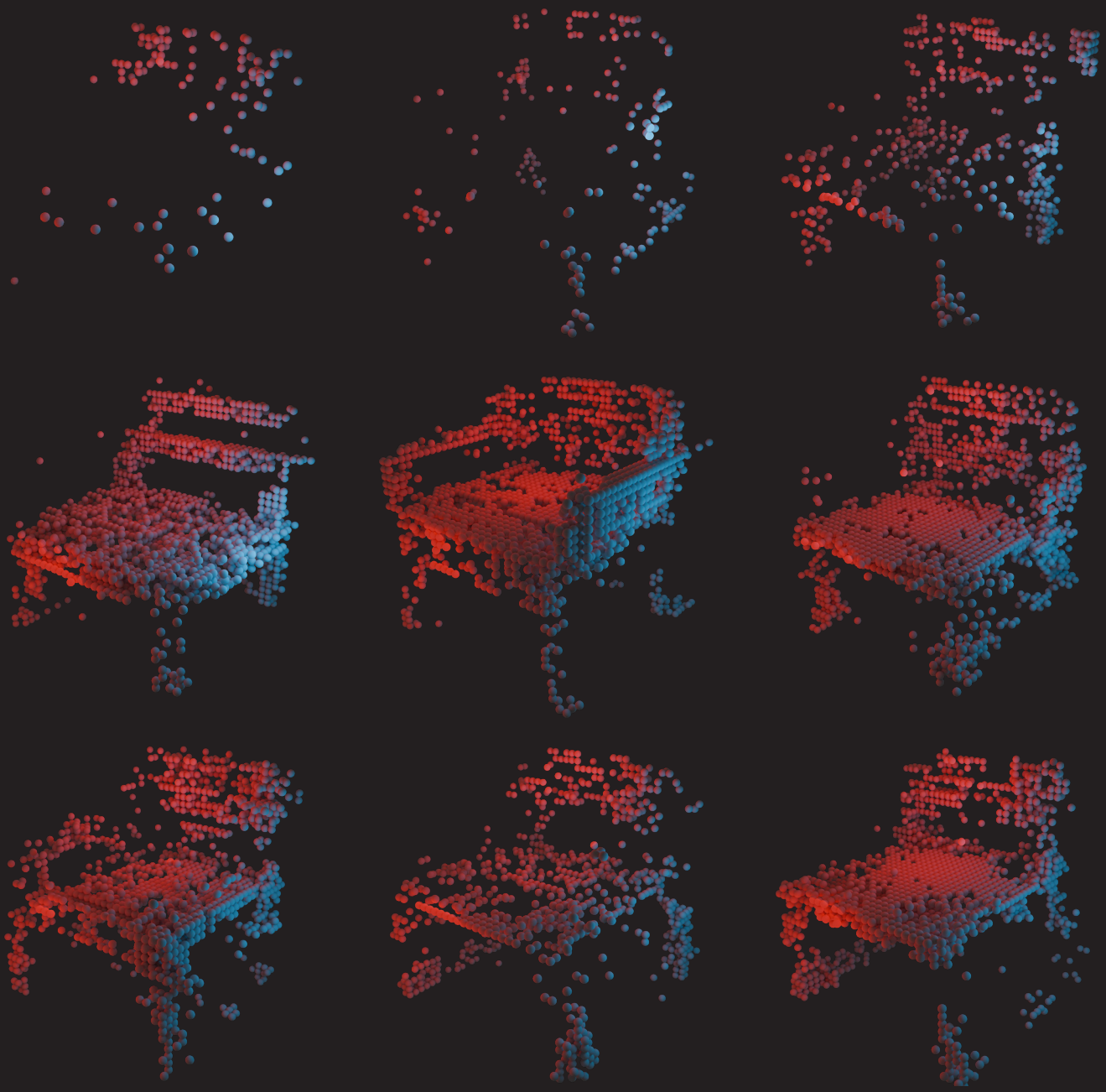


# E.P.I.C 3D

Enhancement and Precision
Iterative Creation for 3D Models

MSc Student: Simos Maniatis
TU Delft | Technische Universiteit

Supervisors: Dr. Charalampos Andriotis
Dr. Michela Turrin, Lisa-Marie Mueller
AR0202 Computational Intelligence For Integrated Design

## The Current State of the Art of Generative Adversarial Networks (GANs)

My research primarily focuses on harnessing the power of Generative Adversarial Networks (GANs) to generate 3D geometry, with a particular emphasis on controlling outputs through realistic variables such as structural parameters or energy performance. This course helped in exposing to the possibilities but also the limitations of this technology. This move beyond generating random geometry based solely on dataset characteristics aims to achieve more targeted and practical applications in fields like architecture and engineering. Of course, this meant that the final product of this strategy would possibly not be a simple GAN, but with an entry-medium level in Python, together with the support of the course's supervising team, we took the decision to focus on the creation of a stable 3DGAN, since that itself would be a more than sufficient challenge and also a really interesting experiment since most applications of this technology are focused into image generation or in general 2D shapes. GANs, which are composed of a generator and a discriminator, form a groundbreaking development in deep learning. The generator is tasked with creating data that simulates real-world information, while the discriminator assesses this data against actual data, creating a competitive environment that enhances the fidelity of the outputs. This adversarial process not only aims to produce data but also strives to refine the outputs to be increasingly indistinguishable from the genuine articles. By integrating these capabilities, my work seeks to have a complete prototype of the generative phase (3DGAN) of an AI tool, which not only generates geometry but also strives to refine it.

**Advancements in High-Resolution and Quality Image Generation**
The GAN landscape has seen significant advancements, particularly in the realm of image generation. The StyleGAN series, including StyleGAN, StyleGAN2, and StyleGAN3, has been particularly influential, offering unprecedented control over image quality and the manipulation of various attributes from facial features to background elements. BigGAN has further pushed the boundaries by enabling large-scale training that enhances both the quality and resolution of images, marking a notable leap in the capabilities of GANs to generate high-fidelity visual content.

**Expanding Beyond Single Images**
The utility of GANs extends beyond static images. Innovations like GANsformers have adapted the transformer architecture to image generation, enhancing the capacity for detail and complexity over long ranges. In the realm of video, GANs have started to synthesize realistic video sequences, showcasing their potential in dynamic media creation. Additionally, 3D generation capabilities of GANs have opened new avenues in virtual reality and game development, where realistic objects and scenes are now being crafted with unprecedented ease and precision.

**Enhanced Control Over the Generation Process**
Key to the evolution of GANs has been the deepening understanding of latent space—the internal 'idea space' of GANs. This understanding has enabled nuanced manipulations such as style mixing and semantic editing, where specific features can be adjusted with remarkable precision. Techniques like GAN inversion, where real images are mapped back into the GAN's latent space, have bridged the gap between synthetic and real, providing powerful tools for real-world image editing.

**Challenges and Ethical Considerations**
Despite these advancements, challenges such as mode collapse, where GANs produce limited varieties of outputs, and the inherent difficulties in training stability remain. Additionally, the ethical implications of GAN technology, particularly in the creation of deepfakes, have prompted a necessary discourse on the responsible use of this technology. It is critical that as GANs continue to develop, they are paired with robust ethical frameworks and techniques to mitigate potential harms.

**Real-World Applications**
Generative Adversarial Networks (GANs) are making significant impacts across various real-world applications, particularly in design and urban planning. One notable implementation is Autodesk's Dreamcatcher, a generative design system that incorporates GAN-powered elements to facilitate architectural planning. In this system, architects can input specific goals and constraints—such as square footage, structural requirements, and site conditions—to generate multiple viable design options. For instance, in a project aimed at relocating an office, Dreamcatcher was instrumental in producing layout options that enhanced employee well-being, optimized sunlight penetration, and improved collaboration areas.

Another innovative application is FacadeGAN, a research project focused on creating realistic building facades in New York City. Utilizing a conditional GAN that processes an image segmentation map detailing windows, doors, and materials, FacadeGAN enables architects to rapidly prototype and visualize facade designs, ensuring that new constructions integrate seamlessly with historical contexts. Additionally, the Bayesian CycleGAN for Urban Planning illustrates the versatility of GANs in environmental and urban management. This tool uses GANs to modify urban satellite imagery across different times of the day, aiding in predictions of how shadows might influence urban spaces, assisting in traffic congestion planning, and evaluating seasonal vegetation changes. This application incorporates a Bayesian approach to introduce uncertainty estimation, enhancing responsible decision-making in urban planning. These examples underscore the growing relevance of GANs in practical scenarios, where they enable more efficient, precise, and innovative solutions in complex fields such as architecture and urban planning.

**The Impact of Generative Adversarial Networks (GANs) on Architecture and the Built Environment**

The evolution of architectural design has been significantly influenced by advancements in technology, and the introduction of Generative Adversarial Networks (GANs) marks a pivotal shift in how architects create and conceptualize spaces. Since their inception by Ian Goodfellow in 2014, GANs have revolutionized several fields by generating synthetic data, and their impact on architecture, particularly in the realm of synthetic design generation, has been profound.

**Foundations of GANs in Architectural Design**

GANs operate through two main components: the generator and the discriminator. The generator creates images from random noise, which are then evaluated by the discriminator. This interaction enhances the generator's ability to produce increasingly realistic images, mimicking actual data. In architecture, this capability allows for the generation of varied design alternatives, pushing the boundaries of traditional design processes.

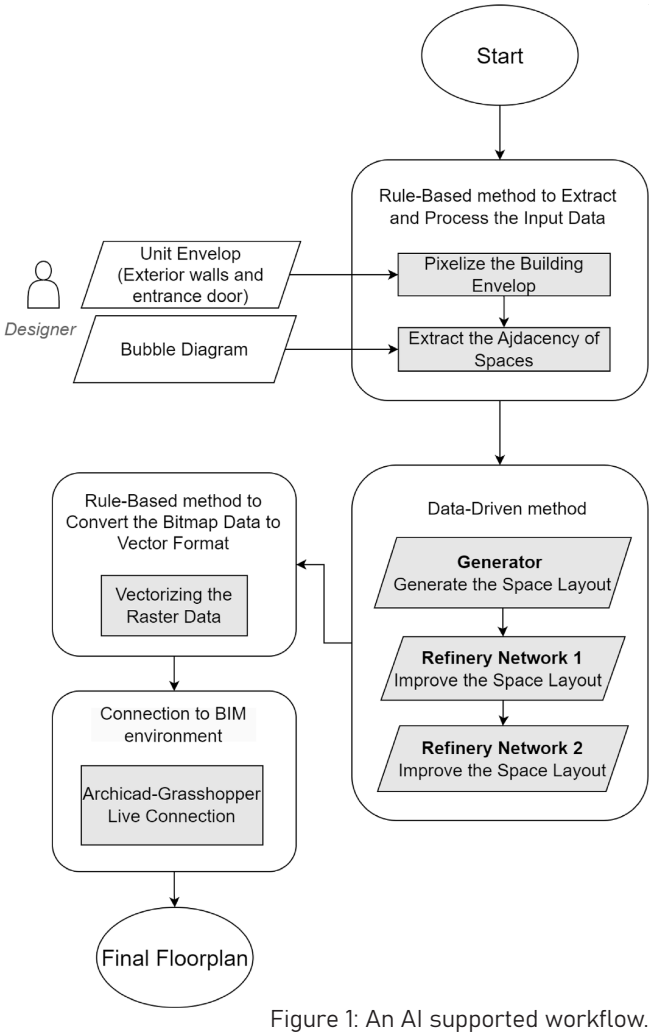



Figure 1: An AI supported workflow. Source: Aalaei, M., Saadi, M., Rahbar, M. and Ekhlassi, A., 2023. Architectural layout generation using a graph-constrained conditional Generative Adversarial Network (GAN). Automation in Construction, 155, p.105053.

**Transition to Architectural Applications**

The initial applications of GANs in architecture focused on image generation, such as the creation of new designs based on existing patterns. However, significant developments occurred with the introduction of image-to-image translation techniques like Pix2Pix in 2018 by Isola et al. This method enabled the conditioning of generated images on other images, facilitating the creation of architectural designs from one form to another, such as converting sketches to detailed layouts.

**Graph Neural Networks and Vectorization**

Recent studies have focused on overcoming the limitations of raster output through the use of graph neural networks (GNNs) and direct vector generation. For instance, Nauata et al. introduced a graph-constrained GAN model that uses convolutional message passing layers to maintain the integrity of spatial relationships in architectural designs. Similarly, researchers like Liu et al. and Luo & Huang have developed methods that generate vector outputs directly, thus preserving the geometrical fidelity of the designs.

**Future Directions and Architectural Autonomy**

The integration of GNNs with GANs to handle graph-structured data offers a promising avenue for further research. This combination allows for the maintenance of topological accuracy in generated designs, aligning more closely with the needs and visions of architects. Moreover, as GAN technology continues to evolve, there is potential for architects to gain greater control over the generated outputs, enabling more customized and contextually appropriate design solutions.

**Why E.P.I.C 3D**

E.P.I.C 3D is an experimental attempt to design an AI tool which stands at the intersection of aspiration and technical possibility, crafting 3D geometries while fine-tuning them against user-defined objectives like energy efficiency and structural resilience. At the heart of E.P.I.C 3D is a neural network that uses latent vectors to generate 'fake' outputs, which are then scrutinized by a discriminator network against a dataset of 'real' 3D objects. The iterative training loop aims to close the gap between generated and genuine objects, aiming for a synthesis of high utility and authenticity.

The deep learning agent within the system brings an additional layer of sophistication, focusing on specific objectives and employing a convolutional neural network (CNN) to develop a custom loss function around these goals. The simulated dataset serves as a dynamic testbed, supplying on-the-fly data that guides the deep learning agent's reward mechanism, thereby driving the optimization of results.

E.P.I.C 3D represents an ongoing experimental process rather than a finalized solution, reflecting the complexity and evolving nature of integrating AI into architectural practices. The endeavor is grounded in a realistic understanding that the road to fully optimized designs is iterative and exploratory. It acknowledges the current capabilities of AI while pushing against the boundaries of what's possible, adapting and learning with each feedback cycle. The tool's structure, illustrated in the accompanying diagram, encapsulates this journey – from concept to simulated test runs to an optimized result, each step is a learning opportunity aimed at bringing us closer to the optimal convergence of form, function, and sustainability.

This report will closely focus on my attempt to expand my knowledge and skills to craft ad 3DGAN which will generate new geometries based on a dataset of 3D objects.


Figure 2: The Challenge
Source: Author

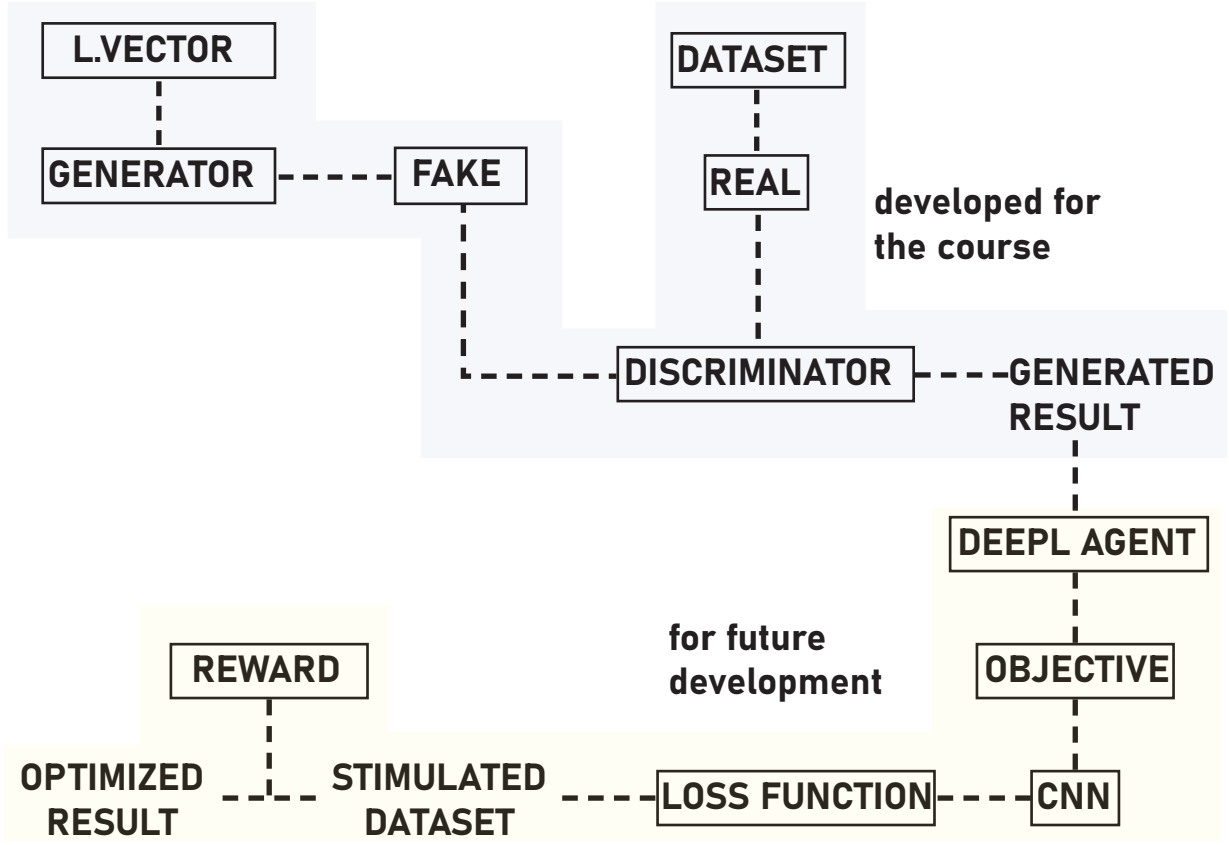



Figure 3: E.P.I.C 3D
Source: Author

## E.P.I.C 3D - GENERATIVE PHASE DEVELOPMENT

**The overall concept**

The project is an attempt to embody the advancement of the domain of 3D data processing, utilizing a deep learning framework to develop and train 3D generative adversarial networks. This script should demonstrate an intricate synthesis of multiple neural network architectures , each chosen to enhance the handling, processing, and visualization of three-dimensional data, which is pivotal in numerous applications ranging from virtual reality to 3D printing.

A really import section of EPIC3D would be the data preprocessing. That's why libraries like Pytorch and Open3D are going to be valuable to fully able to modify and prepare the dataset for training. PyTorch is a popular choice for researchers and developers working in the field of artificial intelligence due to its flexibility, speed, and the ease with which it integrates with CUDA to harness the power of GPU computing. This allows the script to perform data-intensive operations with increased efficiency, a necessary feature given the complex nature of 3D modeling.

In addition the intergration of numpy and scipy further complements this setup, providing robust capabilities for numerical operations and image manipulation—essential tools for preparing and augmenting the training data. This preprocessing stage is crucial for ensuring that the neural network learns to generate new 3D models that are both diverse and realistic.

Visualization tools play a pivotal role in the script not only for the data processing but also for debug and troubleshooting, since errors were bound to happen and feedback was necessary to attempt any kind of fix to the code. These tools are not merely auxiliary; they are integral to the model development process. They provide immediate, actionable insights into the training process, allowing for quick adjustments and optimizations. This real-time feedback is vital for fine-tuning the network, ensuring optimal performance, and diagnosing issues promptly.

The script should be meticulously structured, indicating a high level of organization in managing the training sessions. It will include detailed setup for logging and saving model checkpoints, suggesting a prepared approach to handle extensive and potentially lengthy training processes. This organized framework not only will enhance the efficiency of the training sessions but also helps in maintaining a clear and systematic record of the model's performance over time.

Furthermore, the script must have the ability to automatically detect and utilize available GPU resources to have am optimized computational strategy, ensuring that the training process is not only effective but also resource-efficient. The use

of CUDA for GPU acceleration is indicative of the script's advanced technical capabilities, allowing it to handle the computationally intensive tasks associated with training deep neural networks.

So to conclude the script should be divided into:

1) Data Gathering

2) Data Preprocessing

3) Conversion to Pytorch Tensors

4) Dataset Creation

5) Training of the GAN

6) Generate geometry with the trained model

**First attempt**
**Data Type: Point Cloud**

The first model, was based around the idea that the user would use point clouds either through gathering an expansive collection of 3D objects or through photogrammetry and 3D scanning. Training a 3D Generative Adversarial Network (3DGAN) with point cloud data is an intricate process fraught with numerous technical challenges and potential pitfalls, particularly when managing and mitigating various errors, including mismatching features between the generated and real datasets. For my project mismatching and dimensional errors proved to be the biggest challenge.

The unstructured nature of point clouds, where data points are distributed irregularly in three-dimensional space, poses significant difficulties. Mismatching can also stem from the lack of representational capacity of the network, insufficient or unbalanced training data, or suboptimal network architecture that fails to capture the complexity of spatial relationships within the data.

Moreover, point clouds typically contain varying densities and noise levels, which can exacerbate the mismatching issue. Noise can lead to the generator focusing on irrelevant features or becoming overly specific to the training data's noise characteristics, hence failing to generalize well. Similarly, variations in density within the training data can lead the network to reproduce these inconsistencies, resulting in generated point clouds that are either too sparse or too dense compared to the target data. Implementing robust preprocessing steps to normalize point cloud density and filter out noise is crucial in reducing such errors.

The high dimensionality and sparsity of point clouds also contribute significantly to overfitting and underfitting problems. Overfitting occurs when the network learns details that are too specific to the training data, including noise and anomalies, rather than capturing the general underlying patterns. This is particularly problematic in 3D data, where the curse of dimensionality can lead to vast empty spaces in the data distribution that the network might not learn to fill correctly. Underfitting, on the other hand, might occur if the network is too simple to capture the complex patterns in 3D shapes, leading to overly generalized and undetailed point cloud outputs.

To address these issues, careful attention must be paid to the selection of network architecture. Networks like PointNet or more advanced modifications that can directly process point clouds might be necessary to effectively capture the intricacies of 3D data. Additionally, ensuring the diversity and comprehensiveness of the training dataset is critical to avoid biases and promote better generalization.

Training dynamics between the discriminator and the generator in GANs further complicate the training process. If not properly balanced, the discriminator might become too strong, leading the generator into a mode where it fails to improve, or it might become too weak, making it unable to guide the generator effectively. Techniques such as implementing gradient penalties, adjusting learning rates dynamically, or using different training ratios between the generator and discriminator are potential strategies to maintain training balance and encourage convergence.

In terms of evaluation, traditional metrics may not suffice due to the unique nature of 3D structures. Developing new metrics that consider spatial and geometric accuracies—such as Earth Mover's Distance (EMD) for measuring the distance between two point sets—can provide more meaningful assessments of how well the generated data matches the real data in both form

and distribution. Additionally, incorporating metrics like the K-Nearest Neighbors (KNN) distance can further enhance the evaluation process. KNN can be used to compare the density and distribution of point clouds by measuring the average distance between corresponding points in the nearest neighbor sets of the generated and real datasets. This metric is particularly useful in identifying discrepancies in local point configurations, thereby providing insights into the fidelity and accuracy of the generated 3D models at a more granular level. Integrating both EMD and KNN in the evaluation framework allows for a comprehensive analysis that assesses both global shape alignment and local point distribution, leading to a more robust validation of 3D generative models.

**The Initial structure**

The initial model architecture designed for training a 3D Generative Adversarial Network (3DGAN) reflects a comprehensive approach that emphasizes meticulous preprocessing and thoughtful network training dynamics.

In the preprocessing phase, the script initiates by loading 3D meshes and assessing their structural integrity, specifically examining the presence of triangles. Meshes that lack triangles are discarded, ensuring that only those with a defined triangulated structure proceed. This selection criterion is critical because triangulated meshes provide a consistent foundation for subsequent transformations into point clouds, a format better suited for handling within neural network frameworks.

Once a mesh qualifies, it is converted into a point cloud format. These point clouds are then transformed into PyTorch tensors, facilitating optimized computation during the training phase due to the tensor's compatibility with the PyTorch framework.

Following data conversion, these tensors are compiled into a dataset, which is then split into training and validation sets. This division is strategic, allocating a substantial portion of the data for training while reserving a separate set for validating the model's performance and its ability to generalize to new data.

The architecture includes two primary components: the generator and the critic, both designed to be equal in strength to maintain balance and foster a productive adversarial environment. This equilibrium is essential for ensuring that neither component overpowers the other, allowing both to evolve and refine their capabilities through ongoing competition.

Several key considerations influence the configuration of the neural networks:

The choice and quantity of layers impact the network's ability to process and generate complex 3D structures.

1) Activation functions introduce necessary non-linearities that help in learning intricate patterns.

2) Regularization techniques such as L1 and L2, along with dropout layers, are employed to prevent overfitting.

3) Batch discrimination aids the critic in making more generalized decisions across different batches.

4) Introducing noise to the generator's input promotes variability and realism in the synthetic point clouds.

5) Proper tensor dimensionality and shape are crucial for the effective handling of 3D data by the networks.

After setting up the networks and preparing the data, the model undergoes training where the generator and critic iteratively improve through adversarial dynamics. Post-training, the model is utilized to generate multiple 3D models for evaluation, assessing the trained model's capacity to produce diverse and realistic outputs. This structured approach ensures that the model not only learns effectively but is also capable of generating high-quality 3D models, demonstrating the success and robustness of the training process.

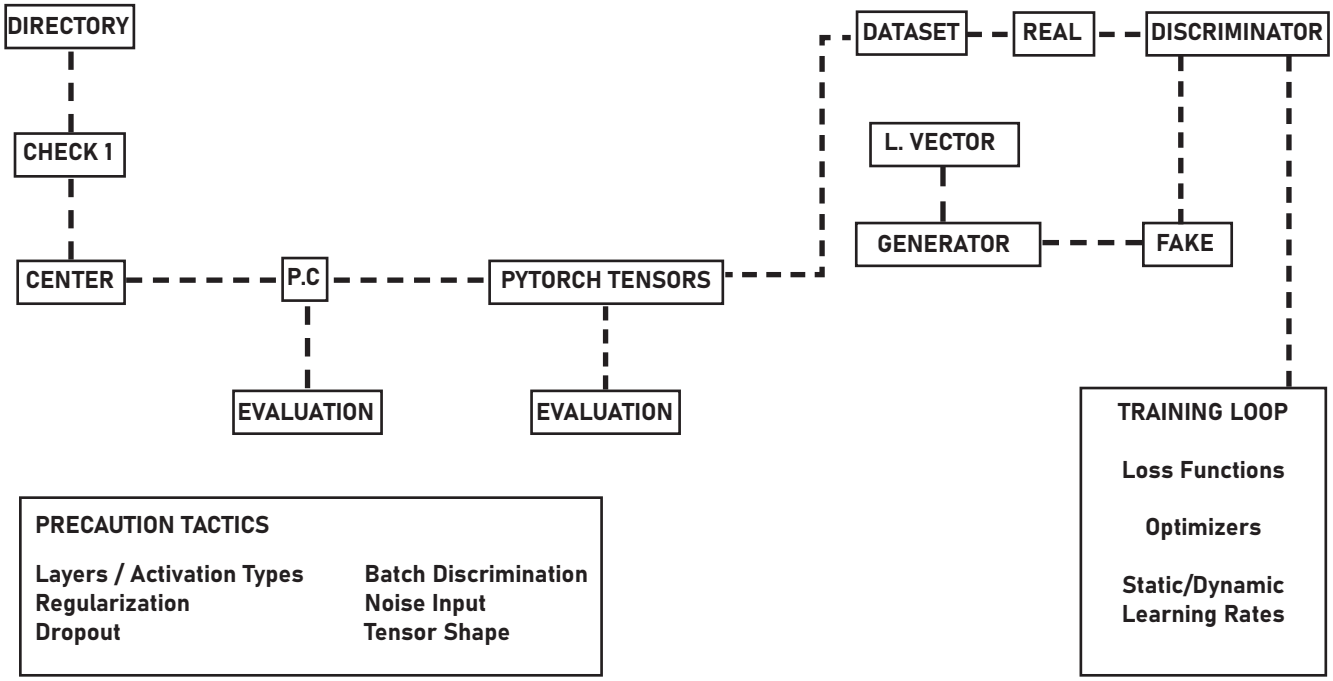



Figure 4: First attempt with point clouds as data
Source: Author

**Model A**
**Dataset: Point Clouds**

The script under consideration employs two sophisticated neural network structures: a Generator and a Critic, each designed to address the complexities associated with generating and evaluating 3D point clouds within a Generative Adversarial Network (GAN) framework.

**Generator Architecture:**

The Generator's architecture is crafted to transform a latent space vector into a three-dimensional point cloud. It comprises a series of fully connected layers that sequentially increase in size: starting from the latent dimension and expanding to 256, 512, and finally 1024 units before reaching the output layer. Each layer, except for the last, is equipped with a ReLU activation function to introduce non-linearity and prevent the vanishing gradient problem. The final output layer, which reshapes the network output to match the flattened 3D point cloud data, employs a Tanh activation to normalize the output values between -1 and 1. This normalization is crucial as it constrains the output to a defined range, promoting consistent data scaling but may potentially limit the diversity of the generated models by compressing all outputs into this interval.

**Critic Architecture:**

Conversely, the Critic is designed with a focus on assessing the realism of generated point clouds. It starts by processing the input through a fully connected layer that reduces the dimensionality to 512, followed by progressive downsizing to 256 dimensions, with each layer followed by a LeakyReLU activation. This choice of LeakyReLU, which allows a small, non-zero gradient when the unit is inactive, helps maintain active neuron states, avoiding issues commonly associated with traditional ReLU units in deep networks. The architecture concludes with a single output unit that provides a scalar value representing the "realness" of the input data, which is crucial for the adversarial training process.

**Hyperparameters and Training Dynamics:**

The learning rate and optimizer's beta1 parameter are crucial hyperparameters in training GANs. The chosen learning rate of 0.00015 is relatively conservative, aiming to provide stable but potentially slow training progression. The beta1 value, set unusually low at 0.3, affects the optimizer's momentum component, significantly influencing the convergence behavior and stability of training. These choices suggest a cautious approach designed to avoid oscillations and instabilities

often encountered in GAN training.

**Loss Functions:**

The loss functions employed further reflect the nuanced balance needed in adversarial training. The Critic's loss is a composite of the differences in evaluations between real and generated data and includes a gradient penalty to stabilize the learning landscape by discouraging sharp gradients, a common source of training instability in GANs. For the Generator, the loss is formulated to encourage it to fool the Critic by maximizing the negative of the Critic's evaluation of the generated data. This setup fosters a competitive environment where both networks strive to outperform each other, driving improvements in the realism of generated outputs.

Overall, the architecture of both networks, coupled with the strategic choice of hyperparameters and loss functions, exhibits a deep understanding of the challenges inherent in training GANs, particularly those dealing with complex 3D data. Nonetheless, the effectiveness of these configurations would greatly benefit from continual evaluation and refinement based on empirical results, with potential adjustments in network complexity, activation functions, and training parameters to better capture the nuances of 3D point cloud generation and evaluation.

**Results after training:**

The trained model generates 2 individual point clouds, one looking really similar to one of the models in the dataset still looking abstract to a significant degree.

The observation that one of the generated models not only resembles but may be nearly identical to one of the models in the dataset indeed warrants further consideration, especially as it might also be indicative of a phenomenon known as model collapse.

Model collapse, a common challenge in training Generative Adversarial Networks, occurs when the Generator begins to produce a limited range of outputs. This issue can manifest in two distinct ways: first, as previously discussed, where the Generator produces abstract or overly simplified outputs; second, which seems to be occurring here, where the Generator overfits to specific instances within the training dataset to the

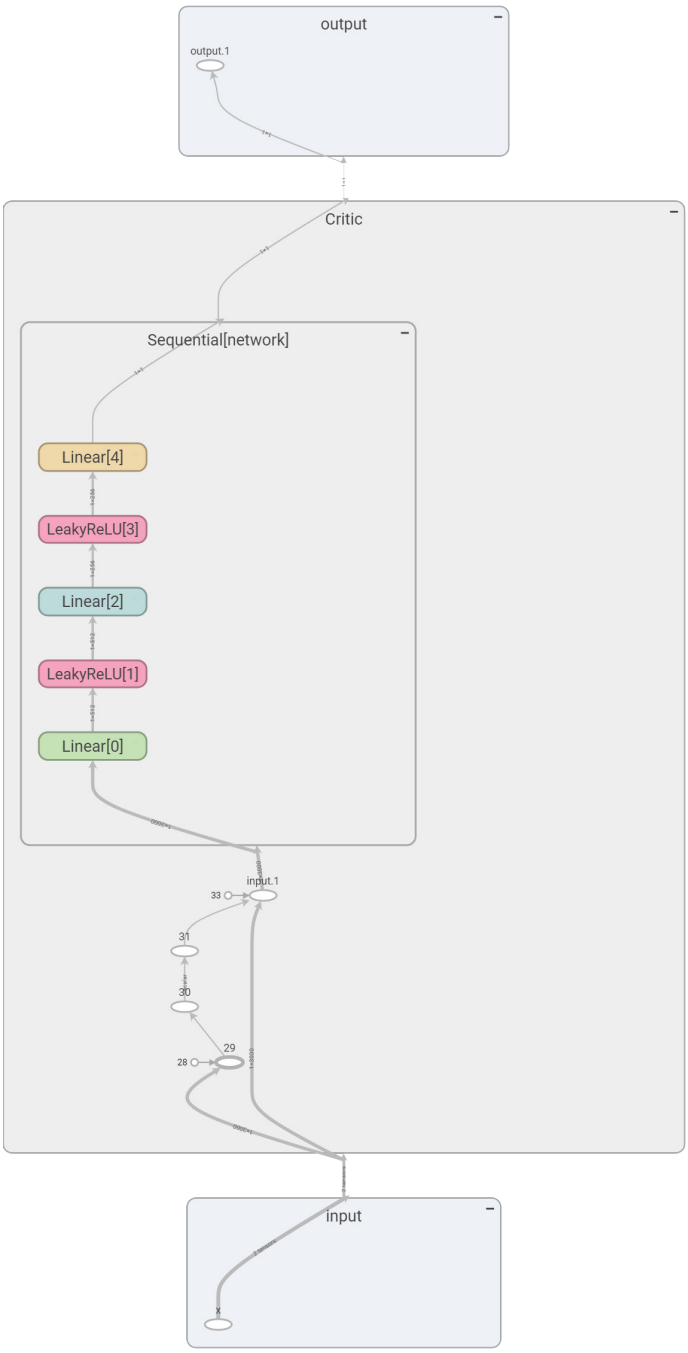



Figure 5: Model A, Neural Network Architecture
Source: Author

extent that it starts replicating those instances almost exactly.

The potential replication of a dataset model points to several critical aspects of the training process and the model's capability:

**Overfitting:** This near-identical replication suggests that the Generator might be too finely tuned to the training data, capturing and replicating the minutest details of certain training examples instead of learning to generalize from broader data features. This overfitting prevents the Generator from innovating new models based on learned data characteristics, instead defaulting to copying examples it has been most exposed to.

**Insufficient Diversity in Training Data:** If the training dataset is not diverse enough, or if the Generator disproportionately focuses on certain data patterns due to biases in the training process, it might result in the Generator mimicking those frequently seen or less complex examples. This would limit the variety of outputs the model can produce, effectively reducing the GAN's utility in applications requiring novel model generation.

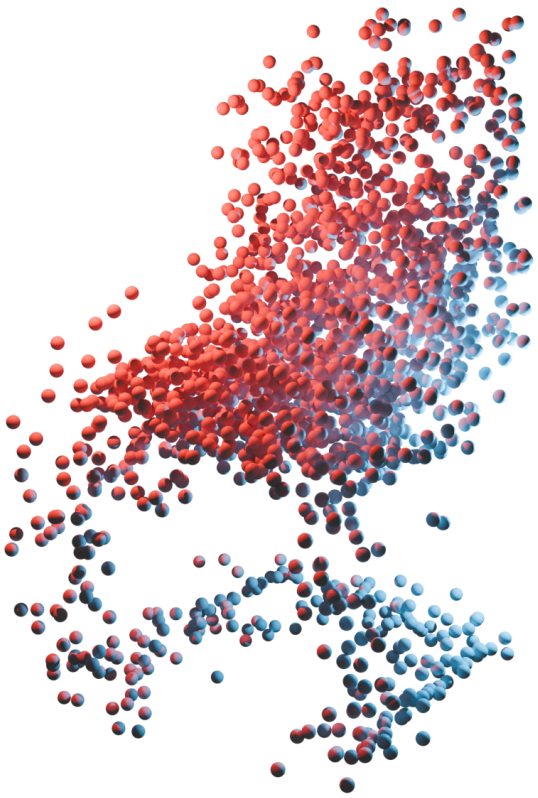
**Hyperparameter and Training Regime Adjustment:** The beta1 value of the Adam optimizer, the number of epochs, and the balance of power between the Critic and the Generator might need reconsideration. Specifically, the training regime should ensure that neither the Generator nor the Critic becomes too dominant, which can destabilize training and lead to poor generalization capabilities.

In sum, while the precise replication of a model from the dataset underscores the Generator's ability to accurately render complex 3D structures, it also highlights the risk of the network learning to replicate rather than generalize. This behavior underscores the need for further tuning and perhaps architectural adjustments to encourage a broader exploration of the model space, ensuring that the GAN not only reproduces existing data but also innovatively generates diverse and novel 3D forms. Addressing these issues will enhance the GAN's applicability in practical scenarios where creativity and variation are prized.

**Suggestions for optimization**

**Modify Network Architecture:**If the dataset is significantly enriched, consider increasing the complexity of the Generator and Critic by adding more layers or adjusting the layer sizes. This can help the network learn more complex and varied patterns.

**Regularization Techniques:** Implement dropout in the Generator to prevent it from relying too heavily on any particular path, encouraging it to explore a wider range of features. Additionally, use regularization methods such as L1 or L2 regularization in the Critic to penalize overly complex models that might memorize training data.



Dataset : List of Objs (3d objects) / latent space : 50
batch size: 4 / learning rate: 0.0001 / epochs: 2000

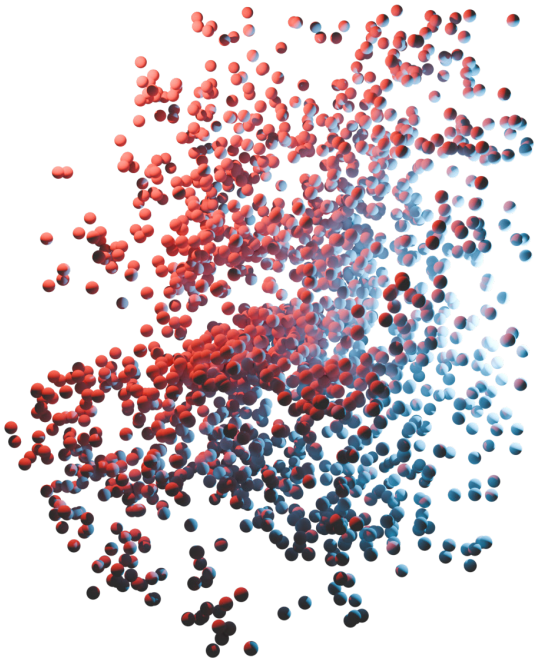


Figure 6: The 2 models generated by model A
Source: Author

**Model B**
Dataset: Point clouds

Modifications

In response to the challenges identified with the 3D Generative Adversarial Network (3DGAN), particularly concerning model collapse and the need for greater diversity and generalization in the generated models, a strategic decision was made to adjust several key parameters of the network's training and architecture. These adjustments are aimed at enhancing the model's ability to learn more complex and varied features from the training data, and to stabilize the training process.

**Learning Rate Adjustment:** The learning rate for both the Generator and the Critic was increased from 0.0001 to 0.0002. This adjustment is intended to accelerate the learning process, allowing the network to explore a broader range of solutions and potentially escape local minima more effectively. A higher learning rate can facilitate faster convergence but requires careful monitoring to ensure it does not lead to instability in training dynamics.

**Increase in Training Duration:** The number of training epochs was doubled from 4000 to 8000. Extending the training duration gives the network more opportunity to learn and refine its generation capabilities over a larger set of iterations. This is particularly useful when dealing with complex 3D data, as it provides ample time for the deep neural networks to adjust their weights and biases to better capture the nuances of the data.

**Expansion of Network Depth:** Both the Generator and the Critic had their architectures expanded by adding two additional layers each. Increasing the depth of the networks is expected to enhance their capacity to model more intricate relationships within the data. More layers allow for a deeper hierarchy of features to be learned, which can be crucial for generating and discriminating between complex 3D point clouds. However, care must be taken to manage the increased risk of overfitting that comes with a larger model, potentially requiring further adjustments to regularization strategies.

These enhancements are designed to address specific issues observed during initial training runs, aiming to improve the overall performance and output quality of the GAN. By implementing these changes, the model is expected to become more robust in handling diverse 3D shapes and structures, thereby increasing its utility and effectiveness in generating realistic and varied 3D models. Continuous monitoring and evaluation will be necessary to assess the impact of these changes and to make further adjustments as needed.

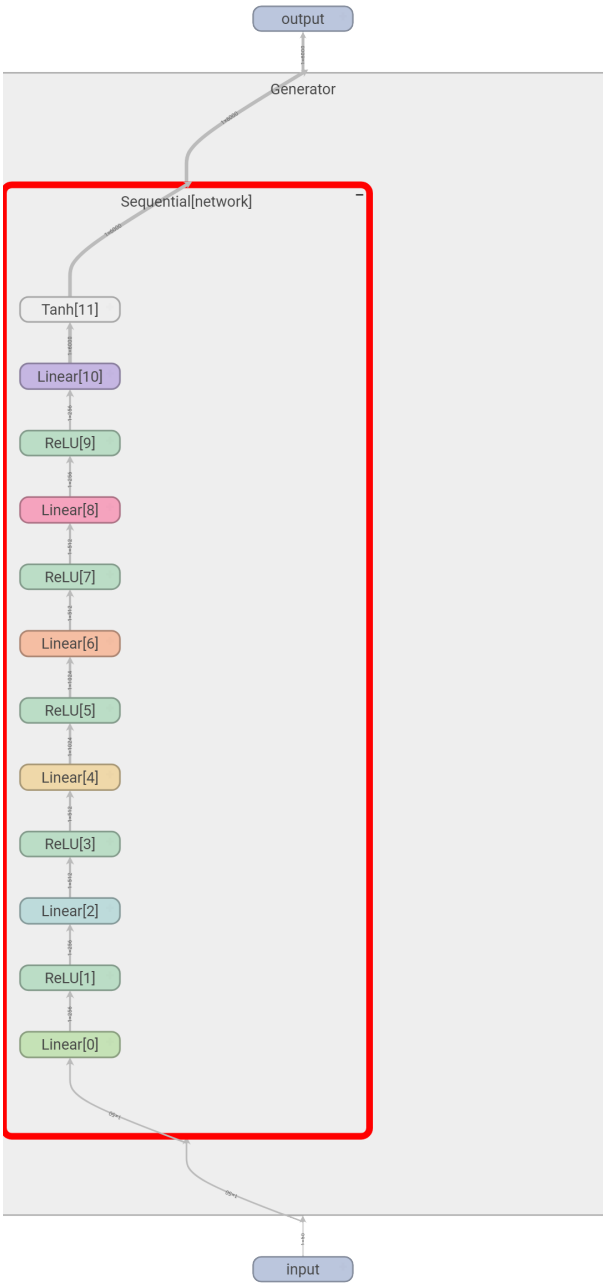



Figure 7: The updated Generator
Source: Author

**Results after training:**

The outcome of the adjustments made to the 3D Generative Adversarial Network (3DGAN), resulting in two generated models with precise and aesthetically pleasing geometry that closely mirror specific models from the dataset, presents both a success and a complication. This situation underscores the capability of the enhanced network architecture to capture and reproduce complex 3D geometries with high fidelity. However, it also highlights a persistent issue of overfitting, where the network does not generalize beyond replicating existing data, but instead directly duplicates models from the training set.

**High Fidelity Reproduction:** The adjustments to increase the learning rate, double the training epochs, and add layers to the network architectures have clearly enhanced the network's ability to process and reproduce the intricate details of 3D models. This indicates that the network has sufficient capacity and is effectively utilizing its deeper architecture to capture detailed features.

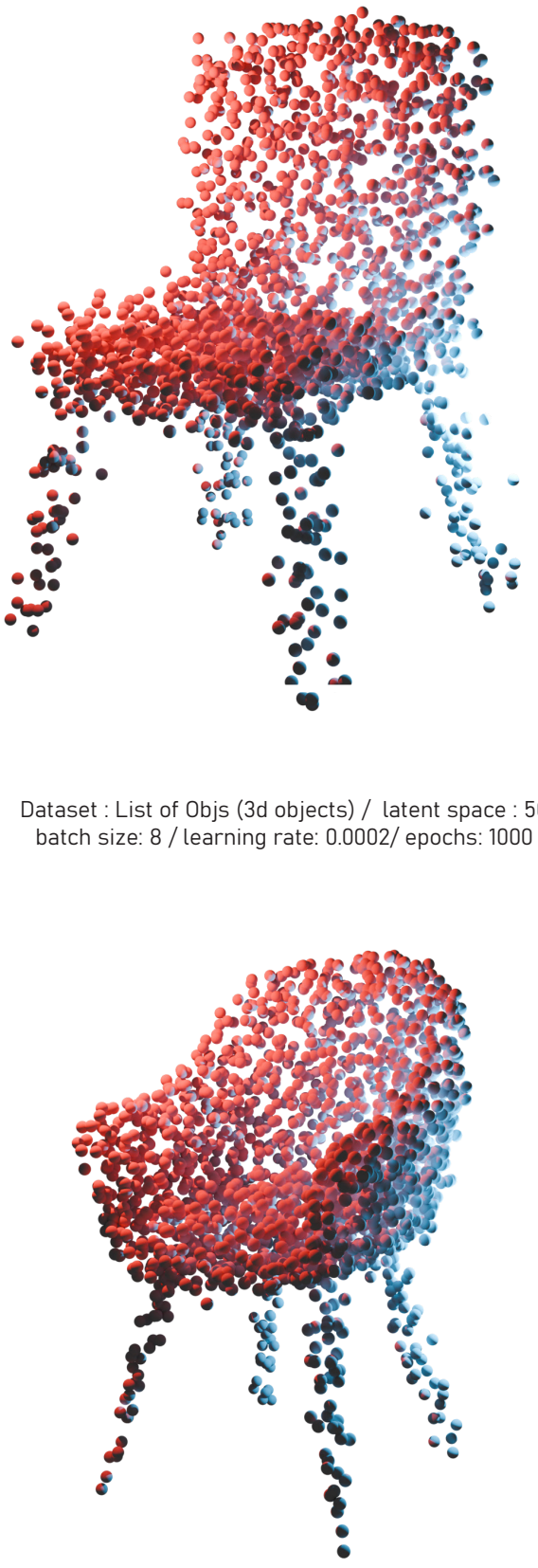
**Lack of Generalization:** The precise replication of models from the dataset as output suggests that while the network has become very effective at learning from the available data, it is primarily memorizing it rather than learning to generate new, unique structures. This is indicative of overfitting, where the network's predictions are tailored to the training data and do not generalize to producing new content.

**Suggestions for optimization**

To enhance the performance of the 3D Generative Adversarial Network and mitigate issues like overfitting, several targeted adjustments can be made:

**Regularization Techniques:** Implementing dropout and L2 regularization within the network can help reduce complexity and promote simpler, more generalized models that do not overfit to the training data.

**Data Augmentation:** Increasing the diversity of the training data through advanced augmentation techniques such as transformations can provide new perspectives and variations for the network to learn, aiding in generalization.



Dataset : List of Objs (3d objects) / latent space : 50
batch size: 8 / learning rate: 0.0002/ epochs: 1000

Figure 7: The 2 models generated by model B
Source: Author

**Early Stopping:** Using an early stoppingmechanism can prevent the network from overtraining by halting the training process when validation performance plateaus or declines, preserving the model's ability to generalize.

**Learning Rate and Epoch Adjustments:** Modifying the learning rate dynamically during training or employing a schedule that decreases the rate over time could prevent rapid convergence on memorized solutions. Additionally, reducing the number of training epochs may help avoid overfitting.

**Enhanced Sampling Strategies:** Sophisticated sampling strategies that weight the likelihood of selecting training samples can encourage the network to explore less common features and scenarios, enhancing generalization.

**Robust Validation Set:** Using a diverse and representative validation set can better assess the network's generalization capabilities and help fine-tune training parameters effectively.

**Model C**
**Dataset: Point Clouds**

**Applied Modifications**

This time dropout layers were included in both the Generator and Critic networks for a significant addition, designed to combat overfitting. By randomly deactivating 30% of neurons during training (dropout rate of 0.3), these layers help ensure that the networks do not depend too heavily on any specific neuron. This promotes redundancy within the network, enhancing its ability to generalize rather than memorize the training data.

**Customized Loss Functions**

The adaptation of loss functions to suit specific model training requirements involves employing an L1 loss for the Generator and a mean squared error (MSE) loss for the Critic. The L1 loss is aimed at minimizing the absolute differences between the generated outputs and the real data, encouraging precise replication of features. The MSE loss used by the Critic, typically employed in regression tasks, should effectively aid in distinguishing real from fake data.

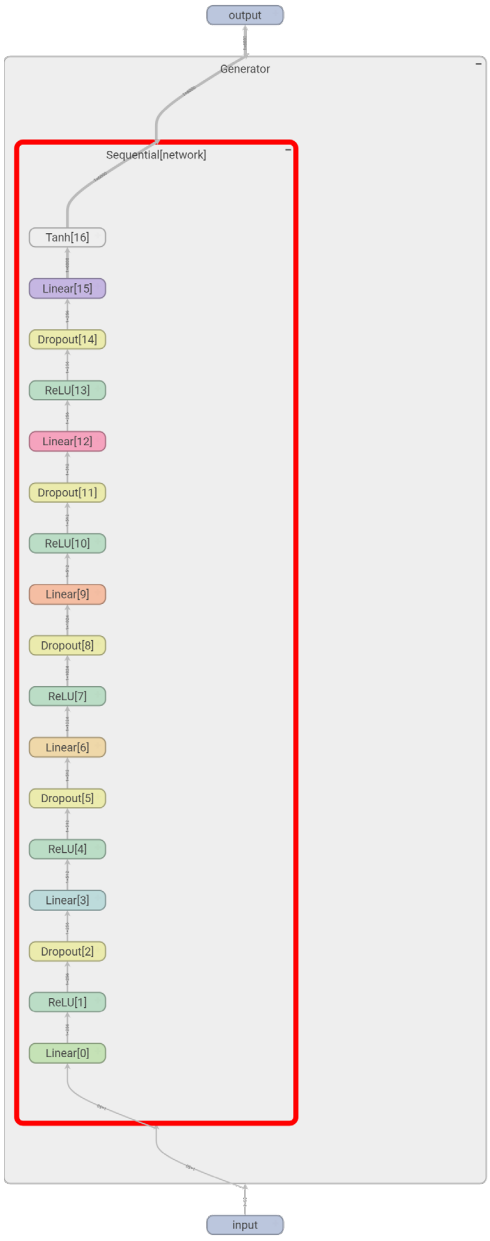



Figure 8: Implementing Dropout
Source: Author

**Training Dynamics**

The training parameters such as the number of epochs and the learning rate, if adjusted as previously discussed, need to be aligned with the increased complexity of the network. These parameters are critical in ensuring that the network training is thorough without leading to overtraining, especially in light of the additional layers and dropout regularization.

**Implications of Network Adjustments**

The dropout implementation directly addresses previous issues of overfitting where models generated were replicating training data too closely. By increasing the network's complexity, there is a potential improvement in the diversity and realism of the generated outputs. However, this increased complexity also raises the risk of overfitting, making the introduction of dropout layers a crucial countermeasure.

**Results after training:**

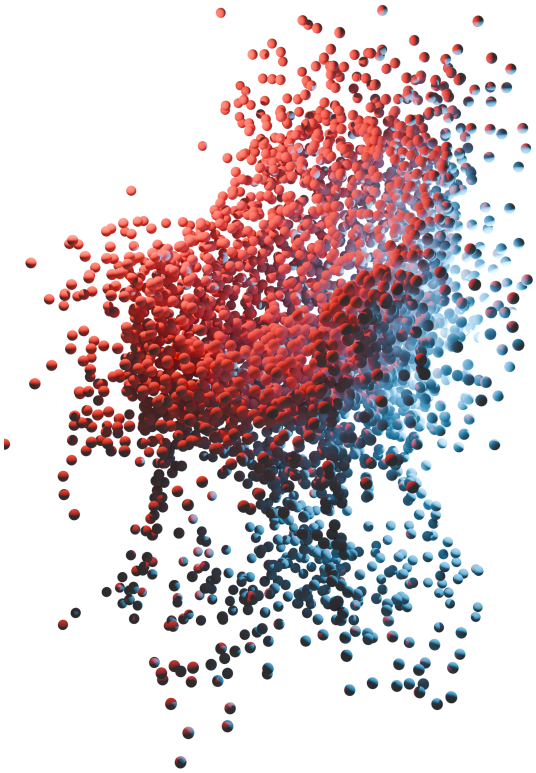
The observed issues in the outcomes of the modified 3D Generative Adversarial Network (3DGAN) script, particularly where both models produced are not only identical to each other but also to one from the dataset, suggest several underlying problems related to the training dynamics and network configuration. These challenges pinpoint key areas of concern and potential adjustment to improve the network's output diversity and learning efficacy.

**Potential Causes**

One significant issue could be that the Critic is overpowering the Generator. This imbalance occurs when the Critic becomes too proficient at distinguishing between real and generated images, limiting the gradient variety available for the Generator. This restriction can force the Generator into settling for producing a narrow range of outputs that it deems safe or effective based on its limited feedback from the Critic. Another contributing factor might be the lack of diversity in the training data. If the data pool or the sampling mechanism does not expose the Generator to a wide array of examples, it will struggle to learn and produce diverse outputs. Additionally, the current network architecture or the chosen parameters could be suboptimal. If the depth, width, activation functions, or dropout settings do not align well with the needs of diverse data learning, or if parameters like learning rate are improperly set, the network might converge too quickly towards suboptimal and non-diverse solutions.

**Solutions**

Addressing the overpowering of the Generator by the Critic is critical. This can be achieved by adjusting the training protocol of the Critic,



Dataset : List of Objs (3d objects) / latent space : 50
batch size: 8 / learning rate: 0.0002 / epochs: 1000

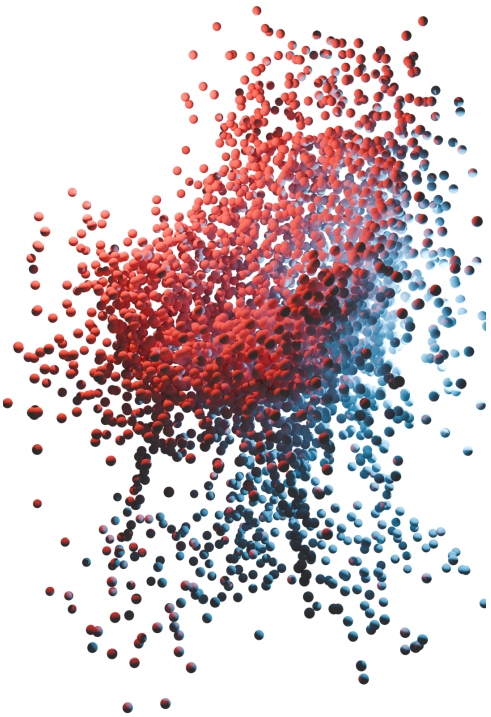


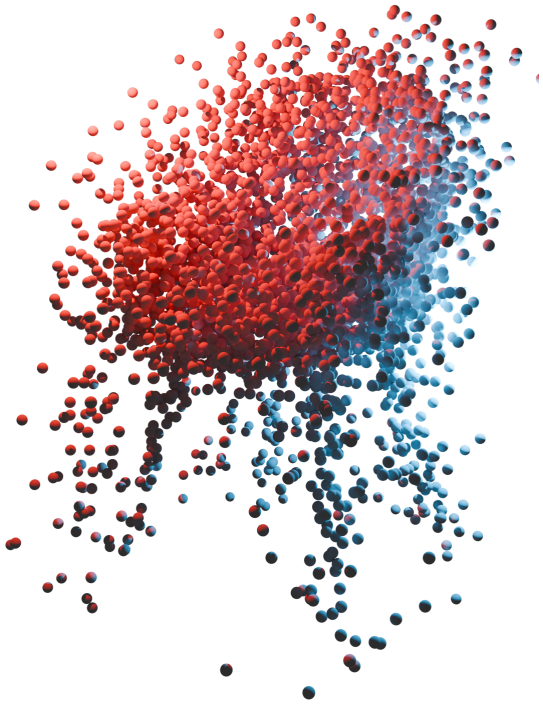
Figure 9: The 2 models generated by model C
Source: Author

either by reducing its training frequency relative to the Generator or tweaking its learning rate. Such changes could enable the Critic to provide richer and more varied feedback, which would allow the Generator to explore and adopt a wider variety of solutions. Enhancing the diversity of input data through robust data augmentation techniques and ensuring diverse and representative sampling during training could also expose the Generator to a broader spectrum of data features. This exposure is crucial for fostering the Generator's ability to learn and produce varied output patterns.

Incorporating more regularization techniques within the Generator or introducing penalties for a lack of diversity in outputs might encourage a broader production range. Techniques such as minibatch discrimination or feature matching can be particularly effective in this regard, as they promote diversity directly within the training process. Furthermore, a thorough reevaluation of the network architecture might be necessary. Ensuring that the network has the appropriate capacity to learn and generalize across various aspects of the data without becoming overly complex is essential. In some cases, simplifying or restructuring the network might lead to better diversity outcomes.

**Model C2**
**Dataset: Point Clouds**

Before moving on to different types of architecture to try to reach the goal set in the beginning, hyperparameter tuning took place to see if all this time it was mostly an issue in the training. Batch size was decreased back to 4 since the dataset isn't that large to begin with so to avoid the trainning looping over and over again looking at the whole dataset at one batch, the batch size was decreased to focus at smaller batches as an attempt to force the generator look into detailed depictions of the geometries. Also at the same the learning rate was decreased by a small margin but in the end the model again generated 2 identical model at the same time. At this point it was clear that it was time to rethink our script stucture, what kind of data it's using and how.



Dataset : List of Objs (3d objects) / latent space : 50
batch size: 4 / learning rate: 0.0001 / epochs: 1000

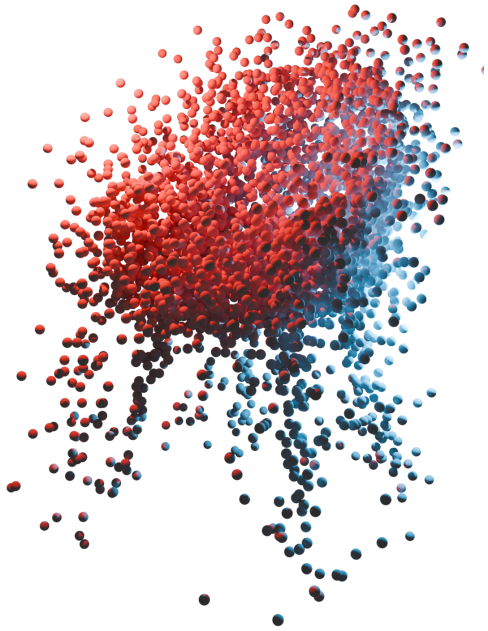


Figure 10: The 2 models generated by model C2
Source: Author

**Model D**
**Dataset: Point Clouds**
**Notes: KNN and EdgeConvolution**

In this approach, the attemt is to try to implement functions that would maybe helo the generator to understand better and in a more accurate way the distribution of the points in the 3D space.

**K-Nearest Neighbors (KNN)**

K-Nearest Neighbors (KNN) is a fundamental algorithm used in both supervised and unsupervised machine learning. In the context of 3D point clouds and neural networks, KNN is employed to calculate the closest neighbors of each point within a point cloud. This method involves measuring distances between points and finding the 'k' nearest points for each point of interest. KNN is crucial for tasks that require understanding the local structure within data, such as classifying points based on the features of their nearest neighbors or, as in the case of 3DGANs, enhancing local feature extraction which is vital for accurately modeling complex geometric structures.

**EdgeConv**

EdgeConv is a type of convolutional operation designed specifically for graph data, which can be applied to non-Euclidean data like point clouds. In a typical convolutional neural network used on images (Euclidean data), the convolution operation aggregates features from a local neighborhood defined by a kernel size. For point clouds, which inherently lack a regular grid structure, EdgeConv defines neighborhoods based on the nearest neighbors (using KNN), creating edges between each point and its neighbors. The EdgeConv operation then computes features not just based on individual points but also on the edges (relationships) between a point and its neighbors, making it highly effective for capturing local geometric structures within the point cloud. This ability makes EdgeConv particularly beneficial for tasks that require detailed understanding and processing of the spatial relationships in 3D data.

**Dynamic Graph CNN (DGCNN)**

The Dynamic Graph CNN, or DGCNN, is an advanced neural network architecture that extends the concept of EdgeConv to a series of layers, allowing for dynamic updating of graph structures at each layer of the network. In DGCNN, the graph is not fixed but is updated dynamically as the data flows through the network. This dynamic updating allows the network to adaptively learn the most relevant neighborhood (graph) structure at different layers, based on the features learned in previous layers. Each layer in a DGCNN can potentially learn a different set of edges, leading to a highly flexible and powerful representation that can adapt to complex and varying patterns in data like 3D point clouds. DGCNNs are particularly useful for processing point cloud data for tasks such as classification, segmentation, and, as in your case, generating detailed and diverse 3D models through a generative adversarial approach.

**Enhanced Neural Network Architectures**

The integration of PyTorch3D for efficient K-Nearest Neighbors (K-NN) computation marks a significant upgrade, facilitating precise distance and index calculations for point cloud data, essential for learning complex geometric forms. The introduction of the EdgeConv and DGCNN modules into the network architecture further refines this approach. EdgeConv layers process edge features by utilizing neighboring point features, significantly enhancing the local feature extraction capabilities. This method is crucial for capturing intricate local structures within the point clouds, thereby enabling the generation of highly detailed and realistic outputs. Building on this, the DGCNN module incorporates multiple EdgeConv layers to progressively refine these features, enhancing the robustness of feature representation.

**Optimization and Loss Functions**

The adoption of specific weight initialization routines standardizes the starting weights across the network, aiming to stabilize training outcomes and avoid poor local minima. Moreover, incorporating the Chamfer Distance as a loss function is pivotal in ensuring that the generated point clouds do not just appear realistic but also align geometrically with real data. This distance measures the similarity between generated and actual point clouds, driving the Generator to minimize these distances, thereby directly addressing the mode collapse issue by encouraging the production of varied and geometrically accurate outputs.

**Implications of These Changes**

These significant upgrades in the network's capabilities are tailored to enhance the geometric fidelity of the models produced by the 3DGAN. By focusing on localized feature processing through EdgeConv layers and ensuring geometric accuracy via the Chamfer Distance, the network is now better equipped to produce diverse and realistic 3D models, which is vital for applications requiring high-detail and precision. These changes specifically aim to counteract the tendency of the Generator to produce identical models, thereby addressing the mode collapse problem and promoting a broader range of output variability.

In summary, the modifications made to the 3DGAN script reflect a concerted effort to enhance the network's ability to generate detailed and diverse 3D point clouds effectively. The incorporation of advanced neural architectures and the optimization of loss functions, specifically designed to meet the unique demands of 3D data processing, mark a significant advancement in the development of generative models capable of handling complex structures. These strategic enhancements not only improve the network's performance but also crucially address the previous issues of mode collapse, where the Generator was limited to producing identical outputs.

Figure 11: The model generated by model D
Source: Author

**Results of training**

The results of the updated 3D Generative Adversarial Network (3DGAN) script, which employs advanced architectures such as KNN, EdgeConv, and DGCNN, leading to the generation of a point cloud that appears dispersed and lacks any discernible object-like structure, indicates that there are underlying issues in how the network processes and generates the data. This outcome suggests several potential areas of concern within the training process, the architecture setup, or the operational dynamics of the network that need addressing to refine the model's output.

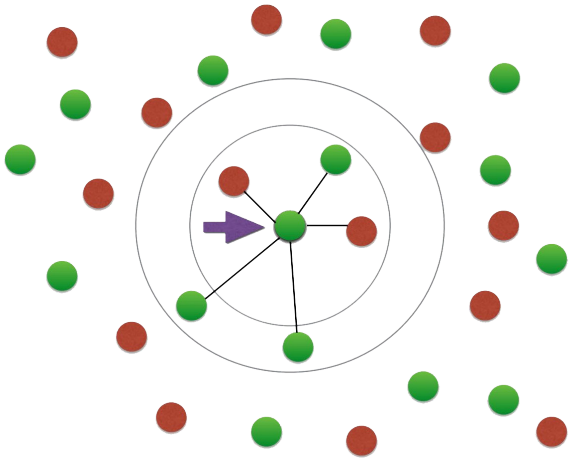


Figure 12: Edge Convolution
Source: Author

**Possible Causes for the Dispersed Point Clouds**

**Ineffective Learning Dynamics:** The network may not be learning effective feature representations necessary for generating coherent structures. This could be due to inadequate tuning of the network parameters or insufficient training time, preventing the network from converging to a useful solution.

**Loss Function Inadequacies:** If the loss functions used (such as Chamfer Distance in this context) are not effectively guiding the network towards generating realistic and coherent outputs, the result could be a failure to minimize the distance in a way that promotes the generation of recognizable objects. This might be due to the scaling of the loss terms or how they are integrated within the training loop.

**Network Architecture Mismatch:** There could be a mismatch in the architecture design, particularly with the EdgeConv and DGCNN layers, which might not be optimally configured to capture and propagate the necessary spatial hierarchies or local-to-global features that are characteristic of object-like structures in point clouds.

## A DIFFERENT DATA TYPE AND A DIFFERENT ARCHITECTURE

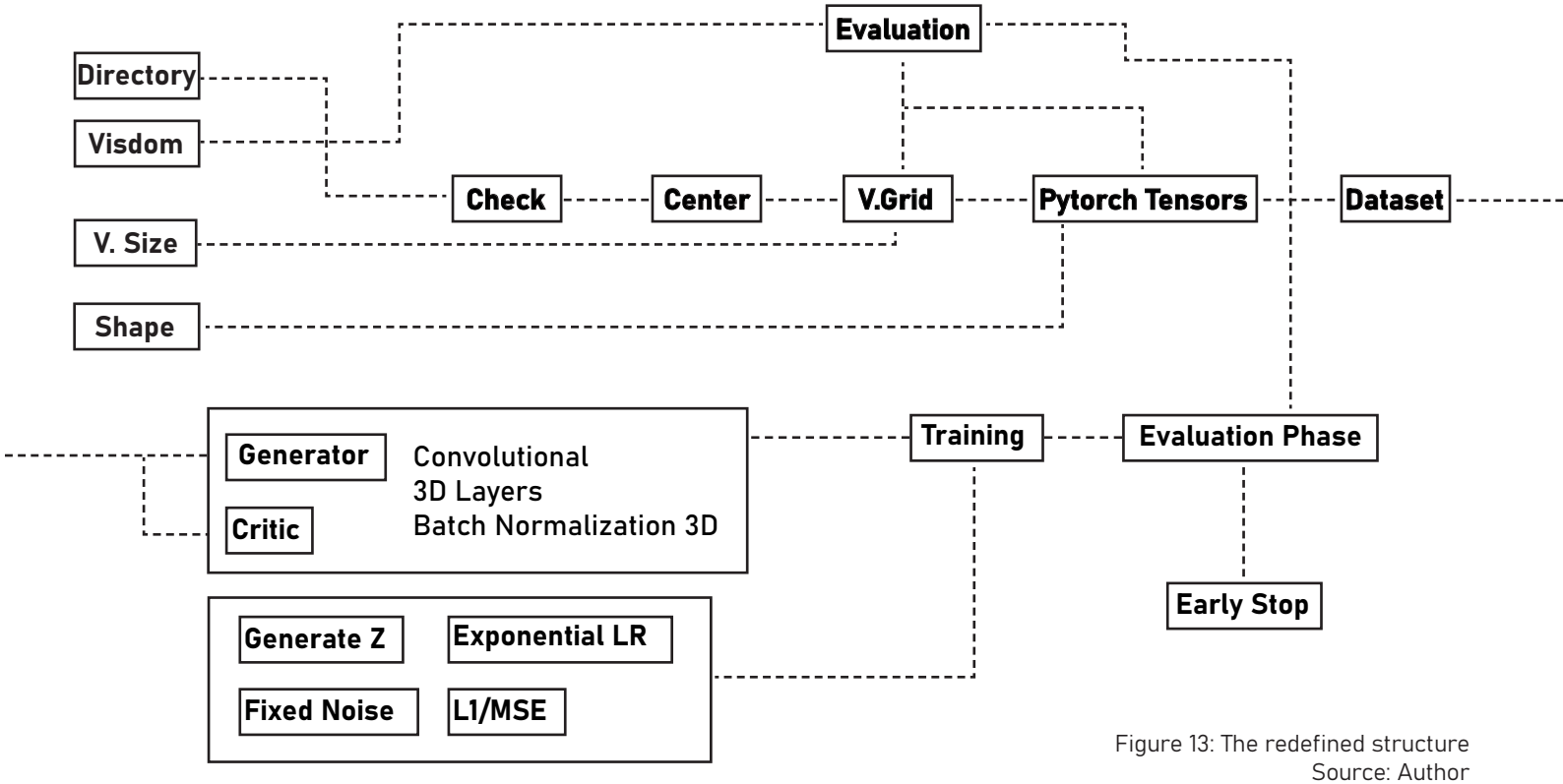



Figure 13: The redefined structure
Source: Author

**Model E**
**Dataset: Voxel Grids**

To address the progression and rationale behind the shift from using point clouds to voxel grids in your 3D Generative Adversarial Network (3DGAN) architecture, we'll begin by discussing the fundamental differences and benefits of voxel grids over point clouds. Following this, I'll analyze the fourth script, which you indicated as the final working version, and highlight what differentiates it from the previous three attempts.

**Advantages of Voxel Grids over Point Clouds**

**Structural Regularity:**

Voxel grids offer a structured, grid-like spatial arrangement, unlike point clouds which are unordered and unstructured. This regularity simplifies the processing with convolutional neural network (CNN) architectures, which excel at capturing spatial hierarchies in data through standard convolutional operations.

**Ease of Processing:**

Because voxel grids align well with traditional 3D convolution operations used in CNNs, they are inherently easier to integrate into many existing deep learning frameworks which are optimized for grid-like data structures (similar to handling 2D pixels in images).

**Improved Feature Learning:**

The structured nature of voxel grids allows for more effective and efficient learning of local and global features. Each voxel's neighborhood is explicitly defined, which aids CNNs in extracting and learning representative features from the structured data.

**Better Handling of Sparsity:**

Voxel representation can handle varying densities, which is often a challenge with point clouds. Sparse regions can be explicitly modeled and processed differently from dense regions, providing a more nuanced approach to handling real-world data variability.

**Compatibility with Up-sampling and Down-sampling:**

Operations like pooling and transposed convolutions are more straightforward and more effective with voxel grids, facilitating operations like feature aggregation and resolution enhancement, which are crucial for generative tasks.

**Script Analysis**

**The script is mainly divided in these sections:**

**Data Preparation:**

Transforming raw 3D data into voxel grid format which involves quantizing the continuous geometric space into discrete grid cells. This step ensures that the 3D shapes are represented in a manner that is amenable to processing by 3D CNNs(PyTorch tensors were used for data management).

**Network Architecture:**

Employing a CNN-based architecture tailored for voxel grid inputs. This would include layers capable of handling three-dimensional data directly, such as 3D convolutions, 3D pooling layers, and 3D transposed convolutions for the generative components of the network.

**Loss Functions:**

Incorporating loss functions that not only measure the realism of the generated outputs against actual data but also perhaps encourage spatial coherence and penalize structural anomalies in the generated voxel grids.

**Training Process:**

Using a training regimen that leverages the adversarial setup, where the generator tries to produce realistic voxel-based outputs that are indistinguishable by the discriminator from real voxel grid samples. This process is iteratively refined through gradient updates based on the performance of the discriminator.

**Differences from previous setups**

The shift to voxel grids as highlighted in the fourth script provides a fundamental change in how data is processed and generated compared to the earlier point cloud-based approaches. This change likely addresses previous challenges related to unstructured data processing, such as the difficulty in capturing and generating coherent spatial structures and the inefficiency in processing due to the irregularity of point clouds.

Moreover, the use of voxel grids aligns with more traditional and well-understood methods in 3D image processing, potentially

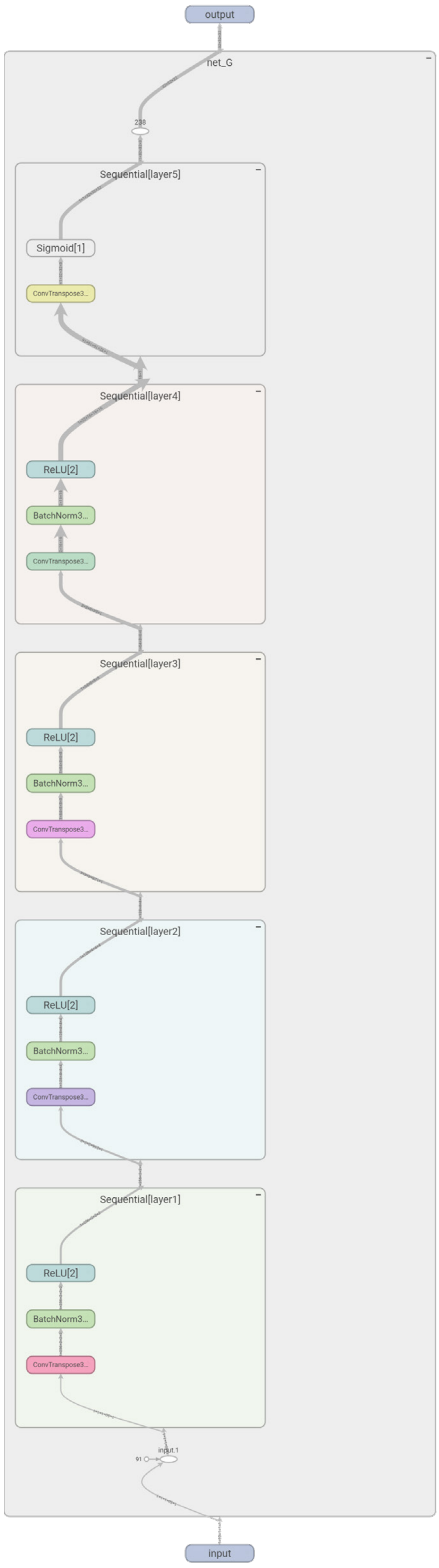



Figure 14: Sequential Layers
Source: Author

reducing the complexity of the network and the training process while improving the stability and quality of the generated outputs.

In summary, the progression to using voxel grids in the final script represents a strategic pivot towards leveraging the structured nature of voxel data to enhance the capability and performance of the 3D-CGAN, facilitating more robust, stable, and high-quality generation of 3D shapes. This approach likely offers a significant improvement over previous attempts, aligning better with conventional 3D processing techniques and potentially offering a more streamlined and effective generative process.

**Visdom and Tensorboard**

Both tools are aimed at enhancing the model development experience by allowing the user to monitor the training process more closely and adjust parameters as needed based on real-time feedback. visdom is typically used for more interactive, immediate visualizations, whereas TensorBoard provides a more structured approach to logging and viewing data over time, which is useful for long training sessions and comparing different training runs. Specifically in our case, the script uses visdom to evaluate the generation during the training process and the tensorboard gives us many logs/graphs to understand relations of functions inside the training loop.

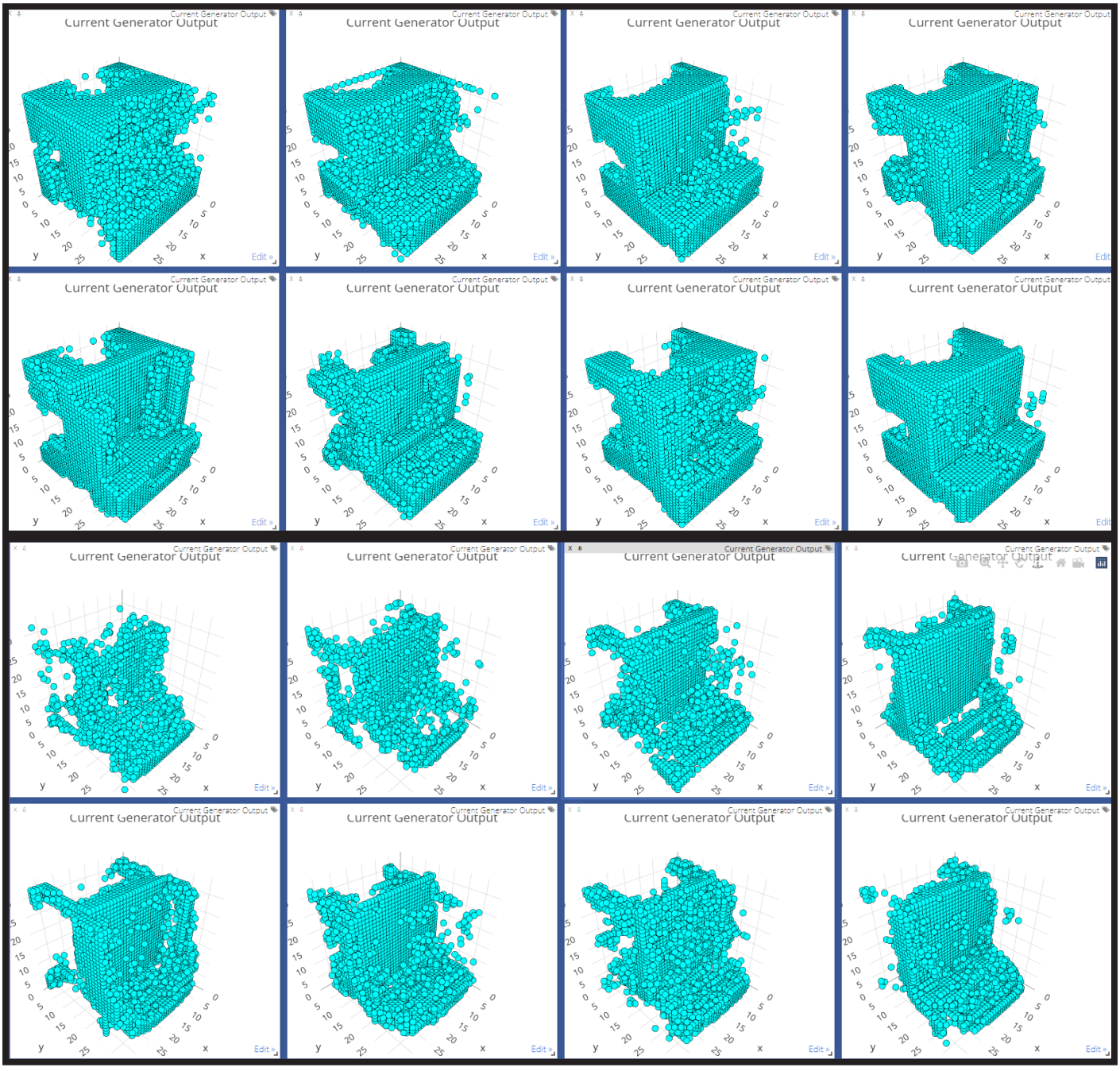



Figure 15: Visdom Evaluation Phase
Source: Author

## EVALUATING PERFORMANCE

Generator Loss Graph:

The generator loss starts high and sees a sharp drop, after which it begins to fluctuate around a value close to 1. This pattern suggests that while the generator is learning and improving its ability to produce realistic outputs at the start, it eventually reaches a point where it doesn't improve much further. This could be due to several reasons:

Nash Equilibrium: It's possible that the generator and discriminator are in a state of equilibrium where both are equally matched, causing the loss to oscillate.

Mode Collapse: The generator could be experiencing mode collapse, where it produces a limited variety of outputs, and therefore, can't minimize its loss further.

Overfitting: The generator might be overfitting to certain features of the training data, failing to generalize and fool the discriminator with new, varied outputs.

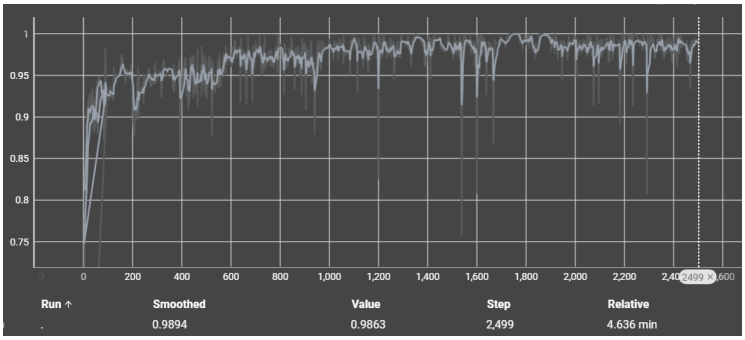



Figure 16: Generator Loss Graph
Source: Author

Discriminator Loss Graph:

The discriminator's loss also demonstrates a significant reduction initially but then shows more variance and tends to settle around a value slightly above 1. This behavior could mean:

Adaptive Discriminator: The discriminator is learning and adapting to the generator's outputs but still struggles to discern perfectly, indicated by the fluctuations.

Learning Pace: The fluctuations might be caused by the pace at which the discriminator learns compared to the generator, and adjusting learning rates or using different optimization techniques might help.

Competitive Dynamics: The two networks are in a competitive dynamic where each improvement by one network leads to adjustments in the other, resulting in the observed fluctuations.

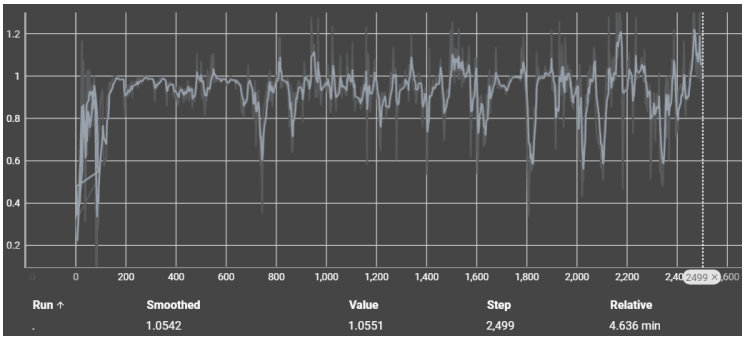



Figure 17: Discriminator Loss Graph
Source: Author

**General Evaluation**

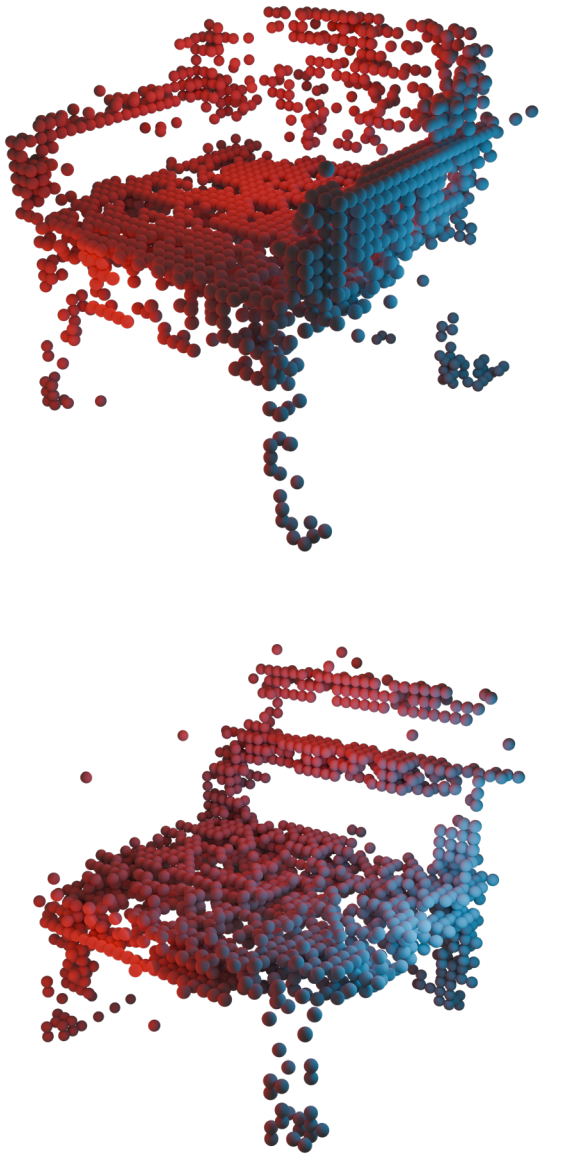
The fluctuations in loss for both networks suggest that the learning rates may be set too high, preventing the losses from converging to a minimum. Adjusting the learning rates could potentially lead to more stable training. The batch size is another factor that can significantly influence the stability of GAN training. Altering the batch size may result in more stable gradients and updates that are less prone to noise.

With respect to the network architecture, if the loss is not decreasing satisfactorily, this could indicate that the architecture may not be sufficiently complex to capture the data distribution, or conversely, it may be overly complex, leading to overfitting. It could be beneficial to explore alternative adversarial loss functions, especially if the current loss functions (MSE for the discriminator and L1 for the generator) are not yielding the desired results.

Incorporating regularization techniques such as spectral normalization or introducing noise to the inputs of the discriminator may improve training stability. As for the training duration, while it is important to train GANs for an adequate number of epochs to allow for convergence, there comes a point where additional training without modifications to the architecture or hyperparameters will not lead to further improvements.

**Conclusion**

The model seems to be on the right track as it's producing distinct models from different latent vectors. This is a positive sign that it's learning to generate variety, which is important in GAN training to avoid repetition and ensure richness in output. While this doesn't rule out all potential issues, it suggests that the network is picking up on how to diversify its creations, though there's still room to see if it can maintain this with a wider range of inputs. It's an encouraging step, but continued monitoring and fine-tuning would be essential to confirm and possibly improve its capability to generalize from the data it's trained on.



Epochs: 2500 / Batch Size: 4 / LR.G: 0.00035
LR.D: 0.00005 / Decay: 0.6 / Latent Space: 50

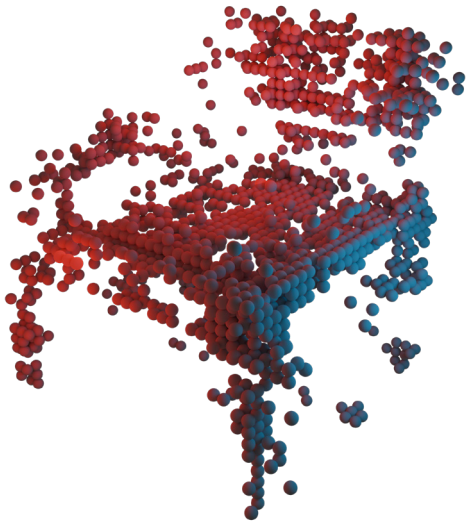


Figure 18: 3 unique models generated by model E
Source: Author

## THOUGHTS ON THE DEVELOPMENT

Embarking on the journey of developing E.P.I.C 3D proved to be a formidable challenge, especially with my foundational skills in Python and a nascent understanding of machine learning. The academic setting was a haven for growth, with professors offering essential materials that catalyzed my study, and yet, I found myself delving deeper into study and research than hands-on development. This approach, while demanding, aligns well with the academic ethos, where understanding takes precedence over creation.

In the course of the project, there were moments of retrospection, particularly with the DGCGAN component. Initially, it seemed like an appropriate choice while dealing with point clouds. However, it was the transition to voxel grids that truly marked a turning point for my work. Their tractable distribution in three-dimensional latent space streamlined development and became a pivotal element in the success of the project.

What ignited my enthusiasm was the realization that GANs in 3D design and engineering are still burgeoning fields, brimming with untapped potential. This insight was both exhilarating and grounding, serving as a reminder of the vast expanse of knowledge yet to be conquered.

However, a word of caution to those who might tread this path: if your Python skills are not yet robust, you may find yourself, as I did, in a relentless pursuit of knowledge to meet the soaring ambitions of your project. Hours of research and self-study became a routine, not just to keep pace but to truly comprehend the intricacies of such an advanced endeavor.

In the end, the rewards were immeasurably fulfilling. The experience of creating something so complex was an invaluable testament to the power of perseverance and passion. Looking forward, I'm eager to integrate the deep learning agent into the process to optimize the geometry. The fusion of generative capabilities and optimization is not just an academic exercise; it paves the way for A.I. tools to be a topic of industry discourse, where their potential is not limited to creation or enhancement, but a convergence of both.

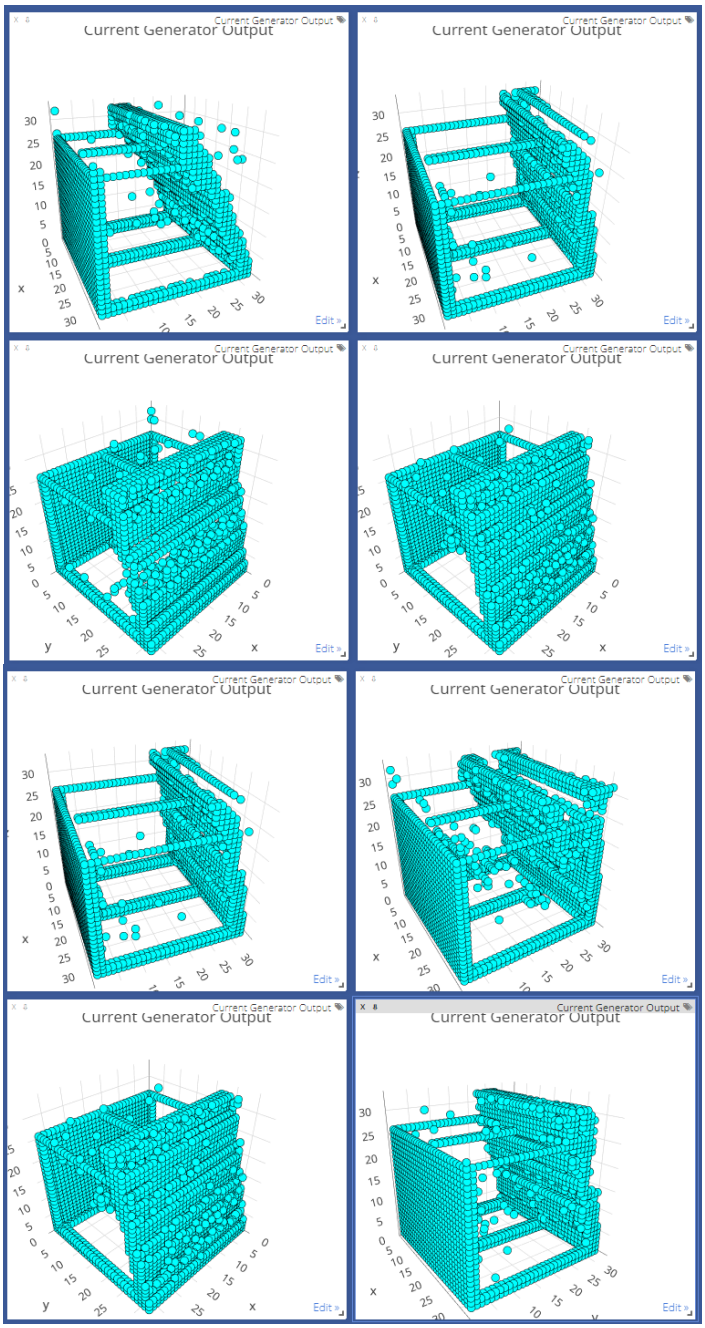



Figure 19: Testing with other types of objects
Source: Author

## REFLECTION ON THE COURSE AND THE WORKSHOPS

The course offered a thoughtfully crafted introduction to the realm of artificial intelligence in the built environment. There's an important point that I believe needs to be emphasized for future iterations: while individuals with prior Python expertise may find themselves at an advantage, the course is also designed as a springboard for beginners to kindle their passion for machine learning. It's crucial to clarify that the goal isn't necessarily to create a fully functional model; the course is about guiding each person's unique learning journey. From my observation, the absence of this clarification led to feelings of inadequacy and demotivation among some students, as they compared their burgeoning skills to those more seasoned in Python. This course should be celebrated as a conduit for personal growth, regardless of the starting point.

Regarding the workshops, the initial focus on probability and optimization was excellent. It provided a foundation for both novices and those already familiar with the relevant software and plugins to explore sophisticated and intelligent applications of these tools. The workshops and lectures that delved into supervised learning and the fundamentals of neural networks also followed a commendable structure.

Workshop content, starting from Probability & Statistics, laid down a strong foundation in core concepts like Estimation, Sampling, and Uncertainty Quantification, and traversed through Optimization's intriguing realms, featuring Topology and Parametric Optimization. The journey through these workshops highlighted the strategic application of these mathematical concepts using a variety of software, which included the use of Grasshopper plug-ins like Topos and Wallacei. These tools, paired with lectures, facilitated an understanding of how sophisticated AI tools could be applied within architectural design.

However, it became apparent that personal study is imperative, especially when dovetailing with the development of individual projects. This requirement posed a considerable challenge for a course accredited with 5 ECTS, as it demands a significant investment of time and effort. On a personal note, my own understanding experienced remarkable growth, which I attribute to a longstanding fascination with this field.

The latter parts of the workshops introduced unsupervised and reinforcement learning, which were informative but perhaps should be positioned merely as introductory for those new to the field. Students without prior experience would benefit from a recommendation to concentrate on supervised learning to ensure a focused and manageable scope for their projects and studies.

## EXTRA BIBLIOGRAPHY USED FOR SELF-STUDY

(this report doesn't contain directly text derived from other professional works)


1)Aalaei, Mohammadreza, et al. "Architectural Layout Generation Using a Graph-Constrained Conditional Generative Adversarial Network (GAN)." Automation in Construction, vol. 155, Nov. 2023, p. 105053, https://doi.org/10.1016/j.autcon.2023.105053. Accessed 20 Apr. 2024.

2)Kench, Steve, and Samuel Cooper. GENERATING 3D STRUCTURES from a 2D SLICE with GAN-BASED DIMENSIONALITY EXPANSION PREPRINT -FEBRUARY 16, 2021. 16 Feb. 2021.

3)Reesink, Thomas. Creating 3D Faces from 2D Images Using GANs 1.

4)Wu, Jiajun, et al. Learning a Probabilistic Latent Space of Object Shapes via 3D Generative-Adversarial Modeling. 2016.---.

5)"Learning Shape Priors for Single-View 3D Completion and Reconstruction." ArXiv.org, 13 Sept. 2018, arxiv.org/abs/1809.05068. Accessed 20 Apr. 2024.---.

6)MarrNet: 3D Shape Reconstruction via 2.5D Sketches. 2017.

# APPENDIX
# Python script of model E
# Analyzed in segments

## IMPORT LIBRARIES

```
import open3d as o3d
import os
import numpy as np
import torch
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from scipy.ndimage import zoom  # Import zoom function from scipy
import torch.optim as optim #for the Adam
import time #for the training loop as a time limit
from tqdm import tqdm
from torch.utils.tensorboard import SummaryWriter
import logging
import visdom #visualization
#from torch.optim.lr_scheduler import ReduceLROnPlateau

# Setting up TensorBoard
log_dir = r'D:\CI\Python Files\EPIC3D_3DCGAN_APRIL2024\TensorBoard'
writer = SummaryWriter(log_dir)
viz = visdom.Visdom() #start up a new session to visualize the eval phase of the training loops
model_save_dir = r"D:\CI\Python Files\try2\objects"
```

## DEVICE SETUP

```
#set device, be careful to detach later when necesary
device = 'cuda' if torch.cuda.is_available() else 'cpu'

#choose folder where all the objs are located
directory_path = r"D:\CI\3D LIBRARY\CHAIRS\chairobj"

#Choose initial voxel size for conversion from obj to voxel grids
voxel_size = 0.1
```

## RESHAPE

```
#Not Desirable, but since there are many errors and mismatches when the tensors are not uniformed,
#we set a target scale (cube-like)
target_shape = (32, 32, 32)  # Define target shape for all tensors

#First check if the file directory has obj files inside, then if those 3D objects have triangles,
#we can procceed or else we ignore them.
def load_and_check_meshes(directory_path):
    files_in_directory = os.listdir(directory_path)
    obj_files = [file for file in files_in_directory if file.endswith('.obj')]
    meshes_with_triangles = []
    for obj_file in obj_files:
        full_path = os.path.join(directory_path, obj_file)
        mesh = o3d.io.read_triangle_mesh(full_path)
        if not mesh.is_empty() and len(mesh.triangles) > 0:
            meshes_with_triangles.append(mesh)
        else:
            print(f"{obj_file} does not contain triangles.")
    return meshes_with_triangles
```

## TRIANGLE CHECK

```
#since sometimes the 3d files may have various sources use this definition to center all at the same origin
def center_mesh(mesh):
    centroid = mesh.get_center()
    mesh.translate(-centroid)
    return mesh
```

## CENTER MESH

```
#depending on the voxel size
def mesh_to_voxel(mesh, voxel_size):
    return o3d.geometry.VoxelGrid.create_from_triangle_mesh(mesh, voxel_size)
```

## CONVERT TO V.GRID

## PYTORCH TENSORS

```
#we convert to pytorch tensors to use later for training the GAN
def voxel_grid_to_tensor(voxel_grid, target_shape):
    voxels = np.asarray(voxel_grid.get_voxels())
    if len(voxels) == 0:
        return torch.empty(0)
    voxel_positions = np.array([voxel.grid_index for voxel in voxels])
    if voxel_positions.size == 0:
        return torch.empty(0)
    max_coords = voxel_positions.max(axis=0)
    tensor_shape = tuple(max_coords + 1)
    voxel_tensor = torch.zeros(tensor_shape, dtype=torch.float32)
    voxel_tensor[tuple(voxel_positions.T)] = 1
```

## RESCALE TENSORS

```
    # Rescale tensor to target shape
    scaling_factors = [target_shape[i] / voxel_tensor.shape[i] for i in range(3)]
    voxel_tensor_rescaled = torch.tensor(zoom(voxel_tensor, scaling_factors, order=0))  # Use nearest neighbor scaling

    # Print the final shape of the tensor
    print("Final tensor shape:", voxel_tensor_rescaled.shape)

    return voxel_tensor_rescaled
```

## VISUALIZATION AND ASPECT RATIO FIX

```
#the following definitions are to check if the conversion from obj to voxel grid and then to tensor went well
#added an aspect ratio fix cause sometimes the bounding box was off which is we weird
def visualize_tensor(tensor):
    fig = plt.figure(figsize=(10, 7))
    ax = fig.add_subplot(111, projection='3d')
    tensor_np = tensor.cpu().numpy()
    filled_voxels = np.argwhere(tensor_np > 0)
    ax.scatter(filled_voxels[:, 0], filled_voxels[:, 1], filled_voxels[:, 2])
    max_range = np.array([filled_voxels[:, 0].max()-filled_voxels[:, 0].min(),
                          filled_voxels[:, 1].max()-filled_voxels[:, 1].min(),
                          filled_voxels[:, 2].max()-filled_voxels[:, 2].min()]).max() / 2.0
    mid_x = (filled_voxels[:, 0].max() + filled_voxels[:, 0].min()) * 0.5
    mid_y = (filled_voxels[:, 1].max() + filled_voxels[:, 1].min()) * 0.5
    mid_z = (filled_voxels[:, 2].max() + filled_voxels[:, 2].min()) * 0.5
    ax.set_xlim(mid_x - max_range, mid_x + max_range)
    ax.set_ylim(mid_y - max_range, mid_y + max_range)
    ax.set_zlim(mid_z - max_range, mid_z + max_range)
    plt.show()

# Main process
meshes = load_and_check_meshes(directory_path)
centered_meshes = [center_mesh(mesh) for mesh in meshes]
voxel_grids = [mesh_to_voxel(mesh, voxel_size) for mesh in centered_meshes]
voxel_tensors = [voxel_grid_to_tensor(voxel_grid, target_shape) for voxel_grid in voxel_grids]t

#give the user the option, to choose one of the objects for evaluation
print("Hey! Choose one of the models for evaluation.")
for i, _ in enumerate(centered_meshes):
    print(f"{i}: Model {i + 1}")

try:
    model_index = int(input("Enter the model number: "))
    if 0 <= model_index < len(centered_meshes):
        print("Visualizing the voxel grid:")
        o3d.visualization.draw_geometries([voxel_grids[model_index]])
        print("Visualizing the tensor representation:")
        visualize_tensor(voxel_tensors[model_index])
    else:
        print("Invalid model number.")
except ValueError:
    print("Please enter a valid integer.")
```

```
#Generator Number of layers=5
class net_G(torch.nn.Module):
    def __init__(self, z_dim, cube_len, bias):
        super(net_G, self).__init__()
        # No need for self.args = args
        self.cube_len = cube_len
        self.bias = bias
        self.z_dim = z_dim
        self.f_dim = 32

        padd = (0, 0, 0)
        if self.cube_len == 32:
            padd = (1,1,1)

        self.layer1 = self.conv_layer(self.z_dim, self.f_dim*8, kernel_size=4, stride=2, padding=padd, bias=self.bias)
        self.layer2 = self.conv_layer(self.f_dim*8, self.f_dim*4, kernel_size=4, stride=2, padding=(1, 1, 1), bias=self.bias)
        self.layer3 = self.conv_layer(self.f_dim*4, self.f_dim*2, kernel_size=4, stride=2, padding=(1, 1, 1), bias=self.bias)
        self.layer4 = self.conv_layer(self.f_dim*2, self.f_dim, kernel_size=4, stride=2, padding=(1, 1, 1), bias=self.bias)

        self.layer5 = torch.nn.Sequential(
            torch.nn.ConvTranspose3d(self.f_dim, 1, kernel_size=4, stride=2, bias=self.bias, padding=(1, 1, 1)),
            torch.nn.Sigmoid()
            # torch.nn.Tanh()
        )

    def conv_layer(self, input_dim, output_dim, kernel_size=4, stride=2, padding=(1,1,1), bias=False):
        layer = torch.nn.Sequential(
            torch.nn.ConvTranspose3d(input_dim, output_dim, kernel_size=kernel_size, stride=stride, bias=bias, padding=padding),
            torch.nn.BatchNorm3d(output_dim),
            torch.nn.ReLU(True)
            # torch.nn.LeakyReLU(self.leak_value, True)
        )
        return layer

    def forward(self, x):
        out = x.view(-1, self.z_dim, 1, 1, 1)
        # print(out.size())  # torch.Size([32, 200, 1, 1, 1])
        out = self.layer1(out)
        # print(out.size())  # torch.Size([32, 256, 2, 2, 2])
        out = self.layer2(out)
        # print(out.size())  # torch.Size([32, 128, 4, 4, 4])
        out = self.layer3(out)
        # print(out.size())  # torch.Size([32, 64, 8, 8, 8])
        out = self.layer4(out)
        # print(out.size())  # torch.Size([32, 32, 16, 16, 16])
        out = self.layer5(out)
        # print(out.size())  # torch.Size([32, 1, 32, 32, 32])
        out = torch.squeeze(out)
        return out
```

## THE GENERATOR

```
#Generator Number of layers=5
class net_G(torch.nn.Module):
    def __init__(self, z_dim, cube_len, bias):
        super(net_G, self).__init__()
        # No need for self.args = args
        self.cube_len = cube_len
        self.bias = bias
        self.z_dim = z_dim
        self.f_dim = 32

        padd = (0, 0, 0)
        if self.cube_len == 32:
            padd = (1,1,1)

        self.layer1 = self.conv_layer(self.z_dim, self.f_dim*8, kernel_size=4, stride=2, padding=padd, bias=self.bias)
        self.layer2 = self.conv_layer(self.f_dim*8, self.f_dim*4, kernel_size=4, stride=2, padding=(1, 1, 1), bias=self.bias)
        self.layer3 = self.conv_layer(self.f_dim*4, self.f_dim*2, kernel_size=4, stride=2, padding=(1, 1, 1), bias=self.bias)
        self.layer4 = self.conv_layer(self.f_dim*2, self.f_dim, kernel_size=4, stride=2, padding=(1, 1, 1), bias=self.bias)

        self.layer5 = torch.nn.Sequential(
            torch.nn.ConvTranspose3d(self.f_dim, 1, kernel_size=4, stride=2, bias=self.bias, padding=(1, 1, 1)),
            torch.nn.Sigmoid()
            # torch.nn.Tanh()
        )

    def conv_layer(self, input_dim, output_dim, kernel_size=4, stride=2, padding=(1,1,1), bias=False):
        layer = torch.nn.Sequential(
            torch.nn.ConvTranspose3d(input_dim, output_dim, kernel_size=kernel_size, stride=stride, bias=bias, padding=padding),
            torch.nn.BatchNorm3d(output_dim),
            torch.nn.ReLU(True)
            # torch.nn.LeakyReLU(self.leak_value, True)
        )
        return layer

    def forward(self, x):
        out = x.view(-1, self.z_dim, 1, 1, 1)
        # print(out.size())  # torch.Size([32, 200, 1, 1, 1])
        out = self.layer1(out)
        # print(out.size())  # torch.Size([32, 256, 2, 2, 2])
        out = self.layer2(out)
        # print(out.size())  # torch.Size([32, 128, 4, 4, 4])
        out = self.layer3(out)
        # print(out.size())  # torch.Size([32, 64, 8, 8, 8])
        out = self.layer4(out)
        # print(out.size())  # torch.Size([32, 32, 16, 16, 16])
        out = self.layer5(out)
        # print(out.size())  # torch.Size([32, 1, 32, 32, 32])
        out = torch.squeeze(out)
        return out
```

## THE DISCRIMINATOR

## HYPERPARAMETERS

```
#Hyperparameters
epochs = 2500
batch_size = 4
d_lr = 0.00005
g_lr = 0.00035
beta1 = 0.6
beta2 = 0.999
z_dim = 50
cube_len = 32
leak_value = 0.2
n_critic = 1
bias = False
device = 'cuda' if torch.cuda.is_available() else 'cpu'
z_dis = "norm"
```

## INITIALIASATION

```
# Instantiate generator and discriminator
generator = net_G(z_dim=z_dim, cube_len=cube_len, bias=bias).to(device)
discriminator = net_D(cube_len=cube_len, leak_value=0.2, bias=bias).to(device)
voxel_tensors = [voxel_grid_to_tensor(voxel_grid, target_shape) for voxel_grid in voxel_grids]
tensor_dataset = torch.utils.data.TensorDataset(torch.stack(voxel_tensors))
```

## DATASET SPLIT

```
total_length = len(tensor_dataset)
train_length = int(total_length * 0.8)
validation_length = total_length - train_length
train_dataset, validation_dataset = torch.utils.data.random_split(tensor_dataset, [train_length, validation_length])
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
validation_loader = torch.utils.data.DataLoader(validation_dataset, batch_size=batch_size, shuffle=False)
```

## OPTIMIZERS

```
d_optimizer = optim.Adam(discriminator.parameters(), lr=d_lr, betas=(beta1,beta2))
g_optimizer = optim.Adam(generator.parameters(), lr=g_lr, betas=(beta1,beta2))
```

## LOSSES

```
criterion_D = torch.nn.MSELoss()  # Critic loss
criterion_G = torch.nn.L1Loss()  # Generator loss
```

## LOGGING

```
# Add model to TensorBoard
sample_z = torch.randn(1, z_dim, 1, 1, 1, device=device)
sample_voxel = torch.randn(1, 1, cube_len, cube_len, cube_len, device=device)
writer.add_graph(generator, sample_z)
writer.add_graph(discriminator, sample_voxel)
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger()
```

## LATENT VECTOR

```
def generateZ(args,batch):

    if   z_dis == "norm":
        Z = torch.Tensor(batch, z_dim).normal_(0, 0.33).to(device)
    elif z_dis == "uni":
        Z = torch.randn(batch, z_dim).to(device)
    else:
        print("z_dist is not normal or uniform")

    return Z
```

## VISDOM SETUP FOR VISUALIZATION

```
def visualize_voxels(voxels):
    # voxels is a binary numpy array of shape (D, H, W)
    voxels = np.pad(voxels, 1, 'constant', constant_values=False)  # Add padding to ensure edges are visible
    x, y, z = voxels.nonzero()
    points = np.column_stack((x, y, z))
    viz.scatter(
        X=points,
        opts=dict(
            title="Current Generator Output",
            markersize=5,
            markercolor=np.array([[0, 255, 255]]),  # red color
            xtickmin=0,
            xtickmax=voxels.shape[0],
            ytickmin=0,
            ytickmax=voxels.shape[1],
            ztickmin=0,
            ztickmax=voxels.shape[2],
            three_d=True
        )
    )
```

## GENERATING SAMPLES FOR VISDOM

```
# Function to visualize generated voxels using Visdom
def save_generated_samples(epoch, generator, fixed_noise):
    print(f"Generating samples for epoch {epoch}")
    with torch.no_grad():
        generated_voxels = generator(fixed_noise).detach().cpu()
    voxel_array = generated_voxels[0].numpy() > 0.1
    visualize_voxels(voxel_array)
    print("Sample visualization complete")
```

## DEFINITION FOR EXPORTING THE GENERATED MODEL

```
def voxel_to_point_cloud(voxel_grid):
    points = np.argwhere(voxel_grid)  # Extract the coordinates of active voxels
    point_cloud = o3d.geometry.PointCloud()
    point_cloud.points = o3d.utility.Vector3dVector(points.astype(float))
    return point_cloud

def save_point_cloud_as_ply(point_cloud, filename):
    o3d.io.write_point_cloud(filename, point_cloud, write_ascii=True)
    print(f"Saved point cloud to {filename}")
```

## TRAINING LOOP A

```
# Fixed noise for watching the progress of generated samples
fixed_noise = generateZ({'z_dim': z_dim}, batch_size)
def train_gan(generator, discriminator, train_loader, validation_loader, fixed_noise, n_critic=1):
    logger.info("Starting GAN Training")

    # Track start time
    start_time = time.time()

    for epoch in range(epochs):
        for phase in ['train', 'val']:
            if phase == 'train':
                generator.train()
                discriminator.train()
            else:
                generator.eval()
                discriminator.eval()
            data_loader = train_loader if phase == 'train' else validation_loader

            running_loss_G = 0.0
            running_loss_D = 0.0
            total_batches = 0

            for voxel_tensors in tqdm(data_loader):
                voxel_tensors = voxel_tensors[0].to(device)
                batch_size = voxel_tensors.size(0)

                noise = generateZ(z_dim, batch_size).to(device)

                ### Discriminator Training
                d_optimizer.zero_grad()
                real_labels = torch.ones(batch_size, 1, device=device)
                fake_labels = torch.zeros(batch_size, 1, device=device)

                # Train with real images
                real_output = discriminator(voxel_tensors)
                d_loss_real = criterion_D(real_output, real_labels)

                # Generate fake images and train
                fake_voxels = generator(noise)
                fake_output = discriminator(fake_voxels.detach())
                d_loss_fake = criterion_D(fake_output, fake_labels)

                # Combine losses and update discriminator
                d_loss = d_loss_real + d_loss_fake
                if phase == 'train':
                    d_loss.backward()
                    d_optimizer.step()

                ### Generator Training
                # Only update generator once per n_critic iterations of discriminator
                if total_batches % n_critic == 0:
                    g_optimizer.zero_grad()
```

```
                # It's important to calculate fake output again since the discriminator is updated
                fake_output = discriminator(fake_voxels)
                g_loss = criterion_G(fake_output, real_labels)
                if phase == 'train':
                    g_loss.backward()
                    g_optimizer.step()

            running_loss_G += g_loss.item() * n_critic  # Adjusted for n_critic scaling
            running_loss_D += d_loss.item()
            total_batches += 1

        epoch_loss_G = running_loss_G / total_batches
        epoch_loss_D = running_loss_D / total_batches

        logger.info(f"Epoch {epoch+1}/{epochs}, Phase: {phase}, Loss G: {epoch_loss_G:.4f}, Loss D: {epoch_loss_D:.4f}")
        if phase == 'train':
            writer.add_scalar('Loss/Train/Generator', epoch_loss_G, epoch)
            writer.add_scalar('Loss/Train/Discriminator', epoch_loss_D, epoch)
        else:
            writer.add_scalar('Loss/Val/Generator', epoch_loss_G, epoch)
            writer.add_scalar('Loss/Val/Discriminator', epoch_loss_D, epoch)

    # Outside of data loader loop - Check epoch for saving generated samples
    if (epoch + 1) % 100 == 0:  # Adjust as needed
        save_generated_samples(epoch + 1, generator, fixed_noise)
        with torch.no_grad():
            generator.eval()
            fixed_output = generator(fixed_noise)  # Assuming fixed_noise is a batch of latent vectors
            voxel_grid = fixed_output.cpu().numpy() > 0.5  # Threshold to create a binary voxel grid
            point_cloud = voxel_to_point_cloud(voxel_grid[0])  # Convert first item in batch
            save_point_cloud_as_ply(point_cloud, f'{model_save_dir}/generated_epoch_{epoch+1}.ply')
            # Update learning rates at the end of each epoch
            #d_scheduler.step(epoch_loss_D)
            #g_scheduler.step(epoch_loss_G)

# Save final model weights after training completes
torch.save(generator.state_dict(), os.path.join(log_dir, 'generator_final.pth'))
torch.save(discriminator.state_dict(), os.path.join(log_dir, 'discriminator_final.pth'))
writer.close()

end_time = time.time()
total_time = end_time - start_time
logger.info(f"Training completed in {total_time // 60}m {total_time % 60}s")


# Start training
train_gan(generator, discriminator, train_loader, validation_loader, fixed_noise)


device = 'cuda' if torch.cuda.is_available() else 'cpu'
z_dim = 50  # Assuming z_dim is 50 as per your setup
generator = net_G(z_dim=z_dim, cube_len=32, bias=False).to(device)

# Correct path to the model file
model_path = r'D:\CI\Python Files\EPIC3D_3DCGAN_APRIL2024\TensorBoard\generator_final.pth'

# Load the trained generator model
generator.load_state_dict(torch.load(model_path, map_location=device))
generator.eval()
```

**TRAINING LOOP B**

**EVERY 100 EPOCHS GENERATE A SAMPLE TO EVALUATE IN VISDOM**

**SAVE TO TENSORBOARD**