D GAN architecture of Wu et al. [2016] uses a 4 x 4 x 4 kernel with a geometry space of 64 x 64 x 64. Because of this, the research of this thesis also uses a 4 x 4 x 4 kernel as the starting point. However,

ID	factors
11	4 layers 48-24-12-2
12	kernel size 6x6x6
13	kernel size 8x8x8
14	5 layers 96-48-24-12-2
15	5 layers 192-96-48-24-2
16	10 layers channels 96-96-48-48-24-24-12-12-2-2
17	10 layers channels 192-192-96-96-48-48-24-24-2-2
18	10 layers channels 512-512-256-256-128-128-64-64-2-2
19	15 layers channels 512-512-512-256-256-256-128-128-128-64-64-64-2-2-2

Table 9.1.: List of the different network depths and widths tested.

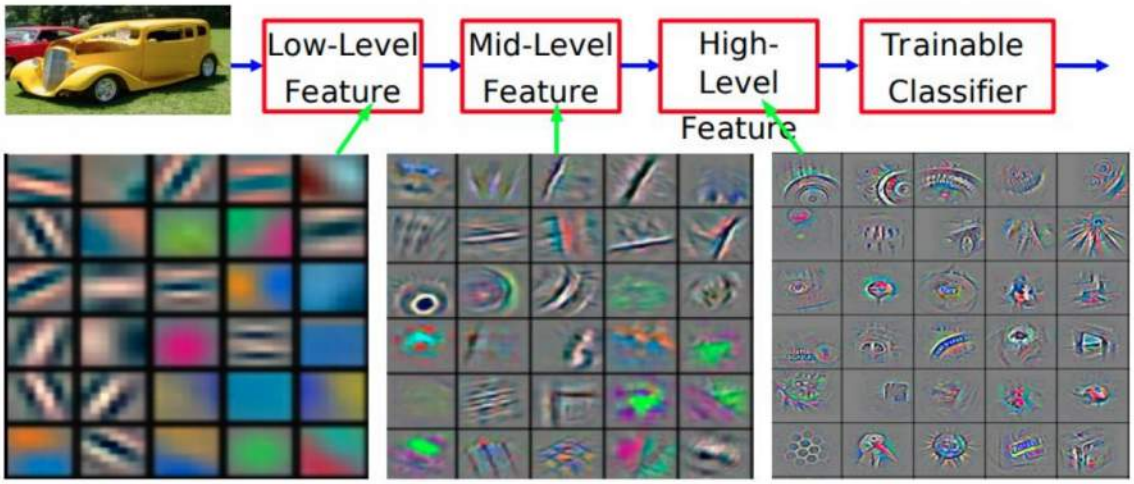


Figure 9.1.: Examples of features that a CNN learns and how these features become more detailed as the network is deeper. Image by Ibrahimli [2022] created using images from Zeiler and Fergus [2013].

different kernel sizes were also tested due to the larger geometry space size. Smaller kernels reduce computational costs, so the sizes that were tested included 6 x 6 x 6 and 8 x 8 x 8.

CHANNELS The number of input channels is called the width of a network. Each channel can learn one specific feature of the network in each layer. Selecting the number of channels is a balance. It is necessary to have enough channels to learn meaningful information, but not too many, so training time is not too long, and so networks do not memorize data set samples. Additionally, remember that the structure of the generator expands each voxel into more voxels through each layer. Therefore, generators usually need many channels in the early layers, where each voxel is expanded into more voxels, so that the channels can represent all the information. The best practice when determining the ANN architecture is to start with a smaller number of channels and then test variations of the architecture that gradually increase the number of channels. The 3D GAN architecture from [Wu et al., 2016] started with 512 channels in their first layer. To simplify this architecture, I started by testing with 48 channels in the first layer. Then, well-performing architectures were tested with 96 channels and 192 channels in the first layer. After running these tests, it was determined that it would also be interesting to review the results of a very wide network. Therefore, I tested two very wide and very deep architectures. Architecture **18R** has a total of ten layers, with the first layer containing 512 channels and architecture **19R** has fifteen layers, with the first layer containing 512 channels.

LAYERS When adjusting ANN architecture, it is also beneficial to add depth when adding width. This is because a network that is too wide and shallow will begin to memorize features instead of learning features that can be generalized to additional data. The more layers that are added to a network, the more detailed the learned features become. Ibrahimli [2022] described this concept shown in Figure 9.1 using images from Zeiler and Fergus [2013]. From the figure, it is possible to see that deeper layers learn more detailed features. The 3D GAN architecture of [Wu et al., 2016] contained five layers. Since starting small is more beneficial, the initial architectures I tested contained only four layers. After determining which hyperparameters were necessary for the best results, I also tested networks with five and ten layers. The deepest model tested had fifteen layers.

PADDING In CNNs, convolutions map an input to an output through the use of a kernel. The size of the output may vary from the size of the input based on the size of the kernel and the stride (how far the kernel is moved). To adjust the size of the output, padding can be added. For convolutional layers,

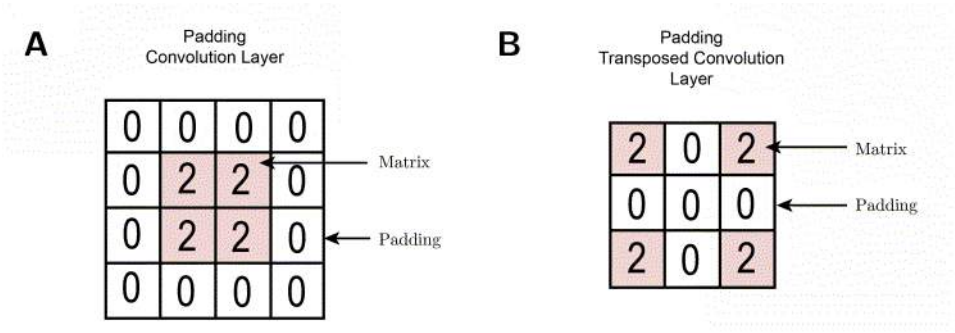


Figure 9.2.: (A) Padding in the convolution layers. (B) Padding in the transposed convolution layers.

padding adds extra voxels on the outside of the geometry space, and for transposed convolutional layers, it creates a larger kernel size by padding between layers. This is shown in Figure 9.2.

For a convolution, the output size is determined by the following formula:

$$output_{size} = \left\lfloor \frac{input\_size + 2 * padding - kernel\_size}{stride} + 1 \right\rfloor \tag{9.1}$$

Therefore, $\lfloor x \rfloor$ is the mathematical floor of x. This means returning the largest integer below or equal to the value of x. For a transposed convolution, the output size is determined by the following formula:

$$output_{size} = (input\_size - 1) * stride - 2 * padding + kernel\_size. \tag{9.2}$$

Using these formulas, the padding can be adjusted to ensure a specific relationship between the input size and the output size.

All experiments in this thesis used the same geometry space of 160 x 160 x 80 and a stride of 2 x 2 x 2. The padding was then adjusted so that the convolution layers output a result in which the output is half the length, width, and height of the input. This means that sometimes the padding was unequal. To demonstrate this, let us review the following scenario. For this thesis, the geometry space is 160 x 160 x 80 voxels. The kernel size and stride is already determined by the network architecture, which means that the padding can vary and will be adjusted to ensure the correctly sized output. This means that when the input is 160 x 160 x 80, the kernel size is 4 x 4 x 4, the stride is 2 x 2 x 2, and the output size needs to be 80 x 80 x 40, the padding needs to be 1 voxel wide. Therefore, padding is used on one side only for each dimension of the matrix (x, y, and z). Some architectures had noise at the boundary conditions as seen in the results of architecture **15J** in Figure 9.3. Noise at the boundary could be caused by uneven padding. To test whether padding had an impact on this noise, two additional variations were tested for architectures **11R**, **15J**, **16R**, and **17R**.

The first variation tested a kernel size of 5 x 5 x 5. Using this size kernel leaves equal padding of 1 voxel on each side of the geometry space when the input, output, and stride values remain the same as mentioned above. The second variation tested filling this equal padding with different values. Padding is automatically filled with zeros in TensorFlow. However, it is also possible to instead reflect the value contained in the matrix. This can also help to reduce the noise around the boundary. The results of these experiments are summarized in Table 9.2. One important aspect to note is to get equal

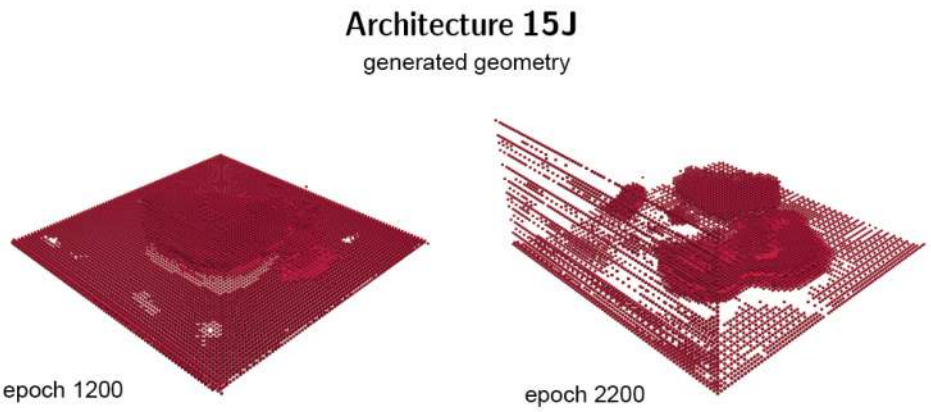


Figure 9.3.: Architecture **15J** had noise at the geometry space boundary during early trainign epochs which could be caused by having unequal padding added during the convolution and transposed convolution layers.

padding on all sides, the kernel was changed to 5 x 5 x 5. So, although this leads to successful results, it is unclear if the success is due to the kernel size or the equal padding. Another test that should be performed is to change the geometry space size to provide equal padding when using a 4 x 4 x 4 kernel. This will make it possible to narrow down if the kernel or the padding impacted the result. Due to the time constraints of this thesis, it was not possible to run this additional test.

ID	factors	padding		
		unequal	equal	reflect
11R	R (Table 8.1) with 4 layers 48-24-12-2	[S][4/5]	[S][4/5]	[S][3]
15J	J (Table 8.1) with 5 layers 192-96-48-24-2	[S][4]	[U][1/2]	[U][1]
16R	R (Table 8.1) with 10 layers (2)96-(2)48-(2)24-(2)12-(2)2	[S][5]	[S][5]	[S][5]
17R	R (Table 8.1) with 10 layers (2)192-(2)96-(2)48-(2)24-(2)2	[S][5]	[S][6]	[S][4/5]

Table 9.2.: List of experiments run to see the impact of padding. [U] indicates that the training was unstable [S] indicates that the training was stable. [1] indicates the output was noise [2] indicates that the output size, shape, and/or proportions are significantly different than the training data [3] indicates that the output is correct in size or proportion but the shape is not accurate [4] indicates that the output is noisy but roughly the correct size, shape, and proportions [5] indicates the output is roughly the correct size, shape, and proportions and has less noise [6] indicates the geometry is very similar to resolution of the training data. The red highlighted rows indicate the best performing architectures.

MEMORIZING DATA As the architectures produced a geometry that was more clear, it was possible to evaluate if the GANs were just memorizing the data set or if they were generating novel buildings. During training, **17R** with equal padding was memorizing specific samples as shown in Figure 9.8, but it was important to determine if it could also produce novel geometry. To review if the **17R** models were memorizing the training data, the **17R** architecture with unequal padding and the **17R** architecture with equal padding were used to generate 100 building geometries each. Then, I selected several of the generated buildings and visually compared them with the data set. Figure 9.4 shows a selection of these comparisons.

There are instances where the generated geometry repeats similar forms. However, there is also a fine line between memorizing and creating new buildings that look similar to the existing data set.

For example, *model 43*, generated by architecture **17R** and shown in Figure 9.4, has massing that is similar to the models in the data set. The massing of the generated model is very similar to *mesh3513* and *mesh4362* from the data set. But when looking more closely, there are also some key differences. When looking at how the different roof slopes come together, the roofs of *model 43* are all at the same height. In *mesh3513*, on the other hand, the roof of the central room is higher and *4362* has a similar feature. Furthermore, the slopes at the roof connections in *model 43* resemble the changes in slopes that can be found in *mesh0109* and *mesh2529* more closely. This suggests that the massing and the roof features are learned from different models and combined to generate a new building mass.

Model 60 generated by architecture **17R** with equal padding and shown in Figure 9.4 is another excellent example. There are only three flat-roofed houses in the data set; therefore, it is easier to compare the generated model with the data set models. In this case, the massing of the generated model is very similar to *mesh1329* from the data set, but when looking at the detail, it incorporates the parapet detailing from *mesh0648* and *mesh3480*. The detail of the parapet is not found in *mesh1329*.

However, with some of the other models, it is harder to judge. *Model 8* generated by architecture **17R** with equal padding, for example, has massing very similar to *mesh3240* including the inset porch on the left wall. There are some clear differences, however, such as the wall on the right of the model does not have openings, while the same wall in *mesh3240* does. *Model 8* also has what looks like the start of a chimney at the top right, as observed in Figure 9.4, but it is hard to tell exactly what the extrusion is. *Mesh3240* has an extrusion for a dormer window in the same area. In this case, it is hard to tell whether the generated model replaced the extrusion with a learned chimney feature such as the chimneys in *mesh0456* and *mesh0718*, or if it just poorly replicated the dormer window. Additionally, it could actually suggest the beginning of the model learning to remove the dormer and replace it with a roof section which would demonstrate that it is learning features.

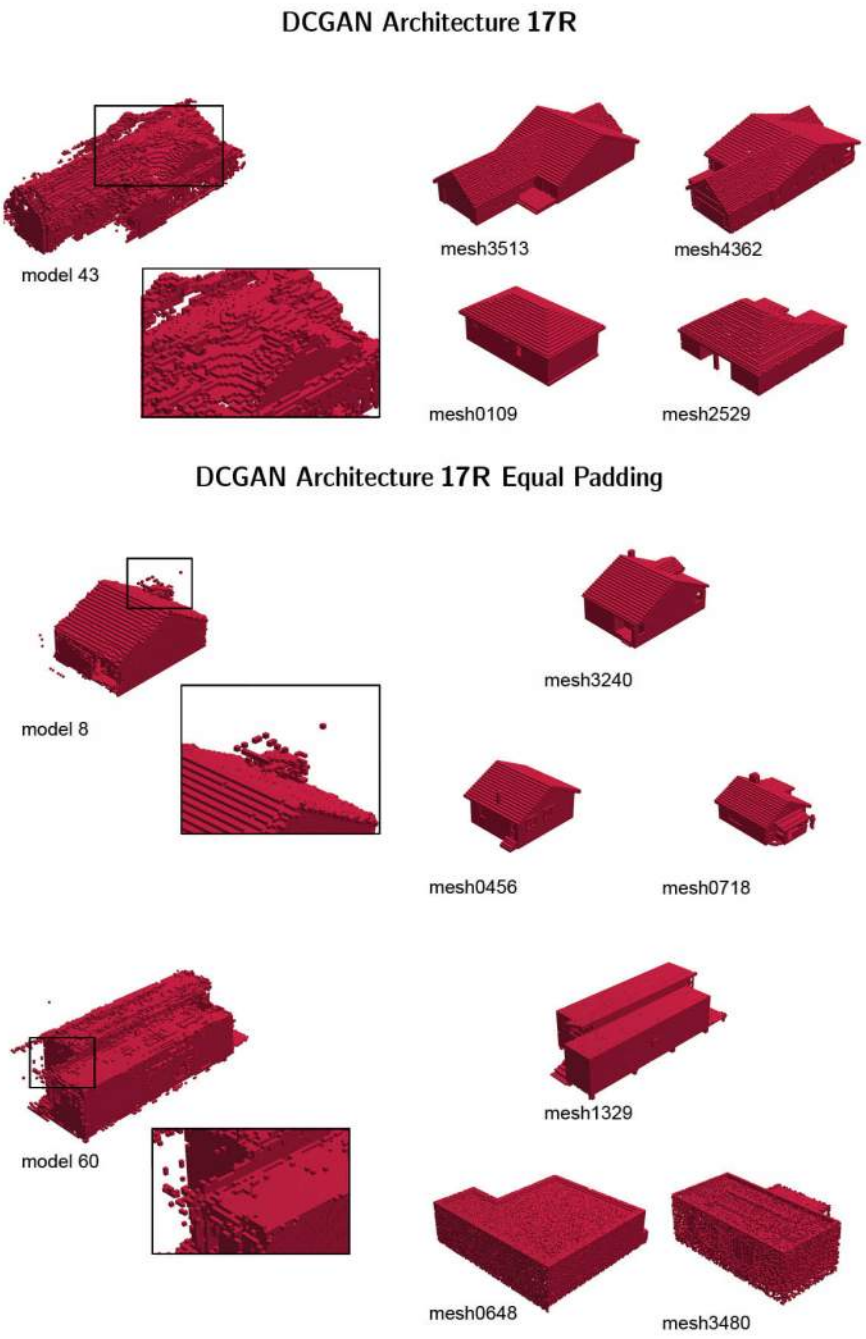


Figure 9.4.: A selection of models generated by architecture **17R** with unequal and equal padding and a comparison to certain training data points to analyze to what extent the DL model is memorizing the training data set.

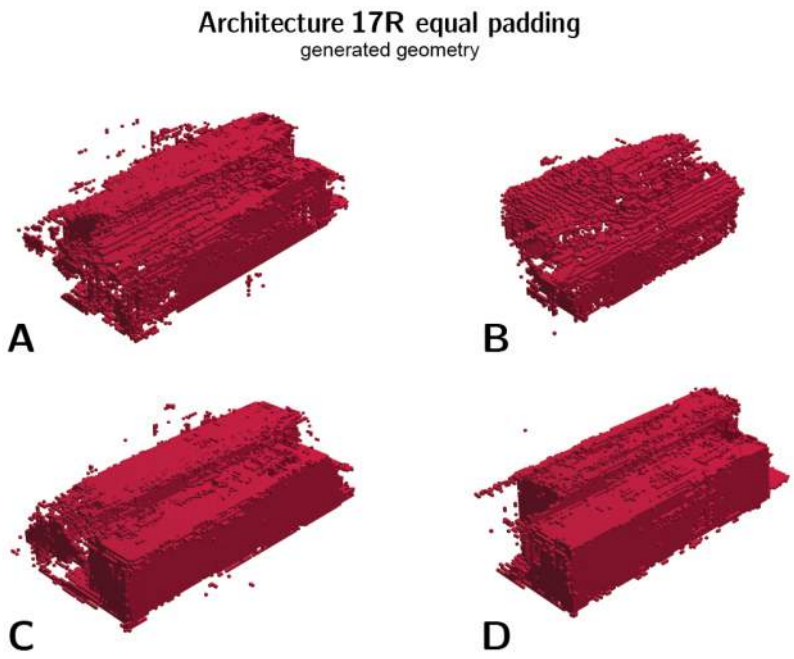


Figure 9.5.: A selection of models generated by architecture **17R** with equal padding which all have similar massing but different building features.

Furthermore, when looking at multiple different generated models from architecture **17R** with equal padding, there is evidence that the DL model is trying to generate buildings with similar massing but with different building features. Figure 9.5 demonstrates an example. Figures 9.5 (A) and (B) appear to have variations of a sloped roof structure. Figures 9.5 (C) and (D) appear to have variations of a flat roof structure, with (C) having no parapet and (D) appearing to have a parapet.

There definitely seems to be some level of memorization happening in the 3D models output during training and for those generated after training, however, there are also signs that the DL models are combining features from numerous different training data points and learning building features such as roof slopes, details of the parapet, and possibly chimney shapes.

ID	factors	training data size	
		10	100
11G	G (Table 7.2) with 4 layers 48-24-12-2	[4]	[S][3]
11H	H (Table 7.2) with 4 layers 48-24-12-2	[2]	[S][2]
11J	J (Table 8.1) with 4 layers 48-24-12-2	[2]	[S][2]
11K	K (Table 8.1) with 4 layers 48-24-12-2	[x]	[S][2]
11L	L (Table 8.1) with 4 layers 48-24-12-2	[x]	[S][2]
11M	M (Table 8.1) with 4 layers 48-24-12-2	[x]	[U][2/3]
11N	N (Table 8.1) with 4 layers 48-24-12-2	[x]	[U][3]
11P	P (Table 8.1) with 4 layers 48-24-12-2	[x]	[U][2]
11Q-A	Q-A (Table 8.1) with 4 layers 48-24-12-2	[x]	[U][1]
11Q-B	Q-B (Table 8.1) with 4 layers 48-24-12-2	[x]	[S][2]
11R	R (Table 8.1) with 4 layers 48-24-12-2	[x]	[S][4/5]
11S-A	S-A (Table 8.1) with 4 layers 48-24-12-2	[x]	[U][1]
11S-B	S-B (Table 8.1) with 4 layers 48-24-12-2	[x]	[S][3]
11T	T (Table 8.1) with 4 layers 48-24-12-2	[x]	[S][2]
11U	U (Table 8.1) with 4 layers 48-24-12-2	[x]	[S][2]
11V	V (Table 8.1) with 4 layers 48-24-12-2	[x]	[S][2]
11W	W (Table 8.1) with 4 layers 48-24-12-2	[x]	[S][2]
12G	G (Table 7.2) with kernel size 6x6x6	[1]	[U][2]
12J	J (Table 8.1) with kernel size 6x6x6	[1]	[S][2]
12R	R (Table 8.1) with kernel size 6x6x6	[x]	[S][2]
13G	G (Table 7.2) with kernel size 8x8x8	[1]	[S][2]
13J	J (Table 8.1) with kernel size 8x8x8	[1]	[U][1]
13R	R (Table 8.1) with kernel size 8x8x8	[x]	[S][3]
14G	G (Table 7.2) with 5 layers 96-48-24-12-2	[1]	[U][2]
14J	J (Table 8.1) with 5 layers 96-48-24-12-2	[1]	[U][2]
14R	R (Table 8.1) with 5 layers 96-48-24-12-2	[x]	[S][3]
15G	G (Table 7.2) with 5 layers 192-96-48-24-2	[1]	[U][2]
15J	J (Table 8.1) with 5 layers 192-96-48-24-2	[U][1]	[S][4]
15R	R (Table 8.1) with 5 layers 192-96-48-24-2	[x]	[S][4]
16G	G (Table 7.2) with 10 layers 96-96-48-48-24-24-12-12-2-2	[x]	[U][2]
16J	J (Table 8.1) with 10 layers 96-96-48-48-24-24-12-12-2-2	[x]	[U][2]
16R	R (Table 8.1) with 10 layers 96-96-48-48-24-24-12-12-2-2	[x]	[S][5]
17R	R (Table 8.1) with 10 layers 192-192-96-96-48-48-24-24-2-2	[x]	[S][5]
18R	R (Table 8.1) with 10 layers 512-512-256-256-128-128-64-64-2-2	[x]	[S][5]
19R	R (Table 8.1) with 15 layers (3)512-(3)256-(3)128-(3)64-(3)2	[x]	[S][1/2]

Table 9.3.: List of experiments run to tune 3D WGAN architecture to generate more refined geometry. All architectures were tested with network depth and width of **11** per Table 9.1 and only the most promising architectures were then tested with different depths, widths, and kernels. When learning rate decay was not used, the learning rate was LR = 0.00005. [x] indicates this combination was not tested with the noted quantity of training samples. [U] indicates that the training was unstable [S] indicates that the training was stable. [1] indicates the output was noise [2] indicates that the output size, shape, and/or proportions are significantly different than the training data [3] indicates that the output is correct in size or proportion but the shape is not accurate [4] indicates that the output is noisy but roughly the correct size, shape, and proportions [5] indicates the output is roughly the correct size, shape, and proportions and has less noise [6] indicates the geometry is very similar to resolution of the training data. The red highlighted rows indicate the best performing architectures.

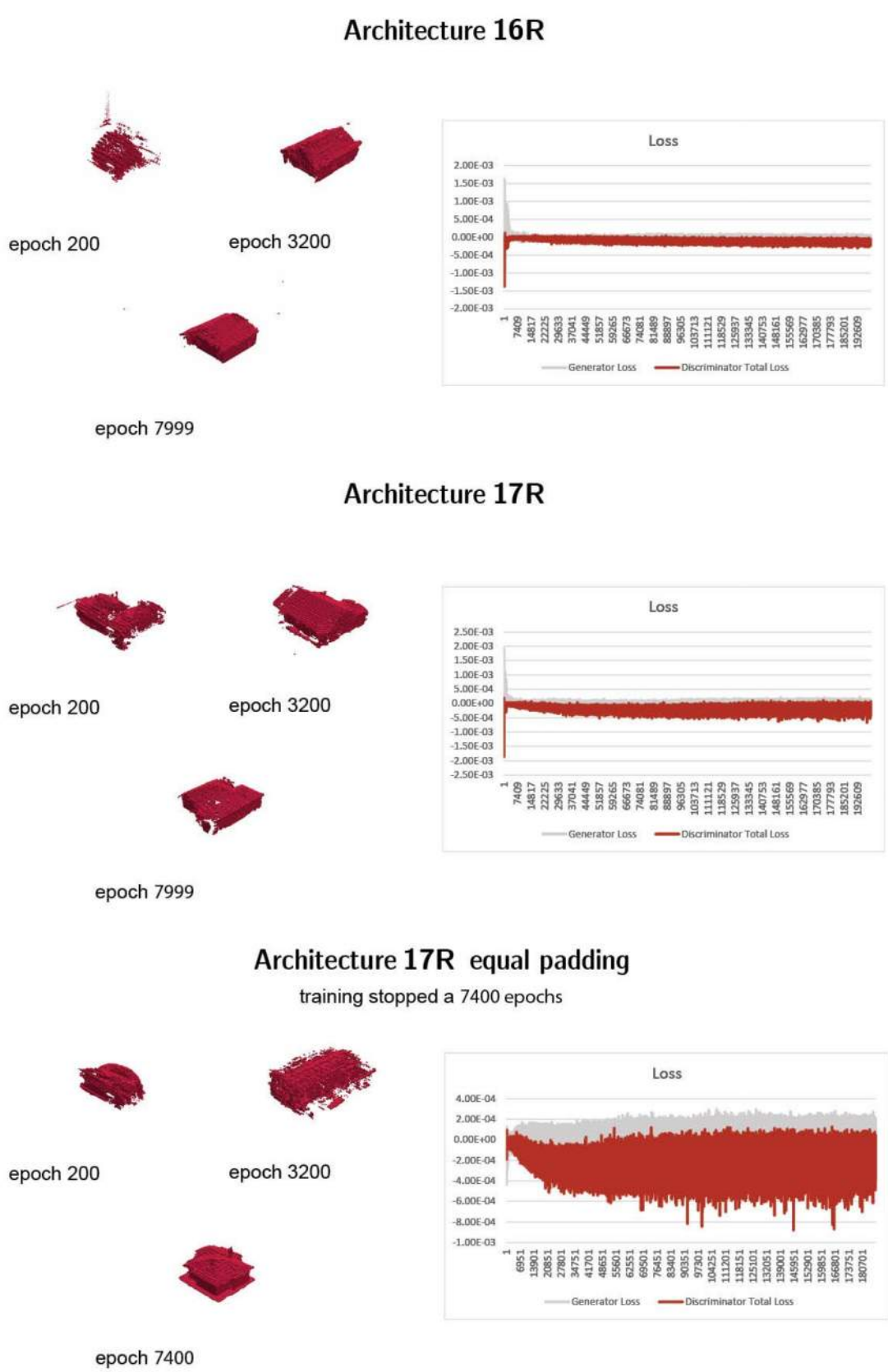


Figure 9.6.: The generated images of three of the best performing architectures, **16R**, **17R** with unequal padding, and **17R** with equal padding, and graphs showing their respective loss.

9.3. Experiment Analysis and Conclusions

It was important to test different network depths and widths along with different kernel sizes because these factors significantly impact the resources needed to train a network. They also have an impact on the output. The results of these experiments are indicated in Table 9.3, and the geometry generated from the best-performing architectures is shown in Figure 9.6. There were benefits to adjusting network depth and width, but adjusting the kernel size did not benefit training.

KERNEL SIZES It was expected that changing to a larger kernel size would not have a positive impact on training. It was still important to test this option due to the larger geometry space used in this research. The results of the experiments aligned with the expected results. For some architectures, using a larger kernel size helped the model learn larger building features, such as walls, but did not produce entire buildings that were the correct size, shape, and proportions.

CHANNELS Increasing the number of channels did not always have a positive result. However, for select architectures that did not perform well with fewer channels, increasing the number of channels did improve the output. For architecture **J**, for example, the previous iterations did not generate meaningful geometry, but with more layers and more channels, it performed fairly well. Increasing the number of channels also helped to improve the performance of the best performing architecture, architecture **R**. Ultimately, architecture **R** still outperformed architecture **J** both when these architectures had fewer channels and when they had more channels.

The experiments that used 512 channels in the first layer of the generator had mixed results. The **18R** architecture, which was identical to **17R** except for the number of channels, performed similarly to, but not better than, **18R**. The comparison of the best generated geometry from **16R**, **17R**, and **18R** is shown in Figure 9.7. The image shows how increasing the channels helped produce a clearer output between **16R** and **17R**, but the clarity is very similar between **17R** and **18R**. However, when adding both channels and depth in architecture **19R**, the model no longer outputs any geometry. Since **17R** performed similarly to **18R**, this suggests that there is not always a benefit to increasing the number of channels. At some point, there are enough channels to learn the features without the need for more. It is necessary to perform more tests on the number of channels, especially when training on a significantly larger data set.

Based on these results, having more channels is beneficial, but they should be balanced with the number of layers and training samples. Additionally, there is a point where more layers do not necessarily improve performance.

LAYERS Increasing the number of layers by a large amount did have an impact on training for well-performing architectures. For architecture **R**, for example, increasing from four layers to ten layers helped reduce the noise in the output. For the architectures tested, architectures that performed well with shallower and narrower structures continued to perform well with wider and deeper structures. For some poorly performing architectures, using wider and deeper networks improved their performance. However, architectures that consistently performed well still outperformed architectures that improved with wider and deeper structures.

Based on these results, I conclude that when the depth and width of the network are adjusted, these parameters can help to lessen the amount of noise in architectures that are performing well. It also shows that if an architecture performs well with a shallow and narrow structure, it will likely do better overall.

PADDING Architectures **11R**, **16R**, and **17R** did have less noise when tested with equal padding. Furthermore, the overall results were equal to or better than when tested with unequal padding. For architecture **17R**, using equal padding had a large impact on the results and allowed this architecture to produce clear geometry that showed the features of the training data in more detail than any other

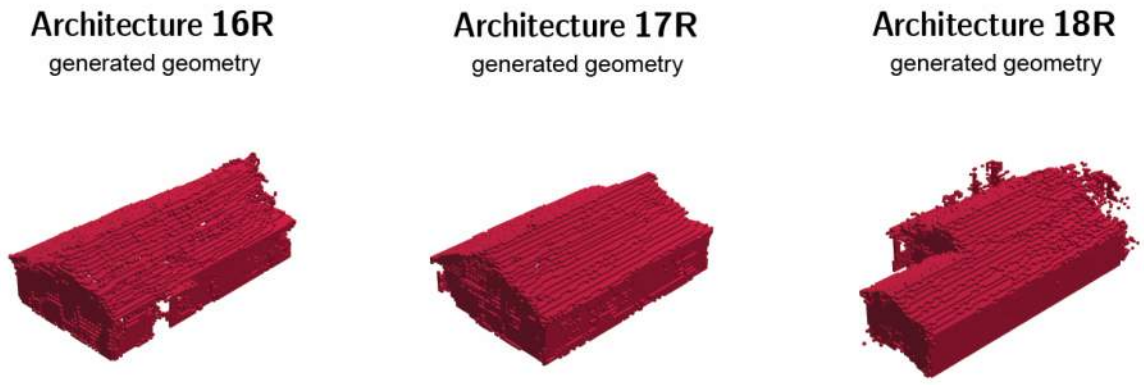


Figure 9.7.: Images showing the best generated building produced by **16R**, **17R**, and **18R** architectures. These all have 10 layers but the number of channels in each layer is increased. The models were trained with 100 training data points.

architecture. However, this architecture also had a tendency to memorize different models in the training data.

When using reflected values from the matrix as the padding instead of zeros, there was no positive impact on the output. In some architectures, it even hurts the training, with the output being only noise. For other architectures the reflect padding did not hurt performance but also did not improve it above the performance with unequal padding. This suggests that using geometry space, kernel, and stride sizes that allow for equal padding on all sides of the geometry space can help to reduce noise in the output and can help the network learn features with a higher level of detail.

An important factor that requires more research is the relationship between kernel size and the amount of padding. To test equal padding, it is possible to either change the geometry space or change the kernel size. Due to time constraints, it was not possible to test equal padding by changing the geometry size during this thesis. Instead, to get equal padding with a geometry size of 160 x 160 x 80, the kernel size was changed to 5 x 5 x 5. This change made the output more clear, but also meant that the network had a tendency to memorize certain geometry from the training data. Therefore, it is also necessary to test the impact of equal padding when it is achieved by changing the geometry space when using a kernel of 4 x 4 x 4. This would allow for a more clear understanding of which part affected training. It could help clarify whether there are certain characteristics that lead the network to memorize the training data.

MEMORIZING DATA Based on reviewing the generated models from architecture **17R**, it is possible to determine that the tested GAN are sometimes memorizing data points. This is shown in Figure 9.8, since the models generated during training are almost identical to those in the training data set. When reviewing 100 models that the architectures generated, however, there are also signs that the model is learning features. As described in Figure 9.4, there are results that suggest that the model combines features from numerous different training data points. The geometry that the models output also shows signs that the model is learning building features such as roof slopes, parapet details, and chimneys.

There appears to be some level of memorization that is happening, and it is important to address this in future research. Two main factors can contribute to a network memorizing training data, and this is the network width and the size of the training data set. One possible cause of the memorization is that the model became too wide with architecture **17**, which means that the number of channels should

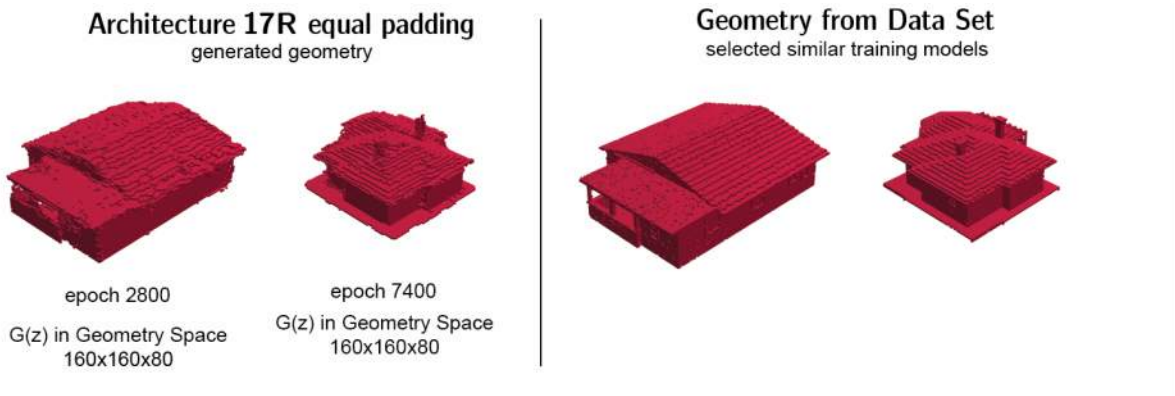


Figure 9.8.: Images showing the results of the **17R** architecture with equal padding. The geometry on the left is the output from the model during training and the geometry on the right shows the training data it memorized. The model was trained with 100 training data points.

be reduced. But the solution may not be that simple. Reducing the number of channels can also have an impact on the resolution of the output. Instead of reducing the number of channels, another option is to implement concepts such as memorization rejection. Memorization rejection penalizes networks when the generated geometry is too similar to the training data set. The penalty encourages the network to learn to generate unique geometry. Research has shown that memorization rejection can help prevent the network from memorizing data while maintaining a high-resolution output [Bai et al., 2022].

These results also suggest that it is important to train and test on a much larger data set. The data set of 100 models could simply be too small. Other deep learning models are trained on thousands and sometimes millions of data points. ImageNet, a well-known data set containing different pictures used to train machine learning models, contains over 1.2 million images. With smaller data sets, it is possible that the number of parameters exceeds the available data. For these reasons, it is also important to further test the proposed architectures with a larger data set.

Of the different architectures tested, architecture **R** continued to generate the best results. The output was geometry of similar size, shape, and proportion to the training data. It also showed signs of learning more detailed characteristics of the data, such as roof slopes, recesses, and chimneys. Based on testing different options, architectures **16R**, **17R** with unequal padding, and **17R** with equal padding generated the most clear geometry. Architecture **16** has 10 layers with {96-96-48-48-24-24-12-12-2-2} channels and architecture **17** has 10 layers with {192-192-96-96-48-48-24-24-2-2} channels as shown in Table 9.1.

10. Network Input Formats and Training

In addition to changing the hyperparameters of the GAN, it was important to understand how different changes to the input for the generator and critic also impact the training. The results of all the architectures that were tested can be seen in Appendix B.

10.1. Critic Input

MULTI-LABEL INPUTS (not implemented) Now that a successful GAN architecture is developed, a possible extension of the research is to generate geometry with multiple labels. As described in Section 5.4 and shown in Figure 6.1, selected labels of 'wall', 'window', 'roof', and 'door', were fairly accurately assigned in all 3D models of the pre-processed data set. The initial intent was to use these labels as input to the critic so that the generator could then generate geometry that comprised of four labels, 'wall', 'window', 'roof', and 'door'. However, after experimenting with this concept and doing some further research, it was discovered that GANs can only generate a single label. This is due to the limitation of the generator. Some researchers have addressed this problem by connecting generator and discriminator pairs that each learn a specific label. Since the research in this thesis focuses on generative geometry instead of labeling data, the approach of using multiple generator and discriminator pairs is not possible. It is not possible because the window geometry generated by a generator discriminator pair that learns the window label needs to align correctly with the wall geometry generated by a generator discriminator pair that learns the wall labels. Otherwise, the window will not be located within the wall.

Instead of using multiple labels, a possible path to generate geometry with more than two labels would be to generate the geometry with the GAN architecture described in this thesis and afterward use this geometry as input for another ANN that learns to label the voxels with the correct label. Based on the success of ANN in labeling data, this method has potential. Due to the time limitations of this thesis, this method was not able to be tested. It remains an opportunity for future research.

SHELL 3D MODELS AND SOLID 3D MODELS The data in the data set comes as shell 3D models. When these shell models were used, however, some architectures learned how to generate certain features such as walls and then repeated this feature many times. One architecture that did this was architecture **11G**, and an example of an output from this architecture can be seen in Figure 10.1. To test whether solid 3D models would address this problem, these shell 3D models were filled to produce solid 3D models as described in Section 6.3 and Figure 6.5. To test the impact of solid 3D models, the network architectures of each performance level [2 to 5] were randomly selected. Then these were trained with solid 3D models.

The results, summarized in Table 10.1, show that there was no positive impact on training when successful architectures were trained with solid models. If the geometry they produced was roughly the correct size, shape, and proportions, then training this architecture on solid 3D models did not benefit the architecture. Additionally, architectures that had the problem of creating many walls, such as architecture **11G**, did not improve their training on solid models. After training on solid models, architecture **11G** generated only a small number of pixels, as shown in Figure 10.1. One architecture that was generating poor results, where the output was not noise but was significantly different in size, shape, and/or proportion than the training result, performed better when trained with solid models.

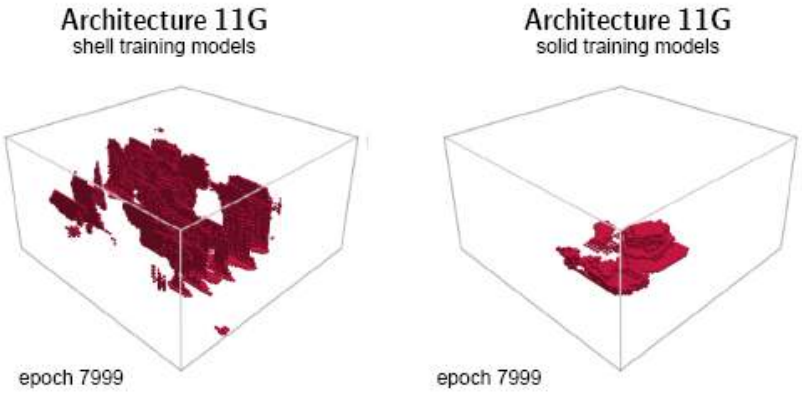


Figure 10.1.: On the left, an example produced by architecture **11G** of how the model learned to generate wall-like layers and kept repeating this feature. On the right, an example produced by architecture **11G** when trained on solid models. The models were trained with 100 training data points.

After training the low-performance architecture, **11N**, with a data set that contained solid 3D models, the network generated geometry that was roughly the correct size, shape, and proportions. This shows that training on solid models is possible; however, the architecture needs to be tuned differently so it does not help with the goals of this thesis.

The results of the tests indicate that training with shell 3D models versus solid 3D models requires different hyperparameter tuning. The results of the training are shown in Figure 10.2. Knowing that training with solid models does not improve output is good because training with solid 3D models is not preferred. In reality, it is best to generate the most accurate geometry, and since buildings have empty space inside, the GANs should generate building models that also have empty space inside. The results of these tests do show that if GANs are used to generate solid 3D models they should also be trained on solid 3D models. Additionally, when using solid 3D models, the hyperparameters need to be tuned differently based on this requirement.

ID	factors	training data type	
		shell	solid
11G	G (Table 8.1) with 4 layers 48-24-12-2	[S][3]	[S][2]
11N	N (Table 8.1) with 4 layers 48-24-12-2	[U][3]	[S][4]
11R	R (Table 8.1) with 4 layers 48-24-12-2	[S][4]	[U][1/2]
11S	S (Table 8.1) with 4 layers 48-24-12-2	[U][1]	[U][1]
15J	J (Table 8.1) with 5 layers 192-96-48-24-2	[S][4]	[U][1/2]

Table 10.1.: List of experiments run with shell 3D models and solid 3D models. [U] indicates that the training was unstable [S] indicates that the training was stable. [1] indicates the output was noise [2] indicates that the output size, shape, and/or proportions are significantly different than the training data [3] indicates that the output is correct in size or proportion but the shape is not accurate [4] indicates that the output is noisy but roughly the correct size, shape, and proportions [5] indicates the output is roughly the correct size, shape, and proportions and has less noise [6] indicates the geometry is very similar to resolution of the training data.

ID	factors	training data size	
		100	200
11R	R (Table 8.1) with 4 layers 48-24-12-2	[S][4/5]	[S][3]
15J	J (Table 8.1) with 5 layers 192-96-48-24-2	[S][4]	[U][1/2]
15R	R (Table 8.1) with 5 layers 192-96-48-24-2	[S][4]	[S][3]
16R	R (Table 8.1) with 10 layers channels 48-48-24-24-12-12-2-2	[S][5]	[S][3/4]

Table 10.2.: List of experiments run with the larger data set to compare training with 100 and 200 3D models. [U] indicates that the training was unstable [S] indicates that the training was stable. [1] indicates the output was noise [2] indicates that the output size, shape, and/or proportions are significantly different than the training data [3] indicates that the output is correct in size or proportion but the shape is not accurate [4] indicates that the output is noisy but roughly the correct size, shape, and proportions [5] indicates the output is roughly the correct size, shape, and proportions and has less noise [6] indicates the geometry is very similar to resolution of the training data.

LARGER TRAINING DATA SET There are limitations on the size of the training data set based on the currently available data. Therefore, 100 single-story, single-family house building models were selected for the training. Existing research on training 3D GANs suggests that 100 3D models is the minimum quantity needed to generate meaningful results. This led to fairly successful results with architecture **R**, but the geometry output from the generator still had a lower level of detail than the 3D models in the training data set. A possible obstacle to more successful training is that ANNs did not have enough data to extract more meaningful information. The data set was therefore doubled to a total of 200 single-story, single-family house building models. Additional information on which models were included in the data set and visualizations of some models can be found in Appendix A.3. All architectures produced less accurate geometry when trained on larger data sets, as indicated in Table 10.2. However, wider and deeper architectures produced a better output. For example, architecture **16R** performed better than architecture **11R** on the larger data set. The generated geometry was still not as good as those trained on 100 models. Figure 10.3 shows the result of architecture **11R** when trained on 100 and 200 training data points and Figure 10.4 shows the result of architecture **16R** when trained on 100 and 200 training data points.

However, training on larger data sets was not explored to the full extent. Increasing from 100 to 200 models is a small increase compared to training on thousands of models. Further testing is required with significantly larger data sets. A possible solution to obtain larger data sets more quickly is by implementing data set augmentation strategies. Data set augmentation copies and changes the existing data points, for example, by rotating a building model, to generate more training data.

Ultimately, there are many benefits of training on large data sets that could not be explored through this thesis. For example, training on larger data sets helps prevent networks from memorizing training models. Initial research on data set size shows that changes would most likely need to be made to the network width and depth of the network, but these changes are worth the extra time since training on larger data sets can lead to a more robust model.

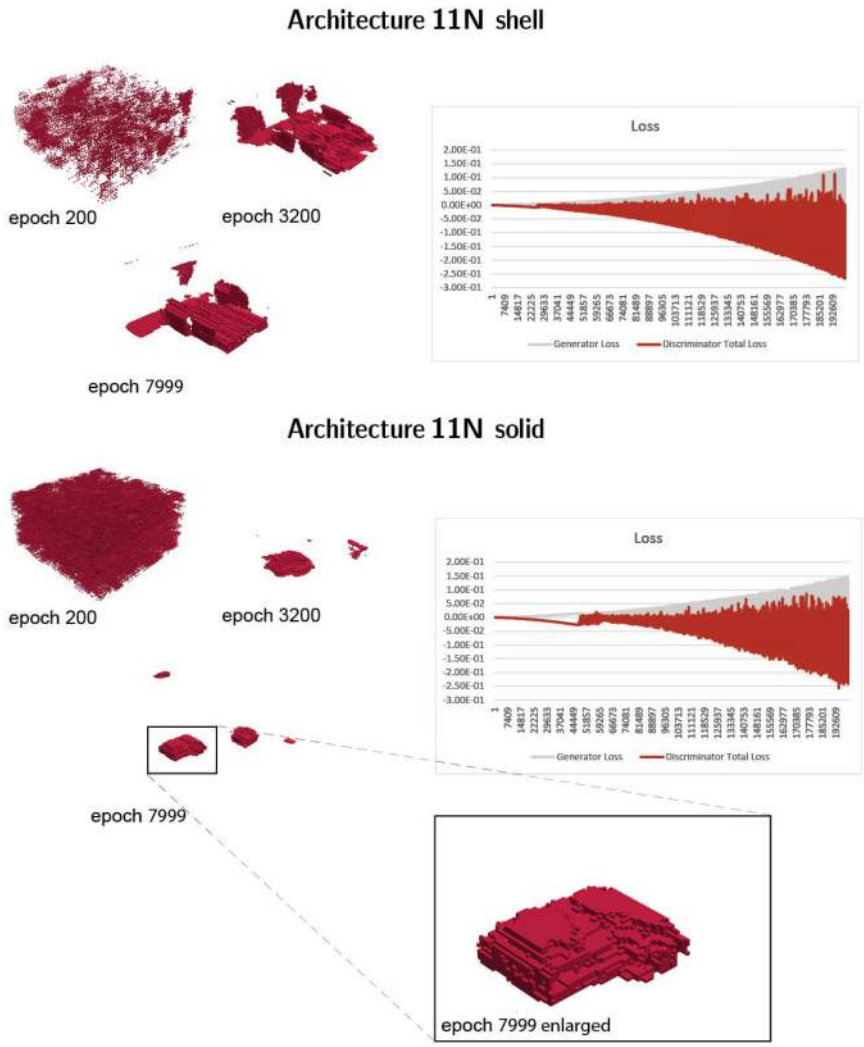


Figure 10.2.: Images showing the results of the **N** architecture trained with hollow shell 3D models on the top and the **N** architecture trained with filled solid 3D models on the bottom. The models were trained with 100 training data points.

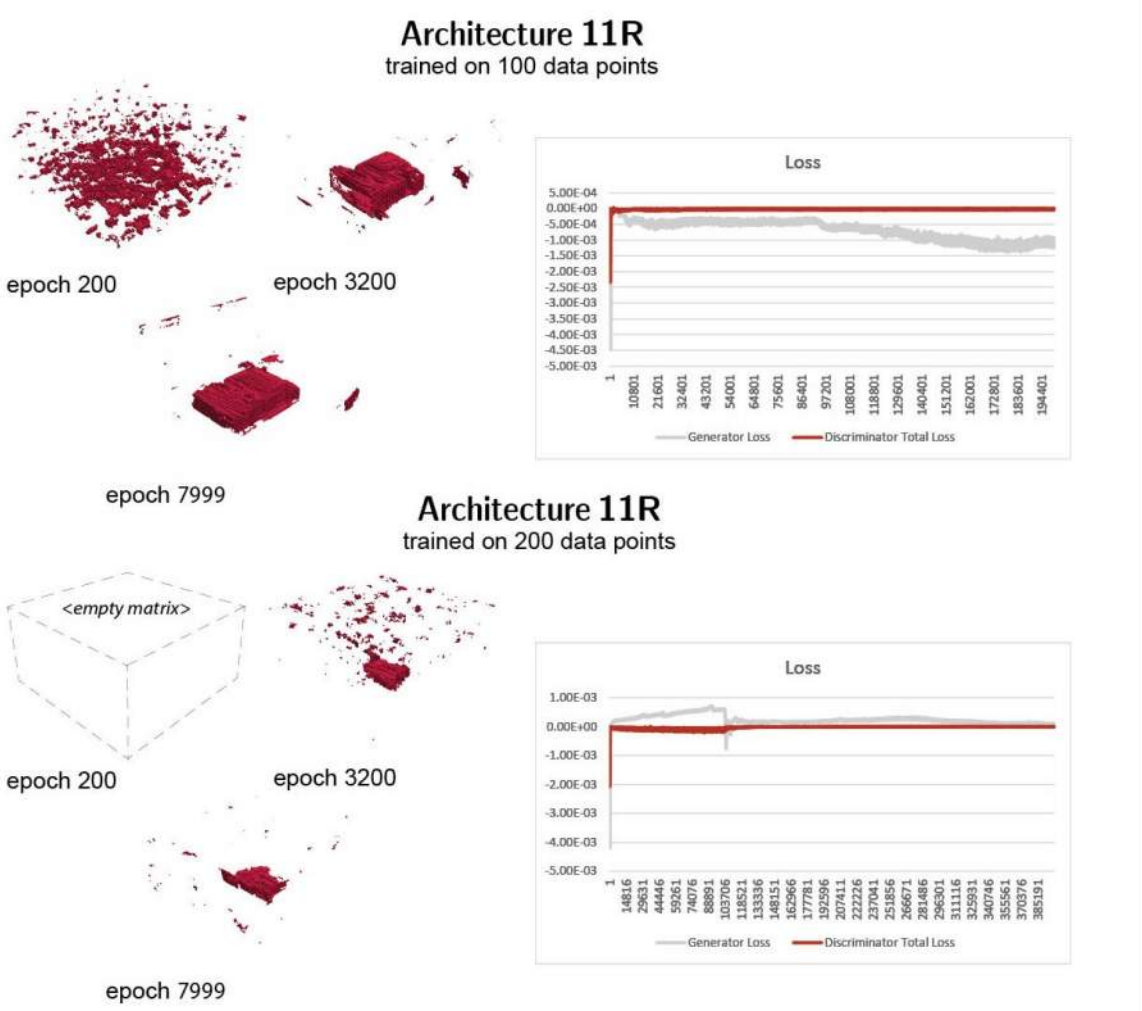


Figure 10.3.: Images showing the results of the **11R** architecture trained with 100 training data points above and 200 training data points below.

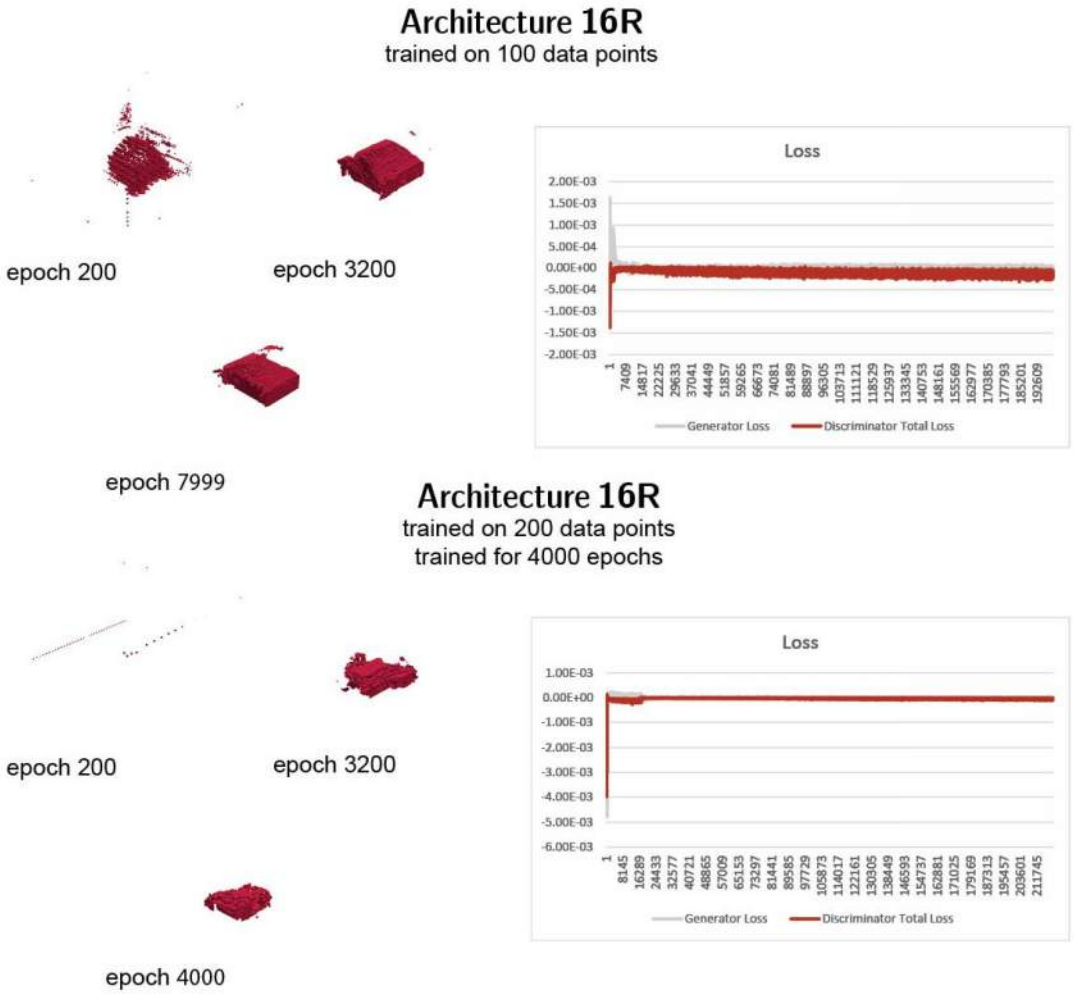


Figure 10.4.: Images showing the results of the **16R** architecture trained with 100 training data points above and 200 training data points below.

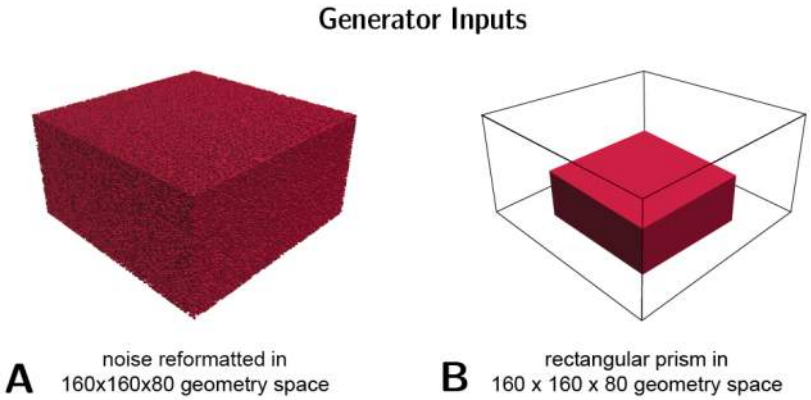


Figure 10.5.: (A) noise as the input to the generator (B) a rectangular prism as the input to the generator

ID	factors	generator input type	
		noise	cube
11R	R (Table 8.1) with 4 layers 48-24-12-2	[S][4/5]	[S][1]
15J	J (Table 8.1) with 5 layers 192-96-48-24-2	[S][4]	[U][1]
15R	R (Table 8.1) with 5 layers 192-96-48-24-2	[S][4]	[S][1]

Table 10.3.: List of experiments run with noise and a rectangular prism (labeled as cube) input to the generator. [U] indicates that the training was unstable [S] indicates that the training was stable. [1] indicates the output was noise [2] indicates that the output size, shape, and/or proportions are significantly different than the training data [3] indicates that the output is correct in size or proportion but the shape is not accurate [4] indicates that the output is noisy but roughly the correct size, shape, and proportions [5] indicates the output is roughly the correct size, shape, and proportions and has less noise [6] indicates the geometry is very similar to resolution of the training data.

10.2. Generator Input

NOISE VS RECTANGULAR PRISMS Typically, the input to the generator is noise as described in Section 3.6 and shown in Figure 10.5 (A). However, when the input to the generator is more similar to the final result, the generator needs to perform fewer modifications to the input. This results in the need for fewer layers in the network and more efficient training. One key part of the generator input is that it must still be random. Because of this, I wanted to test having generator inputs that are rectangular prisms as shown in Figure 10.5 (B). These prisms are generated from random variables for the length, width, and height of the prism within an acceptable range. After the rectangular prisms are generated, a process similar to label smoothing is used to randomly distribute the 0 values between 0.0 and 0.3 and all the 1 values between 0.7 and 1.2. This is important because there is not enough randomness if only 0 and 1 labels are used. Table 10.3 shows the results of these experiments. All architectures only generate noise when using rectangular prisms (referred to as cubes in the table and some images) as the input. An example with two of the outputs is shown in Figure 10.6.

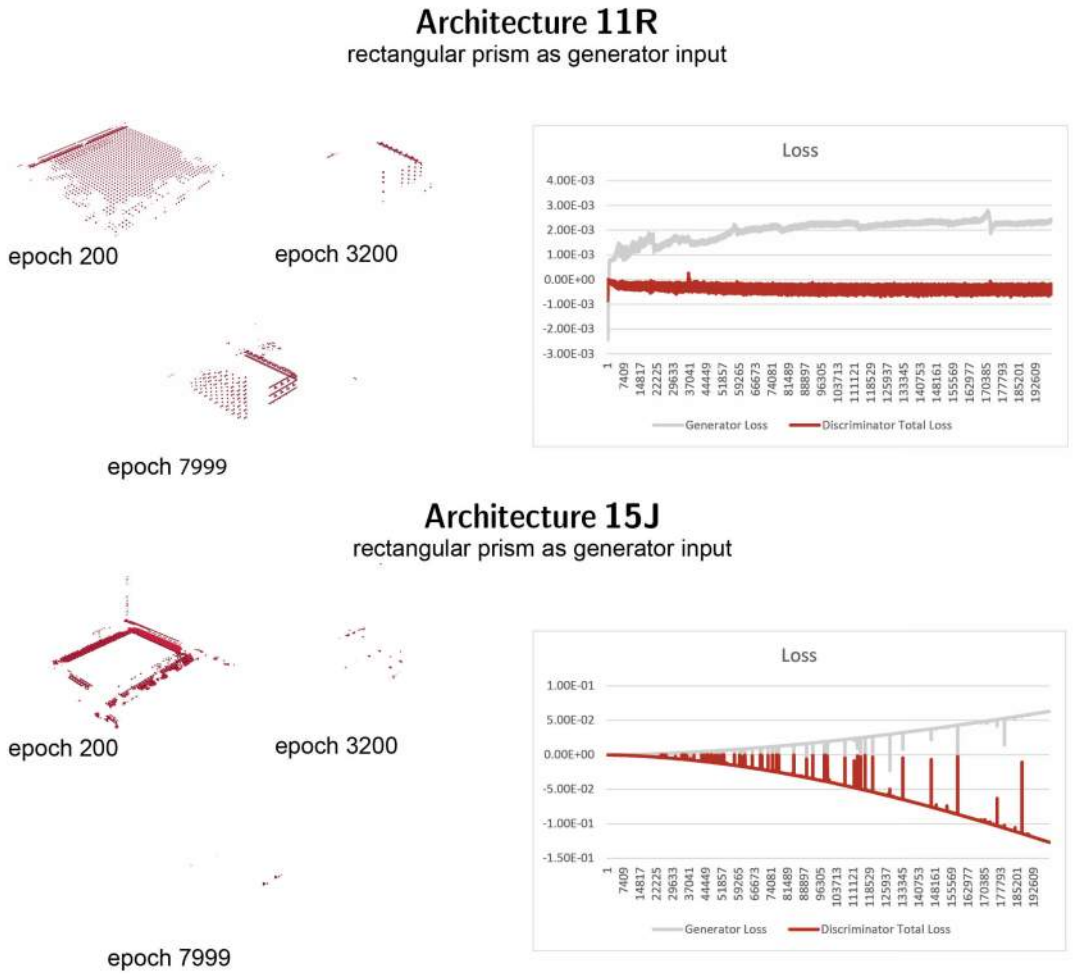


Figure 10.6.: Images showing the results from **11R** and **15J** architectures when using a rectangular prism as the input to the generator. Each model is trained with 100 training data points.

10.3. Experiment Analysis and Conclusions

In addition to exploring different hyperparameters, it was important to test different inputs for the generator and the critic and to analyze the training process. The results are indicated in Table 10.1, Table 10.2, and Table 10.3.

MULTI-LABEL INPUTS After testing and research it was determined that GANs do not work with multi-label inputs. There are various methods to incorporate multiple labels with generated geometry. Due to time restrictions, it was not possible to further test during this thesis, but it remains an opportunity for future research.

SHELL 3D MODELS AND SOLID 3D MODELS After using shell and solid 3D models for the training data set, it became clear that using solid 3D models requires its own hyperparameter tuning. For architectures that performed well, there was no benefit to using solid 3D models.

LARGER TRAINING DATA SET All architectures produced less accurate geometry with the larger data set. This is expected as the architectures need to learn more features. The depth and width of the network seem to affect the performance of DL models when trained on larger data sets. Architecture **16R**, for example, performed better than architecture **11R** on the larger data set. The generated geometry was still not as good as the geometry generated by the DL models when trained on 100 models; however, this aligns with expectations. A larger data set is more complex and has more features that a network must learn. However, it is still important to train on large data sets, as this results in a more robust model. The results of these experiments suggest that larger data sets require deeper and wider networks to maintain a similar level of performance. Large data sets can also help prevent architectures from memorizing the training data set.

NOISE VS RECTANGULAR PRISMS After testing a rectangular prism input on architecture **11R**, **15J**, and **15R**, it became clear that having a cube as an input did not help improve training. All the architectures continued to generate noise throughout the training process. There are additional tests needed to determine whether changing the generator input in other ways can have benefits to training.

Testing different inputs did not have an impact on the best-performing network architecture, but did show that it is necessary to train on significantly larger data sets.

11. Analysis of Final Results

After determining the best architectures, it was necessary to perform some additional testing with these three architectures. The three best performing networks are architecture **16R**, **17R** with unequal padding, and **17R** with equal padding. All **R** architectures use Leaky ReLU in the generator and critic and use the RMSProp optimizer with gradient penalty and a learning rate of 0.00005 as described in Table 8.1. Architecture **16** has 10 layers with {96-96-48-48-24-24-12-12-2-2} channels and architecture **17** has 10 layers with {192-192-96-96-48-48-24-24-2-2} channels as shown in Table 9.1. To test these architectures, it was necessary first to determine the optimal number of epochs for which each should train, and then it was possible to use the trained models to generate 100 3D building models each for evaluation.

11.1. Training Length

When training GANs, it is not possible to use the loss function to identify how many epochs a network needs to train for to learn features. Instead, it is possible to visualize the image of the generator output after each epoch. These outputs can then be compared as shown in Figure 11.1 and Figure 11.2. At a certain point in the training, the geometry is fairly clear and there are no major changes to the output anymore. At this point, the network performs fairly well. For architecture **16R**, this was after 5600 epochs, and for architecture **17R**, this was after 2600 epochs. Part of the reason that the architectures take a different number of epochs to train is that architecture **17R** is updated more times each epoch, so therefore it takes fewer total epochs to train. Based on this, architecture **16R** should be trained for 5600 epochs and architecture **17R** should be trained for 2600 epochs.

11.2. Test Results

Based on the optimal training duration, described in Section 11.1, **16R** and **17R** were trained for 5600 epochs and 2600 epochs, respectively. After training, the networks were used to generate 100 building geometry forms so that all 100 outputs could be compared. There are multiple reasons for evaluating and comparing multiple outputs of the same model. First, it is important to evaluate the generated building geometry to see if the networks generate many of the same building models. In addition, it is important to check that the networks memorize the training data. Due to time constraints, these comparisons were done manually. Figure 11.3 shows visualizations of selected outputs from the training. The 100 generated images and additional larger images can be found in Appendix B.9.

Some of the outputs were very successful, while others were not as successful. By generating 100 models with these three architectures, it was possible to confirm that the networks produce different results for each 100 iterations. There are some models that have similarities, but they are not exact copies.

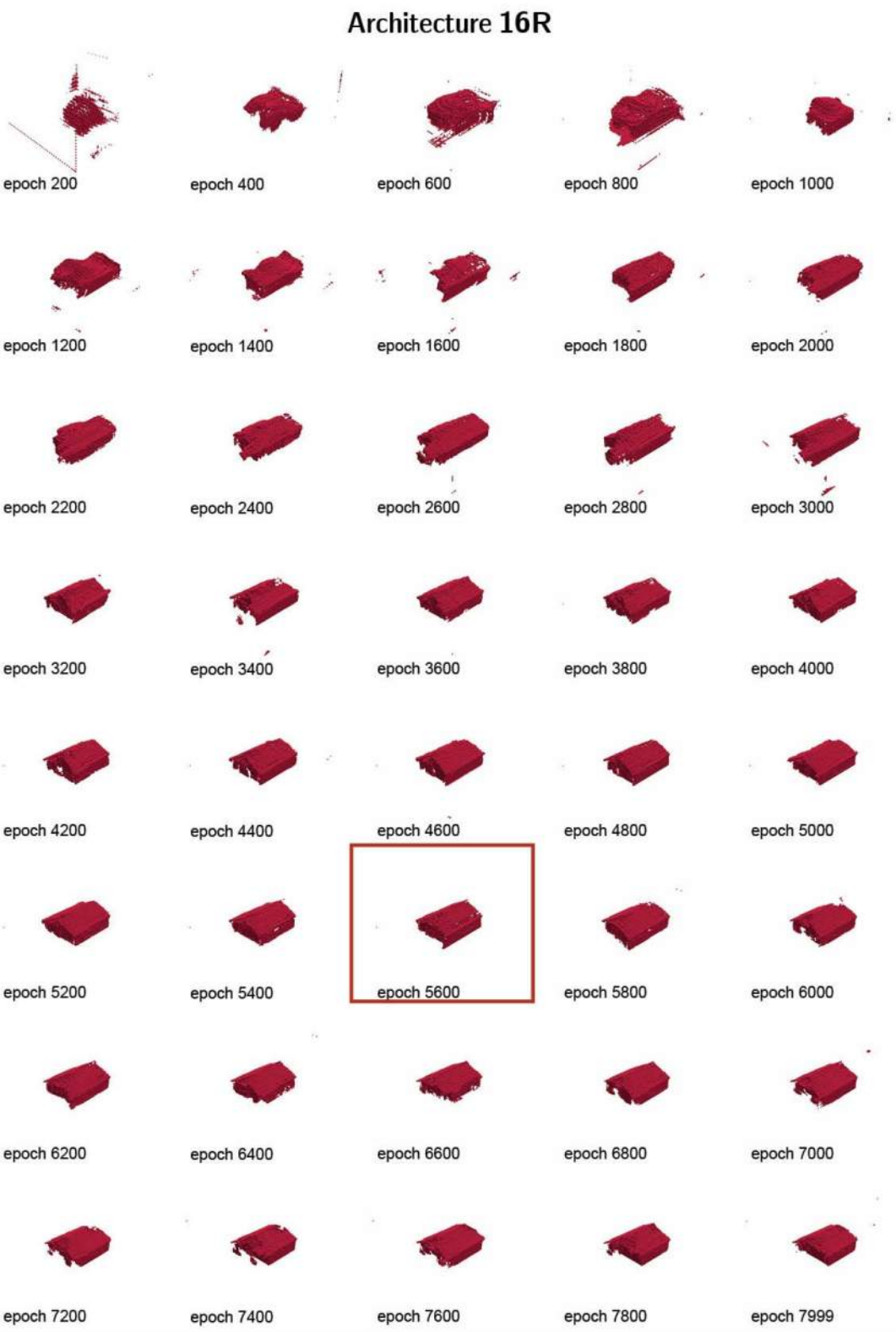


Figure 11.1.: Images generated every 200 epochs during the training of architecture **16R**. The earliest clear output is identified with a red box.

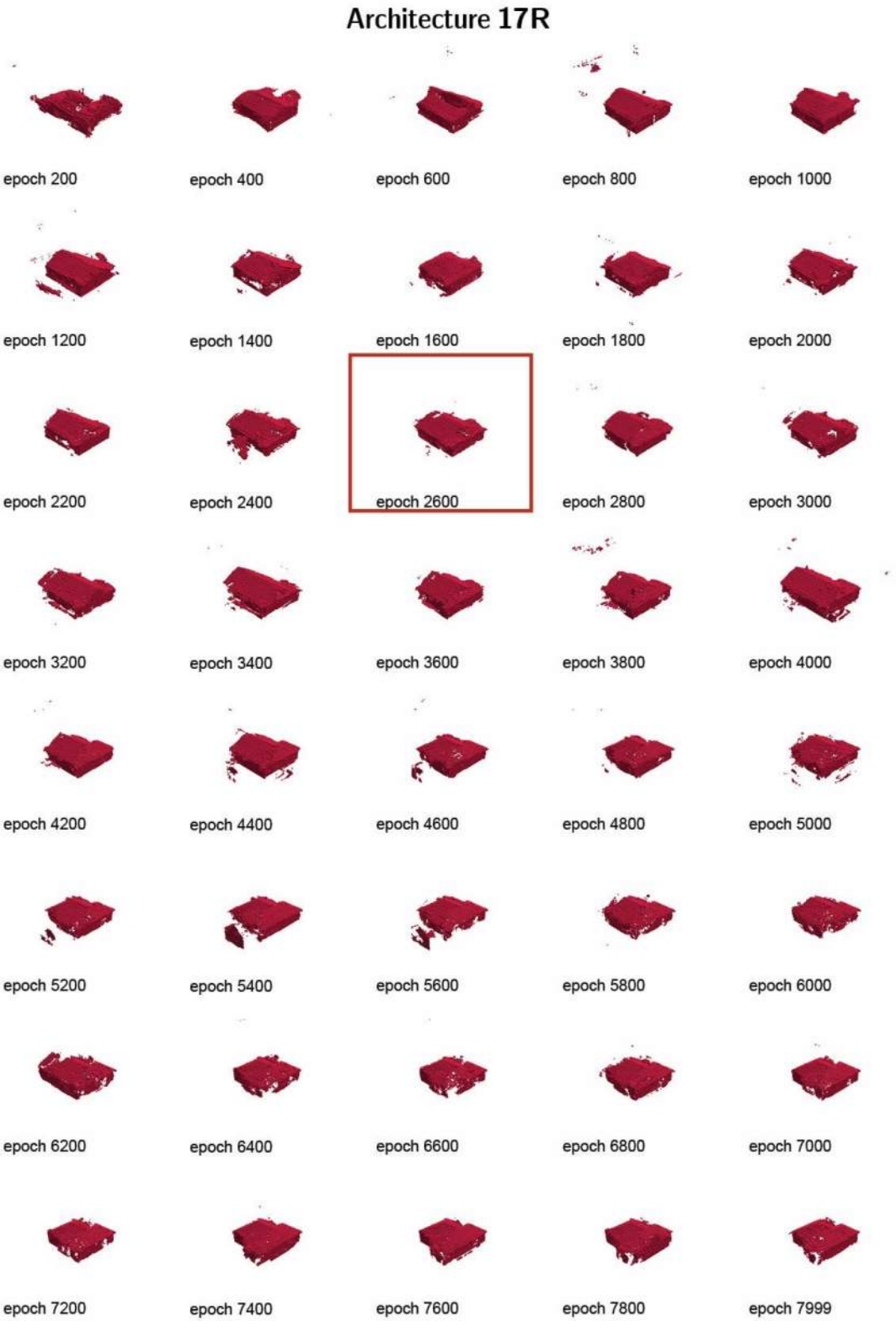


Figure 11.2.: Images generated every 200 epochs during the training of architecture **17R**. The earliest clear output is identified with a red box.

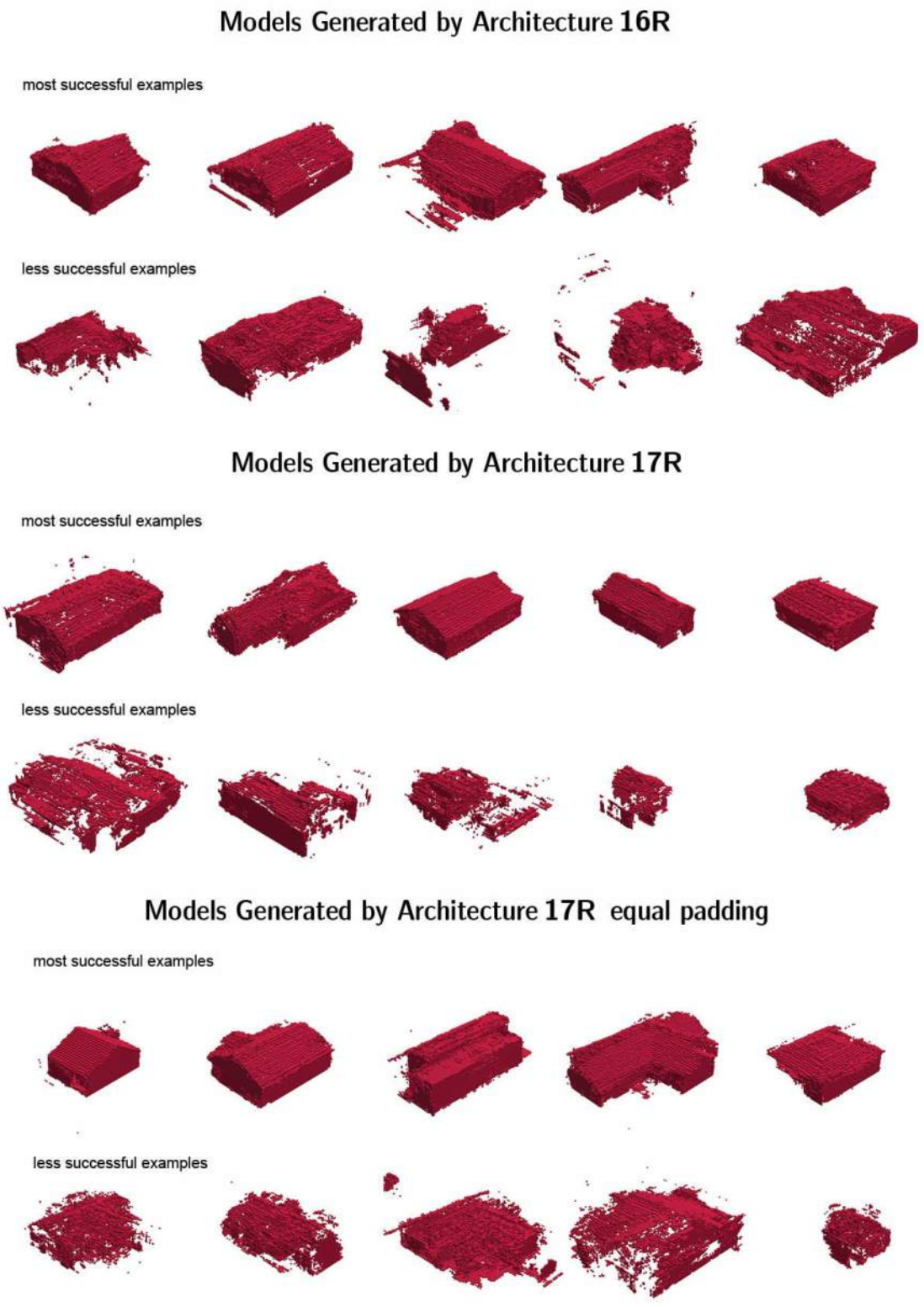


Figure 11.3.: Select examples of the 100 models generated by architectures **16R**, **17R**, and **17R** equal padding.

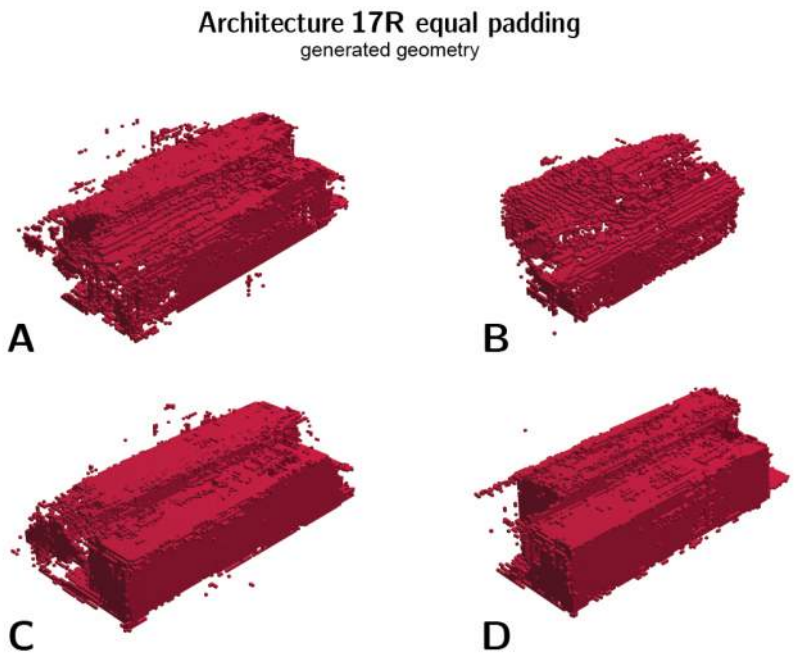


Figure 11.4.: A selection of models generated by architecture **17R** with equal padding which all have similar massing but different building features.

The output was also evaluated with respect to memorizing the training data set. All three architectures picked up training data set features such as the massing of different training data points, but it can be challenging to differentiate between learned and memorized features. **16R** shows very little sign of memorizing data, while **17R** with unequal and equal padding both show some signs of memorization. This was discussed in more detail in Section 9.2 Subsection Memorizing Data. However, the two variations of the **17R** architecture also show signs of learning from the data set, which is also described in detail in the same section. The fact that the architectures appear to be learning building features is demonstrated in Figure 11.4. Of the 100 different outputs that architecture **17R** with equal padding generated, four models have a similar massing. However, other building features are different between iterations. Figure 11.4 (B) for example shows what could be a sloped roof and Figure 11.4 (D) shows a flat roof with detail around the edge that resembles a parapet. This suggests that the DL model did learn feature of the data set and is not simply repeating the exact geometry as the training models.

In reviewing all outputs, the generated models are varied and also have features that suggest that the DL models are learning characteristics found in the training data set. This is shown in figure 11.5 shows select generated models on the left and select models from the training data set for comparison on the right. The variety in the generated models and the fairly high resolution show promise in the tested architectures. However, there is still room for improvement. Currently, some of the generated geometry contains significant amounts of noise, and the unclear models do not look very similar to a building. Further research is required to improve the architectures to a point where they consistently output high-resolution results.

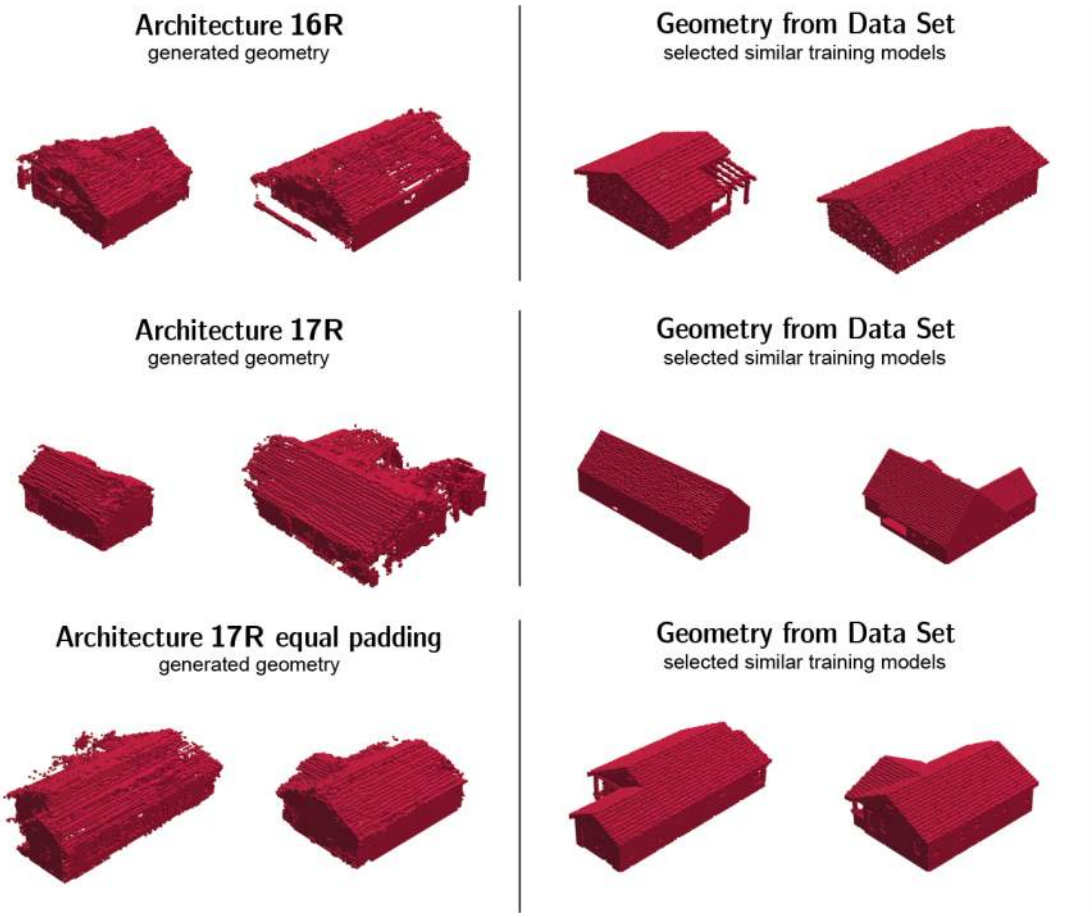


Figure 11.5.: Images showing generated models from the **16R**, **17R**, and **17R** with equal padding architectures compared to select models from the training data set.

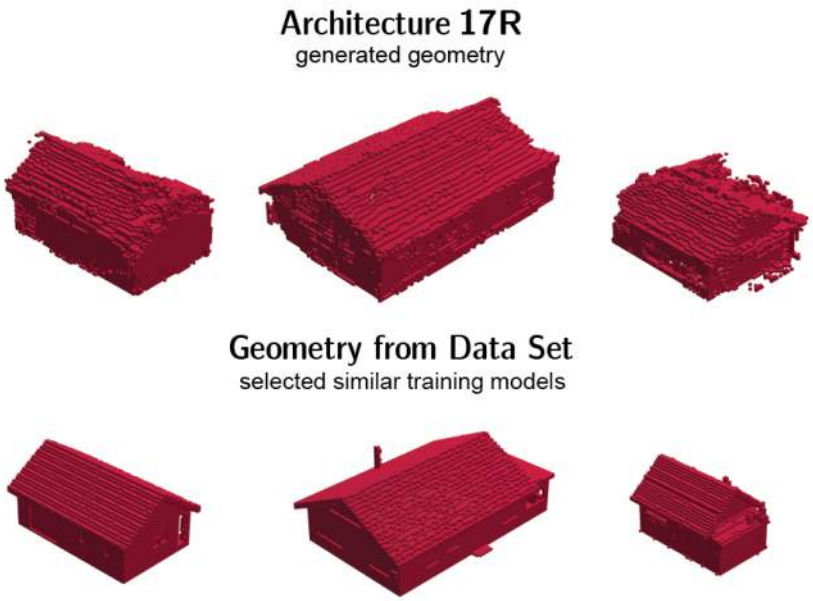


Figure 11.6.: Images showing some of the best generated results of the **17R** architecture. The geometry below is samples from the data set that have similar features. The model was trained with 100 training data points.

11.3. Improved 3D Wasserstein GAN Architecture

When evaluating between the best performing architectures which include **16R**, **17R**, and **17R** with equal padding, the most promising architecture is **17R**. When reviewing **17R** with unequal padding and kernel size 4x4x4 and **17R** with equal padding and kernel 5x5x5, **17R** with equal padding generates a clearer geometry. However, this architecture also showed some tendencies to memorize models from the training data during training, as shown in Figure 9.8. When comparing many outputs of architecture **17R** with equal padding, there are signs of learned features. Because architecture **17R** with equal padding is more likely to memorize features than architecture **17R**, architecture **17R** was selected as the best performing architecture.

Therefore, this thesis introduces a new GAN architecture developed through exploratory research and called Improved 3D Wasserstein Generative Adversarial Network (I3D WGAN) based on architecture **17R**. Figure 11.6 shows some of the best results produced by this architecture. The network learned to create a model with fairly straight walls and a sloped roof as well as details like recesses and changes in plane. Additionally, the resolution of the generated geometry remains fairly high due to the large geometry space.

The architecture of the improved I3D WGAN is shown in Figure 11.7 and is described in the following subsections.

LOSS FUNCTION Wasserstein Loss.

GENERATOR The generator consists of ten fully convolution layers with numbers of channels {192, 192, 96, 96, 48, 48, 24, 24, 2, 2}, kernel sizes {4, 4, 4, 4, 4, 4, 4, 4, 4, 4}, and strides {2, 2, 2, 2, 2, 2, 2, 2, 2, 2}. The kernel and stride sizes are the same in all three dimensions. The activation function used is Leaky ReLU of parameter 0.2 and there are no batch normalization layers. The Sigmoid activation function and a flatten layer are used at the end. The input is a 200-dimensional vector and the output is a $160 \times 160 \times 80$ matrix with values in [0, 1]. The generator uses RMSProp with a Learning Rate=0.00005.

CRITIC As a mirrored version of the generator, the critic takes as input a $160 \times 160 \times 80$ matrix, and outputs a scalar between $-\infty$ and ∞. The critic consists of ten volumetric convolution layers, with numbers of channels {2, 2, 24, 24, 48, 48, 96, 96, 192, 192}, kernel sizes {4, 4, 4, 4, 4, 4, 4, 4, 4, 4}, and strides {2, 2, 2, 2, 2, 2, 2, 2, 2, 2}. The kernel and stride sizes are the same in all three dimensions. There are Leaky ReLU layers of parameter 0.2 without batch normalization layers and no activation function in the final layer. At the end, there is a flatten layer followed by a dense layer with one channel. The generator uses RMSProp with a Learning Rate=0.00005.

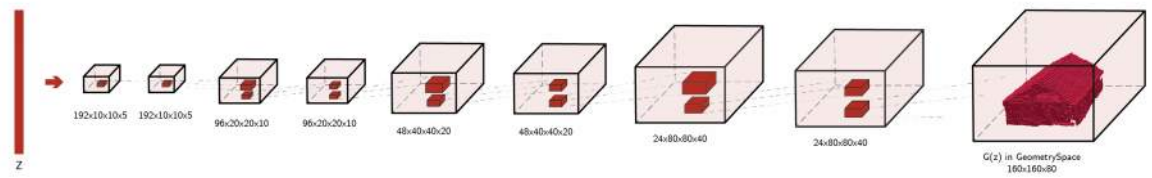


Figure 11.7.: Images showing the **17R** architecture.

11.4. Source Code

The code for architecture **17R** and a selection of the rest of the experiments run for this thesis are located on the public GitHub repository. The public GitHub repository for this project can be found at `https://github.com/lm2-me/3DWGANHouses`.

The two run files in the main folder can be used to train two of the best performing architectures **16R** and **17R** with a small training data set. It is necessary to install all the dependencies required to run the code. There is a Conda environment file included on GitHub that can be used to set up a new environment.

There are also three different Jupyter notebook files located in the public GitHub repository for this thesis. These notebooks demonstrate some specific parts of the project and make them easier to use and understand. Please note that to use a Jupyter notebook, you need to have the appropriate dependencies installed. There is a Conda environment file included on GitHub that can be used to set up a new environment with all the dependencies.

PRE-PROCESSING NOTEBOOK The Jupyter notebook on pre-processing demonstrates the steps that were implemented to clean the point-cloud models so that they can be used for training GANs.

There is a Conda environment file included on GitHub that can be used to set up a new environment with all the dependencies including Open3D which will be needed to visualize the models. Appendix B.10 shows images of the notebook.

GENERATE NOTEBOOKS The Jupyter notebooks on generating geometry allow users to generate new building forms with the trained models that use architecture **16R** and **17R**. There is a Conda environment file included on GitHub that can be used to set up a new environment with all the dependencies including TensorFlow which will be needed to run the code. These notebooks allow those interested to easily generate some geometry using two different trained models. Appendix B.10 shows images of the **16R** notebook.

12. Conclusions

12.1. Research Questions

Through the research completed for this thesis, the following conclusions have been made.

How can a 3D GAN model be trained to produce 3D building geometry given models of single-family homes as an input?

Through the development of a new GAN architecture, I was able to generate a building geometry that incorporated structural features that were present in the data set. This includes features such as flat walls and sloped roofs. Additionally, the models also learned more detailed features, such as recesses in the walls and chimneys. The type of research conducted for this thesis was exploratory research that required researching, coding, training, testing, and documenting the results of many different GAN models. The process started by evaluating existing state-of-the-art models and attempting to generate building geometry with these models. When trained on a data set that contained 3D building models, the state-of-the-art architectures generated only noise. Then, on the basis of the problems that resulted after these evaluations, new hyperparameter options and combinations were found. Key hyperparameters were identified by running experiments and comparing the different results. The best performing architectures were then further developed by exploring the network depth and width, and input options. The best architectures are now able to generate building geometry with large-scale features, such as wall and roof features, and small-scale features, such as recesses in the wall and extrusions similar to dormer windows and chimneys.

What are key aspects of a training data set that are required for generating 3D building designs?

Based on the experiments conducted, it is important that the training data set is clean. This means that labels should be accurate and consistent throughout the data set. The geometry should also be clean, so the models should contain only relevant geometry. For example, when training to produce building geometry, site geometry such as roads and trees should be excluded. The data also needs to be at a consistent scale to each other. For example, the existing building model data set had model sizes normalized so that the largest dimension was one unit. This meant that the 30-story tall buildings and the one-story tall building were approximately one unit tall. The data set was used for labeling data, and, therefore, the scale was not an important factor. However, when creating new building geometry, building elements, such as doors, always have a size that is within a specific range and in a specific proportion to the rest of the building. Therefore, having a consistent scale is more important. Finally, the data set must be large enough for training. At a minimum, 100 models should be used for training, but more data is better.

What are the key network architecture hyperparameters that lead to a stable GAN training when generating 3D geometry?

The experiments tested a number of different hyperparameters, some of which were successful and others were not. I was able to identify several hyperparameters through experiments that led to a successful training. The most critical is to use Wasserstein loss because it has a large impact on training stability. Additionally, the best performing architecture uses gradient penalty, Leaky ReLU in both the generator and critic, and RMSProp as the optimizer. These hyperparameter settings lead to a model that generated geometry that incorporated both large-scale structural features from the data set, such as flat walls and sloped roofs, and small-scale details, such as recessed in the walls. Based on this research, I developed I3D WGAN which incorporates these findings.

The experiments also showed that batch normalization did not improve training. Additionally, when using ADAM as the optimizer, it should be used in conjunction with learning rate decay. RMSProp, on the other hand, performs best with a fixed learning rate. When comparing different optimizers for the application of generating building geometry, RMSProp with a fixed learning rate outperformed ADAM with learning rate decay.

How does network depth, width, and kernel size impact GAN training when generating 3D geometry?

Adjusting the kernel size did not benefit training, but changes in network depth and width had an impact. The width of the architecture refers to the number of channels, which is the number of features that the model will learn in each layer. The depth of the architecture refers to the number of layers that the architecture has. For some architectures, increasing the number of channels, thus increasing the width, had a positive impact on training. There was a similar result for increasing the number of layers, thus increasing the depth, of the architecture. This was especially true for architectures that generated geometry that was similar in size, shape, and proportion to the training data but still had noise around the geometry space. These well-performing architectures benefited from more depth and width. On the other hand, scaling width and depth for architectures generating low-quality outputs yielded inconsistent results. The best performing architecture, architecture **R**, uses leaky ReLU in the generator and discriminator, uses the RMSProp optimizer, and has gradient penalty implemented. For example, this architecture generated geometry with significantly less noise when the depth and width were increased.

Based on these results, I conclude that adjusting the depth and width of the network can help to reduce the amount of noise in architectures that are performing well. For the architectures tested during this thesis, architectures that performed well with shallower and narrower structures continued to perform well with wider and deeper structures. Some architectures that initially performed poorly performed better with wider and deeper networks. However, architectures that consistently performed well still outperformed architectures that improved with wider and deeper structures. Therefore, these experiment results show that if an architecture performs well with a shallow and narrow structure, it will likely perform better overall. I recommend that a practitioner start with a small network in terms of width and depth while tuning hyperparameters to accelerate training for rapid prototyping and to reduce data memorization. Then, it is possible to scale up only when the shallower and narrower models show clearly positive results.

How do the network inputs impact GAN training when generating 3D geometry?

The impact of different inputs was determined by selecting different input options for the generator and the critic and then testing the outcome. I tested hollow shell versus solid training models, the size of the training data set, and rectangular prisms as inputs for the generator.

The most important input parameter to consider is the size of the data set. I was able to test 100 versus 200 training models, but due to time constraints and limited data availability, I was not able to test training on large data sets. When training on data sets containing 100 and 200 models, the resolution of the output decreased when there were more 3D models in the training data set. This is in line with expectations. A larger data set is more complex and has more features that a network must learn. This suggests that larger data sets require deeper and wider networks to maintain a similar level of performance. The architectures that performed best with the smaller training data set also performed best with the larger data set. This shows that if you can train well-performing models on small data sets, it is possible to scale the training data set and still have a model that performs well, although some slight adjustments may be needed.

It is also important to remember that other industries train on data sets consisting of thousands or even millions of data points. ImageNet, a well-known data set containing different pictures used for machine learning, contains more than 1.2 million images. The AEC industry is lacking in data availability (too little data is available to the public), completeness (data lacks information like model scales and labels), and relevance (there are not enough models of the relevant building typologies).

In addition to organically obtaining more data, data set augmentation can be used as a strategy to increase the size of the data set without the need to generate more building models. Data set augmentation uses modified copies of existing data to artificially increase the size of the training data set. For example, copying and rotating each model around a central axis by a set number of degrees. This is a valuable strategy in the interim, as larger data sets are better for training. As more data becomes available in the industry or more data is generated through augmentation, it is better to train DL models on larger data sets. Large data sets benefit the training because they can help prevent the models from memorizing the training data set.

Several other options were also tested when researching different inputs. Training of well-performing models did not improve when the training data set was solid models. Additionally, when rectangular prisms are used as inputs for the generator, the network only outputs noise. Therefore, these two inputs do not benefit the training.

What are the challenges and benefits of using GAN for architectural design?

There are both challenges and benefits to using GANs for architectural design. Through this research, I have identified the following challenges:

- Access to computational resources. To generate 3D geometry in a large geometry space, training needs to take place on a GPU with enough memory. A standard PC will run out of memory when training the models, and training will be too slow to be tractable.
- Having a large enough data set to properly train the model. A minimum data set of 100 models is needed, but more data of good quality is generally better.
- It is currently not possible to know how the trained model makes the connection between the training data and the new generated geometry.
- Small-scale features, such as roof overhangs and complex setbacks, remain difficult for the generator and will likely require additional modifications.
- Models that generate high-resolution outputs also have a tendency to memorize select data set elements, which is a problem when the goal is to generate novel geometry. It can also have legal implications such as copyright infringement.

And the following benefits:

- Once a model is trained, it can be used repeatedly with little to no cost.
- The architecture developed through this thesis can also be tested on other types of problem, generalizing the results beyond single-family houses.
- The architecture determined through this research provides a stable model that can generate 3D geometry in a large geometry space.
- The model can always be fine-tuned or re-trained when more and/or different data becomes available.
- GAN can solve creative tasks and provide one step in the larger picture of automating design to bring good design to the masses.

12.2. Limitations

The research carried out was limited in scope due to several factors.

DATA SET Although an ideal training data set would consist of multifamily buildings, existing training data did not meet this requirement. This is why single-family homes were used instead. In the future, the models can be retrained using the correct building typology.

Additionally, the data set was limited in size due to available data and time constraints. An opportunity for future research includes data set augmentation and data mining. In the future, data mining could be done in the real world by having drones scan and generate 3D building models from these scans. Technology may even allow models to easily be generated from the photos people take on their phones.

TIME CONSTRAINTS Due to time constraints, research on multiple class labels in the generated geometry and incorporating site-responsive design was not completed. There are also a few hyperparameters that could not be researched due to time limitations. These include different learning rates using RMSProp, larger and differently proportioned geometry spaces, and more research about padding. In general, exploring 3D GAN methods is a time-consuming venture, which limits the number of iterations of the architecture and other hyperparameters.

Some key design aspects that were not yet incorporated into the research should also be considered. These including site-responsive outputs and code requirements. Architecture is not generated in isolation, and therefore considering the surrounding site and context of each building through site-responsive design is important to generate successful designs. Additionally, buildings must consider all code requirements and be designed to ensure the health and safety of the occupants. These factors will require human intervention or other generative AI to fill the gaps.

12.3. The Bigger Picture

Ultimately, this research has shown that scaling up the resolution of GANs is possible and there are opportunities to implement deep learning in generative design. Although it is unlikely to find this technology implemented into major BIM software within the next few years, the research has clearly shown that GANs are promising in the use of generative design. In the future, it will be possible to link different research areas to design a complete building. Figure 12.2, for example, shows how existing AI image generators use the geometry created by the GANs I trained and visualize it as photorealistic houses. To help the AI image generators distinguish the planes of the image, each face of the building

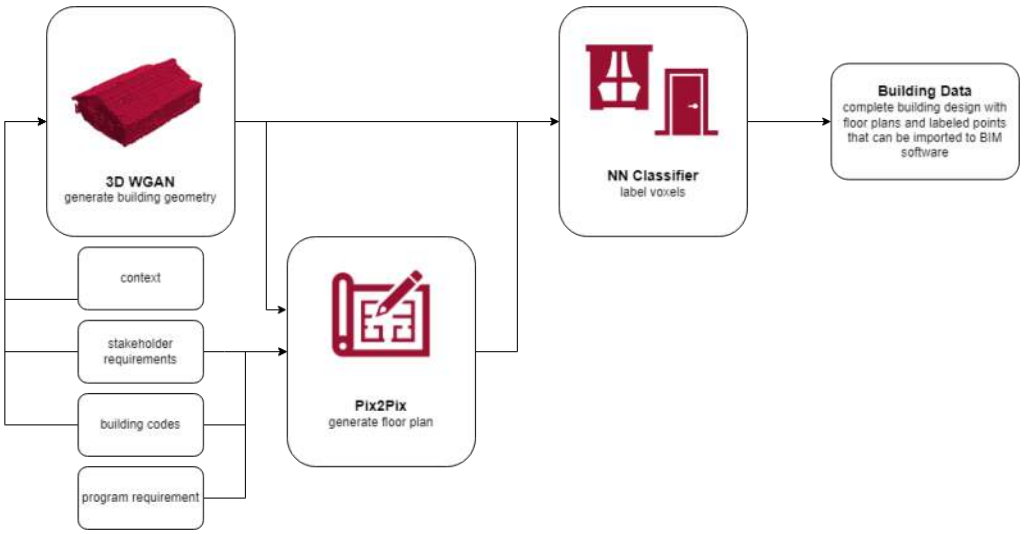


Figure 12.1.: A diagram of the possible future workflow to generate building designs with deep learning processes.

was recolored to a different color. There is also other DL research that can contribute to the goal of automating building design. This includes methods to label 3D models [Selvaraju et al., 2021] and methods to generate floor plans [Rodrigues and Duarte, 2022]. A new workflow would combine all three generative, deep learning processes to produce a building that can be imported into BIM software. Figure 12.1 shows an example of what this workflow could look like. First, the context (surrounding site information), stakeholder requirements, and building codes are analyzed and a 3D building form is developed. The building form, stakeholder requirements, building codes, and program requirements are then used to create an interior floor plan. The form and floor plan are then used to auto-annotate the voxels of the 3D building geometry. This results in a building model that can be imported into 3D modeling software for design documentation. This is one way in which different deep learning methods can come together to deliver a design project in the future.

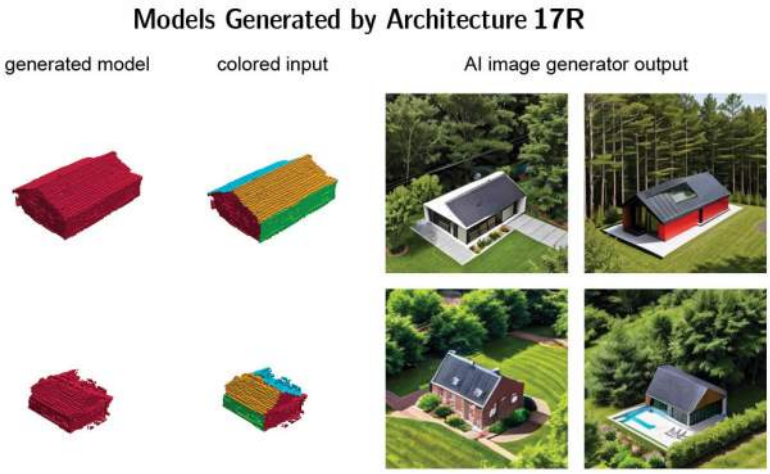


Figure 12.2.: The generated geometry, how it was recolored to clarify the different planes for AI image generators, and the output from those generators imaging the geometry as photo-realistic houses.

12.4. Further Research

There are several areas that present opportunities for further research. Some topics are direct next steps following this thesis research, and these include:

- Implement methods like memorization rejection Bai et al. [2022] to discourage architectures that generate geometry with a high level of detail from memorizing training data.
- Explore the relationship between the kernel size and geometry space size and how this relates to the generated geometry with regard to level of detail and memorization.
- Review additional hyperparameters to see their impact and to see if they would improve the output of the generator. These include testing different learning rates using RMSProp and testing larger and differently proportioned geometry spaces.
- If it is not feasible to add new models to the data set, expand the data set through data augmentation. This would involve creating a new data set that includes only the best building models, but placing each building model in the geometry space at multiple different rotations. This could ensure that GANs do not learn to generate buildings only parallel to the geometry space boundaries.
- Incorporate site information to generate site-responsive geometry. Architecture is never generated in a vacuum, so an important next step is to incorporate context information into the training data so that the algorithm can learn to generate site-responsive building geometry.

In addition, there are opportunities to expand on the research and support the bigger picture of automating building design. These opportunities include:

- Generate a multifamily home data set and retrain the GAN models. As discussed, an optimal building typology would be multifamily housing, but these models are not available in large enough quantities to train a model.
- Train a neural network to label generated geometry with multiple class labels. GANs can only generate a single label, so they cannot generate geometry that distinguishes between voxels representing windows, doors, and walls. It would be useful to us a ANN to label each voxel with the building feature it represents.
- Use other generative methods such as diffusion or flow models to generate building geometry to see how they compare to using GANs. All methods can also explore using different modalities such as graphs, 3D models, and point clouds.
- Integrate simulations to automatically analyze the structural integrity of the generated geometry and use this to guide the search for viable designs. Differential simulators could be used for structural analysis and feedback to the generative models.
- Incorporate additional inputs that allow users to modify the output each time it is generated. For example, how DALL-E2 allows text prompts to adjust the generated images.

13. Reflection

The goal of this thesis was to explore one step in the larger challenge of automating design by researching the opportunities of applying deep learning models to the design of buildings. The resulting DL model came from researching and testing various hyperparameters of 3D GANs to find an architecture that generates building geometry. Through this process, I developed my own improved 3D WGAN architecture to stabilize training and provide more consistent results. The final goal was for the trained model to generate building geometry that has features similar to those in the data set.

13.1. Graduation Process

How is your graduation topic positioned in the Master of Science in Architecture, Urbanism and Building Sciences program and the Building Technology master track?

The Master of Science in Architecture, Urbanism and Building Sciences focuses on innovation within the fields of architecture and engineering. This relates directly to my thesis topic, as my thesis is looking at innovative approaches to the design process by applying deep learning to an architectural problem. The building technology track focuses on integrating design and technical disciplines. This means the program has a multidisciplinary focus that encourages students to explore topics that connect different fields. I am following this multidisciplinary process by connecting expertise from the design, engineering, and informatics fields through my thesis. My topic requires knowledge that is very specific to the built environment and also requires the use of research from the field of informatics. Combining these two different specialties allows me to explore a topic rooted in design and engineering with a new lens.

What value did your approach and methodology bring to your thesis?

For my methodology, I focused on a review of the literature and exploratory experimentation and testing. Both aspects were valuable and allowed me to maintain a methodical approach while applying an innovative process to an established problem. The literature review allowed me to understand the solutions from existing research and also learn how to identify common problems that may need to be addressed. Initially, I had expected that the 3D GAN architecture on which I based my research would need minimal changes to work with my problem due to similarities between the two topics. However, I soon realized that I would need to make significant changes to this structure to stabilize training with the new data set. Here, the use of an ablation study helped. Through the ablation study, I tested hyperparameters individually and in combination to identify which changes helped stabilize training. This systematic approach allowed me to pin-point key aspects of stable training. These hyperparameters can be used for the problem of generating building geometry but can also be used in future research. This systematic approach also means that the steps I took are well documented so future researchers can learn from the process, not just have a new architecture to use. The report also explains why specific hyperparameters are used so future research can expand upon the process and learn from it.

How did the approach influence your result and what were the strengths, weaknesses, opportunities, and threats of your method?

The method and the research has a number of strengths and weaknesses, also summarized in Figure 13.1. One weakness is that a process of trial and error is required to tune and train models. Through the literature review, I realized that researchers have different opinions on how and why certain hyperparameters work. Therefore, when developing the architecture of improved 3D WGANs, I needed to try different hyperparameters and set up new experiments based on the previous results. Another challenge of using DL models is that the model discovers the characteristics of the data set during training. This means that there is currently no way to fully understand the connection between the training data and the result. Finally, training requires a large data set filled with fairly accurate data, which can be challenging to obtain. There are also a number of strengths to the method. Although an optimal data set was not available during the writing of the thesis, the resulting model can be re-trained once this data becomes available. Because research was done for this specific model, the final data set used for training can vary and can be added to over time. Additionally, this method can be applied to other generative design problems. Future researchers will only need to train on different data sets and tune the hyper-parameters to be able to apply the model to different problems. Finally, through the architecture I developed, I was able to generate buildings in a large geometry space. My thesis is the first research that significantly increases the geometry space (from 64 x 64 x 64 to 160 x 160 x 80). By increasing the geometry space by 2.5 times in each direction, I hope to demonstrate how their application can be useful for building scale.

When considering external factors, there are some opportunities and threats to consider, which are also summarized in Figure 13.1. There are a few threats that we need to be aware of. Data sets related to the AEC industry are scarce and the limited availability of relevant data sets remains a threat to continuing research. For this thesis, I compromised on the training data because there was not a large enough data set that contained building models of the optimal building typology. In addition, the success of research in this field is based on access to powerful computational resources. Thankfully, TU Delft has the high-performance computing cluster which I was able to use. Without this resource, it would not have been possible to scale up my research. Despite these threats, there are still opportunities. Currently, there is great interest in AI and this means that many companies and institutions are collecting data to use to train future models. This means that more data will be available in the future. Additionally, the methodology developed through this thesis is not limited to building technology. The model architecture and the research method can be used as a reference for further research in other disciplines. Finally, automation provides access to design to so many more people. It is an important area of research to continue so that everyone can enjoy well-designed spaces.

What are some moral and ethical issues that need to be considered?

When working with machine learning applications, it is important to consider ethical problems that may be present in the current workflow or future applications. Models trained with human-generated and human-selected data will always have biases. The most important aspect is to recognize this and to check for these biases so that the end result is as fair and equal as possible. One aspect is the data itself. Currently there are not many data sets that contain buildings, and the one I found, I would say, contains stereotypes of specific building typologies based on a western lens. Additionally, the buildings included may be related to specific aesthetics versus what is truly found in specific cities, so buildings belonging to middle- and upper-class individuals or high-end developers are more likely to appear. It is important to continue to grow the data sets to be more inclusive and represent all of the architecture we see around us. Additionally, the labels in the data often reflect societal biases. This is why it is critical that data labeling be performed by a large and diverse group of people with different values and belief systems. Even within labeling buildings, labels can have a very specific meaning depending on the region. For example, a porch in the American south is a vernacular architecture feature located in front of the house that is hugely important in the daily social life of people in the

S	INTERNAL **STRENGTHS**
1	Model can be retrained once more data is available
2	Method can be applied to other problems with different data sets
3	Achieveing higher resolution models than past research shows future promise

W	INTERNAL **WEAKNESSSES**
1	Some trail and error required to tune and train model
2	Not possible to know exactly how the model connects the training data to the result
3	Large data sets are needed for training

O	EXTERNAL **OPPORTUNITIES**
1	Current interest in AI likely means more data in the future
2	Not limited to building technology, can support research in other disciplines
3	Automation brings good design to more people

T	EXTERNAL **THREATS**
1	Limited data set availability
2	Optimal building typology data not available
3	Success depends on access to computational resources

Figure 13.1.: A summary of the SWOT analysis of the research approach and method.

region. However, on the east coast of the United States, a porch is frequently located in the back yard and is associated with a more private use for family gatherings and dinner outside. The porch is an example of a similar design feature that has the same name in different regions, but the use, location, and cultural importance vary greatly. Everything we do has nuances, and the best way to ensure that we maintain an ethical approach when applying machine learning is to be aware of our innate biases, check for them, acknowledge them, and correct them when we do realize they exist.

13.2. Societal Impact

What is the academic and societal value of your graduation project?

Existing research on deep learning applications within the architecture, engineering, and construction industry has been completed by various private companies, so there is a lack of published information in the area of building design automation. Additionally, research on 3D geometry generation has focused on small geometry spaces and small objects such as chairs. Based on these gaps, I had the goal of generating building geometries and greatly increasing the size of the geometry space. The gaps identified in my literature review present opportunities to expand industry knowledge. By continuing research on how to automate the design process, the industry can provide access to thoughtful design to a wider audience. This thesis helps address this missing knowledge.

The research that was conducted in this thesis also expands the use of the generative design tools that are used today. By discovering more varied applications of DL in design, these findings can be incorporated into industry tools such as the software used. In the future, generative design tools could propose more than massing, site layouts, and floors plans. The computer can become the design assistant in the way researchers have been striving. This thesis lays the foundation for the use of DL models in generative design tools, so that these tools can propose designs for 3D building geometry. Future researchers can expand on the topic and also explore how GANs and other DL models could design not only building geometry but also building systems.

To what extent has the expected innovation been achieved?

This thesis focused on applying an innovative approach to a known design problem. The research focus turned out to be developing a new GAN architecture for generating building geometry in a larger geometry space. I was able to identify a number of hyperparameters, including Wasserstein loss with a gradient penalty, Leakly ReLU, and the RMSProp optimizer that combined into an architecture that produced building geometry. One aspect of the innovation was the combination of hyperparemeters, and another was the large geometry space and working with building geometry. I was able to expand the size of the geometry space increasing from a size of 64 x 64 x 64 to 160 x 160 x 80.

How does your research impact architectural practice?

Despite improvements in software and automation within the field of architecture, little research has been extended to applying DL to generative problems. To fill this knowledge gap, I wanted to conduct research on the topic of generating 3D building geometry. The research that has been done on this topic was completed by private companies or is not well documented. The goal of this thesis was to begin to fill the gap in industry knowledge about how to automate the design process through DL. By generating building geometry in a large geometry space, this thesis was able to show the potential of GAN. By seeing that GAN can be trained to generate higher-resolution geometry, the industry is hopefully able to recognize the ways in which DL models could be incorporated into generative design tools in the future.

How do you assess the value of the transferability of your project results?

One of the strengths of my project is the transferability of the research. The architecture I developed for improved 3D WGANs performs well in generating building geometry, but it is also applicable to other generative 3D problems. The model can be re-trained with different data sets depending on the requirement. This applies not only to generative design problems within the AEC industry, other fields can also use the architecture for generative design applications. In addition, the methodology used and the lessons learned by creating this architecture can be applied by other researchers in the future. For example, successful architectures tend to perform well when the network is shallow and narrow, so different hyperparameter combinations can be tested and then scaled up afterwards. Regardless of the industry or the field of study that is interested in DL, there is information documented through this thesis that can help others in their development of DL models.

A. Additional Data Set Information

A.1. DCGAN and WGAN Selected Models for Training

Table A.1 indicates which models were used for the training discussed in Chapter 7. When one model was used, it was *mesh2682*. Figure A.1 shows images of the training data set.

name	typology	building type
mesh2682	commercial	house
mesh0067	residential	house
mesh0074	residential	house
mesh0096	residential	house
mesh0109	residential	house
mesh0166	residential	house
mesh0244	residential	house
mesh0379	residential	house
mesh0422	residential	house
mesh0456	residential	house

Table A.1.: Data set model list

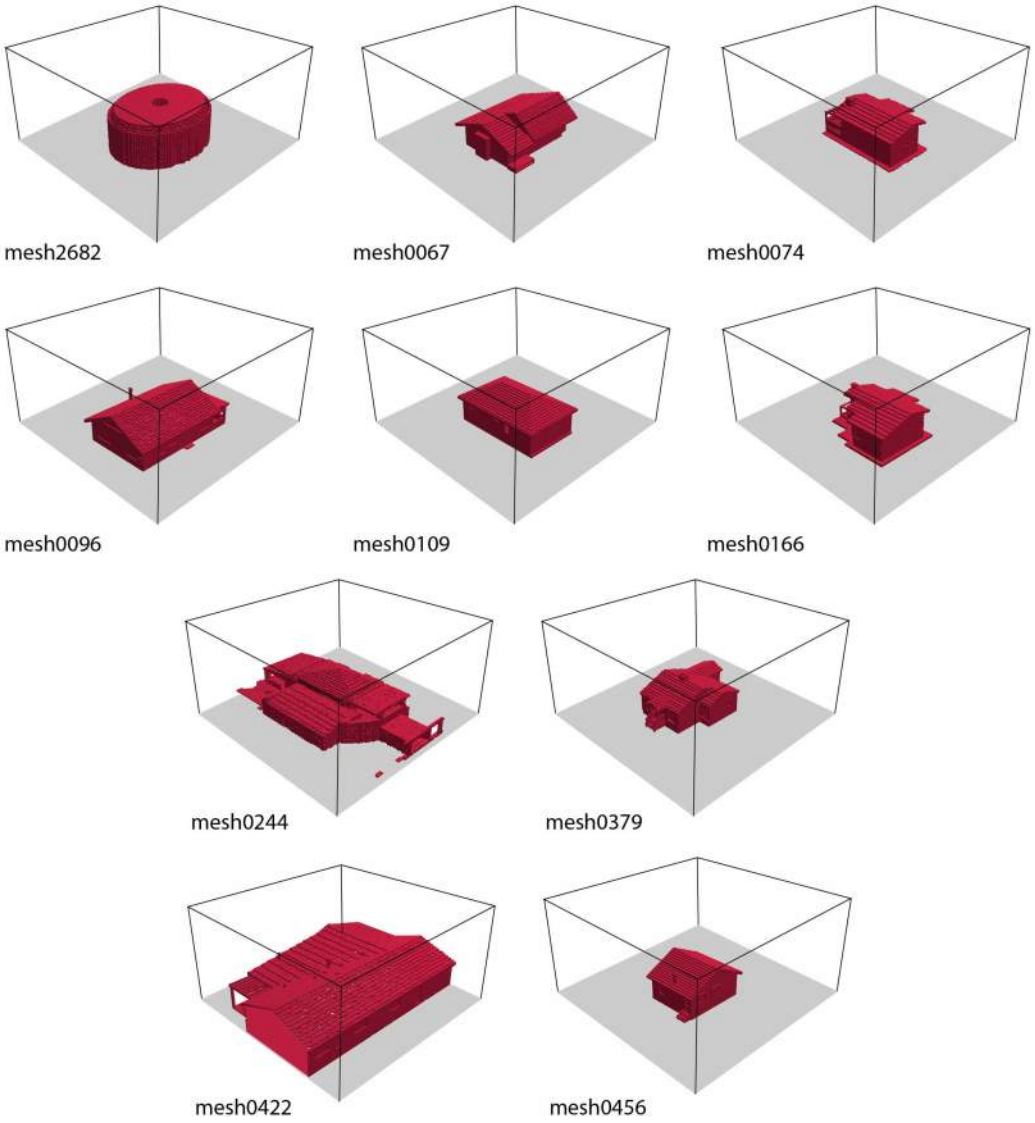


Figure A.1.: Images of the models used for training the initial DCGAN and WGAN.

A.2. WGAN Selected Models for Training

The models discussed in this section were used for experiments in Chapter 8, Chapter 9, and experiments in Chapter 10 where 100 training models were used. Table A.2 indicates which models were used for the training and Figure A.2 shows images of selected 3D models in the training data set.

name	typology	type
mesh0067	residential	house
mesh0074	residential	house
mesh0096	residential	house
mesh0109	residential	house
mesh0166	residential	house
mesh0244	residential	house
mesh0379	residential	house
mesh0422	residential	house
mesh0456	residential	house
mesh0477	residential	house
mesh0498	residential	house
mesh0571	residential	house
mesh0648	residential	house
mesh0658	residential	house
mesh0717	residential	house
mesh0718	residential	house
mesh0829	residential	house
mesh0836	residential	house
mesh0924	residential	house
mesh0944	residential	house
mesh0946	residential	house
mesh0974	residential	house
mesh0976	residential	house
mesh0989	residential	house
mesh0992	residential	house
mesh0993	residential	house
mesh1104	residential	house
mesh1113	residential	house
mesh1131	residential	house
mesh1138	residential	house
mesh1146	residential	house
mesh1156	residential	house
mesh1271	residential	house
mesh1329	residential	house
mesh1373	residential	house
mesh1404	residential	house
mesh1442	residential	house
mesh1498	residential	house
mesh1522	residential	house
mesh1538	residential	house
mesh1694	residential	house
mesh1704	residential	house
mesh1720	residential	house
mesh1891	residential	house
mesh1935	residential	house
mesh2174	residential	house
mesh2200	residential	house
mesh2215	residential	house
mesh2266	residential	house
mesh2277	residential	house
mesh2326	residential	house
mesh2331	residential	house
mesh2523	residential	house
mesh2529	residential	house
mesh2575	residential	house
mesh2627	residential	house
mesh2633	residential	house
mesh2651	residential	house
mesh2720	residential	house
mesh2842	residential	house
mesh2874	residential	house
mesh2965	residential	house
mesh2993	residential	house
mesh3002	residential	house
mesh3007	residential	house
mesh3034	residential	house
mesh3083	residential	house
mesh3094	residential	house
mesh3134	residential	house
mesh3176	residential	house
mesh3240	residential	house
mesh3251	residential	house
mesh3367	residential	house
mesh3372	residential	house
mesh3433	residential	house
mesh3480	residential	house
mesh3513	residential	house
mesh3844	residential	house
mesh3857	residential	house
mesh3979	residential	house
mesh3998	residential	house
mesh4182	residential	house
mesh4195	residential	house
mesh4333	residential	house
mesh4362	residential	house
mesh4466	residential	house
mesh4580	residential	house
mesh4668	residential	house
mesh4729	residential	house
mesh4844	residential	house
mesh4893	residential	house
mesh4904	residential	house
mesh4972	residential	house
mesh5054	residential	house
mesh5076	residential	house
mesh5145	residential	house
mesh5173	residential	house
mesh5211	residential	house
mesh5257	residential	house

Table A.2.: Data set model list

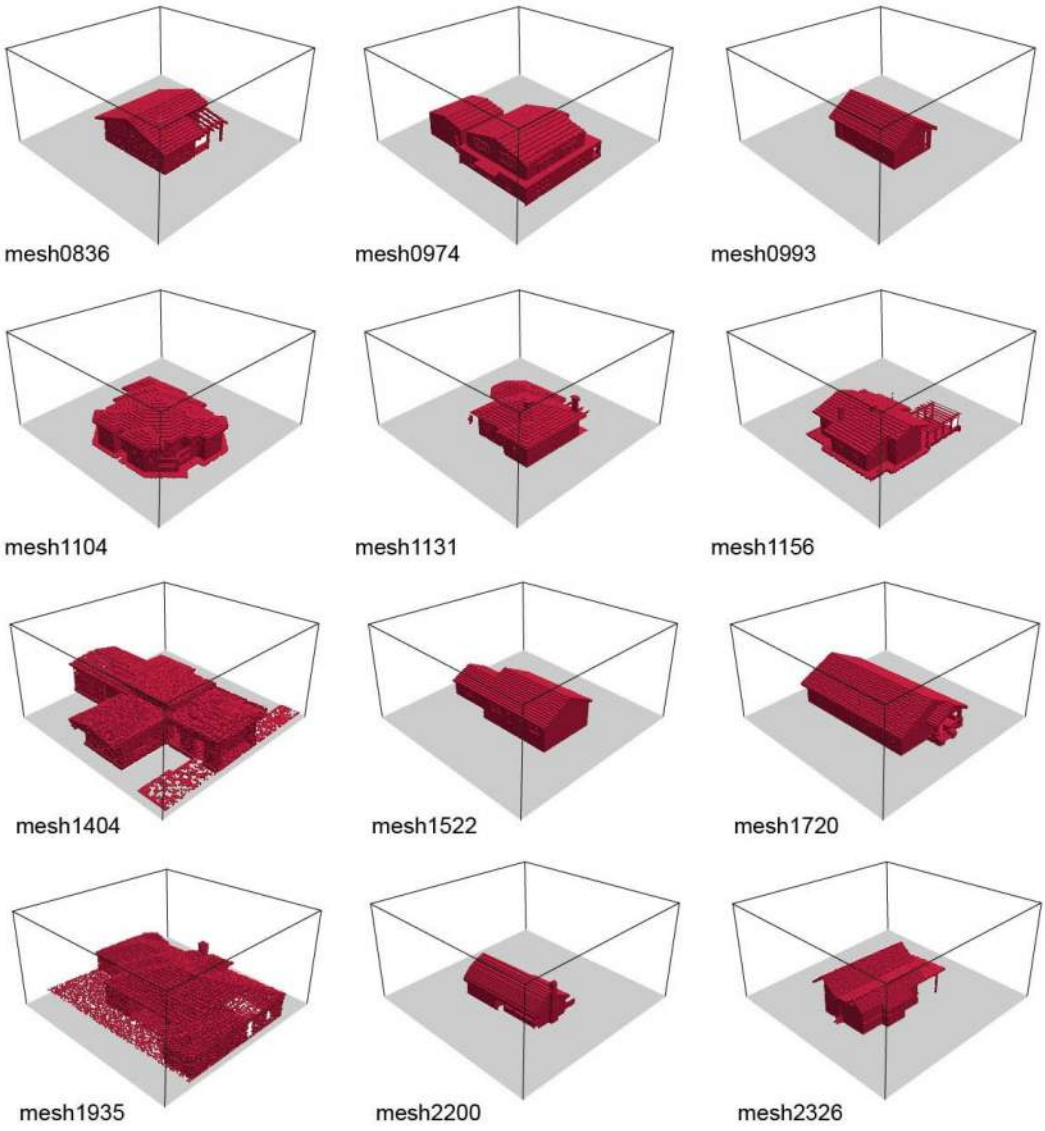


Figure A.2.: Images of a selection of models used for training the WGAN models while tuning hyper-parameters.

A.3. Larger Data Set for Training

Certain experiments in Chapter 10 used a data set of 200 models. Table A.3 indicates which models were used for the training. These models are in addition to those listed in Table A.2. Figure A.3 shows images of selected 3D models in the training data set.

name	typology	type
mesh5203	residential	house
mesh5360	residential	house
mesh5387	residential	house
mesh5438	residential	house
mesh5503	residential	house
mesh5560	residential	house
mesh5582	residential	house
mesh5634	residential	house
mesh5640	residential	house
mesh5678	residential	house
mesh5686	residential	house
mesh5742	residential	house
mesh5819	residential	house
mesh5951	residential	house
mesh5955	residential	house
mesh5957	residential	house
mesh6035	residential	house
mesh6036	residential	house
mesh6051	residential	house
mesh6114	residential	house
mesh6207	residential	house
mesh6415	residential	house
mesh6424	residential	house
mesh6450	residential	house
mesh6505	residential	house
mesh6614	residential	house
mesh6622	residential	house
mesh6668	residential	house
mesh6742	residential	house
mesh6762	residential	house
mesh6816	residential	house
mesh6837	residential	house
mesh6840	residential	house
mesh6883	residential	house
mesh6897	residential	house
mesh6960	residential	house
mesh6995	residential	house
mesh6999	residential	house
mesh7024	residential	house
mesh7219	residential	house
mesh7233	residential	house
mesh7234	residential	house
mesh7377	residential	house
mesh7439	residential	house
mesh7447	residential	house
mesh7474	residential	house
mesh7555	residential	house
mesh7563	residential	house
mesh7583	residential	house
mesh7780	residential	house
mesh7882	residential	house
mesh7901	residential	house
mesh7919	residential	house
mesh8041	residential	house
mesh8042	residential	house
mesh8290	residential	house
mesh8299	residential	house
mesh8452	residential	house
mesh8525	residential	house
mesh8528	residential	house
mesh8547	residential	house
mesh8612	residential	house
mesh8772	residential	house
mesh8882	residential	house
mesh9002	residential	house
mesh9022	residential	house
mesh9023	residential	house
mesh9145	residential	house
mesh9339	residential	house
mesh9405	residential	house
mesh9492	residential	house
mesh9498	residential	house
mesh9532	residential	house
mesh9582	residential	house
mesh9640	residential	house
mesh9803	residential	house
mesh9808	residential	house
mesh9834	residential	house
mesh0439	residential	villa
mesh0476	residential	villa
mesh0823	residential	villa
mesh1061	residential	villa
mesh1261	residential	villa
mesh1337	residential	villa
mesh1410	residential	villa
mesh1794	residential	villa
mesh1796	residential	villa
mesh2615	residential	villa
mesh2816	residential	villa
mesh3101	residential	villa
mesh3275	residential	villa
mesh3500	residential	villa
mesh3665	residential	villa
mesh5116	residential	villa
mesh5389	residential	villa
mesh5507	residential	villa
mesh5860	residential	villa
mesh5915	residential	villa
mesh5948	residential	villa

Table A.3.: Data set model list

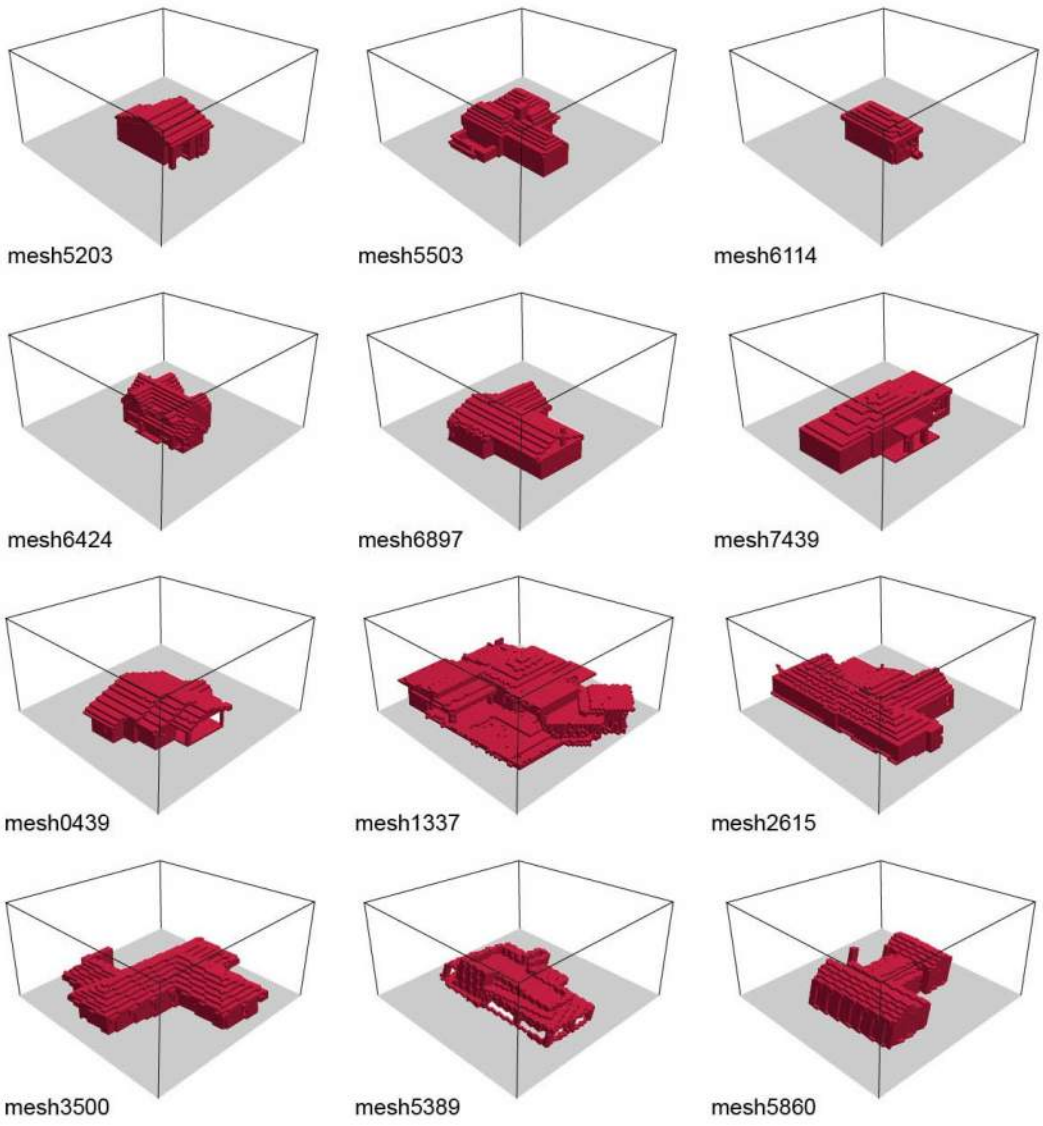


Figure A.3.: Images of a selection of models used for training the WGAN model when using 200 data points.

A.4. Visualizations of Selected Models from Released Training Data Set

The pre-processed data set, HouseNet, discussed in this report can be found on 4TU.ResearchData at https://doi.org/10.4121/4d82052e-650c-4775-8bd9-623df68991b6.v1.

The data set contains models that originated from the BuildingNet v0.1 data set that Selvaraju et al. [2021] released in 2021. The steps that I took to modify the existing data set so that the data aligns with the needs of generative deep learning tasks are outlined in Chapter 5. Figure A.4 shows images

of selected 3D models in the training data set. The modified files that are located on 4TU.ResearchData include the following files for each building model:

- a modified point cloud file (.ply) which came from the original data set and was modified by removing site geometry and relabeling the points with only four labels to make the models suited for deep learning for generative design
- a modified file containing label information (.json) which came from the original data set and was modified to make the models suited for deep learning for generative design
- a pickle file (.pkl) generated through pre-processing which contains information about the point cloud including the file name, label array, color array, typology, and building type information

HouseNet uses the following labels: 1:'wall' 2:'window' 4:'roof' 6:'door'.

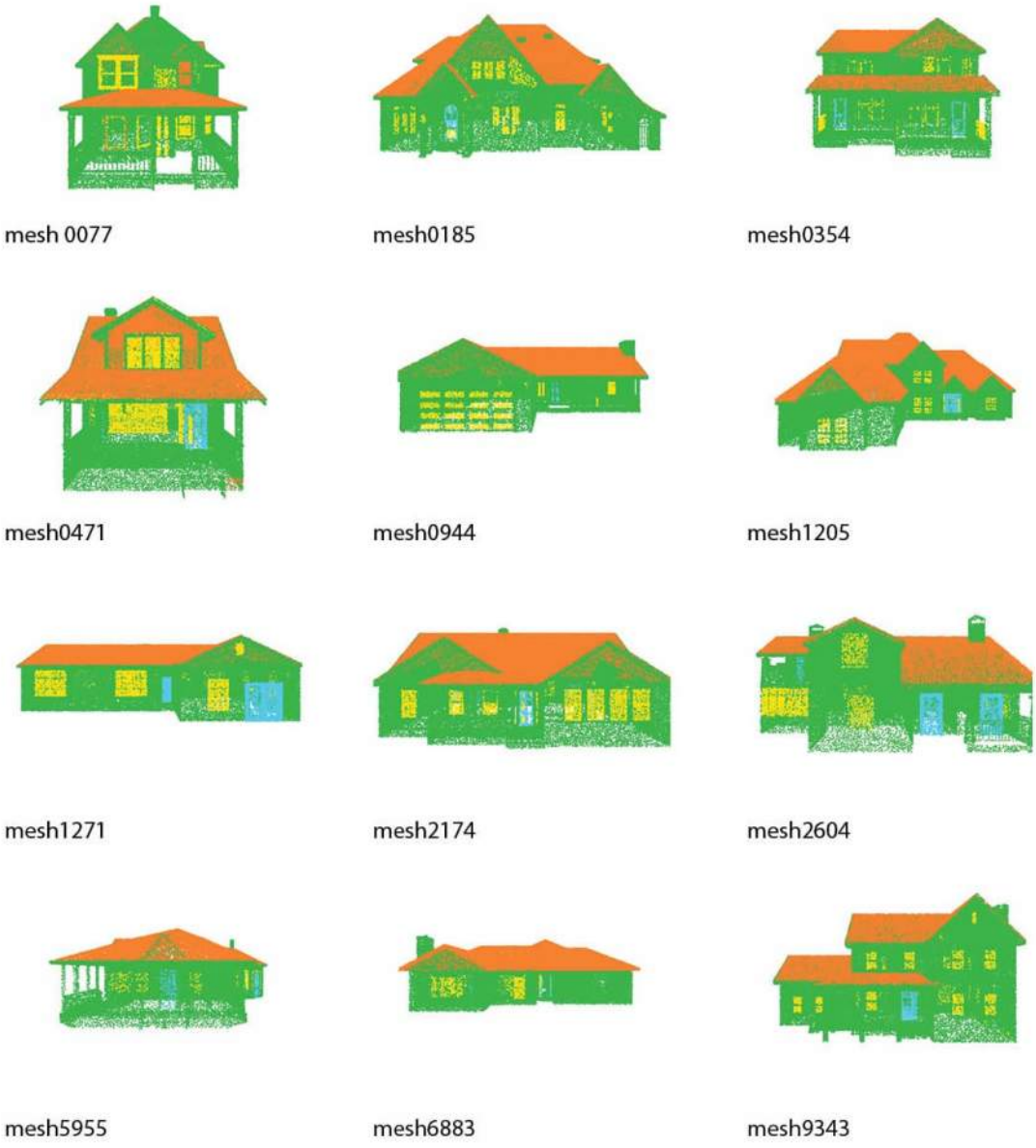


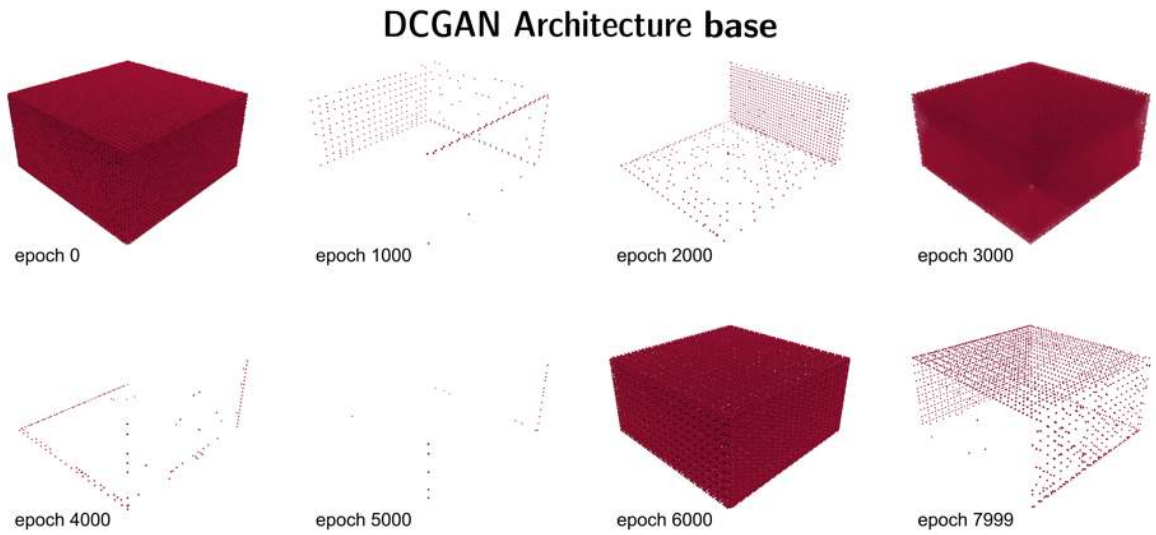
Figure A.4.: Images of a selection of models from the pre-processed data set.

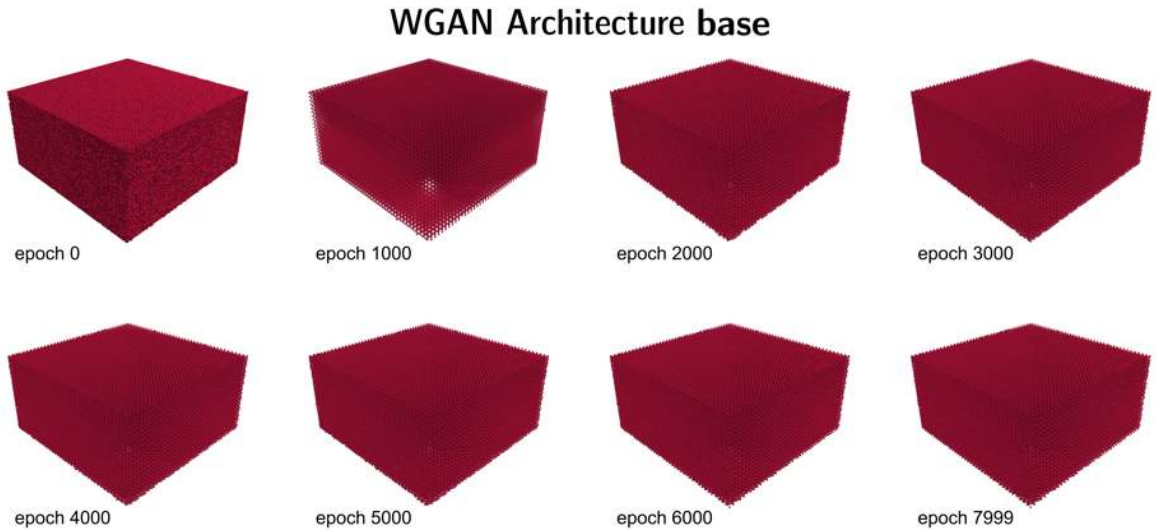
B. Experiment Results

This appendix visualizes geometry that was the output of the generator from the different architectures tested during this thesis. The output was visualized after a certain number of training epochs as noted in the images.

B.1. DCGAN and WGAN Base Architecture

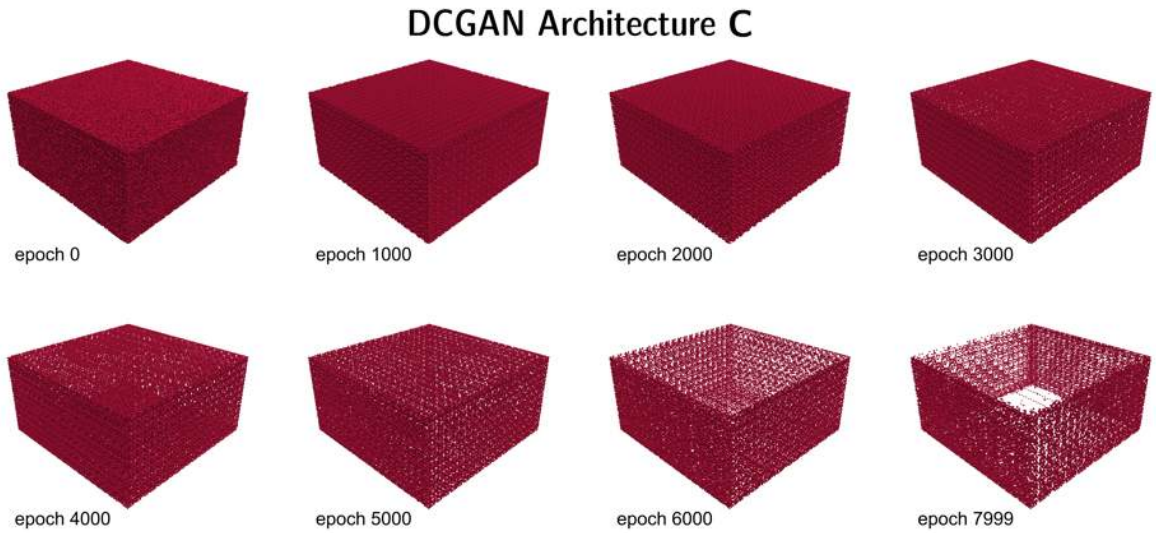
These experiments were run using network width and depth of Architecture **11** as indicated in Table 9.1. These experiments were run with a training data set size of 1 model described in Appendix A.1.

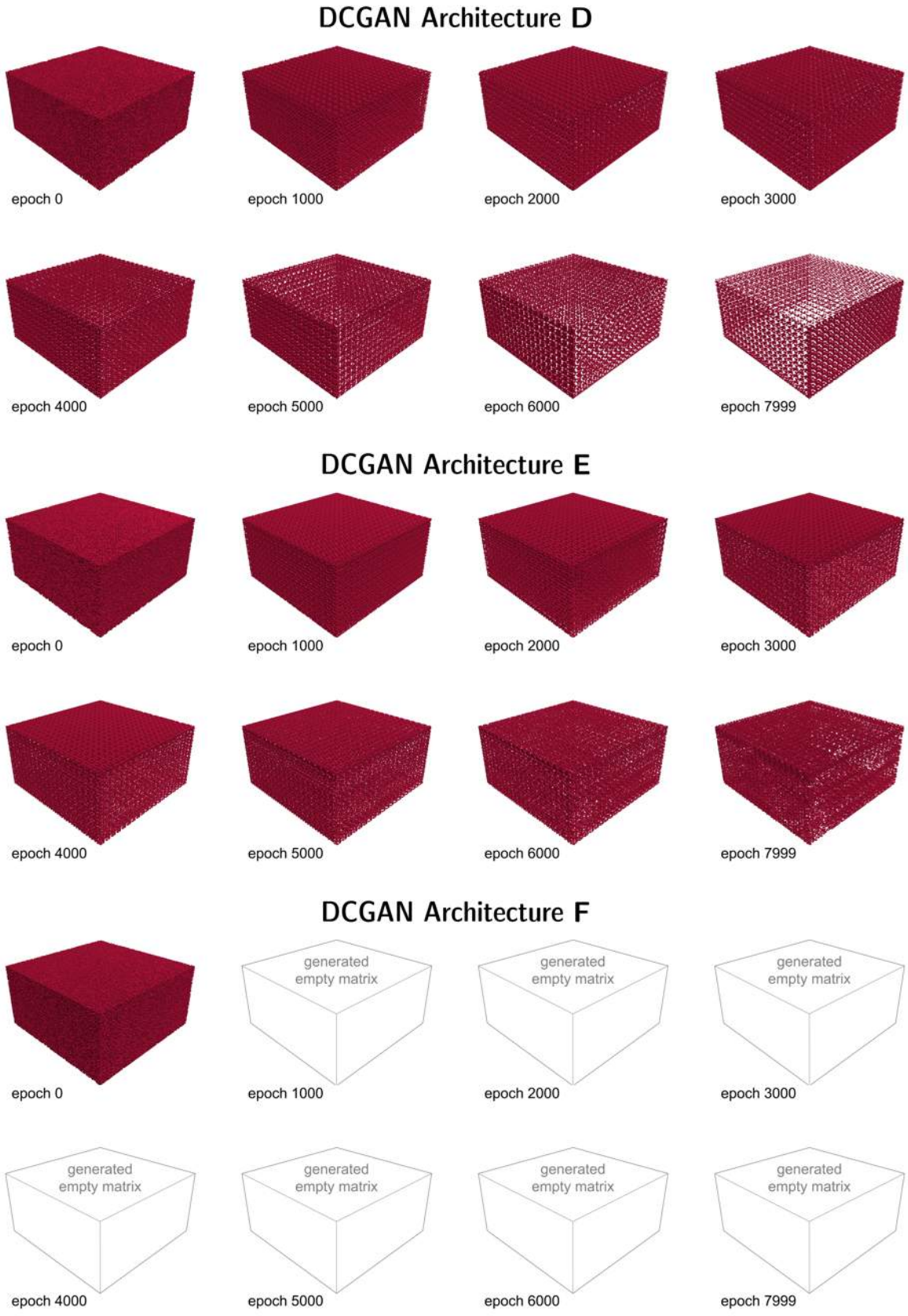


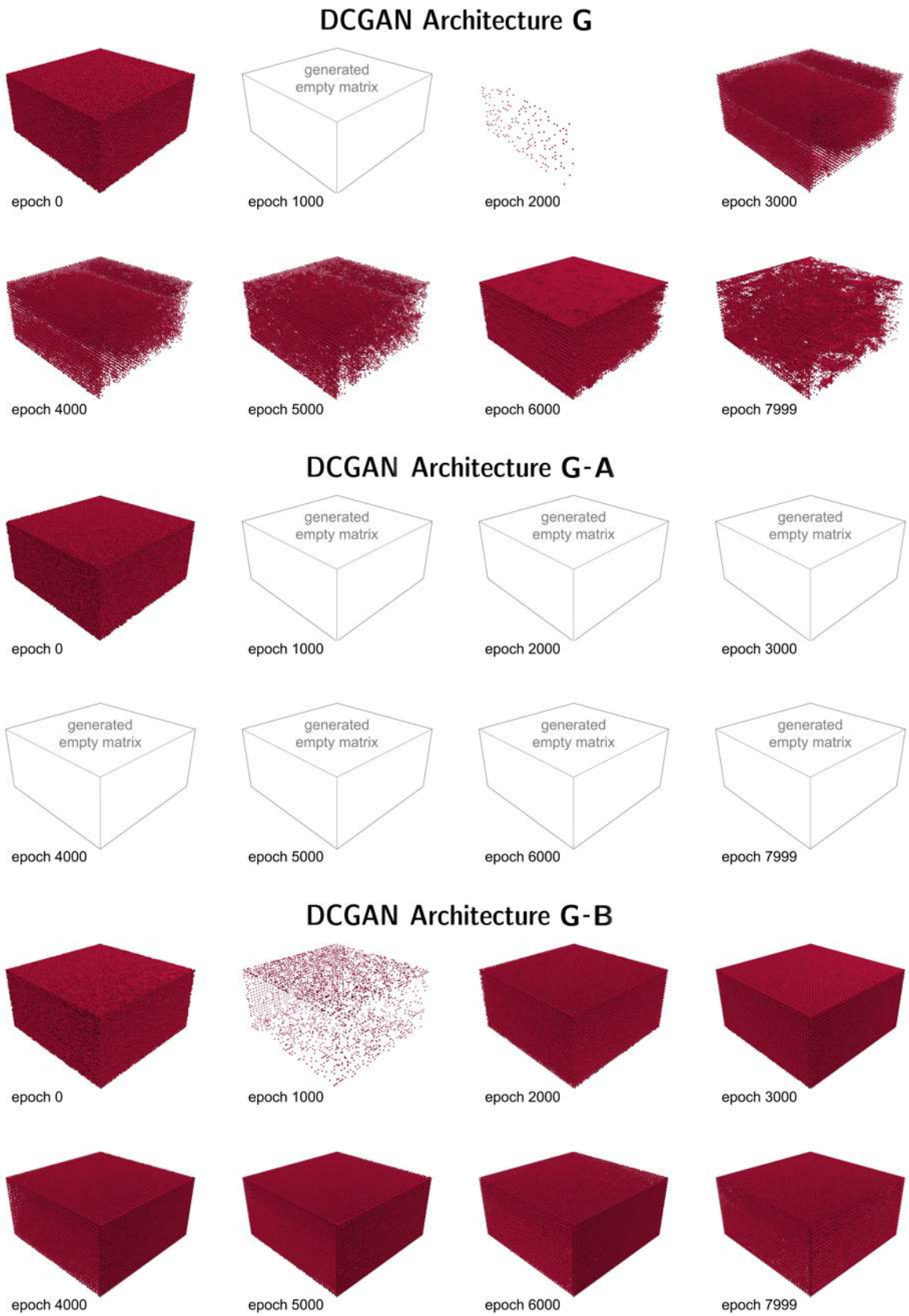


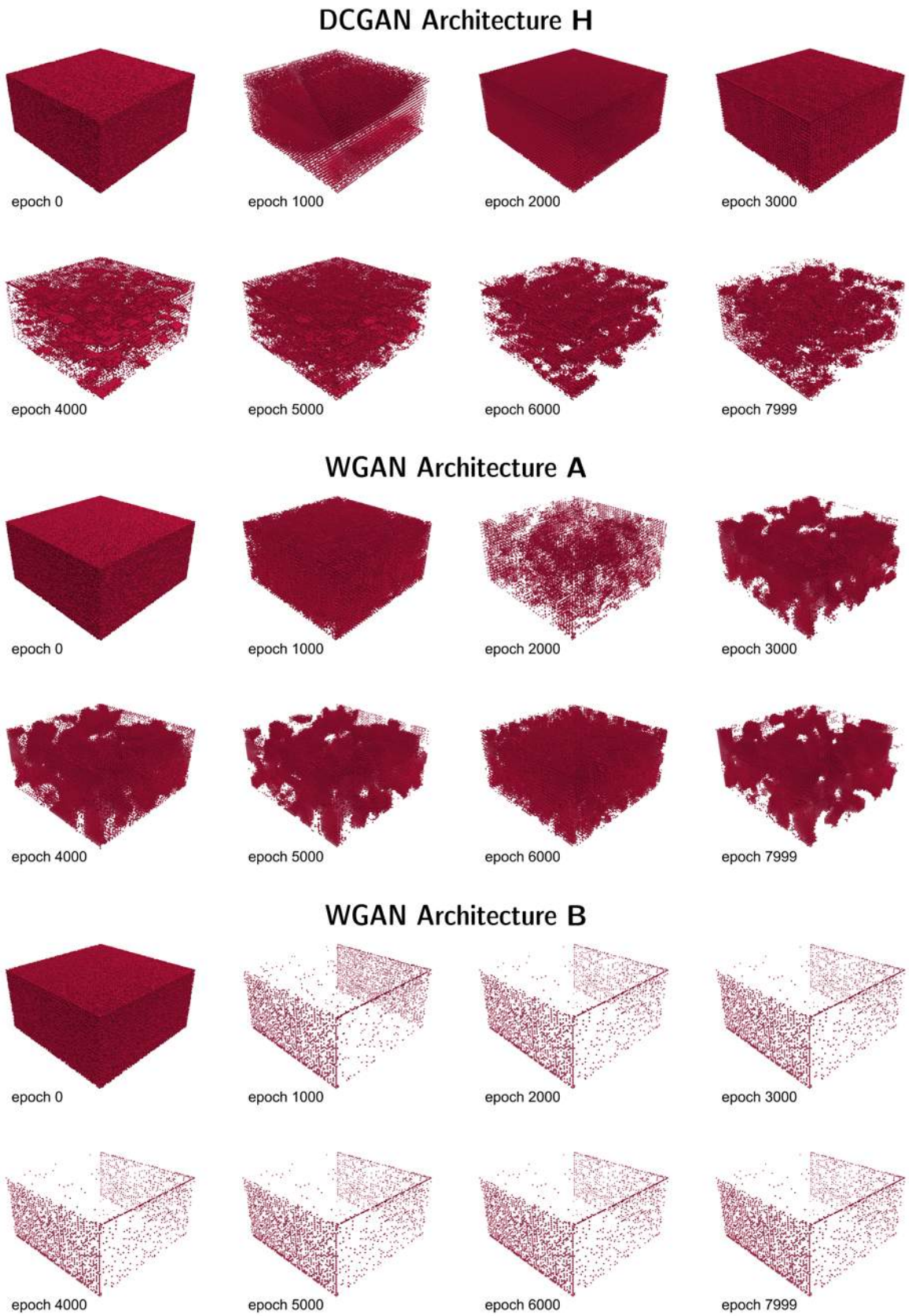
B.2. DCGAN and WGAN Architectures A through H

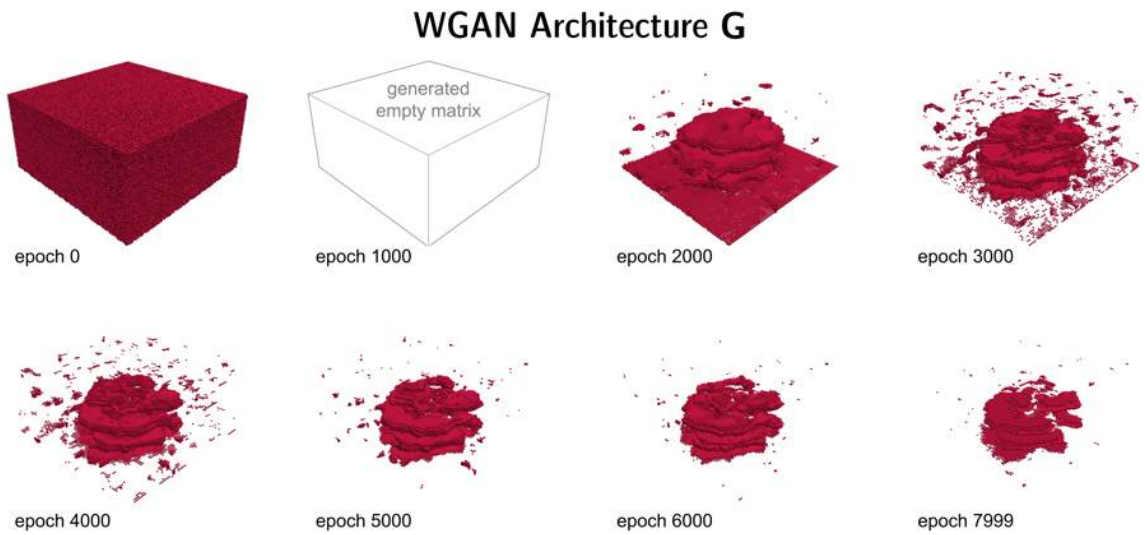
DCGAN was tested with Architectures **C**, **D**, **E**, **F**, **G**, **G-A**, **G-B**, and **H**. WGAN was tested with Architectures **A**, **B**, and **G**. For descriptions of the architectures, please see Table 7.2. These experiments were run using network width and depth of Architecture **11** as indicated in Table 9.1. These experiments were run with a training data set size of 1 model described in Appendix A.1.




DCGAN Architecture D
epoch 0
epoch 1000
epoch 2000
epoch 3000
epoch 4000
epoch 5000
epoch 6000
epoch 7999
DCGAN Architecture E
epoch 0
epoch 1000
epoch 2000
epoch 3000
epoch 4000
epoch 5000
epoch 6000
epoch 7999
DCGAN Architecture F
epoch 0
generated empty matrix
epoch 1000
generated empty matrix
epoch 2000
generated empty matrix
epoch 3000
generated empty matrix
epoch 4000
generated empty matrix
epoch 5000
generated empty matrix
epoch 6000
generated empty matrix
epoch 7999

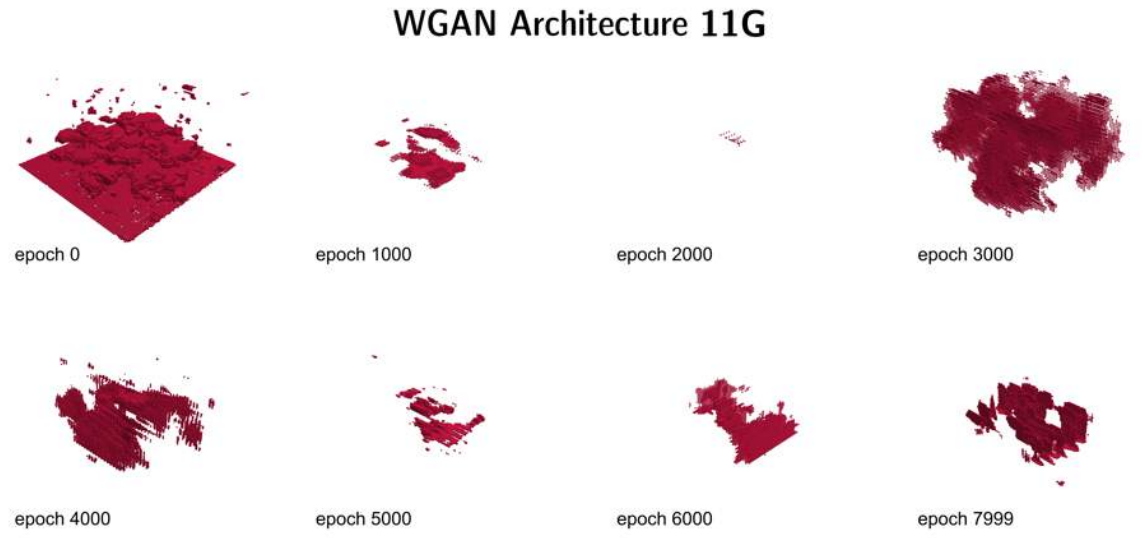

DCGAN Architecture G
epoch 0
generated
empty matrix
epoch 1000
epoch 2000
epoch 3000
epoch 4000
epoch 5000
epoch 6000
epoch 7999
DCGAN Architecture G-A
epoch 0
generated
empty matrix
epoch 1000
generated
empty matrix
epoch 2000
generated
empty matrix
epoch 3000
generated
empty matrix
epoch 4000
generated
empty matrix
epoch 5000
generated
empty matrix
epoch 6000
generated
empty matrix
epoch 7999
DCGAN Architecture G-B
epoch 0
epoch 1000
epoch 2000
epoch 3000
epoch 4000
epoch 5000
epoch 6000
epoch 7999


DCGAN Architecture H
epoch 0
epoch 1000
epoch 2000
epoch 3000
epoch 4000
epoch 5000
epoch 6000
epoch 7999
WGAN Architecture A
epoch 0
epoch 1000
epoch 2000
epoch 3000
epoch 4000
epoch 5000
epoch 6000
epoch 7999
WGAN Architecture B
epoch 0
epoch 1000
epoch 2000
epoch 3000
epoch 4000
epoch 5000
epoch 6000
epoch 7999

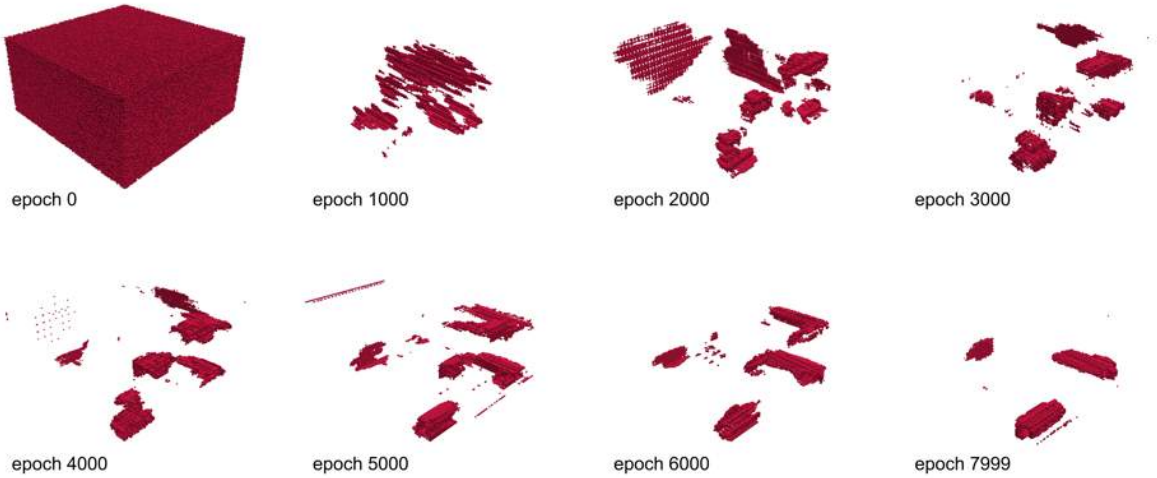


B.3. WGAN Architectures G, J through W

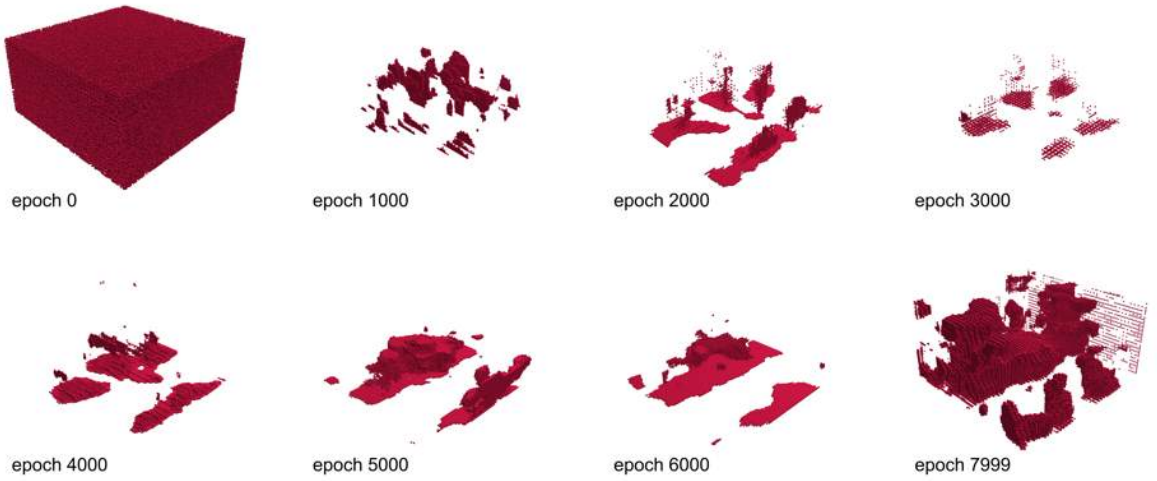
For descriptions of the architectures, please see Table 8.1. These experiments were all run using network width and depth of Architecture **11** as indicated in Table 9.1 and they all had the same training data set with a size of 100 models described in Appendix A.2.



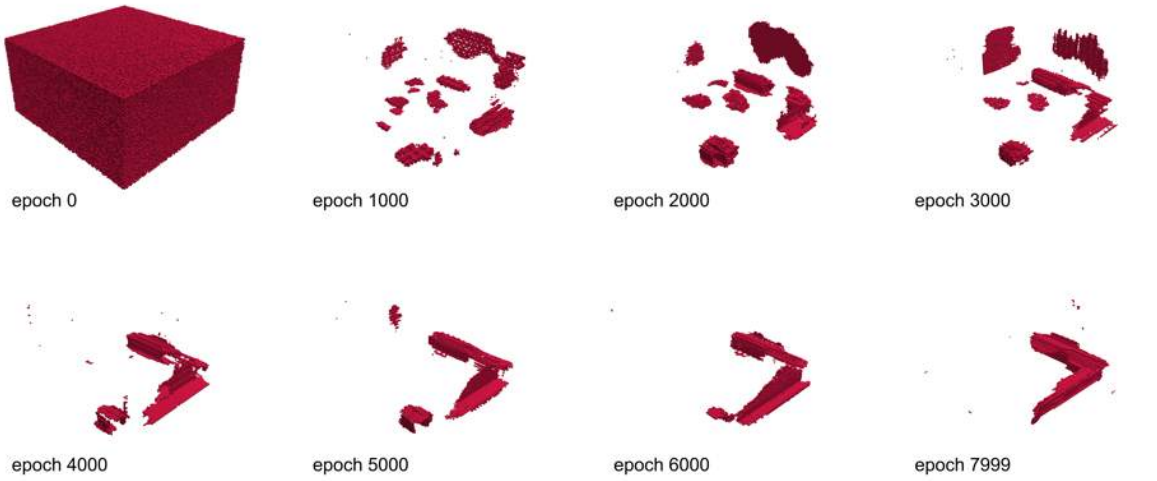
WGAN Architecture 11J



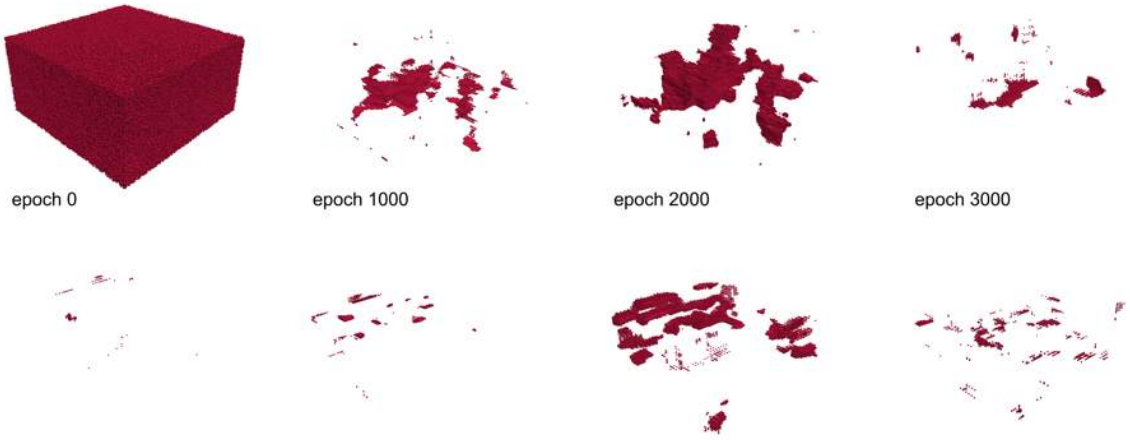
WGAN Architecture 11K



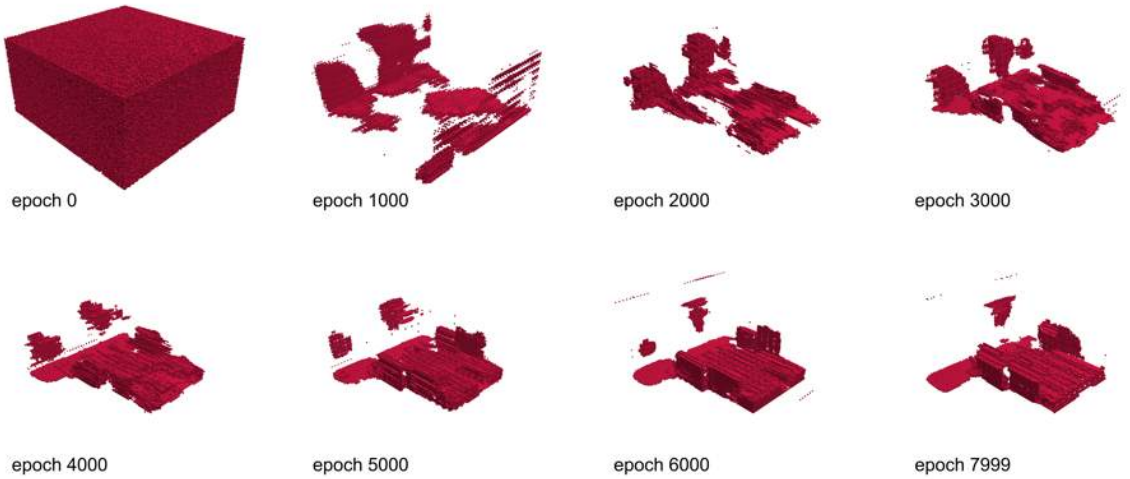
WGAN Architecture 11L



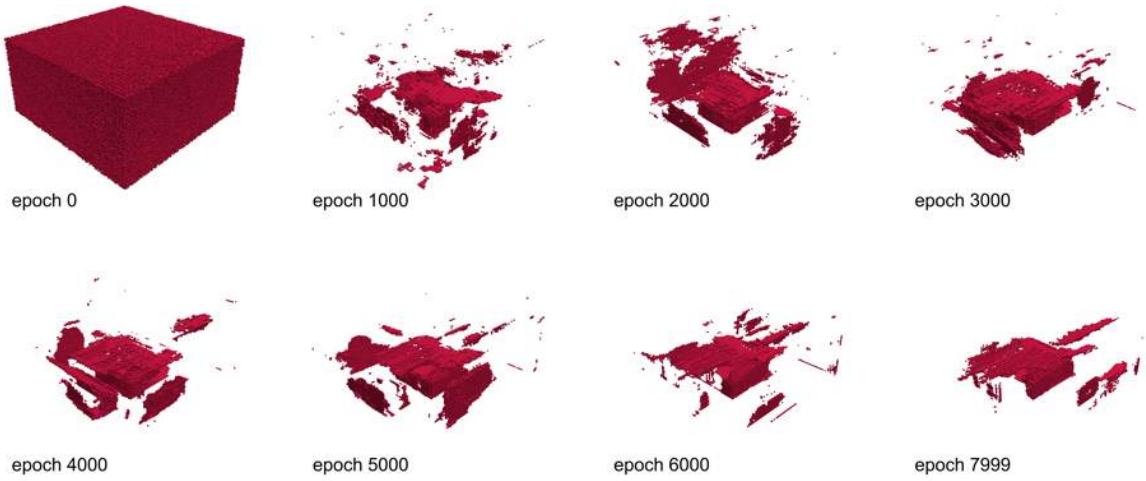
WGAN Architecture 11M

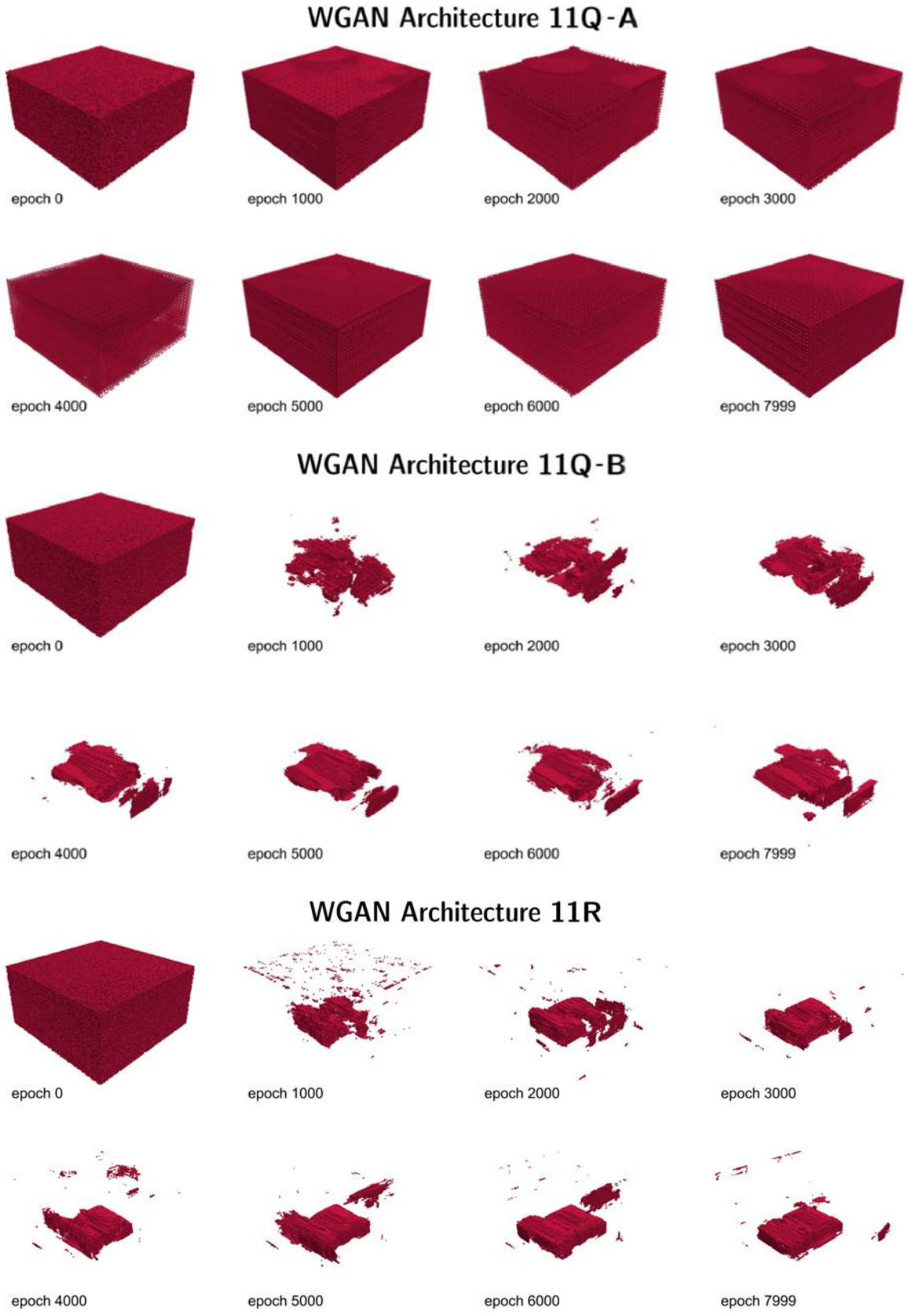


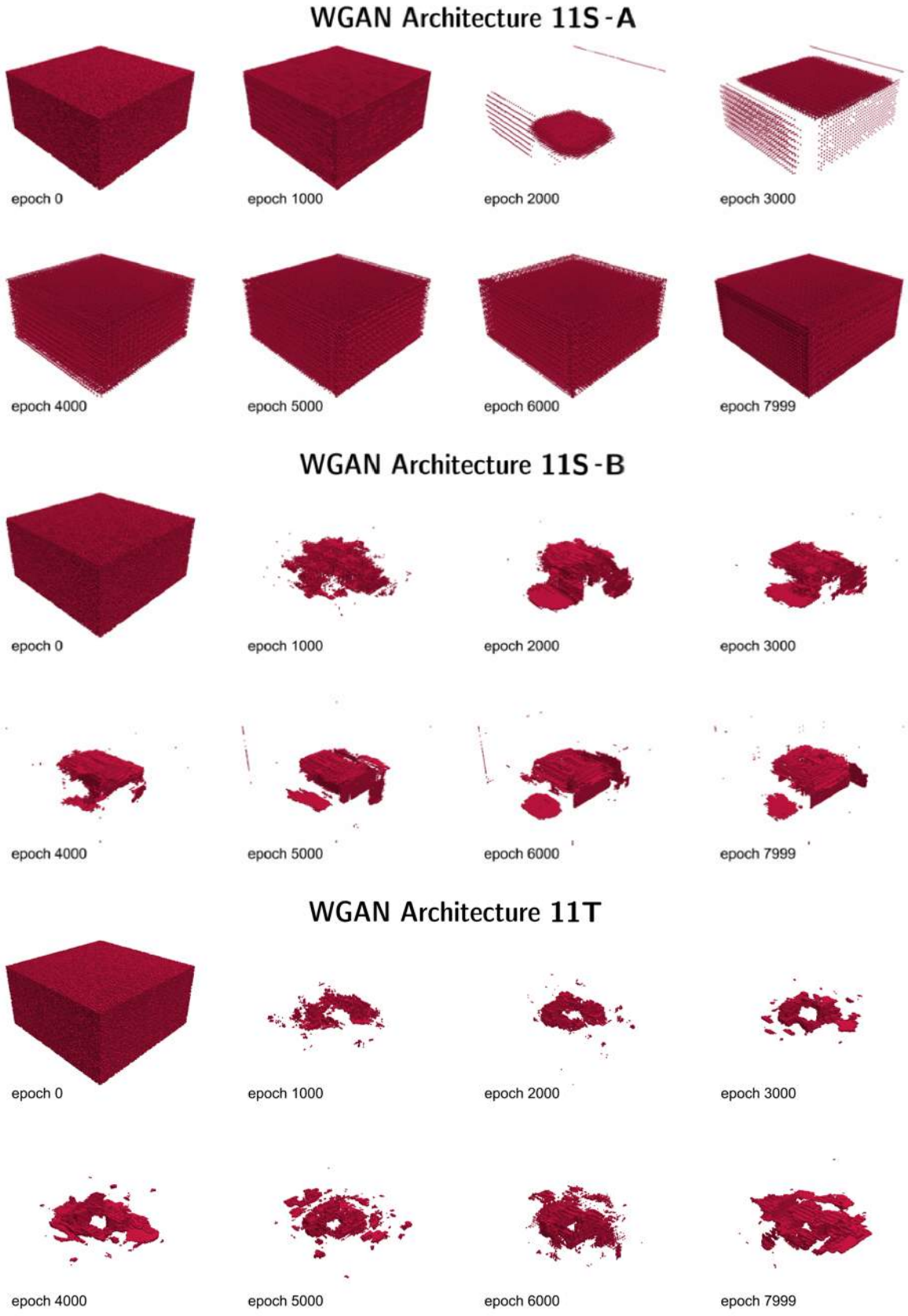
WGAN Architecture 11N



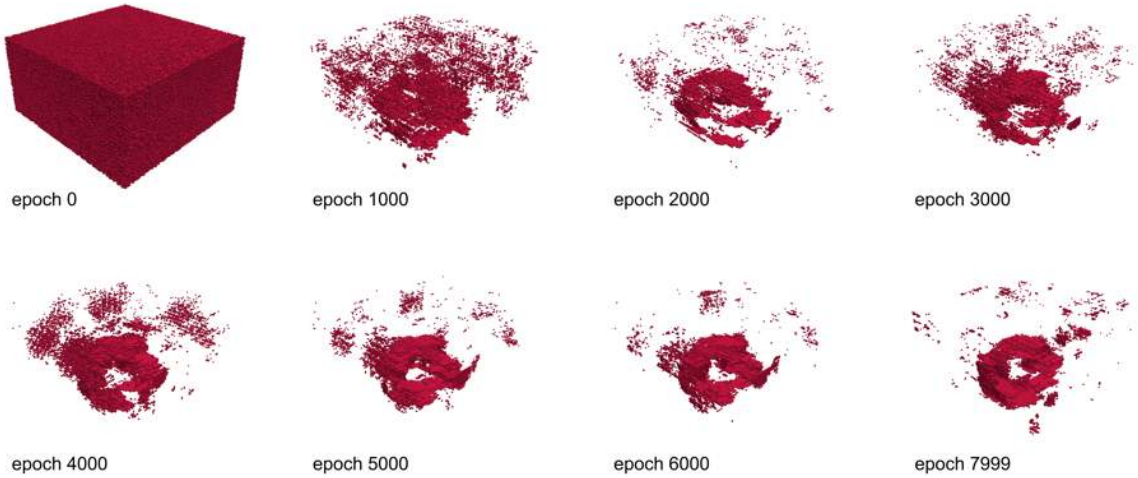
WGAN Architecture 11P



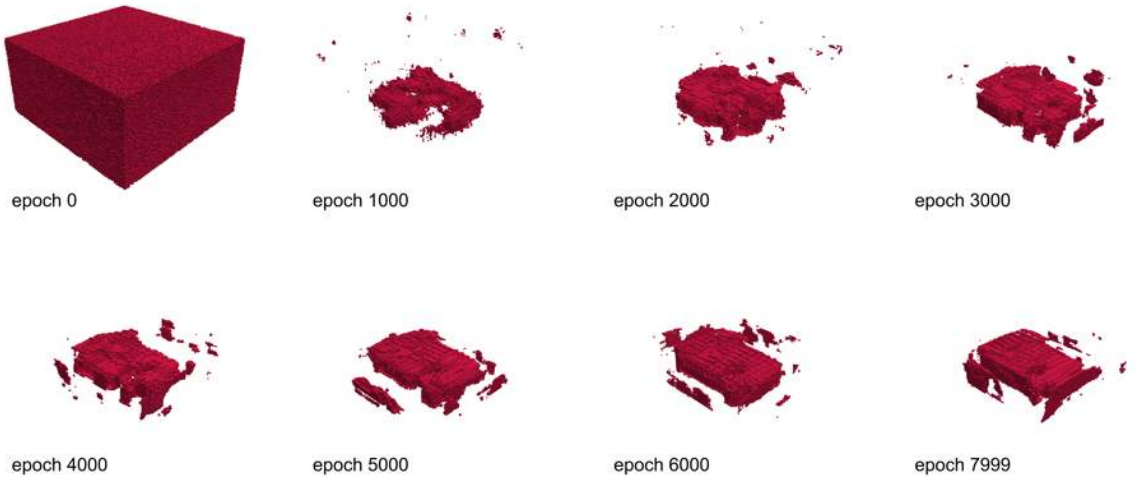

WGAN Architecture 11Q-A
epoch 0
epoch 1000
epoch 2000
epoch 3000
epoch 4000
epoch 5000
epoch 6000
epoch 7999
WGAN Architecture 11Q-B
epoch 0
epoch 1000
epoch 2000
epoch 3000
epoch 4000
epoch 5000
epoch 6000
epoch 7999
WGAN Architecture 11R
epoch 0
epoch 1000
epoch 2000
epoch 3000
epoch 4000
epoch 5000
epoch 6000
epoch 7999


WGAN Architecture 11S-A
epoch 0
epoch 1000
epoch 2000
epoch 3000
epoch 4000
epoch 5000
epoch 6000
epoch 7999
WGAN Architecture 11S-B
epoch 0
epoch 1000
epoch 2000
epoch 3000
epoch 4000
epoch 5000
epoch 6000
epoch 7999
WGAN Architecture 11T
epoch 0
epoch 1000
epoch 2000
epoch 3000
epoch 4000
epoch 5000
epoch 6000
epoch 7999

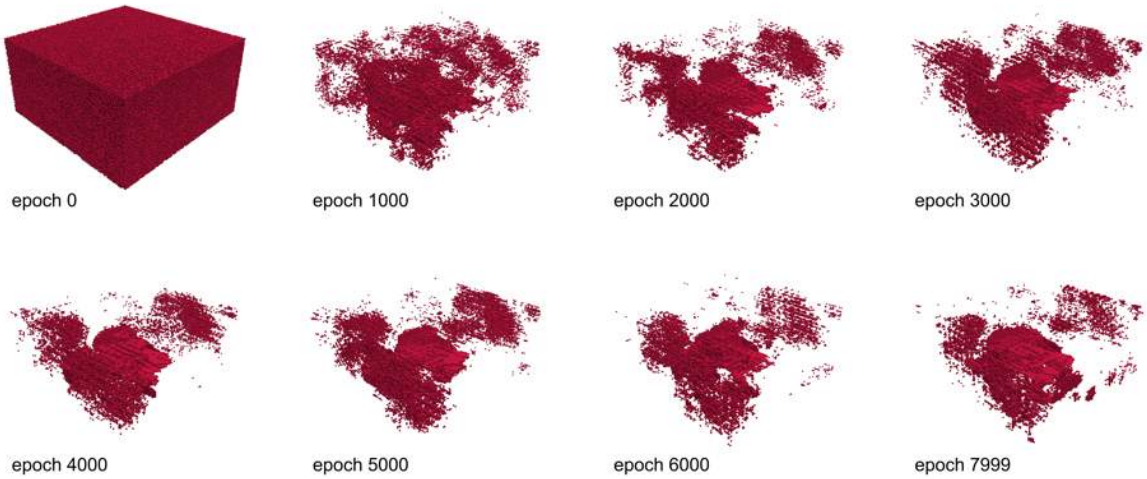
WGAN Architecture 11U



WGAN Architecture 11V

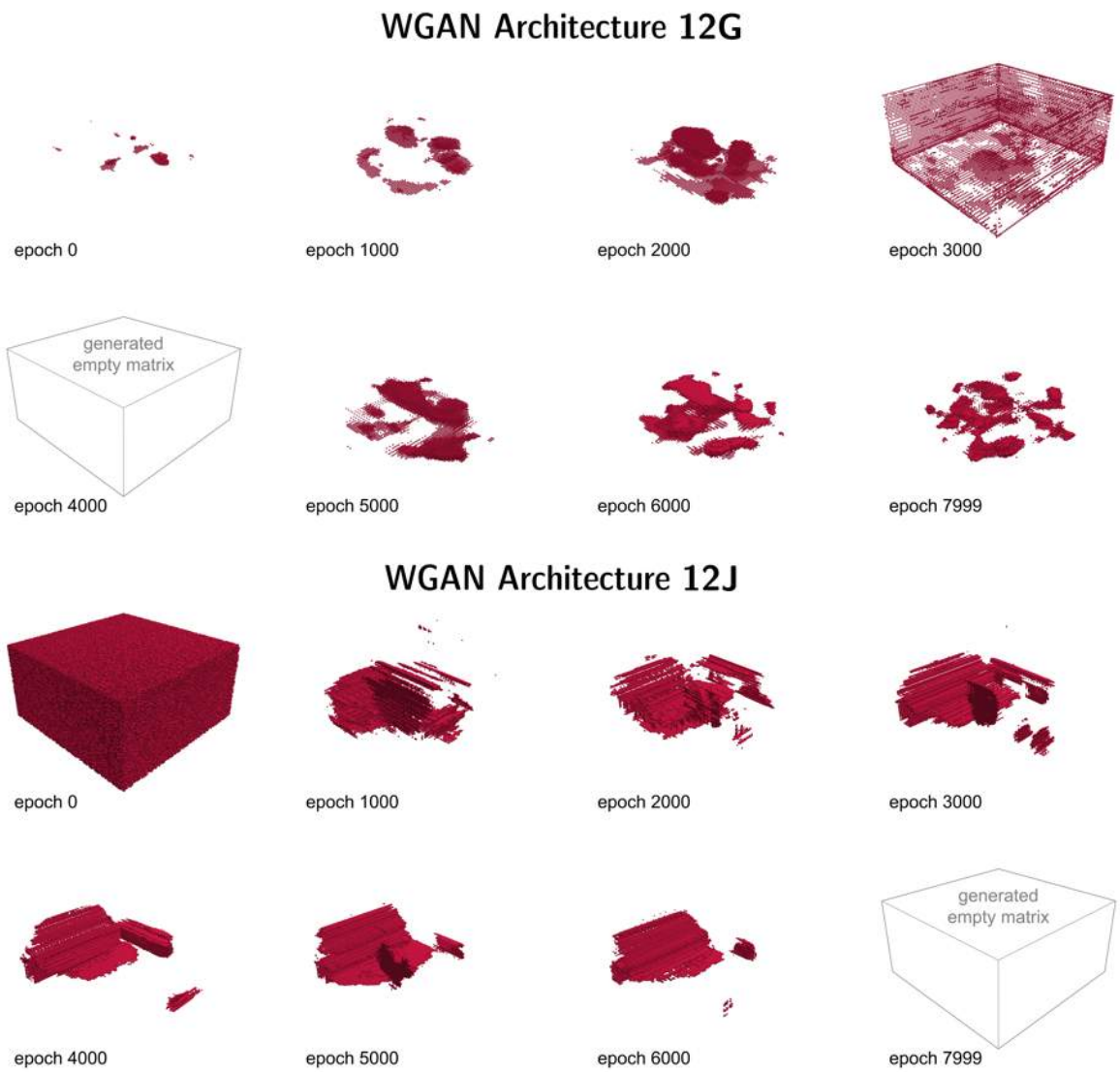


WGAN Architecture 11W

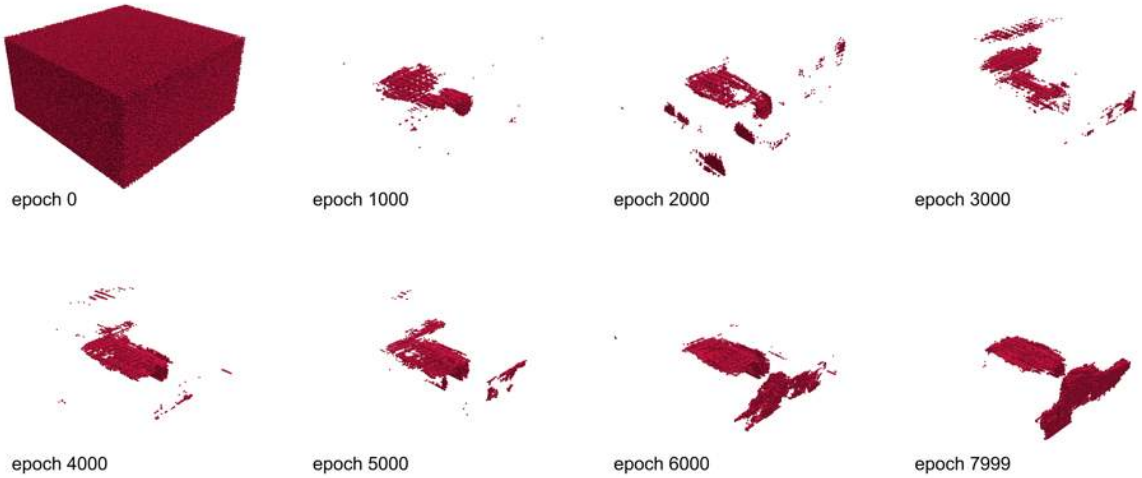


B.4. Network Width and Depth

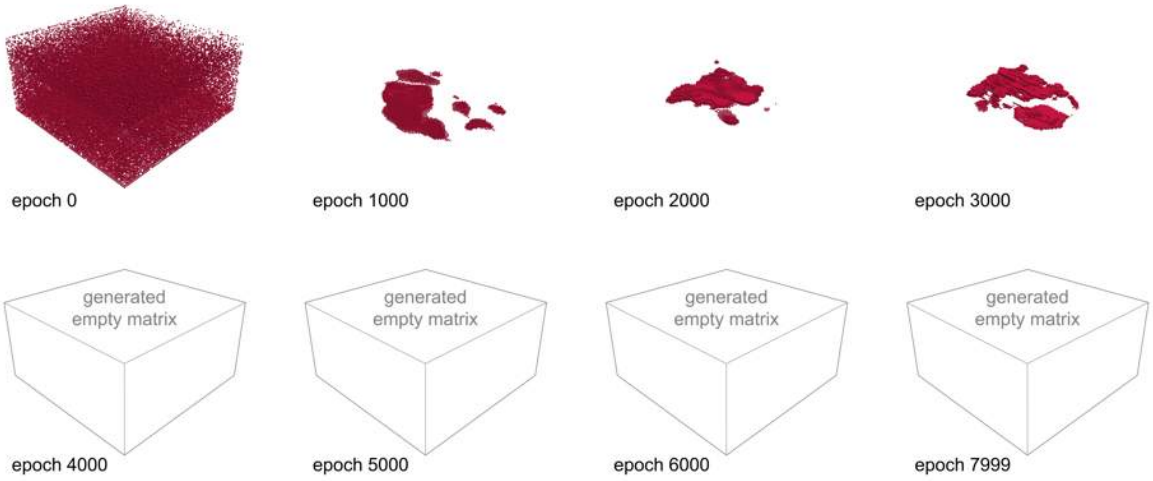
For descriptions of the architectures, please see Table 8.1. These experiments were run using network width and depth of different architectures indicated in Table 9.1. These experiments were run with a training data set size of 100 models described in Appendix A.2.



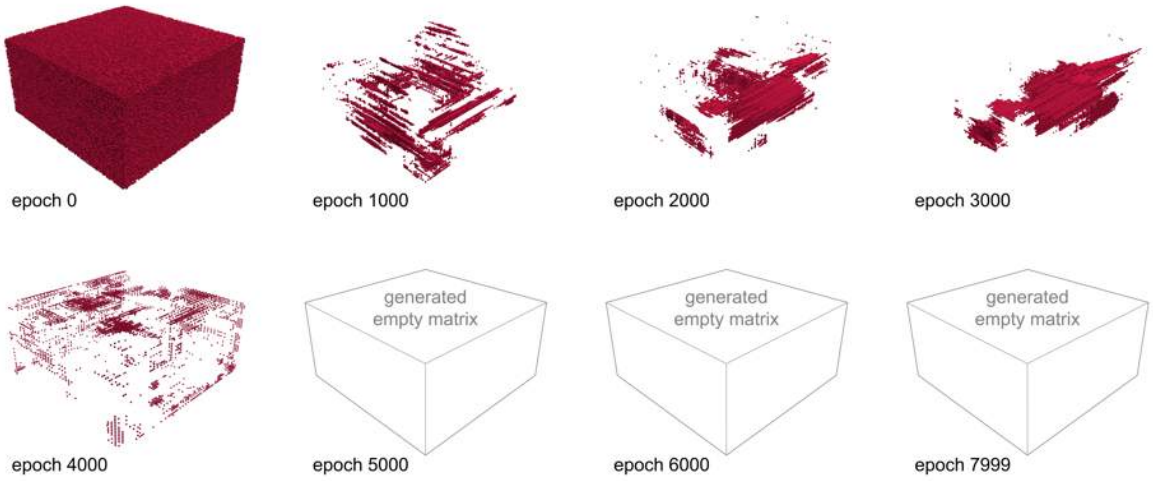
WGAN Architecture 12R



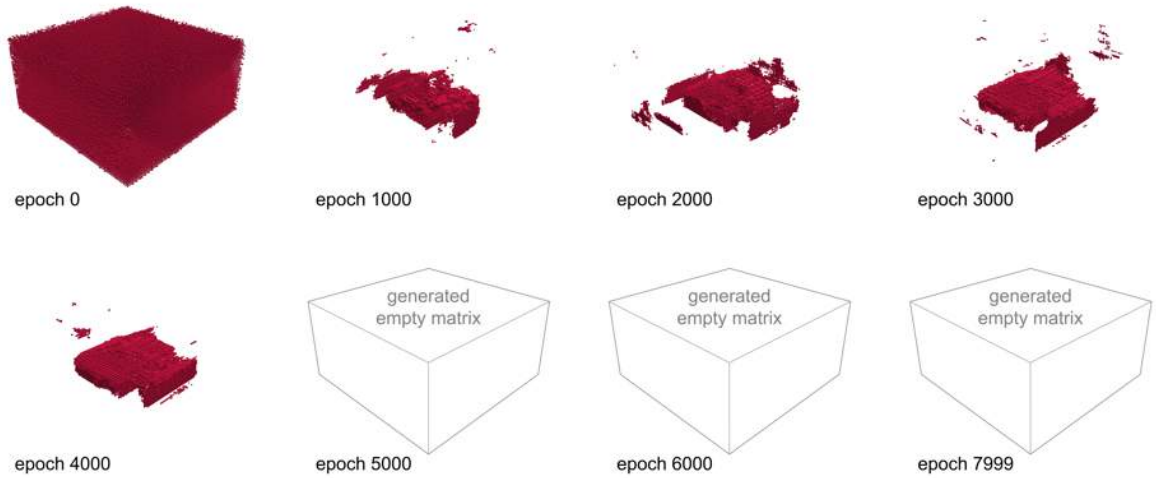
WGAN Architecture 13G



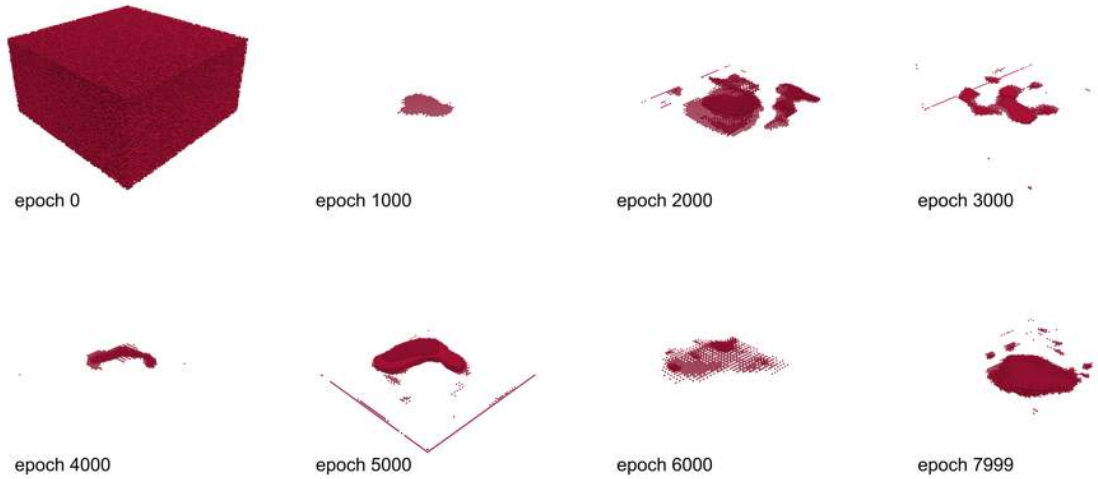
WGAN Architecture 13J



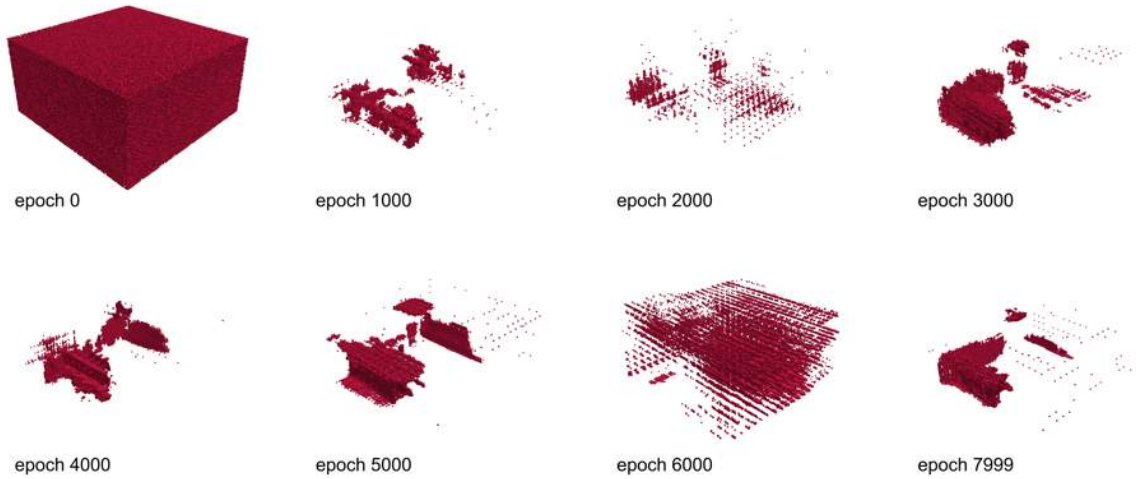
WGAN Architecture 13R



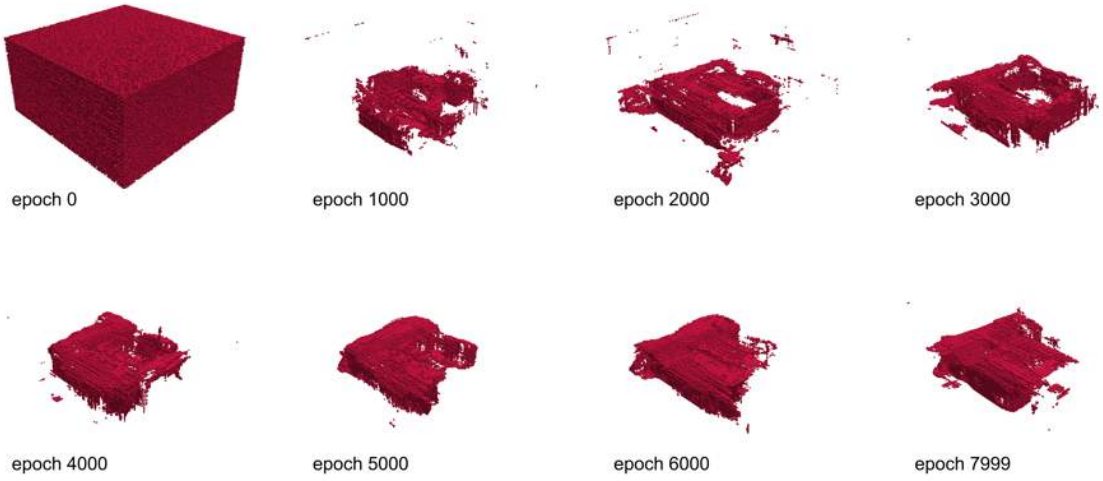
WGAN Architecture 14G



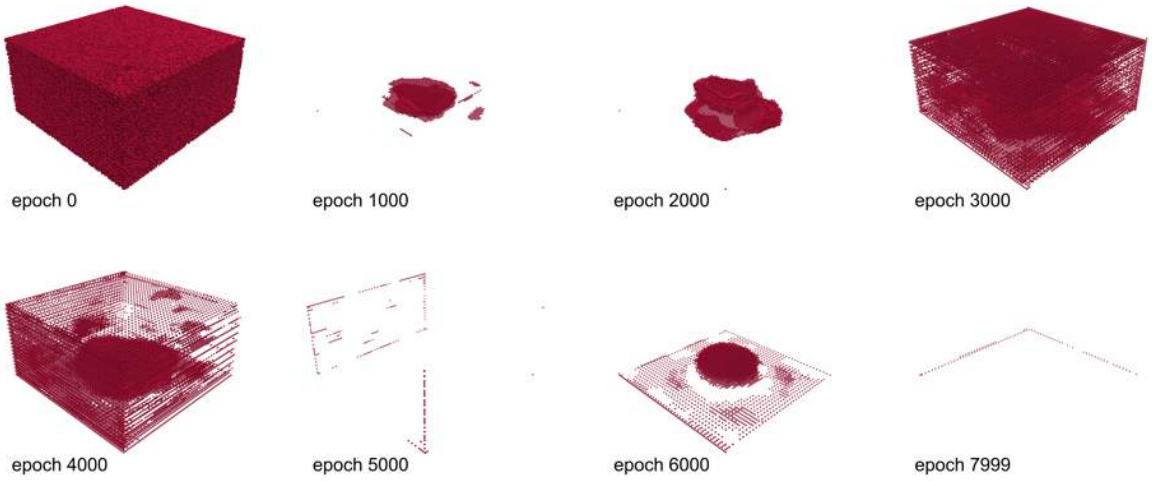
WGAN Architecture 14J



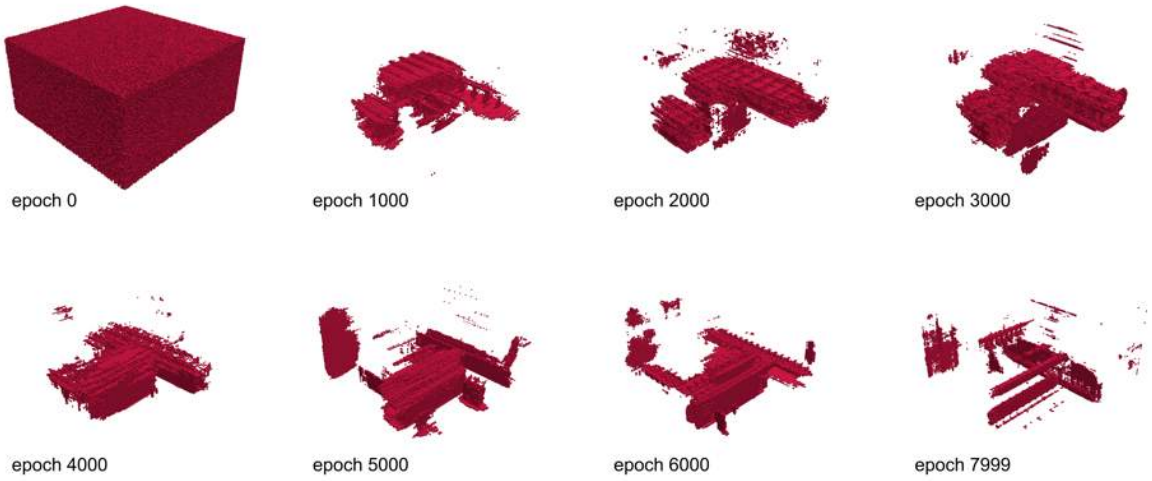
WGAN Architecture 14R



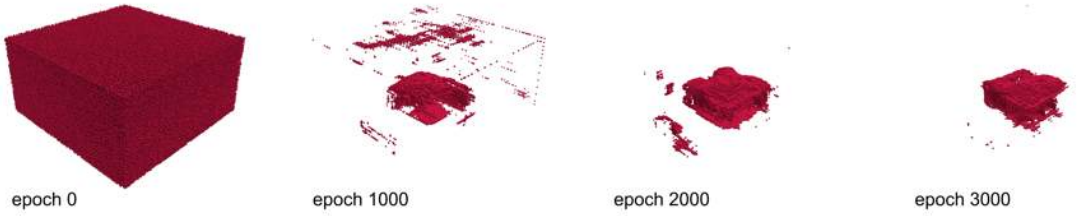
WGAN Architecture 15G



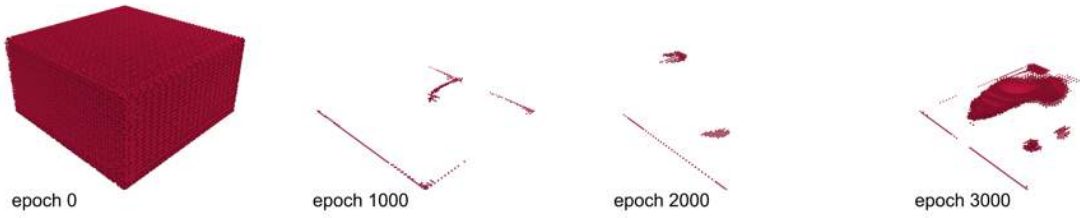
WGAN Architecture 15J

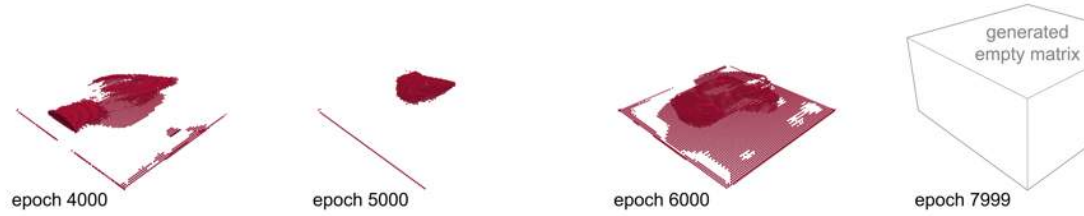


WGAN Architecture 15R

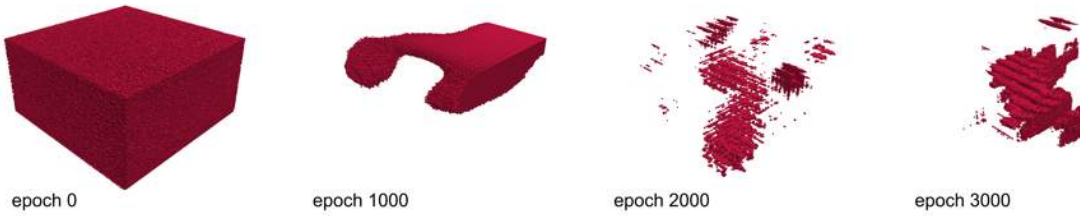


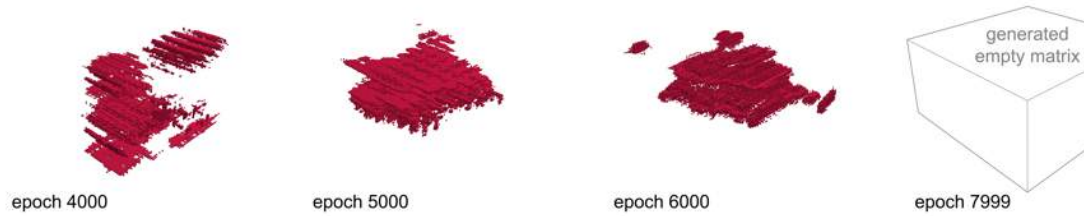
WGAN Architecture 16G



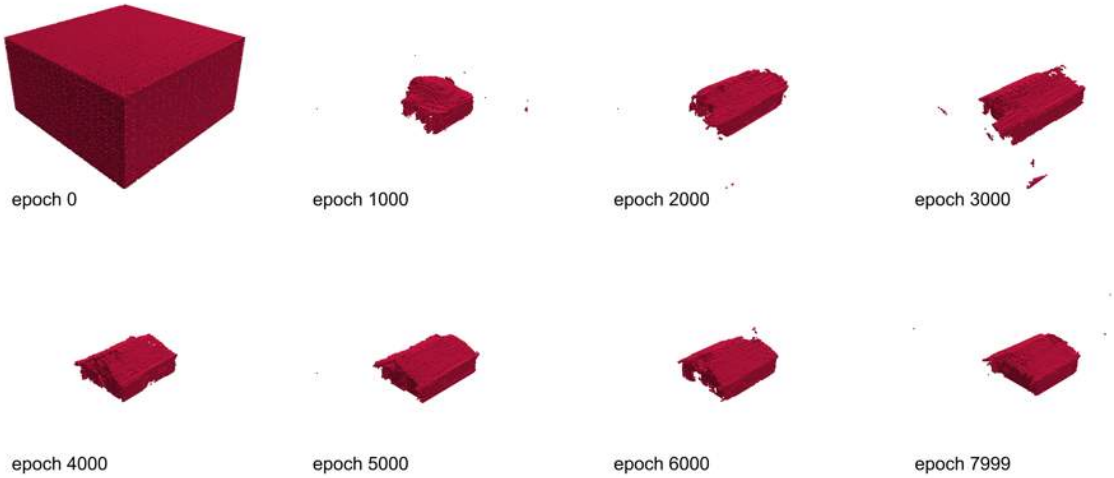


WGAN Architecture 16J

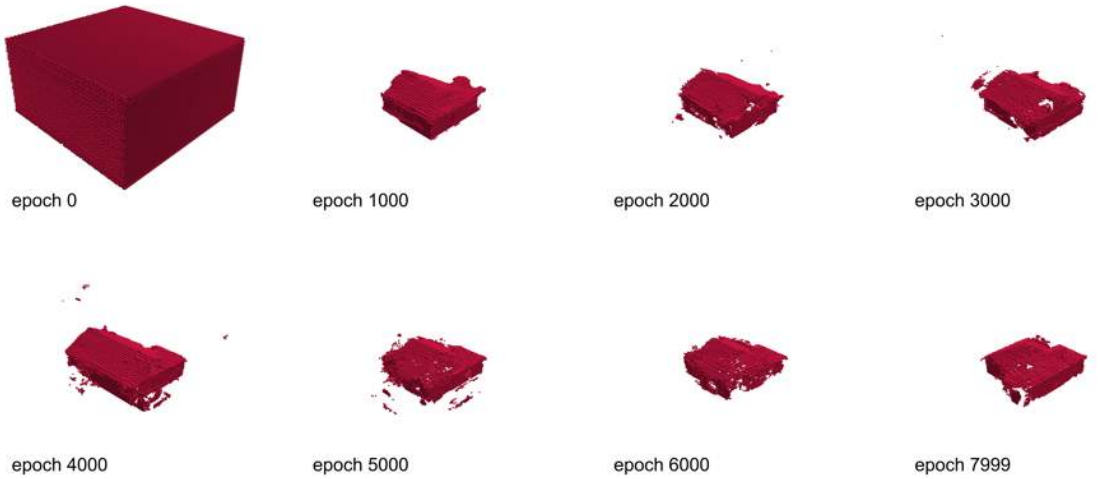




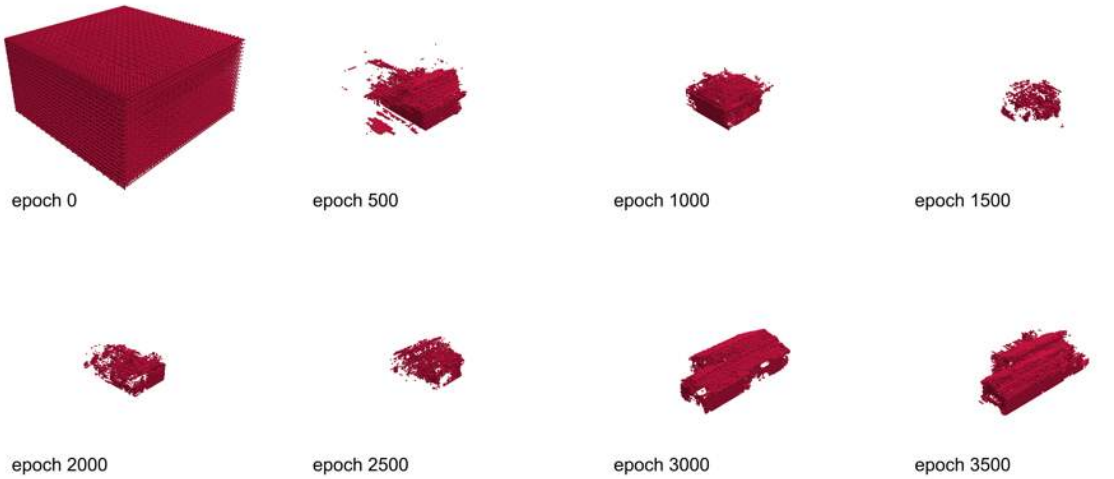
WGAN Architecture 16R

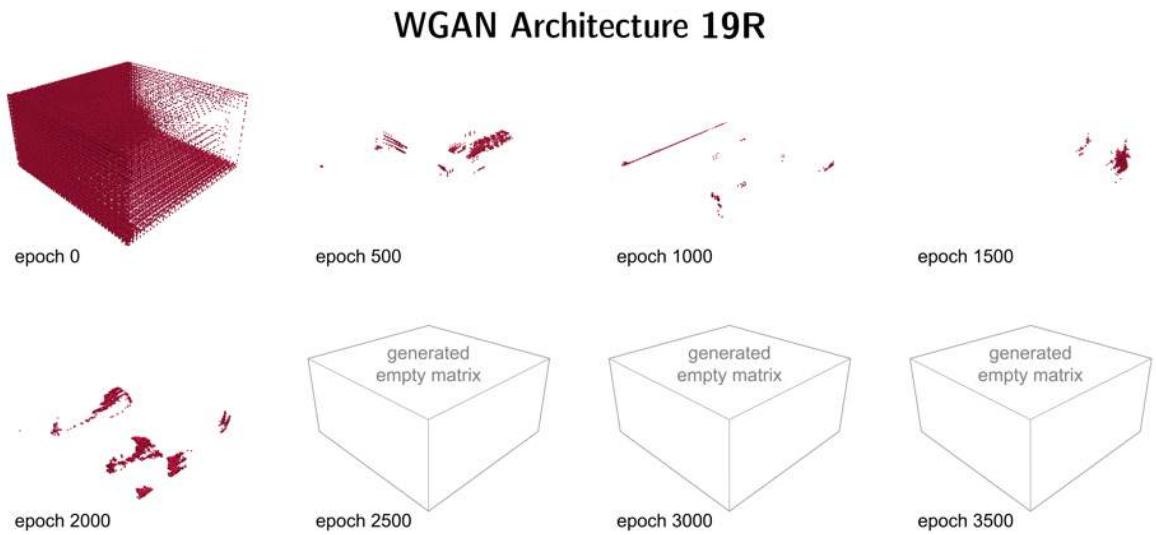


WGAN Architecture 17R



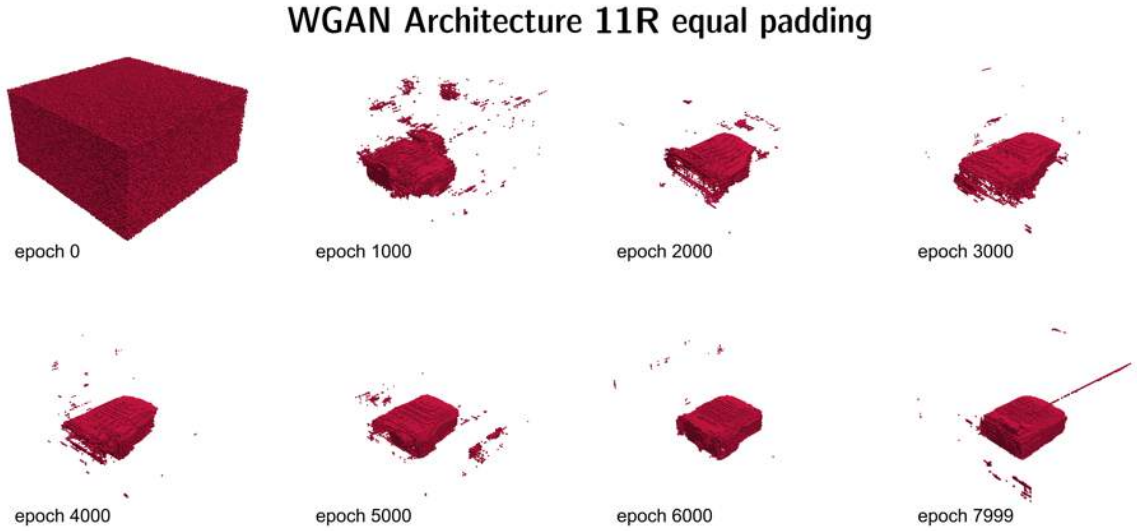
WGAN Architecture 18R



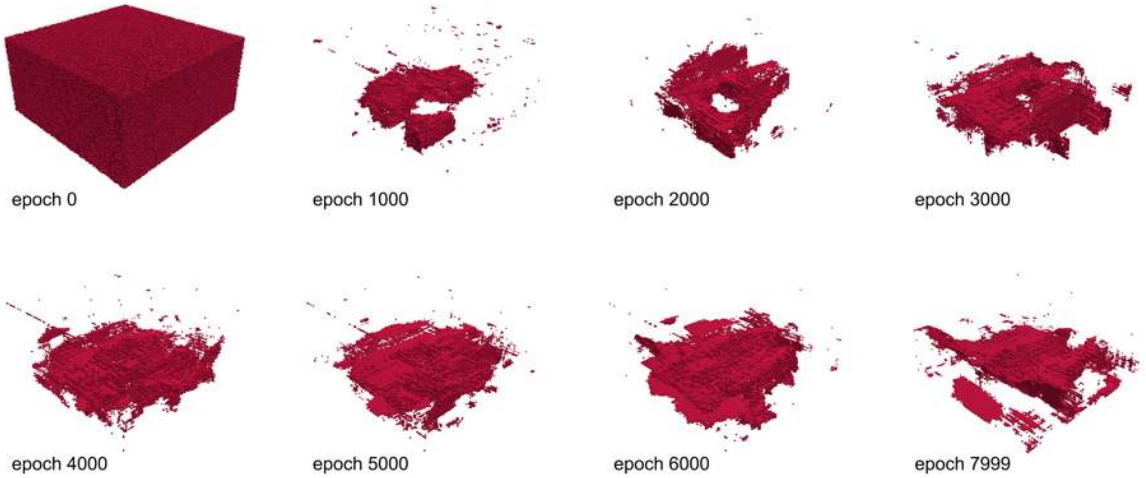


B.5. Padding

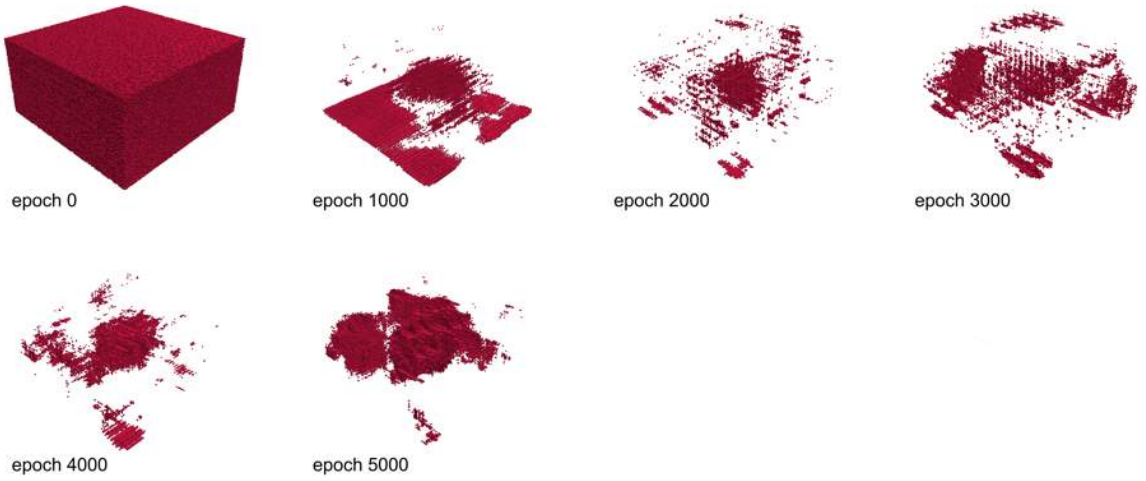
These experiments were run with select architectures indicated in Table 9.2 and a training data set size of 100 models described in Section A.2.



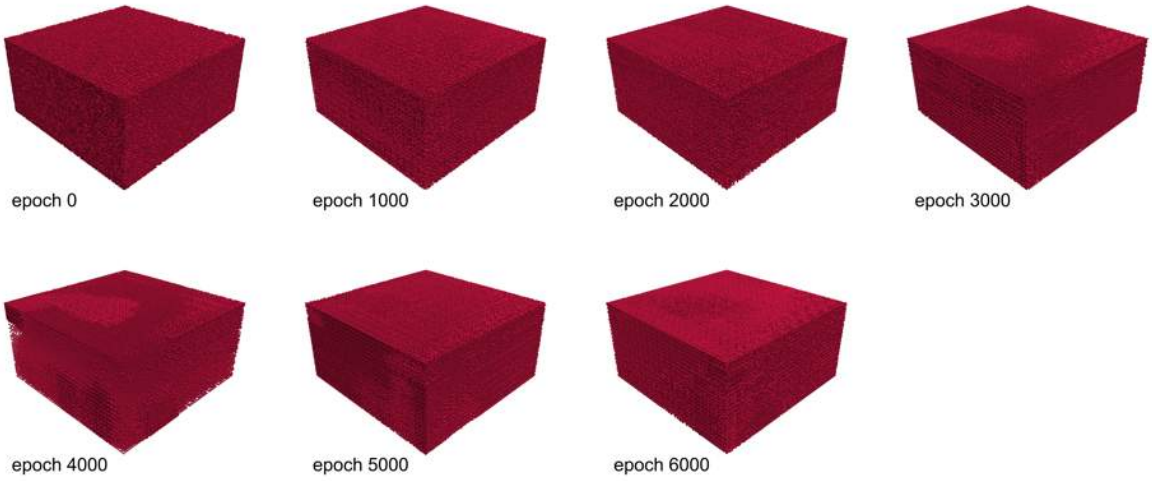
WGAN Architecture **11R** equal reflect padding



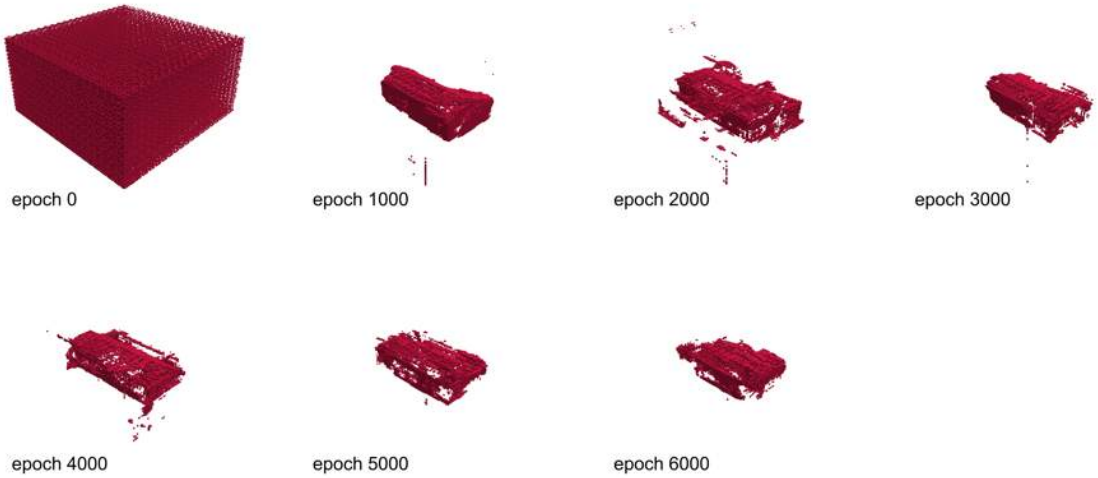
WGAN Architecture **15J** equal padding



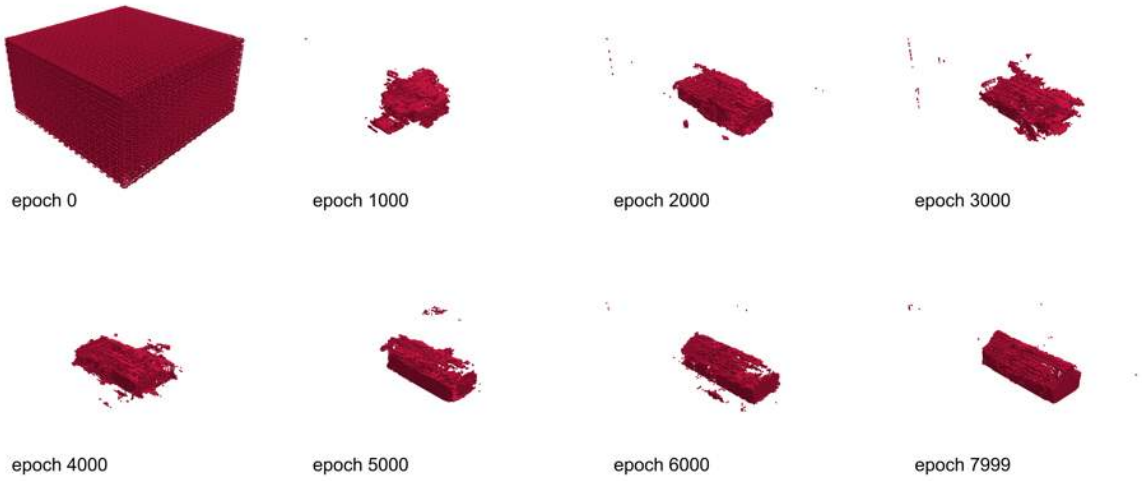
WGAN Architecture **15J** equal reflect padding



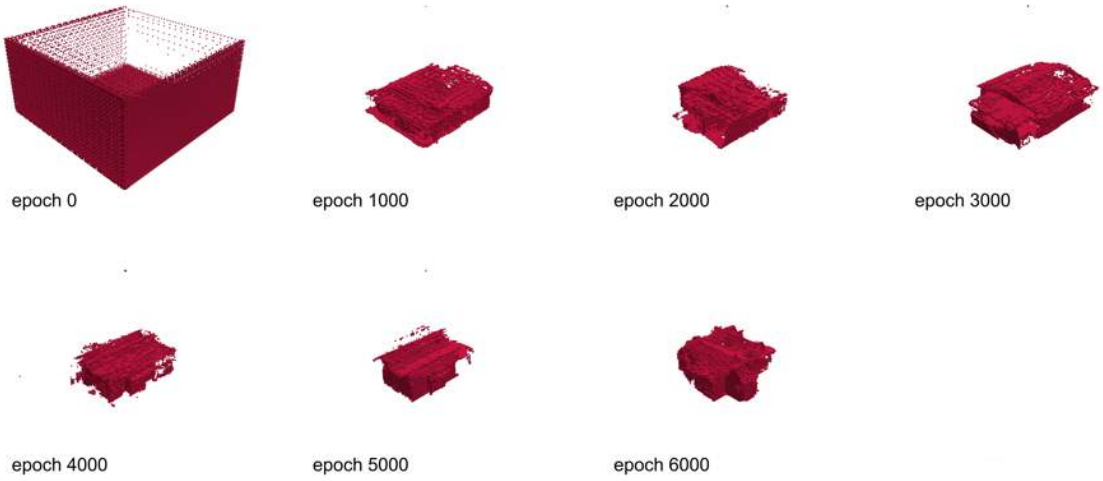
WGAN Architecture **16R** equal padding

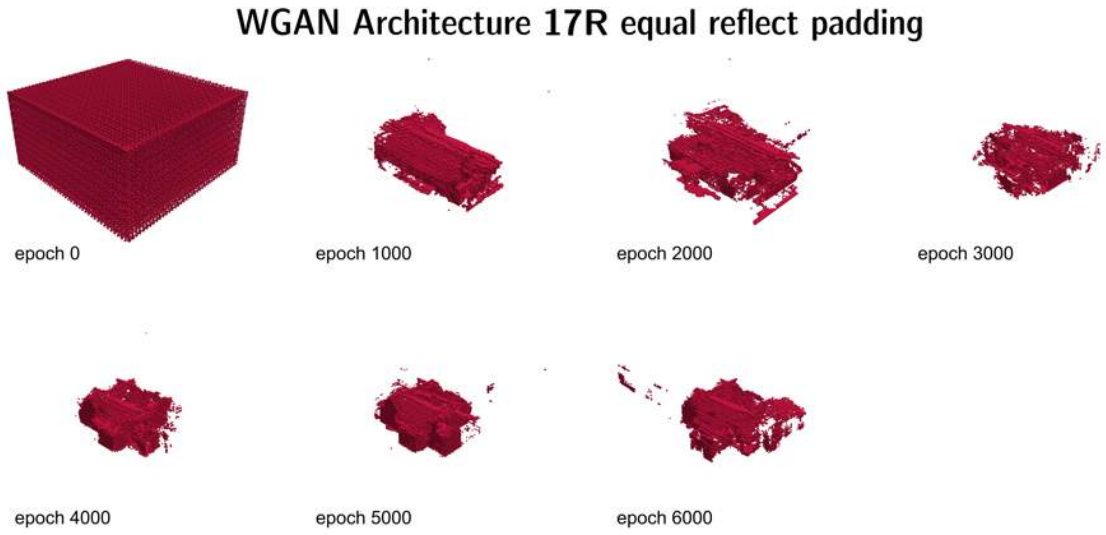


WGAN Architecture **16R** equal reflect padding



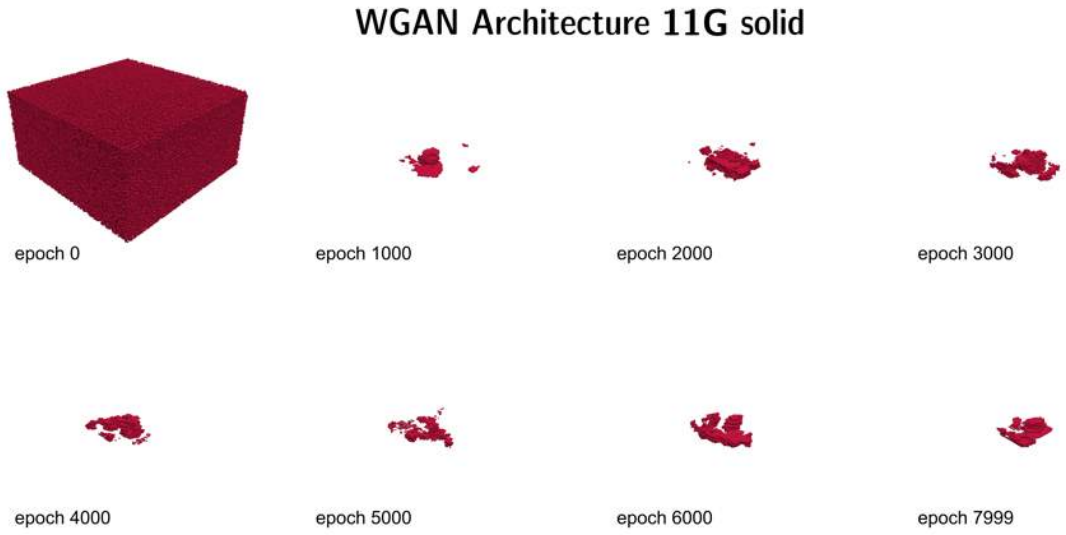
WGAN Architecture **17R** equal padding



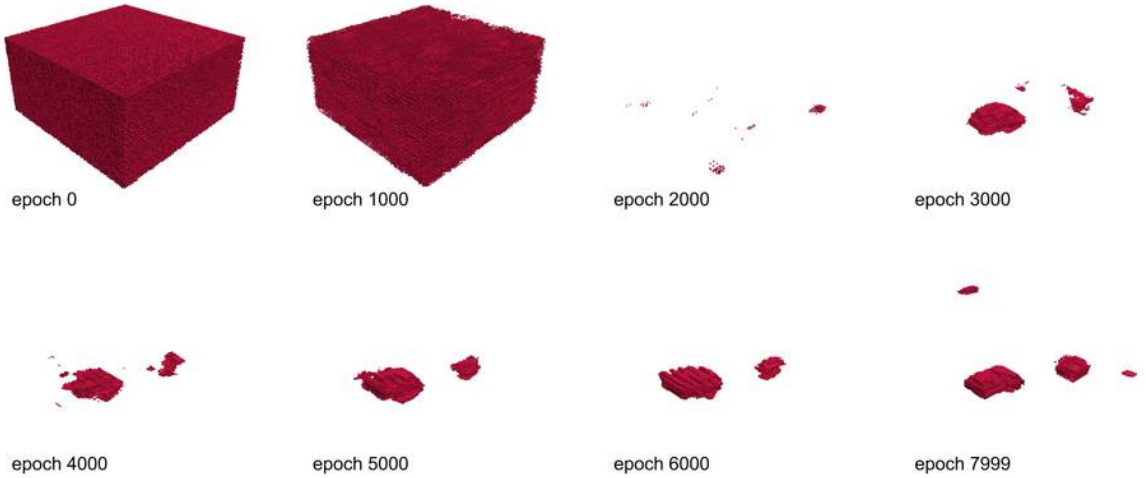


B.6. Solid Model Input

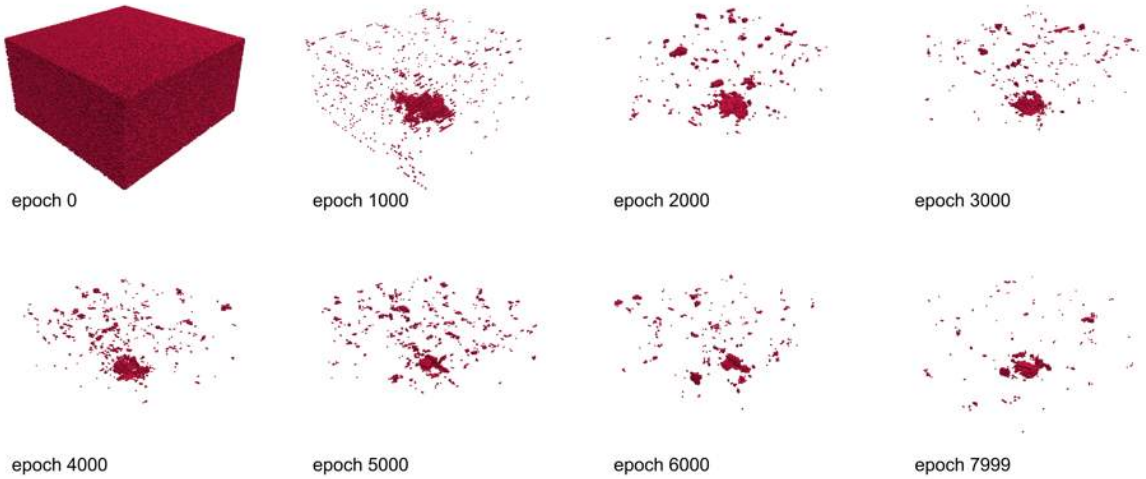
These experiments were run with select architectures indicated in Table 10.1 and a training data set size of 100 models described in Appendix A.2.



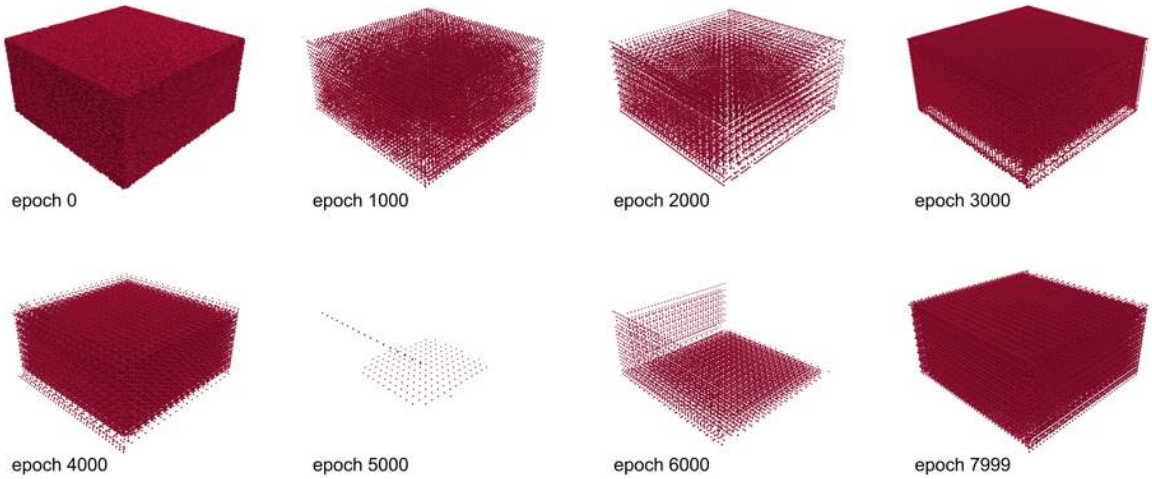
WGAN Architecture **11N** solid

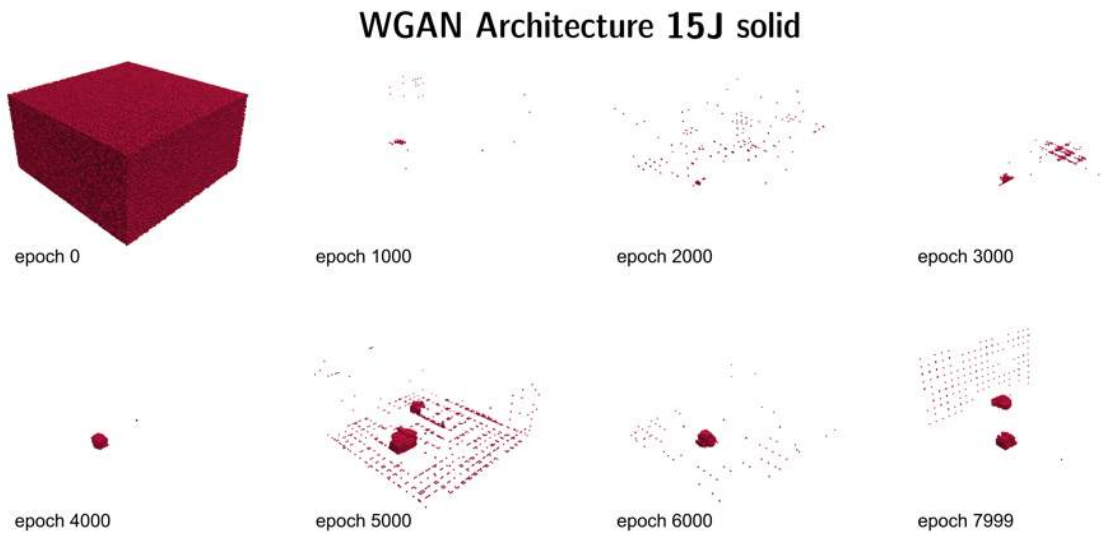


WGAN Architecture **11R** solid



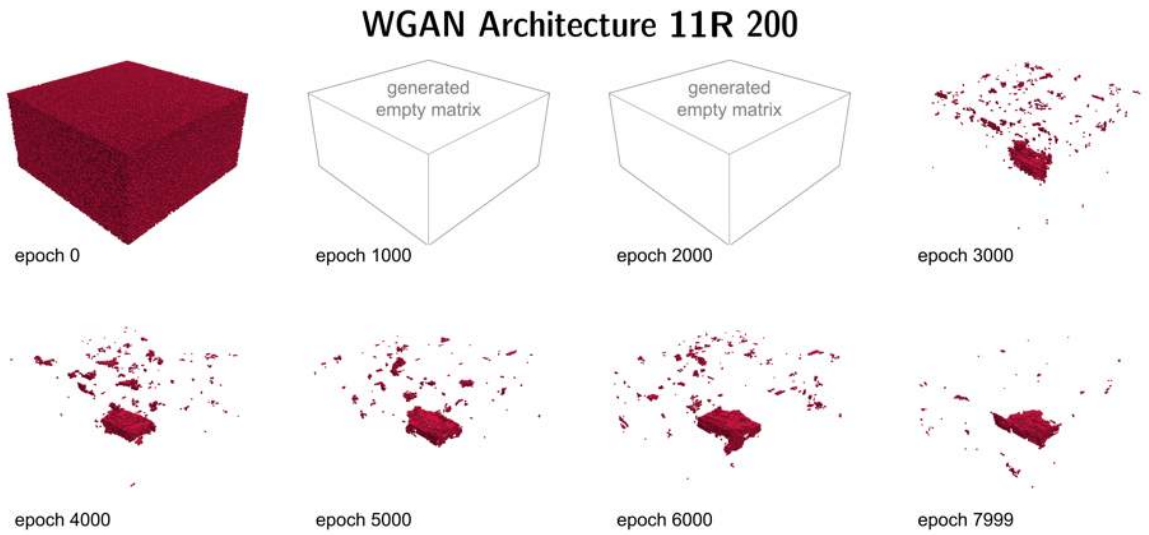
WGAN Architecture **11S** solid



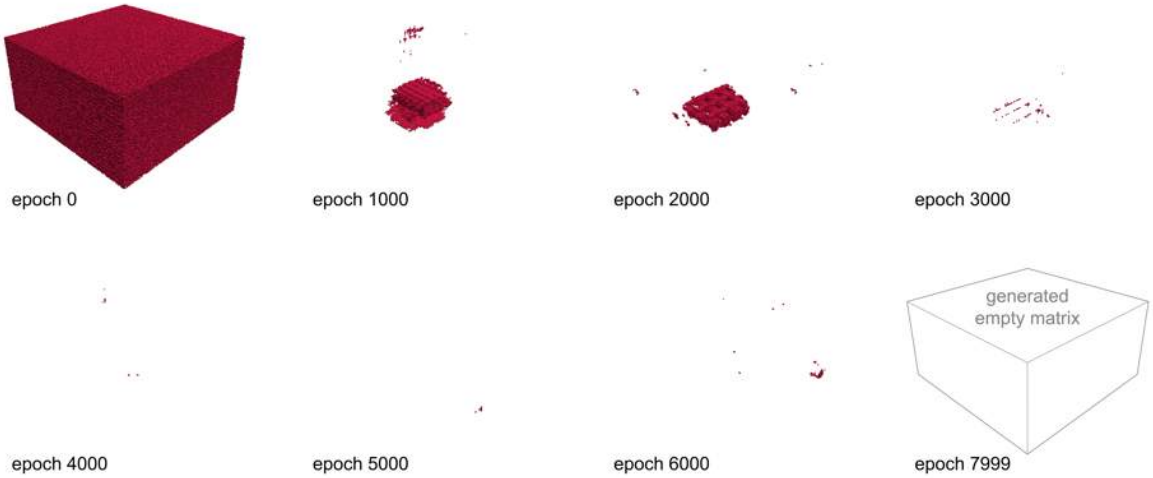


B.7. 200 Model Training Data Set

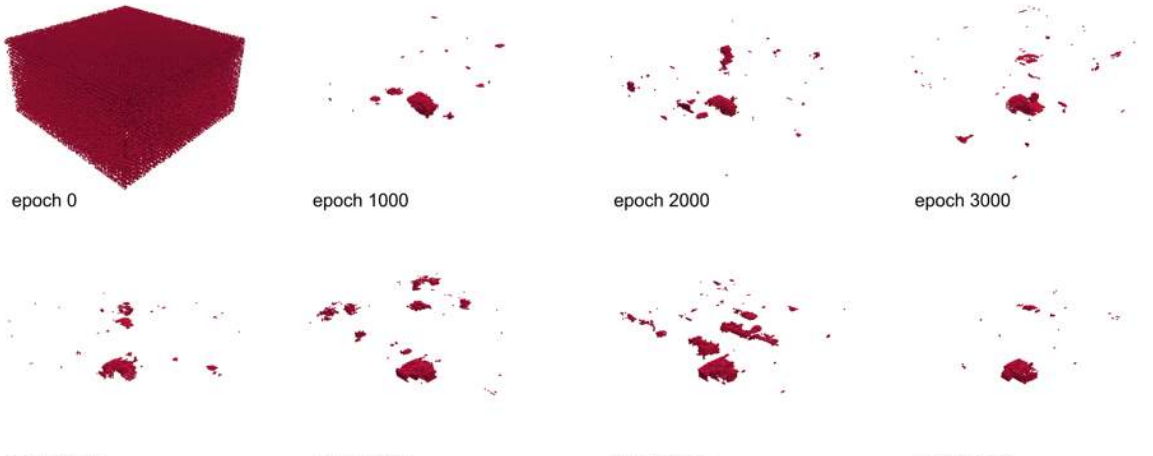
These experiments were run with select successful architectures indicated in Table 10.2 and a training data set size of 200 models described in Appendix A.3.



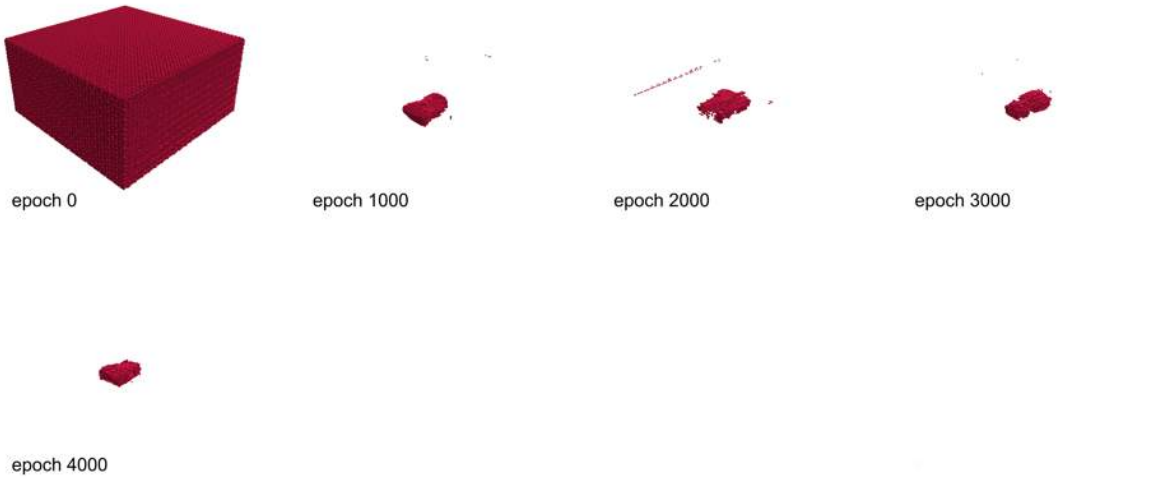
WGAN Architecture 15J 200



WGAN Architecture 15R 200

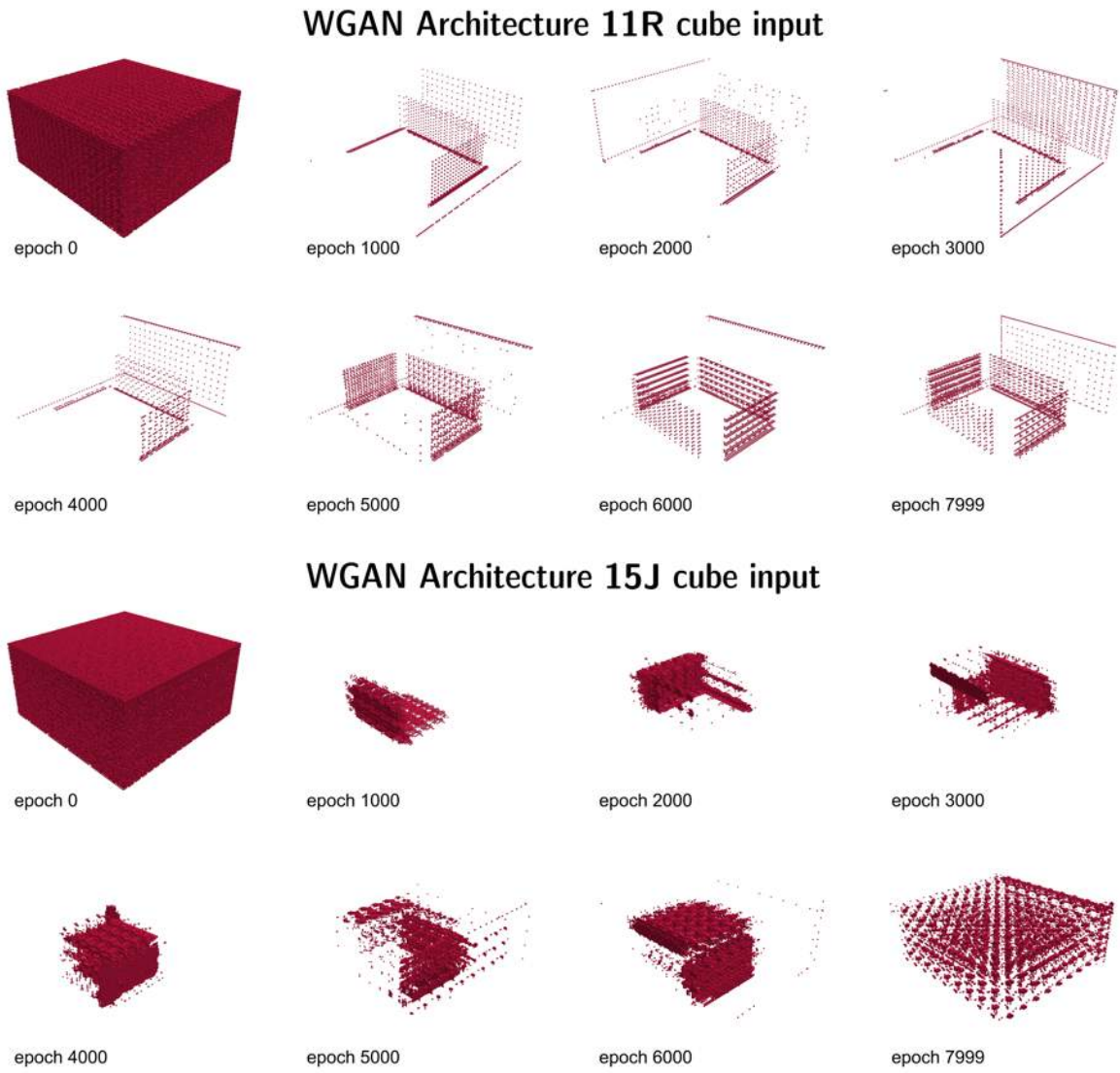


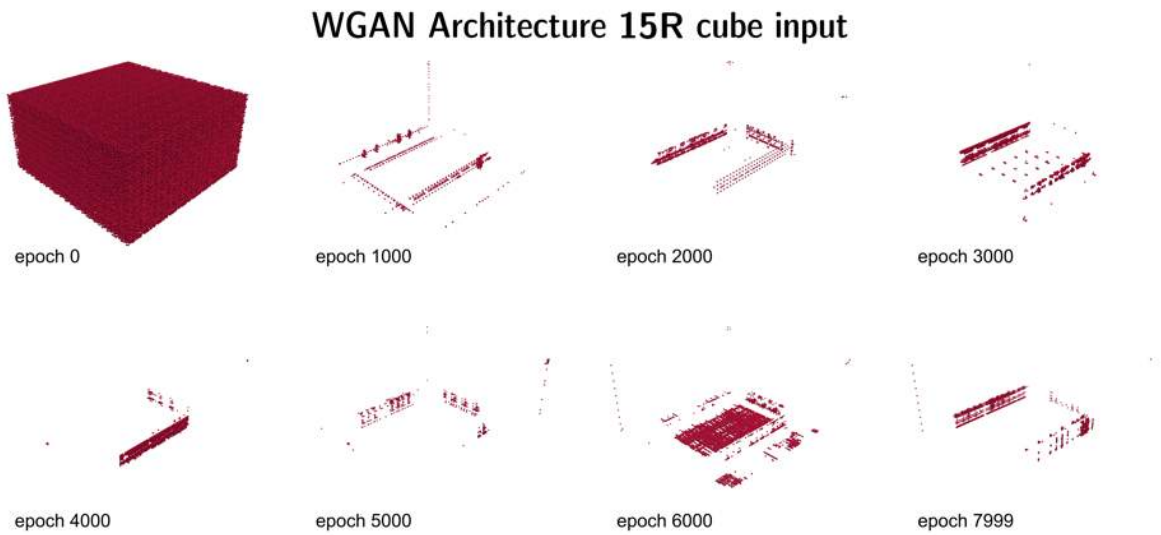
WGAN Architecture 16R 200



B.8. Rectangular Prism Input

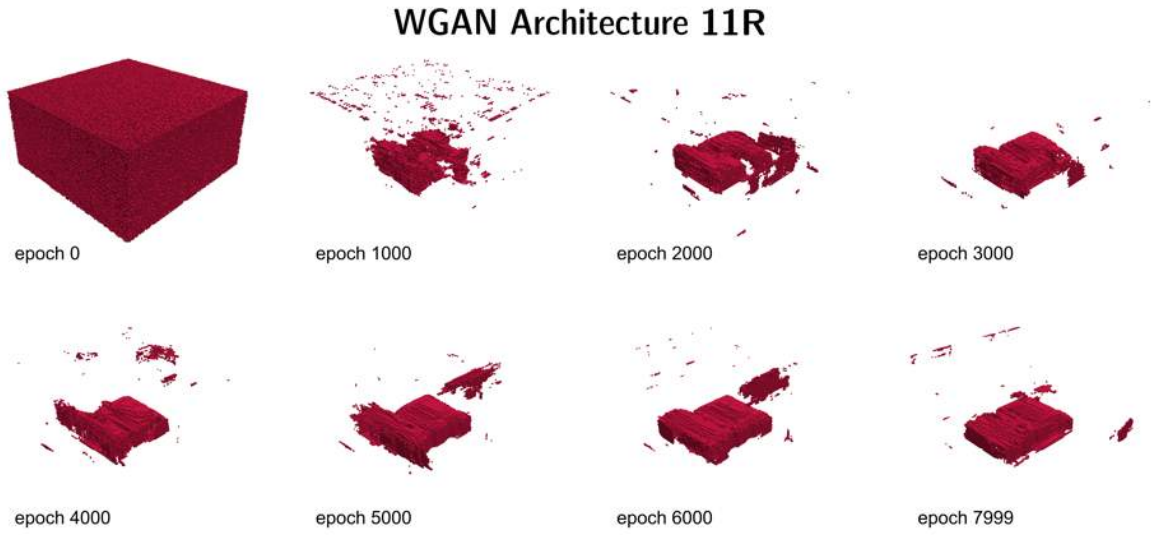
These experiments were run with select architectures indicated in Table 10.3 and a training data set size of 100 models described in Appendix A.2.



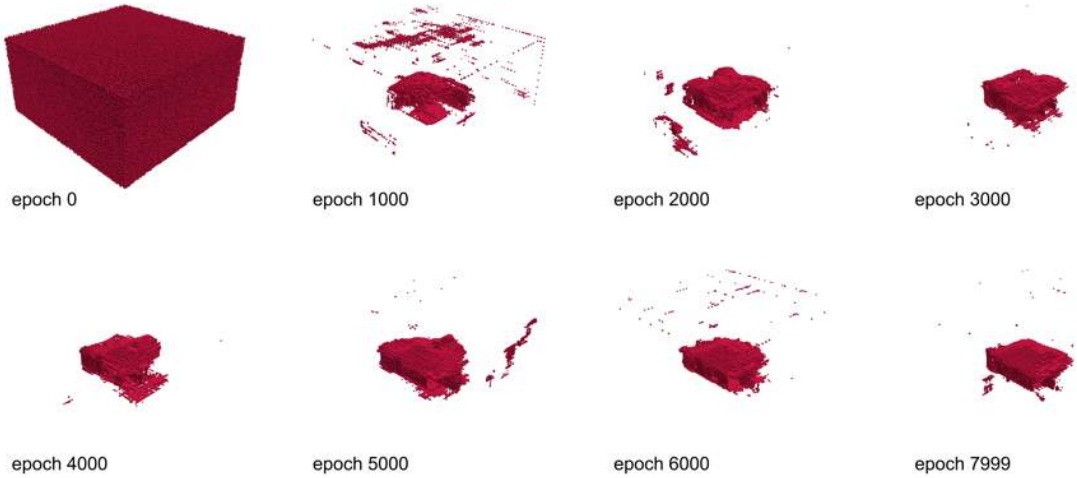


B.9. Best Performing Architecture Results

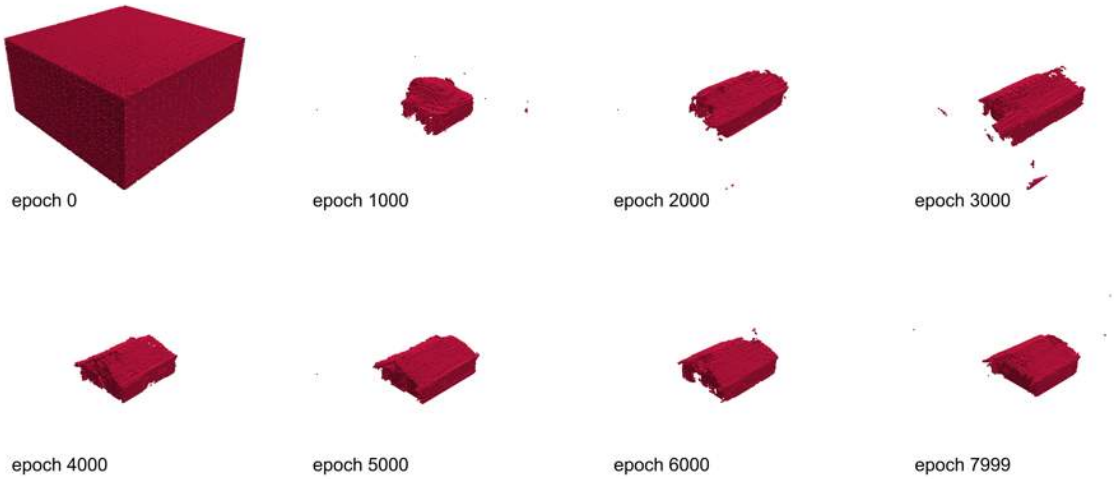
After various experiments were conducted, there were a number of architectures that performed the best. These are the following.



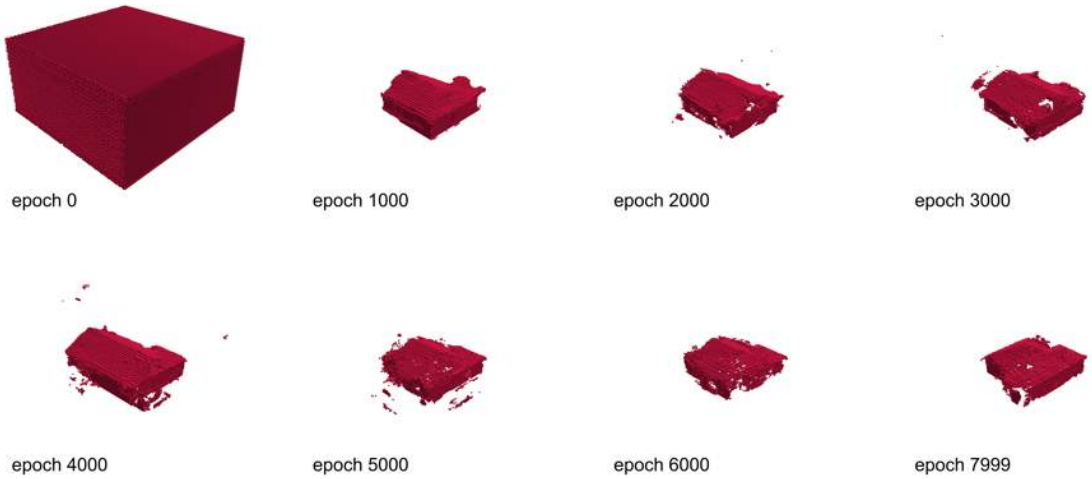
WGAN Architecture 15R

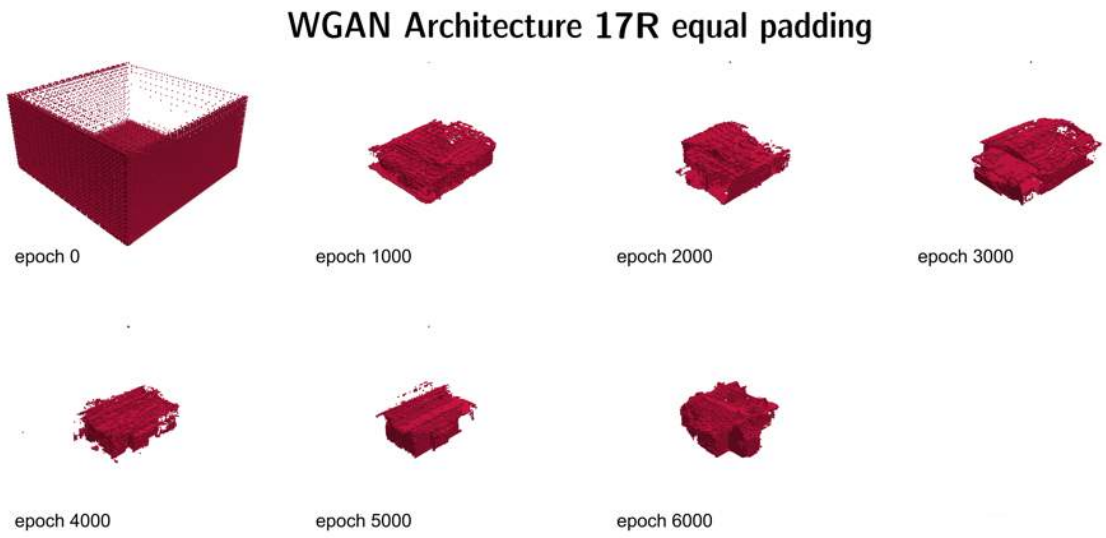


WGAN Architecture 16R



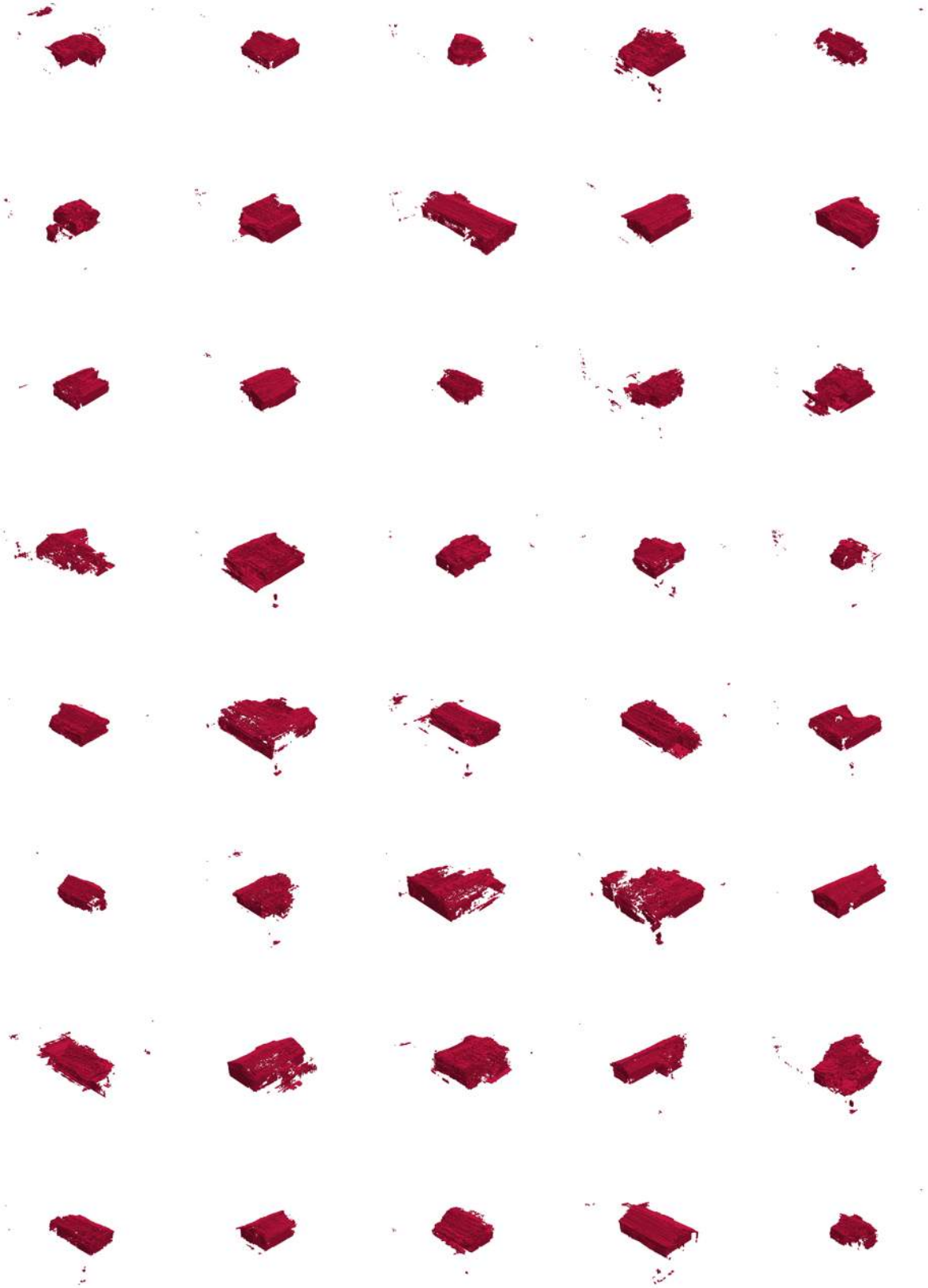
WGAN Architecture 17R

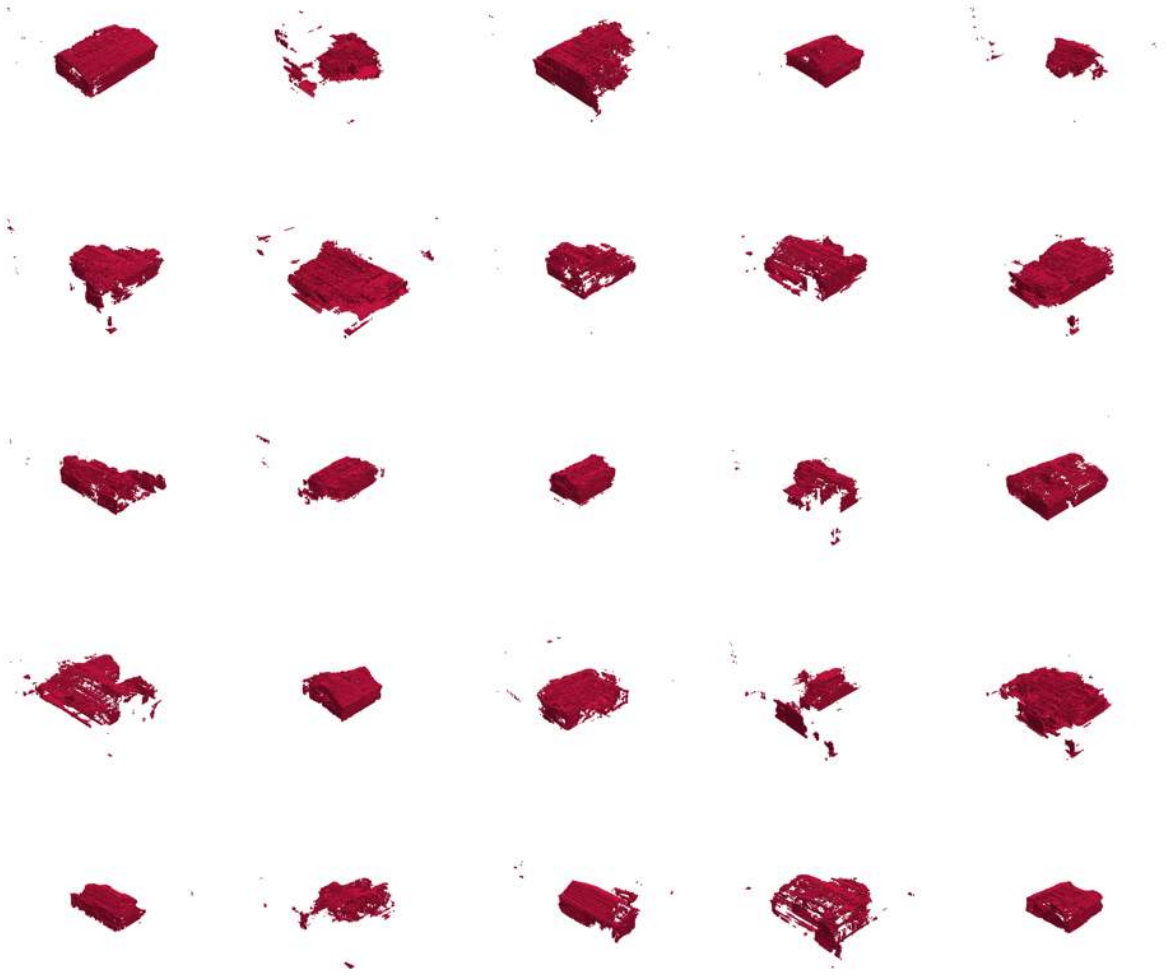


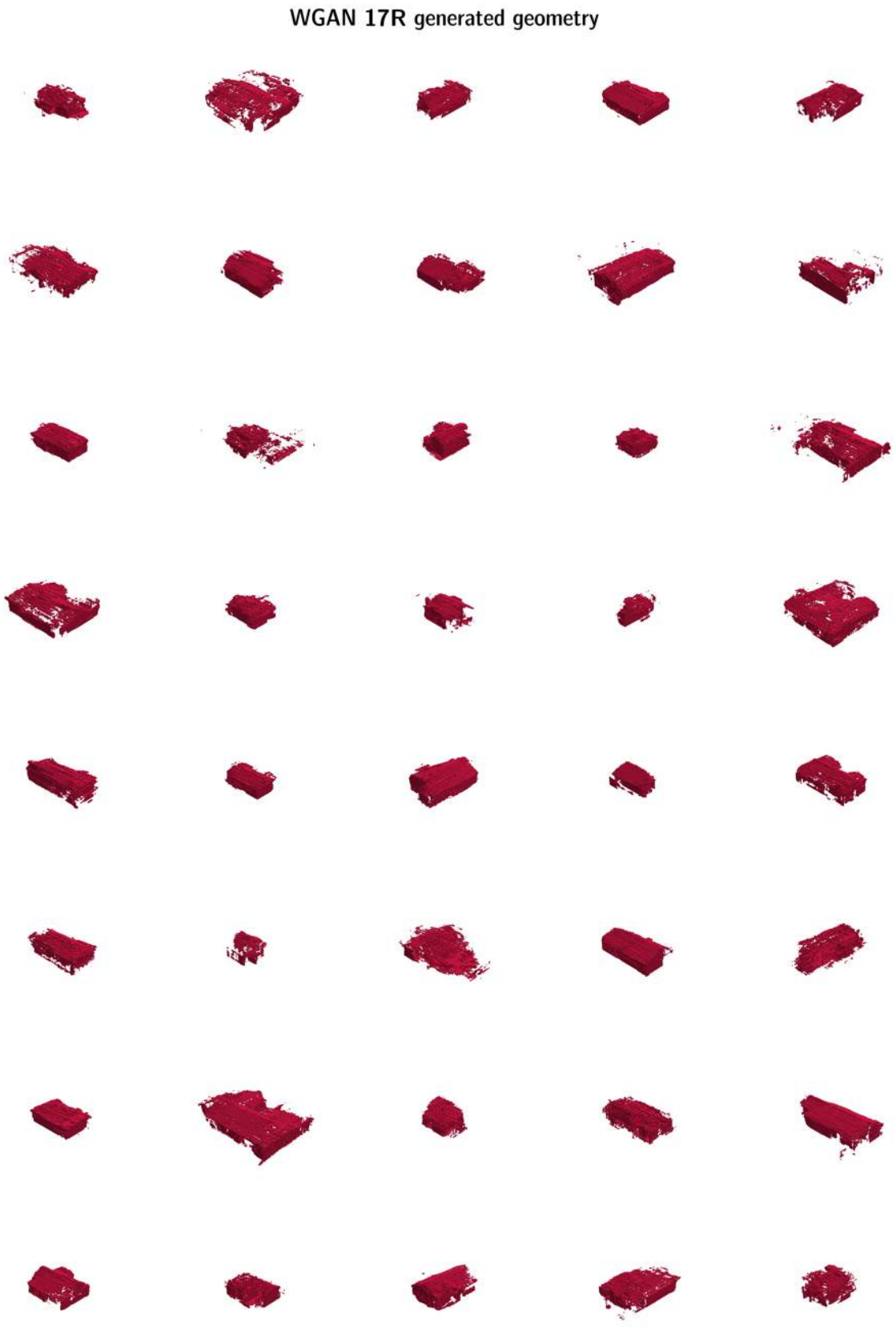


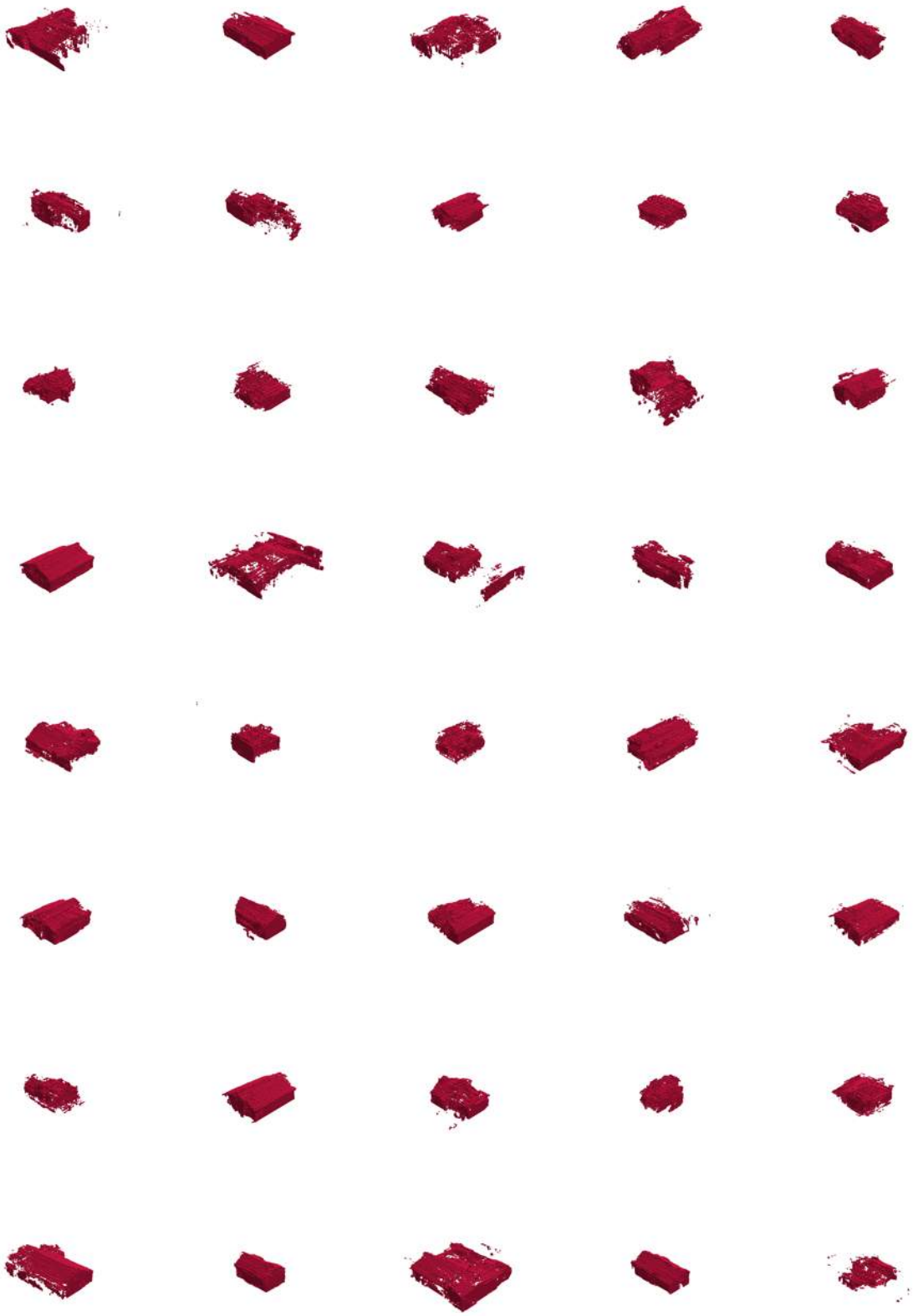
After identifying the best performing architecture, the top three (**16R**, **17R**, and **17R** with equal padding) were used to generate 100 images each. These 100 images are shown here.

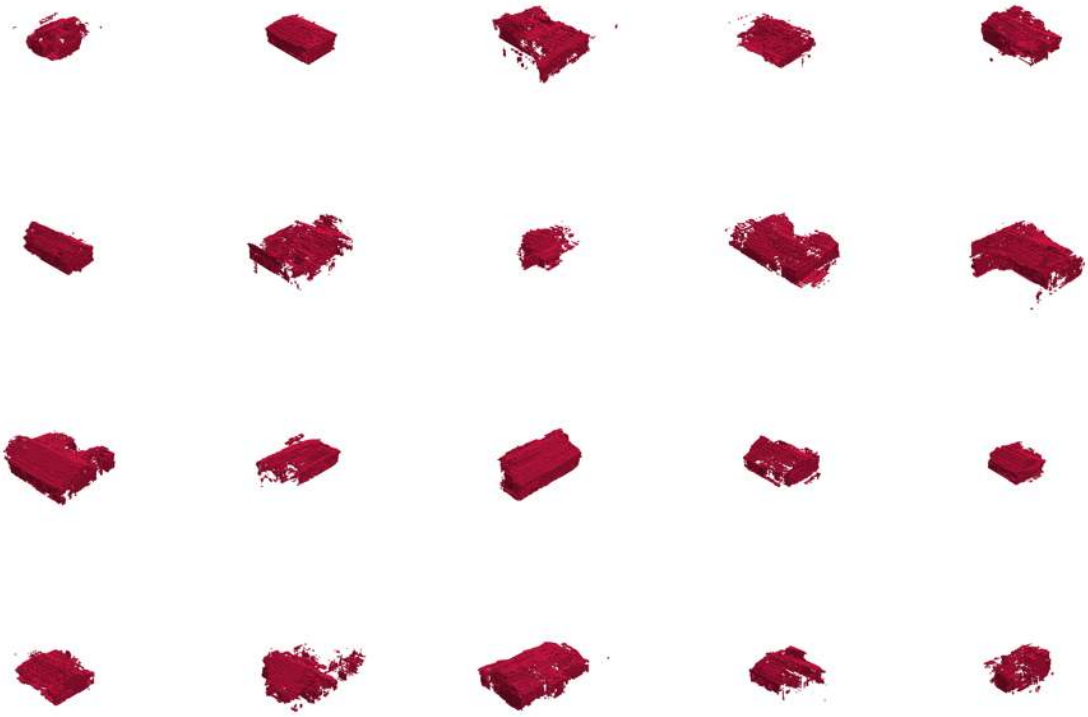

WGAN 16R generated geometry

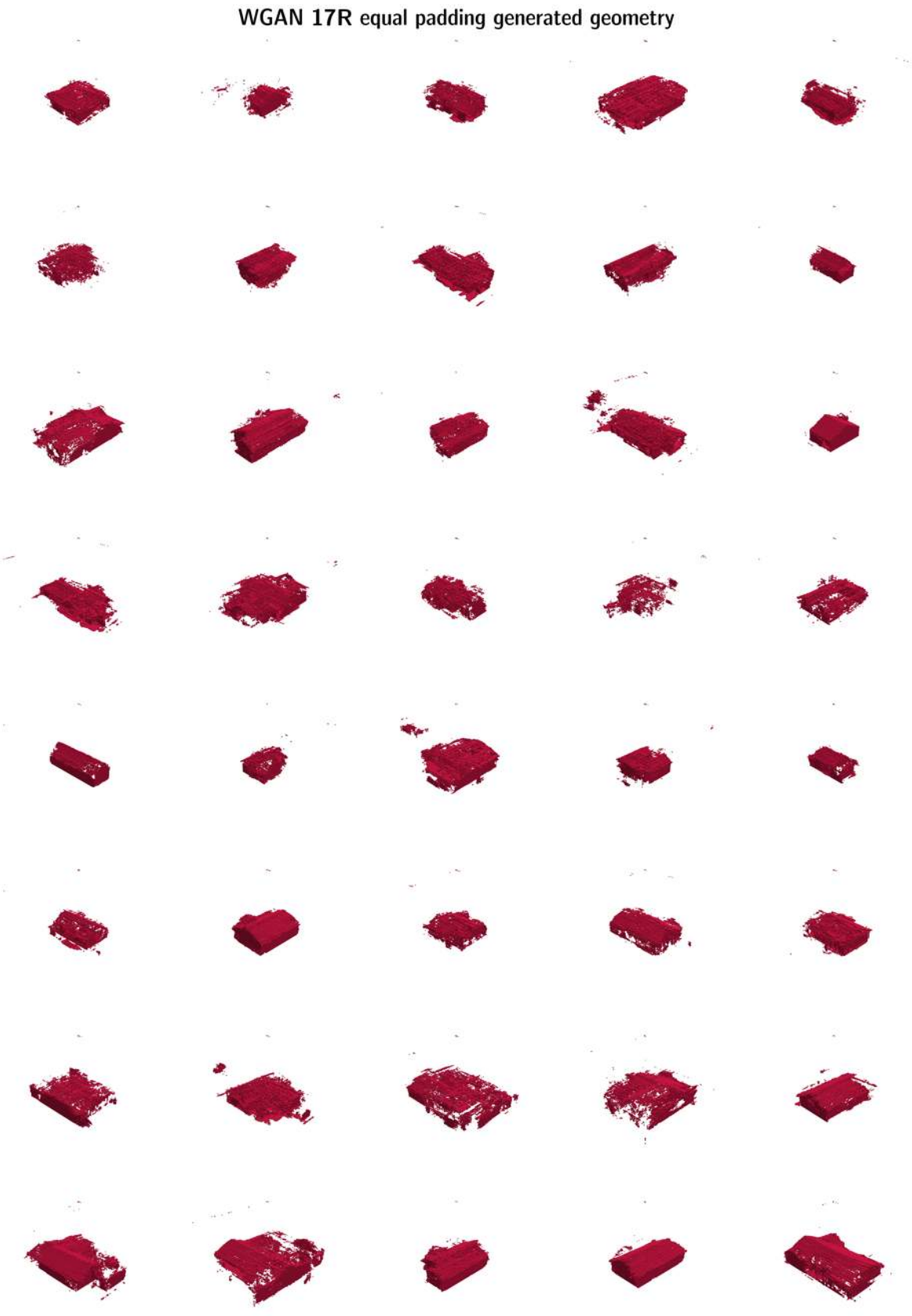


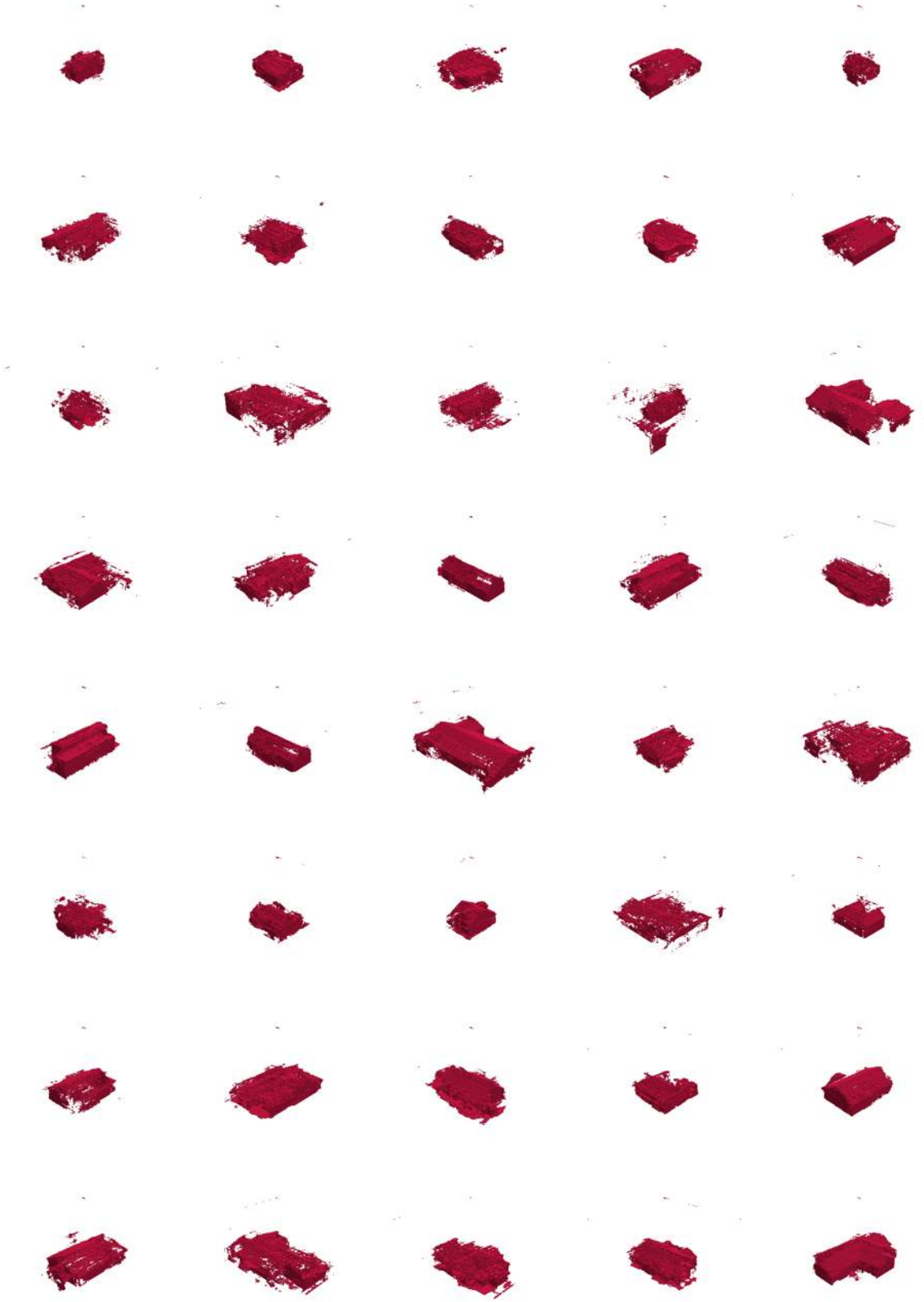


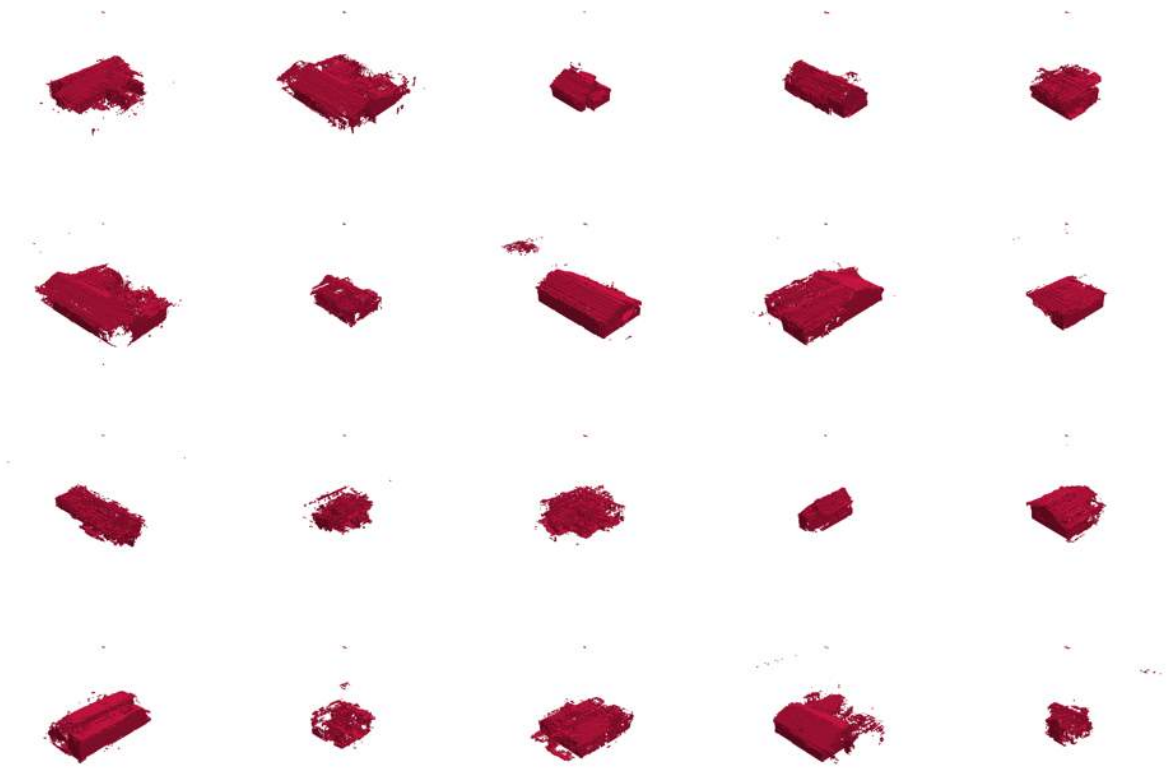

WGAN 17R generated geometry



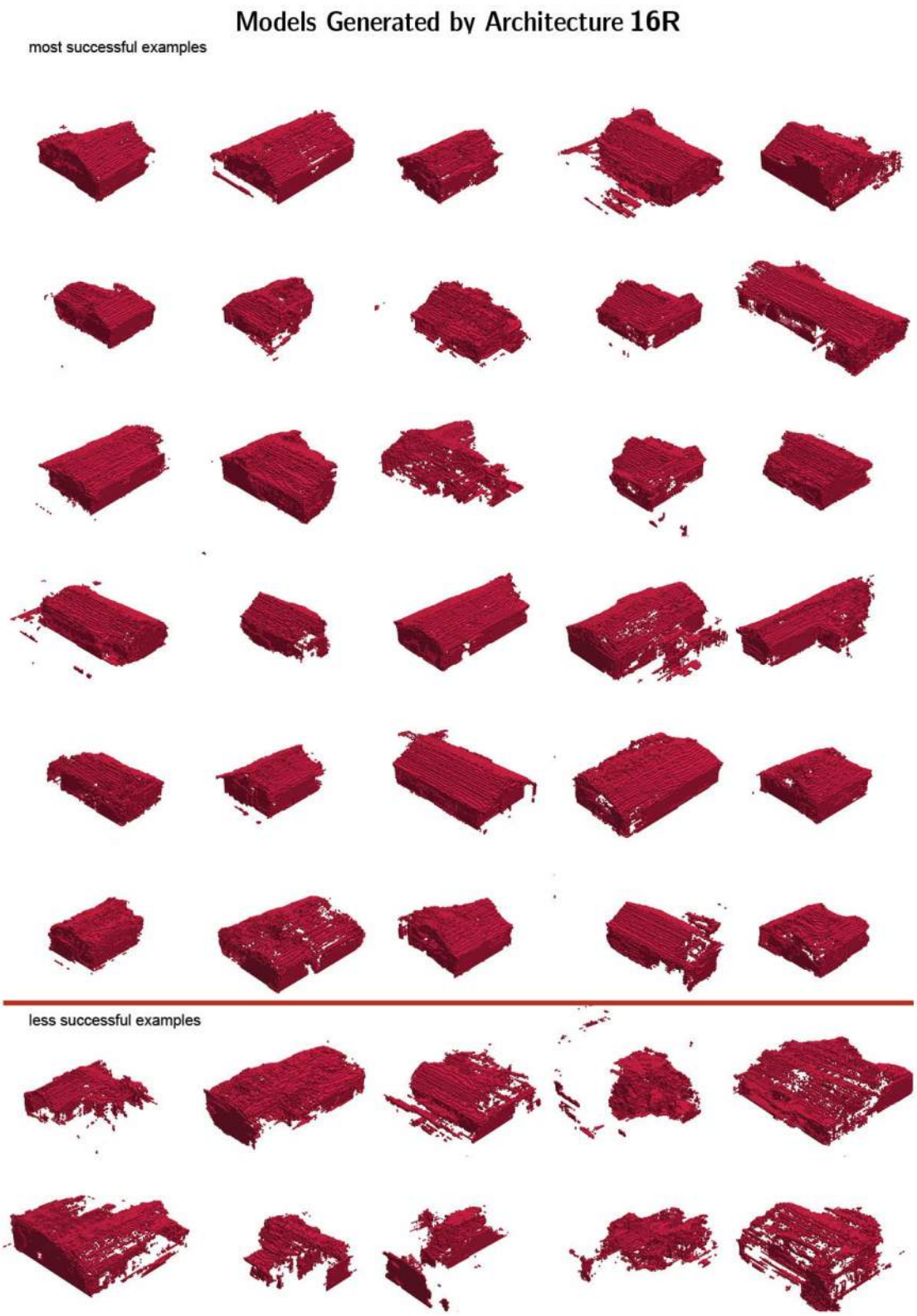


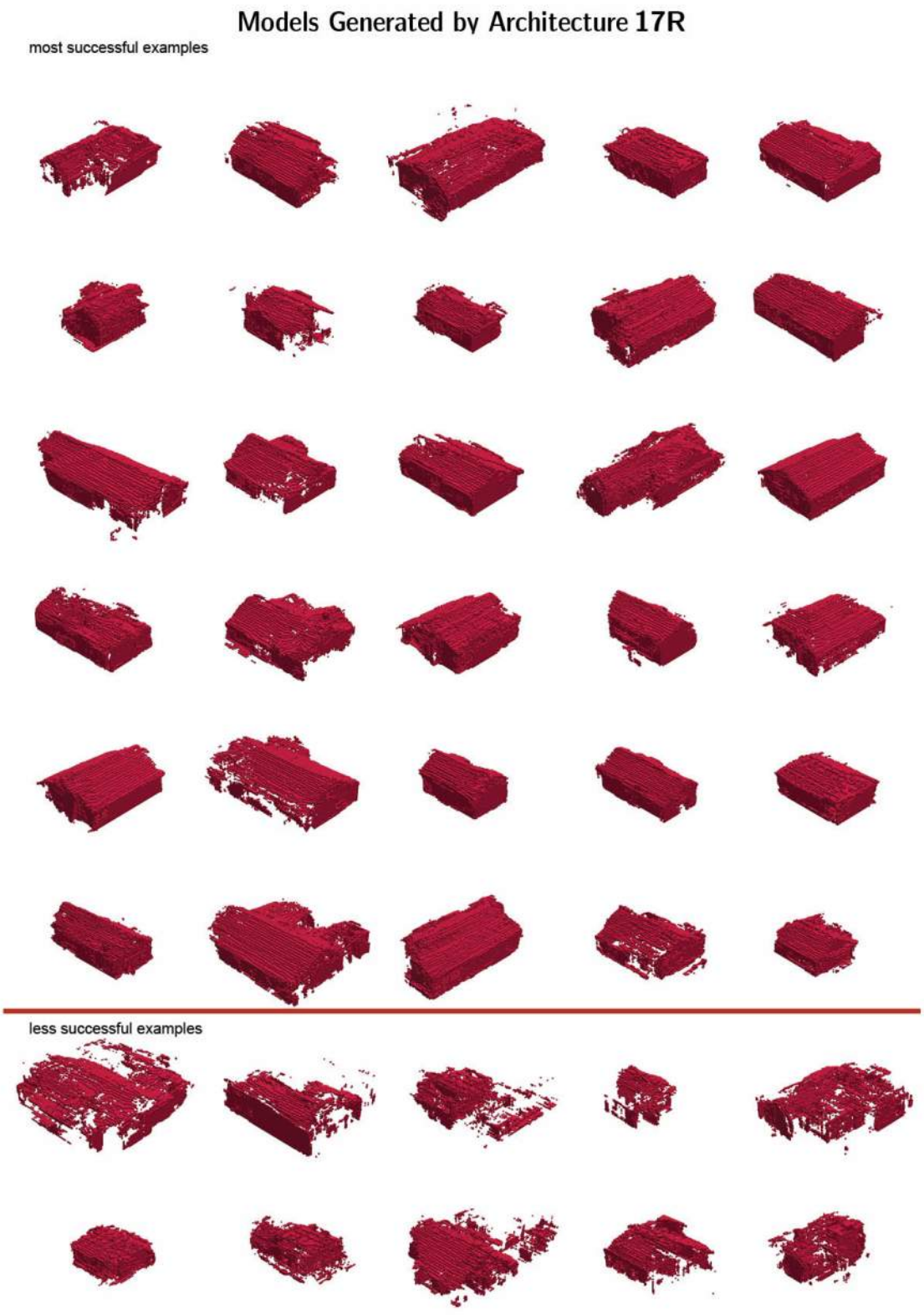

WGAN 17R equal padding generated geometry

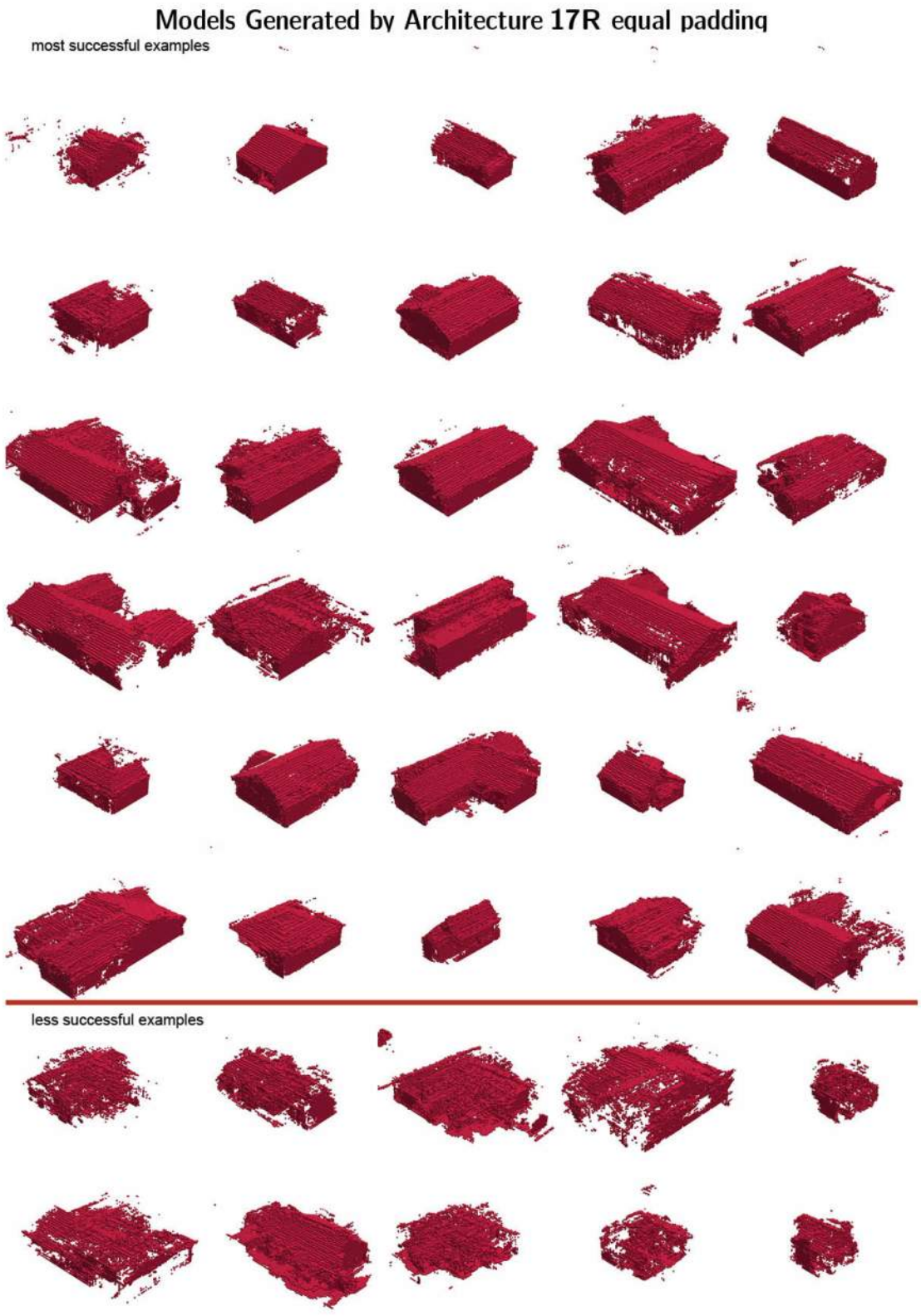




From the complete set of 100 images, select images that represented good results and select images that represented not as good results were enlarged to make it easier to see some of the detail. These images are shown in the following pages.


Models Generated by Architecture 16R
most successful examples
less successful examples


Models Generated by Architecture 17R
most successful examples
less successful examples


Models Generated by Architecture 17R equal padding
most successful examples
less successful examples

B.10. Jupyter Notebook Examples

The code for architecture **16R** and **17R** along with code for a selection of the rest of the experiments is located on the public GitHub repository. The public GitHub repository for this project can be found at https://github.com/lm2-me/3DWGANHouses.

It contains three Jupyter notebooks that aim to demonstrate how some of the code works. Images of selections from two of the notebooks are shown here for reference.

Point Cloud Preprocessing

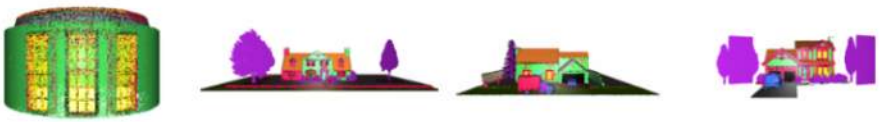
In [1]:
```
#imports and initialize variables
import utilities.preprocessing as preprocess
import open3d as o3d
import os
import numpy as np
import matplotlib.pyplot as plt
import PIL
from PIL import Image
from PIL import ImageDraw
from IPython.display import display

file_path = 'sample_files/pointclouds/original'
save_location = 'sample_files/pointclouds/clean'

image_path_orig = 'sample_files/pointclouds/images_original'
image_path_clean = 'sample_files/pointclouds/images_clean'
```

```
Jupyter environment detected. Enabling Open3D WebVisualizer.
[Open3D INFO] WebRTC GUI backend enabled.
[Open3D INFO] WebRTCWindowSystem: HTTP handshake server disabled.
```

In [2]:
```
#Load pointclouds
all_files = [f for f in os.listdir(file_path) if os.path.isfile(os.path.join(file_path,

for file in all_files:
    if file.endswith('.ply'):
        _, name, pcd, _, _ = preprocess.load_point_cloud_file(file_path, file)
        name = file.replace('.ply','')

        vis = o3d.visualization.Visualizer()
        vis.create_window(window_name=name, width=1000, height=1000)
        vis.add_geometry(pcd)

        image = vis.capture_screen_float_buffer(True)
        path = image_path_orig + "/"
        filename = path + name + ".png"
        print(filename)

        plt.imsave(filename, np.asarray(image), format = 'png', dpi = 150)
        plt.clf()

        vis.destroy_window()
```

```
sample_files/pointclouds/images_original/COMMERCIALhouse_mesh2682.png
sample_files/pointclouds/images_original/RESIDENTIALhouse_mesh8157.png
sample_files/pointclouds/images_original/RESIDENTIALhouse_mesh8494.png
sample_files/pointclouds/images_original/RESIDENTIALhouse_mesh8596.png

<Figure size 640x480 with 0 Axes>
```

In [3]:
```
#show saved images of point clouds
images = []
all_files = [f for f in os.listdir(image_path_orig) if os.path.isfile(os.path.join(image

images = [Image.open(image_path_orig + '/' + x) for x in all_files]

new_im = Image.new('RGB', (1000, 200), (255, 255, 255))

for i, im in enumerate(images):
    width, height = im.size
    target_width = 250 #px
    ratio = width / target_width
    new_height = int(height // ratio)

    im_resized = im.resize((target_width, new_height))

    new_im.paste(im_resized, (im_resized.size[0] * i, 0))

display(new_im)
```



In [4]:
```
#clean and scale points cloud
preprocess.clean_point_clouds(file_path, save_location)

#save images of cleaned pointclouds
all_files = [f for f in os.listdir(save_location) if os.path.isfile(os.path.join(save_lo

for file in all_files:
    if file.endswith('.ply'):
        _, name, pcd, _, _ = preprocess.load_point_cloud_file(save_location, file)
        name = file.replace('.ply','')

        vis = o3d.visualization.Visualizer()
        vis.create_window(window_name=name, width=1000, height=1000)
        vis.add_geometry(pcd)

        image = vis.capture_screen_float_buffer(True)
        path = image_path_clean + "/"
        filename = path + name + ".png"
        print(filename)

        plt.imsave(filename, np.asarray(image), format = 'png', dpi = 150)
        plt.clf()

        vis.destroy_window()
```

```
Cleaned 8 files.
sample_files/pointclouds/images_clean/COMMERCIALhouse_mesh2682.png
sample_files/pointclouds/images_clean/RESIDENTIALhouse_mesh8157.png
sample_files/pointclouds/images_clean/RESIDENTIALhouse_mesh8494.png
sample_files/pointclouds/images_clean/RESIDENTIALhouse_mesh8596.png

<Figure size 640x480 with 0 Axes>
```

In [5]:
```
#show saved images of cleaned point clouds
images = []
all_files = [f for f in os.listdir(image_path_clean) if os.path.isfile(os.path.join(imag

images = [Image.open(image_path_clean + '/' + x) for x in all_files]

new_im_clean = Image.new('RGB', (1000, 200), (255, 255, 255))

for i, im in enumerate(images):
    width, height = im.size
    target_width = 250 #px
    ratio = width / target_width
    new_height = int(height // ratio)

    im_resized = im.resize((target_width, new_height))

    new_im_clean.paste(im_resized, (im_resized.size[0] * i, 0))

display(new_im)
display(new_im_clean)
```

Generated Geometry with Architecture 16R

In [1]:
```
#imports and initialize variabless
import tensorflow as tf
import os
import PIL
from PIL import Image
import math

import wganv16R as gan
import utilities.ganutilities as util

save_location = 'generated/images'
generated_matrices = 'generated/generated_matrices'
```

In [2]:
```
#load network weights
generator = gan.make_generator_model()
discriminator = gan.make_discriminator_model()

generator_optimizer = tf.keras.optimizers.legacy.RMSprop(learning_rate=0.00005)
discriminator_optimizer = tf.keras.optimizers.legacy.RMSprop(learning_rate=0.00005)

checkpoint_dir = 'sample_files/training_checkpoints/16R/ckpt-112'
checkpoint = tf.train.Checkpoint(generator_optimizer=generator_optimizer,
                                 discriminator_optimizer=discriminator_optimizer,
                                 generator=generator,
                                 discriminator=discriminator)

manager = tf.train.CheckpointManager(checkpoint, checkpoint_dir, max_to_keep = 3)

checkpoint.restore(checkpoint_dir)
```

Out[2]:
```
<tensorflow.python.checkpoint.checkpoint.CheckpointLoadStatus at 0x22b1fb03bb0>
```

In [3]:
```
#thank you for your patience while this cell runs
#optional: change how many samples to generate
number_of_samples_to_generate = 4

#generate geometry with trained network
if number_of_samples_to_generate < 11:
    seed = tf.random.normal([number_of_samples_to_generate, 200])
    util.save_generated_matrix('16Rsamples', generator, 0, seed)
    print("SUCCESSFUL: Generated Samples")
else:
    print("FAILED: value of number_of_samples_to_generate must be less than or equal to
```

```
SUCCESSFUL: Generated Samples
```

In [4]:
```
#thank you for your patience while this cell runs
#create and save images of generated matrices
matrices_location = generated_matrices + '/16Rsamples'
util.visualize_files_from_folder(matrices_location, save_location, '16Rsamples')
```

```
located 4 files in generated/generated_matrices/16Rsamples
0 images were already visualized. Generating 4 images, please wait
generated 4 files in generated/generated_matrices/16Rsamples
```

In [5]:
```
#show saved images of geometry
images = []
image_path = save_location + '/16Rsamples'
all_files = [f for f in os.listdir(image_path) if os.path.isfile(os.path.join(image_path

images = [Image.open(image_path + '/' + x) for x in all_files]

new_im = Image.new('RGB', (1000, 330 * (1 + math.ceil(len(all_files)/3))), (255, 255, 25

for i, im in enumerate(images):
    width, height = im.size
    target_width = 330 #px
    ratio = width / target_width
    new_height = int(height // ratio)

    im_resized = im.resize((target_width, new_height))

    new_im.paste(im_resized, (im_resized.size[0] * (i - (math.floor(i / 3) * 3)), math.f

display(new_im)
```

Bibliography

Arjovsky, M., Chintala, S., and Bottou, L. (2017). Wasserstein GAN. *arXiv:1701.07875*.

Augmenta (2022). Augmenta - Automated building design. https://www.augmenta.ai/.

Autodesk (2016). Project Dreamcatcher. https://www.autodesk.com/research/projects/project-dreamcatcher.

Bai, A., Hsieh, C.-J., Kan, W., and Lin, H.-T. (2022). Reducing Training Sample Memorization in GANs by Training with Memorization Rejection.

Banga, S., Gehani, H., Bhilare, S., Patel, S. J., and Kara, L. B. (2018). 3D Topology Optimization Using Convolutional Neural Networks.

Bengio, Y. and Lecun, Y. (1997). Convolutional Networks for Images, Speech, and Time-Series. *The handbook of brain theory and neural networks*.

Biljecki, F., Ledoux, H., and Stoter, J. (2016). An improved LOD specification for 3D building models. In *Computers, Environment and Urban Systems*, pages 25–37.

Borji, A. (2022). Pros and cons of gan evaluation measures. *Journal of Computer Vision and Image Understanding*, 215:103329.

Brake, A. (2023). Voxel model visualizer. Software, Python Library. https://github.com/andreasbrake/voxel-model-visualizer.

Cagan, J. and Antonsson, E. K. (2001). Engineering Shape Grammars. In *Formal Engineering Design Synthesis*, pages 65–92. Cambridge University Press.

CambridgeDictionary (2022). Zero-sum game. https://dictionary.cambridge.org/dictionary/english/zero-sum-game.

CapitalValue (2022). Housing shortage in the netherlands will increase for at least three years to 316,000 homes in 2024. https://www.capitalvalue.nl/en/news/housing-shortage-in-the-netherlands-will-increase-for-at-least-three-years-\to-316000-homes-in-2024.

Carlson, W. E. (1982). An algorithm and data structure for 3D object synthesis using surface patch intersections. *ACM SIGGRAPH Computer Graphics*, 16(3):255–263.

Carta, S. (2021). Self-Organizing Floor Plans. *Harvard Data Science Review*, 3(3).

Chahine, A. (2019). Automation & Why Architects Can't Be Automated. https://www.architecturelab.net/automation-why-architects-cant-be-automated/.

Chen, Z. and Tong, Y. (2017). Face Super-Resolution Through Wasserstein GANs. *arXiv:1705.02438*.

Chokwitthaya, C., Zhu, Y., Mukhopadhyay, S., and Collier, E. (2020). Augmenting building performance predictions during design using generative adversarial networks and immersive virtual environments. *Automation in Construction*, 119:103350.

Ciregan, D., Meier, U., and Schmidhuber, J. (2012). Multi-column deep neural networks for image classification. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3642–3649.

Clevert, D.-A., Unterthiner, T., and Hochreiter, S. (2015). Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs). *Under Review of International Conference on Learning Representations (ICLR) 2016*.

Cosentino, C., Vescio, B., and Amato, F. (2013). Cellular Automata. In Dubitzky, W., Wolkenhauer, O., Cho, K.-H., and Yokota, H., editors, *Encyclopedia of Systems Biology*, pages 381–385. Springer, New York, NY.

Davis, D. (2015). Why Architects Can't Be Automated. https://www.architectmagazine.com/technology/why-architects-cant-be-automated_o.

Delft High Performance Computing Centre (DHPC) (2022). DelftBlue Supercomputer (Phase 1). https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase1.

Delon, G. L. (1970). A Methodology for Total Hospital Design - PMC. *Health Services Research*, 5(3):210–223.

Du, Z., Shen, H., Li, X., and Wang, M. (2022). 3D building fabrication with geometry and texture coordination via hybrid GAN. *Journal of Ambient Intelligence and Humanized Computing*, 13(11):5177–5188.

Eastman, C. (1975). The Use of Computers Instead of Drawings in Building Design. *AIA Journal*, 63.

Girdhar, R., Fouhey, D. F., Rodriguez, M., and Gupta, A. (2016). Learning a Predictable and Generative Vector Representation for Objects. *arXiv: 1603.08637*.

Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep Sparse Rectifier Neural Networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 315–323. JMLR Workshop and Conference Proceedings.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative Adversarial Networks. *Advances in Neural Information Processing Systems*, 3.

Gulrajani, I., Ahmed, F., Arjovsky, M., Dumoulin, V., and Courville, A. (2017). Improved Training of Wasserstein GANs. *arXiv:1704.00028*.

Hinton, G. (2018). Neural Networks for Machine Learning Lecture 6a Overview of mini-batch gradient descent.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence. U Michigan Press, Oxford, England.

IBISWorld (2021). Gloabl Architectural Services Industry Market Research Report. https://www.ibisworld.com/global/market-research-reports/global-architectural-services-industry/.

Ibrahimli, N. (2022). Convolutional Neural Networks.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning*, volume 37 of *ICML'15*, pages 448–456. JMLR.org.

Kasmarik, K. (2023). A framework to integrate generative design techniques for enhancing design automation. *Proceedings of the 15th International Conference on Computer Aided Architectural Design Research in Asia / Hong Kong 7-10 April 2010, pp. 127-136*.

Kelly, T., Guerrero, P., Steed, A., Wonka, P., and Mitra, N. J. (2018). FrankenGAN: Guided detail synthesis for building mass models using style-synchonized GANs. *ACM Transactions on Graphics*, 37(6):216:1–216:14.

Keshavarzi, M. and Rahmani-Asl, M. (2021). GenFloor: Interactive generative space layout system via encoded tree graphs. *Frontiers of Architectural Research*, 10(4):771–786.

Kevin Kelly (2016). *The Inevitable : Understanding the 12 Technological Forces That Will Shape Our Future*. Penguin Books, New York, New York.

Kim, S., Kim, D., and Choi, S. (2020). CityCraft: 3D virtual city creation from a single image. *The Visual Computer*, 36(5):911–924.

Kingma, D. P. and Ba, J. (2014). Adam: A Method for Stochastic Optimization. In *International Conference of Learning Representations*.

Kleinberg, B., Li, Y., and Yuan, Y. (2018). An Alternative View: When Does SGD Escape Local Minima? In *Proceedings of the 35th International Conference on Machine Learning*, pages 2698–2707. PMLR.

Koning, H. and Eizenberg, J. (1981). The Language of the Prairie: Frank Lloyd Wright's Prairie Houses. *Environment and Planning B: Planning and Design*, 8(3):295–323.

Kramer, M. A. (1991). Nonlinear Principal Component Analysis Using Autoassociative Neural Networks. *AIChE Journal*, 37(2).

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521(7553):436–444.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

LeCun, Y., Kanter, I., and Solla, S. (1990). Second Order Properties of Error Surfaces: Learning Time and Generalization. In *Advances in Neural Information Processing Systems*, volume 3. Morgan-Kaufmann.

Lindenmayer, A. (1968). Mathematical models for cellular interactions in development I. Filaments with one-sided inputs. *Journal of Theoretical Biology*, 18(3):280–299.

Maas, A. L., Hannun, A. Y., and Ng, A. Y. (2013). Rectifier Nonlinearities Improve Neural Network Acoustic Models. In *Proceedings of the 30th International Conference on Machine Learning*, volume 28.

Marr, B. (2020). Can Machines And Artificial Intelligence Be Creative? https://www.forbes.com/sites/bernardmarr/2020/02/28/can-machines-and-artificial-intelligence-be-creative/.

McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133.

McManus, K. (2022). Tiny houses: A neat homelessness solution or a reminder of the problem? https://lgiu.org/tiny-houses-a-neat-homelessness-solution-or-a-reminder-of-the-problem/.

Minsky, M. and Papert, S. A. (1969). *Perceptrons*. The MIT Press.

Mordvintsev, A., Olah, C., and Tyka, M. (2015). Inceptionism: Going Deeper into Neural Networks. https://ai.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html.

Nair, V. and Hinton, G. E. (2010). Rectified Linear Units Improve Restricted Boltzmann Machines. In *International Conference on Machine Learning*.

Newton, D. (2019). Generative Deep Learning in Architectural Design. *Technology—Architecture + Design*, 3(2):176–189.

NLTimes (2022). Netherlands won't manage to build 1 million homes in 10 years. https://nltimes.nl/2022/03/11/netherlands-wont-manage-build-1-million-homes-10-years.

OED (2022). Artificial intelligence, n. : Oxford English Dictionary. https://www.oed.com/viewdictionaryentry/Entry/271625.

Oh, S., Jung, Y., Kim, S., Lee, I., and Kang, N. (2019). Deep Generative Design: Integration of Topology Optimization and Generative Models. *Journal of Mechanical Design*, 141(11).

OpenAI (2021). DALL·E: Creating Images from Text. https://openai.com/blog/dall-e/.

Picsart (2023). Meet SketchAI: A New App to Turn Your Drawings into AI Art. https://picsart.com/blog/post/meet-sketchai-a-new-app-to-turn-your-drawings-into-ai-art.

Radford, A., Metz, L., and Chintala, S. (2016). Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks.

Rippingale, J. (2014). Inside England's New Shipping-Container Ghetto. https://www.vice.com/en/article/gq8jyq/i-spent-the-night-in-brightons-homeless-shipping-container-housing-project.

Rodrigues, R. C. and Duarte, R. B. (2022). Generating floor plans with deep learning: A cross-validation assessment over different dataset sizes. *International Journal of Architectural Computing*, 20(3):630–644.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65:386–408.

Ruffle, S. (1986). Architectural Design Exposed: From Computer-Aided Drawing to Computer-Aided Design. *Environment and Planning B: Planning and Design*, 13(4):385–389.

Rumelhart, D. E. and McClelland, J. L. (1987). Learning Internal Representations by Error Propagation. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Foundations*, pages 318–362. MIT Press.

Salimans, T., Goodfellow, I., Zaremba, W., Cheung, V., Radford, A., Chen, X., and Chen, X. (2016). Improved Techniques for Training GANs. In *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc.

Santurkar, S., Tsipras, D., Ilyas, A., and Madry, A. (2019). How Does Batch Normalization Help Optimization? *arXiv:1805.11604*.

Schwartz, A. (2016). A startup from Google's secretive moonshot lab is changing the way we build cities. https://www.businessinsider.com/inside-flux-a-startup-from-googles-secretive-moonshot-lab-2016-6".

Selvaraju, P., Nabail, M., Loizou, M., Maslioukova, M., Averkiou, M., Andreou, A., Chaudhuri, S., and Kalogerakis, E. (2021). Buildingnet: Learning to label 3d buildings. In *IEEE/CVF International Conference on Computer Vision (ICCV)*.

Sharma, A., Grau, O., and Fritz, M. (2016). VConv-DAE: Deep Volumetric Shape Learning Without Object Labels. In Hua, G. and Jégou, H., editors, *Computer Vision – ECCV 2016 Workshops*, Lecture Notes in Computer Science, pages 236–250. Springer International Publishing.

Shelter (2017). 1 in 5 adults suffer mental health problems such as anxiety, depression and panic attacks due to housing pressures. https://england.shelter.org.uk/media/press_release/1_in_5_adults_suffer_mental_health_problems_such_as_anxiety_depression_and_panic_attacks_due_to_housing_pressures.

Simon, H. A. (1970). The Sciences of the Artificial. In *The Sciences of the Artificial*. MIT Press.

Smith, E. and Meger, D. (2017). Improved Adversarial Systems for 3D Object Generation and Reconstruction. *arXiv:1707.09557*.

Sohrab, H. H. (2003). *Basic Real Analysis*. Springer Science & Business Media.

Spacemaker (2023). Spacemaker has a new home. https://www.spacemakerai.com//.

Springenberg, J. T., Dosovitskiy, A., Brox, T., and Riedmiller, M. A. (2014). Striving for Simplicity: The All Convolutional Net. *International Conference on Learning Representations (ICLR)*.

Stiny, G. and Mitchell, W. J. (1978). The Palladian grammar. *Environment and Planning B: Planning and Design*, 5(1):5–18.

Tangelder, J. and Veltkamp, R. (2004). A survey of content based 3D shape retrieval methods. In *Proceedings Shape Modeling Applications, 2004.*, pages 145–156.

TestFit (2023). TestFit: Real Estate Feasibility Platform. https://testfit.io/.

ThisPersonDoesNotExist (2023). This Person Does Not Exist. https://thispersondoesnotexist.com.

van Kaick, O., Zhang, H., Hamarneh, G., and Cohen-Or, D. (2011). A Survey on Shape Correspondence. *Computer Graphics Forum*, 30(6):1681–1707.

Vesely, O. (2022). Building Massing Generation using GAN Trained on Dutch 3D City Models. Master's Thesis, TU Delft.

Victor, D. (2022). World Population Reaches 8 Billion, U.N. Says. https://www.nytimes.com/2022/11/15/world/world-population-8-billion.html.

Vierlinger, R. (2012). Octopus. Software, Grasshopper Plug-In. https://www.food4rhino.com/en/app/octopus.

Wang, B. and Wang, J. (2021). Application of Artificial Intelligence in Computational Fluid Dynamics. *Industrial & Engineering Chemistry Research*, 60(7):2772–2790.

Werbos, P. and John, P. (1974). *Beyond Regression : New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University.

Wikipedia, L. (2023). Activation function graphs. https://en.wikipedia.org/wiki/Activation_function. This work is licensed unchanged under the Creative Commons Attribution 4.0 International License. To view a copy of this license, visit https://creativecommons.org/licenses/by-sa/4.0/.

Wolfram, S. (2002). *A New Kind of Science*. Stephen Wolfram.

Wu, A. N., Stouffs, R., and Biljecki, F. (2022). Generative Adversarial Networks in the Built Environment: A Comprehensive Review of the Application of GANs across Data Types and Scales. *Building and Environment*, 223(109477).

Wu, J., Zhang, C., Xue, T., Freeman, W. T., and Tenenbaum, J. B. (2016). Learning a Probabilistic Latent Space of Object Shapes via 3D Generative-Adversarial Modeling. In *29th Conference on Neural Information Processing Systems*, page 11.

Wu, Z., Song, S., Khosla, A., Yu, F., Zhang, L., Tang, X., and Xiao, J. (2015). 3D ShapeNets: A Deep Representation for Volumetric Shapes. *arXiv:1406.5670*.

Xiang, S. and Li, H. (2017). On the Effects of Batch and Weight Normalization in Generative Adversarial Networks. *arXiv:1704.03971*.

Yamaguchi, K., Sakamoto, K., Akabane, T., and Fujimoto, Y. (1990). A neural network for speaker-independent isolated word recognition. In *Proc. First International Conference on Spoken Language Processing (ICSLP 1990)*, pages 1077–1080.

You, K., Long, M., Wang, J., and Jordan, M. I. (2019). How Does Learning Rate Decay Help Modern Neural Networks? *arXiv:1908.01878*.

Zeiler, M. D. and Fergus, R. (2013). Visualizing and Understanding Convolutional Networks. *arXiv:1311.2901*.

Colophon

This document was typeset using LaTeX, using the KOMA-Script class `scrbook`. The main font is Palatino.

Lisa-Marie Mueller

2023

TU Delft