Deducing Gene Regulatory Networks Using Machine Learning

A Thesis submitted for the completion of requirements for the degree of

Bachelor of Science (Research)

by

Raj Paresh Mehta
Undergraduate Programme
Indian Institute of Science

Under the supervision of

Prof. Mohit Kumar Jolly
BioSystems Science and Engineering, Indian Institute of Science

Prof. Baladitya Suri
Instrumentation and Applied Physics, Indian Institute of Science

Certificate

I hereby certify that Raj Paresh Mehta, a fourth-year undergraduate student of the Bachelor of Science (Research) program at IISc, has worked under my supervision from May 2022 to April 2023. He worked on developing machine-learning models based on deep neural network architectures that can deduce Gene Regulatory Networks from gene-expression data. After going through his thesis, I have found it to be adequate for fulfilling the requirements of his BS (Research) degree.

.....

Dr. Mohit Kumar Jolly
Centre for BioSystems Science and Engineering
Department of Biological Sciences
Indian Institute of Science

Declaration

I, Raj Paresh Mehta, hereby declare that the material contained in this thesis represents original work undertaken by me at the Centre for BioSystems Science and Engineering, Department of Biological Sciences, Indian Institute of Science, between May 2022 and April 2023. I have made my best effort to acknowledge the work (of literary, empirical, and computational nature) of other members of the scientific community to the best of my knowledge. Any omission is unintentional and deeply regretted.

.....

Raj Paresh Mehta
Bachelor of Science (Research) Program
Physics Major
Indian Institute of Science

.....

Dr. Baladitya Suri
Department of Instrumentation and Applied Physics
Indian Institute of Science

Acknowledgements

First and foremost, I would like to thank my supervisor, Dr. Mohit Kumar Jolly, without whom this project would not have been possible. He has been wonderful in guiding me through this interdisciplinary topic, spending countless hours brainstorming on such a niche research area. He has been an amazing mentor, teaching me how to pursue science, research, and life in general. I want to thank him for helping me and getting me through all the obstacles I faced in the past year, something I will be eternally grateful for. He is a very kind and approachable person, a true role model who is fun and pleasant to work with. In the same spirit, I would like to thank Dr. Baladitya Suri, who has been very helpful and motivating not only in the past year but also during the time I studied and interned under him. He has been a great source of inspiration and has played a key role in turning me into an adept researcher. I also want to thank Dr. Jason George, Texas A&M University, for his invaluable input, whose novel perspective helped take this project to the next level.

Further, I also want to thank my labmates - Pradyumna, Kishore, and Sarthak for helping me with the logistics of using the computing clusters, using RACIPE, clearing my doubts, and suggesting the directions this project could take and guiding me through it. I consider myself fortunate to have had such a great support system throughout this project.

I would like to express my gratitude towards IISc and all of the professors for giving me this wonderful opportunity to study at such a prestigious university and providing an excellent education. I also want to thank KVPY and DST for supporting me through this journey.

Finally and above all, I would like to thank my parents, Dr. Paresh Mehta and Dr. Shetal Mehta, and my sister, Dr. Mahendi Mehta, for supporting and encouraging me throughout my undergraduate years, being a constant source of inspiration and affection, helping me through my toughest moments and making this journey pleasant and enjoyable.

Abstract

A Gene Regulatory Network (GRN) is a global map of various physical and biochemical interactions between genes and gene products. They can be mathematically represented as directed graphs and are usually modeled as a set of coupled Ordinary Differential Equations (ODEs). With the improvement of gene expression profiling techniques and the subsequent plethora of gene-expression data generated, the necessity of frameworks that can solve the inverse problem of reconstructing the GRNs reliably from the data is higher now than ever. In this thesis, we propose three machine-learning models with different deep neural network architectures that can deduce GRNs from the gene-expression data assuming the system of genes is modeled as coupled ODEs. We present the benchmarking of these models done on a two-gene system, the toggle switch, and a three-gene system, the toggle triad, which are considered fundamental network motifs. We show that all three models have great performance in predicting the correct GRN (toggle switch/toggle triad) given only the gene-expression data and how they can be improved even more, followed by how such a framework can be used for larger GRNs using edge-deletion and re-training techniques and other possible architectures that can achieve this goal.

List of Figures

Contents

Part I

A Brief Introduction to Neural Networks and Gene Regulatory Networks

Chapter 1

Artificial Neural Networks

1.1 Why Neural Networks?

Before delving into the architecture of Artificial Neural Networks (ANNs) and how it makes them powerful function approximators, we must understand why we need them in the first place.

The simplest form of function approximation is linear regression, a form of supervised learning. In single-variable linear regression, we have an input, independent variable x, a target, dependent variable y, and a linear function f of the form

$$f(x^{(i)}) = \theta_0 + \theta_1 x^{(i)} = \hat{y}^{(i)}$$

The function f is parameterized by two parameters θ_0 and θ_1, which are obtained by minimizing the Mean Squared Error:

$$E = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})^2$$

(where m is the number of training examples). Hence, $f(x^{(i)}) = \hat{y}^{(i)}$ is the predicted value for the i-th example such that the l_2-norm between $y^{(i)}$ and $\hat{y}^{(i)}$ is minimum.

This can be generalized to multivariate linear regression, where instead of just one input variable, we have n variables which can be represented as an $(n+1)$-dimensional feature vector $\mathbf{x}$ with x_0 always equal to 1 and x_i, for $i = 1$ to n, being the input variables [1]. This allows us to have a compact representation for the function f, which we now represent by the hypothesis function h_θ, given by

$$h_{\boldsymbol{\theta}}(\mathbf{x^{(i)}}) = \boldsymbol{\theta} \cdot \mathbf{x}^{(i)} = \hat{y}^{(i)}$$

Here $\boldsymbol{\theta}$ is the parameter vector with θ_0 being the bias term and θ_1 to θ_n being the feature weights. The parameter vector is obtained by minimizing the following Mean Squared Error:

$$MSE = \frac{1}{m}\sum_{i=1}^{m}(\boldsymbol{\theta}^T\mathbf{x}^{(i)} - y^{(i)})^2 \tag{1.1}$$

We can define a feature matrix $\mathbf{X}$ and target matrix $\mathbf{y}$ such that each row of $\mathbf{X}$ is the the i-th feature vector $\mathbf{x}^{(i)}$ and each row of $\mathbf{y}$ is the i-th target value $y^{(i)}$. Minimizing the Mean Squared Error in Eq. (1.1) gives us the normal equation solution

$$\hat{\boldsymbol{\theta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{1.2}$$

$\hat{\boldsymbol{\theta}}$ minimizes MSE.

In terms of computational complexity, this process would be $\mathcal{O}(n^{2.4})$ to $\mathcal{O}(n^3)$ (depending on the method) since the most computationally intensive part is inverting the matrix $\mathbf{X}^T\mathbf{X}$ which is a $(n+1) \times (n+1)$ matrix [1]. Hence, using the normal equation to find the optimum value $\hat{\boldsymbol{\theta}}$ becomes very inefficient when the number of features, n, becomes very large.

In terms of model complexity, or model capacity, a multivariate linear regression model would be insufficient in most real-world scenarios. Since the model is linear in each feature, it would be unable to capture non-linearities in real data, and thus, the hypothesis function it fits won't be able to capture such intricacies. We could try using polynomial regression, where the powers of each feature are added as new features. However, using a d-degree polynomial would increase the number of features from n to $\frac{(n+d)!}{d!n!}$, hence for a large number of initial features n, we won't be able to add much more non-linearity as the size of the matrix $\mathbf{X}^T\mathbf{X}$ would blow up. Furthermore, as d increases, not only will the model start to over-fit the training data, hence losing generalizability and prediction power on unseen data, but we may also start observing Runge's phenomenon - the oscillation at the edges of an interval which occurs when polynomials of high degree are used for polynomial interpolation over equidistant interpolation points. (Appendix A)

Hence, we need a high-capacity model that is not only computationally less expensive to train but also captures the trends in the data well while maintaining generalizability to predict correctly on unseen data. This is where Artificial Neural Networks come into the picture.

1.2 Artificial Neural Networks

Artificial Neural Networks (ANNs) or Multi-Layer Perceptrons (MLPs) comprise multiple layers of nodes or artificial neurons stacked together, containing an input layer, one or more hidden layers, and an output layer [2]. In a fully connected neural network, each node in one layer is connected to each node in the subsequent layer and has a weight and a bias associated with it. Information from nodes in one layer is passed onto the nodes in the next layer through an activation function.

Since the *Perceptron* model, invented by Frank Rosenblatt in 1957, the field of neural networks and deep learning has come a long way. A neural network with only one hidden layer is considered a *Simple Neural Network*, whereas neural networks with two or more hidden layers are considered to be *Deep Neural Networks* (DNNs). Among DNNs, the most commonly used architectures are *Convolutional Neural Networks* (CNNs), *Recurrent Neural Networks* (RNNs), and *Transformers*.

RNNs are suitable for learning from sequential/time-series data and are widely used to solve temporal problems, and they have a unique property of "memory" [3]. Most Deep Neural Networks assume that their input and output have no temporal correlation, whereas RNNs use information from previous time steps to influence how information in the current time step produces an output. However, in the context of my thesis's objective, it is unnecessary to use RNNs, because the time-evolution of gene expression is controlled by a first-order differential equation. Hence, as I will elaborate on later, using a simple, fully-connected neural network is sufficient in the current context.

In order to understand how information flows in feedforward neural networks, consider any two layers of a neural network, Layer A having m neurons and Layer B having n neurons. Let the current value of the i^{th} neuron in the k^{th} layer, also called its activation, be denoted by a_i^k, let w_{ij}^k denote the weight between the i^{th} neuron of the k^{th} layer and j^{th} neuron of the $(k+1)^{th}$ layer, and let b_i^k be the bias associated with the i^{th} neuron in the k^{th} layer. Then, a forward pass from layer A to layer B will give,

$$\begin{bmatrix} a_1^B \\ a_2^B \\ \vdots \\ a_n^B \end{bmatrix}^T = \mathrm{f_{act}} \left(\begin{bmatrix} a_1^A \\ a_2^A \\ \vdots \\ a_m^A \end{bmatrix}^T \times \begin{bmatrix} w_{11}^A & w_{12}^A & \cdots & w_{1n}^A \\ w_{21}^A & w_{22}^A & \cdots & w_{2n}^A \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1}^A & w_{m2}^A & \cdots & w_{mn}^A \end{bmatrix}_{m \times n} + \begin{bmatrix} b_1^B \\ b_2^B \\ \vdots \\ b_n^B \end{bmatrix}^T \right)$$

where $\mathrm{f_{act}}$ is the activation function and acts on the resulting vector element-wise.

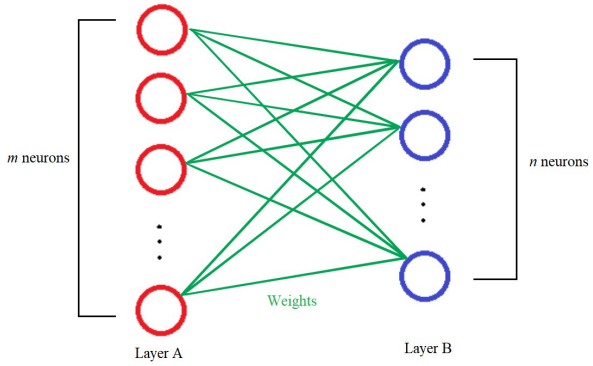


Figure 1.1: Connections between two layers of a fully-connected neural network

The most common activation functions used are *sigmoid, tanh,* and *ReLU*:

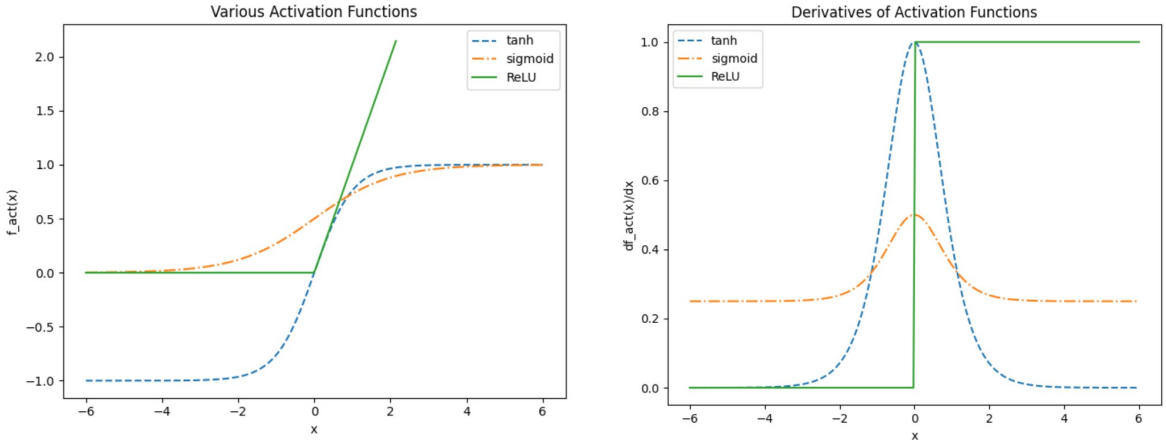


Figure 1.2: Activation Functions

As we can see from the graphs, sigmoid, and tanh activation functions are really good for introducing non-linearities in our model, increasing its capacity, however, they suffer from vanishing gradients as x moves away from zero. On the other hand, ReLU has a constant derivative of 1 as long as $x > 0$, which makes it a suitable activation function if we want fast convergence to a local minima, as we shall see in the next section.

1.3 Optimization Algorithms

For any supervised learning problem, we can define a cost function that measures the difference between the model's prediction and the target value for each input feature vector, which reduces our goal of training the model to optimizing the model parameters such that this difference is minimum. The most commonly used cost functions are:

1. Mean Squared Error: This is commonly used in regression problems, and is defined

as

$$E(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m}(h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}) - y^{(i)})^2$$

2. Binary Cross-Entropy Loss: This loss function is commonly used in binary classification problems and is defined as:

$$E(\boldsymbol{\theta}) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)}\log(h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)})) + (1 - y^{(i)})\log(1 - h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}))$$

Here, the hypothesis function $h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}) \in [0, 1]$ and thus outputs the probability of input $\mathbf{x}^{(i)}$ being of class 1.

3. Categorical Cross-Entropy Loss: This loss function is commonly used in multiclass classification and is defined as:

$$E(\boldsymbol{\theta}) = -\frac{1}{m}\sum_{k=1}^{n}\sum_{i=1}^{m} y_k^{(i)}\log(h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)})_k)$$

for n classes.

Hence, the learning problem is reduced to finding the optimum parameters $\hat{\boldsymbol{\theta}}$ such that

$$\hat{\boldsymbol{\theta}} = \min_{\boldsymbol{\theta}} E(\boldsymbol{\theta})$$

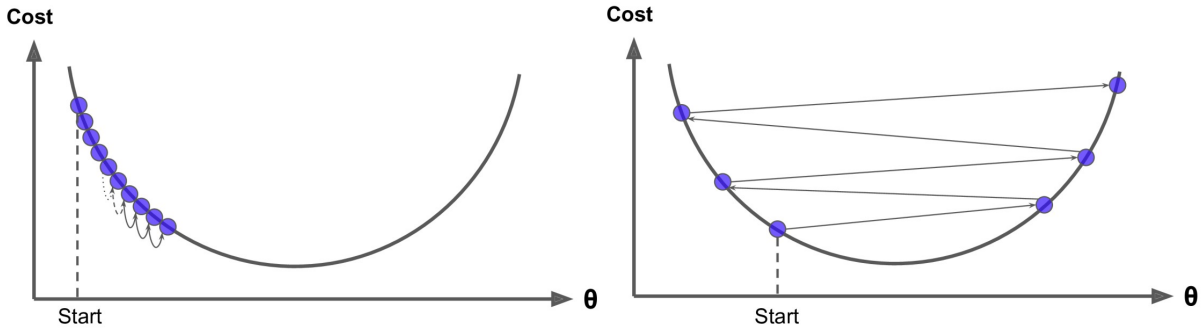
Traditional nonlinear optimization algorithms like the Newton method or the BFGS algorithm have high convergence rates; however, they are only suitable when the Hessian of the function we are trying to minimize can be computed (since they are second-order optimization algorithms), and it has only one local minima. Neural networks, especially deep neural networks, have multiple local minima, and its computationally very expensive to compute second-order derivatives. Hence, we must use first-order optimization algorithms like gradient descent for neural networks.

1.3.1 Gradient Descent

Gradient descent is an iterative optimization algorithm that tweaks the model parameters at each step in the direction that locally minimizes the loss function the most. After random initialization of the parameter vector $\boldsymbol{\theta}$, at each iteration, n, the parameter vector $\boldsymbol{\theta}^{n+1}$ is computed from the current parameter vector $\boldsymbol{\theta}^n$ as follows:

$$\boldsymbol{\theta}^{n+1} = \boldsymbol{\theta}^n - \eta\nabla_{\boldsymbol{\theta}}E(\boldsymbol{\theta}^n)$$

Here, η is the *learning rate* and is a very important hyperparameter. If we use a very low learning rate, then a large number of iterations will be required before we reach a minima; however, if we use a very high learning rate, then we may overshoot a minima. Figure 1.3 shows this for a convex function.



(a) Too low learning rate η (b) Too high learning rate η

Figure 1.3: Problems with sub-optimal learning rate. Ref: [1]

The second important factor is the calculation of the gradient $\nabla_{\boldsymbol{\theta}} E(\boldsymbol{\theta})$.

1. Batch Gradient Descent: In this version of gradient descent, at each step, the gradient is computed using the entire training data (using the full *batch*), i.e., the loss function is computed using all training examples, and the gradient is computed with respect to this. This form of gradient descent is feasible only when the number of training examples is low. Otherwise, calculating the loss using all training examples at each step will be computationally very expensive.

2. Stochastic Gradient Descent: In this version, the loss function is computed using only one training example, and the gradient at that step is computed with respect to it. This training example is randomly picked from the training set and is unique at each step. Once all training examples have been used, the entire process can be repeated again. Going through the entire training set counts as one *epoch*. Optimization can be performed for multiple epochs until we reach a local minima. Since at each step, only one random training example is used, stochastic gradient descent is much more irregular than batch gradient descent, and the algorithm may not reach the minimum value but keep bouncing around it. On the other hand, this may be helpful as the algorithm can jump out of, and hence not get stuck in a local minima, and explore a much larger part of the cost landscape than batch gradient descent, hopefully reaching a global minima.

3. Mini-Batch Gradient Descent: In this version, instead of computing the loss using the entire training set or only one random training example, the loss is computed using a few number of training examples at each step, and the gradient is computed

with respect to it. This number is called the *batch size* and is another important hyperparameter, apart from the number of epochs. Since a few training examples are used at each step, it is slightly more regular than stochastic gradient descent but may struggle with getting stuck in a local optima.

1.3.2 Adam Optimization

The *Adam* optimization algorithm is a really popular first-order gradient-based optimization algorithm proposed by Diederik P. Kingma and Jimmy Lei Ba at the 3rd International Conference for Learning Representations, San Diego in 2015 [4]. *Adam* is based on adaptive estimates of lower-order moments, specifically the first- and second-order moments of the gradients. It uses these estimates to compute the individual adaptive learning rates for different parameters, It is also computationally efficient and has little memory requirements.

The algorithm updates the parameters $\boldsymbol{\theta}$ as follows, (as given in [4]): At each iteration step n, we compute

$$\mathbf{g}_n = \nabla_{\boldsymbol{\theta}} E(\boldsymbol{\theta}_{n-1})$$

$$\mathbf{m}_n = \beta_1 \cdot \mathbf{m}_{n-1} + (1 - \beta_1) \cdot \mathbf{g}_n$$

$$\mathbf{v}_n = \beta_1 \cdot \mathbf{v}_{n-1} + (1 - \beta_2) \cdot \mathbf{g}_n^2$$

$$\hat{\mathbf{m}}_n = \frac{\mathbf{m}_n}{1 - \beta_1^n}$$

$$\hat{\mathbf{v}}_n = \frac{\mathbf{v}_n}{1 - \beta_2^n}$$

$$\boldsymbol{\theta}_n = \boldsymbol{\theta}_{n-1} - \alpha \cdot \frac{\hat{\mathbf{m}}_n}{\sqrt{\hat{\mathbf{v}}_n} + \epsilon}$$

(All operations here, multiplication, square-root, squaring, division, addition, and subtraction, are performed element-wise.)

The first-moment vector, $\mathbf{m}_n$, and the second-moment vector, $\mathbf{v}_n$ are both initialized to a vector of zeros. β_1, $\beta_2 \in [0, 1)$ are exponential decay rates for the moment estimates. Since we initialized the vectors $\mathbf{m}_n$ and $\mathbf{n}_n$ as zero-vectors, it biases the moment estimates towards zero. Hence, $\hat{\mathbf{m}}_n$ and $\hat{\mathbf{v}}_n$ are the bias-corrected first-moment and second raw moment estimates. ϵ is usually chosen to be a very small number, on the order of a millionth. α is the learning rate, which sets a soft upper bound to the adaptive learning rates for each parameter (Appendix B).

Chapter 2

Gene Regulatory Networks

2.1 What are Gene Regulatory Networks?

As gene expression profiling techniques improved in the late 1990s, inference of large-scale Gene Regulatory Networks (GRNs) became possible. We define a GRN as "a network that has been inferred from gene expression data"; however, a GRN may not correspond to the way the genes of interest interact with each other directly, as measured gene expressions can vary with molecular interactions, transcription regulations, and protein interactions [5]. Hence, GRNs provide a global map of various physical and biochemical interactions between genes and gene products.

Various common frameworks used to mathematically model interactions represented in a GRN include coupled ordinary differential equations, synchronous Boolean networks, asynchronous Boolean networks, and stochastic gene networks.

2.1.1 Coupled ODEs

Within this framework, gene expression levels are assumed to be continuous, non-negative real numbers whose values are governed by the following set of coupled ODEs:

$$\frac{dx_j}{dt} = \phi_j(x_1, x_2, \cdots, x_n)$$

for n genes. ϕ_j is a parametrized function that captures how gene expression levels of various genes $(x_1, x_2, \cdots, x_n)$ affect j-th gene's expression levels. Hence, GRNs can be modeled differently using coupled ODEs using different forms of the function ϕ_j. The most common form of ϕ_j uses non-linear shifted Hill functions:

$$\phi_j(x_1, x_2, \cdots, x_n) = G_j * \prod_{i=1}^{n} H^s(x_i, x_i^0, n_{x_i,j}, \lambda_{x_i,j}) - k_j * x_j$$

where

$$H^s(x_i, x_i^0, n_{x_i,j}, \lambda_{x_i,j}) = \frac{(x_i^0)^{n_{x_i,j}}}{(x_i^0)^{n_{x_i,j}} + (x_i)^{n_{x_i,j}}} + \lambda_{x_i,j} * \frac{(x_i)^{n_{x_i,j}}}{(x_i^0)^{n_{x_i,j}} + (x_i)^{n_{x_i,j}}}$$

G_j denotes the production rate, k_j denotes the degradation rate, x_i^0 denotes the threshold value, $n_{x_i,j}$ denotes the Hill coefficient, and, $\lambda_{x_i,j}$ denotes the fold change. All these values parameterize the function ϕ_j.

2.1.2 Boolean Networks

Within this framework, gene expression levels are assumed to be binary - whether a gene is active or its not. Each gene's value is governed by a Boolean function of all genes interacting with it. Furthermore, inputs from all genes can be combined either using an AND operation or an OR operation. For example, consider three genes, A, B, and C, with B activating A and C inhibiting A. Then, the value of A will be governed by the Boolean function

$$A = B \text{ OR } (\text{NOT } C)$$

where $A,\ B,\ C \in \{0, 1\}$ denote the gene expression levels of respective genes at a certain time.

In a GRN, if the gene expressions for all genes are updated simultaneously using Boolean functions, then we call it synchronous update, i.e., each of the Boolean functions takes gene expression levels at time $t-1$ to update gene expression at time t. On the other hand, if gene expressions are not updated simultaneously, then the Boolean functions will take some gene expression levels at time $t-1$ and some at time t to update gene expression at time t. This is known as an asynchronous update. In general, both synchronous and asynchronous Boolean networks converge to similar stable states.

2.1.3 Stochastic Gene Networks

When modeling a GRN using coupled ODEs, we assume that the concentration of various chemicals varies continuously and the dynamics it follows are deterministic. However, the chemical processes of gene expression (transcription and translation) are low-rate, and thus the number of molecules changes in discrete amounts and is governed by random, distinct reaction events [6]. The time evolution of such a system is stochastic of the Markov type, described by the chemical master equation (CME). The CME is a set of ODEs that describe the time evolution of a network of chemical reactions as a stochastic process. It assumes that the reactions occur in a spatially homogenous environment with a fixed volume and temperature. Solving the CME gives the population vector of

the chemical species in consideration at each point in time [7]. The CME is simulated using a modified version of the Gillespie algorithm, which can account for the multiple time-delayed reactions of the underlying biological processes (transcription/translation).

In general, modeling GRNs as coupled ODEs is the most popular method. This is because they give more resolution than Boolean networks and provide a sufficiently detailed view of the complex dynamics exhibited by the GRNs without using computationally intensive models like stochastic networks. Hence, moving forward, we will discuss the properties of GRNs when modeled using coupled ODEs.

The simplest GRN that one can form consists of two genes, whose most common example is the toggle switch, where both genes inhibit each other's activity (directly or indirectly). Due to this mutual suppression, such a GRN exhibits two stable states – (A-high, B-low) and (A-low, B-high), where A and B denote the expression levels of the two genes, respectively. These two stable states can either co-exist for a certain combination of parameters (forming a bistable system) or only exist individually for other combinations of parameters (forming a monostable system). We can study this transition between bistable and monostable behavior by picking one of the parameters and varying it over the entire range of biologically relevant values it can take. We call this parameter a bifurcation parameter, using which we can obtain a bifurcation diagram:

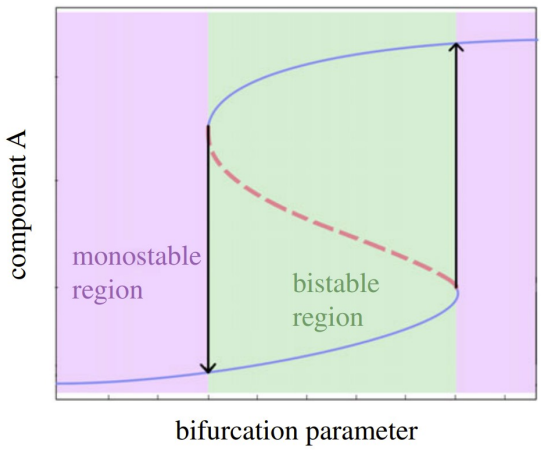


Figure 2.1: This diagram shows how a toggle switch can transition between monostable and bistable behavior. The solid blue lines indicate the stable states, while the red dotted line indicates unstable states. The black arrows show how the system transitions between different states - monostable (purple) and bistable (green). Ref. [8].

We can couple three toggle switches to make a three-gene system, a toggle triad. The toggle triad exhibits single positive states (A-high, B-low, C-low), (A-low, B-high, C-low), and (A-low, B-low, C-high), as well as double positive states (A-high, B-high, C-low), (A-low, B-high, C-high), and (A-high, B-low, C-high). It also exhibits monostability,

bistability, and tristability.

The toggle switch and the toggle triad GRNs hold high biological relevance as there are many examples of gene interactions that can be explained by these GRNs. For example, in hematopoietic stem cells, PU.1 and GATA1 mutually repress each other, which can convert a common myeloid progenitor cell to a myeloid cell fate (PU.1-high, GATA1-low) or an erythroid one (PU.1-low, GATA1-high) [9]. T-bet, GATA3, and RORγT form a toggle triad that leads to three different T-cell states Th1 (T-bet-high, GATA3-low, RORγT-low), Th2 (T-bet-low, GATA3-high, RORγT-low), and Th17 (T-bet-low, GATA3-low, RORγT-high) from a common progenitor state (CD^+T cell) [8].

Large regulatory networks also contain embedded toggle switch and toggle triad motifs, making their study and characterization fundamentally important. In the next section, we will discuss the various methods that have been developed till now for network inference for both small-scale and large-scale networks.

2.2 Various methods of constructing GRNs from gene expression data

One of the earliest models developed for network inference was probabilistic graphical models. In such models, various attributes of the underlying system, such as gene expression levels and cluster assignments of the genes, are assumed to be random variables. The model is then described as a joint probability distribution of all random variables. The joint probability distribution is represented as a product of terms, each product described by a graph relating the involved entities in the product [10].
In Bayesian networks, each product is represented as a conditional probability $P(X_i|\mathbf{U}_i)$ and the joint probability distribution defined over the set $X = \{X_1, X_2, \cdots, X_n\}$ of random variables is given by

$$P(X_1, X_2, \cdots X_n) = \prod_{i=1}^{n} P(X_i|\mathbf{U}_i)$$

where $\mathbf{U}_i \subseteq X$ are the set of variables defined as the "parent" of X_i. The graph consists of nodes representing the random variables and directed edges from the parents of X_i to X_i for all i. The conditional probability $P(X_i|\mathbf{U}_i)$ can be chosen to be any parametric probability representation that aligns with our knowledge and assumptions about the domain.
A second type of probabilistic graphical model is the Markov network, where the joint probability distribution is described as a product of potentials that capture interactions

among a small set of variables and how desirable it is to group them together:

$$P(X_1, X_2, \cdots X_n) = \frac{1}{Z} \prod_{i=1}^{n} \pi_i[\mathbf{C}_i]$$

where Z is the normalization factor and $\pi_i[\mathbf{C}_i]$ is the i-th potential over the variables $\mathbf{C}_i \subseteq X$. The parameter estimation of both $P(X_i|\mathbf{U}_i)$ and $\pi_i[\mathbf{C}_i]$ is usually framed as a Maximum Likelihood problem, and either of the models is chosen post parameter estimation via discrete optimization using scoring methods that measure how each model fits the observed data.

Xiujun Zhang et al. [11] frame the problem of deducing gene regulatory networks in the information-theoretic realm. They use Conditional Mutual Information (CMI) in conjunction with Path Consistency Algorithm (PCA) in order to deduce a sparse GRN from the available gene expression data. The CMI of two genes X and Y given a set of genes Z can be computed as

$$I(X, Y|Z) = \sum_{x \in X, y \in Y, z \in Z} p(x, y, z) \log \frac{p(x, y|z)}{p(x|z) \cdot p(y|z)}$$

where $p(x, y, z)$ denotes the joint probability distribution of x in X, y in Y, and z in Z, $p(x, y|z)$ denotes the joint probability distribution of x in X and y in Y given z in Z and $p(x|z)$ denotes the probability of discrete value x in X given z in Z (similar for $p(y|z)$). Using a Gaussian kernel probability density estimator and Shannon entropy, one can derive that

$$I(X, Y) = \frac{1}{2} \log \frac{|C(X)| \cdot |C(Y)|}{|C(X, Y)|}$$

and

$$I(X, Y|Z) = \frac{1}{2} \log \frac{|C(X, Z)| \cdot |C(Y, Z)|}{|C(Z)| \cdot |C(X, Y, Z)|}$$

where $C(\cdot)$, $|C(\cdot)|$ represents the co-variance matrix and its determinant.

The PC-Algorithm initially constructs a complete, undirected graph $\mathbf{G}$ of all genes, and in the L-th iteration of the algorithm, it first finds out the number T of adjacent genes of genes i and j. If $T < L$, the algorithm stops, else it selects L genes from T and computes the CMI of genes i and j against all such C_T^L combinations. If the maximum CMI obtained is less than some threshold θ, then the edge $G(i, j)$ is deleted. This method is highly accurate and can distinguish direct (causal) interactions from indirect associations. However, this method cannot deduce the directionality of the edges or determine regulatory relations with a time delay.

The above two methods focus on extracting only the topological information of GRNs; they do not extract the underlying dynamical model (in terms of differential equations)

associated with the GRNs. Yunfei Huang et al. [12] employ a sparse inference and active learning scheme where they try to deduce the differential equations governing the dynamics of the data using a library matrix $\boldsymbol{\Theta}(\boldsymbol{Z})$. The library matrix is constructed from linear combinations of suitable functions of $\boldsymbol{Z}$ (the input data containing gene expression values of all genes at all time steps) – powers of $\boldsymbol{Z}$, partial derivatives, and trigonometric functions. Then, the following relation is assumed

$$\mathcal{D}_t(\boldsymbol{z}_l) = \boldsymbol{\Theta}(\boldsymbol{Z})\boldsymbol{\xi}_l$$

where $\boldsymbol{z}_l$ denotes the gene expression data over time for the l-th gene and $\boldsymbol{\xi}_l$ denotes the weight vector for the l-th gene. $\mathcal{D}_t(\boldsymbol{z}_l)$ denotes the finite-difference approximation to the first derivative of $\boldsymbol{z}_l$ at time t. Hence, the problem is reduced to computing optimal $\boldsymbol{\xi}_l$ for the given library matrix and input data. This optimization process is carried out using automatic threshold sparse Bayesian learning (ATSBL). Hence, the resulting optimal $\boldsymbol{\xi}_l$ provides the function(s) contained in the library matrix that can best explain the dynamics shown by the data.

The above method is quite computationally intensive and requires an experienced user to define an appropriate library matrix. As opposed to this, models that do not require a library of functions they can fit the data to but instead can learn the function on their own would relieve them of both these issues. This is where we can employ machine learning strategies to construct models that can learn the regulatory functions on their own and deduce the network topology of the interacting genes, as well as the type and strength of these interactions.

Joanna E. Handzlik and Manu [13] used gene circuits, a data-driven predictive modeling methodology, to determine the non-linear ODEs governing the gene regulations. Gene circuits do not presuppose any particular scheme; instead, they consist of dense interconnections whose parameters are learned using a global nonlinear optimization technique Parallel Lam Simulated Annealing (PLSA). The gene circuit is modeled as a set of coupled ODEs:

$$\frac{dx_i^l}{dt} = R_i \cdot S\left(\sum_{j=1}^{N} T_{ij}x_j^l + b_i c^l + h_i\right) - \lambda_i x_i^l$$

where R_i denotes the maximum synthesis rate attained by the i-th gene, S is the sigmoid activation function, $T_{ij} \in \mathbb{R}$ denotes how the j-th gene regulates the i-th gene (positive for activation and negative for repression), h_i is the threshold value that determines the basal synthesis rate and λ_i is the degradation rate. N is the total number of genes, l denotes the lineage of the gene (neutrophil, progenitor, and erythroid), b_i denotes the implicit influence the lineage has on the model, and c^l is -1, 0, or 1 depending on the lineage. They trained their model on real gene expression data by May et al. [14] and obtained 100 independent models (PLSA being stochastic outputs different parameters

each time a model is trained) out of which 71 met their goodness criteria. They also found that the models train better when the data is not randomized and are robust up to $\pm 70\%$ perturbations in the initial conditions.

The python module GeneNet designed by Tom Hiscock [15] uses a similar strategy for defining gene circuits. The following set of coupled ODEs is used to model the gene circuit:

$$\frac{dy_i}{dt} = \phi\left(\sum_j W_{ij} y_j\right) + I_i - k_i y_i$$

where the values W_{ij} indicate how gene j interacts with gene i, y_i denotes the gene expression value of the i-gene at time t, k_i denotes the degradation rate, and I_i denotes any external input to the gene. The function ϕ is a nonlinear function ($\phi(x) = 1/(\exp(x)-1)$). It employs a Recurrent Neural Network model to optimize the parameters W_{ij} and k_i using automatic differentiation and Adam optimization. Mean squared error is used as a cost function. After the model has been trained once, it is trained once again with regularization using the learned parameters as the starting point. After this, the learned network is "pruned" - values of W_{ij} below a certain threshold ϵ are set to zero.
Huck Yang et al. [16] extended GeneNet to ES-GeneNet, which uses evolutionary algorithms to aid in searching the high-dimensional parameter space and gives significantly better results and performance compared to traditional Adam optimization, which can only explore a limited number of network topologies as it can get stuck in local optima.
An important thing to note about GeneNet and ES-GeneNet is that the cost function is defined only on the value one of the genes takes. Specifically, both these models constraint one of the genes, say y_1 to respond to some input, say x (i.e., $y_1 = x$) based on which the gene interactions produce the expression levels of all other genes at that time, out of which only one of the gene, say y_N is monitored to compute the MSE cost function:

$$C = \sum_x (y_N(x) - \hat{y}_N(x))^2$$

where N is the total number of genes. Hence, they do not perform global optimization based on gene expression levels of all genes.

The methodology devised by Jingxiang Shen et al. [17] works with the following set of coupled ODEs:

$$\frac{dg_i}{dt} = f_i(g_1, g_2, \cdots, g_N, I) - \gamma g_i$$

where g_i denotes the gene expression levels of the i-th gene, N is the total number of genes and I is the input impulse to the system. A neural network block generates the values of f_i from the gene expression levels and input impulse at a certain time t, whose value is then used to compute g_i at time $t + dt$ using a simple Euler step. Hence, their model resembles an RNN in that it has a directed loop. The cost function is MSE loss

defined with some or all of the genes' expression values. In order to ascertain how the j-th gene interacts with the i-th gene, they use the following quantity

$$\Delta_{ji} = f_i(\cdots, g_j) - f_i(\cdots, \lambda g_j)$$

where $\lambda \in (0, 1)$, called the discount factor, controls the magnitude of the perturbation used to measure the change in f_i upon a fold change in g_j. Another way of measuring this is by changing $f_i(\cdots, g_j)$ to $f_i(\cdots, \lambda g_j)$ in the set of coupled ODEs one at a time and measuring by what amount and in what direction (positive or negative) g_i changes. If the change is positive, then the j-th gene represses the i-th gene, and vice-versa. Using this method, one can deduce not only the type of interactions but also the strength of interactions by averaging over all values present in the training data. In order to deduce a sparse topology, one can start with a trained RNN, then remove the weakest link, train it again, and repeat the process until the sparse GRN modeled by the RNN can no longer fit the data well. This model performs well in predicting GRNs similar to the ground truth and scales well with an increasing number of genes too.

Part II

Deducing Gene Regulatory Networks

Chapter 3

Methods

3.1 Architecture of the System of Neural Networks

A group of genes forming a Gene Regulatory Network (GRN) can be modeled as a system of coupled ordinary differential equations. If we have n genes and $\mathbf{x}(t)$ is an n-dimensional vector denoting the gene expression levels at time t, then for the i-th gene, the differential equation looks like

$$\frac{dx_i}{dt} = f_i(\mathbf{x}(t)) - k_i x_i(t) \tag{3.1}$$

where x_i is the expression level of the i-th gene, k_i is the degradation rate, and the function $f_i(\mathbf{x}(t))$ models the effect of gene expression levels of all genes on the i-th gene at time t.

Hence, all the information of a particular GRN is encoded in the functions $f_i(\mathbf{x}(t))$ for all $i = 1$ to n. Now, in order to decode this information using a supervised learning approach, we first assume that the gene expression time-series data, $\mathbf{x}(t)$ is available to us along with the degradation rates k_i for all genes. Then we need to extract the training data from the time-series data - the input feature vectors and the target variable values.

At a particular time t, the functions f_i are solely a function of the current gene expression values. Hence, we can construct a neural network model that takes as input the gene expression vector $\mathbf{x}(t)$ of a particular time step and produces as output the value of f_i at that time step. This implies that $\mathbf{x}(t)$ will be our input feature vectors, and $f_i(t)$ will be our target variable values. In order to extract $f_i(t)$ from the time-series data, we can perform one step of Euler integration on equation 3.1:

$$dx_i = (f_i(\mathbf{x}(t)) - k_i x_i(t)) \cdot dt$$

$$\implies x_i(t + dt) - x_i(t) = (f_i(t) - k_i x_i(t)) \cdot dt$$

$$\implies f_i(t) = \frac{x_i(t+dt) - x_i(t)}{dt} + k_i x_i(t) \tag{3.2}$$

Hence, we can use the gene expression values of the i-th gene at time t and $t + dt$ to calculate the value of $f_i(t)$. Now, even though our input is time-series data since our function $f_i(\mathbf{x}(t))$ depends only on the gene expression values at time t, and no other time step, we can assume that it does not have "*memory*". Hence, we can use neural networks to model $f_i(\mathbf{x}(t))$ with the following basic idea:

> For n genes forming a GRN, we construct n neural networks. The i-th neural network is then trained with input features $\mathbf{x}(t)$ and target variables $f_i(t)$, whose ordering can be shuffled, as there is no sequential dependence. From the trained neural networks, we then deduce the network topology of the GRN.

Using the above idea, we propose three different types of machine learning models:

Model A

The first model constructs a fully-connected neural network with n input nodes, one output node, and n nodes for the hidden layers as well. The activation function from the input layer to the first hidden layer must be the **tanh** function, and the activation function from the last hidden layer to the output layer must be **linear**. The loss function is the Mean Squared Error loss. All such n neural networks are trained, after which the network topology is deduced from the learned weights and biases.

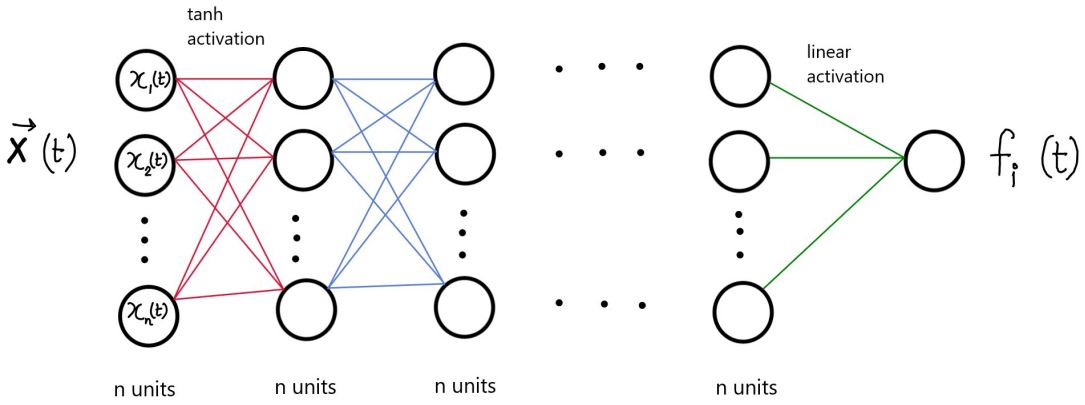


Figure 3.1: Model A - A fully-connected neural network for the i-th gene. n denotes the total number of genes.

Since this is a fully-connected neural network, it is very simple to implement using Keras's Sequential API, and its depth can be easily tuned as well; furthermore, when n is small, the model is inexpensive to train regardless of the depth.

Model B

Unlike Model A's fully-connected layers, this model consists of sparsely connected layers. Each node from the n-node input layer is connected to only certain nodes in the hidden layers, whose outputs in the last hidden layer are then multiplied to give the final one-node output layer. The activation function from the input layer to the first-hidden layer must be **tanh**. The loss function is again the Mean Squared Error loss. The reason for such an architecture is as follows:

Even though the entire neural network fits the function $f_i(\mathbf{x}(t))$, we can control the internal representation such that each "sub"-neural network j learns the function $f_{ij}(x_j(t))$ where

$$f_i(\mathbf{x}(t)) = \prod_{j=1}^{n} f_{ij}(x_j(t)) \tag{3.3}$$

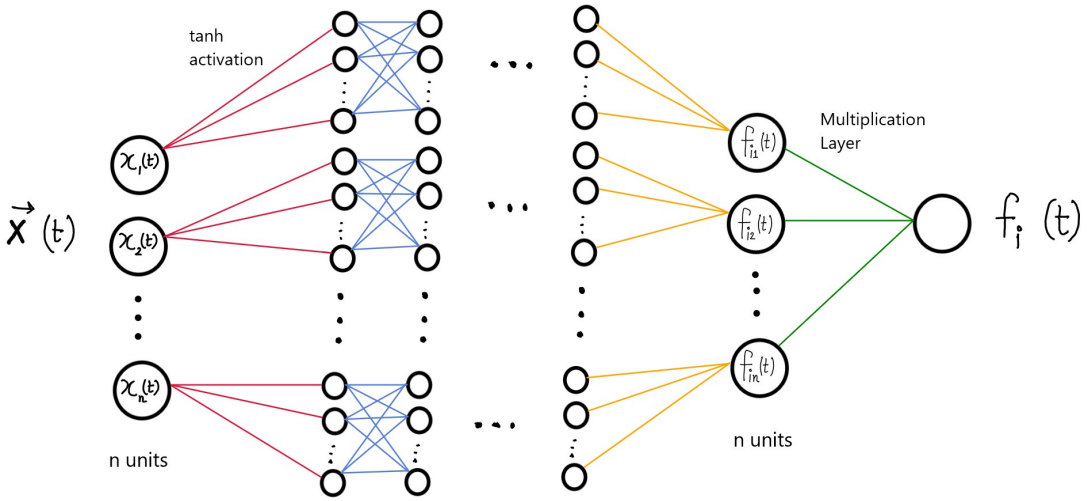


Figure 3.2: Model B - A sparse neural network for the i-th gene. n denotes the total number of genes.

Each of the functions $f_{ij}(x_j(t))$ represents the directed edge from gene j to gene i and encodes how gene j interacts with gene i.

This model's implementation uses Keras's Functional API, and its implementation is slightly more involved than Model A. However, the number of nodes and the depth of each of the sub-neural networks can still be easily tuned. Since the model has sparse connections, it is cheaper than Model A to train.

Model C

Model C is very similar to model B, except that instead of having a tanh activation function and a sub-neural network that learns the function $f_{ij}(x_j(t))$, we replace it with

the shifted-Hill function:

$$H^S(\lambda_{ij}, A_{ij}, G_{ij}, n_{ij}, x_j(t)) = G_{ij}\left(\frac{A_{ij}^{n_{ij}}}{A_{ij}^{n_{ij}} + x_j^{n_{ij}}(t)} + \lambda_{ij} \cdot \frac{x_j^{n_{ij}}(t)}{A_{ij}^{n_{ij}} + x_j^{n_{ij}}(t)}\right) \tag{3.4}$$

The parameters $\lambda_{ij}, A_{ij}, G_{ij}, n_{ij}$ are constrained to be positive. λ_{ij} denotes the fold change in expression of x_i due to x_j, and is greater than 1 for an activating edge and less than 1 for an inhibitory edge. n_{ij} is Hill's coefficient which takes integer values and characterizes the sensitivity of gene i to gene j. A_{ij} is the half-saturation constant and indicates the threshold value of $x_j(t)$ required for 50% response [18]. G_{ij} denotes the production rate.

The parameters λ_{ij}, A_{ij}, and G_{ij} are trainable since they can take continuous values, whereas the parameter n_{ij} is not trainable since it takes integer values. Hence, we can either exhaustively try out all possible combinations for it up to a maximum exponent value (whose running time will be exponential in the square of number of genes) or use a greedy approach (whose running time will be quadratic in the number of genes) to find the combination that reduces the loss maximally (see Appendix C).

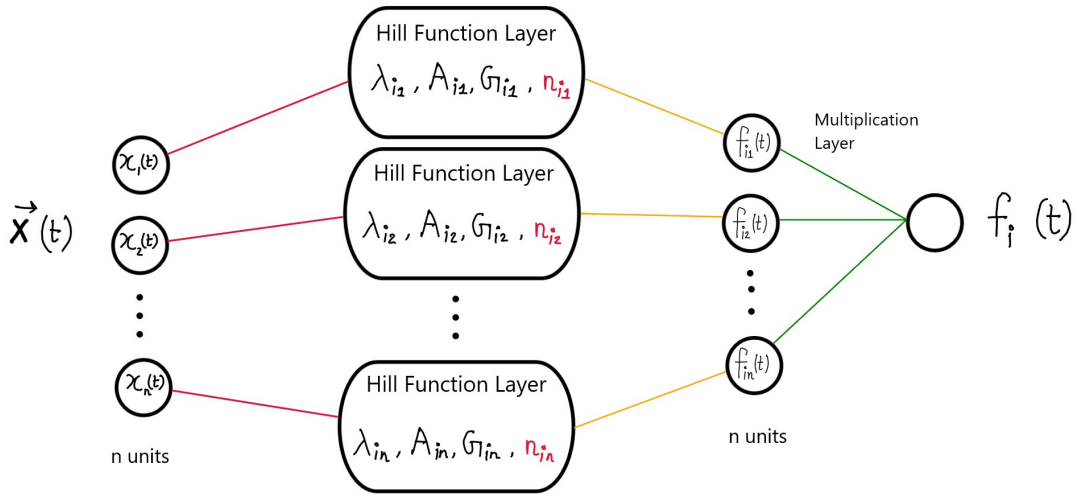


Figure 3.3: Model C - A sparse neural network for the i-th gene using the shifted-Hill function. Note that the number of genes, n, differs from Hill's coefficients n_{ij}.

This model can be implemented using Keras's Functional API. This model can suffer from the exploding gradients problem when the Hill's coefficient is odd, as the denominator in equation 3.4 can tend to zero during the optimization process.

3.2 Why is tanh activation function appropriate?

For Model A and Model B, it is important that the activation function we use is the tanh activation function. There are two main reasons for this. First, unlike the sigmoid activation function and the ReLU activation function that can only output positive values, tanh can output negative values as well. This enables a model to learn activatory responses and inhibitory responses to the gene expression levels far more effectively than using a sigmoid/ReLU activation function.

Second, since we normalize the input data to lie in the range $[0, 1]$, as we can see in Figure 1.2, the tanh activation function provides much more resolution in this range than the sigmoid activation function, and due to a much steeper gradient, it also reaches the minima quicker. This enables faster training times and better model fitting.

Furthermore, it was empirically observed that the value of $\frac{\partial f_i}{\partial x_j}$, which we will derive in the next section, is more stable (i.e., either positive or negative over the entire range of values x_j takes) when we use the tanh activation function instead of the sigmoid activation function. This is very important as the sign of $\frac{\partial f_i}{\partial x_j}$ indicates whether the j-th gene activates or inhibits the i-th gene.

3.3 Derivation of the interaction adjacency matrix

Since the i-th neural network in each model fits $f_i(t)$, the function which encodes how all the genes interact with the i-th gene, we can use its weights and biases to deduce the nature of these interactions. Doing this for all neural networks enables us to construct the interaction adjacency matrix, $\mathbf{M}$, whose ji-th entry tells us how gene j affects gene i: -1 for inhibition, +1 for activation, and 0 for no action.

3.3.1 Model A

For this model, we use the following mapping to construct $\mathbf{M}$:

$$\frac{1}{T}\sum_{t=0}^{T}\frac{\partial f_i(\mathbf{x}(t))}{\partial x_j(t)}\begin{cases} < 0 \implies M_{ji} = -1 \\ > 0 \implies M_{ji} = +1 \\ = 0 \implies M_{ji} = 0 \end{cases} \tag{3.5}$$

where T denotes the total number of time steps/the number of training examples.

To derive the expression for $\frac{\partial f_i(\mathbf{x}(t))}{\partial x_j(t)}$, consider the neural network in Figure 3.1 for the i-th gene, with n genes and two hidden layers. Let $\mathbf{\Theta}^{(1)}$, $\mathbf{b}^{(1)}$, and $\mathbf{\Theta}^{(2)}$, $\mathbf{b}^{(2)}$, and $\mathbf{\Theta}^{(3)}$,

$b^{(3)}$ denote the weights and biases from input to the first hidden layer, from the first hidden layer to the second hidden layer, and from the second hidden layer to the output layer respectively. Let $\mathbf{a}^{(1)}$ and $\mathbf{a}^{(2)}$ denote the vectors with values of the first and second hidden layers, respectively.

$\mathbf{x}(t)$, $\mathbf{b}^{(1)}$, $\mathbf{b}^{(2)}$, $\mathbf{a}^{(1)}$, $\mathbf{a}^{(2)} \in \mathbb{R}^{1\times n}$;

$\mathbf{\Theta}^{(1)} \in \mathbb{R}^{n\times n}$; $\mathbf{\Theta}^{(2)} \in \mathbb{R}^{n\times n}$; $\mathbf{\Theta}^{(3)} \in \mathbb{R}^{n\times 1}$;

and $b^{(3)}$, $f_i(t) \in \mathbb{R}$.

Since we are using tanh activation for the first two layers and linear activation from the second hidden layer to the output layer,

$$\mathbf{a}^{(1)}(t) = \tanh\left[\mathbf{x}(t)\mathbf{\Theta}^{(1)} + \mathbf{b}^{(1)}\right] \tag{3.6}$$

$$\mathbf{a}^{(2)}(t) = \tanh\left[\mathbf{a}^{(1)}(t)\mathbf{\Theta}^{(2)} + \mathbf{b}^{(2)}\right] \tag{3.7}$$

$$f_i(t) = \mathbf{a}^{(2)}(t)\mathbf{\Theta}^{(3)} + \mathbf{b}^{(3)} \tag{3.8}$$

at some time step t. Now, we can rewrite equation (3.8) as

$$f_i(t) = \sum_{l=1}^{n} a_l^{(2)}\theta_l^{(3)} + b^{(3)} \tag{3.9}$$

Taking its derivative with respect to x_j gives

$$\frac{\partial f_i(t)}{\partial x_j} = \sum_{l=1}^{n} \theta_l^{(3)} \frac{\partial a_l^{(2)}}{\partial x_j} \tag{3.10}$$

Now, from equation (3.7),

$$\frac{\partial a_l^{(2)}}{\partial x_j} = \left(1 - \tanh^2\left(\sum_{k=1}^{n} a_k^{(1)}\theta_{kl}^{(2)} + b_l^{(2)}\right)\right) \cdot \left(\sum_{k=1}^{n} \theta_{kl}^{(2)} \frac{\partial a_k^{(1)}}{\partial x_j}\right) \tag{3.11}$$

since $\dfrac{d\tanh(x)}{dx} = 1 - \tanh^2(x)$. We can rewrite this equation as

$$\frac{\partial a_l^{(2)}}{\partial x_j} = \left(1 - \left(a_l^{(2)}\right)^2\right) \cdot \sum_{k=1}^{n} \theta_{kl}^{(2)} \frac{\partial a_k^{(1)}}{\partial x_j} \tag{3.12}$$

Now, from equation (3.6), we get

$$\frac{\partial a_k^{(1)}}{\partial x_j} = \left(1 - \tanh^2\left(\sum_{i=1}^{n} x_i\theta_{ik}^{(1)} + b_k^{(1)}\right)\right) \cdot \theta_{jk}^{(1)} \tag{3.13}$$

which can be rewritten as

$$\frac{\partial a_k^{(1)}}{\partial x_j} = \left(1 - \left(a_k^{(1)}\right)^2\right) \cdot \theta_{jk}^{(1)} \tag{3.14}$$

Now, we can use equations (3.10), (3.12), and (3.14) to get

$$\frac{\partial f_i(t)}{\partial x_j} = \sum_{l=1}^{n}\sum_{k=1}^{n} \theta_l^{(3)}\theta_{kl}^{(2)}\theta_{jk}^{(1)} \left(1 - \left(a_l^{(2)}\right)^2\right)\left(1 - \left(a_k^{(1)}\right)^2\right) \tag{3.15}$$

We can derive a similar expression for a higher or lower number of hidden layers.

3.3.2 Model B

Since for this model, we can directly obtain the value of $f_{ij}(x_j(t))$ from the j-th sub-neural network of the i-th neural network (Figure 3.2), we use the following mapping to construct $\mathbf{M}$:

$$\frac{1}{T}\sum_{t=0}^{T} \frac{\partial f_{ij}(x_j(t))}{\partial x_j(t)} \begin{cases} < 0 \implies M_{ji} = -1 \\ > 0 \implies M_{ji} = +1 \\ = 0 \implies M_{ji} = 0 \end{cases} \tag{3.16}$$

where T denotes the total number of time steps/the number of training examples.

Let us consider the j-th sub-neural network in Figure 3.2 with two hidden layers, each with d units. Let $\mathbf{\Theta}^{(1,j)}$ and $\boldsymbol{b}^{(1,j)}$, $\mathbf{\Theta}^{(2,j)}$ and $\boldsymbol{b}^{(2,j)}$, and $\mathbf{\Theta}^{(3,j)}$ and $b^{(3,j)}$, be the weights and biases from $x_j(t)$ to the first hidden layer, from the first hidden layer to the second hidden layer, and from the second hidden layer to $f_{ij}(t)$. Let $\mathbf{a}^{(1)}$ and $\mathbf{a}^{(2)}$ denote vectors with values of the first and second hidden layers.

$x_j(t)$, $b^{(3,j)} \in \mathbb{R}$;

$\mathbf{\Theta}^{(1,j)}$, $\boldsymbol{b}^{(1,j)}$, $\boldsymbol{b}^{(2,j)}$, $\mathbf{a}^{(1)}$, $\mathbf{a}^{(2)} \in \mathbb{R}^{1\times d}$;

$\mathbf{\Theta}^{(2,j)} \in \mathbb{R}^{d\times d}$;

$\mathbf{\Theta}^{(3,j)} \in \mathbb{R}^{d\times 1}$.

Since we are using tanh activation for the first two layers and linear activation from the second hidden layer to the output layer,

$$\mathbf{a}^{(1)}(t) = \tanh\left(x_j(t)\mathbf{\Theta}^{(1,j)} + \boldsymbol{b}^{(1,j)}\right) \tag{3.17}$$

$$\mathbf{a}^{(2)}(t) = \tanh\left(\mathbf{a}^{(1)}(t)\mathbf{\Theta}^{(2,j)} + \boldsymbol{b}^{(2,j)}\right) \tag{3.18}$$

$$f_{ij}(t) = \mathbf{a}^{(2)}(t)\mathbf{\Theta}^{(3,j)} + b^{(3,j)} \tag{3.19}$$

at some time step t. Now, we can rewrite equation (3.19) as

$$f_{ij}(t) = \sum_{k=1}^{n} a_k^{(2)} \theta_k^{(3,j)} + b^{(3,j)} \tag{3.20}$$

which gives us

$$\frac{\partial f_{ij}(t)}{\partial x_j} = \sum_{k=1}^{n} \theta_k^{(3,j)} \frac{\partial a_k^{(2)}}{\partial x_j} \tag{3.21}$$

Using the same logic as the derivation for Model A, from equation (3.18), we get

$$\frac{\partial a_k^{(2)}}{\partial x_j} = \left(1 - \left(a_k^{(2)}\right)^2\right) \sum_{l=1}^{n} \theta_{lk}^{(2,j)} \frac{\partial a_l^{(1)}}{\partial x_j} \tag{3.22}$$

And from equation (3.17), we get

$$\frac{\partial a_l^{(1)}}{\partial x_j} = \left(1 - \left(a_l^{(1)}\right)^2\right) \theta_l^{(1,j)} \tag{3.23}$$

Combining equations (3.21), (3.22), and (3.23) gives

$$\frac{\partial f_{ij}(t)}{\partial x_j} = \sum_{k=1}^{n} \sum_{l=1}^{n} \theta_k^{(3,j)} \theta_{lk}^{(2,j)} \theta_l^{(1,j)} \left(1 - \left(a_k^{(2)}\right)^2\right) \left(1 - \left(a_l^{(1)}\right)^2\right) \tag{3.24}$$

We can derive a similar expression for a higher or lower number of hidden layers.

3.3.3 Model C

Since for this model, we can directly obtain the value of λ_{ij} from the j-th Hill Function layer of the i-th neural network (Figure 3.3), we use the following mapping to construct $\mathbf{M}$:

$$\lambda_{ij} \begin{cases} < 1 \implies M_{ji} = -1 \\ > 1 \implies M_{ji} = +1 \\ = 1 \implies M_{ji} = 0 \end{cases} \tag{3.25}$$

3.3.4 Comparing The Three Models

In order to compute M_{ji}, Model A requires the gene expression levels of *all* genes over a certain time period, meaning that its value will not be completely independent of all genes apart from gene j. Compared to this, Model B only requires the gene expression levels of the j-th gene to compute M_{ji}, hence it's completely independent of gene expression of all genes except gene j. Model C, on the other hand, does not require any gene expression levels to compute M_{ji}.

3.4 Evaluating Model Robustness

In order to test the performance of these models, we must first benchmark them on time-series gene expression data of known topologies and measure how often they can predict the correct network topology. For this task, two well-studied topologies were chosen - the *Toggle Switch*, a two-gene system with both genes inhibiting each other, and the *Toggle Triad*, a three-gene system with each gene inhibiting the other two genes.

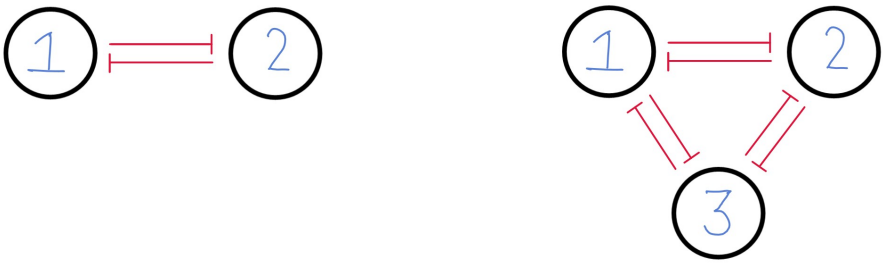


Figure 3.4: Gene circuit on the left corresponds to the Toggle Switch topology; gene circuit on the right corresponds to the Toggle Triad Topology. Note: the symbol in red denotes an inhibitory link.

First, we choose a topology, say the toggle switch, and then we generate an ensemble of circuit models corresponding to it using random circuit perturbation (RACIPE) [19]. We then categorize the randomized parameters generated by RACIPE according to the type of stability - monostability and bistability. Using these parameter sets, we generate the time-series data by numerically solving the coupled Ordinary Differential Equations modeled by RACIPE using 4^{th}-order Runge-Kutta integration. This serves as our training data.

Since both the toggle switch and the toggle triad topologies don't have self-activation or self-inhibiting interactions, all three models were constrained not to predict such interactions. This constraint isn't necessary for the models to work - in fact, without the constraint, there would be more ways the models could fit the data, making the minimization process easier. However, putting these constraints in place helps infer sparse networks and first-order interactions. This constraint was achieved by constraining $\theta^{(1)}_{jk} = 0$ when $j = i$ in equation 3.15 for model A, and by not including the gene-expression data of the i-th gene in the neural network that fits $f_i(t)$ for model B and model C and keeping $M_{ii} = 0$ in the adjacency matrix.

Due to the above constraint, since our adjacency matrix is $n \times n$ for n genes, the diagonal elements will be zero, leaving us with $n^2 - n$ elements whose values can be -1, 0, or 1, which gives us 3^{n^2-n} different combinations. Hence we can theoretically have $3^{n(n-1)}$ different adjacency matrices, each of which maps to a distinct topology T_t, $t = 1,\ 2,\ \cdots\ 3^{n^2-n}$.

To test the robustness of these models, we must evaluate how they perform when trained

on the same dataset multiple times. Each time, after the model is trained, we generate time-series data from it with the same initial values as the training data. The mean squared error between the training data and the generated data indicates how well the model could fit it and reconstruct the original dynamics. Let us define a probability matrix $\mathbf{P} \in \mathbb{R}^{D \times 3^{n(n-1)}}$ (where D denotes the total number of datasets), whose ij-th entry denotes the probability (as predicted by our model) of the i-th dataset corresponding to topology T_j.

First, we initialize each $\mathbf{P}$ entry to zero. Now, let us say we are training our model on the r-th dataset for the s-th time, and after training, we find that the model predicts that the data corresponds to topology T_t and it fits the data with a mean square error of $m_s^{(r)}$. Then, we increment the rt-th entry as follows:

$$P_{rt} = P_{rt} + \frac{1}{m_s^{(r)}}$$

We increment in this manner because we want to give more weight to the topology that was predicted with less reconstruction mean squared error than to a topology predicted with more reconstruction mean squared error.

After we have completed training our model on the r-th dataset a certain number of times, we normalize the r-th row of $\mathbf{P}$:

$$\mathbf{P}_r = \left(\sum_{j=1}^{3^{n(n-1)}} P_{rj} \right)^{-1} \mathbf{P}_r$$

Repeating the above process for all datasets gives the matrix $\mathbf{P}$ whose r-th row gives us the empirical probability of the r-th dataset corresponding to various topologies, weighted by the performance when each topology was predicted. Figure 3.5 summarizes the entire process.

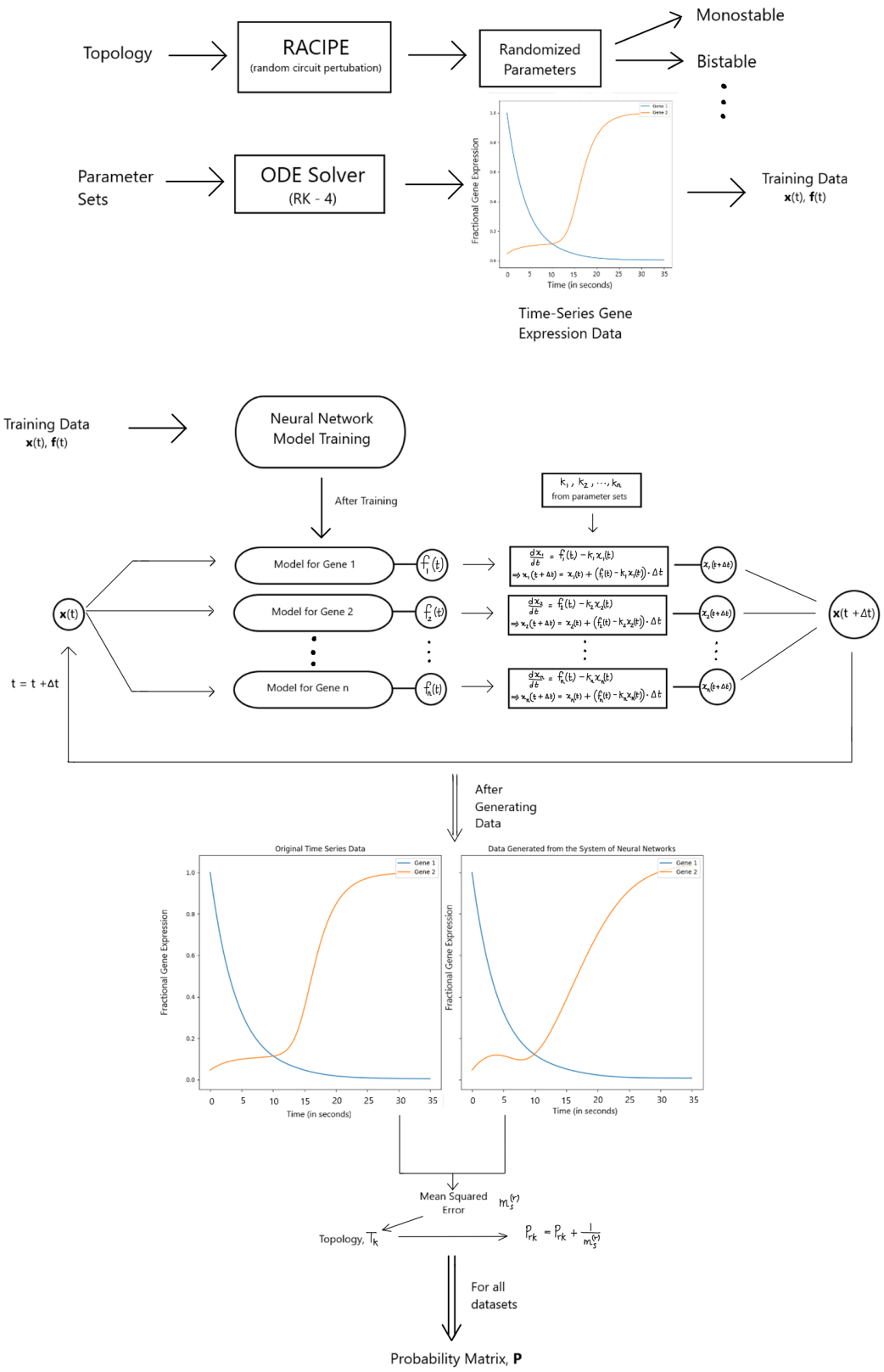


Figure 3.5: Flowchart of the entire pipeline

Chapter 4

Results

The following settings were used when training the models on a dataset and generating time-series data from them:

Parameter	Value
Batch Size	16
Epochs	50
Learning Rate	0.01
Δt	0.01 s

The model was trained on the same dataset 15 times, each time starting from a new random seed for the weights and biases. More runs couldn't be performed on the same dataset because of limited computational power. Training on multiple datasets was done in parallel, with each thread carrying out the process in Figure 3.5 on individual datasets. (*Specifications of the computing cluster* - **CPU** - Intel(R) Xeon(R) Platinum 8173M CPU @ 2.00GHz, 60 CPU cores, 1 thread per CPU core; **RAM** - 64 GB shared memory, L1 cache - 1.9 MiB, L2 cache - 60 MiB, L3 cache - 2.3 GiB; **Discrete GPU** - None).

In general, for n genes, we saw in the last chapter that after constraining self-activity, we could have $3^{n(n-1)}$ different topologies. However, if we were to consider only the dense topologies from these, then we would have only $2^{n(n-1)}$ of them, as each non-diagonal element in the adjacency matrix can only be +1 or -1.

4.1 Toggle Switch

For the toggle switch topology, a total of 778 datasets corresponding to monostable behavior and 222 datasets corresponding to bistable behavior were obtained. For the monostable case, after generating the time-series data, the datasets were sorted in in-

creasing order of A/B value where

$$A = \frac{A_{ss}}{G_A/k_A} \quad \text{and} \quad B = \frac{B_{ss}}{G_B/k_B}$$

denote the steady state values of gene 1 and 2 (A_{ss} and B_{ss}) normalized by the maximum theoretically possible value they can attain (G_A/k_A and G_B/k_B − from RACIPE parameters.) For the bistable case, the time-series data was generated such that they started from a state A-high, B-low and finished at the state A-low, B-high. After this, the datasets were sorted in increasing order of A_{num}/B_{num} where A_{num} denotes the percentage of times the system went to a steady state A-high, B-low, and B_{num} denotes the percentage of times the system went to a steady state A-low, B-high (A_{num} and B_{num} are obtained from RACIPE as well.)

For two genes, we can have $2^{2\cdot(2-1)} = 4$ dense topologies, barring self-activity:

$$\begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix} \quad \text{Toggle Switch } - \text{ Double Inhibition}$$

$$\begin{bmatrix} 0 & -1 \\ +1 & 0 \end{bmatrix} \text{ and } \begin{bmatrix} 0 & +1 \\ -1 & 0 \end{bmatrix} \quad \text{Single Activation and Inhibition}$$

$$\begin{bmatrix} 0 & +1 \\ +1 & 0 \end{bmatrix} \quad \text{Double Activation}$$

In order to compare the performance of various models, we define a quantity, the average probability of predicting topology k as the mean of the k-th column of the probability matrix $\mathbf{P}$. This can be thought of as a metric that quantifies the probability a certain model will predict a particular topology when it is provided with gene expression data from a toggle switch (assuming our datasets explore a large portion of the toggle switch gene expression space.) Ideally, a perfect model will have 100% average probability of predicting toggle switch topology and 0% for the other topologies.

4.1.1 Monostable Case

Model A

On the monostable datasets, model A was tested with one, two, three, and four hidden layers in the neural networks for each gene. We can evaluate how the performance changes with an increasing number of layers by looking at how the average probability and the mean squared error changes with the number of layers:

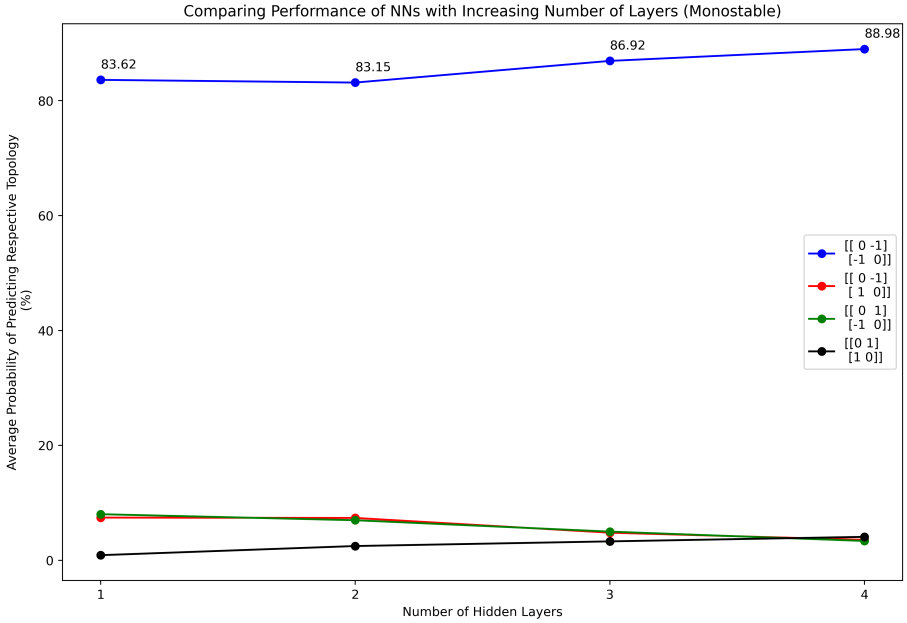


Figure 4.1: This plot shows that, on average, the model's performance on monostable datasets increases as the number of hidden layers increases since the average probability of predicting the toggle switch topology increases from 83.62% with one hidden layer to 88.98% with four hidden layers.

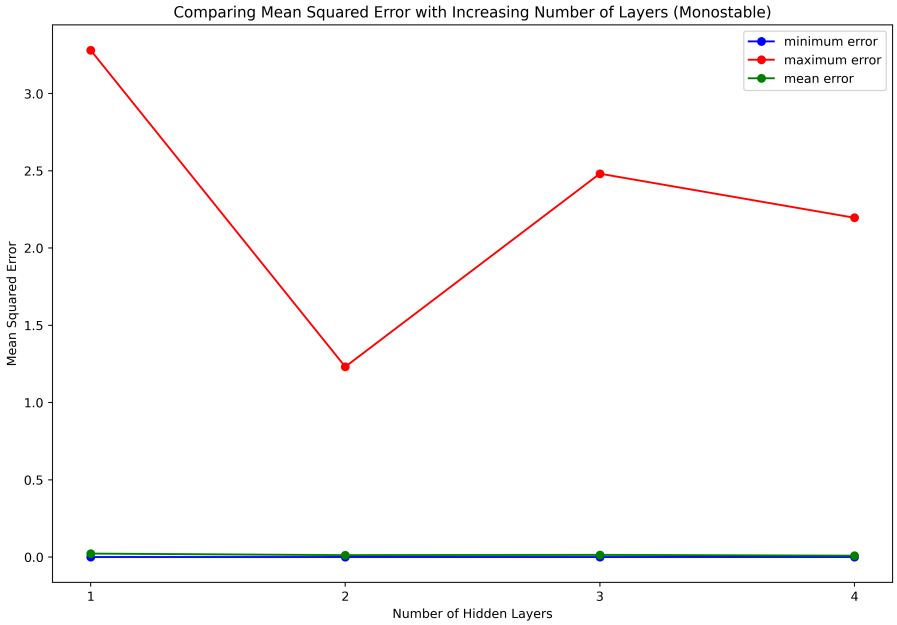
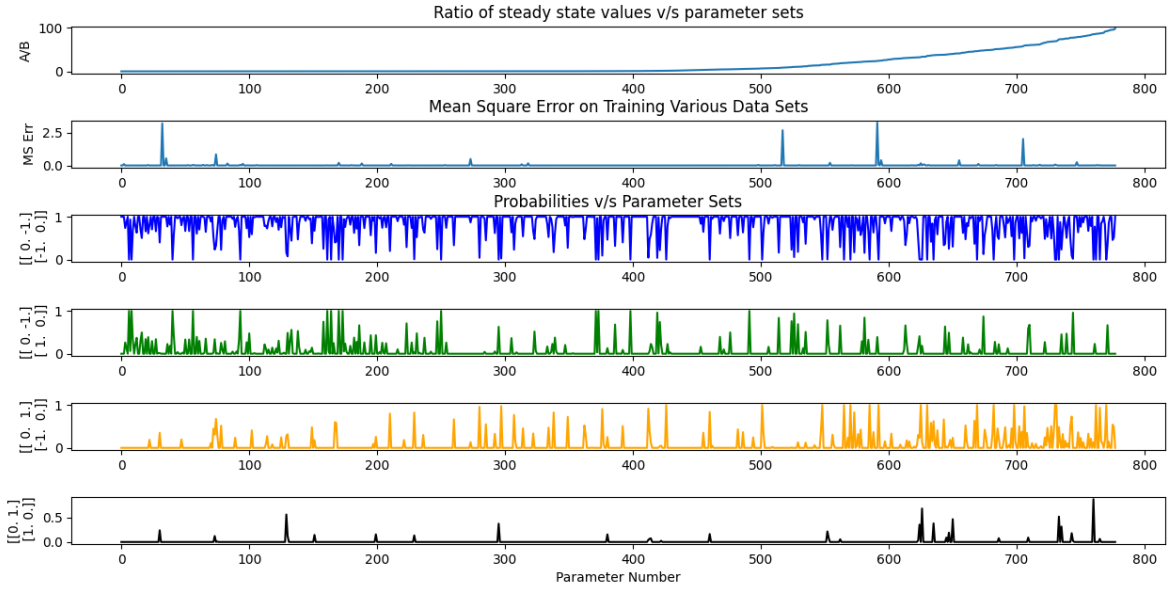
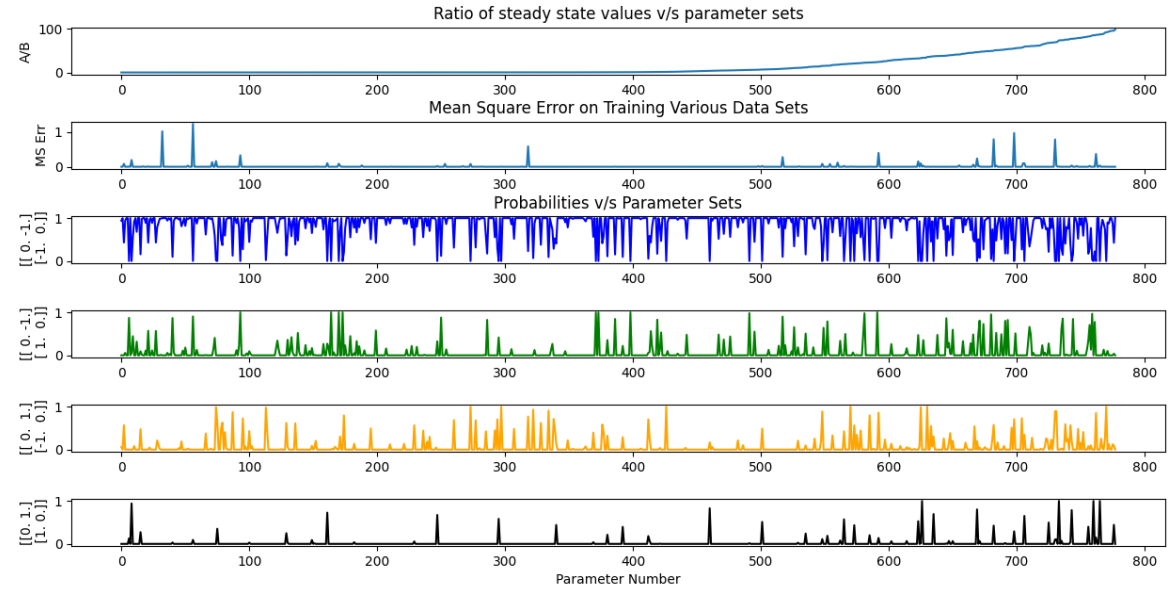


Figure 4.2: This plot shows that, on average, the model can fit the monostable training data well with a higher number of layers, as the minimum MSE, the maximum MSE, and the average MSE all reduce as the number of hidden layers increases.

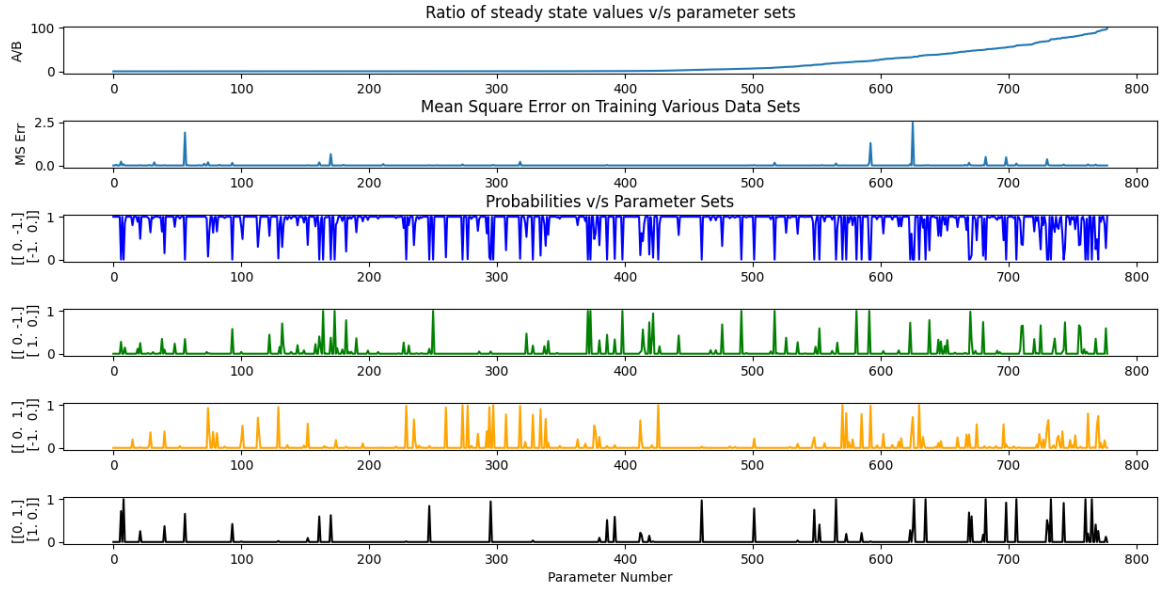
As we can see, the probability of correctly predicting the toggle switch topology increases as we increase the number of hidden layers. The maximum average probability was 88.98% with four hidden layers. The minimum average (mean squared) error over all datasets was 0.0087, obtained with four hidden layers.



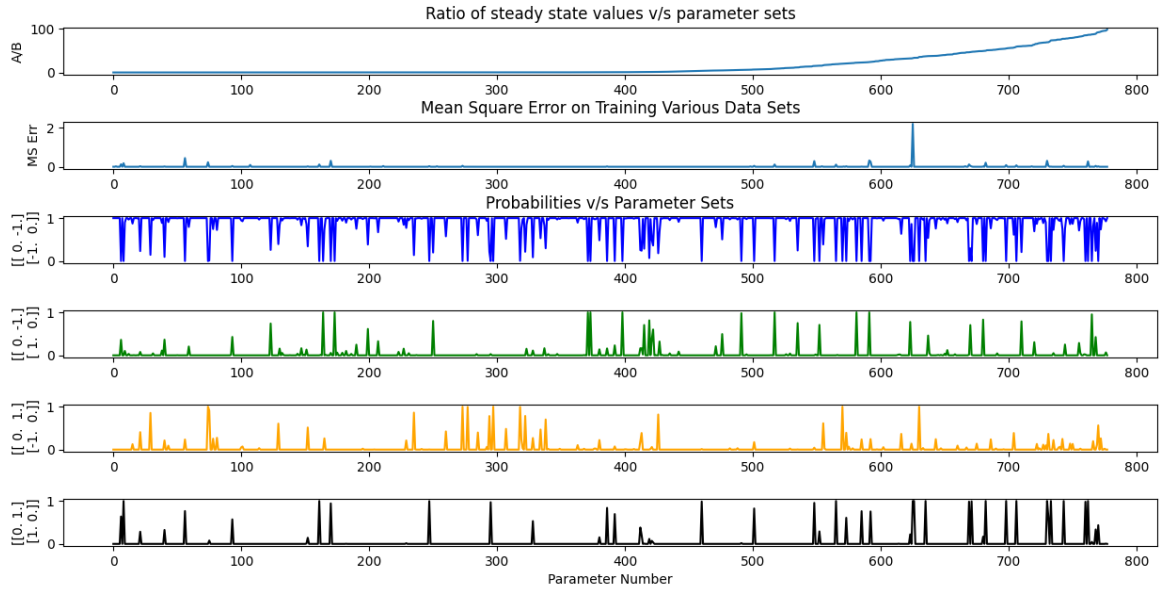
(a) Performance on 1 Hidden Layer



(b) Performance on 2 Hidden Layers



(c) Performance on 3 Hidden Layers



(d) Performance on 4 Hidden Layers

Figure 4.4: These graphs give an overview of how Model A performs on monostable datasets. The datasets have been arranged in an increasing ratio of steady-state values A/B, as described earlier. The dark blue line graph indicates how the probability of predicting toggle switch topology changes with this ratio. An interesting feature that we can see from these graphs is that even when topologies other than the toggle switch are predicted, the Mean Squared Error is low, implying that these other topologies can produce similar time-series data as the toggle switch. On the other hand, there are instances where the Mean Squared Error is abnormally high, yet the toggle switch topology is predicted. However, one must note that these trends decrease drastically as we increase the number of layers, a further indication that Model A with a large number of layers can give more confident predictions.

Model B

On the monostable datasets, model B was tested with two hidden layers in each of the sub-neural networks in each neural network. With two hidden layers, the model had an average probability of 71.58% on correctly predicting the toggle switch topology. The minimum (mean squared) error was 2.74×10^{-6}, the maximum error was 0.104, and the average error over all datasets was 0.02. Hence, the model, on average, performs worse than model A.

Model C

Model C had an average probability of 98.94% on correctly predicting the toggle switch topology. The minimum (mean squared) error was 6.912×10^{-5}, the maximum error was 0.19, and the average error over all datasets was 0.044. Hence, the model, on average, fits the datasets slightly worse than model A, however, it has much better performance than both model A and model B in correctly predicting the toggle switch topology. But this model suffered from the gradients blowing up on 69 datasets, or approximately 8.87% of the datasets hence could not fit the data on them.

4.1.2 Bistable Case

Model A

On the bistable datasets, Model A was tested with one, two, three, and four hidden layers in the neural networks of each gene. The following two plots illustrate a similar trend to model A's performance on monostable datasets:

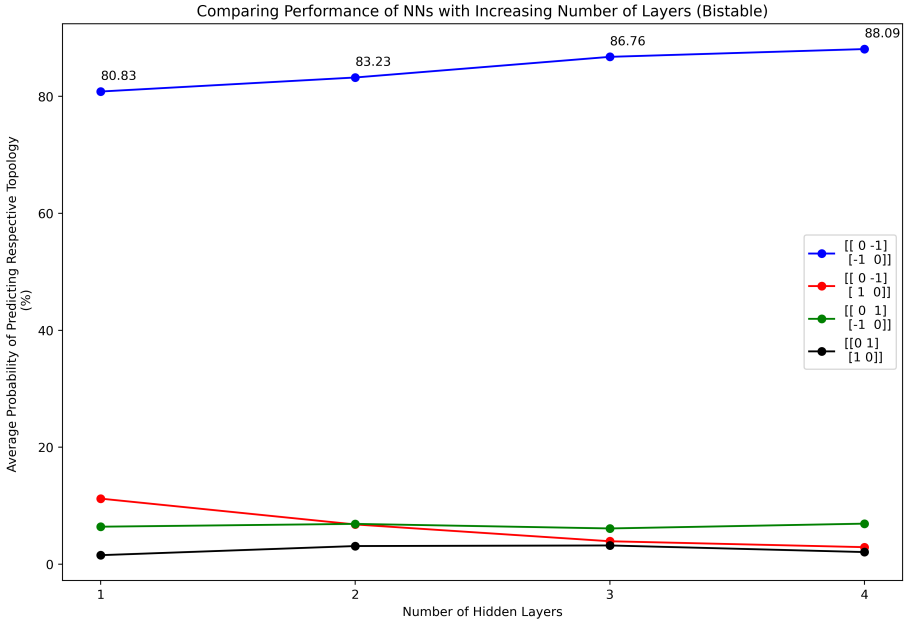


Figure 4.5: This plot shows that, on average, the model's performance on bistable datasets increases as the number of hidden layers increases since the average probability of predicting the toggle switch topology increases from 80.83% with one hidden layer to 88.09% with four hidden layers.

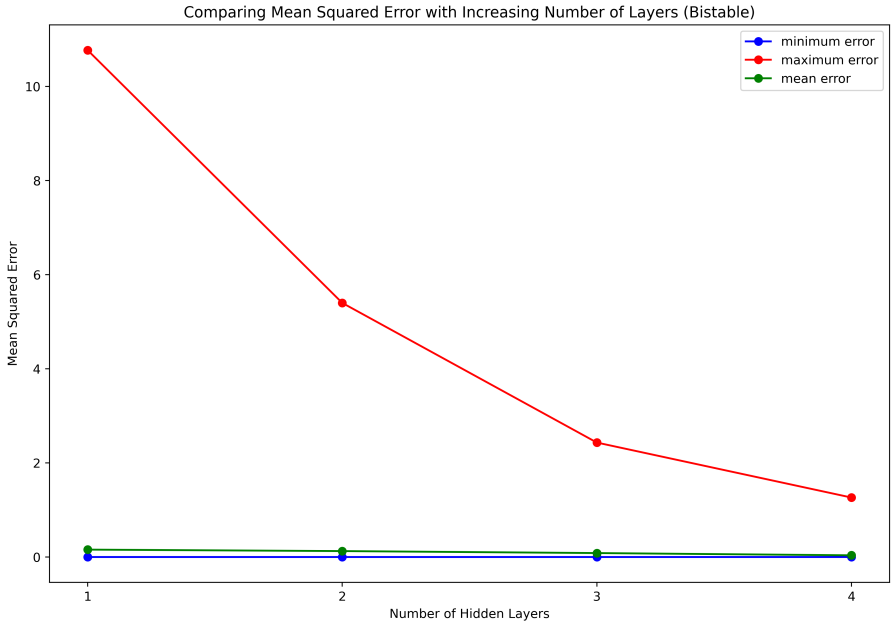


Figure 4.6: This plot shows that the model can fit the bistable training data well with a higher number of layers, as the minimum MSE, the maximum MSE, and the average MSE all reduce as the number of hidden layers increases.

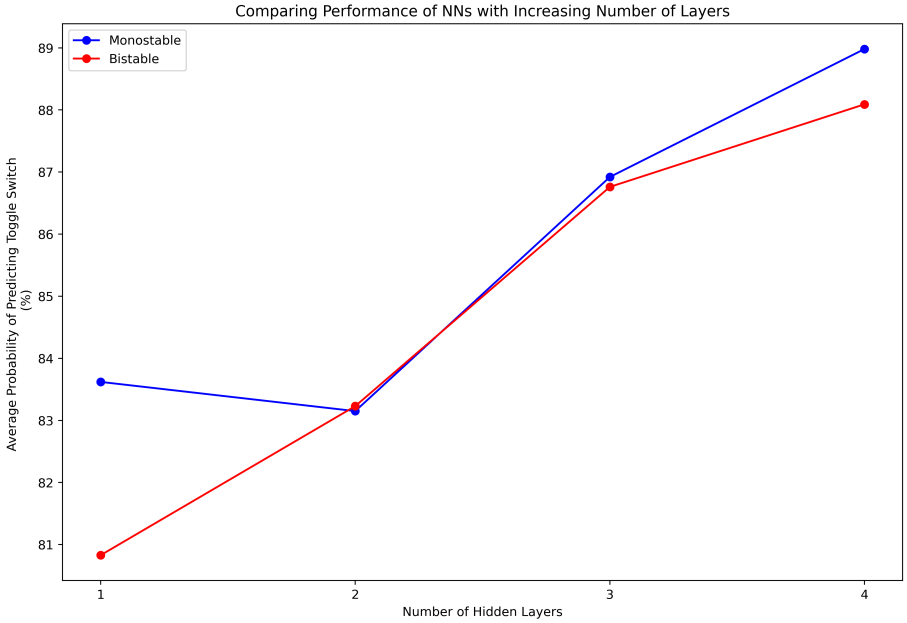
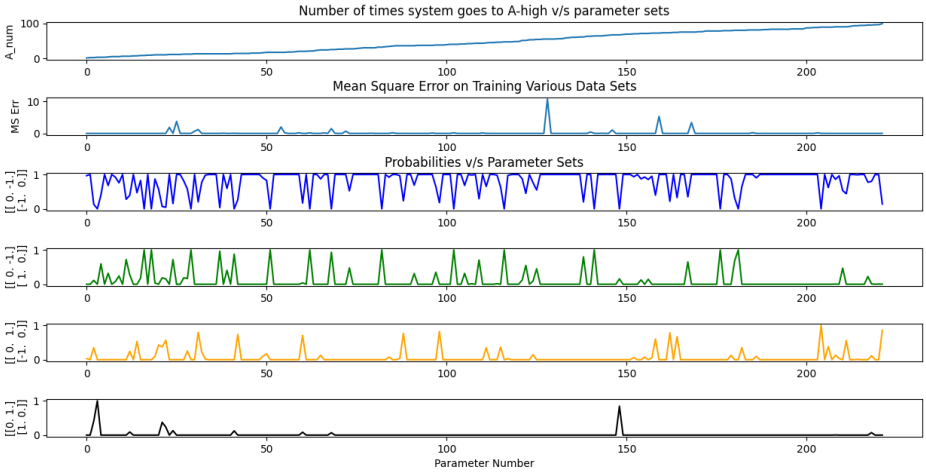
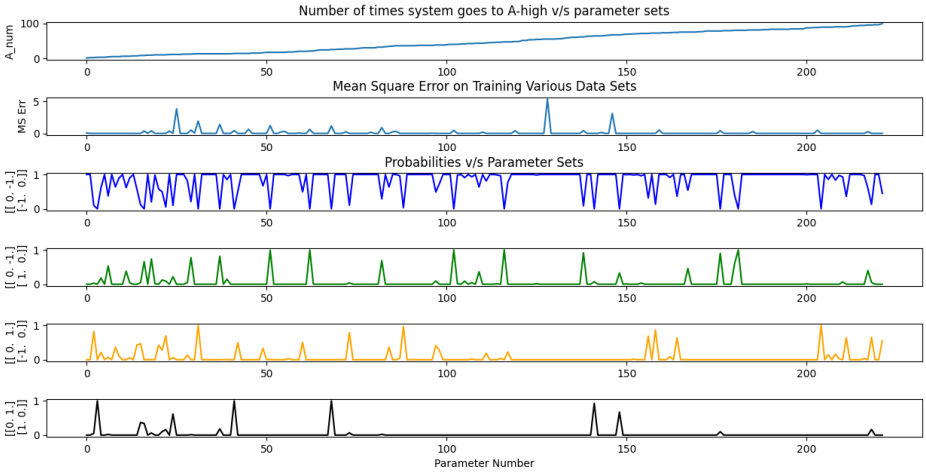


Figure 4.7: This plot shows that the model has similar performance in predicting the toggle switch topology for both monostable and bistable cases, regardless of the number of hidden layers.

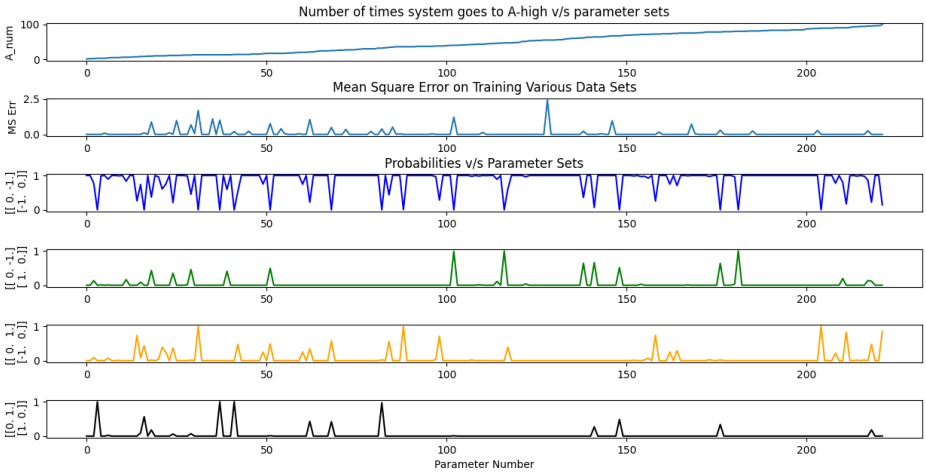
Again, the probability of correctly predicting the toggle switch topology increases as we increase the number of hidden layers. The maximum average probability was 88.09% and the minimum average (mean squared) error over all datasets was 0.0365, both obtained with four hidden layers.



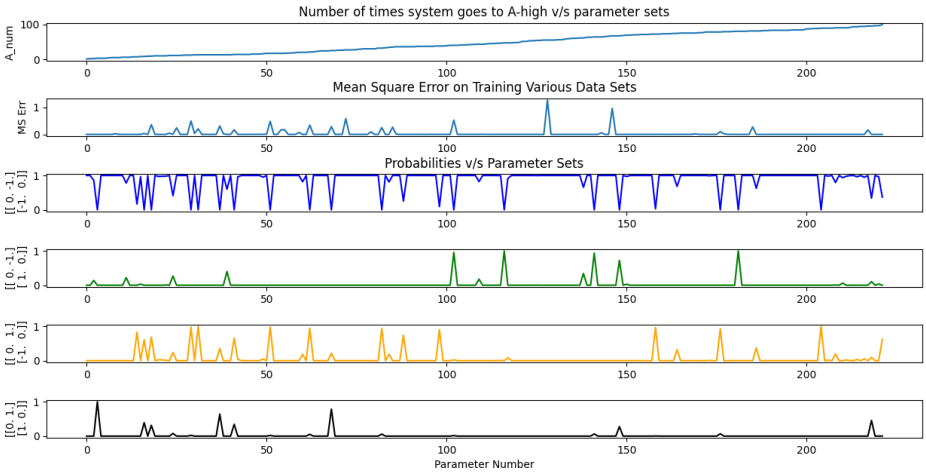
(a) Performance on 1 Hidden Layer



(b) Performance on 2 Hidden Layers



(c) Performance on 3 Hidden Layers



(d) Performance on 4 Hidden Layers

Figure 4.9: These graphs give an overview of how Model A performs on bistable datasets. The datasets have been arranged as described earlier. The dark blue line graph indicates how the probability of predicting toggle switch topology changes with this ratio.

Model B

On the bistable datasets, model B was tested with two hidden layers in each of the sub-neural networks in each neural network. With two hidden layers, the model had an average probability of 67.54% on correctly predicting the toggle switch topology, slightly lower than the monostable case. The minimum (mean squared) error was 2.57×10^{-5}, the maximum error was 0.094, and the average error over all datasets was 0.02. Hence, on average, the model performs worse than model A on bistable cases too.

Model C

Model C had an average probability of 98.81% on correctly predicting the toggle switch topology. The minimum (mean squared) error was 0.0013, the maximum error was 0.195, and the average error over all datasets was 0.058. Hence, the model, on average, fits the datasets slightly worse than model A, however, it has much better performance than both model A and model B in correctly predicting the toggle switch topology. But this model suffered from the gradients blowing up on 16 datasets, or approximately 7.2% of the datasets hence could not fit the data on them.

4.2 Toggle Triad

For the toggle triad topology, a total of 1065 datasets corresponding to monostable behavior were obtained. Even though we have time-series data for three genes, the monostable datasets were sorted in the same manner as the toggle switch. This is because if $A/B << 1$, then the stable state must be A-low, B-high, C-low, if $A/B \approx 1$, then the stable state must be A-low, B-low, C-high, and if $A/B >> 1$, then the stable state must be A-high, B-low, C-low (where A, B, and C denote the three genes). For the bistable case, a total of 787 datasets were obtained. Each of the datasets corresponded to the final steady explored more frequently by the topology with the given set of parameters.

For three genes, we can have $2^{3\cdot(3-1)} = 64$ dense topologies, barring self-activity. In order to simplify our analysis, for the following models, we only consider the topologies that have been predicted for at least 10% of datasets. Within this subset, the five most common types of topologies were:

$$\text{Toggle Triad} - \quad \begin{bmatrix} 0 & -1 & -1 \\ -1 & 0 & -1 \\ -1 & -1 & 0 \end{bmatrix}$$

$$\text{Single Activation Link, for example} - \begin{bmatrix} 0 & +1 & -1 \\ -1 & 0 & -1 \\ -1 & -1 & 0 \end{bmatrix}$$

$$\text{Double Activation Links from the same gene, for example} - \begin{bmatrix} 0 & +1 & +1 \\ -1 & 0 & -1 \\ -1 & -1 & 0 \end{bmatrix}$$

$$\text{Double Activation Links to the same gene, for example} - \begin{bmatrix} 0 & -1 & +1 \\ -1 & 0 & +1 \\ -1 & -1 & 0 \end{bmatrix}$$

$$\text{Double Activation Links between two genes, for example} - \begin{bmatrix} 0 & -1 & -1 \\ -1 & 0 & +1 \\ -1 & +1 & 0 \end{bmatrix}$$

We use the same metrics as we did in the toggle switch to compare the performance of various models - the average probability of predicting a particular topology, and the minimum, maximum, and average (mean squared) error the model gives over all datasets.

4.2.1 Monostable Case

Model A

On the monostable datasets, model A was tested with one, two, three, and four hidden layers in the neural networks of each gene. The following four plots illustrate how the performance changes with an increasing number of layers:

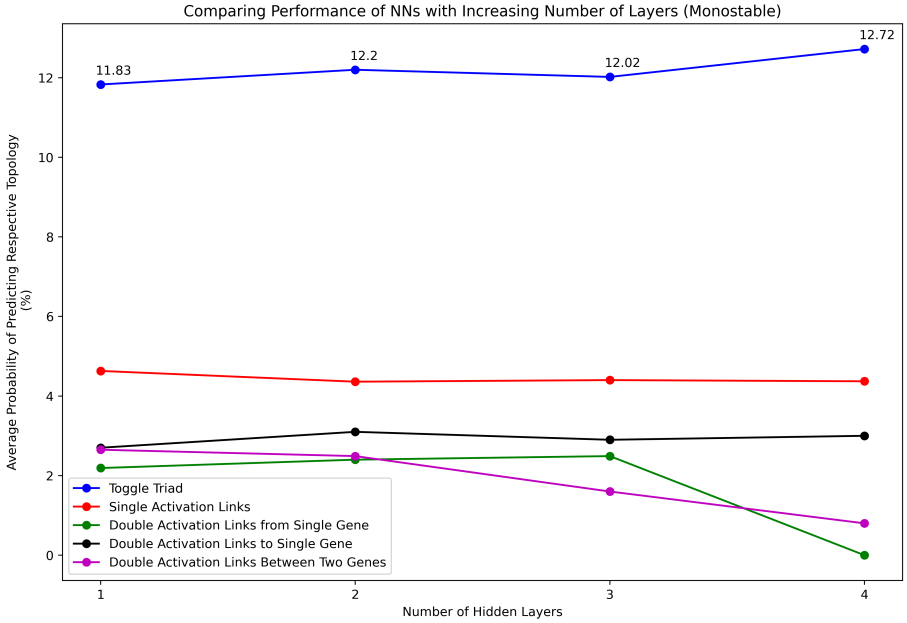


Figure 4.10: This plot shows that, on average, the model's performance on monostable datasets increases slightly as the number of hidden layers increases since the average probability of predicting the toggle switch topology increases from 11.83% with one hidden layer to 12.72% with four hidden layers.

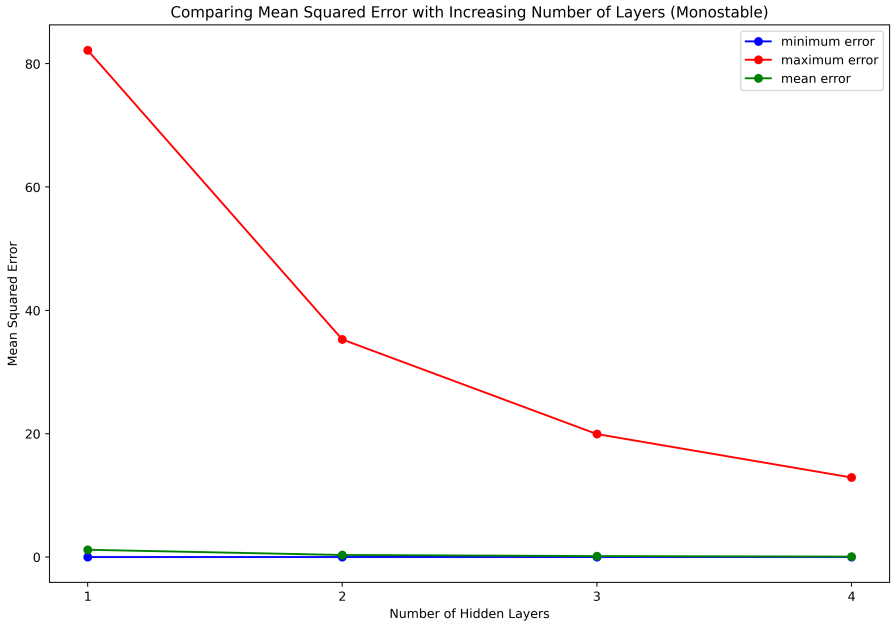


Figure 4.11: This plot shows that the model can fit the monostable training data well with a higher number of layers, as the minimum MSE, the maximum MSE, and the mean MSE all reduce as the number of hidden layers increases.

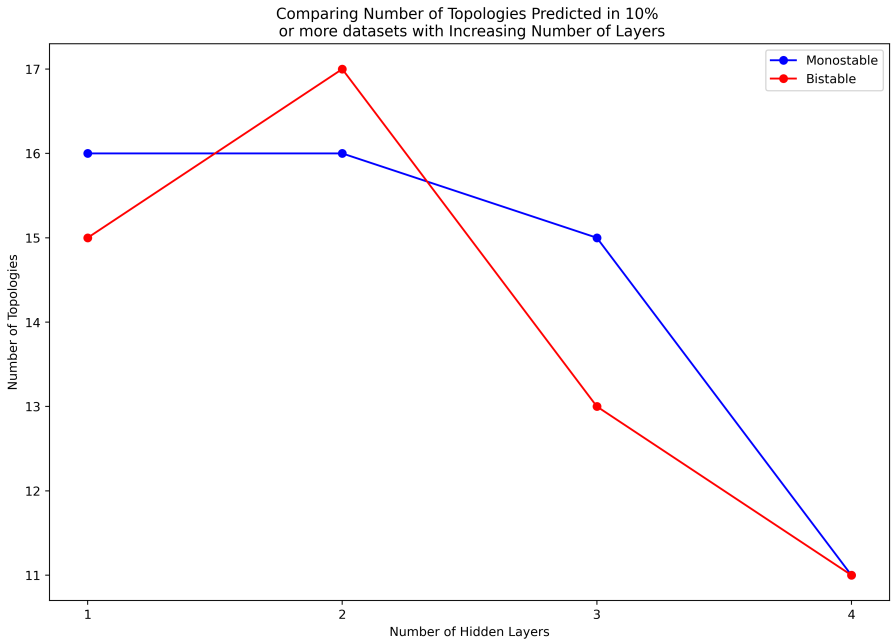


Figure 4.12: This plot shows that the number of topologies predicted in 10% or more of the datasets reduces as we increase the number of hidden layers, reducing the uncertainty.

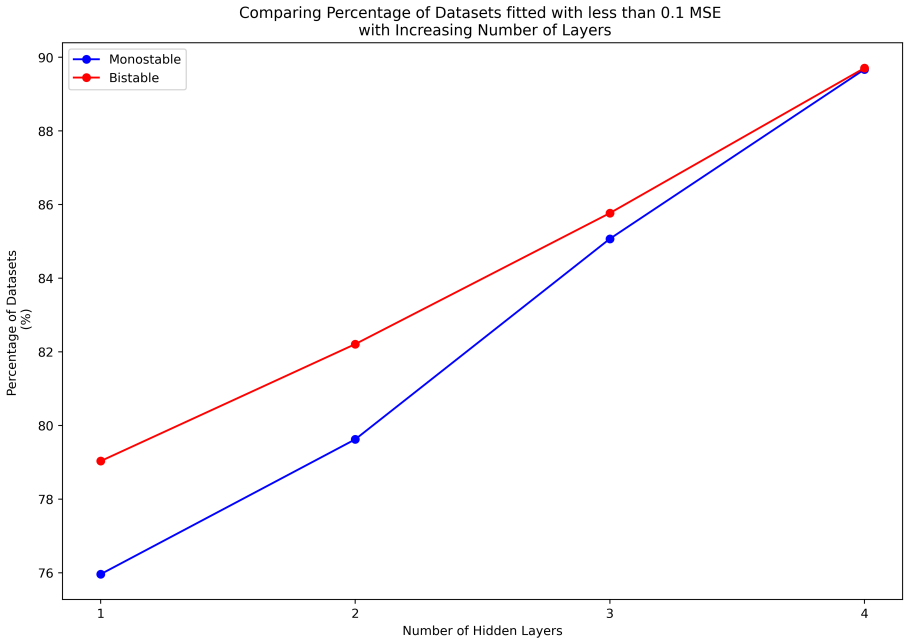


Figure 4.13: This plot shows how the number of datasets fitted with an error of ≤ 0.1 increases with the number of layers. This shows that the model's ability to fit the data increases with the number of layers.

As we can see, the probability of correctly predicting the toggle triad topology increases

as we increase the number of hidden layers. Furthermore, the probability of predicting other topologies and the number of topologies predicted also reduces. The maximum average probability was 12.72%, the minimum average (mean squared) error was 0.069, and the number of topologies predicted was minimum (11), with four hidden layers. Hence, with four hidden layers, this model can predict the correct topology within a single-link impurity with an average probability of 38.94%.

Model B

On the monostable datasets, model B was tested with two hidden layers in each of the sub-neural networks in each gene's neural network. The model had an average probability of only 1.94% on correctly predicting the toggle triad topology, much lower than model A. In fact, all 64 topologies were predicted on 10% or more of the datasets, meaning the model was unable to conclusively determine the correct topology most of the time. However, only 18 topologies were predicted on 20% or more of the datasets. Toggle triad had the highest probability (1.94%), followed by other topologies having one, two, or three activation links with an average probability of 1.6%. Overall, this model did not perform well in correctly predicting the toggle triad topology. The minimum (mean squared) error was 3.04×10^{-5}, the maximum error was 0.247, and the average error over all datasets was 0.03. The model could fit 1039 datasets with an error of less than 0.1, compared to model A's best of 955 datasets. Hence, on average, this model can fit the data slightly better than model A.

Model C

Model C had an average probability of 46.86% on correctly predicting the toggle triad topology, far more than model A or B. Furthermore, only 7 topologies were predicted on 10% or more of the datasets. Apart from the toggle triad topology, the other six topologies were of single activation, with an average probability of 7.37% each. Hence, this model could always predict the correct topology within a single-link impurity. The minimum (mean squared) error was 0.004, the maximum error was 0.338, and the average error over all datasets was 0.122. Compared to models A and B, this model could fit only 347 datasets with an error of less than 0.1. Hence, this model can fit the data quite well, although not as well as models A or B. The model did not suffer from the gradients blowing up on any dataset as the auto-adjusting Hill coefficient scheme was used during training (Appendix C).

4.2.2 Bistable Case

Model A

On the bistable datasets, model A was tested with one, two, three, and four hidden layers in the neural networks of each gene. The following plots illustrate how the performance changes with an increasing number of layers:

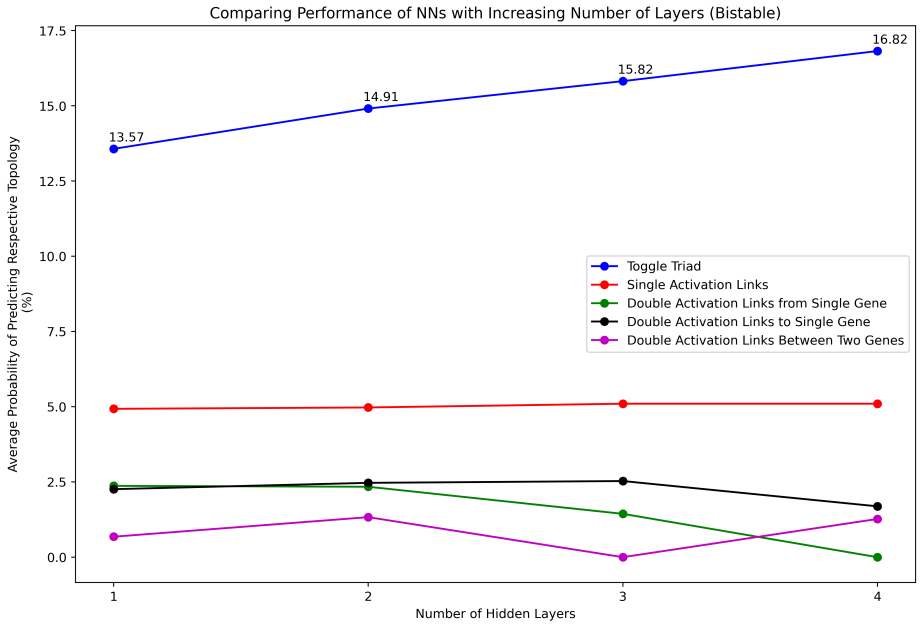


Figure 4.14: This plot shows that, on average, the model's performance on bistable datasets increases as the number of hidden layers increases since the average probability of predicting the toggle switch topology increases from 13.57% with one hidden layer to 16.82% with four hidden layers.

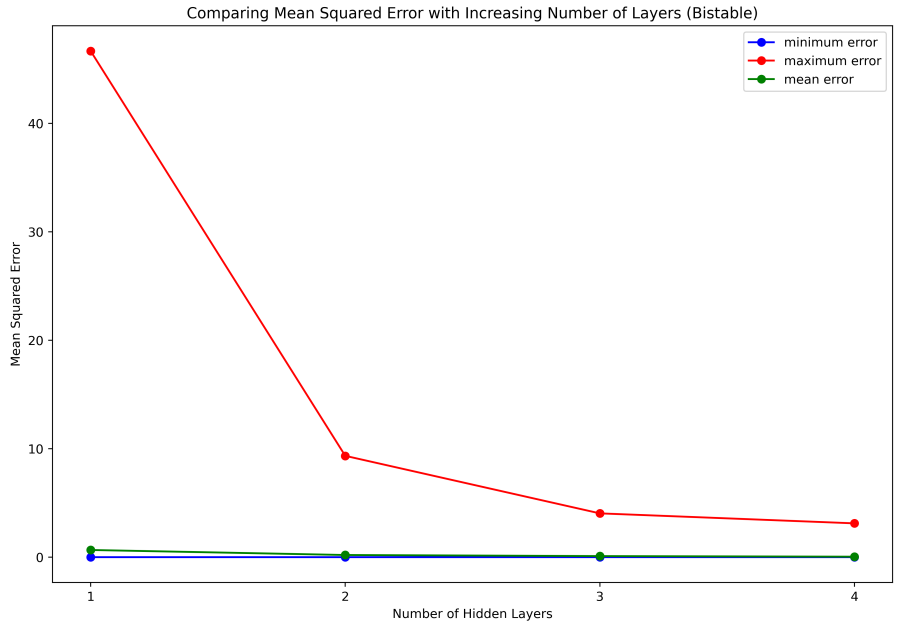


Figure 4.15: This plot shows that the model can fit the bistable training data well with a higher number of layers, as the minimum MSE, the maximum MSE, and the average MSE all reduce as the number of hidden layers increases.

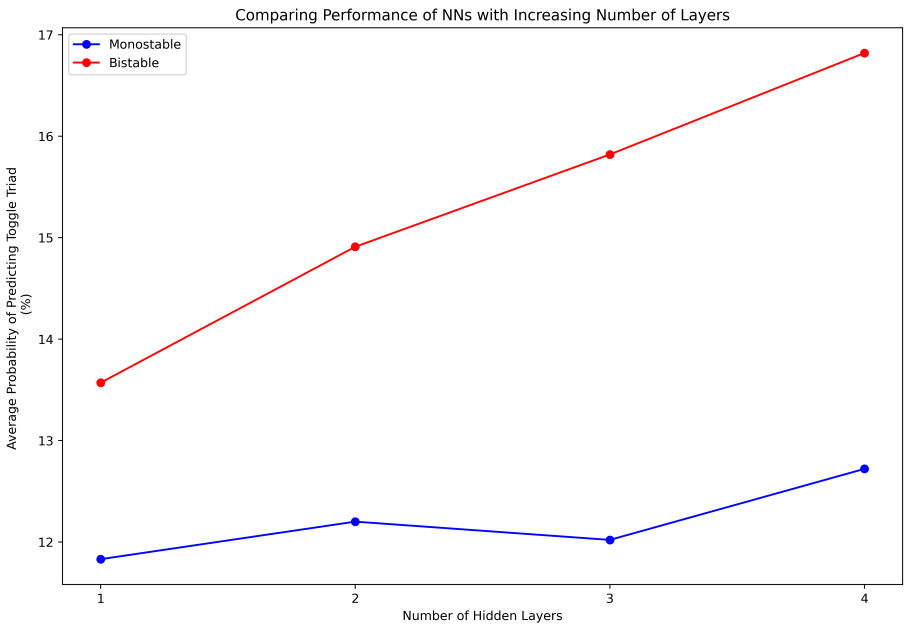


Figure 4.16: This plot shows that the model's ability to predict the correct topology increases (with the number of hidden layers) much faster for the bistable case than the monostable case.

As we can see from these plots, the probability of correctly predicting the toggle triad

topology increases with the number of hidden layers (much faster than it did for the monostable case). The probability of predicting other topologies decreases too, on average. The maximum mean squared error was highest with one hidden layer (but smaller than its value in the monostable case) and decreased rapidly as we increased the number of layers (dropping to levels even lower than what we obtained in the monostable case). The maximum average probability was 16.82%, the minimum average (mean squared) error was 0.045, and the number of topologies predicted was minimum (11), with four hidden layers. Hence, with four hidden layers, this model can predict the correct topology within a single-link impurity with an average probability of 47.42%, higher than the monostable case.
Figures 4.12 and 4.13 also show that as we increase the number of hidden layers, the number of topologies predicted in at least 10% of the datasets and the percentage of datasets fitted with less than 0.1 MSE converge for both the monostable and bistable datasets.

Model B

On the bistable datasets, model B was tested with two hidden layers in each of the sub-neural networks in each gene's neural network. The model had an average probability of only 1.43% on correctly predicting the toggle triad topology, much lower than model A. In fact, all 64 topologies were predicted on 10% or more of the datasets, meaning the model was unable to conclusively determine the correct topology most of the time. However, only 20 topologies were predicted on 20% or more of the datasets. Toggle triad had a probability of (1.46%), followed by other topologies having one, two, or three activation links with an average probability of 1.56%. Overall, this model did not perform well in correctly predicting the toggle triad topology. The minimum (mean squared) error was 1.13×10^{-4}, the maximum error was 0.144, and the average error over all datasets was 0.031. The model could fit 777 datasets with an error of less than 0.1, compared to model A's best of 706 datasets. Hence, on average, this model can fit the data slightly better than model A.

Model C

Model C had an average probability of 48.15% on correctly predicting the toggle triad topology, far more than model A or B (slightly higher than monostable datasets as well). Furthermore, only 7 topologies were predicted on 10% more of the datasets. Apart from the toggle triad topology, the other six topologies were of single activation, with an average probability of 7.35% each. Hence, this model could always predict the correct topology within a single-link impurity. The minimum (mean squared) error was 0.002,

the maximum error was 0.45, and the average error over all datasets was 0.121. Compared to models A and B, this model could fit only 257 datasets with an error of less than 0.1. Hence, this model can fit the data quite well, although not as well as models A or B. The model did not suffer from the gradients blowing up on any dataset, as the auto-adjusting Hill coefficient scheme was used during training (Appendix C).

Chapter 5

Future Outlook

5.1 Improvements to the current architecture

As we saw in the previous chapter, the models we have developed show great potential in robustly identifying the correct gene regulatory networks from the time-series gene expression data provided to it.

We saw that for both the toggle switch and the toggle triad, Model A's ability to predict the correct topology increases, and the maximum and average MSE decreases as we increase the number of hidden layers for both the monostable and bistable datasets. Hence, it only makes sense to keep increasing the number of hidden layers and check at what number these improvements start plateauing. We could also empirically determine how the number of hidden layers required for optimal performance depends on the number of genes.

Model B performed well on toggle switch datasets but poorly on toggle triad datasets. One of the reasons could be that the sub-neural networks were not sufficiently high-capacity models for the individual f_{ij} functions. This can be fixed by increasing both the number of units in each layer and the number of layers in each sub-neural network. An appropriate combination of activation functions between the layers can also help mitigate poor performance.

Model C performs exceptionally well for both toggle switch and toggle triad cases. The only problem it suffers from is the occasional blowing up of gradients while training. An ad-hoc fix to this could be restricting the Hill coefficient (n) to take only even values; however, we may lose the ability to infer the exact gene interactions by doing this.

Apart from these changes, additional analysis can also be performed on the trained networks to quantify their performance and robustness further. First, we would like to check the specific cases when the models failed to predict the correct topology. We could look

at the adjacency matrices they output and devise a distance metric that could tell us how far away they are from the true adjacency matrix (which would not only contain the type of interaction, but also the strength of the interaction). The same distance metric could be used to analyze the datasets that correctly predicted the topology and see how far away their adjacency matrices are from the ground truth.
For the cases where the models failed to predict the correct topology but gave very low MSE, we would want to find out if there were any underlying characteristics/trends in the training data that led to this. The possible sources of these trends could be the final steady state gene-expression values, peak gene-expression values, or even the initial condition gene-expression values.

The training data used was obtained by numerically solving the system of coupled ODEs. Thus the data was high-resolution without any noise affecting it. One more metric we would want to quantify our models on would be how coarse-grained can the training data become before accurate network inference cannot be performed. We can also make the data noisy and quantify the robustness of the models by how large this noise can be before the models fail to converge to the correct topologies.
Another characteristic of our training data was that the gene expression levels always reached a steady state in them. Thus, we could also explore how the models perform when the gene expression levels don't reach a steady state.

5.2 Other Possible ML/DL architectures

All three models proposed in this thesis are based on the assumption that biological gene networks, and hence their mathematical counterparts, do not possess "memory", i.e., the effect all n genes have on the i-th gene depends only on their current expression values, and not on the values they took previously. This is a fairly reasonable assumption, and innumerable models of GRNs have been constructed and studied based on it. However, some recent studies have shown that some GRNs exhibit the property of memory, particularly vertebrate GRNs, and metazoan cell networks ([20], [21], [22], [23]). Thus, we should also investigate how computational models that have a capacity for memory would perform on smaller motifs, such as the toggle switch and the toggle triad, and also on larger GRNs. Two state-of-the-art deep learning models capable of this are the LSTM [24] and the Transformer [25]. LSTMs (Long Short-Term Memory) are a variant of vanilla Recurrent Neural Networks (RNNs) and as the name suggests, it has the capacity to retain both a short-term memory and a long-term memory, learn how much to retain it and affect future time-steps using it. LSTMs can thus be used to learn both local and global effects a gene expression pattern can have. Unlike vanilla RNNs, LSTMs do not suffer from the vanishing gradients problem over time (although it does suffer from

exploding gradients like vanilla RNNs). Transformers, on the other hand, learn local and global patterns using attention and self-attention mechanisms, allowing it to learn complex patterns in data. Using LSTMs and Transformers, one can determine whether the gene-expression time-series datasets exhibit memory while also deducing the underlying topology.

In the models explored in this thesis, all datasets were treated separately, i.e., each model was trained on only one dataset at a time to deduce the topology corresponding to it. However, we could use the plethora of time-series data to train a deep-learning model that learns the discrete conditional probability distributions $P(M_{ji}|\boldsymbol{X})$, i.e., given time-series data $\boldsymbol{X}$, what is the probability of the j-th gene interacting with the i-th gene in an activatory manner ($M_{ji} = 1$), inhibitory manner ($M_{ji} = -1$), or not interacting at all ($M_{ji} = 0$). Such a model could thus leverage the large amounts of training data one can generate through RACIPE simulations. However, a big assumption we make when using such a model is that all patterns of interactions learned from simulated data from small topologies would carry forward to larger, novel, and unexplored topologies as well.

Another potential avenue includes building a classifier that learns from simulated data from all possible topologies that can be constructed from a given number of genes and then classify new, real data accordingly. However, besides being very computationally expensive, it would be difficult to formulate the time-series data in a way appropriate for training a classifier. An interesting technique might be to first train a variational auto-encoder (VAE) [26] to learn the latent probability distribution of data from a particular topology and then build a classifier upon the latent parameters. In such a framework, the time-series data can simply be treated as images of gene expression values in time.

5.3 Applications to Biology and Physics

With an ever-growing amount of gene expression data being collected, it has become more important than ever to have reliable techniques that can reconstruct large GRNs from this data. The methods presented in this thesis are more suitable than traditional methods like Bayesian models, Monte Carlo methods, and exhaustive search because they are much less computationally intensive and scale well with an increasing number of genes. These methods also allow inference of sparse networks – after the networks are trained once, the edge with the least strength is removed, the networks are trained once again, and the process is repeated until we have the most sparse GRN that can fit the data well. Machine learning methods have been quite successful in recovering small well-known GRNs from simulated time-series data, such as the toggle switch and the toggle triad (as shown in this thesis), the French flag circuit, pulse detector, oscillator, and the biological counter (shown by Hiscock [15]), adaptation, controlled oscillation,

and gap gene patterning, as well as large GRNs, such as 10-cell continuous-state cellular automata (shown by Shen et al. [17]). Accurate GRNs have been constructed from real time-series data as well, such as the 12-gene network known to affect the decision between erythrocyte and neutrophil cell fates [13].

In general, being able to robustly deduce plausible GRNs from the time-series data holds a lot of practical significance. An accurate mapping of gene interactions would allow us to control cellular reprogramming and disease progression. A better understanding of these interactions can also help us design synthetic circuits that can achieve specific objectives, such as performing "differentiation therapy" for cancers and making pancreatic cells produce insulin [27]. Hence, it is important to have methodologies that can efficiently and reliably reconstruct GRNs for us.

Even though the methods presented in this thesis have focused on deducing Gene Regulatory Networks, a closer look at the formulation of the problem would reveal that we are essentially trying to use deep learning to solve the inverse problem given a set of coupled ODEs and time-series data. Hence, these methods can be used to solve similar problems in other fields as well. In physics, many times, we cannot derive the differential equations for our system from first principles simply because the system may be in a driven non-equilibrium state, be highly nonlinear, or the dynamics may be occurring at multiple, not well-separated scales [12]. Hence, we may need to use experimental data to come up with phenomenological descriptions of such systems. Using the ML techniques discussed in this thesis can help in such cases. A further addition to these methods could be directly incorporating the physics of the system into the learning process, giving us Physics-Informed Neural Networks (PINNs) that use Physics-Based Deep Learning (PBDL) to learn models that can accurately represent the underlying dynamics shown by the (training) data. PBDL usually incorporates the physics of the system (usually the system of ODEs/PDEs) directly into the loss as a "residual" term which is then minimized during training which either learns the inverse problem or the generative task [28]. These methods can be advantageous over standard supervised learning as the system tries to perform the minimization process while keeping the physics of the system in consideration and can learn one of the several possible modes of the system (depending on the weights initialization) instead of learning the average behavior over all modes.

5.4 Conclusion

With large amounts of gene-expression data being generated every day, it has become important, now more than ever, to have systems in place that can identify the underlying gene interactions and construct Gene Regulatory Networks (GRNs) from them robustly, regardless of the amount of data and the number of genes in consideration. Exhaus-

tively searching which GRN could have produced the gene-expression data or using other traditional methods (Bayesian models, Monte Carlo methods) becomes quite unfeasible once we start dealing with a large number of genes. In this thesis, we have proposed three machine-learning models constructed from deep neural networks with different architectures that have shown the following three important features - quick GRN inference due to fast training achieved by Adam optimization, robustness in predicting the correct topology despite varying forms of gene-expression activity, ease of scalability (no change in code required and negligible increase in training time as the number of genes increases) [Code can be found at https://github.com/raj1401/Deducing-GRNs-Using-ML].

All three models, A, B, and C, performed really well on the two-gene system toggle switch in correctly predicting its topology from the data. However, the performance decreased on data from the three-gene system, toggle triad. This was no surprise, as moving from a two-gene system to a three-gene system increases the possible number of topologies from 4 to 64. Keeping this fact in mind, the fact that all three models still predicted the correct topology much more frequently than others (more so in the case of models A and C) shows us that we are going in the right direction. This is further reinforced by our benchmarking results that showed that the performance of model A continually increases with the number of hidden layers (we expect model B to show a similar trend). Hence, it's almost certain that with many more hidden layers, models A and B will have exceptional performance and be even more robust and "confident" when predicting that the given gene-expression data comes from a certain GRN. Out of all three models, model C gave the best results, predicting the toggle switch topology almost with an average probability of 99% (in both monostable and bistable cases) and predicting the toggle triad topology with an average probability of 48%, and the only other topologies it predicted had just a single link impurity (both monostable and bistable cases). Hence, we believe that it should perform well on data from other topologies as well.

Apart from these three models, we also highlighted how other deep-learning frameworks, such as LSTMs, Transformers, and VAEs, could be utilized to perform GRN inference. Incorporating these techniques have shown great promise in performing GRN inference tasks, and it's only a matter of time before we have frameworks that accurately determine the possible GRNs the genes of interest form from their time-series expression data. Such frameworks should considerably reduce the time biologists have to spend coming up with plausible GRNs from previous knowledge, simulating them, and testing whether their dynamics correspond to the given gene-expression data. Combined with techniques that can sparsen the GRNs predicted by these frameworks, it should provide them with the correct or almost correct GRNs in almost no time, simplifying their task to just tweaking the inferred GRNs until they give satisfactory performance and hold biological relevance.

Appendix A

Runge's Phenomenon

To mathematically reason about Runge's phenomenon, we must first derive the interpolation error expression.

Consider a function f in $C^{n+1}[a,b]$ and a polynomial p_n of degree $\leq n$ that interpolates f at $n+1$ points $x_0, x_1, \cdots, x_n$. Consider an $x \in [a,b]$. If $x = x_i$ for $i = 0, 1, \cdots, n$, then $p_n(x) = f(x)$, so we must consider an $x \neq x_i$ to derive the error expression.

Let us define two functions:

$$w(t) = \prod_{i=0}^{n}(t - x_i)$$

$$\phi_x(t) = f(t) - p_n(t) - \frac{f(x) - p_n(x)}{w(x)}w(t)$$

$\phi_x(t) \in C^{n+1}[a,b]$ and it vanishes at $(n+2)$ points, at x, and at $t = x_i$. From Rolle's theorem, since $\phi_x(t)$ is differentiable, and has $(n+2)$ zeroes, then $\phi'_x(t)$ must have atleast $(n+1)$ zeroes, $\phi''_x(t)$ must have atleast n zeroes, and continuing like this, $\phi_x^{n+1}(t)$ must have atleast one distinct zero in $[a,b]$. Let's call it ξ_x.

Now,

$$\begin{aligned}
\phi_x^{n+1}(t) &= \frac{d^{(n+1)}}{dt^{(n+1)}}\left[f(t) - p_n(t) - \frac{f(x) - p_n(x)}{w(x)}w(t)\right] \\
&= f^{(n+1)}(t) - p_n^{(n+1)}(t) - \frac{f(x) - p_n(x)}{w(x)}\frac{d^{(n+1)}}{dt^{(n+1)}}w(t) \\
&= f^{(n+1)}(t) - p_n^{(n+1)}(t) - \frac{f(x) - p_n(x)}{w(x)}(n+1)!
\end{aligned}$$

Hence, at $t = \xi_x$,

$$\phi_x^{n+1}(t) = 0 \implies f^{(n+1)}(\xi_x) - p_n^{(n+1)}(\xi_x) = \frac{f(x) - p_n(x)}{w(x)}(n+1)!$$

However, since p_n is a polynomial of degree n, $p_n^{(n+1)}(t) = 0 \; \forall \; t$. Thus, we get

$$f(x) - p_n(x) = \frac{1}{(n+1)!} f^{(n+1)}(\xi_x) w(x)$$

Or,

$$f(x) - p_n(x) = \frac{1}{(n+1)!} f^{(n+1)}(\xi_x) \prod_{i=0}^{n} (x - x_i)$$

This is the polynomial interpolation error formula. Now,

$$\max |f(x) - p_n(x)| \leq \frac{\max |f^{(n+1)}(\xi_x)|}{(n+1)!} \max \left| \prod_{i=0}^{n} (x - x_i) \right|$$

Now, if $\max |f^{(n+1)}(\xi_x)| < M$ for some $M \in \mathbb{R} \; \forall \; n$, then

$$\lim_{n \to \infty} \frac{\max |f^{(n+1)}(\xi_x)|}{(n+1)!} = 0$$

Else, if $\max |f^{(n+1)}(\xi_x)|$ grows faster than $(n+1)!$, then

$$\lim_{n \to \infty} \frac{\max |f^{(n+1)}(\xi_x)|}{(n+1)!} \to \infty$$

And we may observe Runge's phenomenon of $f(x) - p_n(x)$ blowing up (as the upper bound going to infinity does not necessarily imply that the error diverges as well).

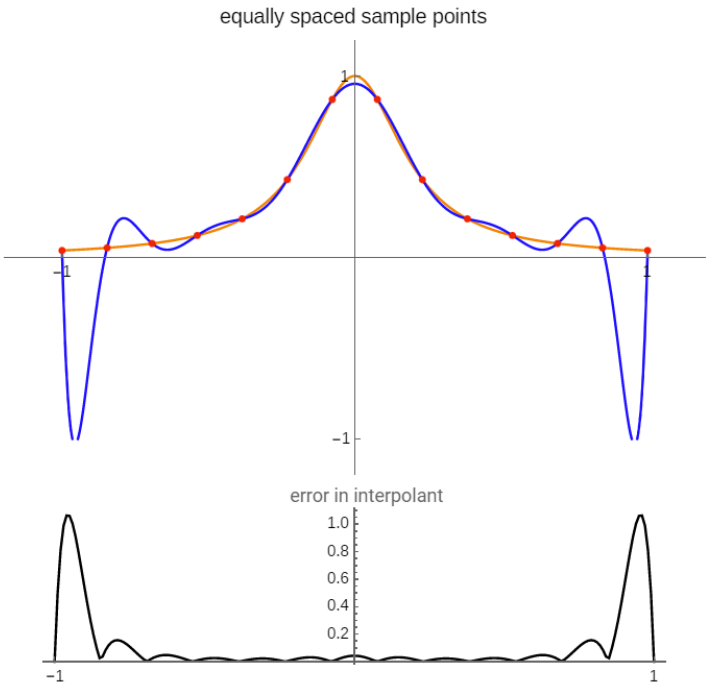


Figure A.1: Runge's Phenomenon. Ref [29]

Appendix B

Adam Optimization

According to the Adam update rule,

$$\boldsymbol{\theta}_n = \boldsymbol{\theta}_{n-1} - \alpha \cdot \frac{\hat{\mathbf{m}}_n}{\sqrt{\hat{\mathbf{v}}_n} + \epsilon}$$

giving us an effective step-size of $\alpha \cdot \frac{\hat{\mathbf{m}}_n}{\sqrt{\hat{\mathbf{v}}_n} + \epsilon}$. Since ϵ is usually chosen to be a very small number, the step size at time t becomes

$$\Delta_t = \alpha \cdot \frac{\hat{\mathbf{m}}_n}{\sqrt{\hat{\mathbf{v}}_n}}$$

Now, since $\hat{\mathbf{m}}_n = \frac{\mathbf{m}_n}{1-\beta_1^n}$ and $\hat{\mathbf{v}}_n = \frac{\mathbf{v}_n}{1-\beta_2^n}$, Δ_t has two upper-bounds:

$$|\Delta_t| \leq \alpha \cdot \frac{(1-\beta_1)}{\sqrt{(1-\beta_2}} \quad \text{if} \quad (1-\beta_1) > \sqrt{(1-\beta_2)}$$

$$|\Delta_t| \leq \alpha \quad \text{otherwise}$$

The first case only occurs in the most severe case of sparsity – the gradient has been zero at all timesteps except the current timestep [4]. When the sparsity is less ($1-\beta_1 = \sqrt{1-\beta_2}$), $|\hat{m}_t/\sqrt{\hat{v}_t}| < 1 \implies |\Delta_t| < \alpha$. In most cases, $\hat{m}_t/\sqrt{\hat{v}_t} \approx \pm 1$, since $|\mathbb{E}[g]/\sqrt{\mathbb{E}[g^2]} \leq 1$, where $\mathbb{E}$ denotes the expectation value. Hence, α approximately bounds the magnitude of the step size taken in the parameter space.

Appendix C

Auto-adjusting Hill coefficient during training

If we constrained the Hill coefficient n to be between 1 and some positive integer N, then exhaustively searching for optimum values of n for G number of genes would mean trying out all possible combinations of n followed by training the neural networks and seeing which minimizes the mean squared error the most. Besides being heavily dependent on the quality of initial values of the weights and biases (or equivalently, how close we reach to the global optima), this method is highly computationally expensive. For G genes, we would have G neural networks, each with a maximum of G different Hill coefficients. Thus, we will have N^{G^2} different possible combinations of Hill coefficients, hence this process is exponential in the square of number of genes ($\mathcal{O}(N^{G^2})$).

Using a greedy approach instead of an exhaustive one, we can proceed as follows. First, initialize all Hill coefficients to 1 and evaluate the mean squared error after training. Then, keep increasing the first Hill coefficient of the first neural network until no performance benefit is produced. This will be the optimum value of the first Hill coefficient. Repeat the process for the second Hill coefficient, which will give its optimum value. This process is repeated until all Hill coefficient values have been obtained. In the worst case, we will have to go through all N Hill coefficients for all G^2 Hill coefficients, giving us $N \cdot G^2$ passes. Hence this process is quadratic in the number of genes ($\mathcal{O}(G^2)$).

Both these methods are quite inefficient, and using them would become unfeasible when the number of genes is very high. Hence, I came up with a scheme using which the Hill coefficient can self-adjust during training itself. First, for the time being, assume that the Hill coefficient, n is continuous, not discrete. We can write the shifted-Hill function as follows:

$$H(n, A, \lambda, x) = \frac{A^n}{A^n + x^n} + \lambda \frac{x^n}{A^n + x^n}$$

Taking its partial derivative with respect to n gives

$$\frac{\partial H}{\partial n} = \frac{(A^n + x^n)(A^n \log A + \lambda x^n \log x) - (A^n + \lambda x^n)(A^n \log A + x^n \log x)}{(A^n + x^n)^2} \tag{C.1}$$

Now, during training, we will be provided with a batch of training data $\boldsymbol{X}_{batch}$ to optimize on. For the ij-th Hill layer, we can compute the derivative in equation C.1 for all values of x between 0 and 1 with some resolution (say 0.01) and find its *minimum* and *maximum* values. We then compute the average value of this derivative over all values of $x_{j,\ batch}$. Now, we use a threshold value $\delta \in [0, 1)$ and change the Hill coefficient as follows:

$$n = \begin{cases} n - 1 & \text{if } n > 1 \text{ and } average > mid + \delta \cdot max\ distance \\ n + 1 & \text{if } average < mid - \delta \cdot min\ distance \\ = n & \text{otherwise} \end{cases} \tag{C.2}$$

where $mid = (minimum + maximum)/2$, $max\ distance = maximum - mid$, and $min\ distance = mid - minimum$. Hence, the Hill coefficient is changed only when the average gradient value (over the batch examples) is close to the minimum and maximum possible values. This process is similar to gradient descent where the parameters are changed in the opposite direction of the gradient.

The threshold value of δ was chosen to be 0.5, and all Hill coefficients were initialized to 2. This method was used on the toggle triad datasets, which contributed to why none of the datasets faced the exploding gradients problem.

Bibliography

[1] Aurelien Geron. *Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow*. O'Reilly Media, Inc., 2019. ISBN: 9781492032649.

[2] IBM. *What are neural networks?* URL: https://www.ibm.com/topics/neural-networks.

[3] IBM. *What are Recurrent Neural Networks?* URL: https://www.ibm.com/topics/recurrent-neural-networks.

[4] Jimmy Lei Ba Diederik P. Kingma. "Adam: A Method For Stochastic Optimization". In: (2015).

[5] Frank Emmert-Streib, Matthias Dehmer, and Benjamin Haibe-Kains. "Gene regulatory networks and their applications: understanding biological and medical problems in terms of networks". In: *Frontiers in Cell and Developmental Biology* 2 (2014). ISSN: 2296-634X. DOI: 10.3389/fcell.2014.00038. URL: https://www.frontiersin.org/articles/10.3389/fcell.2014.00038.

[6] Adam Arkin, John Ross, and Harley H McAdams. "Stochastic Kinetic Analysis of Developmental Pathway Bifurcation in Phage lambda-Infected Escherichia coli Cells". In: *Genetics* 149.4 (Aug. 1998), pp. 1633–1648. ISSN: 1943-2631. DOI: 10.1093/genetics/149.4.1633. eprint: https://academic.oup.com/genetics/article-pdf/149/4/1633/42008581/genetics1633.pdf. URL: https://doi.org/10.1093/genetics/149.4.1633.

[7] Verena Wolf et al. "Solving the chemical master equation using sliding windows". In: *BMC Systems Biology* 4 (Apr. 2010). ISSN: 1752-0509. DOI: 10.1186/1752-0509-4-42. URL: https://doi.org/10.1186/1752-0509-4-42.

[8] Atchuta Srinivas Duddu et al. "Multi-stability in cellular differentiation enabled by a network of three mutually repressing master regulators". In: *Journal of The Royal Society Interface* 17.170 (2020), p. 20200631. DOI: 10.1098/rsif.2020.0631. eprint: https://royalsocietypublishing.org/doi/pdf/10.1098/rsif.2020.0631. URL: https://royalsocietypublishing.org/doi/abs/10.1098/rsif.2020.0631.

[9] Joseph X. Zhou and Sui Huang. "Understanding gene circuits at cell-fate branch points for rational cell reprogramming". In: *Trends in Genetics* 27 (2011). DOI: doi:10.1016/j.tig.2010.11.002. eprint: https://doi.org/10.1016/j.tig.2010.11.002. URL: https://doi.org/10.1016/j.tig.2010.11.002.

[10] Nir Friedman. "Inferring Cellular Networks Using Probabilistic Graphical Models". In: *Science* 303.5659 (2004), pp. 799–805. DOI: 10.1126/science.1094068. eprint: https://www.science.org/doi/pdf/10.1126/science.1094068. URL: https://www.science.org/doi/abs/10.1126/science.1094068.

[11] Xiujun Zhang et al. "Inferring gene regulatory networks from gene expression data by path consistency algorithm based on conditional mutual information". In: *Bioinformatics* 28.1 (Nov. 2011), pp. 98–104. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btr626. eprint: https://academic.oup.com/bioinformatics/article-pdf/28/1/98/16903705/btr626.pdf. URL: https://doi.org/10.1093/bioinformatics/btr626.

[12] Yunfei Huang et al. "Sparse inference and active learning of stochastic differential equations from data". In: *Scientific Reports* 12.1 (Dec. 2022), pp. 98–104. ISSN: 2045-2322. DOI: 10.1038/s41598-022-25638-9. URL: https://doi.org/10.1038/s41598-022-25638-9.

[13] Joanna E. Handzlik and Manu. "Data-driven modeling predicts gene regulatory network dynamics during the differentiation of multipotential hematopoietic progenitors". In: *PLOS Computational Biology* 18.1 (Jan. 2022), pp. 1–31. DOI: 10.1371/journal.pcbi.1009779. URL: https://doi.org/10.1371/journal.pcbi.1009779.

[14] Gillian May et al. "Dynamic Analysis of Gene Expression and Genome-wide Transcription Factor Binding during Lineage Specification of Multipotent Progenitors". In: *Cell Stem Cell* 13.6 (Dec. 2013), pp. 754–768. ISSN: 1934-5909. DOI: 10.1016/j.stem.2013.09.003. URL: https://doi.org/10.1016/j.stem.2013.09.003.

[15] Tom W. Hiscock. "Adapting machine-learning algorithms to design gene circuits". In: *BMC Bioinformatics* 20.1 (Apr. 2019), p. 214. ISSN: 1471-2105. DOI: 10.1186/s12859-019-2788-3. URL: https://doi.org/10.1186/s12859-019-2788-3.

[16] C. H. Huck Yang et al. *Learning Functions in Large Networks requires Modularity and produces Multi-Agent Dynamics*. 2018. arXiv: 1807.03001 [cs.NE].

[17] Jingxiang Shen et al. "Finding gene network topologies for given biological function with recurrent neural network". In: *Nature Communications* 12.1 (May 2021), p. 3125. ISSN: 2041-1723. DOI: 10.1038/s41467-021-23420-5. URL: https://doi.org/10.1038/s41467-021-23420-5.

[18] Pramod R. Somvanshi and Kareenhalli V. Venkatesh. "Hill Equation". In: *Encyclopedia of Systems Biology*. Ed. by Werner Dubitzky et al. New York, NY: Springer New York, 2013, pp. 892–895. ISBN: 978-1-4419-9863-7. DOI: 10.1007/978-1-4419-9863-7_946. URL: https://doi.org/10.1007/978-1-4419-9863-7_946.

[19] Bin Huang, Dongya Jia, and Jingchen Feng. "RACIPE: a computational tool for modeling gene regulatory circuits using randomization." In: BMC Systems Biology, 2018. DOI: 10.1186/s12918-018-0594-6. URL: https://doi.org/10.1186/s12918-018-0594-6.

[20] Surama Biswas et al. "Gene regulatory networks exhibit several kinds of memory: quantification of memory in biological and random transcriptional networks". en. In: *iScience* 24.3 (Feb. 2021), p. 102131.

[21] Javier Macia, Blai Vidiella, and Ricard V Solé. "Synthetic associative learning in engineered multicellular consortia". en. In: *J R Soc Interface* 14.129 (Apr. 2017).

[22] RA Watson et al. "Artificial Life Conference XII". In: *Odense, Denmark* (2010), pp. 194–201.

[23] Wen Xiong and James E Ferrell Jr. "A positive-feedback-based bistable 'memory module' that governs a cell fate decision". en. In: *Nature* 426.6965 (Nov. 2003), pp. 460–465.

[24] Sepp Hochreiter and Jrgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. eprint: https://direct.mit.edu/neco/article-pdf/9/8/1735/813796/neco.1997.9.8.1735.pdf. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

[25] Ashish Vaswani et al. "Attention is All you Need". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[26] Lucas Pinheiro Cinelli et al. "Variational Autoencoder". In: *Variational Methods for Machine Learning with Applications to Deep Networks*. Cham: Springer International Publishing, 2021, pp. 111–149. ISBN: 978-3-030-70679-1. DOI: 10.1007/978-3-030-70679-1_5. URL: https://doi.org/10.1007/978-3-030-70679-1_5.

[27] Kishore Hari et al. "Landscape of epithelial–mesenchymal plasticity as an emergent property of coordinated teams in regulatory networks". In: *eLife* 11 (Oct. 2022). Ed. by Sandeep Krishna, Aleksandra M Walczak, and Jean Clairambault, e76535. ISSN: 2050-084X. DOI: 10.7554/eLife.76535. URL: https://doi.org/10.7554/eLife.76535.

[28] Nils Thuerey et al. *Physics-based Deep Learning*. WWW, 2021. URL: https://physicsbaseddeeplearning.org.

[29] Chris Maes. *Runge's Phenomenon*. URL: https://demonstrations.wolfram.com/RungesPhenomenon/.