Evolving Neural Circuit Policies

Chace Caven
Georgia Institute of Technology
Atlanta, GA
chace@gatech.edu

Abstract

Analyzing the internals of complex signals is typically achieved via autoencoders or contrastive learning. We propose an alternative: designing small and sparse causal generative models via an evolutionary search algorithm. Instead of producing a latent space, our technique produces computation graphs that model systems. When tested on a sample synthetic dataset generated by a linear dynamical system, our technique requires a fewer number of candidates than a randomized search to find sparse networks that fit the data well. Additionally, we discovered that the nature of the evolutionary approach optimized for networks that converged faster under a fixed number of training steps without overfitting. This opens a new method of analyzing non-convex optimization: using non-differentiable search methods to produce hypotheses for future research. For example, a sparse structure discovered here could be scaled up and analyzed for similar convergence properties, or theoretically analyzed.

1. Introduction

In signal processing applications, such as time series forecasting, predictive control, and electroencephalogram (EEG) analysis, a challenge is to extract "latent factors" from noisy complicated data via an approximate bijection from a high-dimensional input space to a low-dimensional latent space. Then, latent factors are used for clustering, classification, or prediction.

Ideally, the latent factor approach produces a richer representation of an input signal that simplifies the problem for human scientists. Although this approach has had success in some domains, it is a constant problem to "de-tangle" latent spaces to make them more interpretable and useful.

This project aims to develop a new approach to understanding the internals of complicated systems. Although latent spaces offer high-level details, this project aims to capture low-level details by constructing a causal generative model of the underlying system.

While this could be accomplished with a dense state-space model trained auto-regressively, such an approach lacks any form of explainability and thus does not help understand the underlying system. Instead, this project aims to create a state-space model with a high degree of sparsity that may be useful for understanding underlying systems.

For example, when analyzing brain activity via EEG, the input, output, and even brain activity in certain regions is known. The unknowns are the mechanisms by which neural populations communicate and deliver certain behaviors in response to stimuli.

1.1. Problem Statement

In this article, we will explore, motivate, and test a class of signal analysis problems where the desired result is a maximally simple policy that explains changes in observed variables.

The inputs of this problem are three time series: (1) stimuli, (2) observed hidden states, and (3) observed behaviors.

The output will be a policy that connects current stimuli and past hidden states to behaviors and future hidden states. The exact structure of the policy is explained in detail in Section 3.

We propose an evolutionary search algorithm to design neural network architectures that solve this signal analysis problem. The evolutionary search is explained in Section 3.

1.2. Motivation

In the field of deep learning and signal processing, many resources are put into large, deep, and saturated neural networks that take long training runs to converge.

This work aims to tackle the opposite problem: designing small and sparse neural networks with limited data supply that converge quickly without overfitting.

In the short term, this effort directly helps fields such as EEG analysis, where data supply is limited. In the long term, harnessing genetic algorithms for neural network design has interesting applications in control theory, two-player games, and other reinforcement learning problems.

The initial aim of this project was simply to design computation graphs that explain timeseries data. In practice, this was implemented as training networks for a fixed number of steps. Then, by the nature of the genetic algorithm, a secondary objective emerged. Networks that may end up fitting well but take too many training steps to obtain a low error, will be filtered out by the evolutionary process. Also, networks that fit very well but overfit will report higher validation loss and be filtered out by survival of the fittest.

This opens an interesting field within the study of optimization: do some architectures that have inherently better learning properties? If so, can we search for those architectures with a genetic algorithm? Do convergence properties of small structures stay as the number of parameters is increased?

2. Related Work

1. *Liquid Neural Networks* - Recurrent neural networks defined by neural ODEs are effective at modeling time series data. Hasani et. all [3] propose a new formulation of a continuous-time recurrent network that enables a dynamic time constant, represented by the following equation:

$$\frac{d\mathbf{x}}{dt} = -(\frac{1}{\tau} + f(\mathbf{x}, \mathbf{I}, t, \theta))\mathbf{x} + f(\mathbf{x}, \mathbf{I}, t, \theta)A \qquad (1)$$

   where $\mathbf{x}$ is the hidden state, $\mathbf{I}$ is the input at time t, and f is a function parameterized by θ. This is important as the time-constant of the network at time t is

$$\tau_{sys} = \frac{\tau}{1 + \tau f(\mathbf{x}, \mathbf{I}, t, \theta)} \qquad (2)$$

   most notably a function of the input $\mathbf{I}$. Because the time-constant of the ODE changes based on different inputs, these networks are called *liquid time-constant*.
2. *Neural Circuit Policies* - Lechner et. all [4] extend liquid time-constant networks in a sparse network design inspired by the *C. elegans* nematode. They propose an algorithm for wiring a 4-layer sparse network composed of liquid time-constant cells. The approach is validated on a car-driving task. A single algorithm with 19 control neurons and 253 synapses learns to map 32 input features into steering commands.
   The resulting system showed greater generalizability and robustness compared to systems with orders-of-magnitude larger black-box systems.
3. *Fixed-topology evolutionary search* - Genetic algorithms are efficient parallel search processes used for non-differentiable optimization problems. *Neuroevolution* refers to optimizing the weights of a neural network in this manner.
   Gomez [1] introduces methods for a *fixed-topology* search, i.e., where the structure of the neural network is fixed and only the weights are optimized. The dissertation develops three components: (1) the genetic algorithm itself, (2) an incremental evolution objective, and (3) a technique for making such networks robust to input noise.
4. *NeuroEvolution of Augmenting Topologies (NEAT)* - Stanley and Miikkulainen [5] present a joint optimization neuroevolution technique where weights and structure are simultaneously evolved. They introduce three ideas: (1) using historical markers to align genomes during crossovers, (2) splitting populations into species to protect structural innovation, and (3) starting from a maximally simple network and complexifying from there.
5. *Synthesis of Tailored Architectures (STAR)* - Thomas et all. [6] propose searching over very large (100-300 million parameter) model architectures via an evolutionary approach. Each candidate network is characterized by a string of layer types and a dictionary of residual connections. To evaluate a candidate, a model is constructed by stacking the appropriate layers and adding the denoted skip connections. After the model is trained, the "fitness" of the model is a mixture of the evaluation score, the number of parameters, and the cache size. By simultaneously optimizing these metrics, the authors produce novel, cheaper, and better models.

This work uses neural circuit policy architectures and performs a evolutionary search similar to NEAT. However, unlike NEAT, we do not use the genetic algorithm to optimize for network parameters. Instead, as described in Section 3: Approach, we use the genetic algorithm to discover topologies (i.e., architectures) which are optimized well by gradient descent.

3. Approach

The bulk of this work is creating a neuroevolution approach to designing small, sparse liquid time-constant networks. While each individual network is optimized via gradient descent, different network structures are searched via a genetic algorithm.

3.1. Network Definition

We consider a recurrent, sparse liquid time-constant network. Consider a directed graph $G = (V, E)$. Each vertex $v_i \in V$ have hidden state x_i and parameters leakage conductance g_i, membrane capacitance C_{m_i}, and resting potential x_{leak_i}.

Each edge $e_{ij} \in E$ is described by a synaptic weight w_{ij}, reversal potential E_{ij}, and two parameters γ_{ij} and μ_{ij}, such that we have the modified sigmoid activation

$$\sigma_{ij} = \frac{1}{1 + \exp\left(-\gamma_{ij}(x_j - \mu_{ij})\right)} \tag{3}$$

Let $\tau_i = C_{m_i}/g_i$ be the time-constant of neuron i. Then we define the effect of neuron j on neuron i as follows:

$$\dot{x}_i = -(\frac{1}{\tau_i} + \frac{w_{ij}}{C_{m_i}}\sigma_{ij})x_i + (\frac{x_{leak_i}}{\tau_i} + \frac{w_{ij}}{C_{m_i}}\sigma_{ij}E_{ij}) \tag{4}$$

This system is approximated by a semi-implicit Euler approach with a fixed step size, enabling backpropagation through time.

In practice, G has fixed vertices for input and output values. Then, to evaluate the network, input vertices take the values of corresponding inputs, and the values of output vertices are considered to be the output of the network.

3.2. Genomes

We follow the approach given by the NEAT algorithm. We maintain a collection of candidates, each described by a *genome*. The genome is defined in such a way where it can be used to construct a network (the phenotype).

For this work we choose an explicit representation of a directed graph G in the genome. Each genome consists of a variable number of node genes and connection genes. Node genes have an ID number and a type, either "input", "hidden", or "output". Connection genes have the IDs of the nodes they connect, a flag to denote whether it is enabled or disabled, and an "innovation number" which assists in aligning genomes for crossovers.

While in the original NEAT work, network parameters were included in the genome, in this work, network parameters are treated as trainable via gradient descent. This serves to simplify the genome and reduce the number of independent variables in the study.

3.3. Fitness

For each item in the solution space (i.e., possible genomes), we compute the *fitness* of that item by training via gradient descent for a fixed number of steps.

Given an input sequence (I_i), hidden sequence (x_i), and output sequence (o_i), we train a recurrent network described in section 3.1 in the following way: given subsequences $I_{0:t}$ and $x_{0:t}$, predict x_{t+1} and o_{t+1}.

To evaluate a genome, first the corresponding network is constructed. The Adam optimized is used to train the network on a training set for a fixed number of epochs. After training, the network is evaluated on a validation dataset.

The fitness of an individual is then the negative log of the mean squared error, i.e.,

$$R = -\log \frac{1}{N} \sum_{n=1}^{N} ||y_n - \hat{y_n}||^2 \tag{5}$$

In early testing, we tried using the raw mean squared error value. However, this leads to the genetic algorithm overly penalizing poor performance and under rewarding good performance.

3.4. Mutations

A core part of the genetic algorithm is to increase genetic diversity through randomized mutations. In this study, there are two types of mutations:

1. *Add Connection*. In this mutation, a single connection gene is added that initializes a new directed connection between existing nodes in the genome.
2. *Add Node*. In this mutation, we consider an existing directed edge AB. A new node C is created. We also create two edges, AC and CB, and disable edge AB.

3.5. Crossovers

Because different graph structures may contain different nodes and vertices, it is non-trivial to compute the "midpoint" between two candidates. To handle this situation, we follow the NEAT algorithm and align genes based on common history; more specifically, each time a mutation occurs an "innovation number" is attached to that particular gene. When two genomes undergo crossover, corresponding innovation numbers are matched and attributes of each gene are randomly selected. Genes without matching innovation numbers are called disjoint. We keep all disjoint genes from the higher-fitness parent and discard other disjoint genes.

3.6. Speciation

When a mutation occurs, it is unlikely that the new structural addition is immediately useful. Thus, to protect structural innovation, we split the whole population into buckets, or species, based on genetic similarity.

In the course of the genetic algorithm, individuals compete within their respective species, not within the global population. This enables new structures to optimize and achieve better performance that may require multiple steps.

In order to compute genetic similarity between two genomes, we take the individual differences between shared genes and a fixed penalty for each disjoint gene. The total is divided by the number of total nodes and total connections in the network. We use a fixed threshold to determine whether two genomes belong to the same species.

3.7. Evolution Hyperparameters

We initialize our evolutionary population with 25 candidate networks, each with 10 neurons and a sparsity level of 0.6 randomly initialized following the procedure in [4]. Each generation, the bottom 50% of individuals are discarded. The remaining individuals are paired and undergo crossovers as described in Section 3. After 25 new candidates are generated via crossover, there is a 40% chance of

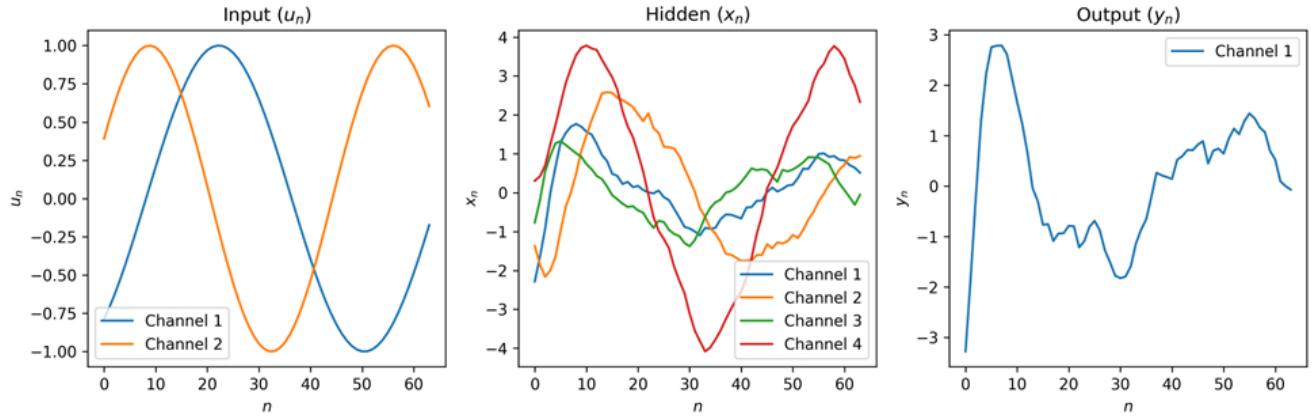

Figure 1. Example item from the synthetic dataset. Candidate networks take in a subsequence $u_{0:t}$ and $x_{0:t}$ and predict y_t and x_{t+1}. We initialize via $x_0 \sim \mathcal{N}(0, 1)$. The training dataset is 4096 samples and the validation dataset is 512 samples. Each candidate network is trained for 2 epochs on the training set, then frozen and evaluated on the testing set.

mutating via adding a connection, 10% chance of mutating via removing a connection, 20% chance of mutating via adding a neuron, and 15% chance of mutating via removing a neuron. There is also a 5% chance of a synapse switching polarity.

When evaluating each candidate, we use the Adam optimizer as implemented in the PyTorch framework. We use a fixed learning rate of 0.1 and momentum coefficients $\beta_1 = 0.9$, $\beta_2 = 0.999$. We use a fixed batch size of 64, and train for 2 epochs over the training dataset. We do not shuffle the training dataset.

The evolutionary algorithm runs until an individual has reached 1.0 fitness, i.e., a validation loss of 0.36. This threshold is arbitrary and mainly serves as a natural stopping point for the experiment.

3.8. Baselines

In this study we care about producing network *structures* that achieve a small test error, not necessarily about the test error itself. This differs from other studies which base their evaluations on the performance of produced models.

The baselines are then other methods of producing sparse network structures. The main comparison questions will be:

1. Does the evolutionary approach take fewer evaluations to discover higher performing individuals than randomized search?
2. Do evolved networks converge faster under fixed training steps?
3. Does the evolutionary approach generate consistent structures or substructures that is not found in randomized search?

This study will be successful if the evolutionary approach is able to more quickly find higher scoring candidates. It would also be very interesting if the evolutionary process creates substructures with good convergence properties, although we only do qualitative evaluations as we do not propose a precise definition of "good convergence".

4. Results

4.1. Dataset

In this study, we look at two problems: (1) developing generative causal models and (2) optimizing for fast convergence under a fixed number of training steps.

However, convergence is heavily related to the dataset used, as each gradient descent step is based on the loss value, which is based on the input given to a network.

To reduce the number of variables, we synthetically generate all of the data used in this experiment. As we are searching over many sparse networks, we want a task that is "easy" for a large fully-connected network but "hard" as networks become more sparse. We also want some ground-truth information about the internal system our approach is trying to model. To conform to these goals, we directly generate:

- inputs $(u_i)_{i \in \mathbb{N}} \subset \mathbb{R}^2$
- hidden states $(x_i)_{i \in \mathbb{N}} \subset \mathbb{R}^4$
- outputs $(y_i)_{i \in \mathbb{N}} \subset \mathbb{R}$

which are related via a linear dynamical system parameterized by matrices $A \in \mathbb{R}^{4\times4}$, $B \in \mathbb{R}^{4\times2}$, $C \in \mathbb{R}^{1\times4}$, and $D \in \mathbb{R}^{1\times2}$. The system evolves as follows:

$$x_{i+1} = Ax_i + Bu_i + w \tag{6}$$

and

$$y_i = Cx_i + Du_i + v \tag{7}$$

where $w \sim \mathcal{N}(0, \sigma_w^2 I)$ and $v \sim \mathcal{N}(0, \sigma_v^2 I)$ In this task, we set $\sigma_w^2 = 0.01$ and $\sigma_v^2 = 0.1$.

In this experiment, we generate sequences of length 64. This length is arbitrary; we considered that longer sequences would produce the same experimental results at a greater computational cost, but shorter sequences may lead to different experimental results.

To achieve a simplified system, we also produce simplified inputs. To produce the ith coordinate of an input sequence (o_n), we first pick frequency $\omega \sim \text{Uniform}(1, 5)$ and phase $\phi \sim \text{Uniform}(0, 2\pi)$. Then, we compute $(o_n)_i = \sin(\omega \cdot \frac{n}{64} + \phi)$

We also pick a fixed symmetric matrix A that leads to a stable system. We set

$$A = \begin{pmatrix} 0.8 & -0.3 & 0 & 0 \\ 0.3 & 0.8 & 0 & 0 \\ 0 & 0 & 0.7 & -0.4 \\ 0 & 0 & 0.4 & 0.7 \end{pmatrix} \tag{8}$$

The matrices B, C, and D are randomly sampled from $\mathcal{N}(0, 1)$ and do not contribute to the stability of the system.

4.2. Fitness and sparsity of evolved networks

The evolutionary algorithm ran for 76 generations before producing an individual with fitness higher than 1.0, evaluating a total of 3800 candidates, i.e., 50 per generation. From generation 0 to 50, candidate fitness increased without a major increase in the number of synapses present in each network. From generation 50 to 60, however, on average fitness did not change significantly but the number of synapses increased. See figure 2.

One conjecture to explain this phenomenon is that the evolutionary algorithm underwent overfitting; in order to squeeze more performance out of networks, it had to add more connections.

In the evolutionary process, we did not explicitly add a term penalizing networks with too many synapses. We conjecture that in the evolutionary process, it is helpful to have networks with too many connections, as they can mutate to create unexplored sparse structures. We leave this as a direction for future research.

One interesting note is that every generation contained poor performing networks. Fitness range $(-1.5, 0.5)$ contains candidates from generation 1 all the way to generation 76. This could be explained by failed mutations, i.e., candidates which had important neurons or synapses randomly deleted. Another possible explanation is that a population size of 50 enabled a wide range of individuals to survive each generation, i.e., at least 25 individuals were worse than around -1.0 fitness.

Regardless of reason, however, a high genetic diversity is a good sign that the evolutionary approach was a good fit for this particular optimization problem. Genetic diversity is discussed in more detail in [5].

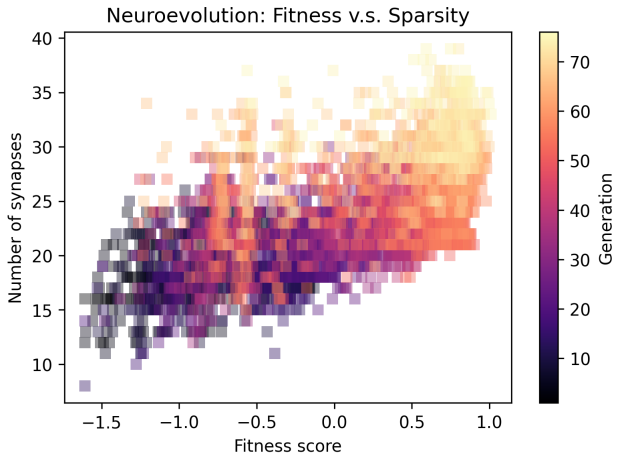

Figure 2. Each square represents a candidate network, containing some number of synapses, (i.e., liquid network connections) and producing some fitness score. The color of each square represents which generation the candidate came from. Darker squares are from earlier generations, and lighter squares are from later generations.

We now transition to analyzing properties of discovered networks, most importantly, training convergence.

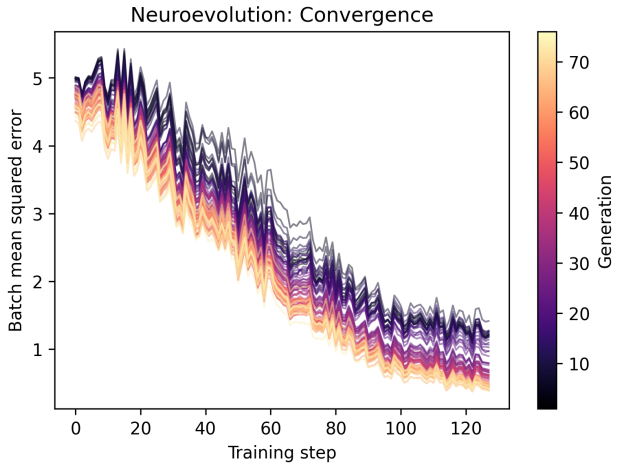

Figure 3. Each line represents the highest performing candidate network from each generation. Although fitness is measured through validation loss, here we plot training loss on each batch across the dataset. Darker colors are from earlier generations, and lighter colors are from later generations. Note the ordering of batches in the training dataset was not shuffled in between runs, explaining the similar jumps in train loss.

4.3. Convergence of designed networks

The primary question of this study is whether evolutionary algorithms can effectively design generative causal networks. A secondary question, however, is whether these designed networks exhibit interesting convergence properties. Recall that when evaluating each candidate, we train for 2 epochs. Note that we do not train until the model stops improving.

This implicitly changes the goal of the evolutionary algorithm to search for networks that *learn fast* rather than just *perform well*. To analyze this property, we take the highest-fitness candidate from each generation, re-initialize, and re-train for 2 epochs. Recall that before a candidate is trained, its parameters are randomly initialized. In theory, improvements in the genetic algorithm could be the result of random luck. To disprove this conjecture, we re-train the networks with a new random initialization.

We see that while the training curve of all candidates looks almost identical, as the ordering of batches is not randomized, candidates from later generations typically have curves shifted down and left, signifying getting better, faster.

Networks from later generations also exhibit a lower starting loss as well. This is an unexpected observation similar to the results of [2] that will be discussed in Section 5.

4.4. Neuroevolution vs. random search

One of the goals of the experiment was to compare neuroevolution to a randomized search. We hypothesized that an evolutionary approach could find higher-performing individuals faster than a randomized search. We hoped to use the randomization procedure from [4] as a comparison point.

However, upon analysis of the evolved networks, most have connections that have zero probability of being generated by the randomized algorithm. For example, evolved networks tend to have connections straight from input neurons to output neurons, where [4] never has connections of this type. This invalidated any point of comparison between randomized searches and the proposed evolutionary search.

We save for future research creating a more comparable randomized search to evaluate the cost-effectiveness of the evolutionary approach.

5. Conclusion

We introduce applying an evolutionary algorithm to design sparse causal generative models of systems with partially observed internals (i.e., hidden variables). Through nearly one hundred generations and a couple thousands evaluated candidates, we demonstrate the ability for a modified NEAT [5] to generate neural circuit policies [4] that autoregressively model a fixed linear dynamical system with sinusoidal inputs.

We find that designed networks (1) obtain lower loss values within the same number of training steps and (2) start with a lower loss value.

Considering (1), one conjecture for why some networks perform better relates to the mechanism by which gradients are copied upon an addition operator. If we have a neuron with many incoming synapses, its gradient is copied many times during backpropagation.

One observation is that evolved networks tend to have connections straight from input to output nodes. This could act as a ResNet-style skip connection and speed up convergence of networks.

Another possibility is that the networks themselves did not train faster, they just started better. In [2], Gaier et. all introduce the idea that some architectures are more stable under random initializations than others. It is possible that the evolutionary algorithm selected architectures that would start out performing better under random initialization.

5.1. A new avenue for studying optimization

As stated in the introduction, studying optimization of neural networks is difficult as there are few theoretical structures to rely on. Thus, optimization is studied empirically by the community of deep learning scientists.

An interesting side effect of this paper is applying automated procedures to empircally study optimization; in this case, deploying an evolutionary algorithm to search for structures that learn the best, the fastest. It seems this is a ripe area for future research, especially at a time when deep learning architectures are becoming more standardized, and new optimizers no longer outperform staples, i.e., Adam and AdamW.

References

[1] Faustino John Comez. *Robust Non-linear Control through Neuroevolution*. PhD thesis, University of Texas at Austin, 2002. 2

[2] Adam Gaier and David Ha. Weight agnostic neural networks, 2019. 6

[3] Ramin Hasani, Mathias Lechner, Alexander Amini, Daniela Rus, and Radu Grosu. Liquid time-constant networks, 2020. 2

[4] Mathias Lechner, Ramin Hasani, Alexander Amini, Thomas A Henzinger, Daniela Rus, and Radu Grosu. Neural circuit policies enabling auditable autonomy. *Nature Machine Intelligence*, 2(10):642–652, 2020. 2, 3, 6

[5] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002. 2, 5, 6

[6] Armin W. Thomas, Rom Parnichkun, Alexander Amini, Stefano Massaroli, and Michael Poli. Star: Synthesis of tailored architectures, 2024. 2