Bootstrapping Generative AI To Generating Their Own Training Data

Master Thesis by Mathias Scholz at IT University of Copenhagen

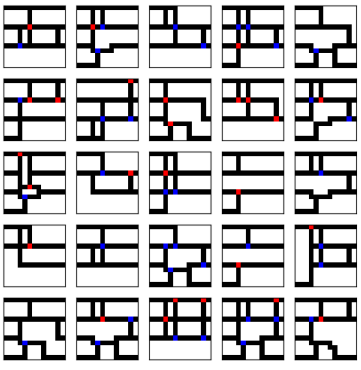
Submission June 2. 2025

Supervisor Sebastian Risi

Co-Supervisor Milton Llera Montero

Abstract

In this thesis, a method 'bootstrapping' is developed as a way of using generative AI to train themselves iteratively. Through a domain of circuit development, the ability of circuits to be successfully simulated was used as a criterion for synthetic samples, ensuring a minimal level of quality for generated training data. A categorical diffusion model is trained on an initial small dataset and then used to generate new samples, which in turn are added to the model's training data. Of the generated circuits, only the least complex circuits are kept in training data, gradually guiding the dataset toward simpler and more efficient representations. The bootstrapping method was capable of increasing an initial dataset of just 4 circuits to over 10.000 circuits over a 100 iterations of bootstrapping. The method successfully generated diverse samples exploring the circuit domain well, even when biases were induced into the model. As synthetic samples are added to the training data, features prevalent in the initial generations may compound into future generations, yet the bootstrapped model still managed to sample less represented features in subsequent generations. Bootstrapping significantly expands available training data while maintaining diversity, illustrating a viable path for generative model development in complex domains. The code written over the course of this thesis is available at: `https://github.com/MathiasBoi6/GenerativeCircuits`.



Contents

1 Introduction

Training neural networks on vast amounts of data has led to significant breakthroughs in numerous domains. Whether in Natural Language Processing tasks, Image Classification or other tasks, the deep learning approach has demonstrated remarkable success [Lim, Lee, and Jo 2025]. However, these approaches usually require large amounts of data, which is not always available, especially in niche domains.

In domains with no existing data, but where sampling is tractable, training data can be created through repeated sampling. Evolutionary methods and reinforcement learning use samples of the domain to update their state, often updating towards samples of specific properties, like those that maximize some value. But these approaches can lack in diversity and struggle when exploring especially large and sparse domains [Lehman and Stanley 2010; Lehman and Stanley 2011]. When the goal is to generate samples across the entire domain, rather than finding some optimal samples, another approach to finding these samples could be generative models. Generative models aim to learn a distribution over data and generate synthetic data that replicate features of the training data. By using these synthetic data points as samples of the domain, generative models can be used to sample from and explore the domain [Torrado et al. 2020].
Denoising Diffusion Models (DDMs) [Sohl-Dickstein et al. 2015, Nichol and Dhariwal 2021] are a class of generative neural networks that learn to transform a simple probabilistic distribution, into a distribution of a dataset. These models learn how to iteratively denoise Gaussian noise into samples of the data. This iterative approach to learning the transformation of a distribution helps make DDMs flexible allowing them to be applied for many domains [Friedman 2022, Friedman 2023, Ambrogioni 2023]. DDMs have in recent years [Lim, Lee, and Jo 2025] been gaining traction, but one problem with these models is that they tend to 'hallucinate' producing synthetic samples that have some distorted properties [Aithal et al. 2024]. By training such a DDM on a novel domain, could their ability to interpolate over data samples, and their tendency to hallucinate, be leveraged to iteratively extend a dataset with semantically consistent in-domain synthetic samples absent from the original data?

Augmenting training data with synthesized data has seen success in increasing model performance, but this has mostly been for strengthening the existing capabilities of the AI models [Wu et al. 2023, Trabucco et al. 2023, D. Zhang, Wang, and Luo 2024, Islam et al. 2024]. Extensive use of synthetic data in generative models has been found to lead to Model Autophagy Disorder, where generative models either lose their ability to generate diverse or high-quality samples [Alemohammad et al. 2024]. The loss of diversity comes from adding generated samples that are similar to the original samples to the training data, while the loss in quality comes from errors and imperfections in the synthetic data getting compounded as the model continues training on its generations.

Yet, in domains where evaluating the validity of the generation is tractable, this quality degradation could be mitigated by evaluating and filtering samples prior to their inclusion in the training data. This is possible in domains where the sample data must follow some rules, such that all data that follow these rules would be valid samples.

Low-level electronic circuits serve as the test domain for evaluating whether generative models can effectively be bootstrapped, to produce and expand their own training data. For any sample configuration of a circuit and the components on it, it is possible to simulate the circuit and get a reading for what signals it would emit. Whether the circuit emits a signal will be the verification requirement of the synthetic data, and that it is different enough from the existing training data. The aim is to determine whether this approach can produce novel and interesting circuits beyond those present in the training data.

2 Related work

2.1 Diffusion

Denoising Diffusion Models (DDMs) are a class of neural networks that implement a generative procedure via a denoising process [Sohl-Dickstein et al. 2015]. These models are trained to gradually remove noise in a sample, which in turn can be used to infer new samples from pure noise.

Training DDMs is done through a diffusion process. Noise is added incrementally to the data over a series of timesteps, forming a Markov chain where generating each noisy sample depends only on the previous sample. The model is trained to isolate noise in the samples, and having many timesteps enables it to learn a smoother denoising function, enhancing robustness at all steps.

To generate new samples, the model is given pure noise, and the predicted noise is subtracted from the given input. The Markov process through which noise was added, is inverted to get a function that given a noisy sample and the isolated noise can partially remove the noise, restoring the sample to the state of an earlier timestep. DDMs may not predict the exact noise correctly from the maximal timestep, instead, denoising is performed iteratively, partially denoising the samples at each iteration.
For Gaussian noise which is the most common method for diffusion, noise can be added through the function [Ho, Jain, and Abbeel 2020]:

$$q(x_t|x_{t-1}) = \mathcal{N}(x^{(t)}; x^{(t-1)}\sqrt{1-\beta_t}, \mathbf{I}\beta) \tag{1}$$

$q(x_t|x_{t-1})$ is the distribution of data at timestep t after adding noise to the distribution at timestep $t-1$. This value is computed as the standard Gaussian

distribution of $x^{(t-1)}\sqrt{1-\beta_t} \ \wedge \ \mathbf{I}\beta$, where β_t is a scalar value for the amount of noise added for timestep t, and x_0 is the raw sample.
The root is taken of the rate of change β to keep the variance of the distribution constant. As the timestep approaches the final step T the value of the distribution becomes $\mathcal{N}(x^{(t)}; 0, \mathbf{I})$, where x_T is distributed according to the standard normal distribution.

For efficient computation, the noise at a timestep t can also be calculated directly from x_0 through the cumulative product of the rate of change $\alpha_t = \prod_{t=1}^{T} 1 - \beta_t$ such that x_t can be computed directly as:

$$x_t = \sqrt{\alpha_t}x_0 + \sqrt{1-\alpha_t}\epsilon_t, \quad \epsilon \sim \mathcal{N}(0, \mathbf{I}) \tag{2}$$

In the denoising process, a sample x_{t-1} can be obtained by sampling from a Gaussian distribution whose mean is defined by subtracting the scaled predicted noise ϵ_θ from the current noisy sample x_t, with the scaling determined by the noise added in the step β_t.

$$\mathcal{N}\left(x_{t-1}; \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\alpha_t}}\epsilon_\theta(x_t.t)\right), \frac{1-\alpha_{t-1}}{1-\alpha_t}\beta_t\right) \tag{3}$$

2.2 Generative Bootstrapping

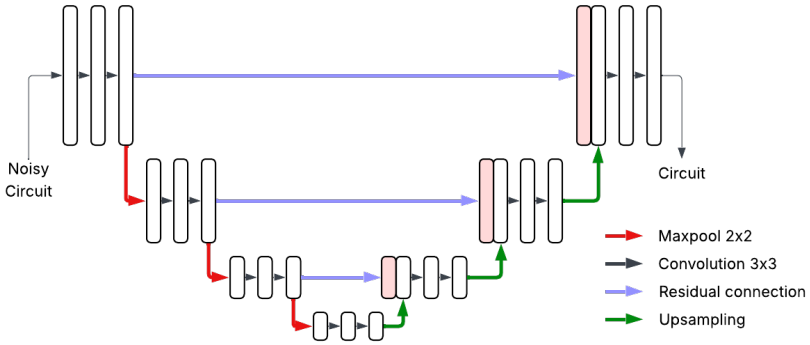


Figure 1: Illustration of U-Net. A neural network architecture that utilizes downsampling and upsampling to concentrate relevant information.

Circuits can be developed at multiple levels of abstraction, with prior work on generative modeling representing circuits as sequences of tokens [Li et al. 2024]. For this thesis circuits are instead represented as images, using Denoising Diffusion Models and U-Nets for the generative process [Ronneberger, Fischer, and

Brox 2015]. U-Nets, displayed in Figures 1, are a neural network architecture that consists of downsampling and upsampling layers. They concentrate input information into a concise latent space before unraveling the model's prediction. By concentrating the global features of the input data, convolutional layers can efficiently capture and propagate global and high-level information, while the remaining layers learn a gradual encoding and decoding of the data. The latent space serves as a compact representation for generation, conceptually similar to how a genome encodes the morphogenetic developmental process of an organism [Mitchell and Cheney 2025, Hartl and Levin 2025].

While any generative model capable of producing novel samples could support bootstrapping, the hallucinating tendencies displayed by DDMs serve as a potential driver for searing underrepresented areas of the circuit domain [Aithal et al. 2024]. Additionally, DDMs have been found to behave similarly to evolutionary processes, performing directed denoising towards the data distribution in a manner similar to directed selection [Y. Zhang et al. 2024, Hartl, Y. Zhang, et al. 2024]. The diversity of data that DDMs can describe and their flexibility in doing so make them particularly suited to open-ended exploration, where the goal is not just optimization but the continual discovery of novel, high-dimensional patterns. When unbounded, generative models can find many samples of ambiguous quality. However, by including a minimal criterion, restricting the model to learning only from samples relevant to the target domain, the quality of generated samples can be ensured [Lehman and Stanley 2010, Soros, Cheney, and Stanley 2016.

3 Method

3.1 Circuits

The circuits used in this project are abstracted from printed circuit boards, or breadboards. Specifically, these circuits are represented as images with a width and height of 13, and with every pixel having a discrete value between 0 and 3. These discrete values map to respectively 'empty', 'wire', 'AND-gate', and 'NAND-gate', and are one-hot encoded over the channels of the image, making for a 4-channel image [Murphy 2022]. For the logical components representing AND- and NAND-gates, each adjacent pixel corresponds to its inputs and output. The pixel to the left serves as one input, while the pixels directly above and below together represent the second input. The gate emits the signal it produces onto the pixel right of itself. Wires above and below a gate are considered to be connected, allowing a vertical connection through gates. The inputs connected to the circuits are placed at fixed points along the leftmost column of the circuits, and the outputs along the rightmost.

The gates selected for the circuits were purposely chosen to include only a few basic logic gates, with the intention of using the lack of gates as a po-

tential benchmark of the bootstrapping approach. All logic operators can be recreated from composing NAND-gates, and if the generative model infers any such operator, it would indicate that it has captured non-trivial logical behavior.

From a generated circuit, all combinations of input states can be simulated, and with this, the behavior of the circuit can be measured. By iterating over all possible input states, a truth table for the circuit can be generated. These truth tables are fed to the generative model, as embedding data for the model to learn the relationship between the truth tables and their related circuit.

3.2 Categorical Diffusion

For images in color, diffusion is typically done through Gaussian noise, with the Diffusion Denoising Model learning to predict the degree of noise in every pixel of every color channel [Ho, Jain, and Abbeel 2020. However, when modeling circuits, components are treated as discrete entities to enable straightforward simulation and interpretation. For this reason, categorical noise is more appropriate as it models discrete data. Preliminary experimentation showed promise using categorical diffusion, with Appendix A providing a simple demonstration of the generative process used in this project. To use categorical noise, components are one-hot encoded with each channel describing the probability of the class, similar to work done by [Austin et al. 2021, Hoogeboom et al. 2021]. With one-hot categorical values, the images fed to the diffusion model are always binary values. This restricts the input space to a finite set of valid states, potentially simplifying the learning task by eliminating the need to model continuous-valued distributions.

In a categorical diffusion model, noise is a combination of the chance that a pixel has been 'flipped', changed to another category, along with what the pixel has been changed to. Adding noise can be done by interpolating between a 100% chance of being its current value and a uniform probability per class. With a class-wise probability, the noisy image is generated by sampling a categorical distribution over the probabilities.

$$x_t = C\left(\alpha_t x_0 + \frac{(1-\alpha_t)}{K}\right) \tag{4}$$

A noisy sample x_t is computed directly from the clean sample x_0, by interpolating between itself and the uniform distribution of classes $1/K$. The result of this interpolation defines the class probabilities of a categorical distribution C from which x_t is then sampled.

Denoising Diffusion Models train to isolate the noise in noisy images, and then partially remove this noise from the image during denoising. However, a shortcut is employed in this project whereby the model directly denoises the input images, producing the fully denoised circuit in a single step [Song, Meng, and

Ermon 2022]. This process is performed iteratively, predicting the fully denoised sample from a timestep t, and re-adding noise to timestep $t-1$, progressively refining the prediction.

$$x_{t-1} = C\left(\alpha_{t-1}\epsilon_\theta(x_t, t) + \frac{(1-\alpha_{t-1})}{K}\right) \tag{5}$$

The model prediction $\epsilon_\theta(x_t, t)$ is used to generate the partially denoised sample x_{t-1} through re-adding noise to the predicted circuit. Figure 2 shows the diffusion process for a categorical diffusion model, as well as the denoised predictions of a trained model at every iteration of denoising.

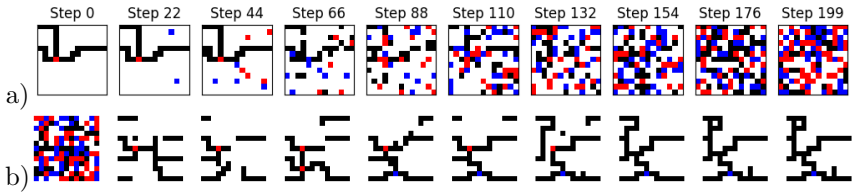


Figure 2: a) Categorical diffusion of a circuit over 200 steps. b) Generation from noise to circuit, showing pure noise on the left, and the predicted circuit at each step of denoising.

3.3 Conditioning

While bootstrapping is used to generate a large dataset of usable circuits, inferring new circuits with desired behavior requires conditioning the generative model [L. Zhang, Rao, and Agrawala 2023]. This is done by providing an encoding of the target behavior as an additional input, allowing the model to learn an association between behavior and circuit layout. The diffusion models trained in this project use conditioning on truth tables of every training sample. Truth tables describe the functional behavior implemented by a circuit, and by conditioning the model on this data, the model may learn what components and composition of wires lead to certain behaviors of circuits.

Conditioning is done by encoding the truth table associated with each circuit, into a vector, and then embedding that vector into the DDM during training. The truth tables used for these circuits are of fixed size, with 4 input and output columns and 16 rows for combinations of input states. The circuits may not make use of all the provided inputs, and therefore the entries of the truth table have 3 states of 'off', 'on', or 'not applicable'. When only some inputs are connected, a minimal truth table of 2^n rows is generated, corresponding to the possible combination of input states and then padded, marking the unused rows with 'not applicable'.

Each entry in the tables is first individually encoded from their state value.

With these encoded values, the rows of the table are encoded taking into account the position of each value. With the encoded rows, columns are encoded through a transformer encoder, using the rows and columns as keys and values, to generate the final embedding [Vaswani et al. 2017, Bahdanau et al. 2014, Arnab et al. 2021, Zhou et al. 2023].

3.4 Bootstrapping

To bootstrap a generative model, a small dataset is used to train the model, which is then used to generate new samples, augmenting its own dataset. To ensure high-quality samples and avoid the generative model degrading from compounding errors in generation, only synthetic samples that meet a minimally viable constraint are selected as new training data [Lehman and Stanley 2010]. The procedure is illustrated with Figure 3 showing the training process on the top and the generation process below, with a shared diffusion model between the two.

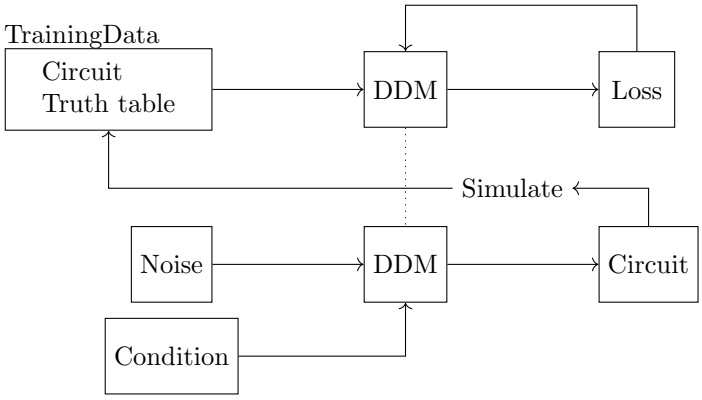


Figure 3: Bootstrapping setup. On top Denoising Diffusion Model (DDM) is trained with training data. Below DDM is used to generate new training samples from noise. Generated circuits are simulated and added to the training data.

With diffusion models, inference generates new samples that follow a learned data distribution [Ho, Jain, and Abbeel 2020]. By continuously augmenting the training data with these new samples, the training data may eventually cover the domain it represents. The function learned by the model to approximate the data distribution is often imprecise, particularly when training data is limited. This misrepresentation of the data leads to the model hallucinating features of the domain during generation [Aithal et al. 2024]. While hallucinated outputs are often undesirable in image generation, they can be advantageous in domains where low-quality generations can be filtered, as they may support broader exploration of the target domain.

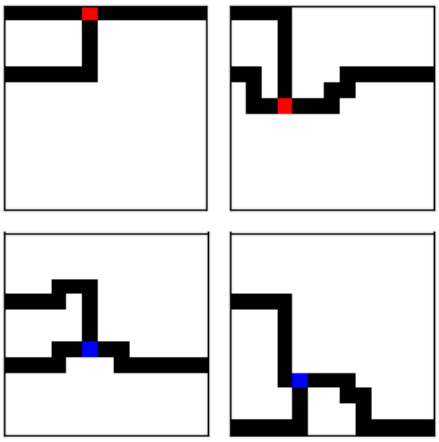


Figure 4: Initial circuit dataset containing 2 samples of AND-gates (red), and 2 samples of NAND-gates (blue)

The initial training data used in this project consists of the four circuits shown in Figure 4, along with their associated truth tables. These circuits are constructed manually, with 2 examples of using AND-gates and NAND-gates. Generation is done through training a categorical diffusion model on this training data and inferring from it. The samples inferred vary highly in quality, especially at the start when the dataset is very small. These circuits may contain unnecessary elements—such as wires or gates—that are either disconnected or have no impact on the intended logic. When such circuits are added to the training data, the model learns to replicate these non-functional elements and propagate them into future generations of inferred data [Aithal et al. 2024]. To alleviate the effect structurally flawed circuits have on the generative model, generated circuits are pruned from excess wires and redundant components, and circuits that are not sufficiently connected are discarded.

There can be many functionally equivalent ways to construct circuits, with some of these circuits having redundantly connected components. Keeping every such circuit as a training sample may cause the generative model to learn to include these redundancies in their generative process. To maintain high-quality training data, only the circuits using the least amount of wires and components are used in the training data. This ensures that for every functional behavior, only the most compact circuits are retained. As bootstrapping generates new circuits, initially inefficient ones are replaced, gradually refining the training data. This setup acts similarly to the "MAP-elites" strategy, storing a single optimally found solution per functional behavior Mouret and Clune 2015. Even if the generative model may produce overly complex solutions, subsequent

iterations may discover improvements, gradually converging towards minimal designs.
The pruning and filtering actions carried out are:

- Excess wire is removed. Wires that are not connected with at least 2 adjacent pixels are recursively removed, practically truncating unused wires. As these wires are not connected, they serve no purpose to the circuits.
- Gates that are not connected on both of their inputs and their output are replaced with wire. Gates that are not connected do not hold functional importance.
- Circuits must have at least one gate, otherwise they are not added to the training data. This is to encourage interesting circuits, as circuits without gates can be trivial to construct.
- If there is a 2 by 2 section of the circuit that consists of exclusively wires, a 'wire block', the circuit is removed. While such circuits could be reasonably constructed, often one of the wires could be removed without the circuit losing its functionality. These circuits are removed to prevent the generative model from fixating on this often redundant structure.

As the training data is updated, the generative model needs to be retrained to fit the new dataset, to update its knowledge about the domain. Training a new model every time can be expensive, instead, one DDM is used throughout training, retraining every time the training data is updated. This can be problematic as the DDM may struggle to adapt from the initially small dataset, but the domain it is training to model is consistent with the added training data. It is important that the time to bootstrap is minimized as many domains are very large. The circuit domain with 4 inputs and outputs has 16^{16} possible unique ways to be connected, while the images have 13^{13^4} possible states. Without a proper way to determine the relevance of individual circuit functions, bootstrapping needs to develop a large enough subset of the domain to be able to train a model to accurately recreate specific circuit functionalities.

3.5 Architecture and bootstrapping setup

The categorical DDM used in this thesis uses a U-Net with three down and up blocks, two of which include cross-attention weighing the value of information from the embedding [Ronneberger, Fischer, and Brox 2015, Bahdanau et al. 2014, Rombach et al. 2022]. Each block includes two convolutional layers and has 128 channels (extended from the original 4 of the circuits). The diffusion is done over 200 steps with a linear schedule, such that the noise is added linearly. For conditioning, truth tables of 16 rows and 8 columns are encoded into a space of size 4 by 128. Together with the embedding model, the DDM has a parameter count of just over 13 million parameters.

Before bootstrapping is performed, the generative model is first trained on the initial data to ensure a solid starting point for generation. Then, in each iteration of bootstrapping the model is retrained on the current data, updating the model to include information of the newest samples. As a result, the model is trained twice on the initial data, and once every time the training data is updated.
After training, the model is used to generate 10 batches of 256 circuits, generating up to 2560 new potential training samples. Inference is done over 3 steps of 200 steps diffusion has been trained over, encouraging exploration of other circuits over recreation of known circuits.

As the initial amount of training data is quite low, the samples were repeated (from 4 to 16 samples) to more efficiently make use of batching for the first session of training, but reduced back to 1 sample per circuit after the training data has been expanded. To train the categorical DDM to generate circuits, each iteration of bootstrapping trains the model for up to a 100 batches. Having no test or validation set, the model is trained with a hybrid approach, either till it reaches a low loss (less than 0.05 cross-entropy loss for the batch) or till it has trained for a 100 batches. This was done to keep the model reasonably accurate without spending a lot of time training. As the training data increases, the amount of batches in an epoch increases, and training for a set amount of epochs can cause the model to overfit the data, while training till a low amount of loss has been reached becomes harder as the amount of samples increases. The choice of the up to 200 initial training epochs was decided from an empirical test in Appendix B, of the amount of unique circuits generated over a batch for multiple epochs.

While bootstrapping, the generation rate and quality are measured to get insight into the viability of bootstrapping. The measures used are:

- Amount of training data.
- 'Samples Added to Training Data', The rate at which training data is increased.
- 'Shorter Samples', The rate at which training data is replaced with shorter samples.
- Generation and Simulation errors.
  - 'Duplicate', Generated sample is a copy of a sample in the training data.
  - 'New Duplicate', Generated sample is a copy of a sample within the currently generated batch.
  - 'Has Square', Generated circuit fails the requirement to not have a wire block (2 by 2 square of wires).

  - 'No gate', Generated circuit fails requirement that circuits must have at least one gate.
  - 'Not sufficiently Connected', Generated circuits must have at least 2 inputs and 1 output.
- 'Successful Samples', Circuits generated without errors.

All trials in the project are run thrice, except when stated otherwise.

4 Results

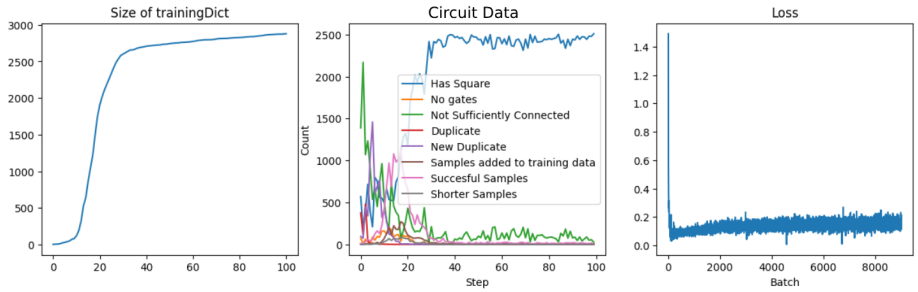


Figure 5: Graphs from training a 100 iterations of bootstrapping, showing the size of the training data over each iteration (left), the result and error distribution over each iteration (middle), and the loss of the DDM (right). The process generates several circuits usable but begins stagnating with an increase in errors.

This first trial was run thrice, with the results shown in Figure 5. For both attempts, the amount of training data follows a piecewise linear function, increasing the amount of data samples fast at first, but at a diminished rate in later iterations of the bootstrapping. From the middle graph showing 'Circuit Data', the pink line showing generated circuits that work and are not duplicates of existing samples, grows initially, generating more usable circuits as the diversity of the training data increases, but later falls to the point of barely any acceptable circuits being generated. Of the 2560 circuits generated every iteration of bootstrapping, over 2000 of these were removed due to various errors and the training data ended up getting updated with less than 20 new circuits and 10 shorter circuits every iteration. The main error causing these circuits to be filtered is the amount of circuits that are generated, which contain wire blocks. This is a sign of degeneration, that the model is learning to generate wire blocks without them being present in the training data. A subset of the last batch of circuits created can be seen in Figure 6, which also shows that

the circuits that are being generated by the end are full of redundant components and wires. Though these circuits are inefficient, it is not directly bad for bootstrapping that the circuits are generated. These chaotic circuits all use an excessive amount of wires and components, that serve no functional purpose, but they would get replaced when more efficient circuits are discovered. The problem, however, is that these more efficient circuits are not being generated.

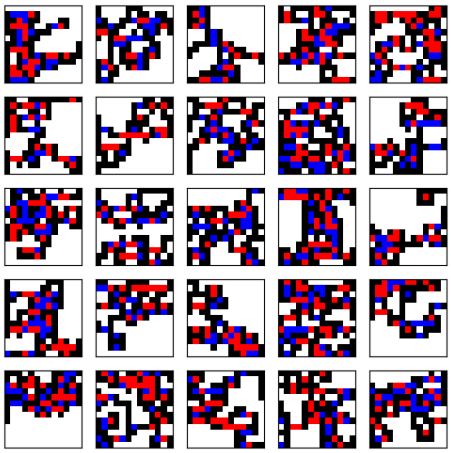


Figure 6: 25 samples of circuits generated by the end of training. These circuits are messy with several irrelevant components.

Though further iterations of bootstrapping may lessen the problem of these wire squares, with over 80% of the circuits generated being wasted, it would take many iterations of bootstrapping before any significant change happens.

4.1 Reducing wasted generations

With many of the generated circuits not being used for training due to failing various requirements, much of the time spent bootstrapping is wasted. Taking action to prevent the circuits from not adhering to the requirements could increase the rate at which bootstrapping can be performed.
The requirement for circuits to not have wire blocks was added to mitigate inefficient circuits from being generated. By removing the constraint, the errors associated with it would be removed, and since inefficient circuits get replaced when shorter circuits are found, the inefficient wire blocks may remove themselves over further bootstrapping.

Removing the wire block constraint is bound to induce more degenerate circuits with redundant wires. However, should bootstrapping work with only the strict requirements necessitated by the domain, it would indicate that the method may generalize well to other domains.

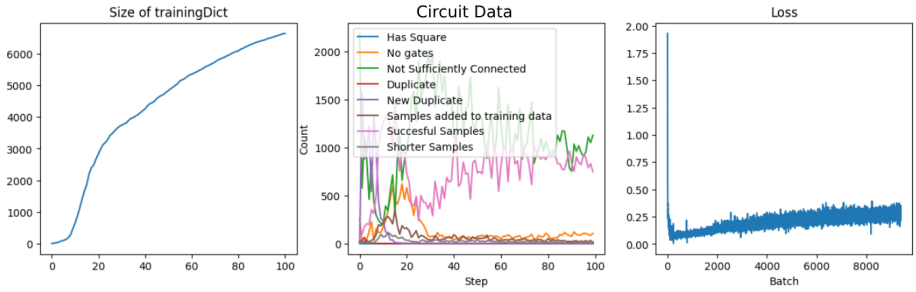


Figure 7: Graph from training a 100 iterations of bootstrapping with the minimal requirements. With fewer circuit errors training data is increased at a higher rate.

Being more lenient with what circuits are acceptable does lead to a stark increase in the amount of acceptable circuits generated per batch. This increase in acceptable generations is not leading to a significant increase in the growth of the training data, nor are more circuits getting updated with shorter versions. The circuits that end up being generated are often overly complex with many components having no functional importance. This leads to many of the generated circuits having the same functionality and competing for the same few entries in the training data. From Figure 7, the brown and gray lines representing the samples extending the training data and the samples that are shorter than existing circuits, are both consistently low. Showing no noticeable improvement from the previous experiment shown in Figure 5. Along with there being no significant improvement to the growth of training data, simulating these circuits takes a lot longer, due to many gates being redundantly interconnected.

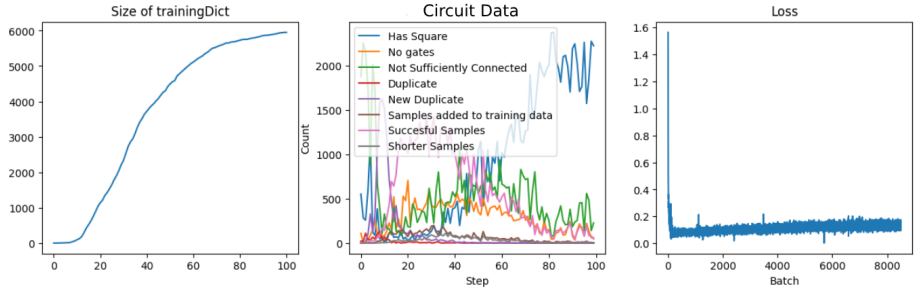


Figure 8: Graph from training a 100 iterations of bootstrapping, while restricting component blocks. Despite the restriction on generated circuits, the 'Has Square' error persists.

Alternatively, being more strict, further constraining generated circuits, saw a

significant improvement in circuit quality. Displayed in Figure 8, the result of restricting component blocks, any 2 by 2 square of either wires or components, is shown. This setup performed well initially with few circuit errors. As bootstrapping continued, despite not having any component blocks in the training data, the generative model would learn to produce them, resulting in the high amount of 'Has Square' errors, ending with about 2000 errors in each iteration by the end. One reason as to why the component blocks get generated despite there being no representation of them within the training data, is similar structures do get accepted, with crossing wires often leading to undiscovered circuit functions.

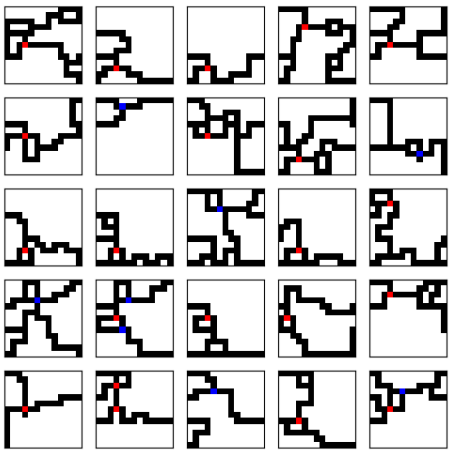


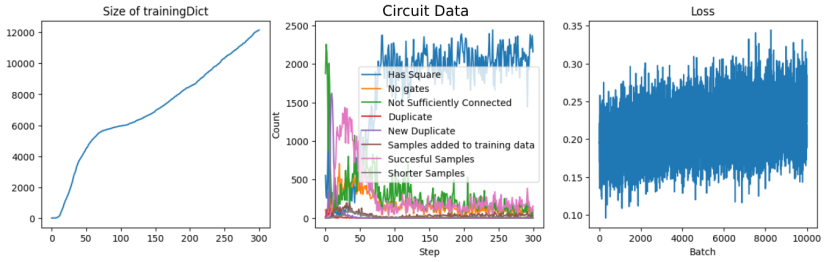
Figure 9: Example circuits generated at the end of bootstrapping while restricting component blocks. The generated circuits are reasonably constructed, but their behavior is mostly trivial.

The circuits generated under the component block constraint were more often shorter than existing circuits and were consistently well connected between inputs and outputs. Figure 9 shows examples of the generated circuits, with many of the circuits looking reasonable, but with trivial behavior.

4.2 Continued training

Extending a single one of the trials restricting component blocks for another 200 iterations showed no significant change in the bootstrapping metrics. After around the 100th iteration, the circuit failure rate and success rate are effectively unchanged. From Figure 10, it can be seen that the amount of training data increases linearly with the amount of bootstrapping after the first 100 iterations. Linear regression on the last 200 iterations, shows that bootstrapping does continue to extend and improve the created dataset, improving circuits at

a)



b)

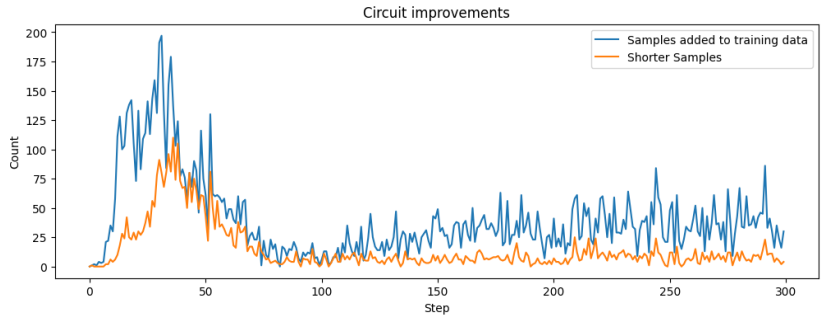


Figure 10: a) Graph over training data size (left), the result and error distribution over each iteration (middle), and loss of the DDM (right) for 300 iterations of bootstrapping. The loss graph is only for the last 100 iterations of bootstrapping. b) Graph over circuits added to and replacing samples of training data, isolated from the above circuit data. Bootstrapping shows continued exploration of the domain and improvement of found circuit functions.

the rate of 7.4 circuits per iteration, and extending the data by 32.58 circuits per iteration.
While the amount of training data was increased from further bootstrapping, the rate cannot stay unchanged and would be bound to decrease as more circuit behaviors are found, given that the domain is finite. A generative model does not need all possible samples of a domain to learn a meaningful representation. The amount of samples necessary depends on the complexity of the domain as well as the amount of and the diversity of the samples.

The quantity of circuits generated is not a good representation of how the domain is covered, as many circuits are trivial to construct. Instead to measure how well the generated training data covers the domain, 'gate coverage' and 'gate representation' are used as metrics to represent the entire domain. By gate coverage, the representation of logic operations generated is measured from how many of the different ways to construct the functions 'AND, OR, NAND, NOR, XOR, and XNOR' are represented in the data. These logical operations are only considered in the form of two inputs connected to a single output. With gate representation, the placement of gates is measured to quantify whether

Trial	AND	NAND	OR	NOR	XOR	XNOR	AND-Representation	NAND-Representation
No Squares	24	24	24	24	0	0	36	58
Accept Squares	15	23	24	24	0	0	143	143
No Squares - further trained	24	24	24	24	0	0	143	143
Higher Inference	14	17	18	18	0	0	42	42

Table 1: Coverage of logic operators and representation of components in the generated circuits for each trial. (Left) Trials. (Middle) Logic operator coverage and the amount of circuits of each function that were present in the generated training data. (Right) Component representation, the amount of different pixels on which the AND and NAND components have been placed in the generated circuit images. This data is used as a metric to determine how well the circuit domain is covered. None of the trials generated any XOR or XNOR logic operations and struggled to produce non-trivial circuits.

the training data explores the various component positions afforded by the domain, specifically assessing coverage across all pixel locations. These measures describe a subset of the domain but contain non-trivial functions and behaviors. From Table 1 it can be seen that the functions that are trivial to produce are well covered, with many trials showing complete coverage out of the 24 different ways to connect two inputs and one output. The non-trivial functions of XOR and XNOR, which would be great estimators of domain knowledge, are not represented within the data for any of the trials. From the gate representation measure, the models do not show signs of significantly overproducing one of the given components. For the trials with less restrictions or further training, the training data includes placement of components within 143 pixels, which is the entire images except the input and output columns, showing a complete exploration of the image domain.

4.3 Increasing inference

For all trials inference has been performed over 3 steps out of the 200 steps diffusion has been trained over. Inference affects the precision of the generated works, with higher inference generations having more fine-grained control over the generated product, leading to higher correspondence with the dataset. When the DDMs are trained only on the initial dataset, using a low number of inference steps helps prevent the model from replicating training data too closely and encourages the generation of novel circuits. However, as the training data increases, even a high amount of inference steps struggle to exactly reproduce training samples, while few steps cause the generated circuits to become more random, losing relevant structures like connected wires.

Increasing inference steps during circuit generation leads to some clear improve-

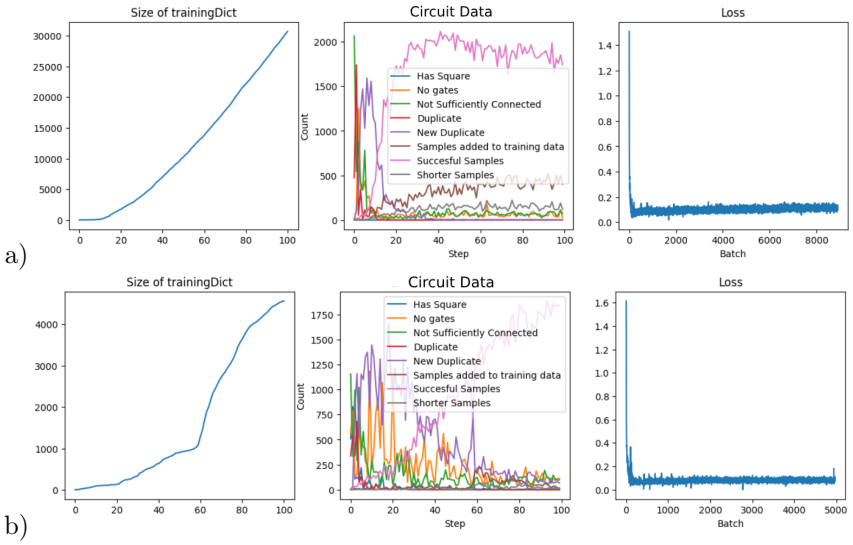


Figure 11: Bootstrapping results of performing bootstrapping with increased inference. a) higher inference without square restriction. b) higher inference with strict square restriction. a) shows a much higher rate of circuits being added to the training data and replaced with faster versions. b) does not show this significant increase, but the rate of successfully generated samples was increasing throughout and the 'has square' error is barely present.

ments in bootstrapping. Figure 11 shows the results of using 13 steps of inference, with one very noticeable effect being the stark increase in training data for the model that does not restrict component blocks. This model has generated far more circuits than previous attempts and at a much faster rate. A majority of generated circuits are successful, and of these more circuits have been either unique functions added to the training data, or shorter circuits.
While not having a large improvement in the size of the training data, the model that restricts component blocks also showed relevant improvements. This model showed a low rate of successful circuits initially, but this rate increased throughout the bootstrapping procedure, showing a linear increase over the duration of the bootstrapping. By the end of bootstrapping most of the generated circuits from the model were successful, and the 'has square' error is insignificant. For both trials, with and without component block restraints, the experiment was repeated three times, showing similar results.

The higher inference trial shows a small amount of gate representation (from Table 1), meaning that when this method generates new circuits, it is likely to use gate positions that have been used before, in the training data. Despite the higher inference version having generated a far larger dataset, the low gate representation means that the method is not broadly exploring the domain. From

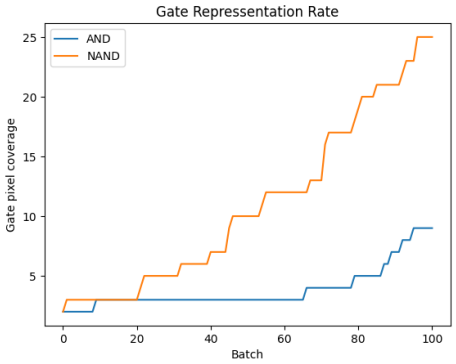


Figure 12: Gate representation over a 100 iterations of bootstrapping for higher inference model. The generative model shows a bias towards NAND-gates, yet despite the data imbalance AND-gates resurface, having their representation in the training data increased.

Figure 12 the gate representation level at each iteration of bootstrapping is displayed. The representation of NAND-gates increases throughout bootstrapping while AND-gates stagnate early. Although the growth of AND-gates stagnates early on, their representation begins to rise again near the end of bootstrapping, even as the training data at that stage favors NAND-gates.

4.4 Larger starting dataset

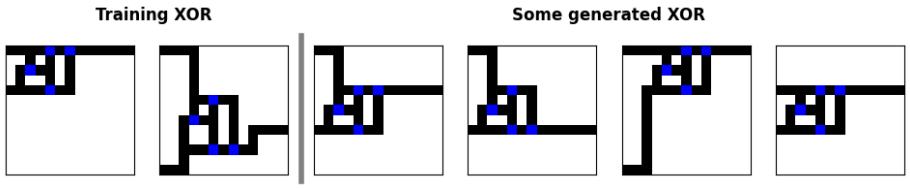


Figure 13: On the left are training samples of XOR operations. On the right, are examples of generated XOR operations. The generated circuits show a similar pattern of components, shifted and slightly adjusted to construct new circuits.

Expanding the initial dataset to include two examples of both XOR and XNOR operations (a total of 8 circuits in the dataset), bootstrapping was able to discover more ways to construct those operations. Table 2 shows the gate representation and coverage for a trial with the extended dataset, and Figure 13 illustrates a few of the generated XOR operations. This trial used the high inference approach of using 13 steps to develop the circuits from random noise. The extended dataset did allow the model to generate more of the otherwise

non-trivial logic operators XOR and XNOR. However, given the preexisting samples in the training data, this outcome is not unexpected. Of the XOR and XNOR operators that were generated, the synthetic samples follow the same underlying configuration of components as those in the training data, with only slight adjustments to construct new distinct truth tables. This dataset severely constrained the construction of AND-gates in the generated circuits, however, XOR and XNOR are simpler to construct with NAND-gates, making NAND gates more common as in the synthetic samples.

	AND	NAND	OR	NOR	XOR	XNOR
Gate Coverage	11	16	17	17	10	5
Gate Representation	4	63	N/A	N/A	N/A	N/A

Table 2: Gate coverage and representation for extended dataset trial. This trial constructed multiple examples of XOR and XNOR operations but exhibited a very low representation of AND gates.

4.5 Execution time

For circuit generation, bootstrapping has consisted of training, inference, and simulation, which are all time-consuming processes. To get an idea of how long it would take to generate a synthetic dataset, the time taken to generate circuits has been measured. The circuits generated, are of a low resolution and high quantity. The device used for this project was a 'RTX 3070' GPU and it handled 256 circuits at a time well. The VRAM used for this would be $256 * 13 * 13 * 4$ (batch, height, width, One-hot classes) floating point numbers for the images, along with $256 * 128 * 4$ floating point numbers for the encoded truth tables making for a total of 304128 floating point numbers or $\sim 1.2 * 10^6$ bytes, which is well below a single Giga Byte of VRAM.

Subject	**Allocate circuits**	**Use transformer**	**Perform inference**	**Total Circuit Generation Time**	**Clean/Prune Circuits**	**Hash Circuits**	**Total Table Simulation Time**
Untrained Model	1.86	0.024	0.56	2.54	0.46	0.004	23.82
Squarefull Model	1.80	0.024	0.64	2.55	0.40	0.002	26.85
Squareless Model	1.81	0.024	0.57	2.50	0.32	0.003	0.44

Table 3: The time taken to generate and simulate a batch of 256 circuits for a model trained with the strict no component block constraint 'Squareless', the model without the constraint 'Squarefull', and an untrained model. The time to simulate circuits is one of the largest bottlenecks for the bootstrapping of circuits, Improving the simulation speed may significantly improve the rate at which bootstrapping is performed for circuits.

Table 3 shows the time taken for 3 different models to generate and simulate circuits. With all models using the same network architecture, circuit generation time is more or less the same, spending most of the time on allocating memory rather than performing inference. For Simulation speed, the model restricting component blocks had a far lower simulation time than the others. This model spent most of the simulation time on cleaning circuits from excess wires. Simulation is done by tracing which components are connected to each other and simulating updates to wire and component states recursively. When wires and components are placed haphazardly, as when generated through an untrained model, many gates may become redundantly connected, with several connections that do not impact the function of the circuit. The model trained without restricting component blocks, performed simulation a lot slower as it generates more interconnected gates. The model restricting component blocks performed way better during simulation, showing a significant decrease in simulation time.
With the bootstrapping performed over 10 iterations of batches of 256 circuits, generation and simulation time takes in total between 29.39 and 278.88 seconds, depending on how well-behaved the generated circuits are. Therefore, restricting when circuits are meet the minimal criteria of the domain, can reduce the time it takes to bootstrap by a lot for circuits.

4.6 Deeper model

With all the generated circuit data, the generative model should have enough data to learn features of the domain, and somewhat be able to infer how circuits should be constructed to match a given truth table. However, despite training on the data, the models could not reproduce the functions described in a given embedding. Even further, the models struggle to reproduce the functions covered by the training data.
A possible reason for this inadequacy is that the target domain is too large and could not be described within the bounds of the neural network. Investigating this, a much deeper model was trained on one of the generated datasets.
With three pooling layers upon 13 by 13 pixel images, the latent space of the trained U-Nets was already very succinct and should be able to transfer most long-range dependencies in the circuits. Therefore to make a deeper neural network, the amount of layers per block was increased, from 2 layers per block to 5, making for a roughly 150% increase in parameters. Despite the massive increase in model size, there was little effect on the generated circuits, with the generated samples not adhering to the provided embeddings. A test, detailed in Appendix C, was also performed with a Gaussian diffusion model, showing no improvements over the categorical models otherwise used in the project.

5 Discussion

5.1 Failing to learn the domain

An issue that was prominent while doing bootstrapping, was the 'square error', a requirement added to encourage realistic circuits. The circuits that were being filtered from this requirement would at the start only be a few of the generated circuits, but over the course of the bootstrapping would end up being the majority of the circuits. Actions were taken to mitigate this problem through both loosening and narrowing the constraints on circuits, but the main problem that had been causing the issue was that the models did not learn the domain.

Electronic Design Automation is a domain different from typical image generation. With circuits, the inputs and outputs of different components can have decencies over long ranges of the generated image, and these dependencies can have a significant impact on the behavior of the circuits. Modeling this with diffusion on a shallow U-Net, it is limited how much of this domain the model can learn over each timestep. Yet even with a deeper model this saw no improvement.

The implementation of circuits as images implicitly includes a problem for the model of understanding planar graphs. A planar graph is a graph where no edges overlap. Similarly, for circuits, wires can overlap, but doing so in this setting merges their charges. When and how wires should be connected is something the model should take account of, to be able to generate circuits of specific properties. Learning how to place components and the functional relations between connected components may be too advanced a problem for the DDMs that were used.

The iterative application of DDMs allows for a progressive refinement, where each step learns a distribution of the data at that timestep. For image generation, this progressive refinement is demonstrated during denoising in the form of DDMs first recreating the main features, before filling in details. Images have a lot of locally relevant information which makes Convolutional Neural Networks well suited for them. For circuit generation, however, information is not as locally relevant and it can be hard to infer far-away dependencies from local values. When connecting gates with wires, there may not be any information to infer about which direction the wires should be extended towards. While the iterative approach of DDMs allows for some propagation of information in the form of slowly filling in details, they always generate towards what would be the likely state of a circuit at the given timestep. This restriction is essential to how DDMs are trained, but is inflexible in how information necessary for further denoising can be stored. A more flexible method may be necessary for learning the circuit domain, such as recurrently applied Neural Networks applied end-to-end over the denoising process [Gilpin and Gilpin 2019, Mordvintsev et al. 2020, Palm et al. 2022, Hartl, Risi, and Levin 2024, Kalkhof et al. 2024].

5.2 Biases, Hallucinations and Bootstrapping

One rationale for using generative AI to bootstrap its own training data is that it shares a similar exploration vs exploitation tradeoff to that found in reinforcement learning. Generative models can be similarly leveraged through the noise vs replication relation for under- and overfit generative models to generate data. This intuition is misplaced, while an untrained model generates noise and an overfit model generates replications of training samples, a trained model does not linearly interpolate between the two. As models train they may learn an arbitrary subset of relevant features of the domain. Instead, to modulate between the exploration of new diverse samples, and higher quality samples that align closely with existing data, adjusting the amount of steps done during inference is preferable. However, the generative model does still need to be trained.

For the DDMs trained on circuit data, the generative models consistently displayed a trend of first learning the placements of wires before learning to include gates in their denoising process. Such a model would have a large bias towards circuits without wires which would be propagated into future generations as the synthetic data gets added to the training data. To avoid poor generations, there needs to be a stopping criterion for the model which ensures that the model is neither underfit nor overfit, such that the generated samples are most likely to be appropriate for the domain.
The model should be trained to minimize loss over the domain, which can be measured through a proxy using a validation set. While a portion of the training data is typically reserved for validation, the synthetic data that is generated by the model may not be optimal solutions for the circuit behaviors they represent, and using these to validate the model is a less reliable metric. Using a fixed-sized handwritten validation set may work well, but it would put a larger burden on the user to construct more data.

Because bootstrapping was performed with very little initial training data, when synthetic data was added to the dataset, the impact of the synthetic data quickly outweighed the initial data. If the generative model has a bias towards a specific attribute, this attribute is propagated into future generations. The initial circuit dataset consisted of two examples of AND-gates and two examples of NAND-gates, yet the model would at first prioritize generating just the NAND-gates, which increased the bias in the training data towards the NAND-gates. As a generative model continues generating new samples such biases are bound to happen, but they are much more prominent on smaller datasets. Even so, most of the trained models do end up with reasonable coverage of the gate components and show signs of continued diverse exploration. Specifically, the gate representation metric is increasing for both gate components shown in Figure 12.

Model Autophagy Disorder (MAD) is the effect of generative models sampling from a region of higher uncertainty, between modes, of their target distribution and being trained on that data, compounding the attributes of the uncertain

sample into the model. In this bootstrapping framework, where criteria for viable generations can be measured, MADness still affects the generative models leading to degenerate generations like the circuits generated in Figure 6, characterized by redundant components. Imposing restrictions on visual features improves the appearance of the circuits, but MADness continues to influence the generative process in other ways, affecting features that are not obviously tied to visual appearances. Despite this, as unviable generations are filtered out, the remaining hallucinations are functional circuits with features that are underrepresented or absent in the training data.

5.3 Search limitation

When generating circuits for extending training data, the generative model was supplied a noisy image and embedding. This data was translated into a generated circuit with an associated truth table. The circuit that ends up being generated may be different from the training data, but it is within the learned representation of the generative model. This generative model may be under or overfitted to the training data, in which case its learned representation will not accurately describe the domain.
As the amount of training data increases and begins saturating, the random search for circuits may reach a point where most newly generated samples are within the parts of the domain that the training data already describes. Bootstrapping resembles a random walk, sequentially sampling related points within the domain. While this approach can explore the entire domain space, it may struggle in comparison to other methods that can leverage novelty mechanisms in their search. Generative models do afford some steering mechanisms for directing generation, with embedded data being a potential driver for directing generation toward unexplored areas of the domain.

6 Future work

In this work, bootstrapping was attempted in a domain of circuit generation. The chosen generative model may not have been suitable for the domain, and it remains to be seen how useful the bootstrapping approach can be. To better evaluate the effectiveness of bootstrapping, it should be attempted within a well-documented domain that enables comparison with alternative approaches. Domains in which reinforcement learning has demonstrated strong performance serve as especially interesting comparisons since the bootstrapping process is similarly capable of balancing exploration and exploitation.
If comparing the approaches in a task of optimizing a specific value, reinforcement learning is likely the superior method in most cases, however, when compared in terms of their ability to generate a dataset for a domain, like using novelty search and saving the unique samples, how do the methods compare to each other? In these open-ended domains, bootstrapping may have an advantage for promoting diversity rather than convergence.

In a new white paper from Google DeepMind, a similar process to bootstrapping has been used to find and improve optimizations in several fields, with their new model AlphaEvolve [DeepMind 2024]. In their paper, they use a Large Language Model (LLM) to suggest algorithms, evaluate them, and iterate upon the algorithms. Leveraging the knowledge contained within a LLM they, among other things, found an optimization for computer chips at the Register Transfer Level. While the methods used in this project and theirs are different, they both build upon using generative models to iterate upon their own generations and using automatic evaluations to measure the quality of generated samples. Their paper shows concrete applications for this process, finding several improvements to problems that can be automatically evaluated. While the bootstrapping used in this project showed no distinct improvements, the discoveries from AlphaEvolve document that bootstrapping can lead to notable breakthroughs.

7 Conclusion

Many deep learning models benefit from having access to large amounts of data. While there are datasets for specific/general tasks, smaller domains may have little to no data available. To alleviate this, a method, bootstrapping, has been developed for generating samples of a domain. With bootstrapping, a generative model is trained on an initially small dataset, and its own synthetic generations are used to expand the dataset, thereby increasing its own knowledge of the domain. Training generative models on synthetic data can lead to Model Autophagy Disorder, where the quality of generations degrades as imperfections in the synthetic data compounds in future generations. However, this problem can be mitigated if samples of the target domain can be evaluated for their quality. Testing this approach in a domain of circuit generation, whether the synthetic circuits can be simulated is used as a minimal criterion for whether a generated circuit can be added to the training data. For the generative model, a denoising diffusion model working with categorical noise is trained to generate its own dataset.

The bootstrapping approach is tested under different settings, and measured for its ability to cover the domain as well as the rate by which it is done. From the tests, bootstrapping was found to be able to massively increase the amount of training data available for the domain. The diversity of the synthetic samples allowed the generative model to explore the domain well, including regions that were not covered by the initial data. Bootstrapping shows promise for developing generative models in open-ended tasks and comparing it against established methods could bring valuable insight into capabilities and limitations.

References

Lehman, Joel and Kenneth O Stanley (2010). "Revising the evolutionary computation abstraction: minimal criteria novelty search". In: *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pp. 103–110.

— (2011). "Novelty search and the problem with objectives". In: *Genetic programming theory and practice IX*, pp. 37–56.

Bahdanau, Dzmitry et al. (2014). "Neural Machine Translation by Jointly Learning to Align and Translate". In: *arXiv: Computation and Language*.

Mouret, Jean-Baptiste and Jeff Clune (2015). *Illuminating search spaces by mapping elites*. arXiv: 1504.04909 [cs.AI]. URL: https://arxiv.org/abs/1504.04909.

Ronneberger, Olaf, Philipp Fischer, and Thomas Brox (2015). *U-Net: Convolutional Networks for Biomedical Image Segmentation*. arXiv: 1505.04597 [cs.CV]. URL: https://arxiv.org/abs/1505.04597.

Sohl-Dickstein, Jascha et al. (2015). "Deep Unsupervised Learning using Nonequilibrium Thermodynamics". In: *arXiv: Learning*.

Soros, LB, Nick Cheney, and Kenneth O Stanley (2016). "How the strictness of the minimal criterion impacts open-ended evolution". In: *Artificial Life Conference Proceedings*. [Accessed 11-11-2024]. MIT Press One Rogers Street, Cambridge, MA 02142-1209, USA journals-info . . ., pp. 208–215. URL: https://www.uvm.edu/neurobotics/pubs/pdf/2016_SorosCheneyStanley_HowTheStrictnessOfTheMinimalCriterionImpactsOpenEndedEvolution_ALIFE.pdf.

Vaswani, Ashish et al. (2017). "Attention is All you Need". In: *Neural Information Processing Systems*.

Gilpin, William and William Gilpin (2019). "Cellular automata as convolutional neural networks." In: *Physical Review E*. DOI: 10.1103/physreve.100.032402.

Ho, Jonathan, Ajay Jain, and Pieter Abbeel (2020). *Denoising Diffusion Probabilistic Models*. arXiv: 2006.11239 [cs.LG]. URL: https://arxiv.org/abs/2006.11239.

Mordvintsev, Alexander et al. (2020). "Growing Neural Cellular Automata". In: *Distill*. DOI: 10.23915/distill.00023.

Torrado, Rubén Rodríguez et al. (2020). "Bootstrapping Conditional GANs for Video Game Level Generation". In: *2020 IEEE Conference on Games (CoG)*. DOI: 10.1109/cog47356.2020.9231576.

Arnab, Anurag et al. (2021). "ViViT: A Video Vision Transformer". In: *IEEE International Conference on Computer Vision*. DOI: 10.1109/iccv48922.2021.00676.

Austin, Jacob et al. (2021). "Structured Denoising Diffusion Models in Discrete State-Spaces". In: *Neural Information Processing Systems*.

Hoogeboom, Emiel et al. (2021). "Argmax Flows and Multinomial Diffusion: Learning Categorical Distributions". In: *Neural Information Processing Systems*.

Nichol, Alex and Prafulla Dhariwal (2021). "Improved Denoising Diffusion Probabilistic Models". In: *International Conference on Machine Learning*. DOI: null.

Friedman, Roy (2022). *A simplified overview of Langevin Dynamics*. URL: https://friedmanroy.github.io/blog/2022/Langevin/.

Murphy, Kevin P. (2022). *Probabilistic Machine Learning: An introduction*. Chapter 1.5: Data. MIT Press. URL: http://probml.github.io/book1.

Palm, Rasmus Berg et al. (2022). "Variational Neural Cellular Automata". In: *International Conference on Learning Representations*. DOI: null.

Rombach, Robin et al. (2022). "High-Resolution Image Synthesis with Latent Diffusion Models". In: *Computer Vision and Pattern Recognition*. DOI: 10.1109/cvpr52688.2022.01042.

Song, Jiaming, Chenlin Meng, and Stefano Ermon (2022). *Denoising Diffusion Implicit Models*. arXiv: 2010.02502 [cs.LG]. URL: https://arxiv.org/abs/2010.02502.

Ambrogioni, Luca (2023). "In Search of Dispersed Memories: Generative Diffusion Models Are Associative Memory Networks". In: *Entropy*. DOI: 10.3390/e26050381.

Friedman, Roy (2023). *Annealed Importance Sampling*. URL: https://friedmanroy.github.io/blog/2023/AIS/.

Trabucco, Brandon et al. (2023). "Effective Data Augmentation With Diffusion Models". In: *International Conference on Learning Representations*. DOI: 10.48550/arxiv.2302.07944.

Wu, Wangyu et al. (2023). "Image Augmentation with Controlled Diffusion for Weakly-Supervised Semantic Segmentation". In: *IEEE International Conference on Acoustics, Speech, and Signal Processing*. DOI: 10.1109/icassp48485.2024.10447893.

Zhang, Lvmin, Anyi Rao, and Maneesh Agrawala (2023). "Adding Conditional Control to Text-to-Image Diffusion Models". In: *IEEE International Conference on Computer Vision*. DOI: 10.1109/iccv51070.2023.00355.

Zhou, Qiong et al. (2023). "Unlocking the Transferability of Tokens in Deep Models for Tabular Data". In: *arXiv.org*. DOI: 10.48550/arxiv.2310.15149.

Aithal, Sumukh K et al. (2024). "Understanding Hallucinations in Diffusion Models through Mode Interpolation". In: *Neural Information Processing Systems*. DOI: 10.48550/arxiv.2406.09358.

Alemohammad, Sina et al. (2024). "Self-Improving Diffusion Models with Synthetic Data". In: *arXiv.org*. DOI: 10.48550/arxiv.2408.16333.

DeepMind (2024). *AlphaEvolve: A Gemini-powered coding agent for designing advanced algorithms*. Tech. rep. White paper. DeepMind. URL: https://storage.googleapis.com/deepmind-media/DeepMind.com/Blog/alphaevolve-a-gemini-powered-coding-agent-for-designing-advanced-algorithms/AlphaEvolve.pdf.

Hartl, Benedikt, Sebastian Risi, and Michael Levin (2024). "Evolutionary Implications of Self-Assembling Cybernetic Materials with Collective Problem-Solving Intelligence at Multiple Scales". In: *Entropy*. DOI: 10.3390/e26070532.

Hartl, Benedikt, Yanbo Zhang, et al. (2024). *Heuristically Adaptive Diffusion-Model Evolutionary Strategy*. arXiv: 2411.13420 [cs.NE]. URL: https://arxiv.org/abs/2411.13420.

Islam, Khawar et al. (2024). "Diffusemix: Label-Preserving Data Augmentation with Diffusion Models". In: *Computer Vision and Pattern Recognition*. DOI: 10.1109/cvpr52733.2024.02608.

Kalkhof, John et al. (2024). "Frequency-Time Diffusion with Neural Cellular Automata". In: *arXiv.org*. DOI: 10.48550/arxiv.2401.06291.

Li, Xihan et al. (2024). "Circuit Transformer: End-to-end Circuit Design by Predicting the Next Gate". In: *arXiv.org*. DOI: 10.48550/arxiv.2403.13838.

Zhang, Dan, Jingjing Wang, and Feng Luo (2024). "Directly Denoising Diffusion Models". In: *International Conference on Machine Learning*. DOI: 10.48550/arxiv.2405.13540.

Zhang, Yanbo et al. (2024). *Diffusion Models are Evolutionary Algorithms*. arXiv: 2410.02543 [cs.NE]. URL: https://arxiv.org/abs/2410.02543.

Graves, Alex et al. (2025). *Bayesian Flow Networks*. arXiv: 2308.07037 [cs.LG]. URL: https://arxiv.org/abs/2308.07037.

Hartl, Benedikt and Michael Levin (2025). *What does evolution make? Learning in living lineages and machines*. DOI: https://doi.org/10.31219/osf.io/r8z7c. URL: https://osf.io/preprints/osf/r8z7c_v1.

Lim, Yechan, Sangwon Lee, and Junghyo Jo (2025). "Data augmentation using diffusion models to enhance inverse Ising inference". In: *null*.

Mitchell, Kevin J. and Nick Cheney (2025). *The Genomic Code: The genome instantiates a generative model of the organism*. arXiv: 2407.15908 [q-bio.OT]. URL: https://arxiv.org/abs/2407.15908.

A Diffusion model verification

Before performing bootstrapping, the chosen categorical diffusion model was tested to make sure it was capable of generative modeling. The task chosen to measure this was one-hot encoded binarized MNIST [Graves et al. 2025]. MNIST is a dataset of low-resolution hand-drawn digits. The digits were binarized, meaning each pixel was set to either an "on" or "off" state. After binarization, the digits were distributed across 11 channels, one for each digit, and an additional "none" channel. For example, the digit '2' was represented by setting the value to 1 in the pixels that make up the shape of the '2' in its corresponding channel. At the same time, the pixels that were not part of the '2' were given a value of 1 in the "none" channel. All other pixels in all channels were set to 0. Training a categorical diffusion model which predicts the output rather than the noise, resulted in the digits generated in Figure 14. As these digits were reasonably recognizable, this diffusion process was used for bootstrapping.

B Relation between amount of training and circuit generation

For bootstrapping, optimizing the rate at which new usable samples are generated can increase the rate at which the training data can be expanded. To find the point at which the Denoising Diffusion Models generate the highest throughput of usable circuits, the number of circuits generated per batch in multiple epochs of training has been measured. Figure 15 presents sampled data across epochs, measuring the amount of new unique circuits that are generated per batch, as well as how many of those are usable circuits. As the generative model continues training, the amount of unique circuits generated falls, while the amount of these unique circuits that live up to the minimum requirements of the domain increases. Mainly the requirement of having connections between the inputs and outputs.

C Gaussian diffusion on circuit domain

As the categorical diffusion models trained for the project did not learn the domain well, a comparison is done with unmodified Gaussian diffusion. Using a generated dataset, a Gaussian diffusion model was trained with Mean Squared Error loss and used to infer new circuits. In Figure 16 the resulting circuits can be seen. The circuits are padded to have a shape of 16 by 16 causing noise in the padded area. Within the non-padded area, the generated circuits are not functional with several being unconnected.

Figure 14: One-hot encoded binarized MNIST generated by a categorical diffusion model.

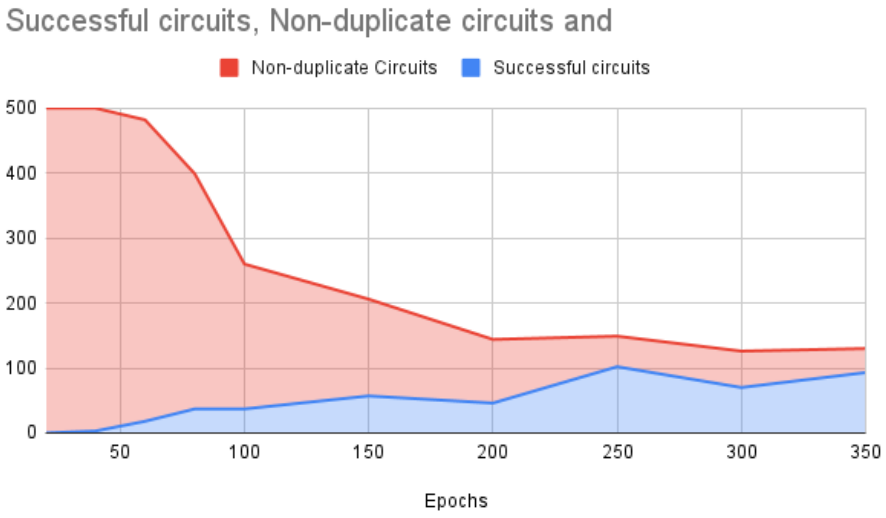


Figure 15: Sampled data of the circuits generated per batch across epochs of training. The amount of unique circuits falls as the generative model continues training, while the amount of these that can be simulated increases.

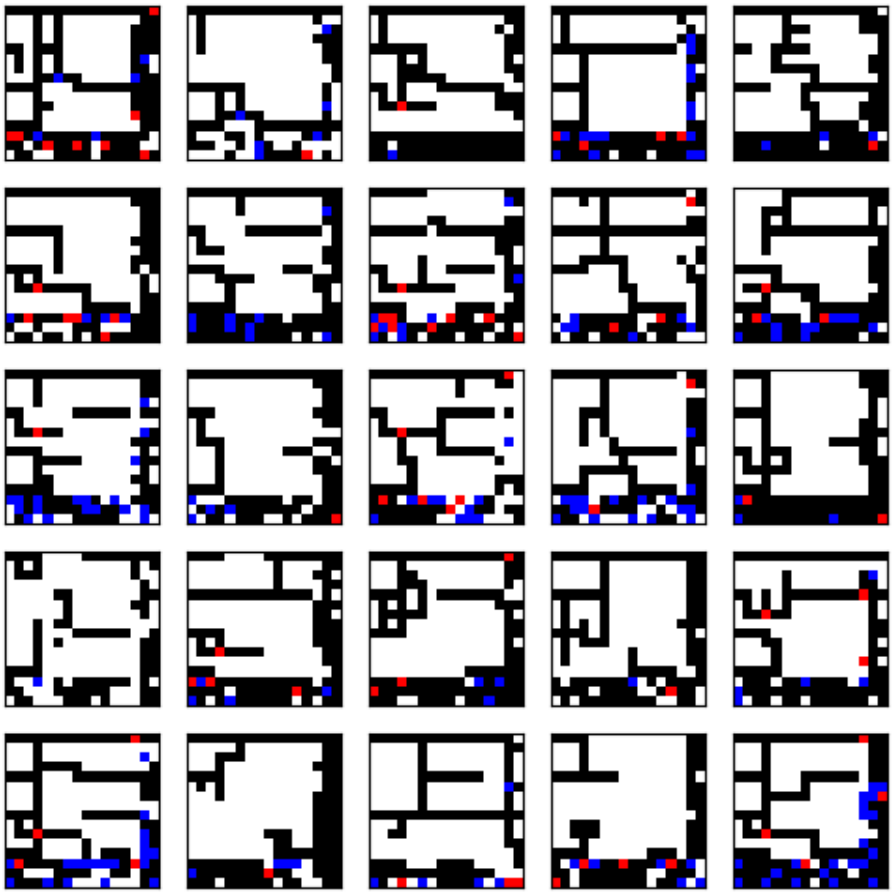


Figure 16: Diffusion model trained with Gaussian noise, on a generated dataset. Images are padded causing noise in the bottom right, however, within the non-padded sections, circuits are not complete.